Flutter Uygulamalarında Ölçeklenebilir Vektör Grafikleri (SVG)

Modern mobil geliştirmede çözünürlükten bağımsız arayüz tasarımı ve performans optimizasyonu.

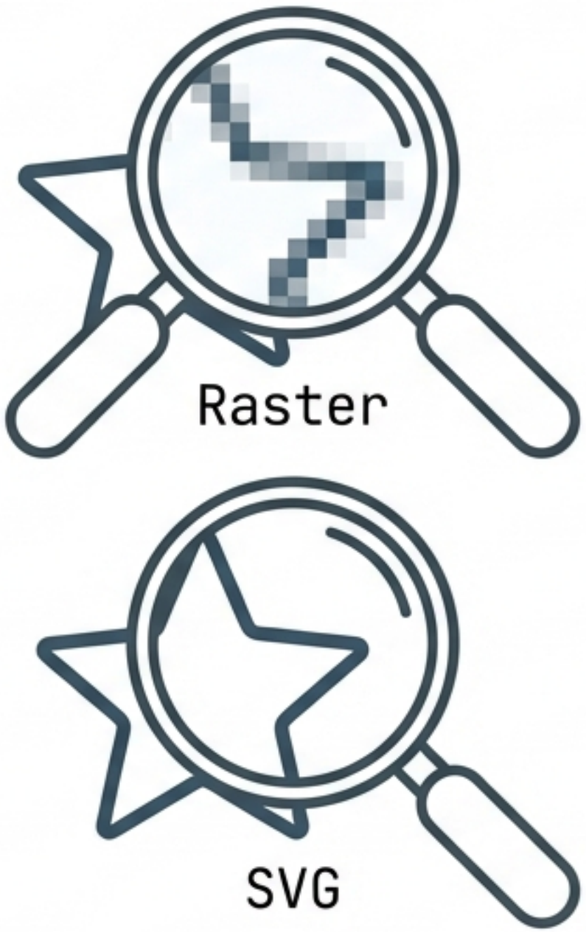
Based on 'Flutter Complete Reference' - Chapter 18

Çözünürlük Bağımsızlığının Gücü

SVG formatı, piksel matrisleri yerine şekiller, yollar (paths) ve dolgular üzerine kuruludur. Bu sayede donanım özelliklerinden tamamen bağımsızdır.

- **Tek Dosya, Sonsuz Ölçek:** 6 inçlik bir telefon veya 12 inçlik bir tablet fark etmeksizin, görüntü kalitesi asla bozulmaz.

- **İş Akışı Kolaylığı:** Her ekran yoğunluğu (1x, 2x, 3x) için **n** sayıda farklı varyant üretmek ve sürdürmek yerine tek bir dosya yeterlidir.



Büyük Karşılaşma: SVG vs. PNG

Mükemmel dosya formatı yoktur, sadece senaryoya uygun format vardır. Seçim yaparken aşağıdaki teknik kriterleri göz önünde bulundurun.

Kriter	SVG (Vektör)	PNG (Raster)
Ölçeklenebilirlik	Mükemmel, kayıpsız büyütme	Pikselleşme ve bulanıklık riski
Dosya Boyutu	Genellikle çok küçük (KB)	Çözünürlük arttıkça boyut artar (MB)
Karmaşıklık / Detay	Basit geometrik çizimler için ideal	Fotoğraf ve detaylı gölgeler için ideal
İşlemci Yükü (Decoding)	Yüksek. Motorun matematiksel hesaplama yapması gerekir.	Düşük. Basit renk haritası, hızlı decode edilir.

Sonuç: Şeffaflık ve detay gerektiren karmaşık görseller için PNG; ikonlar, logolar ve geometrik arkaplanlar için SVG kullanın.

Karar Rehberi: Hangi Formatı Seçmeliyim?

Profesyonel İpucu

Görüntü çok detaylıysa, SVG gösterimi grafik motorunun decode etmesi için pahalı (expensive) bir işlem olabilir. Performans testleri yapmadan karar vermeyin.

Araç Seti ve Hazırlık

Paket: flutter_svg

Dan Field tarafından geliştirilen bu paket, Flutter ekosisteminin standart SVG render çözümüdür.

```yaml
dependencies:
  flutter:
    sdk: flutter
  flutter_svg: ^2.0.0
```

Kritik Adım: Optimizasyon

Kodu yazmadan önce varlıklarınızı (assets) optimize etmelisiniz. “vecta.io/nano” gibi araçlar kullanarak dosya boyutunu küçültün ve gereksiz metadata’yı temizleyin.

Temel Uygulama: Asset Yükleme

```dart
SvgPicture.asset(
  'myassets/question.svg',
  width: 120,
  fit: BoxFit.contain,
  placeholderBuilder: (BuildContext context) =>
    const CircularProgressIndicator(),
)
```

- **width/height:** Görüntü boyutlarını belirler.
- **fit:** Görüntünün kutu içine nasıl yerleşeceğini kontrol eder.
- **placeholderBuilder:** Görüntü işlenirken (decode) gösterilecek geçici widget.

Zorluk Seviyesi Artıyor: Ağdan Yükleme

Standart 'SvgPicture.network' metodu mevcuttur ve tıpkı asset yükleme gibi çalışır.

```
SvgPicture.network('https://site.com/image.svg')
```

Kritik Kısıtlama

Bu sınıf, bağlantı hatalarını (connection errors) yönetmek için yerleşik bir yol sunmaz.

- Kullanıcı internet bağlantısını kaybederse ne olur?
- Sunucu 404 veya 500 hatası verirse ne olur?

Profesyonel bir uygulamada, ağ hatalarını yakalamak ve kullanıcıyı bilgilendirmek zorundayız. Bunun için daha sağlam bir mimariye ihtiyacımız var.

Çözüm Mimarisi: Robust Networking

Basit bir URL yüklemesi yerine, HTTP isteği ile görüntü işlemeyi birbirinden ayırıyoruz.

HTTP İstemcisi (Dio)

İstek Başlat

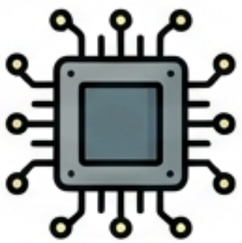
Download

Raw Bytes (`List<int>`)

State Management

`FutureBuilder` (Loading/Error)

Decoding



`SvgPicture.memory`

Paint

Vektör Ekrana Çizimi

Adım 1: Networking Katmanı (Dio)

```dart
class Downloader {
  static final _opt = BaseOptions(
    baseUrl: 'https://api.myapp.com/',
    responseType: ResponseType.bytes // ÖNEMLİ
  );
  static final _client = Dio(_opt);

  Future<List<int>> start(String url) async {
    final request = await _client.get<List<int>>(url);
    return request.data!;
  }
}
```

Neden Bytes?

SVG'yi string olarak değil, 'ResponseType.bytes' kullanarak ham bayt (raw bytes) olarak indiriyoruz. Bu, bellekten okuma işlemi için gereklidir.

Static Client

Aynı istemci örneğini (instance) yeniden kullanarak kaynak tüketimini optimize ediyoruz.

Adım 2: UI Mantığı ve State Yönetimi

```dart
class _SvgFromWebState extends State<SvgFromWeb> {
  late final Future<List<int>> svgImage;

  @override
  void initState() {
    super.initState();
    // İstek sadece widget oluşurken bir kez başlatılır
    svgImage = downloader.start();
  }
  // build metodu buraya gelecek...
}
```

Gereksiz İstekleri Önleme

FutureBuilder build metodu her tetiklendiğinde Future'ı yeniden çalıştırabilir. Bunu önlemek için:

1. StatefulWIdget kullanın.
2. Future işlemini 'initState' içinde bir değişkene atayın.
3. FutureBuilder'a bu değişkeni verin.

Adım 3: Builder ile Durum Kontrolü

```dart
if (snapshot.hasError) {
  return const ErrorWidget(); // Hata Durumu
}
if (snapshot.hasData) {
  return SvgPicture.memory(   // Başarı Durumu
    Uint8List.fromList(snapshot.data!),
    placeholderBuilder: (_) => const DecoderLoader(),
  );
}
return const NetworkLoader(); // Yükleme Durumu
```

UX Notu: Kullanıcı Durumları

NetworkLoader
(İndiriliyor)

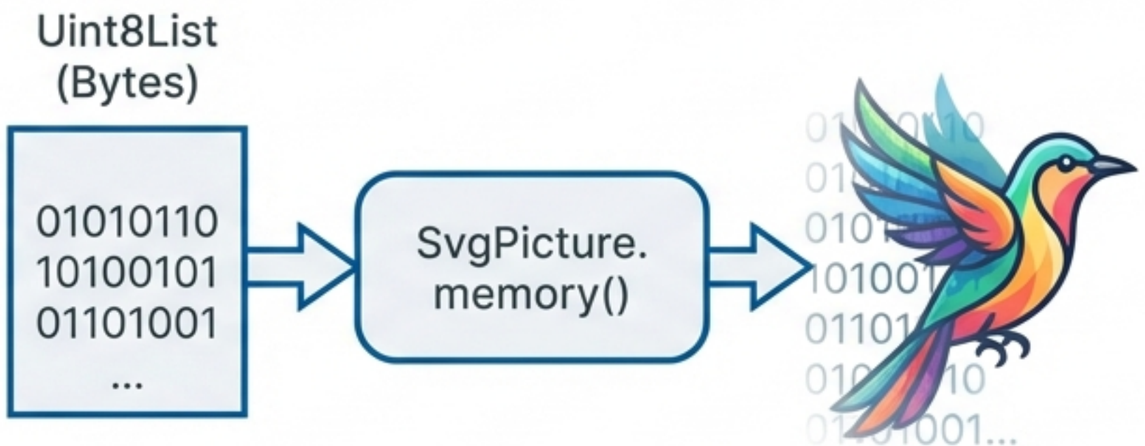
DecoderLoader
(İşleniyor)

SvgPicture
(Görünüyor)

Bellekten Ekrana: Decoding

Dio ile indirdiğimiz ‘List<int>’ verisini, ‘SvgPicture.memory()’ kurucusu (constructor) ile görsele dönüştürüyoruz.

Bu yöntem, geliştiriciye indirilen veri üzerinde tam kontrol sağlar (önbellekleme, doğrulama, manipülasyon).



Profesyonel SVG Yönetimi: Özet

- **Doğru Formatı Seç**
  Karmaşık detaylar için PNG, ölçeklenebilirlik için SVG kullanın.
- **Optimize Et**
  Kodlamadan önce Nano araçlarıyla dosya boyutunu küçültün.
- **Hata Yönetimi**
  Ağ işlemleri için basit yöntemler yerine Dio + FutureBuilder mimarisini kullanın.
- **Kullanıcı Deneyimi**
  Her aşamada (indirme ve decode) kullanıcıya geri bildirim (loader) verin.

SVG sadece bir dosya formatı değil, modern ve duyarlı (responsive) arayüzlerin temel taşıdır.