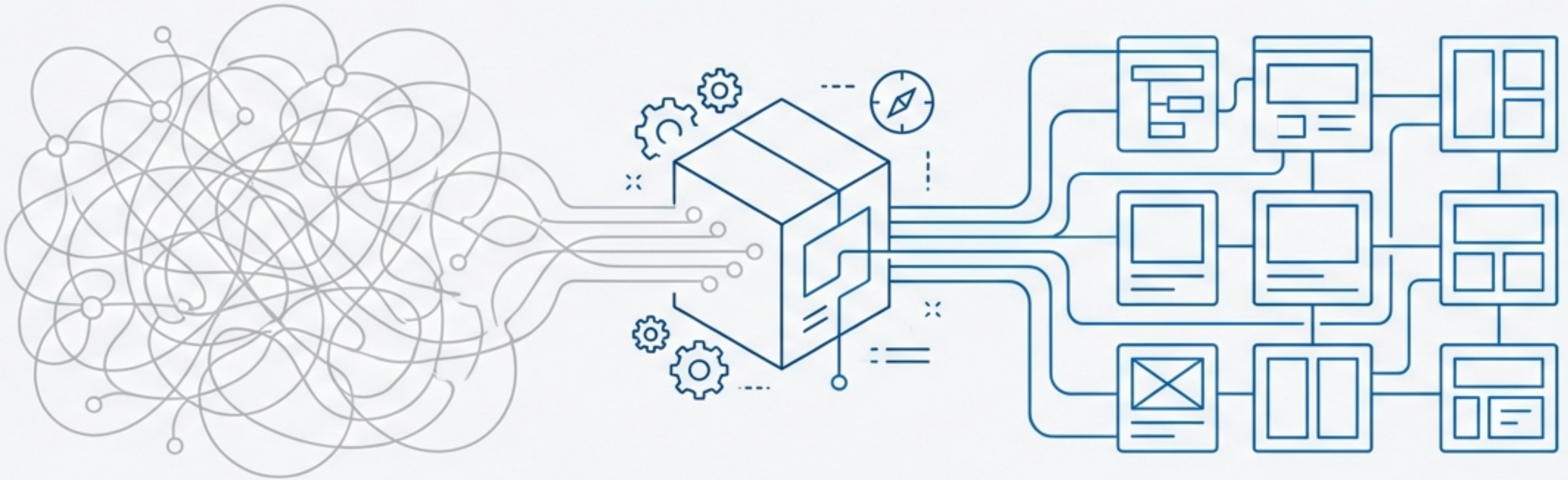
Flutter'da Usta İşi Arayüzler

Yaygın Tuzaklardan Profesyonel Çözümlere Giden Yol



Bu sunum, sadece çalışan değil, aynı zamanda sağlam, uyarlanabilir ve bakımı kolay, profesyonel düzeyde arayüzler oluşturmanın temel ilkelerini ele alacaktır.

Harika Bir Flutter Arayüzünün Dört Temel Taşı

1. Platform Stratejisi

Uygulamanızın farklı işletim sistemlerinde tutarlı ve yönetilebilir kalmasını sağlamak.

2. Duyarlı Tasarım

Arayüzünüzün her ekran boyutuna ve yönelimine kusursuzca uyum sağlaması.

3. Kısıtlama (Constraint) Yönetimi

“Unbounded height/width” gibi yaygın layout hatalarından kaçınmak.

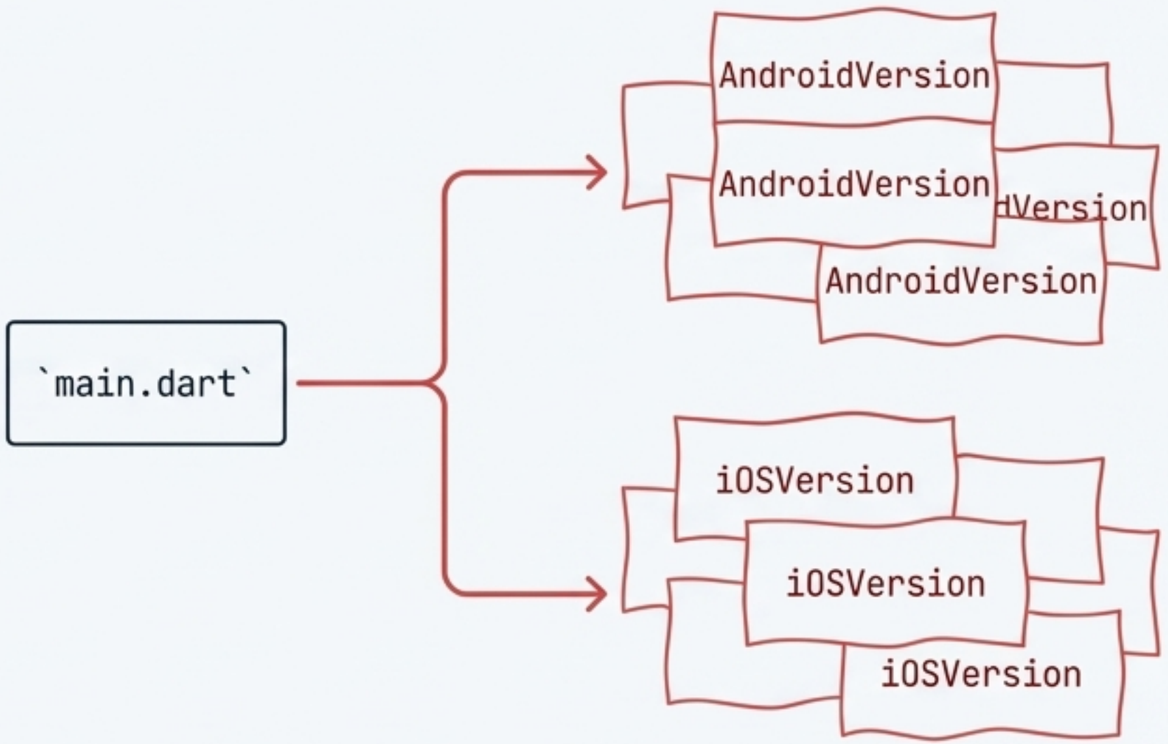
4. Merkezi Tema Yönetimi

Uygulama genelinde tutarlı ve kolayca güncellenebilir bir görsel kimlik oluşturmak.

Zorluk 1: Platform Kaosu ve Kod Tekrarı

“Uygulamam Android ve iOS için iki ayrı tasarıma mı sahip olmalı?”

Tek bir Flutter projesi içinde `if (Platform.isAndroid)` kullanarak iki farklı uygulama (`runApp(AndroidVersion())` vs `runApp(iOSVersion())`) çalıştırmanın getirdiği tehlikeler anlatılır. Bu yaklaşım, Flutter’ın tek kod tabanı avantajını ortadan kaldırır ve bakımı iki kat zorlaştırır.



Bu yöntem, iki ayrı native proje yönetmekle aynı anlama gelir ve Flutter’ın temel amacına aykırıdır.

Çözüm: Tek Kod Tabanı, Birleşik Marka Kimliği

YAPMAYIN

```
// YANLIŞ YAKLAŞIM
if (Platform.isAndroid) {
  runApp(const AndroidVersion());
} else {
  runApp(const iOSVersion());
}
```

YAPIN

Tek bir uygulama (`runApp(RestaurantApp())`) çalıştırın. Platforma özel küçük farkları (bir ikon, bir metin vb.) gerektiği yerde, widget'ın içinde ele alın.

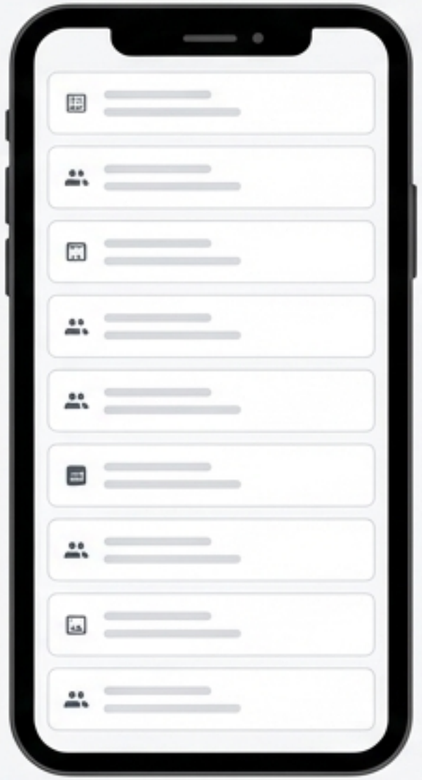
```
// DOĞRU YAKLAŞIM
class RestaurantApp extends StatelessWidget {
  String _logoName() {
    if (Platform.isIOS) { return "Welcome iOS user!"; }
    return "Welcome Android user!";
  }
  // ... build method ...
}
```

****Platformlar için değil, markanız için tasarlayın. Tek bir kod tabanının gücünü koruyun.****

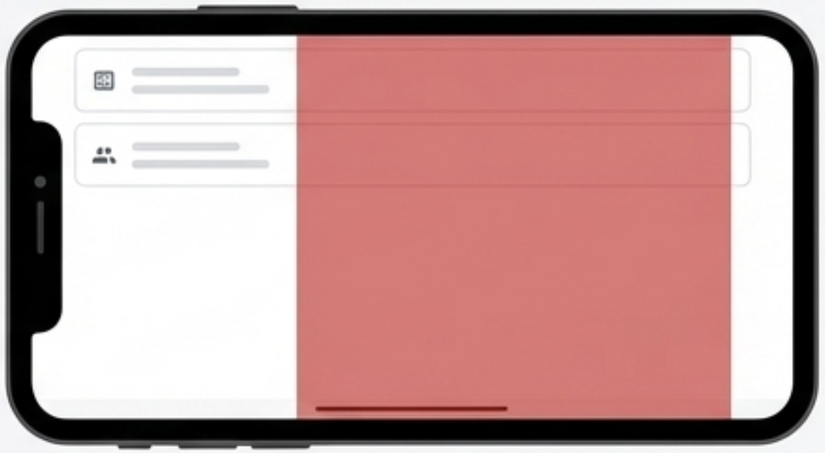
Zorluk 2: Farklı Ekranlarda Kırılan Katı Arayüzler

“Listem telefonda iyi görünüyor, ama tablete veya yatay moda geçince neden tüm boşluklar ziyan oluyor?”

Dikey modda düzgün çalışan basit bir `ListView'ın, ekran genişlediğinde (yatay mod veya tablet) nasıl verimsiz ve estetikten uzak hale geldiğini anlatır. Genişleyen alanda içerik yayılmaz, bu da kötü bir kullanıcı deneyimi yaratır.



DİKEY MOD

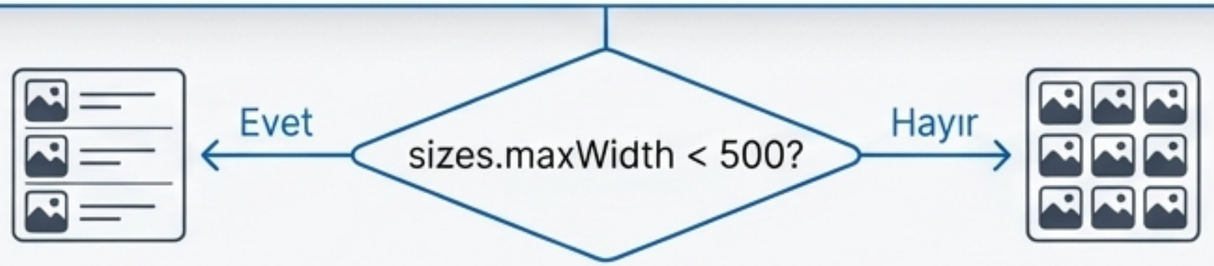


YATAY MOD

Çözüm: `LayoutBuilder` ile Alanı Akıllıca Kullanan Duyarlı Arayüzler

`LayoutBuilder`, tüm ekranın değil, **içinde bulunduğu ebeveyn widget'ın izin verdiği boyutları** size verir. Bu sayede, mevcut alana göre dinamik olarak farklı widget'lar (örneğin `ListView` veya `GridView`) gösterebilirsiniz.

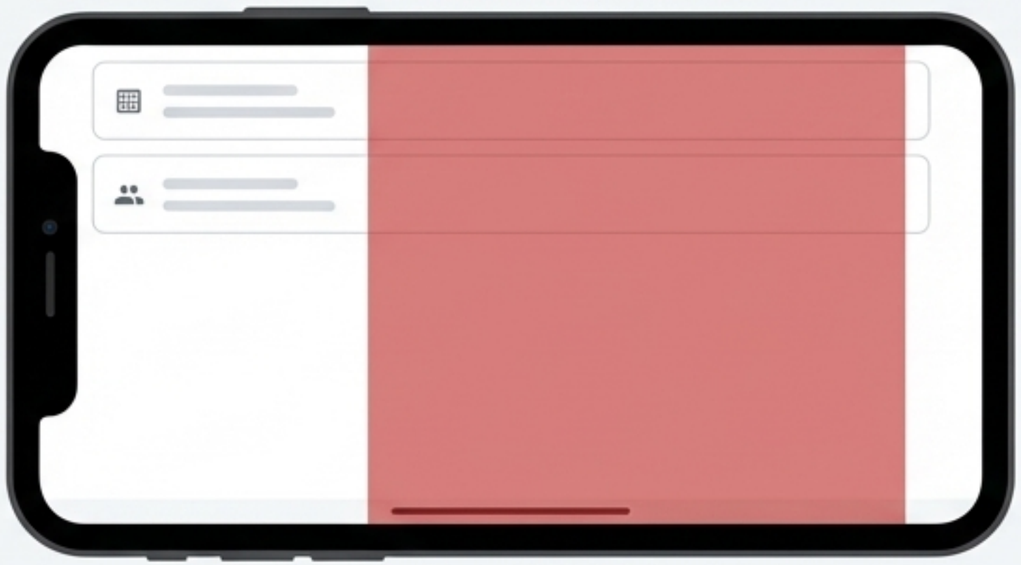
```
LayoutBuilder(
  builder: (BuildContext context, BoxConstraints sizes) {
    if (sizes.maxWidth < 500) {
      // Dar ekranlar için liste göster
      return const ListData();
    }
    // Geniş ekranlar için grid göster
    return const GridData();
  }
)
```



`LayoutBuilder`, arayüzünüzün mevcut alana akıllıca uyum sağlamasını sağlar.

`LayoutBuilder` Etkisi: Önce ve Sonra

ÖNCE 👎

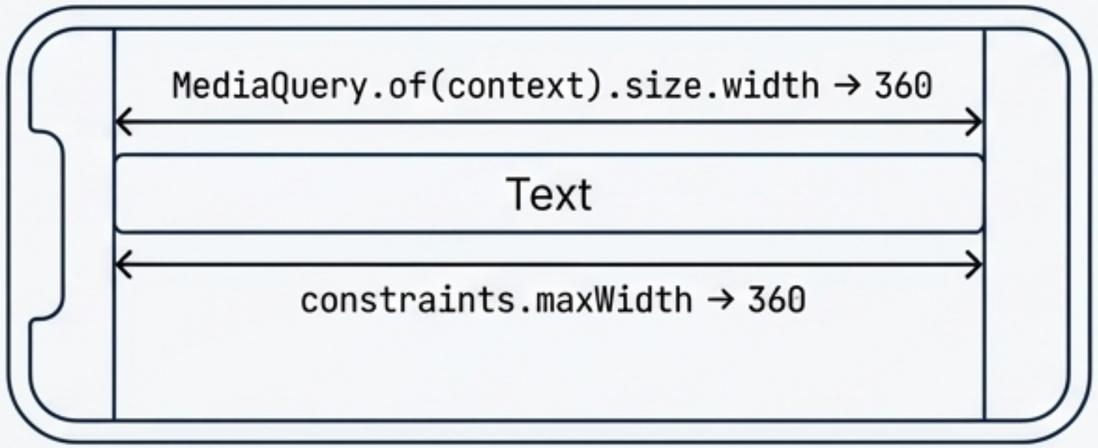


SONRA 👍

Profesyonel İpucu: `LayoutBuilder` mı, `MediaQuery` mi?

`MediaQuery`

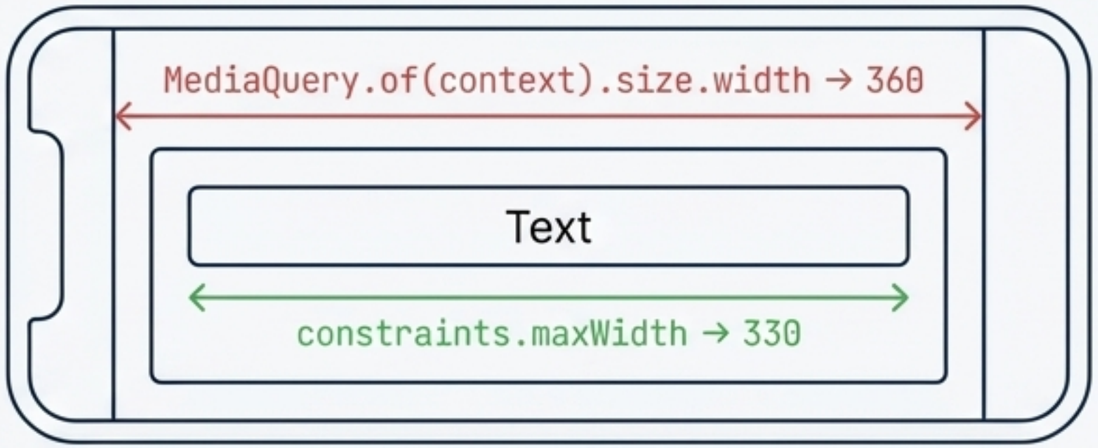
Cihazın **fiziksel ekranı** hakkında genel bilgi verir. Widget’ın kendi yerel kısıtlamalarını (örneğin bir `Padding` içindeyse) bilmez.



Kök widget'ta ikisi de aynıdır.

`LayoutBuilder`

İçinde bulunduğu **ebeveyn widget’ın sağladığı alanı** verir. **Duyarlı** yerleşimler için en doğru ve güvenilir araç budur.



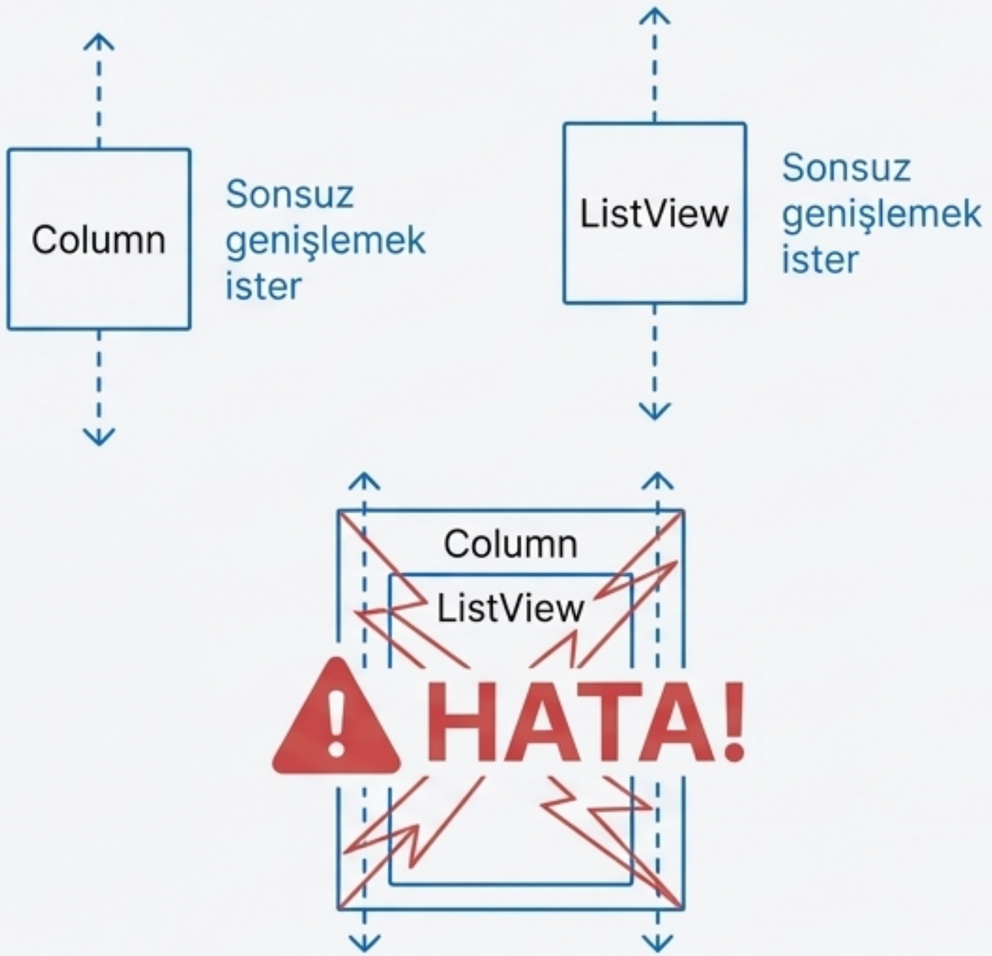
`LayoutBuilder` padding'i hesaba katar.

Yerleşim için ebeveynin sınırlarına güvenin (LayoutBuilder’), cihazın fiziksel ölçülerine değil (MediaQuery’).

Zorluk 3: “Unbounded Height” Hatası ve Çöken Arayüzler

“`Column` içine kaydırılabilir bir `ListView` koyduğumda uygulamam neden çöküyor?”

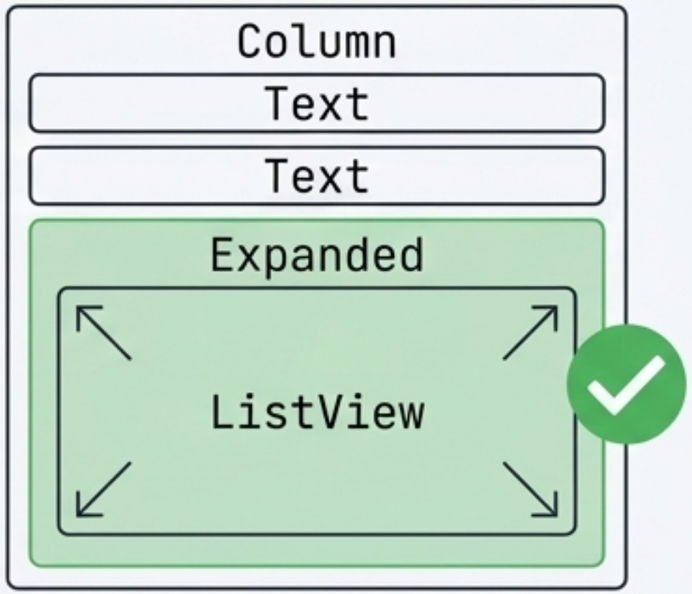
Temel çakışmayı açıklayın: `Column` dikeyde mevcut tüm alanı kaplayacak şekilde genişlemek ister. `ListView` da dikeyde sonsuz bir alana ihtiyaç duyar. Bu iki “sonsuza genişleme” eğilimi çakıştığında, Flutter bir boyut belirleyemez ve hata verir.



Çözüm 1: `Expanded` ile Kısıtlamaları Evcilleştirin

`Expanded`, altındaki widget'a (`ListView`) `Column` veya `Row` içinde **kalan tüm boş alanı kaplamasını** söyler. Bu, **`ListView`**'a sonsuz değil, sonlu ve belirli bir yükseklik verir, böylece çakışmayı çözer.

```
Column(
  children: [
    const Text("My name"),
    const Text("My surname"),
    Expanded( // ListView'ı sarar
      child: ListView(...),
    ),
  ]
);
```



Kaydırılabilir bir widget'ı `Column` veya `Row` içine yerleştirirken, onu `Expanded` ile sarmalayın.

Alternatif Çözümler ve Doğru Kullanım Senaryoları

`Expanded` En Yaygın Çözüm

Ne Yapar?: Kalan tüm boş alanı doldurur.

Ne Zaman Kullanılır?: Bir `Column/`Row` içinde esnek, doldurucu alan gerektiğinde.

```
Expanded(
  child: ListView(),
)
```

`shrinkWrap: true`

Ne Yapar?: Listenin yüksekliğini, içeriği kadar ayarlar.

Ne Zaman Kullanılır?: İçeriği az olan ve ekrana sığacağı bilinen kısa listeler için.

⚠ **Uyarı:** İçerik taşarsa hata verir.

```
ListView(
  shrinkWrap: true,
  ...
)
```

Sabit Yükseklik

Ne Yapar?: Listeye `height: 200` gibi sabit bir yükseklik verir.

Ne Zaman Kullanılır?: Listenin boyutları üzerinde tam ve kesin kontrol gerektiğinde.

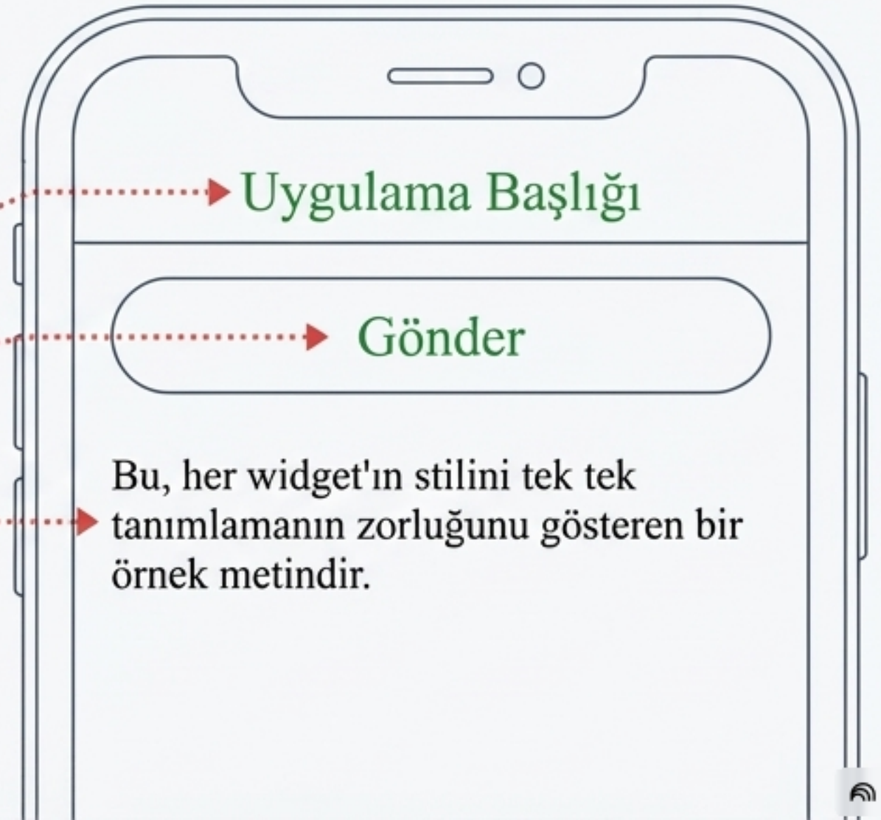
```
SizedBox(
  height: 200,
  child: ListView(),
)
```

Zorluk 4: Tutarsız ve “Hard-coded” Stiller

“Uygulamamın ana rengini veya yazı tipini değiştirmek neden tüm projeyi değiştirmemi gerektiriyor?”

Her widget'ı tek tek stillendirmenin getirdiği bakım kabusunu gösterin. Her `Text` widget'ına ayrı `TextStyle` atamak, küçük bir değişikliği bile saatler süren bir işe dönüştürür.

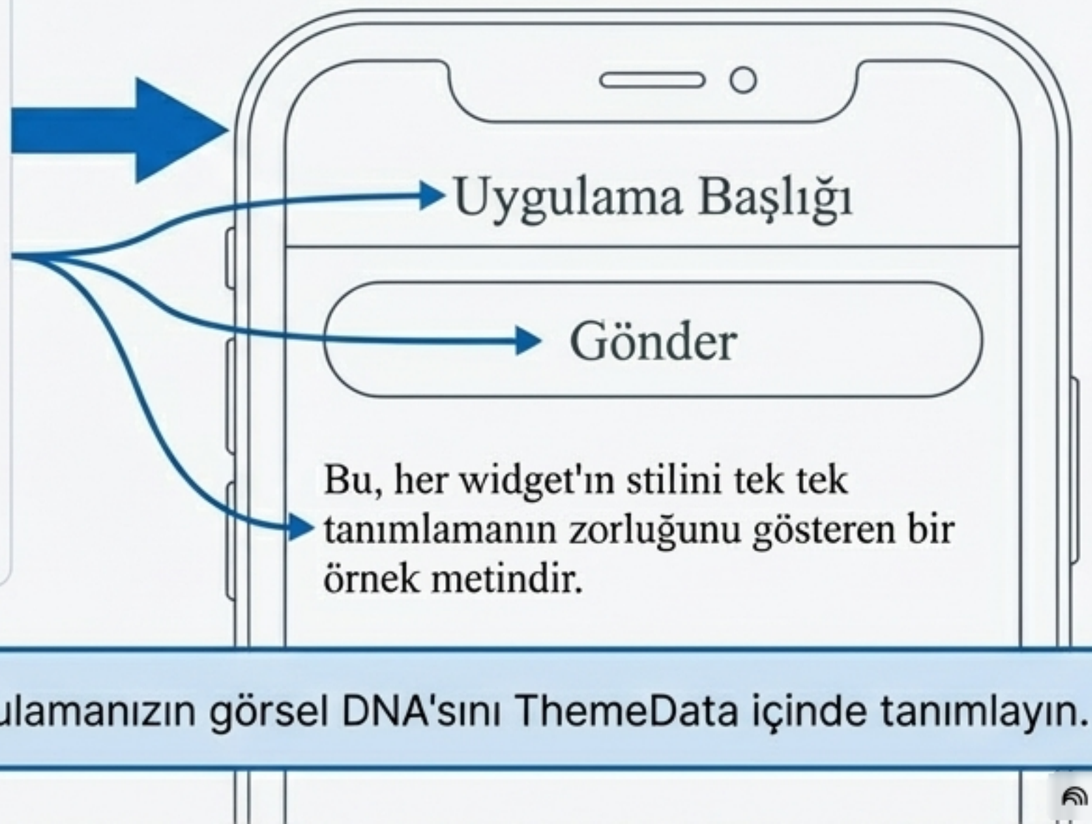
```
// Her widget için bunu tekrarlamak...
Text(
  "Başlık",
  style: TextStyle(
    fontFamily: "Times New Roman",
    fontSize: 18,
    color: Colors.green,
  ),
)
```



Çözüm: ThemeData ile Stili Merkezileştirin

MaterialApp widget’ının theme parametresini kullanarak renkleri, yazı tiplerini ve stilleri tek bir yerden tanımlayabilirsiniz. ThemeData'daki tek bir değişiklik anında tüm uygulamaya yansır.

```
MaterialApp(
  theme: ThemeData(
    primaryColor: Colors.green,
    accentColor: Colors.red,
    fontFamily: "Georgia",
    buttonColor: Colors.red,
  ),
  // ...
)
```



Golden Rule: Stilleri asla tek tek widget'lara yazmayın. Uygulamanızın görsel DNA'sını ThemeData içinde tanımlayın.

Temaları Genişletmek ve Yerel Olarak Ezmek (Override)

Mevcut Temayı Genişletme (`.copyWith()`)

`ThemeData.dark()` gibi önceden tanımlanmış bir temayı alıp, sadece istediğiniz birkaç özelliğini değiştirmek için kullanılır.

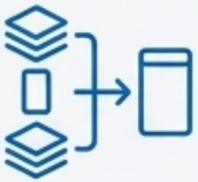
```
// Koyu temanın sadece ana rengini değiştir
theme: ThemeData.dark().copyWith(
  primaryColor: Colors.grey,
)
```

Bölgesel Tema Değişikliği (`Theme` Widget'ı)

Uygulamanın genel temasını, sadece belirli bir alt ağaç için geçersiz kılmak amacıyla kullanılır.

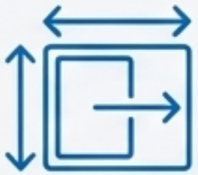

```
// Genel tema dark(), ama footer ve altındakiler light()
Theme(
  data: ThemeData.light(),
  child: const MyFooter(),
);
```

Ustalığa Giden Yol: Dört Altın Kural



TEK KOD TABANI
Platformlar için değil, markanız için birleşik bir tasarım oluşturun. `if (Platform)` kontrollerini `main` 'de değil, widget'ların içinde yapın.

DUYARLI OLUN
Arayüzünüzü `LayoutBuilder` ile ebeveyn widget'ın boyutlarına göre şekillendirin, `MediaQuery` 'nin ekran boyutlarına değil.



SINIRLARI BELİRLEYİN
`Column` / `Row` içindeki kaydırılabilir widget'ları `Expanded` ile sarmalayarak “unbounded height/width” hatalarını önleyin.

MERKEZİLEŞTİRİN
Uygulamanızın görsel kimliğini `ThemeData` ile tek bir yerden yöneterek tutarlılık ve kolay bakım sağlayın.

Artık sadece çalışan değil, aynı zamanda zarif, esnek ve profesyonel düzeyde bakımı kolay arayüzler oluşturmak için gerekli temel bilgilere sahipsiniz.