Flutter ile Ağ İşlemleri ve HTTP Yönetimi

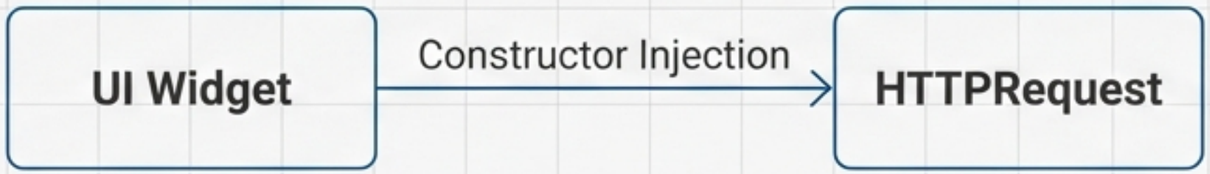
Sağlam, Ölçeklenebilir ve Verimli API Bağlantıları Kurmak

Temeller: Sürdürülebilirlik İçin Mimari

Kodumuz zaman içinde kolayca bakımı yapılabilir ve okunabilir olmalıdır. Bu noktada Tek Sorumluluk Prensibi (SRP) ve Bağımlılık Enjeksiyonu (DI) devreye girer.

```
// 'extends' değil 'implements' kullanılmalı
abstract class HTTPRequest<T> {
  Future<T> execute();
}
```

Bu arayüz, UI widget'larına kurucu (constructor) aracılığıyla enjekte edilecek ve SRP'ye saygı duyulmasını sağlayacaktır.



Sözleşmeyi Uygulamak: GET İstekleri

```dart
import 'package:http/http.dart' as http;

class RequestItem implements HTTPRequest<Item> {
  final String url;
  const RequestItem({required this.url});

  Future<Item> execute() async {
    final response = await http.get(Uri.parse(url));

    if (response.statusCode != 200) {
      throw http.ClientException('Hata oluştu!');
    }
    return _parseJson(response.body);
  }

  Item _parseJson(String response) => Item.fromJson(jsonDecode(response));
}
```

İstek: http.get(url) asenkron çalışır ve bir Response döner.

Doğrulama: 200 harici kodlar (örn. 404, 500) hata olarak ele alınmalıdır.

Dönüşüm: JSON verisi _parseJson ile model sınıfına çevrilir.

Yanıtın (Response) Anatomisi

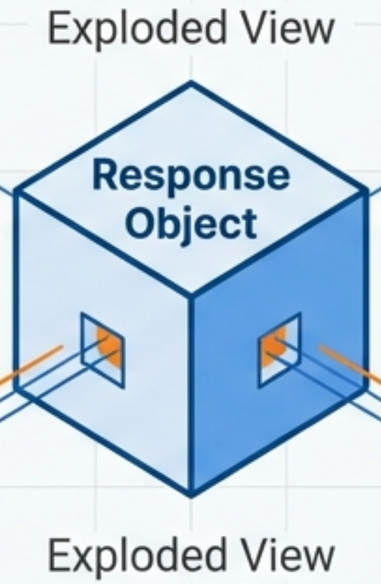
http paketi sayesinde asenkron istekler sonucunda zengin özelliklere sahip bir Response nesnesi elde ederiz.

`body (String)`

Yanıtın gövdesi (payload). Genellikle JSON formatındadır.

`statusCode (int)`

İstek durum kodu. Başarılı (200), Bulunamadı (404) veya Sunucu Hatası (500).



`contentLength (int)`

Yanıt gövdesinin bayt cinsinden boyutu.

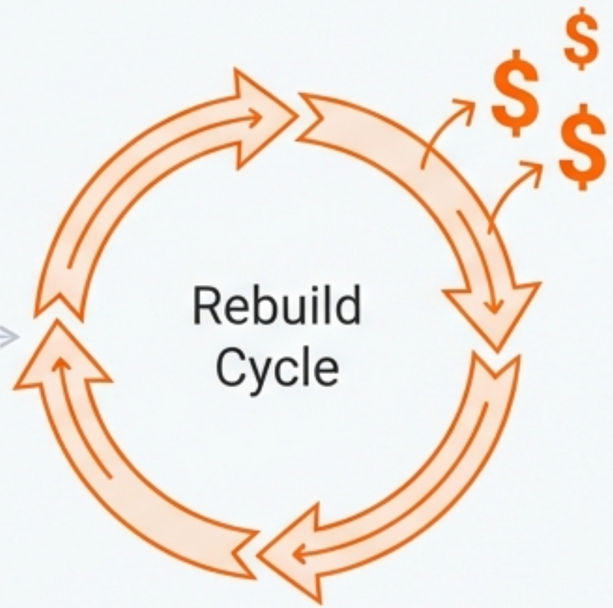
`headers (Map)`

Sunucudan dönen başlık bilgileri.

⚠ Kritik Hata: Build Metodu Tuzağı

⚠ API çağrılarını build metodunun içine yerleştirmek, kaynak israfına ve gereksiz tekrarlara yol açar.

```dart
@override
Widget build(BuildContext context) {
  // BU KESİNLİKLE YANLIŞTIR
  final futureItem = widget._request.execute();
  // ...
}
```



Flutter build metodunu tahmin edemeyeceğiniz sıklıkta çağırır. Yukarıdaki kod, widget her yeniden oluşturulduğunda (rebuild) yeni bir HTTP isteği başlatır.

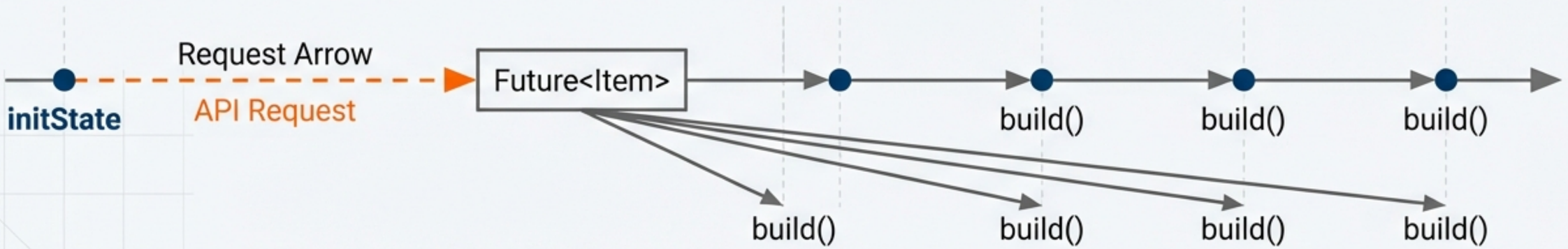
Çözüm: Future Önbellekleme

HTTP istekleri gibi Future işlemlerinde, nesneyi önbelleğe almak ve yalnızca bir kez kullanmak için StatefulWIdget kullanılmalıdır.

```dart
class _HTTPWidgetState extends State<HTTPWidget> {
  late final Future<Item> futureItem;

  @override
  void initState() {
    super.initState();
    // execute() yalnızca bir kez, başlangıçta çalışır.
    futureItem = widget._request.execute();
  }
}
```

Anahtar Nokta: '**late final**' değişken kullanarak değeri **initState** içinde yalnızca bir kez atarız. Yeniden oluşturma (rebuild) durumunda API çağrısı tekrarlanmaz.



Tek bir istek initState'de başlatılır, sonraki build'ler önbelleğe alınmış Future'ı kullanır.

Veriyi Görselleştirme: FutureBuilder

FutureBuilder, bekleme süresini, hataları ve başarı durumunu yönetmek için en temiz yöntemdir.

```
FutureBuilder<Item>(
  future: futureItem,
  builder: (context, snapshot) {
    if (snapshot.hasError) return const ErrorItemWidget();
    if (snapshot.hasData) return SuccessItemWidget(snapshot.data);

    return const Center(child: CircularProgressIndicator());
  }
);
```

Veri Gönderimi: POST İstekleri

POST isteği GET'ten farksızdır, ancak sunucuya gönderilecek bir veri yükü (payload) gerektirir.

```
final response = await http.post(url,
  body: 'Bu string verisini gönder',
  encoding: Encoding.getByName('utf-8'),
);
```

Body Types

String

İçerik tipi otomatik olarak text/plain ayarlanır.

List

İstek gövdesi için bayt listesi olarak kullanılır.

Map

Form alanları gibi davranır ve içerik tipi application/x-www-form-urlencoded olarak ayarlanır.

Başlıklar ve Kimlik Doğrulama

API'ler genellikle 'Authorization' gibi özel başlıklar (headers) gerektirir. Bu başlıklar hem get() hem de post() isteklerinde bir Map olarak gönderilir.

```
final response = await http.get(url,
  headers: {
    'Authorization': 'your_api_key_here',
    'Content-Type': 'application/json',
  }
);
```

Dikkat: Sunucu, anahtarı 'x-api-key' gibi farklı bir isimle talep edebilir. Kodlamadan önce API dokümantasyonunu mutlaka kontrol edin.

Optimizasyon: HTTP İstemcisi

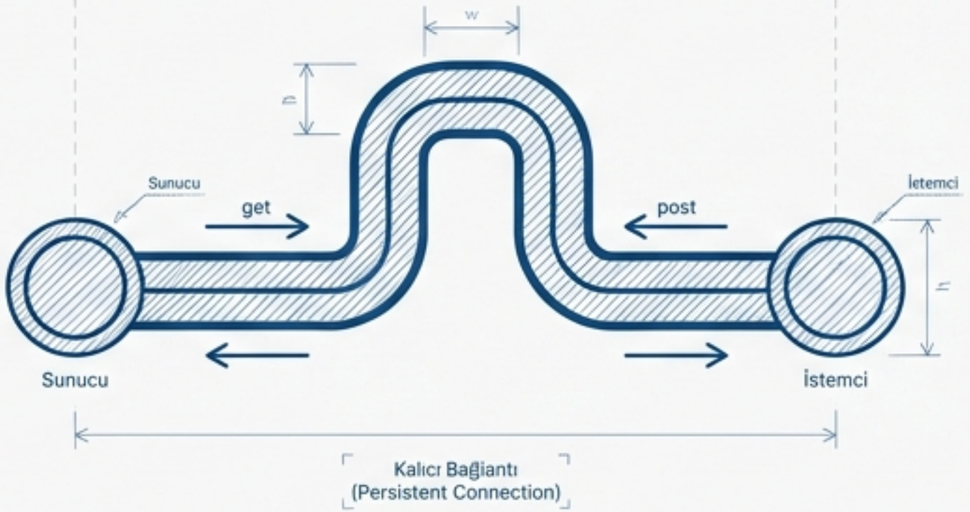
Tekil İstek

http.get(...) kullanmak yeterlidir.
Geçici Client oluşturur ve kapatır.

Çoklu İstek (Client)

Aynı sunucuya birden fazla istek yapılacaksa http.Client() kullanılmalıdır.

```dart
final client = http.Client();
try {
  final one = await client.get(...);
  final two = await client.post(...);
} finally {
  client.close(); // İşlem bitince mutlaka kapatılmalıdır!
}
```



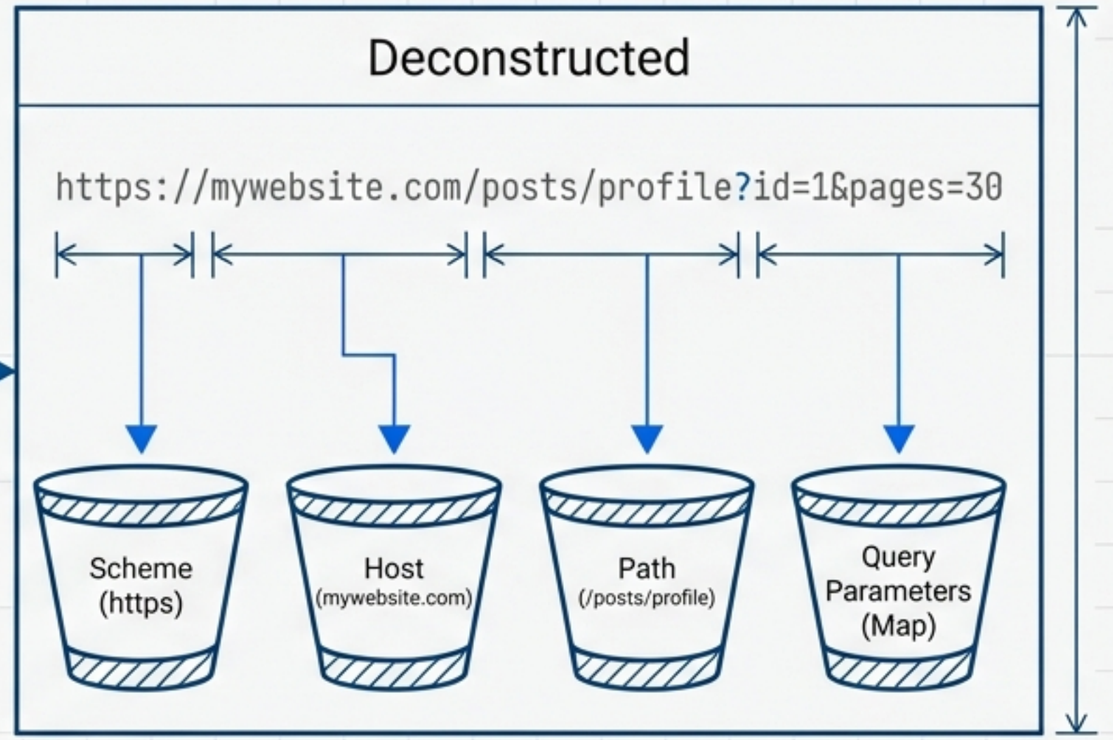
Hassas Kontrol: URI Nesnesi

Karmaşık URL'ler ve sorgu parametreleri (query parameters) ile çalışırken basit string birleştirme yerine Uri nesnesi kullanılmalıdır.

```dart
final params = {'id': '1', 'pages': '30'};

// Domain, path ve parametreleri ayrıştırın
final uri = Uri.https('mywebsite.com', '/posts/profile', params);

await http.get(uri);
```



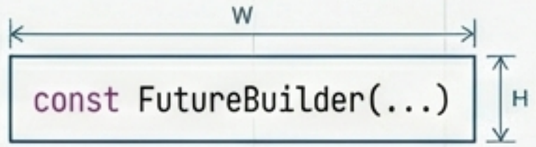
Bu yöntem daha esnektir ve özel karakterlerin güvenli bir şekilde işlenmesini (encoding) sağlar.

Özet ve En İyi Pratikler

Önbellekleme: Gereksiz ağ trafiğini önlemek için Future isteklerini her zaman initstate içinde başlatın.

Kullanıcı Deneyimi: FutureBuilder kullanarak yükleniyor, hata ve başarı durumlarını kullanıcıya yansıtın.

Performans: Sıkça rebuild olan widget'larda (özellikle FutureBuilder içinde) mümkün olduğunca **const** constructor kullanın.



Kaynak Yönetimi: Eğer http.Client() kullanıyorsanız, işiniz bittiğinde **client.close()** ile bağlantıyı sonlandırmayı unutmayın.

Güvenlik: API anahtarlarını ve başlıkları (headers) dokümantasyona uygun şekilde yönetin.

Artık Uygulamalarınız Dünyaya Bağlı.

Doğru mimari ile kurulan bağlantılar, sadece veri taşımaz; güven ve performans taşır.