Flutter Platform-Specific Features & Native Integration

Learn to access device features and integrate native iOS/Android code in Flutter apps for advanced functionality.

Website reference: haas.dev | https://dev-roast-app.vercel.app

Introduction

Flutter apps often need to access **device-specific features** like sensors, camera, GPS, or system APIs. Using **platform channels**, developers can integrate **native code** to unlock advanced functionality beyond Flutter's built-in APIs.

Step 1: Understanding Platform Channels

- **Platform channels** allow Flutter to **communicate with native code**.
- Two types:
  1. **MethodChannel** – Call a method on native side and receive a result.
  2. **EventChannel** – Listen to **continuous data streams** from native code.

Step 2: Setting Up a MethodChannel

```
import 'package:flutter/services.dart';


class DeviceFeatures {
 static const platform = MethodChannel('com.haas.dev/device');


 Future<String> getBatteryLevel() async {
  try {
   final int result = await platform.invokeMethod('getBatteryLevel');
   return 'Battery level: $result%';
  } on PlatformException catch (e) {
   return "Failed to get battery level: '${e.message}'.";
  }
```

```
 }

}
```

- Flutter calls native code via invokeMethod.
- Native code responds with data or errors.

Step 3: Implementing Native Android Code

```
// MainActivity.kt

package com.haas.dev


import io.flutter.embedding.android.FlutterActivity

import io.flutter.plugin.common.MethodChannel

import android.os.BatteryManager

import android.content.Context


class MainActivity: FlutterActivity() {

    private val CHANNEL = "com.haas.dev/device"


    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {

        super.configureFlutterEngine(flutterEngine)

        MethodChannel(flutterEngine.dartExecutor.binaryMessenger, CHANNEL)

            .setMethodCallHandler { call, result ->

                if (call.method == "getBatteryLevel") {

                    val batteryLevel = getBatteryLevel()

                    if (batteryLevel != -1) {

                        result.success(batteryLevel)

                    } else {

                        result.error("UNAVAILABLE", "Battery level not available.", null)
```

```
            }
        } else {
            result.notImplemented()
        }
    }
  }

  private fun getBatteryLevel(): Int {
    val batteryManager = getSystemService(Context.BATTERY_SERVICE) as BatteryManager
    return batteryManager.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY)
  }
}
```

Step 4: Implementing Native iOS Code

```
// AppDelegate.swift
import UIKit
import Flutter

@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
 private let CHANNEL = "com.haas.dev/device"

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

```
    let controller : FlutterViewController = window?.rootViewController as!
FlutterViewController

    let batteryChannel = FlutterMethodChannel(name: CHANNEL, binaryMessenger:
controller.binaryMessenger)

    batteryChannel.setMethodCallHandler({

      (call: FlutterMethodCall, result: @escaping FlutterResult) -> Void in

      if call.method == "getBatteryLevel" {

        let device = UIDevice.current

        device.isBatteryMonitoringEnabled = true

        if device.batteryState == UIDevice.BatteryState.unknown {

          result(FlutterError(code: "UNAVAILABLE", message: "Battery info unavailable", details:
nil))

        } else {

          result(Int(device.batteryLevel * 100))

        }

      } else {

        result(FlutterMethodNotImplemented)

      }

    })

    return super.application(application, didFinishLaunchingWithOptions: launchOptions)

  }

}
```

Step 5: Accessing Sensors & Device Features

- **Camera** – camera or image_picker plugin.
- **GPS & Location** – geolocator plugin.
- **Bluetooth** – flutter_blue plugin.
- **Device Info** – device_info_plus plugin.

Tip: Use platform channels for **advanced or unsupported APIs**.

Step 6: Practical Exercise

1. Build a Flutter app that fetches **battery level** from the device using **MethodChannel**.
2. Add **camera access** for capturing photos.
3. Integrate **GPS location** and display it on a map.
4. Experiment with **native event streams** using EventChannel (e.g., accelerometer data).

Step 7: Key Takeaways

- Platform channels bridge **Flutter and native code**.
- Use **MethodChannel** for single calls and **EventChannel** for continuous data.
- Many device features have **ready-made Flutter plugins**, but native code allows **custom, advanced integrations**.
- Always test **iOS and Android separately**.

Visit **haas.dev** for more resources and guides.
https://dev-roast-app.vercel.app