Flutter'ın Yapı Taşları

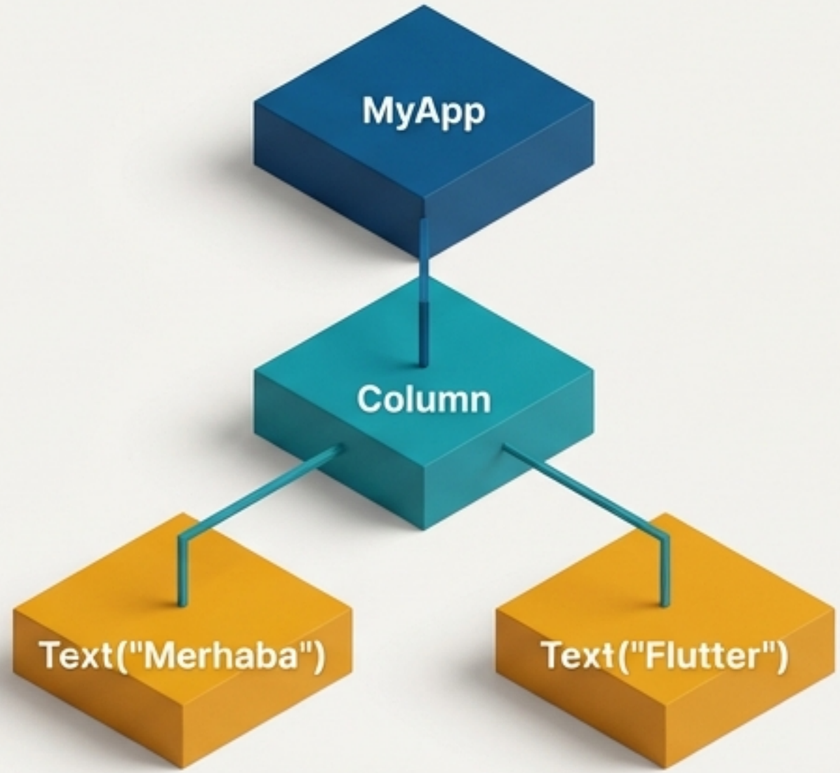
Widget'lar, Durum (State) ve Anahtarlar (Keys) ile Sağlam Arayüzler İnşa Etmek

Flutter'da bir kullanıcı arayüzü oluşturmak, dijital yapı taşlarını bir araya getirmeye benzer. Gördüğünüz, dokunduğunuz ve etkileşimde bulunduğunuz her şey bir 'Widget'tır. Bu sunumda, en temel taştan başlayarak karmaşık ve dinamik yapılar oluşturmayı adım adım keşfedeceğiz.

Her Şey Bir Widget'tır ve Bir Ağaç Oluşturur

Flutter'da widget'ları iç içe yerleştirerek 'Widget Ağacı' adı verilen bir hiyerarşi oluşturursunuz. `runApp()` metodu, bu ağacın kökünü belirler ve uygulamanızı başlatır.

```dart
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text("Merhaba"),
        Text("Flutter"),
      ]
    );
  }
}
```



Pro İpucu

`build(BuildContext context)` metodu, arayüzünüzü oluşturan yeni yaprakları (widget'ları) ağaca ekler. `context` parametresi, widget'ın ağaçtaki konumu hakkında Flutter'a kritik bilgiler verir.

İlk Yapı Taşlarımız: Metin ve Kapsayıcı

En temel görevlerle başlayalım: ekrana bir metin yazdırmak ve onu özelleştirilebilir bir kutlu içine almak.

`Text` Widget'ı

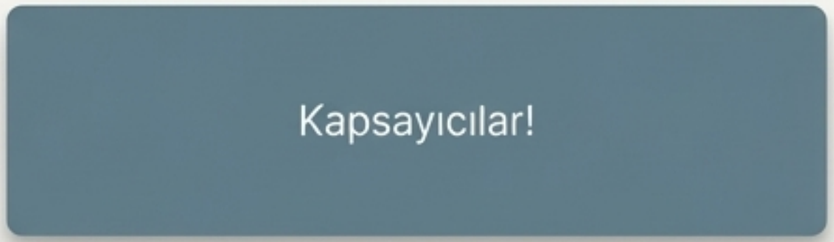
Ekranda bir metin parçası görüntülemek için kullanılır. `TextStyle` ile renk, font boyutu ve daha fazlası üzerinde tam kontrol sağlar.

```
const Text(
  "Ekranda bir metin",
  style: TextStyle(
    color: Colors.amber,
    fontSize: 16,
    wordSpacing: 3,
  ),
);
```

`Container` Widget'ı

HTML'deki `<div>` etiketine benzer. Boyutlandırma, konumlandırma, boyama ve süsleme için kullanılan çok amaçlı bir kapsayıcıdır.

```
Container(
  height: 80,
  width: 260,
  color: Colors.blueGrey,
  alignment: Alignment.center,
  child: const Text("Kapsayıcılar!", style: TextStyle(...)),
);
```



Blokları Düzenlemek: `Row` ve `Column`

Widget'ları yatay (yan yana) veya dikey (alt alta) dizmek için `Row` ve `Column` kullanılır. Bu iki widget, Flutter'daki çoğu arayüz düzeninin temelini oluşturur.

`Row` (Yatay Hizalama)

Çocuk widget'larını yatay bir eksende yan yana yerleştirir.

```
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: const [
    Text("Merhaba"),
    Text("Flutter!"),
    Text("!!"),
  ],
)
```

`Column` (Dikey Hizalama)

Çocuk widget'larını dikey bir eksende alt alta yerleştirir.

```
Column(
  mainAxisAlignment: MainAxisAlignment.center,
  children: const [
    Text("Merhaba"),
    Text("Flutter!"),
    Text("!!"),
  ],
)
```

Hizalamada Ustalaşmak: `mainAxisAlignment`

`mainAxisAlignment` özelliği, **`Row`** ve **`Column`** içindeki çocukların ana eksen (yatay/dikey) boyunca nasıl dağıtılacağını kontrol eder.

Bu hizalama seçenekleri hem `Row` hem de `Column` için aynı şekilde çalışır.

Taşma Sorunu ve Kaydırılabilir İçerik: `Column` vs. `ListView`

Ekranınıza sığmayan içerik, bir `Column` içinde "overflow" hatasına neden olur.
Çözüm: **Kaydırma davranışı ekleyen `ListView` kullanmaktır.**

Sorun (`Column`)

`Column` widget'ı varsayılan olarak kaydırma yapmaz. İçeriği ekrana sığmazsa, çalışma zamanında bir hata alırsınız.

Çözüm (`ListView`)

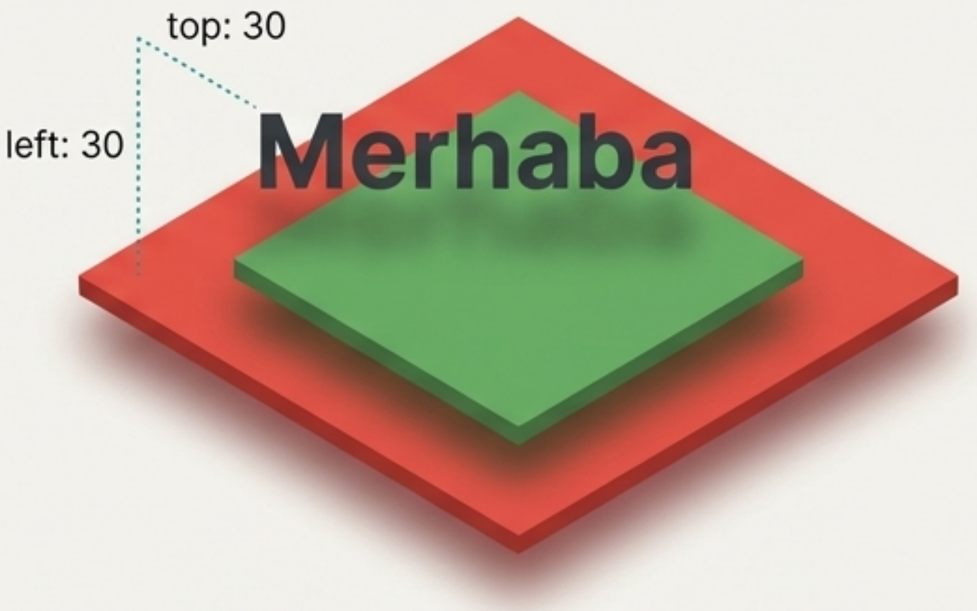
`ListView`, temelde kaydırma özelliğine sahip bir `Column"dur. Uzun veya dinamik listeler için idealdir.

```
// Uzun listeler için 'builder' yapıcı metodunu kullanmak
// performansı artırır.
ListView.builder(
  itemCount: myList.length,
  itemBuilder: (context, index) {
    return Text("${myList[index]}");
  },
)
```

Blokları Üst Üste Koymak: `Stack` ve `Positioned`

Widget'ları üst üste bindirmek ve ekranda serbestçe konumlandırmak için `Stack` kullanın. Çocukların sıralaması, hangisinin önde olacağını belirler (listedeki son eleman en öndedir).

```
Stack(
  children: [
    Container(width: 100, height: 100, color: Colors.red),
    Container(width: 80, height: 80, color: Colors.green),
    Positioned(
      left: 30,
      top: 30,
      child: Text("Merhaba"),
    ),
  ]
)
```



Kritik Bilgi

`Stack` içindeki widget’lar, listenin sonundakiler en üste gelecek şekilde çizilir. `Positioned` ile bir çocuğu `Stack`’in kenarlarına göre hizalayabilirsiniz.

Kritik Soru: Ya Yapımız Değişime Uğrarsa?

Şimdiye kadar ‘sabit’ veya ‘değişmez’ arayüzler inşa ettik. Peki ya bir kullanıcı etkileşimi, bir veri akışı veya bir ağ isteği sonucunda arayüzümüzün dinamik olarak güncellenmesi gerekirse?

Bu noktada Flutter’ın en temel iki widget türü devreye girer: **StatelessWidget** ve **statefulWidget**.

Değişmez Yapı Taşı: `StatelessWidget`

Bir `StatelessWidget`, bir **fotoğraf** gibidir. Çekildiği andaki durumu yansıtır ve sonradan değişmez.

Zamanla değişmeyecek arayüz parçaları oluşturmak için kullanılır. Bu widget'lar yalnızca kendi kurucu metodlarından (constructor) aldıkları `final` olarak işaretlenmiş verilere dayanır.


```dart
class PersonWidget extends StatelessWidget {
  final String name;
  final String age;

  const PersonWidget({
    required this.name,
    required this.age
  });

  @override
  Widget build(BuildContext context) {
    // ... build UI using name and age
  }
}
```

Temel Kural:
Eğer bir widget'ın tüm değişkenleri `final` olabiliyorsa ve `const` bir kurucu metoda sahipse, o bir `StatelessWidget`'tır.

Dinamik Yapı Taşı: `StatefulWidget`

Bir **`StatefulWidget`**, bir **dijital ekran** gibidir. İçeriği, kullanıcı eylemleri veya yeni verilerle her an değişebilir ve güncellenebilir.

Zamanla durumu değişebilecek dinamik arayüz parçaları oluşturmak için kullanılır. Bir butona tıklama, bir HTTP isteğinin yanıtı gibi olaylara tepki verir.

Kod Yapısı

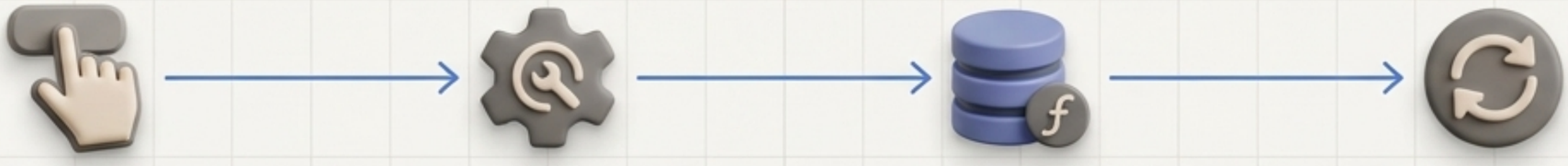
Bir **`StatefulWidget`** iki sınıftan oluşur: Widget'ın kendisi (`Counter`) ve onun yönetilebilir durumu (`_CounterState`).

```dart
// 1. Widget'ın kendisi (Ağaca eklenen kısım)
class Counter extends StatefulWidget {
  const Counter();

  @override
  _CounterState createState() => _CounterState();
}
```

```dart
// 2. Widget'ın durumu (Değişkenleri tutan ve güncellenen kısım)
class _CounterState extends State<Counter> {
  int _counter = 0;

  // ... build method and state logic
}
```

Bir `StatefulWidget` Nasıl Çalışır: `State` ve `setState()”

Değişikliğin sırrı, widget yeniden oluşturulduğunda bile 'hayatta kalan' `State` nesnesinde ve arayüzü yeniden çizmesi için Flutter'ı tetikleyen `setState()` metodundadır.



1. Kullanıcı Eylemi: Kullanıcı `IconButton`'a dokunur.

2. `setState()` Çağrısı: `onPressed` geri çağrısı, `setState()` metodunu tetikler.

3. Durum Güncellemesi: `setState()` içindeki kod çalışır (`_counter++`).

4. Yeniden İnşa (Rebuild): Flutter, bu widget'ın `build()` metodunu tekrar çağırarak arayüzü günceller.

```dart
class _CounterState extends State<Counter> {
  int _counter = 0; // Bu değişken yeniden inşalarda sıfırlanmaz.

  void _increment() {
    setState(() {
      // Bu çağrı Flutter'a durumun değiştiğini bildirir.
      _counter++;
    });
  }
  // ...
}
```

Pro İpucu

Durum değişkenlerinizi (`_counter` gibi) `build()` metodunun ***dışında*** tanımlayın. Aksi takdirde, her yeniden inşada ilk değerlerine sıfırlanırlar!

Doğru Yapı Taşını Seçmek: Ne Zaman Stateless, Ne Zaman Stateful?

İki widget türü arasında performans farkı yoktur. Seçim, kodun okunabilirliği ve widget'ın amacına bağlıdır. İşte basit bir kılavuz:

`StatelessWidget` Kullanın...

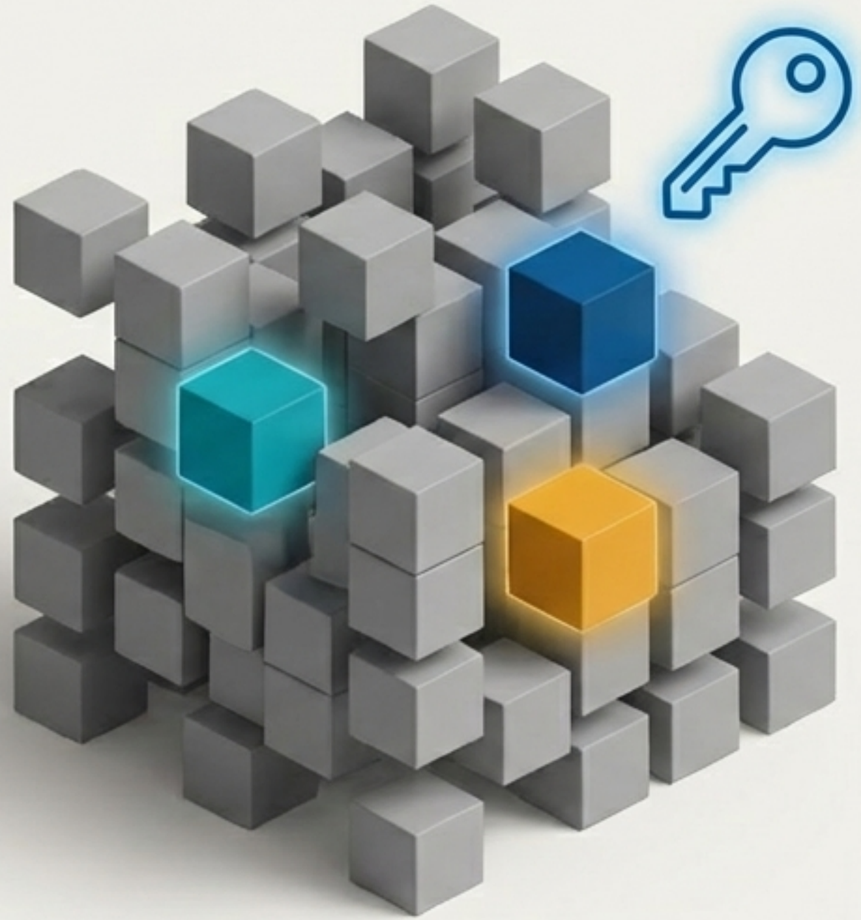
- Widget'ın tüm örnek değişkenleri `final` olarak işaretlenebiliyorsa.
- Widget'ın dışarıdan bir bağımlılığı yoksa ve içeriği tamamen statikse (örneğin bir başlık veya ikon).
- Yalnızca kurucu metodundan aldığı verileri görüntüleyen, 'tekrar kullanılabilir' basit bileşenler oluşturuyorsanız.

`StatefulWidget` Kullanın...

- Widget'ın yaşam döngüsü boyunca değişebilecek (`final` olmayan) değişkenleri varsa.
- Kullanıcı etkileşimine (tıklama, kaydırma) veya zamanla değişen verilere (animasyonlar, veri akışları) yanıt vermesi gerekiyorsa.

Blokları Tanımlamak: Flutter'da Anahtarlar (Keys)

Bir `Widget` ağacı büyüdüğünde ve dinamik olarak değiştiğinde, Flutter'ın belirli bir widget'ı benzersiz bir şekilde tanıması gerekebilir. Anahtarlar (`Key`), widget'lara atanan birincil anahtar (primary key) gibi davranır.



Ne Zaman Gerekir?

Çoğu durumda anahtarlara ihtiyacınız olmaz. Ancak özellikle şu durumlarda kritik hale gelirler:

- Aynı türden widget'ların yer aldığı bir koleksiyonu (örneğin bir liste) yeniden sıraladığınızda veya değiştirdiğinizde.
- Widget'ların durumunu (state) korumak istediğinizde (örneğin, sekmeler arasında geçiş yaparken bir listenin kaydırma konumunu korumak).
- Testler sırasında belirli bir widget'ı kolayca bulmak istediğinizde.

```
Text(
  "Benzersiz bir öğe",
  key: ValueKey("item-id-0025"),
)
```

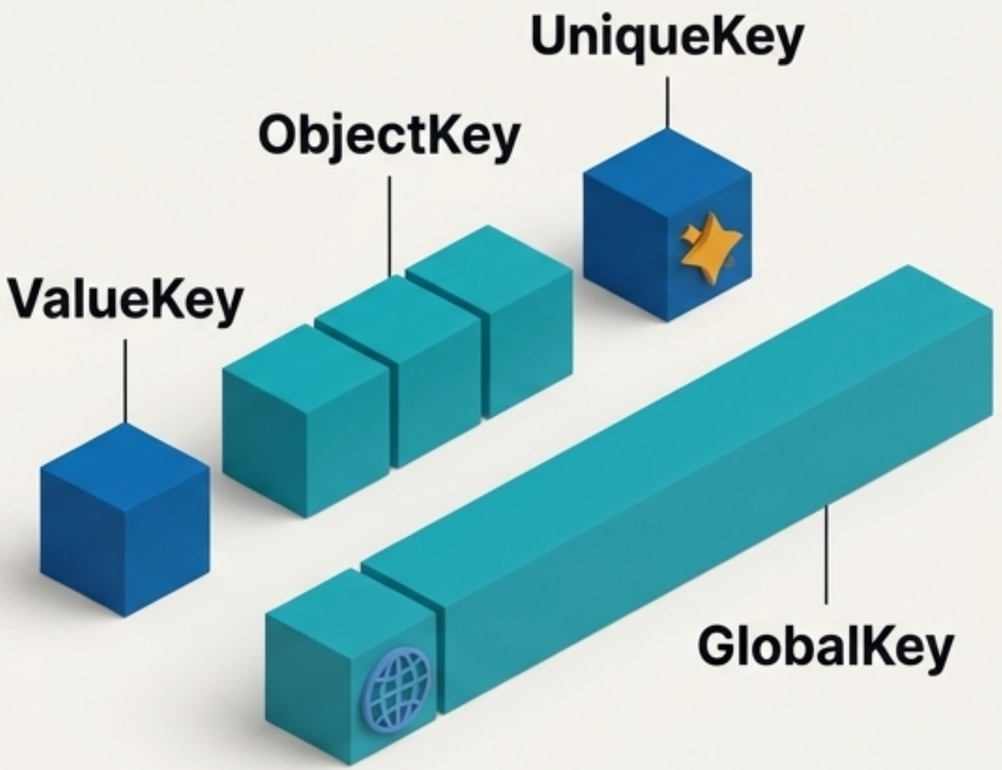
Anahtar Türleri: Hangi Durumda Hangisi Kullanılır?

Her durum için farklı bir anahtar türü vardır. Seçim, widget'ı benzersiz kılan verinin doğasına bağlıdır.

ValueKey

Kullanım: Widget'ı temsil eden tekil, basit ve değişmez bir değer (bir ID, bir String) olduğunda kullanılır. En yaygın kullanılan anahtar türüdür.

```
ValueKey('user-123')
```



ObjectKey

Kullanım: Tek bir alan benzersiz olmadığında, ancak birden çok özelliğin birleşiminin benzersiz olduğu karmaşık bir nesne olduğunda kullanılır.

```
ObjectKey(Task(owner, date, duration))
```

UniqueKey

Kullanım: Sabit ve benzersiz bir değeriniz olmadığında kullanılır. Bu anahtar yalnızca kendisine eşittir ve her oluşturulduğunda yeni bir tane üretilir.

GlobalKey

Kullanım: Tüm uygulama genelinde benzersiz bir anahtar gerektiğinde kullanılır. Genellikle widget'ların durumuna ağacın farklı yerlerinden erişmek için (örneğin Form yönetimi) kullanılır.

Anahtarlar İş Başında: Kaydırma Konumunu Korumak

Anahtarların gücünü gösteren en iyi örneklerden biri, sekmeli bir yapıda listelerin kaydırma konumunu korumaktır. `PageStorageKey` olmadan, her sekme değişiminde liste başa döner.

Senaryo: İki sekmeniz var ve her ikisinde de uzun, kaydırılabilir bir `ListView` bulunuyor.

Anahtar Olmadan

`PageStorageKey` ile


```
TabBarView(
  children: [
    ListView.builder(
      // Bu anahtar, bu listenin durumunu saklar.
      key: const PageStorageKey('list1'),
      itemBuilder: (context, index) { ... },
    ),
```

```
    ListView.builder(
      // Bu anahtar da ikinci listenin durumunu saklar.
      key: const PageStorageKey('list2'),
      itemBuilder: (context, index) { ... },
    ),
```

Sonuç: `PageStorageKey` kullanarak, kullanıcı sekmeler arasında geçiş yaptığında Flutter her listenin nerede kaldığını hatırlar ve kusursuz bir kullanıcı deneyimi sunar. Yapı taşlarınızı doğru bir şekilde tanımlamak, bu tür gelişmiş davranışları mümkün kılar.