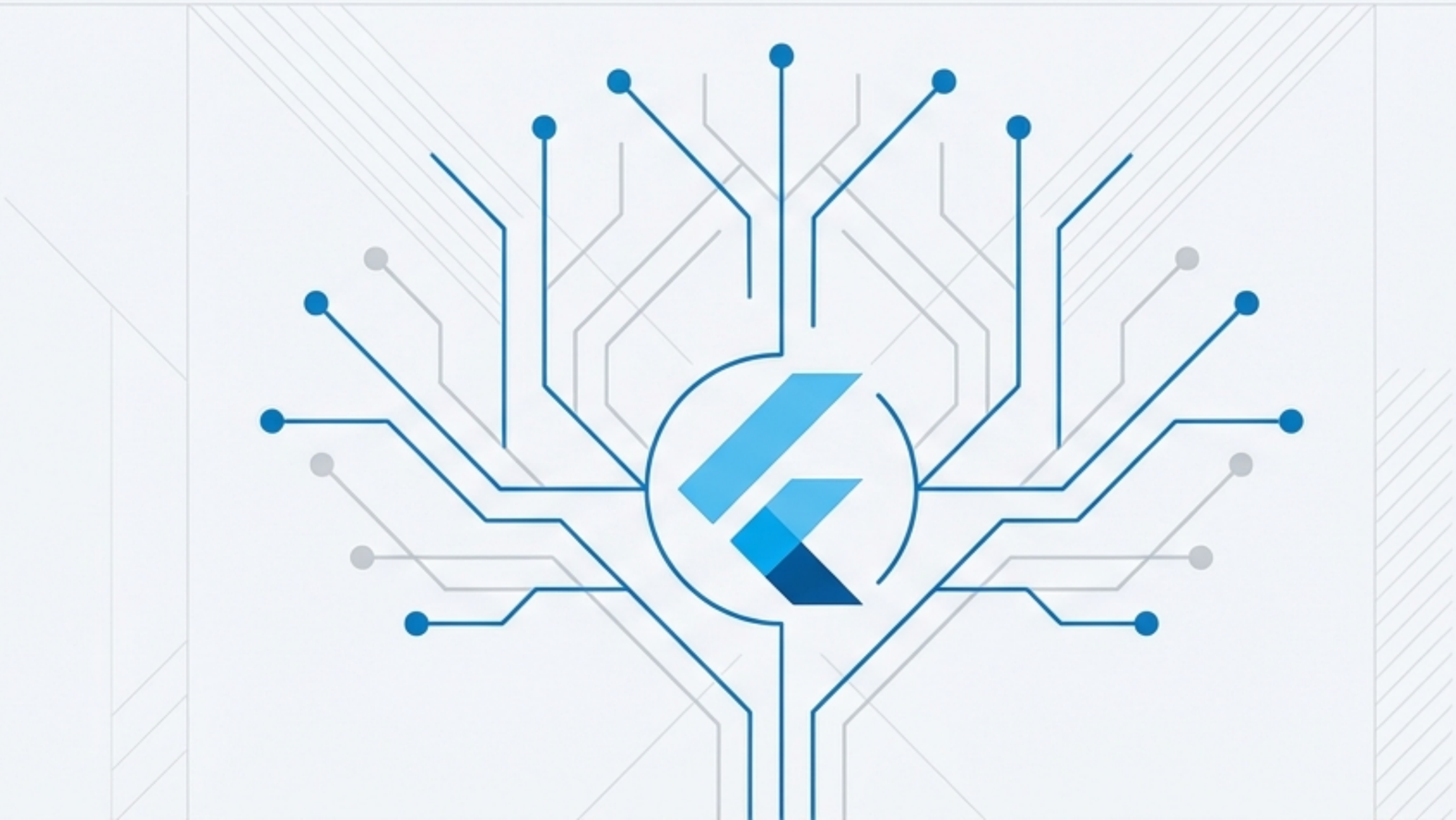
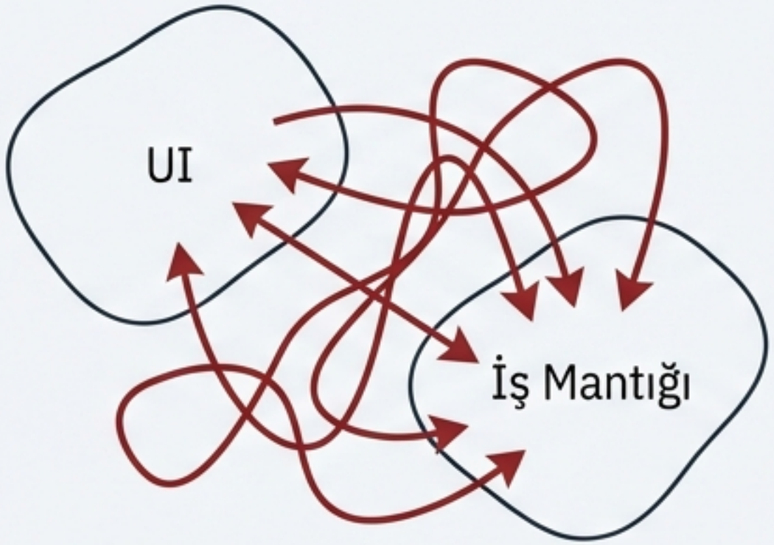
Flutter'da Durum Yönetimi: `Provider` ile Ustalığa Giden Yol

Kodunuzu basitleştirin, performansı artırın ve ölçeklenebilir uygulamalar yaratın.



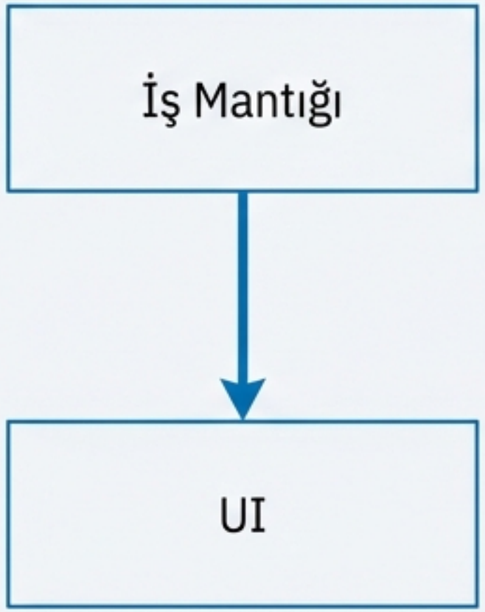
Neden `setState`'in Ötesine Geçmeliyiz?

Geleneksel Yaklaşımın Zorlukları (`setState`)



- Arayüz (UI) ve İş Mantığı (Business Logic) iç içe geçer.
- Durumu widget ağacının derinliklerine iletmek karmaşıklaşır ve kodun okunabilirliğini düşürür ("prop drilling").
- Tek Sorumluluk Prensibi (Single Responsibility Principle) ihlal edilir.

`Provider`'ın Vaadi



İş Mantığı: `ChangeNotifier` kullanan ayrı bir sınıfta toplanır.

Arayüz (UI): Sadece durumu göstermek ve olayları tetiklemekle sorumlu olur.

Adım 1: Durum Modelini `ChangeNotifier` ile Oluşturmak

- `ChangeNotifier` bir mixin'dir. Temel görevi: Değişiklik olduğunda dinleyicileri (listener) haberdar etmek.
- Bu haber verme işini `notifyListeners()` metodu ile yapar. Bu metot, `setState`'in mantıksal karşılığıdır: Değişkeni güncelledikten sonra UI'a 'yeniden çiz" sinyali gönderir.

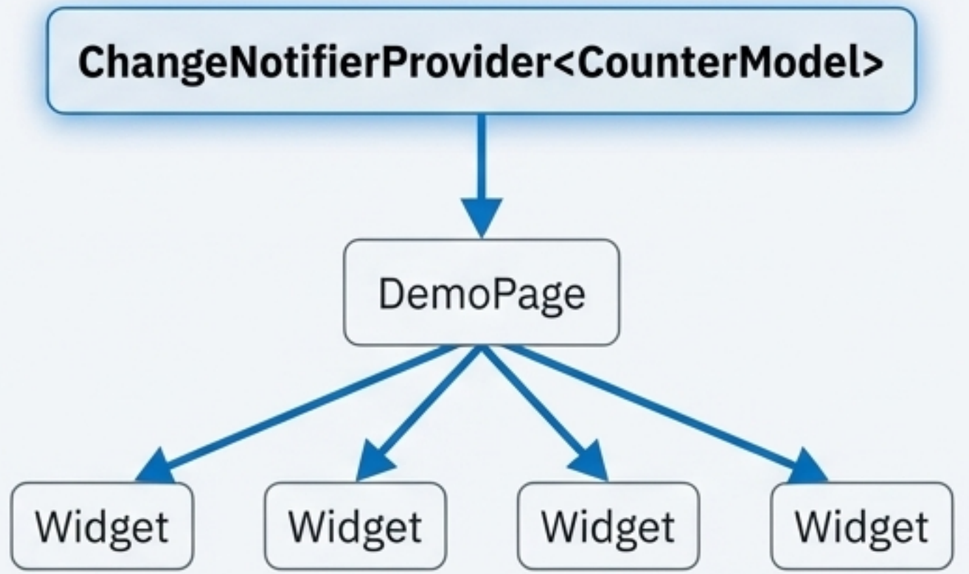
```dart
// Mixin, notifyListeners() metodunu içerdiği için gereklidir
class CounterModel with ChangeNotifier {
  int _counter = 0;
  int get currentCount => _counter;

  void increment() {
    _counter++;
    notifyListeners();
  }
}
```

`ChangeNotifier` mixin'i kullanılır.

UI'ı güncelleme sinyali

Adım 2: Modeli `ChangeNotifierProvider` ile Ağaca Sunmak

- `ChangeNotifierProvider`, modelimizin bir örneğini (instance) oluşturur ve onu widget ağacına ‘sunar’.
- `child’ olarak tanımlanan widget ve onun altındaki tüm widget’lar bu örneğe erişebilir.
- Bu, ağacın tepesine tüm alt dalların ulaşabileceği bir ‘veri deposu’ yerleştirmek gibidir.

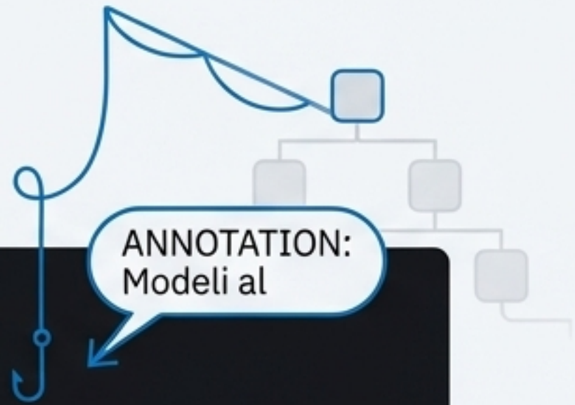



```
ChangeNotifierProvider(
  create: (context) => CounterModel(),
  child: const DemoPage(),
);
```

Model burada yaratılıyor ve sunuluyor. Artık `DemoPage` ve altındakiler `CounterModel`'a erişebilir.

Adım 3: `Provider.of` ile Duruma Erişmek ve Kullanmak

- `Provider.of<T>(context)` metodu, widget ağacında yukarı doğru bakar ve belirttiğiniz `T` tipindeki (örneğin `CounterModel`) ilk provider'ı bulup onun örneğini size verir.
- Bu örnek üzerinden hem veriyi okuyabilir (`counter.currentCount`) hem de metotları çağırarak durumu değiştirebilirsiniz (`counter.increment()`).




```
// build metodu içinde:

final counter = Provider.of<CounterModel>(context);
```

Veriyi okumak için:

Modeldeki veriyi UI'da göster.

```
Text("${counter.currentCount}"),
```

Metodu çağırmak için:

Modeldeki metodu tetikleyerek durumu değiştir.

```
onPressed: () => counter.increment(),
```

`Provider` Akışının Özeti: 3 Adımda Reaktif Döngü

1 UI Olayı

Kullanıcı butona tıklar -> `counter.increment()` çağrılır.

2 Model Güncellenir

`_counter` değeri artar ve `notifyListeners()` tetiklenir.

3 UI Yeniden Çizilir

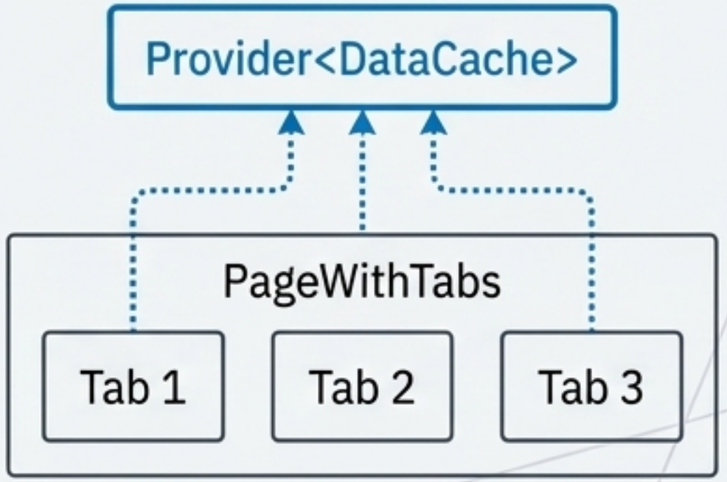
`ChangeNotifierProvider` değişikliği algılar ve `Provider.of` kullanan dinleyici widget'ların yeniden çizilmesini (rebuild) sağlar.

Bu döngü, UI ve iş mantığını birbirinden ayırarak temiz, test edilebilir ve reaktif bir yapı oluşturur.

Bildirimlerin Ötesinde: Bağımlılık Yönetimi için `Provider`

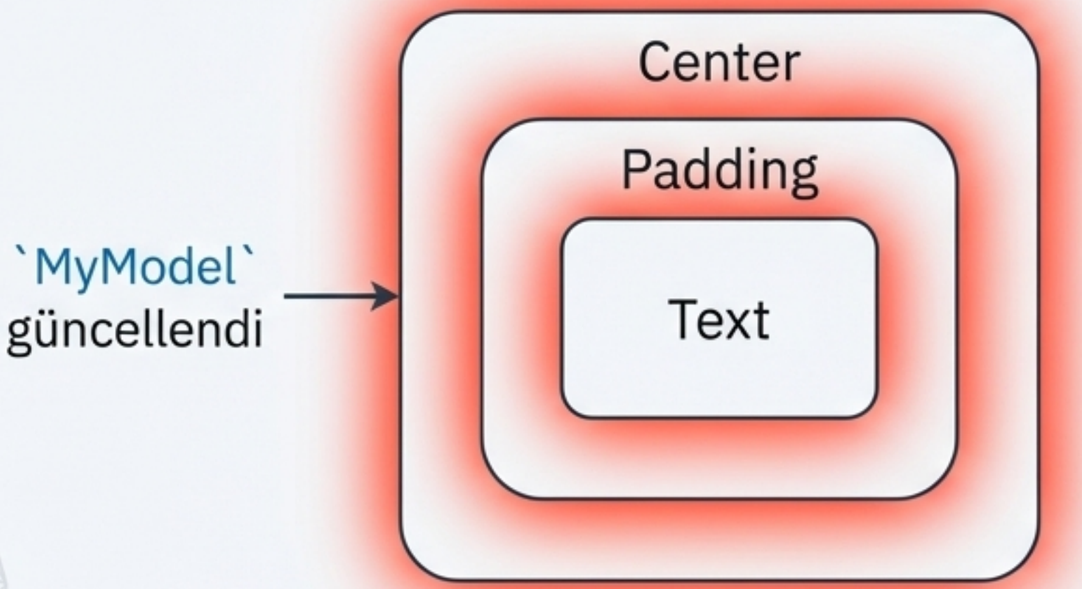
- `ChangeNotifierProvider` değişikikleri dinler. Peki ya sadece bir nesneyi (örneğin bir servis sınıfı veya veri cache'i) yeniden çizim tetiklemeden ağaca sunmak istersek?;
- Temel `Provider` widgetı tam olarak bu işi yapar. Bir nesneyi oluşturur ve alt widget'ların erişimine sunar, ancak değişiklikleri dinlemez.
- Bu, servis sınıflarını 'inject' etmek veya verileri sayfalar arasında taşımak için mükemmel bir yöntemdir.

```
// DataCache sınıfı, sayfalar arasında veri tutan bir depodur.
// Provider ile oluşturulduğunda, PageWithTabs ve alt sekmeleri
// aynı DataCache örneğine erişebilir.
Provider(
  create: (context) => DataCache(),
  child: PageWithTabs(),
);
```



Performans Sorunu: Gereksiz Yeniden Çizimler

`Provider.of(context)` kullanıldığında, `build` metodunun **tamamı** yeniden çalışır. Sadece `Text` widget'ı değişse bile, `Center` ve `Padding` gibi statik widget'lar da gereksiz yere yeniden çizilir.

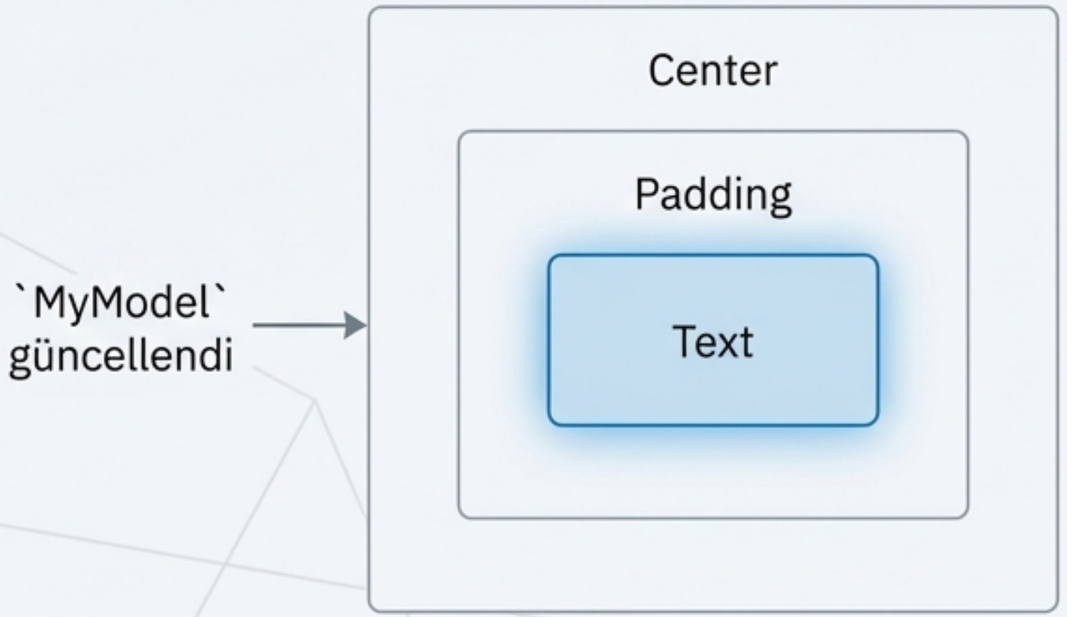


Sorunlu Kod

```
// TÜM WIDGET YENİDEN ÇİZİLİR
build(context) {
  final value = Provider.of<MyModel>(context);
  return Center(
    child: Padding(
      padding: EdgeInsets.all(15),
      child: Text("${value.text}"),
    ),
  );
}
```

Çözüm: `Consumer` ile Granüler Güncellemeler

`Consumer` widget'ı, yeniden çizilmesi gereken kısmı tam olarak sarmalar. Değişiklik olduğunda, sadece `builder` fonksiyonu içindeki widget'lar güncellenir. Bu, maksimum performans için 'nokta atışı' yeniden çizim sağlar.



Optimize Edilmiş Kod

```
// SADECE TEXT WIDGET'I YENİDEN ÇİZİLİR
build(context) {
  return Center(
    child: Padding(
      padding: EdgeInsets.all(15),
      child: Consumer<MyModel>(
        builder: (_, model, __) => Text("${model.text}"),
      ),
    ),
  );
}
```

Usta Seviye Optimizasyon: `Selector`

- `Consumer`'dan bile daha hassas kontrol gerektiğinde kullanılır.
- Bir modelin sadece belirli bir parçasındaki değişikliği dinlemek istediğinizde idealdir. Örneğin, bir `User` modelinde sadece `name` değiştiğinde yeniden çizim yapmak, `age` değiştiğindeyse yapmamak.
- `selector` callback'i, dinlenecek değeri döndürür. `builder` sadece bu değer değiştiğinde çalışır.

```
Selector<Person, String>(
  selector: (context, person) => person.name,
  builder: (context, name, child) {
    // Bu builder sadece 'name' değiştiğinde çalışır.
    return Text(name);
  }
);
```

Sadece `name` alanını dinle. Modeldeki diğer alanlar değişse bile yeniden çizim tetiklenmez.

Ayrıca, `shouldRebuild` parametresi ile yeniden çizim için özel bir mantık da tanımlayabilirsiniz.

Asenkron Dünya: `FutureProvider` ile Veri Yükleme

Bir `Future`'ın (örneğin bir API çağrısı) sonucunu widget ağacına sunmanın en zarif yolu. Yüklenme (loading), hata (error) ve veri (data) durumlarını yönetmeyi basitleştirir, sizi `FutureBuilder` karmaşasından kurtarır.

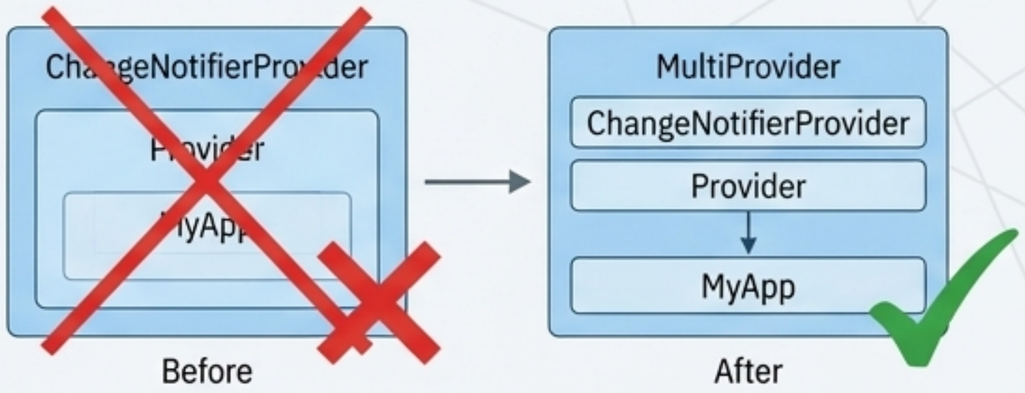
```
FutureProvider<MyData>(
  create: (_) => fetchNetworkData(),
  catchError: (context, error) => MyData.withError(error),
  initialData: MyData.initial(),
  child: MyPage(),
);
```

- create: Çalıştırılacak `Future`ı döndürür.
- catchError: `Future` hata verdiğinde çağrılır. Hata verebilecek `Future`lar için kullanılması zorunludur.
- lazy: `true` (varsayılan) ise `Future` sadece ilk kez okunduğunda çalışır. `false` ise anında çalışır.

Temiz Kod Mimarisi: `MultiProvider` ve Doğru Kapsam

`MultiProvider` ile Kod Düzeni

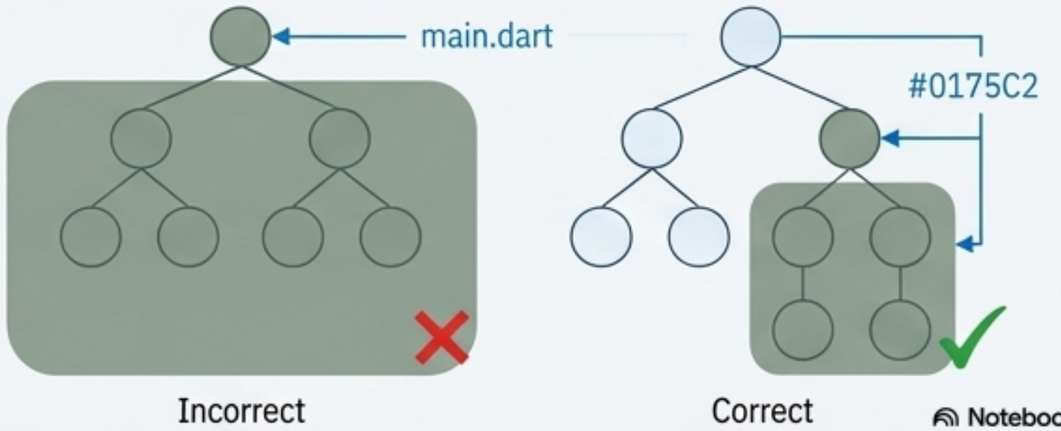
- Uygulamanızda birden çok provider kullanmanız gerektiğinde, onları iç içe yazarak ‘Provider Cehennemi’ (nested providers) oluşturmaktan kaçının.
- `MultiProvider’, tüm provider’larınızı tek bir listede, okunabilir bir şekilde gruplamanızı sağlar.




```
MultiProvider(
  providers: [
    ChangeNotifierProvider(create: (_) => AuthModel()),
    Provider(create: (_) => ApiService()),
  ],
  child: const MyApp(),
)
```

İyi Pratik: Doğru Kapsam (Scoping)

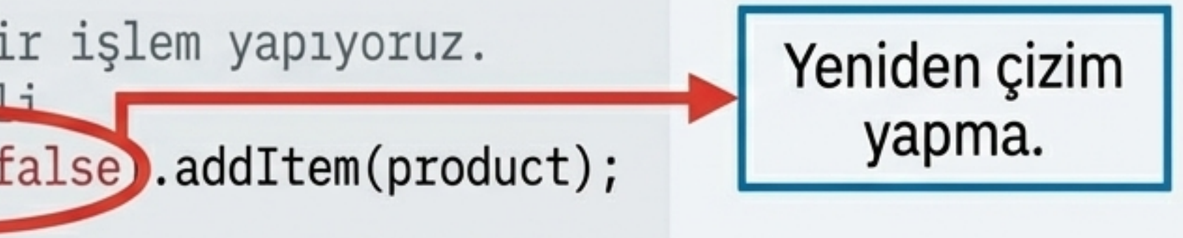
Bir provider’ı, ona ihtiyaç duyan ilk widget’ın hemen üzerine yerleştirin. Onu gereğinden fazla ‘yukarıya’ ‘yukarıya’ (örneğin `main.dart`’a) koymak, kapsamı kirletir ve gereksizdir.



Kritik Parametre: `listen: false` Ne Zaman Kullanılır?

- **Amaç:** Modele sadece bir metot çağırmak için erişmek istediğinizde kullanılır. UI'ın bu değişikliği dinleyip yeniden çizilmesine gerek yoksa, `listen: false` ile gereksiz `rebuild''leri önlersiniz.
- **En Yaygın Kullanım Yeri:** `onPressed` veya benzeri bir callback fonksiyonu içinde.

```
onPressed: () {
  // UI'ın dinlemesine gerek yok, sadece bir işlem yapıyoruz.
  // Bu nedenle yeniden çizim tetiklenmemeli.
  Provider.of<CartModel>(context, listen: false).addItem(product);
}
```



⚠ ZORUNLULUK

`Provider.of''u `build` metodu **DIŞINDA** (`initState`, bir callback vb.) çağırdığınızda `listen: false` kullanmak zorunludur. Aksi takdirde çalışma zamanı hatası (runtime exception) alırsınız.

Daha Az Kod, Daha Çok Anlam: Modern Provider Sözdizimi

`provider` 4.1.0 ve sonrası sürümlerle gelen `BuildContext` extension metotları, kodu daha kısa ve okunabilir hale getirir.

Amaç	Eski Yöntem (Detaylı)	Yeni Yöntem (Modern ve Okunabilir)
Dinle ve Yeniden Çiz	`Provider.of<T>(context)`	`context.watch<T>()`
Sadece Oku (Dinleme)	`Provider.of<T>(context, listen: false)`	`context.read<T>()`
Seçerek Dinle	`Selector<T, R>(...)`	`context.select<T, R>(...)`

Öneri: Yeni ve daha modern oldukları için eski yöntemler yerine bu extension metotlarını tercih edin.

Nihai Özet: Hangi Durumda Neyi Kullanmalı?