

Complete Deployment, CI/CD & Production Readiness



This module covers the complete lifecycle of a Flutter application from development to production. We will use a functional **Weather App** (integrated with the Open-Meteo API) as our practical example to demonstrate deployment pipelines, security hardening, app signing, and store submission.

Part 1: The Foundation (Weather App Codebase)

Before deploying, we need a working application. This simple Weather App fetches data from the open web, making it ideal for testing permissions and network security in release builds.

Dependencies ([pubspec.yaml](#)):

```
None
dependencies:

  flutter:

    sdk: flutter

  http: ^1.2.0

  # Used for secure storage example in Security section

  flutter_secure_storage: ^9.0.0

  # Required for Crash Reporting

  firebase_core: ^2.27.0

  firebase_crashlytics: ^3.4.18


dev_dependencies:

  flutter_launcher_icons: ^0.13.1

  flutter_native_splash: ^2.4.0
```

lib/main.dart:

```
None
import 'dart:convert';

import 'package:flutter/material.dart';

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

import 'package:firebase_core/firebase_core.dart';

import 'package:firebase_crashlytics/firebase_crashlytics.dart';

import 'package:flutter_secure_storage/flutter_secure_storage.dart';

import 'firebase_options.dart'; // Generated by FlutterFire CLI

void main() async {

  WidgetsFlutterBinding.ensureInitialized();

  // Initialize Firebase (Critical for Production)

  // Run `flutterfire configure` to generate firebase_options.dart

  await Firebase.initializeApp(

    options: DefaultFirebaseOptions.currentPlatform,

  );

  // Pass all uncaught errors to Crashlytics

  FlutterError.onError =
  FirebaseCrashlytics.instance.recordFlutterFatalError;

  runApp(const WeatherApp());
}
```

```
class WeatherApp extends StatelessWidget {  
  const WeatherApp({super.key});  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'OpenMeteo Weather',  
      debugShowCheckedModeBanner: false,  
      theme: ThemeData(  
        primarySwatch: Colors.blue,  
        useMaterial3: true,  
      ),  
      home: const WeatherHomePage(),  
    );  
  }  
}  
  
class WeatherHomePage extends StatefulWidget {  
  const WeatherHomePage({super.key});  
  
  @override  
  State<WeatherHomePage> createState() => _WeatherHomePageState();  
}
```

```
class _WeatherHomePageState extends State<WeatherHomePage> {

  // Coordinates for London (Default)

  final double lat = 51.5074;

  final double lon = -0.1278;

  bool _isLoading = true;

  double? _temperature;

  String _weatherCode = "";

  String _errorMessage = "";

  @override

  void initState() {

    super.initState();

    _fetchWeather();

  }

  Future<void> _fetchWeather() async {

    setState(() {

      _isLoading = true;

      _errorMessage = "";

    });

    try {

      // Open-Meteo URL (No API Key needed)
```

```

        final url = Uri.parse(

'[https://api.open-meteo.com/v1/forecast?latitude=$lat&longitude=$lon
&current=temperature_2m,weather_code](https://api.open-meteo.com/v1/f
orecast?latitude=$lat&longitude=$lon&current=temperature_2m,weather_c
ode)');

        final response = await http.get(url);

        if (response.statusCode == 200) {

            final data = json.decode(response.body);

            final current = data['current'];

            setState(() {

                _temperature = current['temperature_2m'];

                _weatherCode = _mapWmoCode(current['weather_code']);

                _isLoading = false;

            });

        } else {

            throw Exception('Failed to load weather:
${response.statusCode}');

        }

    } catch (e) {

        setState(() {

            _errorMessage = e.toString();

            _isLoading = false;

```

```

    });

    // Log non-fatal errors to Crashlytics

    FirebaseCrashlytics.instance.recordError(e, null, reason:
'Weather Fetch Failed');

  }

}

String _mapWmoCode(int code) {

  if (code == 0) return "Clear Sky";

  if (code >= 1 && code <= 3) return "Partly Cloudy";

  if (code >= 45 && code <= 48) return "Fog";

  if (code >= 51 && code <= 67) return "Rain";

  if (code >= 71 && code <= 77) return "Snow";

  return "Unknown ($code)";

}

@override

Widget build(BuildContext context) {

  return Scaffold(

    appBar: AppBar(title: const Text("London Weather")),

    body: Center(

      child: _isLoading

        ? const CircularProgressIndicator()

        : _errorMessage.isNotEmpty

          ? Text("Error: $_errorMessage", style: const

```

```

TextStyle(color: Colors.red))

      : Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          const Icon(Icons.cloud, size: 80, color:
Colors.blue),

          const SizedBox(height: 20),

          Text(
            "${_temperature?.toStringAsFixed(1)}°C",
            style: const TextStyle(fontSize: 48,
fontWeight: FontWeight.bold),
          ),

          Text(
            _weatherCode,
            style: const TextStyle(fontSize: 24, color:
Colors.grey),
          ),

          const SizedBox(height: 30),

          ElevatedButton(
            onPressed: _fetchWeather,
            child: const Text("Refresh"),
          )
        ],
      ),

    ),
  ),
)

```

```
);  
}  
}
```

Part 2: The Deployment Pipeline

2.1 Preparation: Visual Identity

Professional apps require branded icons and splash screens.

1. Launcher Icons

Create `flutter_launcher_icons.yaml` in the project root:

```
None  
flutter_icons:  
  android: "launcher_icon"  
  ios: true  
  image_path: "assets/icon.png"  
  adaptive_icon_background: "#2196F3"  
  adaptive_icon_foreground: "assets/icon_foreground.png"  
  remove_alpha_ios: true # Critical for App Store validation
```

Run: `dart run flutter_launcher_icons`

2. Native Splash Screen

Create `flutter_native_splash.yaml` in the project root:

None

```
flutter_native_splash:  
  
  color: "#E3F2FD"  
  
  image: assets/splash_logo.png  
  
  android_12:  
  
    image: assets/splash_logo.png  
  
    icon_background_color: "#E3F2FD"
```

Run: `dart run flutter_native_splash:create`

2.2 Build Configuration: Flavors

Flavors allow you to install "Dev" and "Prod" versions of your app side-by-side.

Android Setup (android/app/build.gradle):

Inside the android { ... } block:

None

```
flavorDimensions "default"  
  
productFlavors {  
  
  dev {  
  
    dimension "default"  
  
    applicationIdSuffix ".dev"  
  
    resValue "string", "app_name", "Weather Dev"  
  
  }  
  
  prod {  
  
    dimension "default"  
  
    resValue "string", "app_name", "Weather"  
  
  }  
  
}
```

```
}
```

iOS Setup (Xcode):

1. **Schemes:** Create a new Scheme named `dev`.
2. **Configurations:** Duplicate "Release" to "Release-dev".
3. **Bundle Identifiers:**
 - Prod: `com.company.weather`
 - Dev: `com.company.weather.dev`
4. **Dynamic App Name (Critical Detail):**
 - Open `Info.plist`.
 - Change `Bundle display name` to `$(PRODUCT_NAME)`.
 - In Build Settings, search for **Product Name**. Set `Weather` for Release and `Weather Dev` for Release-dev. This ensures the app name on the iPhone home screen changes based on the flavor.

2.3 App Signing (Critical)

Apps must be digitally signed to verify their authenticity.

Android: Generating the Keystore

1. **Run Command:** Open a terminal in the project root.

None

```
keytool -genkey -v -keystore android/app/upload-keystore.jks -keyalg  
RSA -keysize 2048 -validity 10000 -alias upload
```

**KEEP THIS FILE SAFE. IF YOU LOSE IT, YOU CANNOT UPDATE
YOUR APP ON GOOGLE PLAY.**

2. Create **key.properties**: In android/key.properties:

None

```
storePassword=your_password_here  
keyPassword=your_password_here  
keyAlias=upload  
storeFile=upload-keystore.jks
```

3. Configure Gradle (android/app/build.gradle):

None

```
def keystoreProperties = new Properties()  
def keystorePropertiesFile = rootProject.file('../key.properties')  
if (keystorePropertiesFile.exists()) {  
    keystoreProperties.load(new  
    FileInputStream(keystorePropertiesFile))  
}  
  
android {  
    signingConfigs {  
        release {  
            keyAlias keystoreProperties['keyAlias']  
            keyPassword keystoreProperties['keyPassword']  
            storeFile file(keystoreProperties['storeFile'])  
            storePassword keystoreProperties['storePassword']  
        }  
    }  
}
```

```

    buildTypes {
        release {
            signingConfig signingConfigs.release
        }
    }
}
}

```

iOS: Certificates & Provisioning

1. **Apple Developer Account:** Required (\$99/year).
2. **Xcode Signing:**
 - Open **Runner.xcworkspace**.
 - Select the "Runner" target.
 - Go to **Signing & Capabilities**.
 - Check **"Automatically manage signing"**.
 - Select your Team. Xcode will generate the **Distribution Certificate** and **Provisioning Profile** automatically when you Archive.

2.4 CI/CD Fundamentals

Automate your build process using GitHub Actions.

Create **.github/workflows/main.yml**:

```

None
name: Flutter Release Build

on:

  push:

    branches: [ "main" ]

jobs:

  build:

    runs-on: ubuntu-latest

```

```
steps:

  - uses: actions/checkout@v3

  - uses: subosito/flutter-action@v2

    with:

      flutter-version: '3.19.0'

  - run: flutter pub get

  - run: flutter test

  - name: Build Android App Bundle

    run: flutter build appbundle --release --flavor prod

  - name: Upload Artifact

    uses: actions/upload-artifact@v3

    with:

      name: app-bundle

path:
build/app/outputs/bundle/prodRelease/app-prod-release.aab
```

Part 3: Production Readiness & Security

3.1 Performance Optimization

Before release, profile your app in "Profile Mode" (`flutter run --profile`).

1. **Rendering Performance:**
 - Open **Flutter DevTools > Performance**.
 - Enable "Performance Overlay".
 - Ensure bars stay below 16ms (60 FPS).
2. **Memory Leaks:**
 - Open **Memory** tab.

- Look for objects that persist after a screen is closed (e.g., Controllers not disposed).

3. Widget Rebuilds:

- Use `const` constructors everywhere possible.
- Use `RepaintBoundary` for complex animations.

3.2 Security Best Practices

Security is mandatory for 2024/2025 submission compliance.

A. Secure Storage (Runtime Secrets)

Never store sensitive data (JWT Tokens, User IDs) in Shared Preferences.

None

```
// Correct way using flutter_secure_storage

final storage = const FlutterSecureStorage();

await storage.write(key: 'auth_token', value: 'secret_jwt_123');
```

B. Build-Time Secrets (API Keys)

Do not commit API Keys to GitHub. Use `--dart-define`.

1. Run command: `flutter run --dart-define=API_KEY=12345`
2. Access in Dart:

None

```
const apiKey = String.fromEnvironment('API_KEY');
```

C. iOS Privacy Manifests (Mandatory)

Apple rejects apps accessing certain APIs without declaration.

1. In Xcode, Right-click Runner > New File > App Privacy.
2. Name it `PrivacyInfo.xcprivacy`.
3. Add `NSPrivacyAccessedAPITypes` for `UserDefaults` (used by most Flutter plugins):
 - Type: `NSPrivacyAccessedAPITypeUserDefaults`
 - Reason: `CA92.1` (App Functionality)

D. Code Obfuscation & ProGuard

Obfuscation hides your Dart code structure. ProGuard shrinks Android native code.

1. Fix ProGuard for JSON (android/app/proguard-rules.pro):

If R8 shrinks your code, JSON parsing will fail because class names change.

```
None
# Keep Flutter internal wrappers

-keep class io.flutter.app.** { *; }

-keep class io.flutter.plugin.** { *; }

# Keep your model classes if using reflection (optional but recommended)

-keep class com.example.weather.models.** { *; }
```

2. Obfuscated Build Command:

When building for release, use this command to scramble symbols:

```
None
flutter build appbundle --obfuscate --split-debug-info=./debug-info
```

E. Permissions

Android: Explicitly declare Internet permission in `android/app/src/main/AndroidManifest.xml` (release builds do not imply it like debug builds do):

```
None
<manifest ...>

    <uses-permission android:name="android.permission.INTERNET"/>

    <application ...>
```

Part 4: App Store Submission

4.1 Versioning

In `pubspec.yaml`:

None

```
version: 1.0.0+1
```

- **1.0.0:** Version Name. Visible to users.
- **+1:** Version Code. Must increment for every single upload.

4.2 Google Play Console Walkthrough

1. **Create Account:** Pay \$25 one-time fee.
2. **Create App:** Click "Create app" > Enter Name/Language > Select "Free".
3. **Store Listing:** Upload the icon (512x512), Feature Graphic (1024x500), and Screenshots.
4. **Privacy Policy:** You must provide a URL to a privacy policy (hosted on a simple website/GitHub page).
5. **Releases:**
 - Go to **Testing > Internal testing** (for your team).
 - "Create new release".
 - Upload the **.aab** file generated in Step 2.3.
 - Rollout.

4.3 Apple App Store Connect Walkthrough

1. **Archive Build:**
 - Open **Runner.xcworkspace** in Xcode.
 - Select "Any iOS Device".
 - **Product > Archive**.
2. **Upload:**
 - Once archived, click **Distribute App**.
 - Select **App Store Connect > Upload**.
 - Xcode validates and uploads the build.
3. **App Store Connect Website:**
 - Go to **My Apps > New App**.
 - **TestFlight Tab:** Your uploaded build appears here. Add testers.
 - **App Store Tab:** Select the build from TestFlight. Fill in keywords, support URL, and screenshots.

- Submit for Review.

Part 5: Maintenance & Configuration (Firebase)

5.1 Configuring Firebase (Critical Step)

For Crashlytics to work, you must link your app to Firebase servers.

1. Install Firebase CLI:

None

```
npm install -g firebase-tools  
  
firebase login  
  
dart pub global activate flutterfire_cli
```

2. Configure App: Run this command in your project root:

None

```
flutterfire configure
```

- Select your Firebase project.
 - Select platforms (Android, iOS).
 - This generates `lib/firebase_options.dart` and adds `google-services.json` (Android) / `GoogleService-Info.plist` (iOS).
3. **Verify:** Now the `Firebase.initializeApp()` in `main.dart` will work correctly.
Without this step, the app will crash immediately on launch.