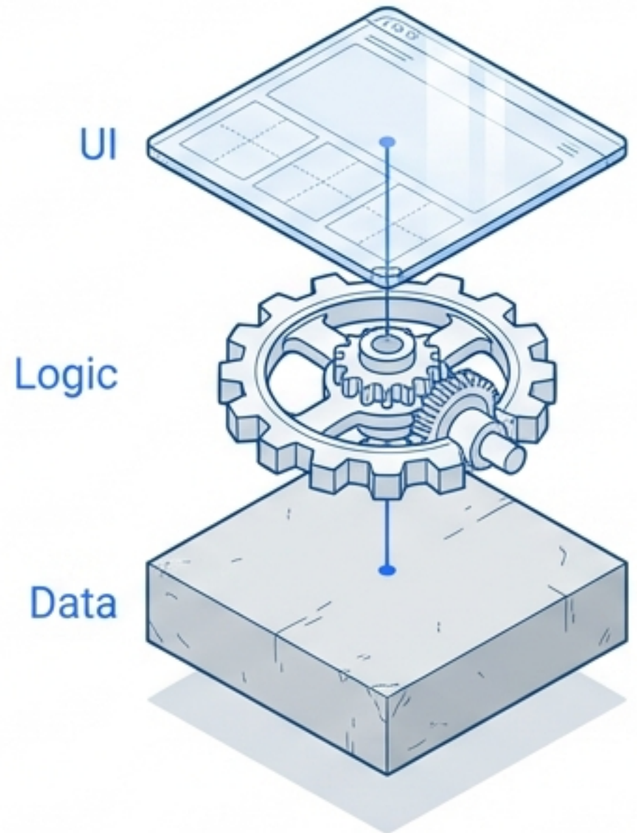
Flutter’da İleri Seviye REST API Mimarisi

REST API Mimarisi

Verimli Veri Çekme, JSON Ayrıştırma ve SOLID Prensipleri

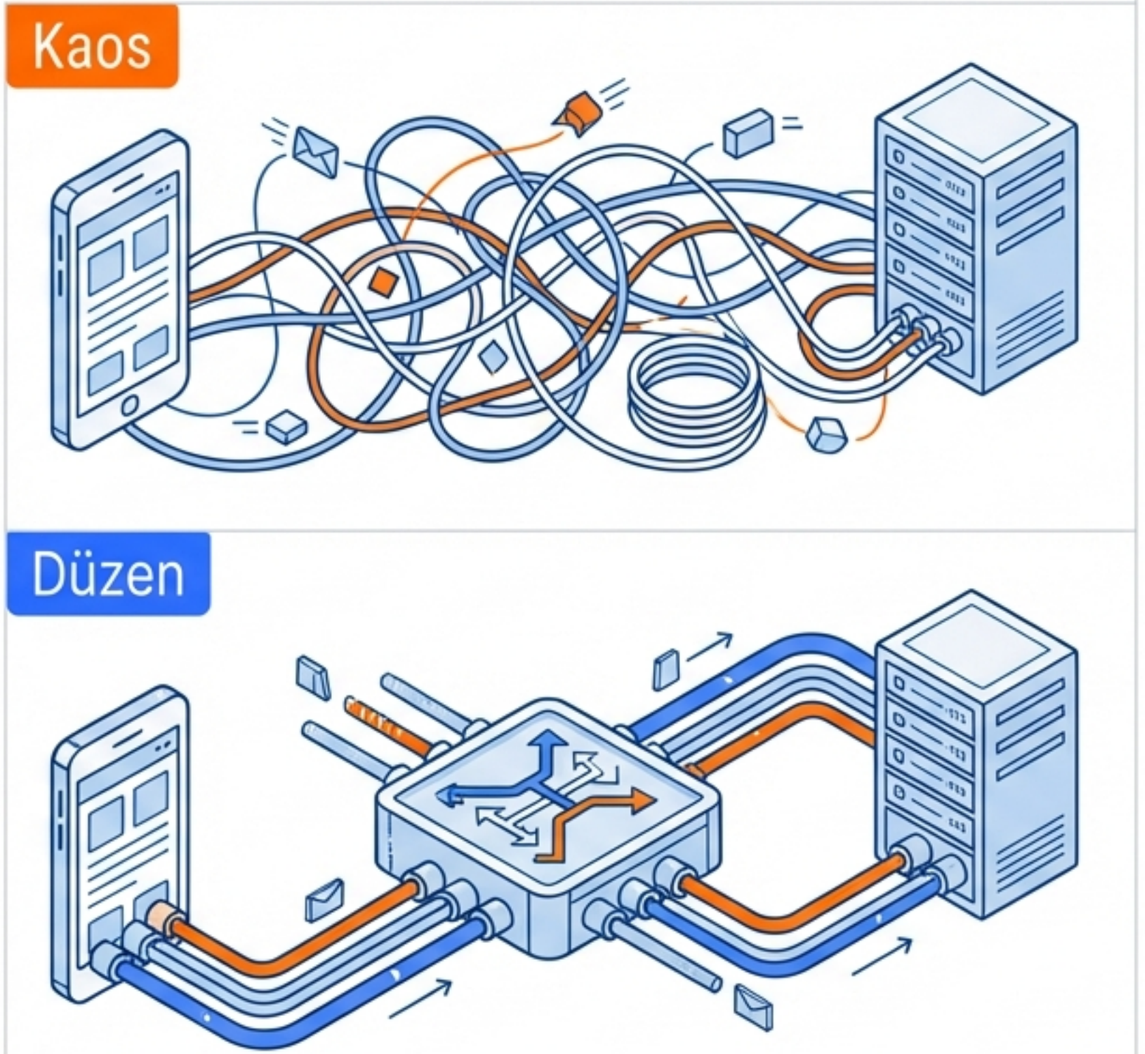


Ölçeklenebilir Yazılım Mimarisi Serisi

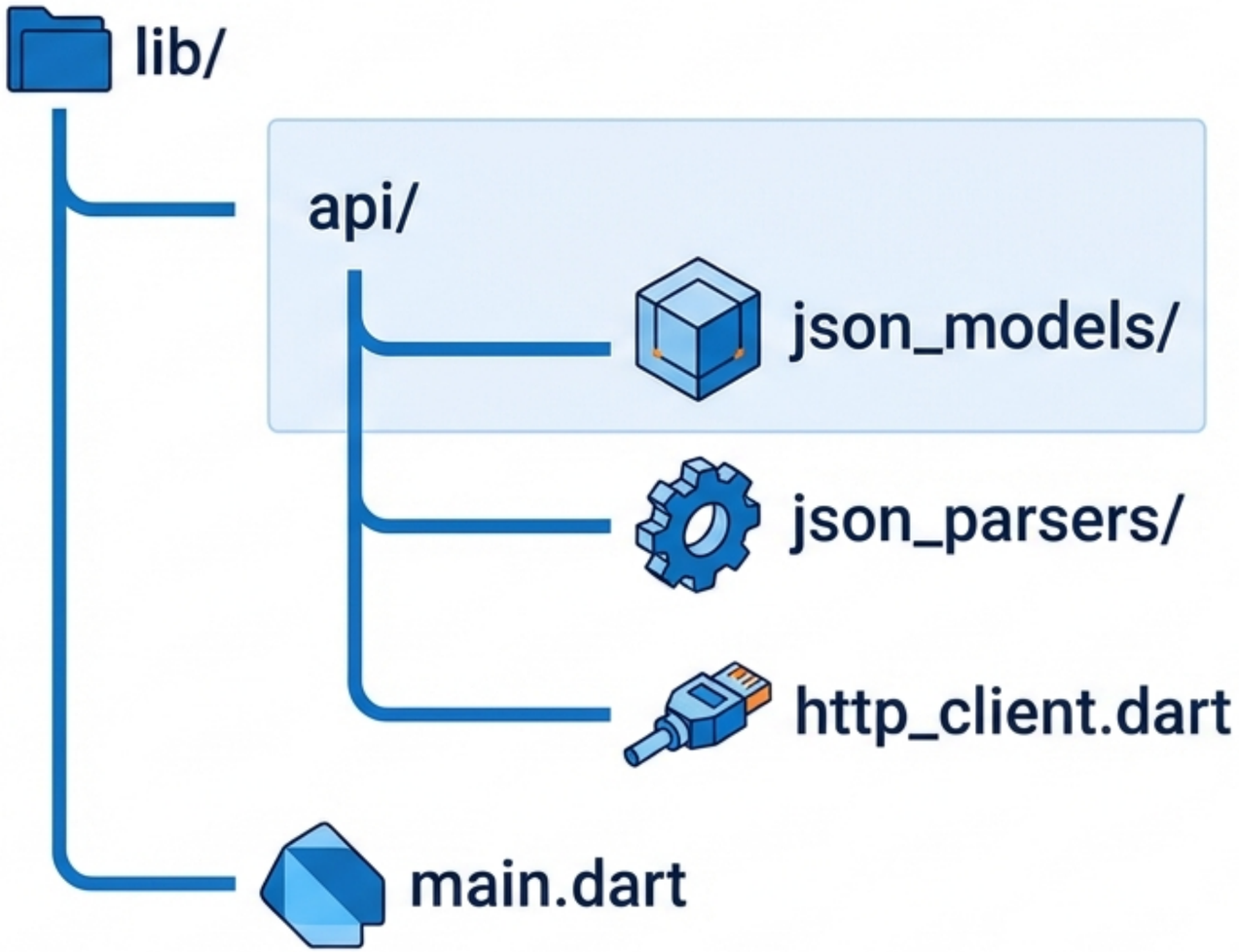
Amaç: Sadece Veri Çekmek Değil, Yönetilebilir Kod Yazmak

Firebase kullanılmayan projelerde, standart bir JSON REST servisiyle (örneğin `jsonplaceholder`) iletişim kurarken ölçeklenebilir bir yapıya ihtiyaç vardır.

- **Sürdürülebilirlik:** Kodun bakımı ve güncellenmesi kolay olmalı.
- **Ayrıştırma (Separation of Concerns):** Her sınıfın tek bir görevi olmalı.
- **Standartlaştırma:** Tüm API istekleri ortak bir mantıkla yönetilmeli.



Mimari Planı: Klasör Yapısı



Veri Tanımları:
JSON verisini Dart nesnesine çeviren sınıflar.

İş Mantığı:
Veri işleme ve ayrıştırma katmanı.

İletişim:
Ağ isteklerini yöneten merkezi birim.

1. Katman: Veriyi Modellemek (Single Object)

Örnek: /posts/10 endpoint'i tek bir JSON objesi döner.

```dart
// lib/api/json_models/post.dart
part 'post.g.dart';

@JsonSerializable()
class Post {
  final int userId;
  final int id;
  final String title;
  final String body;

  const Post(this.userId, this.id, this.title, this.body);

  factory Post.fromJson(Map<String, dynamic> json) =>
      _$PostFromJson(json);

  Map<String, dynamic> toJson() => _$PostToJson(this);
}
```

Otomatik Kod Üretimi

Generated dosyasından gelen metot

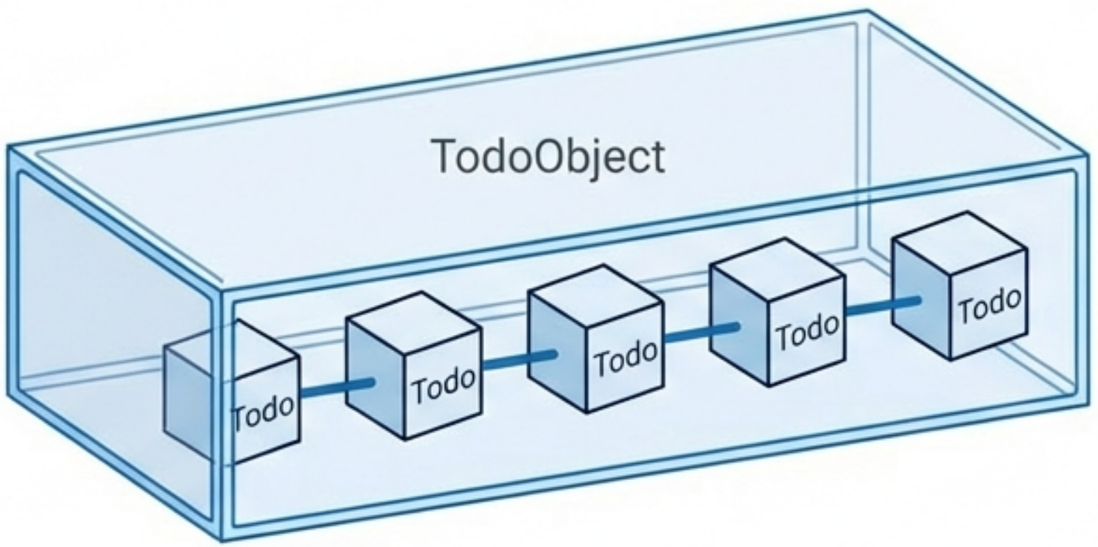
json_serializable paketi kullanılarak, manuel mapping hatasından kurtuluyoruz. `flutter pub run build_runner build` komutu arka plandaki işi yapar.

Karmaşık Yapılar ve Listeler

Senaryo A: Düz Liste (Direct List)

API doğrudan `[...]` dönerse, liste manuel olarak işlenir. Sadece içerideki obje modellenir.

Senaryo B: Sarılmış Liste (Nested List)

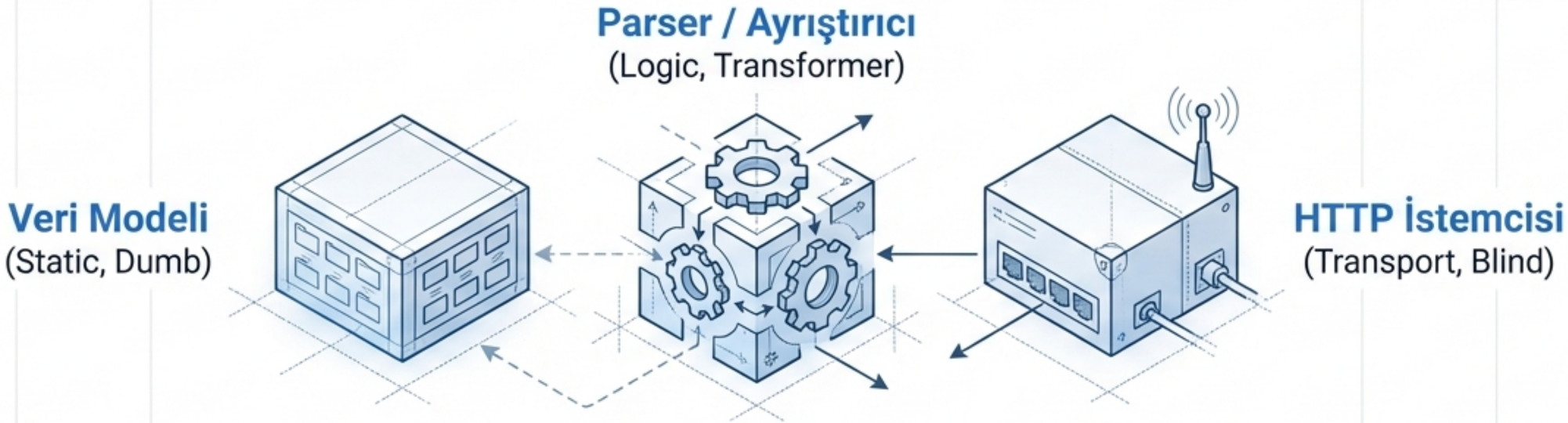



```dart
@JsonSerializable(explicitToJson: true)
class TodoObject {
  final List<Todo> data;
  const TodoObject(this.data);
  // ... factory methods
}
```

Eğer JSON `{'data': [...]}` şeklindeyse, listeyi tutan bir kapsayıcı (wrapper) model gereklidir.

2. Katman: Ayrıştırma Mantığı ve SOLID

Neden model ve istemci arasına bir katman daha koyuyoruz?



1. Single Responsibility Principle (Tek Sorumluluk): Sınıflar küçük, öz ve amaca yönelik olmalı. İstemci veriyi taşır, Parser veriyi anlar.

2. Bakım Kolaylığı: Parser değiştirmek, ağ katmanını veya UI'ı bozmamalı.

3. Zayıf Bağımlılık (Loose Coupling): Hiyerarşide güçlü bağımlılıklar olmamalı.

Temel Yapı ve Mixin Kullanımı

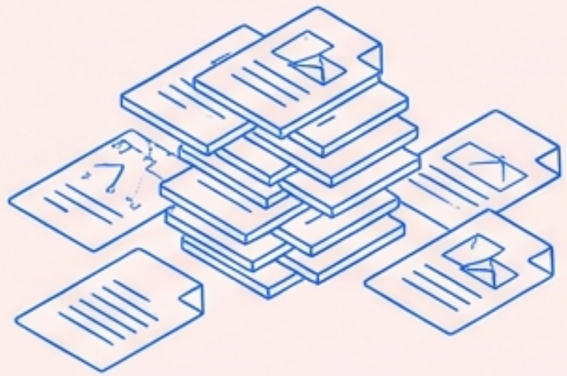
Kod tekrarını önlemek için yetenek tabanlı ayrıştırma

```dart
// lib/api/json_parsers/json_parser.dart

abstract class JsonParser<T> {
  const JsonParser();
  Future<T> parseFromJson(String json);
}

mixin ObjectDecoder<T> on JsonParser<T> {
  Map<String, dynamic> decodeJsonObject(String json) =>
    jsonDecode(json) as Map<String, dynamic>;
}

mixin ListDecoder<T> on JsonParser<T> {
  List<dynamic> decodeJsonList(String json) =>
    jsonDecode(json) as List<dynamic>;
}
```

Merkezi decode mantığı. Her parser kendi `**jsonDecode**''unu yazmak zorunda kalmaz.

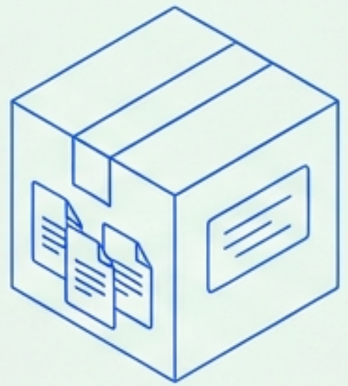
Kütüphane Yönetimi: export Modeli

Kötü Pratik (Individual Imports)



```
import '.../todo_parser.dart';
import '.../post_parser.dart';
import '.../user_parser.dart';
```

İyi Pratik (Library Export)



```
import 'package:myapp/api/json_parsers/json_parser.dart';
```

lib/api/json_parsers/json_parser.dart

```
library json_parser;
export "post_parser.dart";
export "todo_parser.dart";
export "object_decoder.dart";
```

Tek bir giriş noktası, daha temiz import listesi.

Somut Ayrıştırıcılar (Concrete Parsers)

PostParser (Tekil Obje)

```
class PostParser extends JsonParser<Post>
        with ObjectDecoder<Post> {
  const PostParser();

  @override
  Future<Post> parseFromJson(String json) async {
    final decoded = decodeJsonObject(json);
    return Post.fromJson(decoded);
  }
}
```

TodoParser (Liste)

```
class TodoParser extends JsonParser<List<Todo>>
      with ListDecoder<List<Todo>> {
  const TodoParser();

  @override
  Future<List<Todo>> parseFromJson(String json) asy
    return decodeJsonList(json)
        .map((value) => Todo.fromJson(value))
        .toList();
  }
}
```

Merkezi decode mantığı. Her parser kendi `jsonDecode`unu yazmak zorunda kalmaz. (Central decode logic. No parser has to write its own `jsonDecode`.)

Geleceğe Hazırlık: Genişletilebilirlik

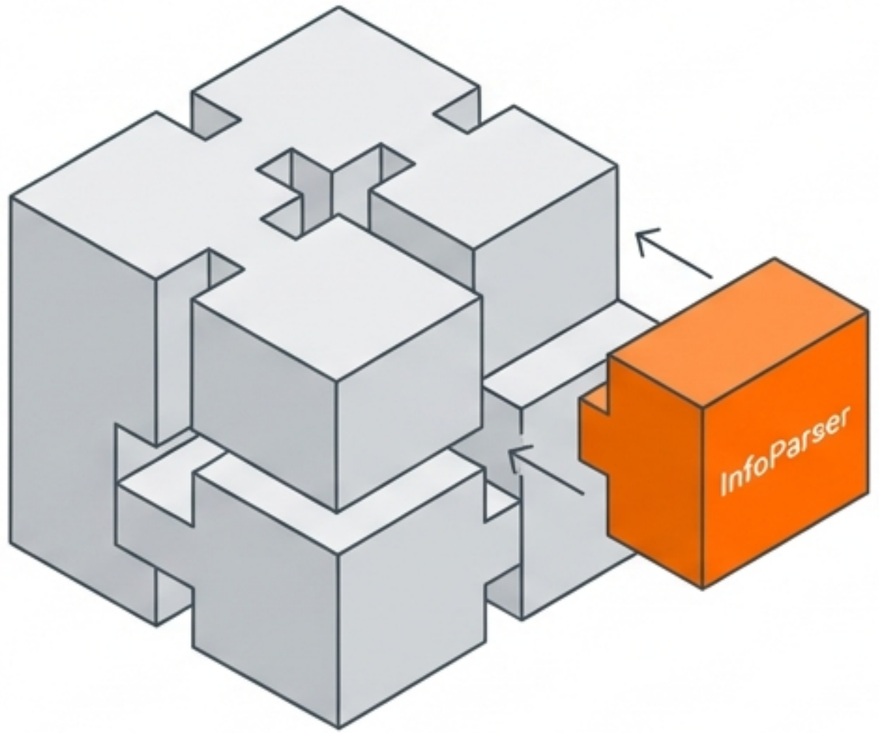
Open-Closed Principle: Gelişime açık, değişime kapalı.

Senaryo: Yeni bir `/info` endpoint'i eklendi.

→ **Info Modelini Oluştur**
Data Class

→ 2. **InfoParser Sınıfını Türet**
extends JsonParser

→ 3. **Mevcut Koda Dokunma**
No changes to HTTP Client or other parsers




```
class InfoParser extends JsonParser<Info> with ObjectDecoder<Info> {
  ...
}
```

Mevcut sistemin kararlılığı korunur.

3. Katman: HTTP İstemcisi (Networking)

`http` paketi yerine, yapılandırma gücü nedeniyle `dio` tercih ediyoruz.

```dart
// lib/api/http_client.dart
class RequestREST {
  final String endpoint;
  final Map<String, dynamic> data;

  const RequestREST({required this.endpoint, this.data = const {}});

  // Statik Client: Her istek için yeniden oluşturulmaz
  static final _client = Dio(
    BaseOptions(
      baseUrl: "https://jsonplaceholder.typicode.com/",
      connectTimeout: 3000,
      receiveTimeout: 3000,
      headers: {"api-key": "your_key_here"},
    )
  );
  // ...
}
```

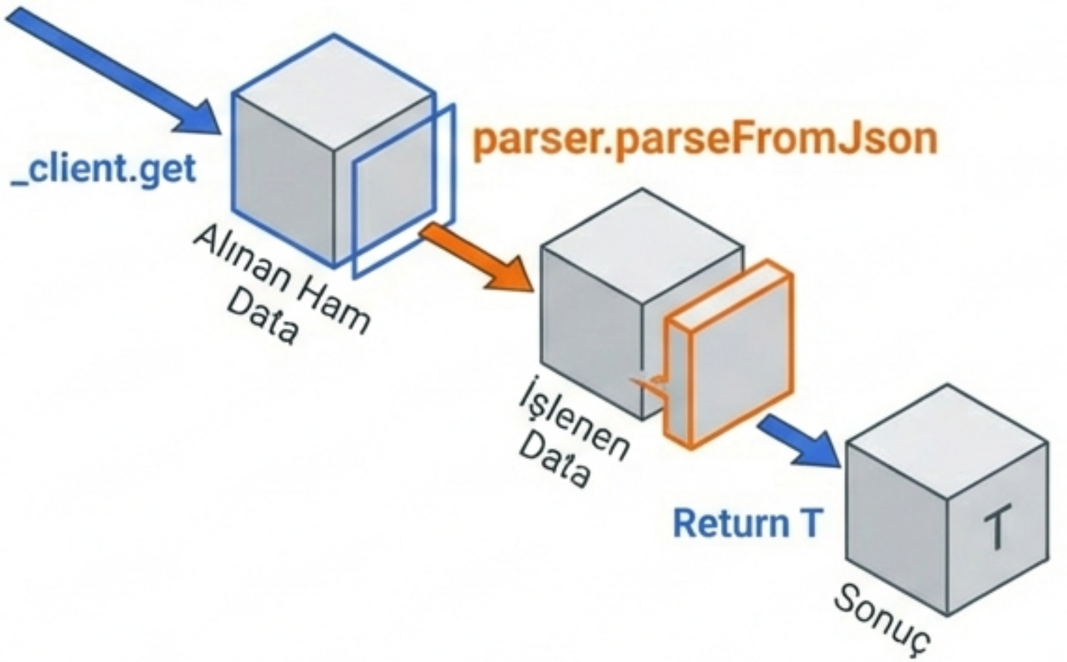
Performans için Tekil Instance

Merkezi Konfigürasyon

Generic İstek Metotları ve Dependency Injection

Sihrin gerçekleştiği yer: İstemci veri tipini bilmez, Parser bilir.

```
Future<T> executeGet<T>(JsonParser<T> parser) {
  async  async
  final response = await _client.get(endpoint);
  return parser.parseFromJson(response.data);
}
```



Metot Enjeksiyonu:
`parser` parametre olarak gelir.
Bu sayede `RequestREST` sınıfı, `Post` veya `Todo` sınıflarına bağımlı değildir.

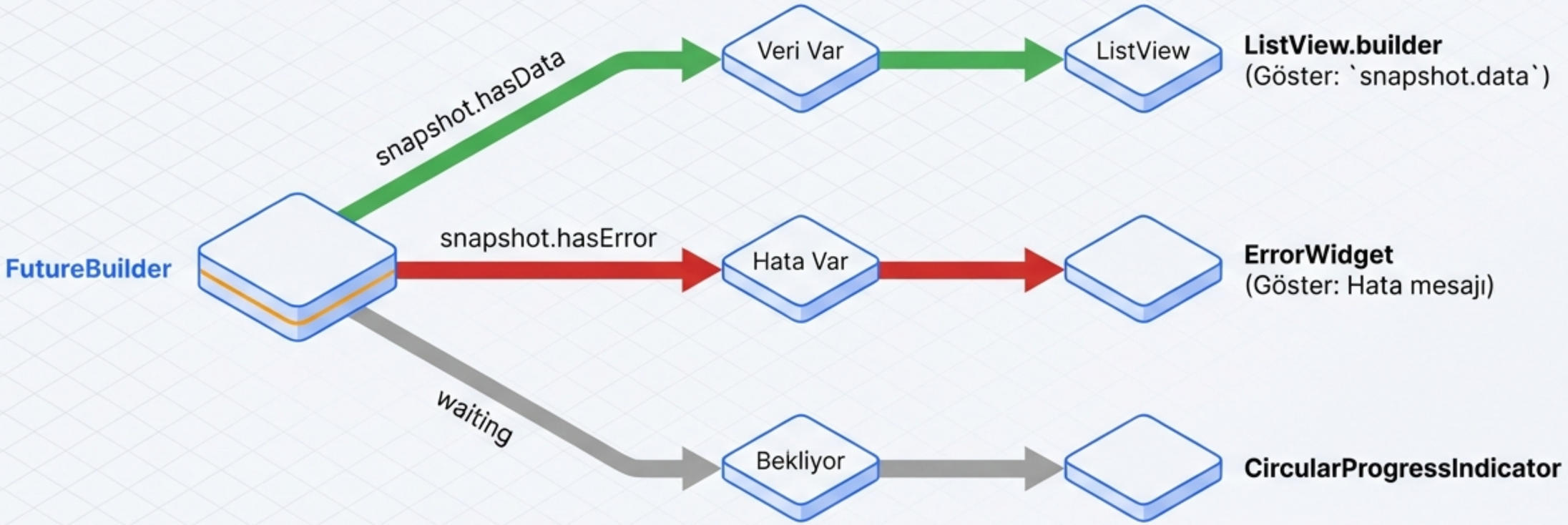
4. Katman: UI Entegrasyonu

Kullanım Kolaylığı

```
late final Future<List<Todo>> todos;

@override
void initState() {
  super.initState();

  // Endpoint ve Parser'ı ver, gerisini mimari halletsin.
  todos = RequestREST(endpoint: "/todos")
      .executeGet<List<Todo>>(const TodoParser());
}
```

Geliştirici sadece ‘nereye gideceğini’ (endpoint) ve ‘nasıl çevireceğini’ (parser) söyler.

Veriyi Göstermek: FutureBuilder




```dart
if (snapshot.hasData) {
  final data = snapshot.data ?? []; // Null Safety
  return ListView.builder(...);
}
```

Özet ve Kazanımlar

Model
(Veri Yapısı)

Parser
(Çeviri)

Client
(Taşıma)

UI
(Sunum)

Test Edilebilirlik

Her parça izole edilmiştir, bağımsız test edilebilir.

Okunabilirlik

Generic yapılar ve Mixin'ler kod tekrarını önler.

Esneklik

Yeni endpoint eklemek mevcut kodu bozmaz.

Bir sonraki projenizde spagetti kod yerine, katmanlı ve SOLID prensiplerine uygun bu mimariyi deneyin.