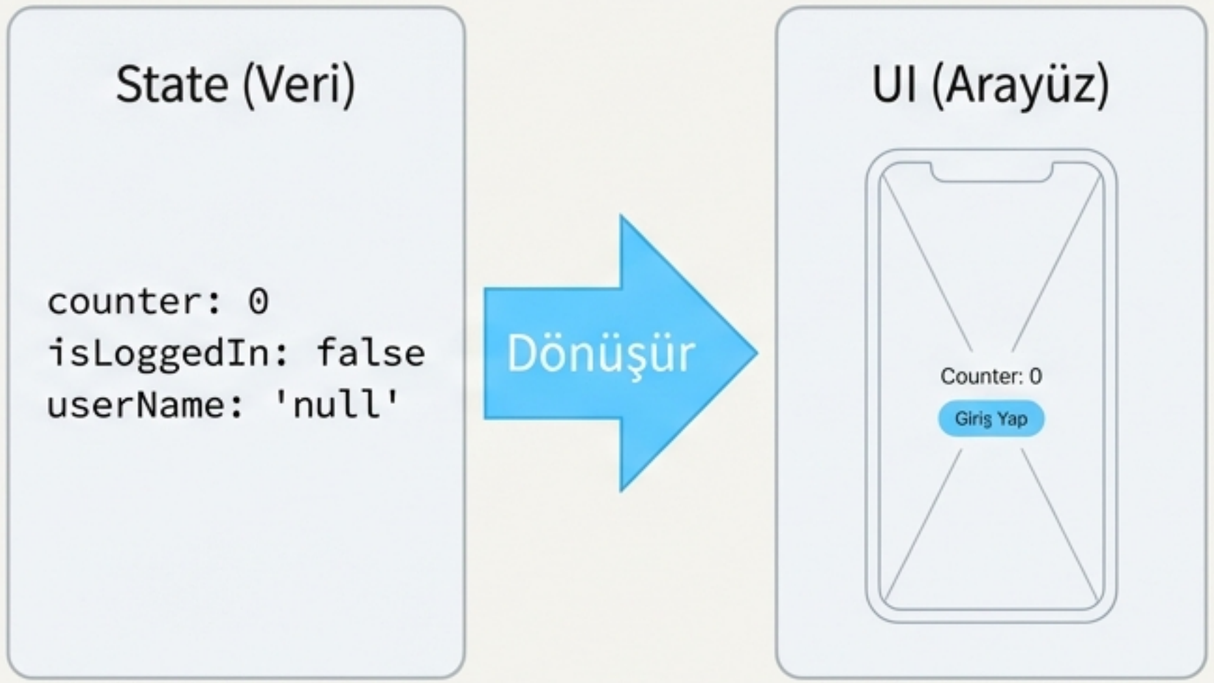
Her Flutter Geliştiricisinin Yolculuğu: State Management

`setState''in Derinliklerine Bir Bakış ve Sınırlarını Anlamak

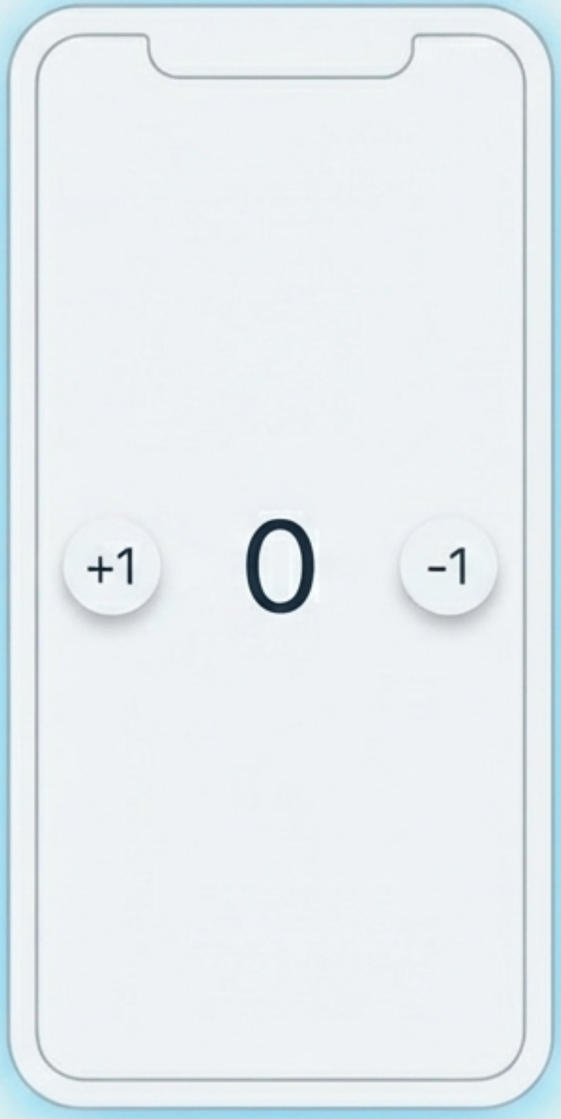
Maceranın Başlangıcı: 'State' Nedir?

- Flutter'da 'state', uygulamanızın herhangi bir anındaki verisidir. Bir düğmenin rengi, bir metin alanındaki yazı, bir kullanıcının oturum bilgisi, hepsi birer 'state'dir.
- Kullanıcı etkileşimi veya dış kaynaklardan gelen verilerle bu 'state' değişir.
- State management, bu değişiklikleri verimli ve öngörülebilir bir şekilde yönetme sanatıdır. İyi yapılandırılmış bir kod, okunması ve bakımı kolay bir uygulama demektir.



İlk Görev: Sayaç Uygulaması ve `setState`

- Her geliştiricinin karşılaştığı ilk ve en temel state management aracı `setState`'tir.
- Bu temel aracı, en basit görev Bu temel aracı, en basit görev olan bir sayaç uygulaması üzerinden inceleyeceğiz.
- Amacımız sadece nasıl çalıştığını değil, *neden* bu şekilde çalıştığını ve getirdiği sonuçları anlamak.



Mekanizmayı Anlamak: `StatefulWidget` ve `State` Sınıfı

`setState` kullanmak için iki sınıflı bir yapıya ihtiyacımız var. Biri widget ağacında yer alırken, diğeri onun durumunu yönetir ve yeniden çizimlerden etkilenmez.

```
// 1. StatefulWidget: Widget ağacında yer alan,
// değişmez (immutable) yapı.
class DemoPage extends StatefulWidget {
  const DemoPage(); // Değişmez olduğu için const olabilir.

  @override
  // 2. State Nesnesi Oluşturma: Bu metot, widget'ın
  // durumunu tutacak nesneyi oluşturur.
  _DemoPageState createState() => _DemoPageState();
}

// 3. State Sınıfı: Widget'ın asıl "durumunu" tutan
// ve yöneten özel (private) sınıf.
class _DemoPageState extends State<DemoPage> {
  int _counter = 0; // 4. Yönetilen "state" verisi.

  // ... metotlar ve build metodu burada yer alır ...
}
```

1. **StatefulWidget**: Widget ağacında yer alan, değişmez (immutable) yapı.
2. **State Nesnesi Oluşturma**: Bu metot, widget'ın durumunu tutacak nesneyi oluşturur.
3. **State Sınıfı**: Widget'ın asıl "durumunu" tutan ve yöneten özel (private) sınıf.
4. Yönetilen "state" verisi.

Büyüyü Başlatmak: `setState` Çağrısı

State'i değiştiren fonksiyonlar `setState`'i çağırır. Bu, Flutter'a 'Veri değişti, bu widget'ı ve altındakileri yeniden çiz' demenin yoludur.

```
void _increment() {
  // 1. setState'i bir callback fonksiyon ile çağır.
  setState(() {
    // 2. Bu blok içinde state'i güncelle.
    _counter++;
  });
  // 3. Callback bittiğinde, Flutter bu widget için
  //    build() metodunu yeniden tetikler.
}

void _decrement() {
  setState(() => _counter--);
}
```

1. setState'i bir callback fonksiyon ile çağır.

2. Bu blok içinde state'i güncelle.

3. Callback bittiğinde, Flutter bu widget için build() metodunu yeniden tetikler.

Sonuç: Arayüzün Yeniden İnşası (`build`)

`**setState**` çağrısından sonra, `build` metodu en güncel state verisiyle tekrar çalışır. `**Text**` widget'ı, `_counter`'ın yeni değerini alarak arayüzü günceller.

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceAround,
  children: [
    FlatButton(
      child: const Text("+1"),
      onPressed: _increment,
    ),
    // build metodu her çalıştığında, _counter'ın
    // güncel değeri burada kullanılır.
    Text("$_counter",
      style: const TextStyle(fontSize: 30),
    ),
    FlatButton(
      child: const Text("-1"),
      onPressed: _decrement,
    ),
  ],
)
```

build metodu her çalıştığında, _counter'ın güncel değeri burada kullanılır.

İlk Zafer: Çalışıyor! Ama Ne Pahasına?

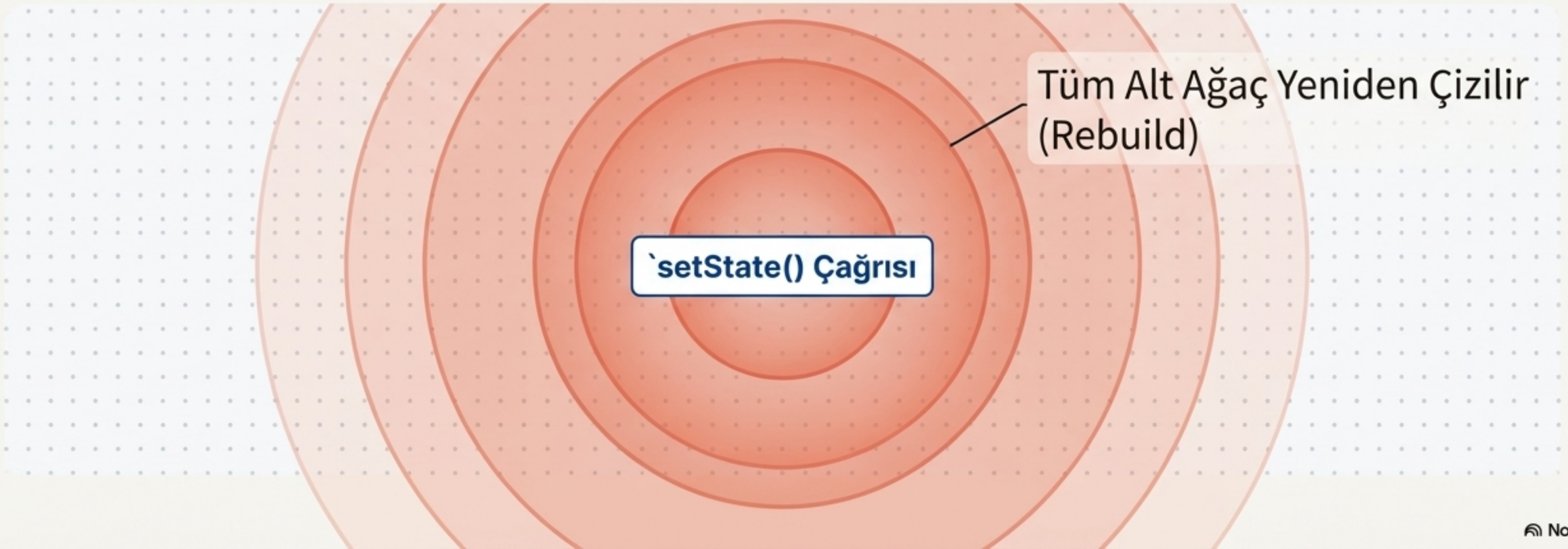
- Sayaç uygulamamız mükemmel çalışıyor. `setState` basit, anlaşılır ve işini yapıyor.

- Peki, bu basitlik büyük ve karmaşık uygulamalarda bir avantaja mı yoksa bir tuzağa mı dönüşüyor?

- Şimdi `setState''in yüzeyin altındaki maliyetlerini inceleyelim.

Görünmeyen Tehlike: Kontrolsüz Yeniden Çizimler

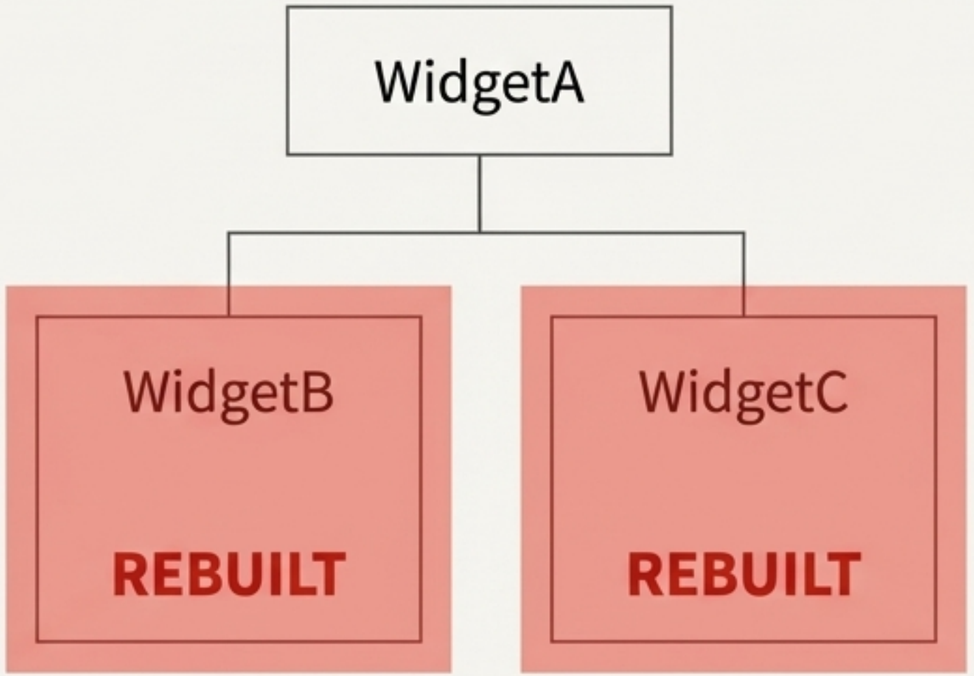
`setState`'in en büyük problemi performansı değil, ***verimsizliğidir***. `setState` çağrıldığında, sadece değişmesi gereken widget'ı değil, mevcut widget'ı ve **tüm altındaki widget ağacını yeniden çizer**.



Kodda Dalga Etkisi: Gereksiz `build` Çağrıları

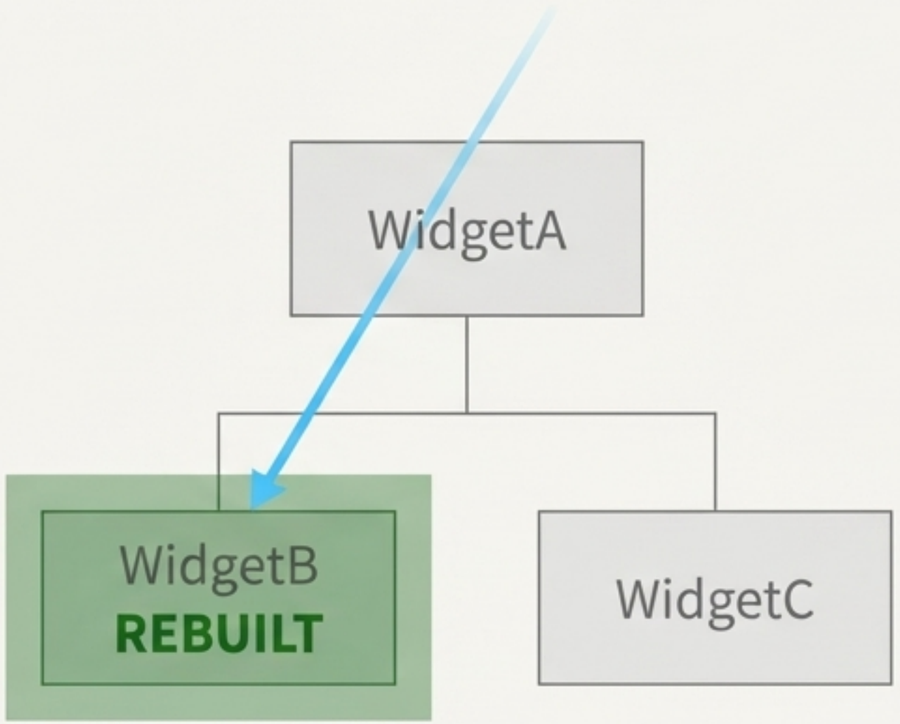
Bu örnekte, `RaisedButton`a tıklandığında `_WidgetAState` içinde `setState` çağrılır. `_value` sadece `WidgetB` tarafından kullanılsa bile, `WidgetC` de gereksiz yere yeniden çizilir.

```
// _WidgetAState
build(context) {
  return Column(
    children: [
      // Bu widget state'e bağlı.
      WidgetB("$_value"),
      // Bu widget'ın state ile ilgisi yok.
      const WidgetC(),
      RaisedButton(
        onPressed: () => setState(() {
          _value += 10;
        }),
      ),
    ]
  );
}
```



İdeal Senaryo: Cerrahi Hassasiyet

Verimli bir state management, tüm ağacı yeniden çizmek yerine, sadece state değişikliğinden etkilenen widget'ları hedefler. Bu, kaynak israfını önler ve karmaşık arayüzlerde performansı korur.



İkinci Kusur: Sorumlulukların Karışması

`_DemoPageState` sınıfı, Tek Sorumluluk Prensibi'ni (Single Responsibility Principle) ihlal eder. Tek bir sınıf, birden fazla görevi üstlenir:

- **UI Mantığı:** Arayüzü çizer (`build` metodu).
- **İş Mantığı:** `_counter` değerini yönetir.
- **State Yönetimi:** Değişiklikleri `setState` ile tetikler.

Bu durum, kodun test edilmesini, bakımını ve ölçeklenmesini zorlaştırır.

Hüküm: `setState`'in Karnesi

`setState`, Flutter'ı öğrenirken güçlü bir başlangıç noktasıdır, ancak profesyonel uygulamalar için bilinçli kullanılmalıdır.

Artıları (Pros)

✅ **Basit ve Anlaşılır**: Flutter'ın temelinde yer alır, öğrenmesi kolaydır.

✅ **Hızlı Prototipleme**: Küçük, izole widget'lar veya basit ekranlar için idealdir.

Eksileri (Cons)

⚠️ **Verimsiz Yeniden Çizimler**: Tüm alt ağacı yeniden oluşturur.

⚠️ **SRP İhlali**: UI, iş mantığı ve state yönetimini birbirine karıştırır.

⚠️ **Zayıf Ölçeklenebilirlik**: Uygulama büyüdükçe state'i yönetmek ve takip etmek zorlaşır.

Aslında Bir Çözüm Var: `InheritedWidget`in Karmaşıklığı

- Flutter, ‘cerrahi hassasiyet’ ile yeniden çizim yapmak için `InheritedWidget` adında bir mekanizma sunar.
- **Ancak:** Bu mekanizmayı doğrudan kullanmak aşırı derecede karmaşıktır ve çok fazla standart (boilerplate) kod gerektirir.
- Bu yüzden topluluk, bu karmaşıklığı soyutlayan daha basit çözümler geliştirmiştir.

Kaynak metinden alıntı: “You NEVER want to deal with it... there are many libraries (such as provider) that do all this tedious work for you!”

Yaşam Döngüsü Temelleri: `initState` ve `dispose`

`State` nesnesini tam olarak anlamak, yaşam döngüsü metotlarını bilmekten geçer.

`initState()`

- State nesnesi oluşturulduğunda **sadece bir kez** çalışır.
- Tek seferlik başlangıç ayarları (veri çekme, controller başlatma) için kullanılır.

```
@override
void initState() {
  super.initState();
  // Controller'ları veya abonelikleri başlat.
}
```

`dispose()`

- Widget ağaçtan kaldırıldığında **sadece bir kez** çalışır.
- Kaynakları temizlemek (controller'ları yok etmek, stream'leri kapatmak) için zorunludur.

```
@override
void dispose() {
  // Bellek sızıntılarını önlemek için temizlik yap.
  _myController.dispose();
  super.dispose();
}
```

Yolculuk Devam Ediyor: `setState`'in Ötesi

`setState'i anlamak, her Flutter geliştiricisi için temel bir adımdır. Onun sayesinde 'yeniden çizim' (rebuild) mekanizmasının nasıl çalıştığını öğreniriz.

Ancak büyük, ölçeklenebilir ve bakımı kolay uygulamalar inşa etmek için daha güçlü araçlara ihtiyacımız var. `setState'in ortaya çıkardığı sorunlar (verimsiz rebuild'ler, karışık sorumluluklar), `provider` ve `bloc` gibi çözümlerin neden var olduğunu mükemmel bir şekilde açıklar.

Maceranın bir sonraki bölümü, bu modern araçları keşfetmek olacak.