Flutter Documentation Guide

Table of Contents

1. Introduction to Flutter

What is Flutter?

Flutter is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. Unlike other cross-platform frameworks, Flutter doesn't rely on web views or the host platform's native components. Instead, it uses its own rendering engine to draw UI components.

Key Features of Flutter

- Faster development cycle with hot reload
- Consistent UI across platforms
- Native-like performance
- Extensive customization possibilities
- Strong community support
- Backed by Google

2. Understanding Flutter Architecture

Key Components

1. **Dart Platform:** Flutter uses the Dart programming language, which offers features like JIT (Just-In-Time) compilation for development and AOT (Ahead-Of-Time) compilation for release builds.
2. **Flutter Engine:** Written in C/C++, the engine provides low-level rendering support using Skia, Google's graphics library.
3. **Foundation Library:** Core libraries written in Dart, providing basic classes and functions.
4. **Widget Layer:** Flutter's UI components built using a composition-based model.
5. **Material & Cupertino Libraries**: Pre-designed widgets that implement Material Design (Android) and Cupertino (iOS) design principles.

Flutter's Rendering Process

1. Flutter builds a widget tree representing the UI
2. Flutter converts the widget tree into a render tree
3. The render tree handles layout and painting
4. The Flutter engine renders the UI on the screen using Skia

3. Setting Up the Flutter Development Environment

System Requirements

- **Windows**: Windows 7 SP1 or later (64-bit)
- **macOS**: macOS 10.12 Sierra or later
- **Linux**: Ubuntu, Debian, or other Linux distros with desktop support

Installation Steps

1. Download Flutter SDK

Visit flutter.dev and download the Flutter SDK for your operating system.

2. Extract the SDK

Extract the downloaded archive to a location of your choice (avoid paths with special characters or spaces).

3. Add Flutter to Path

Add the Flutter “bin” directory to your system PATH

Windows :

```
set PATH "%PATH%;C:\path\to\flutter\bin"
```

macOS/Linux:

```
export PATH="$PATH:`pwd`/flutter/bin"
```

4. Add Flutter to Path

Run Flutter Doctor

```
flutter doctor
```

This command checks your environment and displays a report of the status of your Flutter installation.

5. Install IDE Extensions

- **VS Code**: Install the "Flutter" and "Dart" extensions
- **iOS** (macOS only): Install Xcode and set up a simulator

4. Creating Your First Flutter App

Create a New Flutter Project

```
flutter create my_first_app
cd my_first_app
```

Project Structure

- **android/**: Android-specific code
- **ios/**: iOS-specific code
- **lib/**: Dart code, where most development happens
- **pubspec.yaml**: Project configuration file
- **test/**: Test files

Run Your App

```
flutter run
```

Understanding the Default App

The default main.dart file contains a simple counter app. Let's break it down:

```
import 'package:flutter/material.dart';
void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
class MyHomePage extends StatefulWidget {
  MyHomePage({Key? key, required this.title}) : super(key: key);
  final String title;

  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            Text('You have pushed the button this many times:'),
            Text(
              '$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Key concepts:

- runApp(): Entry point for a Flutter app
- StatelessWidget: A widget that doesn't maintain state
- StatefulWidget: A widget that maintains state
- build(): Method that describes the part of the UI represented by the widget
- setState(): Method used to update the UI when state changes

5. Flutter Widgets

Types of Widgets

1. StatelessWidget

Widgets that don't store state information and remain unchanged throughout their lifecycle.

```
class MyText extends StatelessWidget {
  final String text;

  MyText(this.text);

  @override
  Widget build(BuildContext context) {
    return Text(text);
  }
}
```

2. StatelessWidget

Widgets that can change their appearance in response to events.

```
class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int count = 0;


  void increment() {
    setState(() {
      count++;
    });
  }


  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Count: $count'),
        ElevatedButton(
          onPressed: increment,
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

Common Widgets

Layout Widgets

- **Container**: A rectangular visual element
- **Row/Column**: Linear arrangement of children
- **Stack**: Overlay children on top of each other
- **ListView**: Scrollable list of widgets
- **GridView**: Scrollable grid of widgets

UI Widgets

- **Text**: Display text
- **Image**: Display images
- **Button**: Various button types (ElevatedButton, TextButton, IconButton)
- **TextField**: Text input
- **Checkbox/Radio/Switch**: Selection controls
- **Card**: Material design card

Structural Widgets

- **Scaffold:** Basic material design structure
- **AppBar**: Top app bar
- **Drawer**: Side navigation menu
- **BottomNavigationBar**: Bottom navigation buttons
- **TabBar**: Tabbed navigation

Widget Lifecycle

- **createState():** Called when creating a stateful widget
- **initState()**: Called when inserting the widget into the tree
- **build()**: Called when building the widget
- **didUpdateWidget()**: Called when the parent widget changes configuration
- **setState()**: Called when the widget's state is changed
- **dispose()**: Called when removing the widget from the tree

6. State Management

Types of Widgets

1. **Ephemeral (Local) State**: State that belongs to a single widget
   - Managed with setState()
   - Example: Form field values, animation state

2. **App (Shared) State**: State shared across multiple widgets
   - Requires state management solutions
   - Example: User data, authentication status

Types of Widgets

1. Provider

A simple approach using InheritedWidget.

```
// Create a model
class CounterModel extends ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();
  }
}

// Provide the model to the widget tree
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CounterModel(),
      child: MyApp(),
    ),
  );
}

// Use the model in a widget
```

```
class CounterDisplay extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text(
      'Count: ${context.watch<CounterModel>().count}',
    );
  }
}
```

2. **Bloc/Cubit**

Separates UI from business logic using streams.

3. **Riverpod**

An improved version of Provider with better dependency management.

4. **GetX**

A lightweight state management solution with dependencies and route management.

5. **Redux**

A predictable state container inspired by JavaScript Redux.

6. **MobX**

Makes state management reactive and observable.

7. Navigation and Routing

Basic Navigation

```
// Navigate to a new screen
Navigator.push(
  context,
  MaterialPageRoute(builder: (context) => SecondScreen()),
);

// Return to previous screen
Navigator.pop(context);
```

Named Routes

```
// Define routes
MaterialApp(
  initialRoute: '/',
  routes: {
    '/': (context) => HomeScreen(),
    '/details': (context) => DetailScreen(),
    '/settings': (context) => SettingsScreen(),
  },
);

// Navigate using named routes
Navigator.pushNamed(context, '/details');
```

Passing Data Between Screens

```dart
// First screen
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (context) => DetailScreen(item: item),
  ),
);

// Detail screen
class DetailScreen extends StatelessWidget {
  final Item item;

  DetailScreen({required this.item});

  @override
  Widget build(BuildContext context) {
    // Use item data
  }
}
```

Advanced Routing

For more complex routing needs, consider packages like:

- **go_router**: Declarative routing
- **auto_route**: Route generation
- **GetX**: Context-free navigation

8. Working with APIs

HTTP Requests with http Package

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

Future<List<Post>> fetchPosts() async {
  final response = await http.get(
    Uri.parse('https://jsonplaceholder.typicode.com/posts'),
  );

  if (response.statusCode == 200) {
    List<dynamic> data = jsonDecode(response.body);
    return data.map((json) => Post.fromJson(json)).toList();
  } else {
    throw Exception('Failed to load posts');
  }
}

class Post {
  final int id;
  final String title;
  final String body;

  Post({required this.id, required this.title, required this.body});

  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}
```

Using API Data in UI

```
class PostsScreen extends StatefulWidget {
  @override
  _PostsScreenState createState() => _PostsScreenState();
}

class _PostsScreenState extends State<PostsScreen> {
  late Future<List<Post>> futurePosts;

  @override
  void initState() {
    super.initState();
    futurePosts = fetchPosts();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Posts')),
      body: FutureBuilder<List<Post>>(
        future: futurePosts,
        builder: (context, snapshot) {
          if (snapshot.hasData) {
            return ListView.builder(
              itemCount: snapshot.data!.length,
              itemBuilder: (context, index) {
                return ListTile(
                  title: Text(snapshot.data![index].title),
                  subtitle: Text(snapshot.data![index].body),
                );
              },
            );
          } else if (snapshot.hasError) {
            return Center(child: Text('${snapshot.error}'));
          }
          return Center(child: CircularProgressIndicator());
        },
      ),
    );
  }
}
```

Alternative Approaches

- **dio**: Feature-rich HTTP client
- **retrofit**: Type-safe HTTP client
- **GraphQL**: For GraphQL APIs using packages like graphql_flutter

9. Flutter Packages and Dependencies

Managing Dependencies with pubspec.yaml

```yaml
name: my_app
description: A new Flutter project.

dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.5
  provider: ^6.0.3
  shared_preferences: ^2.0.15

dev_dependencies:
  flutter_test:
    sdk: flutter
  flutter_lints: ^2.0.0
```

Installing Dependencies

```
flutter pub get
```

Popular Flutter Packages

- **http/dio**: Network requests
- **provider/bloc/riverpod**: State management
- **shared_preferences**: Local data storage
- **sqflite**: SQLite database
- **path_provider**: File system access
- **camera**: Camera access
- **firebase_core**: Firebase integration
- **flutter_secure_storage**: Secure storage
- **flutter_local_notifications**: Local notifications
- **image_picker**: Pick images from gallery/camera

Creating Your Own Packages

1. Create a new Flutter package:

```
flutter create --template=package my_package
```

2. Develop your reusable functionality
3. Publish to pub.dev or use locally

10. Testing Flutter Applications

Managing Dependencies with pubspec.yaml

Types of Tests

1. Unit Tests

Test individual functions and methods.

```dart
// Function to test
int add(int a, int b) {
  return a + b;
}

// Test file
import 'package:test/test.dart';
import 'package:my_app/calculator.dart';

void main() {
  test('add function returns correct sum', () {
    expect(add(2, 3), 5);
    expect(add(-1, 1), 0);
    expect(add(0, 0), 0);
  });
}
```

2. Widget Tests

Test individual widgets.

```dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:my_app/widgets/counter.dart';

void main() {
  testWidgets('Counter increments when button is tapped', (WidgetTester
tester) async {
    await tester.pumpWidget(MaterialApp(home: Counter()));

    // Verify initial count is 0
    expect(find.text('Count: 0'), findsOneWidget);

    // Tap the button
    await tester.tap(find.byType(ElevatedButton));
    await tester.pump();

    // Verify count is now 1
    expect(find.text('Count: 1'), findsOneWidget);
  });
}
```

3. Integration Tests

Test how multiple widgets and services work together.

```
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';
import 'package:my_app/main.dart';

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  testWidgets('Complete login flow works', (WidgetTester tester) async {
    await tester.pumpWidget(MyApp());

    // Enter username and password
    await tester.enterText(find.byKey(Key('username')),
'user@example.com');
    await tester.enterText(find.byKey(Key('password')), 'password123');

    // Tap login button
    await tester.tap(find.byType(ElevatedButton));
    await tester.pumpAndSettle();

    // Verify we're on the home screen
    expect(find.text('Welcome'), findsOneWidget);
  });
}
```

Running Tests

```
# Run unit and widget tests
flutter test

# Run integration tests
flutter test integration_test
```

11. Deploying Flutter Applications

Android Deployment

Generate a Keystore

```
keytool -genkey -v -keystore ~/key.jks -keyalg RSA -keysize 2048
-validity 10000 -alias key
```

Configure Signing

Create `android/key.properties`:

```
storePassword=<password>
keyPassword=<password>
keyAlias=key
storeFile=<path to key.jks>
```

Configure `android/app/build.gradle`:

```
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 keystoreProperties['storeFile'] ?
file(keystoreProperties['storeFile']) : null
            storePassword keystoreProperties['storePassword']
        }
    }
    buildTypes {
        release {
            signingConfig signingConfigs.release
        }
    }
}
```

Build APK/App Bundle

```
# For APK
flutter build apk --release

# For App Bundle
flutter build appbundle --release
```

iOS Deployment

Configure App Signing

1. Open the iOS module in Xcode
2. Configure Signing & Capabilities with your Apple Developer account

Build IPA

```
flutter build ipa --release
```

Web Deployment

```
flutter build web --release
```

The output is in the build`/web` directory and can be deployed to any web hosting service.