T.C.

KIRKLARELİ ÜNİVERSİTESİ

TEKNİK BİLİMLER MESLEK YÜKSEKOKULU

BİLGİSAYAR PROGRAMCILIĞI BÖLÜMÜ

MOBİL PROGRAMLAMA DERSİ PROJE ÖDEVİ

PLATFORMA ÖZEL (NATIVE) KOD ENTEGRASYONU

ÖĞRENCİ ADI SOYADI NUMARASI:

GÜLSENA YILMAM 1247008005

ÖĞRETİM ELEMANI:

DR. ÖĞR. ÜYESİ NADİR SUBAŞI

KIRKLARELİ

12/2025

İÇİNDEKİLER

TABLOLAR

KISALTMALAR

SDK: Software Development Kit

API: Application Programming Interface

VM: Virtual Machine

iOS: iPhone Operating System

UI: User Interface

JSON: JavaScript Object Notation

POS: Point of Sale

1. GİRİŞ

Mobil uygulama geliştirme ekosistemi son yıllarda hızla büyümüş ve kullanıcıların günlük yaşamının vazgeçilmez bir parçası hâline gelmiştir. Bu gelişimle birlikte, hem Android hem de iOS üzerinde çalışabilen uygulamalar geliştirmek yazılım sektöründe önemli bir gereklilik hâlini almıştır. Google tarafından geliştirilen Flutter framework'ü, "tek kod tabanı ile çoklu platform" (write once, run anywhere) yaklaşımıyla bu ihtiyaca güçlü bir çözüm sunmaktadır.

Çapraz platform (cross-platform) teknolojilerinin en büyük zorluğu, cihazın donanım özelliklerine — kamera, sensörler, Bluetooth, pil durumu vb. — erişim sırasında ortaya çıkar. Güvenlik ve mimari gereklilikler nedeniyle işletim sistemleri bu erişimi yalnızca kendi yerel (native) dilleri üzerinden sağlanan API'lerle mümkün kılar.

Çalışmanın Amacı ve Kapsamı

Bu raporun amacı, Flutter'da cihaz donanımına erişmek için gerekli olan **Native Kod Entegrasyonu**nu ve bu sürecin temel bileşeni olan **MethodChannel mimarisini** ayrıntılı şekilde incelemektir. Çalışmada Flutter'ın mimarisi, native kod ihtiyacı, MethodChannel'ın veri akışı, serileştirme (serialization) süreçleri ve örnek bir sensör uygulaması üzerinden kod düzeyinde açıklamalar ele alınmaktadır.

Mobil Uygulama Geliştirme Yaklaşımları

Günümüzde mobil uygulama geliştirme iki temel kategoriye ayrılır:

Native (Yerel) Geliştirme: Android için Kotlin/Java, iOS için Swift/Obj-C kullanılarak yapılan geliştirme türüdür. Donanım erişimi ve performans açısından en yüksek verim elde edilir ancak her platform için ayrı kod yazılması bakım maliyetini artırır.

Cross-Platform Geliştirme: Flutter veya React Native gibi araçlarla tek bir dil kullanarak her iki platform için çıktı alınmasını sağlar. Geliştirme süresi avantajlıdır fakat belirli donanım işlemlerinde native entegrasyon gerektirir.

Bu çalışma, Flutter'ın donanım erişim kısıtlarını native kod entegrasyonu ile nasıl aştığını teknik açıdan açıklamaktadır.

Proje Konusu ve Amacı

Flutter, tek bir kod tabanı ile Android ve IOS uygulamaları geliştirmeye olanak sağlayan modern bir framework'tür. Flutter'ın güçlü yönlerinden biri, ihtiyaç duyulduğunda platforma özgü API'lere erişebilmesidir. Bu sayede, Flutter paketlerinde bulunmayan donanım veya işletim sistemi özellikleri de uygulamalara entegre edilebilir. Bu entegrasyon, **Platform Channels** yapısının bir parçası olan **MethodChannel** üzerinden gerçekleşmektedir.

Bu çalışma, MethodChannel kullanılarak Android ve iOS tarafındaki donanım bileşenlerine — pil seviyesi, ortam ışığı sensörü, ivmeölçer, manyetik sensör vb. — nasıl erişilebileceğini açıklamayı ve bu süreci örnek bir uygulama üzerinden ayrıntılı olarak incelemeyi amaçlamaktadır.

Platform Channel Nedir?

Flutter uygulamaları Dart diliyle yazılır; ancak donanım seviyesindeki birçok bilgiye yalnızca Android'de **Kotlin/Java** ve iOS'te **Swift/Objective-C** ile erişilebilir. Platform Channel, Dart ile native kod arasında veri alışverişi sağlayan bir köprü görevi görür.

Platform Channel mimarisi üç bileşenden oluşur:

1. Dart tarafındaki MethodChannel tanımı
2. Android (Kotlin) tarafındaki MethodCallHandler
3. iOS (Swift) tarafındaki MethodCallHandler

Bu yapı sayesinde Flutter ve işletim sistemi arasında güvenli, kontrollü ve çift yönlü iletişim sağlanır.

2. TEORİK ÇERÇEVE VE TEMEL KAVRAMLAR

MethodChannel yapısının anlaşılabilmesi için öncelikle Flutter'ın çalışma mantığının ve Native kod kavramının irdelenmesi gerekmektedir.

Flutter Mimarisi ve Çalışma Yapısı

Flutter, klasik çapraz platform araçlarından (örneğin Webview tabanlı araçlar) farklı olarak, kendi rendering (çizim) motoruna sahiptir. **Skia** adı verilen bir grafik motoru üzerinde çalışan Flutter, arayüz elemanlarını (widget) işletim sisteminin native bileşenlerini kullanmadan, doğrudan ekrana çizer.

Uygulama mantığı Dart dili ile yazılır ve Dart Sanal Makinesi (Dart VM) üzerinde çalışır. Ancak Dart VM, işletim sisteminin çekirdeği ile doğrudan konuşamaz. Bir sandbox (kum havuzu) içinde çalışır. Bu izolasyon, güvenlik ve taşınabilirlik sağlasa da donanım erişimini kısıtlar.

Native (Yerel) Kod Nedir?

Native kod, mobil cihazın işletim sistemi üreticisi (Google veya Apple) tarafından sağlanan SDK (Software Development Kit) ve diller kullanılarak yazılan koddur.

Android Tarafı: Linux çekirdeği üzerine kurulu yapıda, donanım sürücüleriyle Java veya Kotlin dili üzerinden iletişim kurulur.

iOS Tarafı: Unix tabanlı yapıda, Swift veya Objective-C dili ile Cocoa Touch kütüphaneleri kullanılarak donanıma erişilir.

Platforma Özgü Entegrasyon İhtiyacı

Flutter uygulamasının standart paketleri (package) birçok genel ihtiyacı karşılasa da, bazı spesifik durumlarda native entegrasyon zorunlu hale gelir. Bu durumlar şunlardır:

1. Cihaz sensörlerinden (İvmeölçer, Jiroskop, Işık Sensörü) ham veri alma.

2. Arka plan servisleri (Background Services) ve işletim sistemi tetikleyicileri.
3. Özel donanım aksesuarları ile (Örn: POS cihazı, Termal Yazıcı) iletişim kurma.
4. Henüz Flutter paketi olarak yayınlanmamış yeni işletim sistemi özelliklerini kullanma.

Flutter Engine ve Embedder Katmanı

Flutter mimarisinin temelinde Framework, Engine ve Embedder olmak üzere üç katman bulunur. Framework katmanı Dart ile yazılan UI bileşenlerini yönetirken, Engine C++ tabanlı çalışarak çizim, animasyon ve Dart kodunun çalıştırılmasını sağlar. Embedder ise uygulamanın Android ve iOS üzerinde çalışmasını mümkün kılan köprü katmandır. Native kodla iletişimi sağlayan MethodChannel mekanizması da bu Embedder yapısı üzerinden işletilir. Bu katmanlar Flutter'ın hem yüksek performanslı hem de platform bağımsız çalışmasının temelini oluşturur.

Platform Kanal Alternatifleri: EventChannel ve BasicMessageChannel

Flutter'ın native platformlarla iletişim kurmak için sunduğu temel yapı MethodChannel olmakla birlikte, farklı veri akışı türleri için iki alternatif kanal daha bulunmaktadır: **EventChannel** ve **BasicMessageChannel**. Bu kanallar, MethodChannel'ın tamamlayıcısı niteliğinde olup belirli senaryolarda daha uygun çözümler sunmaktadır.

EventChannel: Flutter ile native taraf arasında sürekli veri akışı (stream) gerektiğinde kullanılan iletişim yapısıdır. Özellikle sensör verileri, GPS konum bilgisi gibi gerçek zamanlı ve tekrar eden veriler için tasarlanmıştır. Native taraf Stream benzeri sürekli veri üretir, Flutter tarafı bunu dinler. Bu nedenle ışık sensörü gibi değişken donanımsal bilgiler için EventChannel'ın MethodChannel'a göre daha uygun olduğu durumlar bulunmaktadır.

BasicMessageChannel: Dart ile native platform arasında genel amaçlı veri alışverişi için kullanılır ve string/tabanlı mesajlaşma mantığıyla çalışır. JSON formatında yapılandırılmış verilerin gönderilmesi için uygundur.

3. METHODCHANNEL MİMARİSİ

Flutter ve Native platformlar (Android/iOS) birbirinden farklı bellek alanlarında ve farklı süreçlerde (process) çalışır. Bu iki yapının birbiriyle konuşabilmesi için bir "İletişim Protokolü"ne ihtiyaç vardır. Flutter ekosisteminde bu protokol **MethodChannel** olarak adlandırılır.

MethodChannel Nedir?

MethodChannel, adından da anlaşılacağı üzere, belirli bir "yöntem"in (method) karşı tarafta çalıştırılması için kullanılan bir "kanal"dır (channel). İstemci-Sunucu (Client-Host) mimarisine benzer bir yapı sergiler.

Client (İstemci): Genellikle Flutter tarafıdır. İsteği gönderir.

Host (Sunucu): Android veya iOS tarafıdır. İsteği dinler, işler ve cevap döner.

Çalışma Prensibi ve Veri Akışı

İletişim süreci şu adımlardan oluşur:

1. **Kanal Tanımlama:** Hem Flutter hem de Native tarafta aynı isme sahip (örneğin "com.ornek.sensor/isik") bir kanal oluşturulur.
2. **Çağrı (Invocation):** Flutter, invokeMethod('getLightLevel') komutu ile kanala mesaj bırakır.
3. **Yönlendirme:** Flutter Engine, bu mesajı platform kanalına yönlendirir.
4. **Yakalanma (Handling):** Native taraftaki MethodCallHandler, gelen mesajı yakalar. Metod ismini kontrol eder.
5. **İşlem:** Native kod sensör verisini okur.
6. **Dönüş:** Sonuç, aynı kanal üzerinden Flutter'a geri gönderilir.

Asenkron İletişim Yapısı

Mobil uygulamalarda arayüzün donmaması (UI Blocking) hayati önem taşır. Sensör okuma veya veritabanı sorgusu gibi işlemler zaman alabilir. Bu nedenle MethodChannel iletişimi **asenkron** (eşzamansız) olarak tasarlanmıştır. Flutter tarafında bu işlem Future (Gelecek) nesnesi ile yönetilir ve await anahtar kelimesi ile sonucun gelmesi beklenirken arayüz çalışmaya devam eder.

Hata Yönetimi (Error Handling)

MethodChannel kullanılırken native tarafta karşılaşılan hatalar Flutter'a belirli istisna türleriyle iletilir. Native kod bir işlemi yapamadığında `notImplemented()` döndürerek metodun tanımlı olmadığını bildirir. Beklenmeyen durumlarda ise hata kodu, mesajı ve isteğe bağlı ayrıntılar `error()` yanıtıyla Flutter tarafına aktarılır. Bu yapı, platformlar arası iletişimde hata kaynağının net şekilde belirlenmesine olanak tanır.

Veri Tipi Eşleşmeleri ve Serileştirme (Serialization)

Dart dilindeki veri tipleri ile Native dillerdeki veri tipleri aynı değildir. Veriler kanaldan geçerken Binary (İkili) formata çevrilir. Bu işleme "Serialization", verinin tekrar okunabilir hale gelmesine "Deserialization" denir. Flutter StandardMessageCodec sınıfını kullanarak bu dönüşümü otomatik yapar.

Tablo 1:

Dart (Flutter)	Android (Kotlin/Java)	iOS (Swift)
null	null	nil
bool	java.lang.Boolean	NSNumber (bool)
int	java.lang.Integer	NSNumber (int)
double	java.lang.Double	NSNumber (double)
String	java.lang.String	NSString
List	java.util.List	NSArray
Map	java.util.HashMap	NSDictionary

4. UYGULAMA VE KOD İNCELEMESİ: ORTAM IŞIĞI SENSÖRÜ

Bu bölümde, teorik olarak anlatılan yapının kod üzerinde nasıl uygulandığı gösterilecektir. Senaryo olarak; bir Flutter uygulamasının, cihazın ortam ışığı sensöründen "Lüks (Lux)" değerini anlık olarak çekmesi işlemi ele alınmıştır.

Senaryo Tanımı

Uygulama arayüzünde bir buton ve bir metin alanı bulunacaktır. Butona basıldığında Native tarafa istek atılacak, Native taraf donanım sensörünü okuyup değeri döndürecektir.

Flutter (Dart) Tarafı Kodlaması

Flutter tarafında öncelikle bir MethodChannel nesnesi tanımlanmalı ve invokeMethod fonksiyonu try-catch bloğu içinde (hataları yakalamak için) çağrılmalıdır.

Dart

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart'; // MethodChannel için gerekli paket

class LightSensorPage extends StatefulWidget {
  @override
  _LightSensorPageState createState() => _LightSensorPageState();
}

class _LightSensorPageState extends State<LightSensorPage> {
  // 1. Kanalın Tanımlanması (Benzersiz bir isim olmalı)
  static const platform = MethodChannel('com.ornek.uygulama/sensor');

  String _sensorDegeri = 'Veri bekleniyor...';

  // 2. Native Koda İstek Gönderme Fonksiyonu
  Future<void> _getLightLevel() async {
```

```
    String sensorSonucu;
    try {
      // 'getLightSensorData' metodu native tarafta aranacak
      final int result = await platform.invokeMethod('getLightSensorData');
      sensorSonucu = 'Ortam Işığı: $result lux';
    } on PlatformException catch (e) {
      sensorSonucu = "Sensör hatası: '${e.message}'.";
    }

    // Arayüzü güncelleme
    setState(() {
      _sensorDegeri = sensorSonucu;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Native Entegrasyon Örneği')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(_sensorDegeri, style: TextStyle(fontSize: 20)),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: _getLightLevel,
              child: Text('Sensörü Oku'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Android (Kotlin) Tarafı Kodlaması

Flutter projesinin android/app/src/main/kotlin/.../MainActivity.kt dosyası açılarak aşağıdaki kodlar eklenir. Burada MethodCallHandler ile Flutter'dan gelen çağrı dinlenir.

Kotlin

```
import android.content.Context
import android.hardware.Sensor
import android.hardware.SensorEvent
import android.hardware.SensorEventListener
import android.hardware.SensorManager
import io.flutter.embedding.android.FlutterActivity
import io.flutter.embedding.engine.FlutterEngine
import io.flutter.plugin.common.MethodChannel
```

```
class MainActivity: FlutterActivity() {
    private val CHANNEL = "com.ornek.uygulama/sensor"
    private var sensorManager: SensorManager? = null
    private var lightSensor: Sensor? = null

    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)

        // Sensör Yöneticisini Başlatma
        sensorManager = getSystemService(Context.SENSOR_SERVICE) as SensorManager
        lightSensor = sensorManager?.getDefaultSensor(Sensor.TYPE_LIGHT)

        // MethodChannel Tanımlama
        MethodChannel(flutterEngine.dartExecutor.binaryMessenger,
CHANNEL).setMethodCallHandler {
            call, result ->
            if (call.method == "getLightSensorData") {
                // Işık değerini oku (Basitleştirilmiş senaryo)
                val luxValue = getSensorData()
                if (luxValue != -1) {
                    result.success(luxValue) // Başarılı sonuç döndür
                } else {
                    result.error("UNAVAILABLE", "Sensör bulunamadı.", null)
                }
            } else {
                result.notImplemented()
            }
        }
    }

    // Sensör verisini okuyan yardımcı fonksiyon (Örnek amaçlıdır)
    private fun getSensorData(): Int {
        // Gerçek uygulamada burada SensorEventListener kullanılır.
        // Bu örnekte simüle edilmiş bir değer veya son okunan değer döner.
        return 120;
    }
}
```

iOS (Swift) Tarafı Kodlaması

iOS tarafında ios/Runner/AppDelegate.swift dosyası düzenlenir.

Swift

```
import UIKit
import Flutter

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
```

```
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey:
Any]?
  ) -> Bool {

    let controller : FlutterViewController = window?.rootViewController as!
FlutterViewController
    let sensorChannel = FlutterMethodChannel(name: "com.ornek.uygulama/sensor",
                                             binaryMessenger: controller.binaryMessenger)

    sensorChannel.setMethodCallHandler({
      (call: FlutterMethodCall, result: @escaping FlutterResult) -> Void in

      if call.method == "getLightSensorData" {
        // iOS'te statik veri örneği
        result(Int(120))
      } else {
        result(FlutterMethodNotImplemented)
      }
    })

    return super.application(application, didFinishLaunchingWithOptions: launchOptions)
  }
}
```

iOS'ta Ortam Işığı Sensörü ile İlgili Sınırlamalar

iOS işletim sistemi, ortam ışığı sensörüne doğrudan erişim sağlayan bir API sunmamaktadır. Bu nedenle cihazdan gerçek sensör verisi almak mümkün değildir. Geliştiriciler genelikle ekran parlaklığı gibi dolaylı göstergeler üzerinden tahmini değer üretir. Bu durum iOS ve Android platformları arasında donanım erişimi açısından önemli bir fark oluşturmaktadır.

Gerçek Sensörlerde EventListener Kullanımı (Android)

Bu raporda örnek uygulama basitliği açısından statik bir değer üzerinden ışık sensörü verisi döndürülmüş olsa da, gerçek bir Android cihazdan sensör verisi okuma işlemi SensorEventListener üzerinden gerçekleştirilir. Android işletim sistemi, donanım sensörlerinden gelen verileri olay (event) tetiklemeleri şeklinde iletir. Bu nedenle, ortam ışığı sensörü gibi sürekli değişen verilerin gerçek zamanlı okunması için bu mekanizmanın kullanılması gerekmektedir.

Android'de gerçek sensör akışı şu şekilde işlenir:

1. **SensorManager** aracılığıyla istenilen sensör (ör. TYPE_LIGHT) kaydedilir.
2. Uygulama, **SensorEventListener** arayüzünü implement eder.
3. Veriler sensörde değiştikçe onSensorChanged() metodu tetiklenir.

4. Elde edilen değer MethodChannel veya EventChannel üzerinden Flutter'a iletilebilir.

5. ANALİZ VE DEĞERLENDİRME

Yapılan uygulama ve teorik incelemeler sonucunda, Flutter MethodChannel yapısının sağladığı avantajlar ve dikkat edilmesi gereken noktalar aşağıda analiz edilmiştir.

Performans Değerlendirmesi

MethodChannel, verileri serileştirerek (byte array'e çevirerek) taşıdığı için çok büyük verilerin (örneğin yüksek çözünürlüklü bir videonun her karesinin) bu kanal üzerinden taşınması performans kaybına yol açabilir (Latency). Ancak, sensör verileri, metinler veya yapılandırma ayarları gibi küçük veri paketlerinde gecikme süresi milisaniyeler mertebesindedir ve kullanıcı tarafından hissedilmez.

Avantajlar ve Dezavantajlar

Tam Donanım Erişimi: Flutter'ın kısıtlamalarına takılmadan cihazın tüm gücü kullanılabilir.

Mevcut Kütüphanelerin Kullanımı: Halihazırda Java veya Swift ile yazılmış eski kütüphaneler Flutter projesine dahil edilebilir.

Platform Bağımlılığı: Native kod yazıldığında, "tek kod tabanı" prensibinden kısmen uzaklaşılır. Android ve iOS için ayrı ayrı kod bakımı yapılması gerekir.

Hata Yönetimi: Hataların iki farklı tarafta (Dart ve Native) ayrı ayrı ele alınması gerekir.

MethodChannel – EventChannel – FFI Kısa Karşılaştırması

Flutter ile native kod arasındaki iletişimde üç farklı yaklaşım kullanılabilir. MethodChannel tek seferlik çağrılar ve basit veri transferleri için uygundur. Sürekli akış gerektiren sensör, konum veya anlık veri takibi gibi işlemlerde EventChannel tercih edilir. Daha düşük seviyeli kütüphanelere erişim gerektiğinde ise FFI (Foreign Function Interface) daha yüksek performans sağlar fakat uygulama karmaşıklığını artırır. Bu çalışma kapsamında tekil veri okuma ön planda olduğu için MethodChannel en uygun tercih olmuştur.

Güvenlik Boyutu

Flutter ile native kod arasında kurulan MethodChannel yapısı, platformlar arası veri alışverişine izin verdiği için belirli güvenlik risklerini beraberinde getirir. Dart kodunun bir sandbox içinde çalışması güvenlik açısından avantaj sağlasa da, native taraf ile iletişim kurulduğunda işletim sisteminin kaynaklarına doğrudan erişildiği için dikkat edilmesi gereken noktalar bulunmaktadır.

Bu çalışma kapsamında güvenlik açısından değerlendirilen başlıca noktalar şunlardır:

Yetkilendirilmiş Erişim: Native tarafta yalnızca belirli method çağrılarına izin verilmiş, kanal üzerinden gelen istekler method ismi bazında doğrulanmıştır. Böylece dışarıdan hatalı veya yetkisiz bir çağrı yapılması engellenmiştir.

Platform İzin Kontrolleri: Sensör verilerine erişim Android ve iOS işletim sistemlerinde belirli izinlere tabi olduğundan, uygulamanın çalışması için gerekli izinler kontrol edilmiştir.

Hatalı Mesaj Yönetimi: Flutter tarafında try-catch mekanizması, Native tarafta ise MethodCallHandler üzerinden hata yönetimi uygulanarak beklenmeyen veri girişlerinin uygulamayı bozmasının önüne geçilmiştir.

Kanal Adı Güvenliği: MethodChannel için kullanılan kanal adının benzersiz ve tahmin edilmesi zor olması, farklı eklentiler veya üçüncü taraf bileşenlerin yanlışlıkla aynı kanalla iletişim kurmasını engellemiştir.

6. SONUÇ

Bu raporda, mobil programlama dersi kapsamında Flutter framework'ünün native kod entegrasyonu incelenmiştir. Flutter, UI geliştirme konusunda güçlü araçlar sunsa da, donanım tabanlı işlemlerde native dillerin desteğine ihtiyaç duymaktadır.

MethodChannel mimarisi, Dart VM ile native işletim sistemi arasında güvenli, asenkron ve standartlara dayalı bir köprü kurarak bu ihtiyacı karşılamaktadır. Geliştirilen örnek "Ortam Işığı Sensörü" uygulaması, bir komutun Flutter arayüzünden başlayıp, native katmanda işlenip tekrar arayüze dönmesi sürecini başarılı bir şekilde simüle etmiştir. Sonuç olarak, modern mobil uygulama geliştirmede, sadece framework bilgisi değil, platformların native yapılarına (Android/iOS) hakim olmak da nitelikli uygulamalar geliştirmek için elzemdir.

7. KAYNAKÇA

1. Flutter Official Documentation - "Platform Channels". https://flutter.dev/docs/development/platform-integration/platform-channels

2. Android Developers Guide - "Sensors Overview". https://developer.android.com/guide/topics/sensors

3. Apple Developer Documentation - "Core Motion". https://developer.apple.com/documentation/coremotion