‟O Flutter Kavramları, Yapılar ve Araçlar

Ù 1. API Türleri

• RESTful API → JSON tabanlı, en yaygın kullanılan.
• GraphQL → İhtiyaç kadar veri çekme, esnek sorgular.
• gRPC → Google'ın geliştirdiği hızlı protokol.
• WebSockets → Gerçek zamanlı iletişim (chat, bildirim).
• Firebase APIs → Realtime Database, Firestore, Auth, Cloud Functions.

Ù 2. HTTP & API Paketleri

• http → Basit HTTP istekleri için.
• dio → Gelişmiş (interceptor, logging, cancel, timeout).
• chopper → Retrofit tarzı client.
• retrofit.dart → Annotation tabanlı, otomatik client.

Ù 3. State Management (Durum Yönetimi)

• Provider → En temel, Flutter ekibi öneriyor.
• Riverpod → Provider'ın gelişmiş sürümü, daha güvenli.
• Bloc / Cubit → Event → State mantığı, büyük projeler için.
• GetX → Minimalist, reactive programlama.
• MobX → Observer tabanlı state yönetimi.
• Redux → Global state, büyük ekiplerde tercih.

Ù 4. Mimari Yaklaşımlar

• MVC (Model-View-Controller)
• MVVM (Model-View-ViewModel)
• Clean Architecture
• Entity, Use Case (Domain), Repository (Data), UI (Presentation)
• Layered Architecture (Katmanlı yapı) → Data, Domain, Presentation Layer

Ù 5. Katmanlar ve Dosya Yapısı

• lib/
• core/ → sabitler, yardımcı fonksiyonlar
• data/ → modeller, repositories
• domain/ → entities, usecases

• presentation/ → pages, widgets, providersÙ 6. Veri Tabanları
• SQLite (sqflite paketi)
• Hive (hafif NoSQL, local)
• ObjectBox (hızlı NoSQL DB)
• Drift (Moor) (reactive SQLite ORM)
• Firebase Firestore

Ù 7. Network & Cache

• SharedPreferences → basit local key-value.
• Hive → offline cache.
• dio + interceptors → API caching.

Ù 8. Asenkron Programlama

• Future → tek seferlik async iş.
• Stream → sürekli veri akışı (chat, müzik).
• Isolate → ayrı thread'de ağır işlem.

Ù 9. UI / State Araçları

• Material Design Widgets
• Cupertino Widgets
• Animations → lottie, rive.
• Responsive UI → flutter_screenutil, responsive_framework.
• Theme Management → dynamic theming, dark mode.

Ù 10. Testing & QA

• Unit Test → iş mantığı testleri.
• Widget Test → tek widget testleri.
• Integration Test → uçtan uca test.
• mockito / mocktail → test mockları.

Ù 11. Dependency Injection

• get_it → Service Locator.
• riverpod → DI + state yönetimi.
• injectable → otomatik DI.

Ù 12. Build & Deployment• flavors → dev / staging / prod ortamları.

• fastlane → otomatik build & publish.

• code signing → Android keystore, iOS provisioning.

Ù 13. CI/CD

• GitHub Actions
• Bitrise
• Codemagic
• GitLab CI

Ù 14. Diğer Önemli Konular

• Internationalization (i18n) → çoklu dil desteği.
• Accessibility (a11y) → engelli kullanıcılar için erişilebilirlik.
• Error Handling & Logging → sentry, firebase crashlytics.
• Analytics → firebase analytics, amplitude.
• Push Notifications → firebase messaging, one signal.
• App Security → secure storage, obfuscation.

Ù 15. Performans ve Optimizasyon

• Const Widgets, RepaintBoundary
• ListView.builder, Sliver Widgets
• Flutter DevTools → memory & performance
• Image & Asset optimizasyonu → cached_network_image, svg

Ù 16. Paket Yönetimi ve Monorepo

• Melos → Monorepo yönetimi
• Pubspec.yaml yönetimi → dependency_overrides
• Custom Packages → ortak kodların paketlenmesi

Ù 17. Navigation & Routing

• Navigator 1.0 (classic push/pop)
• Navigator 2.0 / Router API
• go_router, auto_route

Ù 18. Native Entegrasyon

• Platform Channels, pigeon
• MethodChannel / EventChannelÙ 19. UI/UX Gelişmiş Konular
• CustomPainter, Shader/Skia
• Hero Animations / Page Transitions

- 3D & Game Development → flame engine

Ù 20. State + Side Effects Yönetimi

- Cubit/Bloc side effects → BlocListener
- riverpod async/family providers
- redux middlewares
- event_bus

Ù 21. Uygulama Mimarisi İleri Konular

- DDD (Domain Driven Design)
- Microfrontends / Modular Flutter
- Feature-first file structure
- SOLID Principles

Ù 22. Araçlar ve Debugging

- Flutter DevTools → Memory, Performance, Network
- dart analyze, lint kuralları
- VSCode / Android Studio eklentileri

Ù 23. Yeni Nesil Flutter Konuları

- Flutter Web, Flutter Desktop
- Impeller (yeni rendering engine)
- FFI (Dart ↔ Native C/C++ kodu)
- WASM (WebAssembly desteği)

1 RESTful API ve SQL Karşılaştırması

RESTful API, veritabanı işlemlerini (okuma, yazma, güncelleme, silme) web üzerinden JSON tabanlı olarak yapmamızı sağlar. Aşağıdaki tablo, SQL sorguları ile RESTful API isteklerinin karşılaştırmasını göstermektedir.

SQL Komutu	RESTful API	Açıklama
SELECT	GET	Veri okuma, listeleme veya tek kaydı getirme
INSERT	POST	Yeni veri ekleme
UPDATE	PUT / PATCH	Var olan veriyi güncelleme
DELETE	DELETE	Veri silme

Kısa Açıklamalar

GET → Verileri sunucudan çekmek için kullanılır (SQL'deki SELECT gibi).

POST → Sunucuya yeni veri eklemek için kullanılır (SQL'deki INSERT gibi).

PUT / PATCH → Mevcut veriyi güncellemek için kullanılır (SQL'deki UPDATE gibi).

DELETE → Veriyi silmek için kullanılır (SQL'deki DELETE gibi).

Bu yapı sayesinde Flutter gibi istemciler, RESTful API üzerinden arka plandaki veritabanı ile iletişim kurar. Kullanıcı uygulamada işlem yaptığında, bu işlem API aracılığıyla SQL sorgusuna dönüşür ve veritabanında uygulanır.

RESTful API Nerelerde Kullanılır?

Firebase Firestore: Flutter uygulamalarında Firestore'dan veri almak için REST API veya SDK kullanılır.

Kendi Backend Sunucun: Node.js, Django, .NET gibi teknolojilerle REST API yazılabilir. GET/POST/PUT/DELETE ile kullanıcı, ürün, sipariş işlemleri yapılır.

Üçüncü Parti API'ler: Hava durumu (Weather API), ödeme (Stripe), sosyal medya (Twitter), yapay zeka (OpenAI) gibi servisler REST API sunar.

SQL ile İlişki: Uygulama direkt SQL çalıştırmaz. Backend SQL'i çalıştırır, sonucu JSON olarak REST API ile gönderir.

2 Flutter HTTP ve Dio Paketleri

2. HTTP & API Paketleri

Flutter'da API istekleri için en çok kullanılan iki paket vardır: http ve dio.

1. HTTP Paketi

Flutter'da 'http' paketi, basit HTTP istekleri (GET, POST, PUT, DELETE) yapmak için kullanılır. Örneğin bir web servisten veri almak veya sunucuya veri göndermek için tercih edilir.

Avantajları

- Kullanımı çok kolaydır.
- Küçük projeler için idealdir.
- Ekstra ayar gerektirmez.

Sınırlılıkları

- Gelişmiş özellikleri yoktur.
- Büyük projelerde yönetim zor olabilir.
- Loglama, interceptor gibi özellikler bulunmaz.

Gerçek Hayat Benzetmesi

http paketini, sadece telefonla arama yapabilen basit bir tuşlu telefona benzetebiliriz. Temel ihtiyacı karşılar ama gelişmiş özellikler sunmaz.

Örnek Kullanım (GET)

Bir API'den kullanıcı listesi çekmek için basit bir GET isteği:

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

void fetchUsers() async {
  final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/users'));

  if (response.statusCode == 200) {
    print(response.body);
  } else {
    print('Hata: ${response.statusCode}');
  }
}
```

1. http Paketi

- Basit HTTP istemcisidir.

- Küçük projeler için uygundur.

- Fazladan özellikleri yoktur, hafif ve kullanımı kolaydır.

2. Dio Paketi

Dio, http paketine göre daha gelişmiş bir ağ (networking) paketidir. Interceptor, timeout, iptal etme, loglama gibi özelliklere sahiptir.

Avantajları

• Loglama desteği vardır.
• Timeout ayarlanabilir.
• İstekleri iptal edebilirsiniz.
• Interceptor ile her isteğe otomatik token ekleme yapılabilir.

Sınırlılıkları

• Küçük projeler için gereksiz olabilir.
• Öğrenmesi http'ye göre biraz daha karmaşıktır.

Gerçek Hayat Benzetmesi

Dio paketini, akıllı telefona benzetebiliriz. Hem arama yapar hem de birçok ekstra özellik sunar.

Örnek Kullanım (GET)

Bir API'den kullanıcı listesi çekmek için Dio kullanımı:

```
import 'package:dio/dio.dart';

void fetchUsers() async {
 var dio = Dio();
 final response = await dio.get('https://jsonplaceholder.typicode.com/users');
 print(response.data);
}
```

2. dio Paketi

- http paketine göre daha gelişmiştir.
- Orta ve büyük ölçekli projelerde tercih edilir.
- Sağladığı özellikler: Interceptor, logging, timeout, istek iptal etme, dosya yükleme.

3. Karşılaştırma (http vs dio)

Özellik	http	dio
Kullanım Kolaylığı	Çok kolay	Biraz daha karmaşık
Interceptor	✗ Yok	✓ Var
Loglama	✗ Yok	✓ Var
Timeout	✗ Yok	✓ Var
İptal Etme	✗ Yok	✓ Var
Küçük Projeler	✓ İdeal	✗ Fazla karmaşık olabilir
Büyük Projeler	✗ Yönetimi zor	✓ Çok uygun

4. Hangi Durumda Hangisi?

- Küçük, basit uygulamalarda → http
- Büyük, kompleks uygulamalarda → dio

2.5 Mimari olarak MVVM

Model, viev model, viev olarak kullanılan bu mimariyi bilmek en mantıklısı

3 Flutter Katmanlı Yapı (Layered Architecture)

1. core/

• Amaç: Projede tüm katmanlar tarafından ortak kullanılan şeyler.
• İçerik: constants.dart, utils.dart, theme, renkler, stringler.

2. data/

• Amaç: Veri kaynaklarını yönetmek (API, DB, local).
• Alt klasörler:
- models/: API veya DB verisini temsil eden sınıflar
- repositories/: Veri kaynaklarına erişim sağlayan sınıflar

3. domain/

• Amaç: İş kuralları ve uygulama mantığı.
• Alt klasörler:
- entities/: Saf veri nesneleri
- usecases/: İş kurallarını kapsayan fonksiyonlar

4. presentation/

• Amaç: UI ve kullanıcı etkileşimi.
• Alt klasörler:
- pages/: Ekranlar
- widgets/: Tekrar kullanılabilir UI bileşenleri
- providers/: State yönetimi (Riverpod, Bloc, Provider)

Özet Mantık

• core/: Ortak araçlar, sabitler
• data/: Veri kaynakları ve repository
• domain/: İş mantığı ve saf veri
• presentation/: UI ve state management

Avantajları:
• Kod modüler ve okunabilir
• Test yazmak kolay
• Büyük projelerde karmaşa azalır

4 Flutter State Management (Durum Yönetimi)

Flutter'da State Management, yani durum yönetimi; uygulamanın ekranlarında gösterilen verilerin kullanıcı etkileşimlerine göre güncellenmesini sağlar. En çok kullanılan yöntemler: Provider, Riverpod, Bloc/Cubit'tir.

1. Provider

- En temel state management çözümüdür.
- Flutter ekibi tarafından önerilmektedir.
- Küçük ve orta ölçekli projelerde yeterlidir.
- Ancak proje büyüdükçe karmaşıklaşabilir.

2. Riverpod

- Provider'ın geliştirilmiş sürümüdür.
- Context bağımlılığı yoktur, daha güvenlidir.
- Test yazmak daha kolaydır.
- Hem küçük hem büyük projelerde kullanılabilir.
- Provider'ın zayıf yönlerini kapatır ve daha modern bir yapıya sahiptir.
- Uzun vadede öğrenilmesi en mantıklı çözümlerden biridir.

Riverpod Kullanım Türleri

1. StateProvider → Basit değer tutmak için (sayı, boolean vs.)
2. StateNotifierProvider → Daha karmaşık state yönetimi için (liste, obje vs.)
3. FutureProvider → Asenkron veriler için (örn: API çağrısı).
4. StreamProvider → Sürekli akan veriler için (örn: Firebase veritabanı).

3. Bloc / Cubit

- Event → State mantığı ile çalışır.
- Büyük, kurumsal projeler için uygundur.
- Öğrenme eğrisi diğerlerine göre daha diktir.
- Çok disiplinli ve katı bir yapı sunar.

Hangi State Management Öğrenilmeli?

- Provider öğrenmesi kolaydır, başlangıç için kullanılabilir.
- Riverpod uzun vadeli projeler için en mantıklı seçimdir.
- Bloc ise büyük ve kurumsal projeler için tercih edilir.

Riverpod öğrenmek, hem başlangıç hem ileri seviye için seni uzun süre götürür.

5 Flutter Veri Tabanları: Hive, SQLite ve Firebase Firestore

Flutter uygulamalarında veri saklamak için farklı veritabanı çözümleri vardır. Ancak çoğu senaryo için Hive, SQLite ve Firebase Firestore yeterlidir. Bu üç teknoloji, offline ve online ihtiyaçları karşılamak için en yaygın kullanılan çözümlerdir.

1. Hive

• Tür: NoSQL (Key-Value)
• Kullanım: Offline, hızlı depolama
• Performans: Çok hızlı (Dart native)
• Veri Yapısı: Basit obje/JSON
• Avantaj: Küçük-orta veriler, ayarlar ve cache için çok uygun.
• Öğrenme: Kolay

2. SQLite

• Tür: SQL (İlişkisel)
• Kullanım: İlişkisel veriler, tablolar
• Performans: Orta (C köprüsü üzerinden)
• Veri Yapısı: Tablo & sütun ilişkileri
• Avantaj: Karmaşık ilişkisel veriler için ideal.
• Öğrenme: Orta seviye

3. Firebase Firestore

• Tür: NoSQL (Cloud tabanlı)
• Kullanım: Online + senkronizasyon
• Performans: Orta (internet bağlantısına bağlı)
• Veri Yapısı: JSON döküman koleksiyonları
• Avantaj: Kullanıcı verisi ve cloud tabanlı uygulamalar için çok güçlü.
• Öğrenme: Kolay-Orta

Karşılaştırma Tablosu

Özellik / DB	Hive	SQLite	Firebase Firestore
Tür	NoSQL (Key-Value)	SQL (İlişkisel)	NoSQL (Cloud)
Kullanım Alanı	Offline, hızlı	İlişkisel veriler,	Online +

	depolama	tablolar	senkronizasyon
Performans	Çok hızlı	Orta	Orta
Veri Yapısı	Obje/JSON	Tablo & sütun	JSON koleksiyonları
Offline Desteği	✓	✓	✓ (cache)
Online Senkronizasyon	✗	✗	✓
Öğrenme Kolaylığı	Kolay	Orta	Kolay-Orta
Popülerlik	Yüksek	Yüksek	Çok yüksek
Mantıklı Olduğu Yer	Ayarlar, cache	Karmaşık ilişkisel veriler	Cloud tabanlı uygulamalar

Sonuç

• Hive → Offline & hızlı.
• SQLite → İlişkisel veri için ideal.
• Firebase Firestore → Online & senkronize uygulamalar için güçlü.

Bu üç teknolojiyi öğrenmek Flutter projelerinde %90 ihtiyacı karşılayacaktır.

6 Flutter Network & Cache

Flutter uygulamalarında veri çekme ve önbellekleme kullanıcı deneyimi için çok önemlidir. Bu alanda öne çıkan çözümler: SharedPreferences, Hive ve Dio + Interceptors.

1. SharedPreferences

• Basit key-value saklama çözümü.
• Küçük veriler (tema, dil, giriş flag) için ideal.
• Önbellek için uygun değil.

2. Hive

• Offline NoSQL veritabanı.
• API'den gelen verileri cihazda saklamak için ideal.
• Offline cache ve hızlı veri erişimi sağlar.

3. Dio + Interceptors

• Gelişmiş HTTP paketi.
• API cevaplarını cache edebilir.
• Kullanıcıya hızlı veri sunmak ve offline/yarı-offline senaryolarda veri göstermek için kullanılır.

Örnek Uygulama: Haber / Blog Uygulaması
1. İlk açılış: API'den veri alır, cihazda cache'e kaydeder.
2. Sonraki açılış: Cache hızlı gösterim, ardından API'den güncelleme.
3. İnternet yoksa: Cache'den son veri gösterilir.

Özet Karşılaştırma

Teknoloji	Kullanım Alanı	Öne Çıkan Nokta
SharedPreferences	Küçük ayarlar, flag	Hafif ve basit
Hive	Offline cache, veri saklama	Hızlı, offline destek
Dio + Interceptors	API caching, online cache	Hızlı tepki, offline/yarı-offline destek

Sonuç

• Offline veri için: Hive
• Online veri ve API caching için: Dio + Interceptors
• Küçük ayarlar ve flag için: SharedPreferences

Bu kombinasyon, uygulamanın hızlı, stabil ve kullanıcı dostu çalışmasını sağlar.

7 Flutter Testing & QA - Pratik Bilgi

Flutter uygulamalarında testler genellikle üç ana türde yapılır: Unit Test, Widget Test ve Integration Test. Mockito veya Mocktail ile API/DB simülasyonu yapılabilir.

1. Unit Test → İş mantığını test eder. Fonksiyon ve methodlar için hızlı ve basit.
2. Widget Test → Tek bir widget'ın doğru çalışıp çalışmadığını kontrol eder.
3. Integration Test → Uçtan uca uygulama senaryolarını test eder.
4. Mockito / Mocktail → Gerçek API veya veritabanı olmadan test yapmayı sağlar.

Pratik Not

Küçük veya solo projelerde test yazmak genellikle gereksizdir. Hatalar manuel olarak bulunup düzeltilir. Test kodları yazmak ve sonra silmek zaman kaybı olabilir. Büyük projelerde veya ekip işlerinde testler kritik ve önerilir.

8 Flutter Asenkron Programlama

Flutter uygulamalarında UI'nin donmaması ve işlemlerin arka planda çalışabilmesi için asenkron programlama çok önemlidir. Burada üç temel kavram öne çıkar: Future, Stream ve Isolate.

1. Future

• Tek seferlik async işlem için kullanılır.
• Örnek: API çağrısı, dosya okuma, küçük hesaplama.
• İşlem tamamlanınca sonuç döner.

2. Stream

• Sürekli veri akışı için kullanılır.
• Örnek: Chat mesajları, müzik akışı, sensör verileri.
• Stream birden fazla değer üretebilir ve listen ile takip edilir.

3. Isolate

• Ağır veya uzun süren işlemleri ayrı thread'de çalıştırmak için kullanılır.
• UI thread'ini bloke etmez.
• Senin senaryonda (not defteri, şifre uygulaması) gerek yok.

Özet

Kavram	Kullanım Alanı	Notlar
Future	Tek seferlik async işler	Çoğu senaryo için yeterli
Stream	Sürekli veri akışı	Chat, sensör, müzik gibi durumlar
Isolate	Ağır/uzun işlem	Sadece çok ağır işlemlerde gerekli, senin uygulamada gerek yok

Sonuç

Senin uygulaman için Future ve Stream tamamen yeterlidir.
Isolate öğrenmek şimdilik gerekli değil.

9 Flutter UI / State Araçları

Flutter uygulamalarında görsellik ve kullanıcı deneyimi için kullanılan temel UI ve state araçları:

1. Material Design Widgets → Google'ın tasarım sistemi. Android benzeri modern tasarımlar. Örnek: Scaffold, AppBar, ElevatedButton.
2. Cupertino Widgets → Apple iOS stiline uygun widget'lar. Örnek: CupertinoButton, CupertinoNavigationBar.
3. Animations (Lottie, Rive) → UI'yi canlı ve interaktif yapmak için animasyonlar. Lottie: JSON animasyonlar, Rive: interaktif animasyonlar.
4. Responsive UI (flutter_screenutil, responsive_framework) → Farklı ekran boyutlarına uyum sağlar. Font, padding ve boyutları otomatik ayarlar.
5. Theme Management (Dynamic Theming, Dark Mode) → Uygulama temalarını yönetir. Örnek: ThemeData.light(), ThemeData.dark().

Özet Tablo

Araç	Kullanım Alanı	Avantaj
Material Design Widgets	Android/modern tasarım	Hızlı ve uyumlu UI
Cupertino Widgets	iOS tasarımı	iOS kullanıcı deneyimi
Animations (Lottie, Rive)	UI animasyonları	Etkileşimli ve çekici UI
Responsive UI	Farklı ekran boyutları	Tüm cihazlarda uyum
Theme Management	Tema & renk yönetimi	Dark mode ve dynamic theming

10 Flutter Dependency Injection - get_it vs Riverpod

get_it – Service Locator Pattern

Tanım: Flutter'da popüler Service Locator implementasyonu. Singleton'lar ve servisleri kaydetme/erişme imkanı sağlar.

Temel Özellikler:

- Basit Kurulum: Minimal konfigürasyon gerektirir
- Singleton Desteği: Tek instance garanti eder
- Lazy Loading: İhtiyaç duyulduğunda instance oluşturur
- Factory Support: Her çağrıda yeni instance

Avantajlar:

- Öğrenmesi kolay
- Minimal kod yazımı
- Hızlı kurulum
- Performanslı

Dezavantajlar:

- Global state
- Test zorluğu
- Compile-time güvenlik yok
- State management yok

Riverpod – DI + State Management

Tanım: Hem dependency injection hem de state management için kapsamlı çözüm. Provider pattern'ın geliştirilmiş versiyonu.

Temel Özellikler:

- Compile-time Safety: Derleme zamanında hata yakalama
- BuildContext Bağımsızlığı: Her yerden erişim
- Auto-disposal: Otomatik temizleme
- Testing Friendly: Kolay mock'lama

Avantajlar:

- Type-safe
- Reaktif programlama
- Otomatik dispose
- Test dostu
- State management dahil

Dezavantajlar:

- Öğrenme eğrisi

- Daha fazla boilerplate
- Konsept karmaşıklığı

Karşılaştırma Tablosu

Özellik	get_it	Riverpod
Kurulum Kolaylığı	Çok Kolay	Orta
Öğrenme Eğrisi	Düşük	Orta-Yüksek
Type Safety	Runtime	Compile-time
State Management	✗	✓
Testing	Zor	Kolay
Performance	Yüksek	Yüksek
Auto Disposal	✗	✓

Hangi Yaklaşımı Seçmeli?

get_it Seçin:
- Basit projelerde
- Hızlı prototipleme
- Minimal kurulum istediğinizde
- Sadece DI ihtiyacı varsa

Riverpod Seçin:
- Büyük/karmaşık projelerde
- State management gerekirse
- Type safety önceliyse
- Test coverage önemliyse

Sonuç:
- get_it: Basitlik ve hız arıyorsanız ideal. Küçük-orta projeler için mükemmel.
- Riverpod: Büyük projeler ve kapsamlı state management için güçlü çözüm. Öğrenme eğrisi var ama uzun vadede değer katıyor.
- Tavsiye: Yeni başlayanlar get_it ile başlayıp, proje büyüdükçe Riverpod'a geçiş yapabilir.

11 Flutter Build & Deployment - Pratik Bilgi

Flavors

- Farklı ortamlar için (development, staging, production) ayrı konfigürasyonlar oluşturmayı sağlar.
- Örnek: Test sunucusu ve gerçek sunucu için ayrı API adresleri tanımlanabilir.

Kullanımı:

- Ortamlar arasında hızlı geçiş sağlar.
- Büyük ve profesyonel projelerde kullanılır, küçük projelerde genellikle gerek yok.

Fastlane

- Flutter uygulamasının build, test ve mağazaya gönderme süreçlerini otomatikleştiren araç.

Kullanımı:

- Tek komutla Android/iOS için APK/IPA oluşturabilir ve yayınlayabilir.
- Profesyonel ekiplerde ve sık güncelleme gereken projelerde tercih edilir.
- Küçük veya tek kişilik projelerde manuel build yeterlidir.

Code Signing

- Android için Keystore, iOS için Provisioning Profile.
- Uygulamanın mağazalarda imzalanmış ve güvenli olmasını sağlar.

Kullanımı:

- Zorunlu, çünkü mağazalar imzasız uygulamaları kabul etmez.
- Android'de basit keystore dosyası ile çözülür.
- iOS'de Apple Developer hesabı ve provisioning profile gerekir.

Özet

- **Code Signing:** Kesinlikle gerekli.
- **Flavors:** Büyük projelerde tavsiye edilir, küçük projelerde opsiyonel.
- **Fastlane:** Profesyonel ekiplerde ve sık güncellemede faydalı, küçük projelerde opsiyonel.

12. Diğer Önemli Konular

Internationalization (i18n)

- Uygulamanın birden fazla dili desteklemesini sağlar.
- Kullanıcı diline göre içerik ve metinleri otomatik olarak gösterir.

Neden Önemli?

- Global kullanıcı kitlesi hedefleyen projelerde kullanıcı deneyimi için şarttır.

Accessibility (a11y)

- Engelli kullanıcılar için uygulamanın erişilebilir olmasını sağlar.
- Örnek: Ekran okuyucularla uyum, kontrast, font büyüklüğü, dokunmatik alanlar.

Neden Önemli?

- Kullanıcı tabanını genişletir ve yasal standartlara uyum sağlar.

Error Handling & Logging

- Uygulama hatalarını yakalama, raporlama ve kaydetme işlemleri.
- Araçlar: Sentry, Firebase Crashlytics.

Neden Önemli?

- Hataları erken tespit edip çözmek, uygulama kalitesini artırır.

Analytics

- Kullanıcı davranışlarını ve uygulama performansını ölçme.
- Araçlar: Firebase Analytics, Amplitude.

Neden Önemli?

- Hangi özelliklerin popüler olduğunu görmeyi sağlar.
- Gelecek güncellemeler ve geliştirmeler için veri sunar.

Push Notifications

- Kullanıcılara uygulama içi veya dışı bildirim gönderme.
- Araçlar: Firebase Messaging, OneSignal.

Neden Önemli?

- Kullanıcı etkileşimini ve uygulama bağlılığını artırır.

App Security

- Uygulama verilerini ve kullanıcı bilgilerini koruma.
- Araçlar: Secure Storage, Obfuscation.

Neden Önemli?

- Kullanıcı verisi güvenliği, mağaza onayları ve güvenilirlik için şarttır.

13 Paket Yönetimi ve Monorepo

Melos

- Monorepo yönetimi için kullanılan bir araç.
- Tek bir repo içinde birden fazla paket ve proje yönetimini kolaylaştırır.

Neden Önemli?

- Büyük projelerde ve ekip çalışmalarında, ortak paketleri ve bağımlılıkları yönetmek için kullanılır.
- Tek bir komutla tüm paketleri build, test ve publish edebilirsin.

Pubspec.yaml Yönetimi

- Flutter projelerinin bağımlılıklarını ve paket sürümlerini tanımlayan dosya.
- `dependency_overrides` ile belirli paket sürümlerini zorlayabilirsin.

Neden Önemli?

- Farklı paket sürümleri nedeniyle oluşabilecek çatışmaları önler.
- Projeyi tutarlı ve stabil hâle getirir.

Custom Packages

- Kendi ortak kodlarını paket hâline getirip diğer projelerde kullanabilirsin.
- Örnek: Ortak widget'lar, servisler veya utility fonksiyonlar.

Neden Önemli?

- Kod tekrarını azaltır ve ekip içinde standardizasyon sağlar.
- Büyük projelerde kod bakımını ve güncellemeyi kolaylaştırır.

14 UI/UX Gelişmiş Konular

CustomPainter, Shader / Skia

- Flutter'da kendi özel çizimlerinizi yapmanızı sağlar.
- `CustomPainter` ile canvas üzerinde şekiller, grafikler ve animasyonlar oluşturabilirsiniz.
- Skia, Flutter'ın kullandığı yüksek performanslı 2D çizim motorudur.

Avantajları:

- Standart widget'lar ile mümkün olmayan özel tasarımları yapabilme imkânı.
- UI'ye tamamen özgün görünüm kazandırır.

Hero Animations / Page Transitions

- Sayfalar arası geçişlerde nesnelerin animasyonlu şekilde taşınmasını sağlar.
- `Hero` widget'ı ile bir widget bir sayfadan diğerine büyüyerek veya küçülerek geçebilir.

Avantajları:

- Kullanıcı deneyimini zenginleştirir.
- Uygulamayı daha akıcı ve modern gösterir.

3D & Game Development → Flame Engine

- Flutter ile basit 3D veya oyun geliştirme imkânı sağlar.
- `Flame Engine`, 2D oyun geliştirmek için popüler Flutter oyun motorudur.

Avantajları:

- Flutter sadece UI değil, basit oyun ve interaktif uygulamalar için de kullanılabilir.
- Animasyon ve performans optimizasyonları ile görsel açıdan güçlü projeler yapılabilir.

- **Özet:**
- Bu konular daha çok ileri seviye UI/UX ve oyun geliştirme projeleri için gereklidir.
- Standart uygulamalar için çoğu zaman gerek yoktur; ama görsellik ve kullanıcı deneyimini ön planda tutan projelerde fark yaratır.