Cross-Platform Mobile Development with Flutter
Outline

• Cross-Platform Development Tools
  • Different approaches

• Flutter Overview
  • Dart

• Hands-on with Flutter
  • Declarative, reactive-style UI programming

• Discussion on Native vs Multiplatform development
The Mobile Framework Market

- Developing multiple versions of the same application is costly!
- Smartphone platform market has recently been stable, but *smart devices* market is expanding:
  - IoT, TV, Wearables, Vehicles, ..

<table>
<thead>
<tr>
<th>Platform</th>
<th>Market Capture*</th>
<th>Language</th>
<th>IDE</th>
<th>Market Platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android</td>
<td>72.92%</td>
<td>Java, Kotlin</td>
<td>Android Studio</td>
<td>Google Play Store</td>
</tr>
<tr>
<td>iOS</td>
<td>26.53%</td>
<td>Objective-C, Swift</td>
<td>Xcode</td>
<td>iTunes App Store</td>
</tr>
<tr>
<td>Others (Windows, Blackberry, ..)</td>
<td>&lt; 0.5%</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Native Development

Building software via the “original” platform, using the Framework SDK/APIs directly

- **Android**
  - Android SDK + Android Studio
  - Kotlin/Java

- **iOS**
  - Xcode
  - Objective-C / Swift

If you are launching a product, maybe you also need web/desktop app too?
Cross-Platform Development

- Alternative principle to native app development
- Cross-Platform Tools (CPT), sometimes Multi-platform tools
- Share a code base between platforms
  - As large as possible (100% generally not possible)

“The ultimate goal of cross-platform mobile app development is to achieve Native app performance and run on as many platforms as possible”[1]

- Decrease dev. and maintenance cost of applications

Cross-Platform Development

• Alternative principle to native app development
• Cross-Platform Tools (CPT), sometimes Multi-platform tools
• Shared code base between platforms
  • As large as possible (100% generally not possible)
• Decrease dev. and maintenance cost of applications

“The ultimate goal of cross-platform mobile app development is to achieve Native app performance and run on as many platforms as possible”[1]

Can you name any multiplatform technologies you are already using daily?

Web-based CPT

Historically the most common cross-platform approach.

- App UI with HTML + CSS
- Interactive functionality with JavaScript

Strategies:

1. **“Normal webpage”**
   - Load the app. from a remote webserver (with a browser)

2. **Hybrid Application**
   - Webpage contained locally on the device, using a web-to-native wrapper
   - Uses platform’s built-in browser engine
     - Bridging between the WebView and Wrapper Application
   - Can be distributed on the platforms application market
More approaches and examples

• Hybrid
  • Apache Cordova, PhoneGap, Ionic

• Interpreted Approach
  • A dedicated interpreter on the device executes non-native code
  • e.g. Titanium, React Native (Javascript)

• Cross compiled
  • Non-native code is compiled to native during build
  • e.g. Xamarin (C#, .NET), Flutter (Dart)

• Others
Interpreted approach

• Don’t rely on a WebView
• But can still write programs in non-native language
• For example, Javascript with an interpreter:
  • JavaScriptCore - the default interpreter on iOS devices
  • On Android devices, V8 is a widely used engine
• Also uses a bridging layer like Hybrid approach
Cross-compile approach

• Code is compiled to native byte code
• Thus, **no bridging layer**
• Instead, the CPT Framework SDK includes a cross-compiler which maps functionality to the underlying platforms’ SDK
  • E.g., Xamarin
Adapted from Bjørn-Hansen et al.

Runtime Abstraction Layer: WebView/Interpreter
or
Mappings between CPT framework SDK and native SDK

Native SDK API-s
- Location
- Camera
- Storage
- Phone
- ...

Adapted from Bjørn-Hansen et al.
Introducing Flutter

Flutter is an app SDK for building high-performance, high-fidelity apps for iOS, Android, and web (tech preview) from a single codebase.

The goal is to enable developers to deliver high-performance apps that feel natural on different platforms. We embrace differences in scrolling behaviors, typography, icons, and more.

• Unveiled 2015, @ Dart Developer Summit under the name Sky
  • Stable v1.0 released December 2018
• Developed by Google, open-source

https://flutter.dev/docs/resources/technical-overview
Flutter – a set of tools

- Reactive-style programming framework
- Dart – modern object-oriented language
- Skia 2D rendering engine
  - Works with different hardware and softw. platforms
    - Google Chrome, Android, Firefox, Sublime Text 3
    - OpenGL ES2-accelerated
- Set of widgets – uniform design language
- Development tools
- Can target
  - Android
  - iOS (still need Apple hardware to develop)
  - (Web applications)
Dart

• Created by Google in 2011, OO language
• Syntax similar to Java, Javascript
• Statically typed, has type inference

// Define a function.
printInteger(int aNumber) {
  print('The number is $aNumber.'); // Print to console.
}

// This is where the app starts executing.
main() {
  var number = 42; // Declare and initialize a variable.
  printInteger(number); // Call a function.
}

Web-based dart console: https://dartpad.dev/
https://dart.dev/samples
Some differences to Java, Kotlin

- Dart doesn’t have keywords `public`, `protected`, `private`.
  - If an identifier starts with an underscore (_), it’s private to its library.
    
    ```dart
    var _gameScore = 0;
    ```

- Unlike Kotlin, you have to handle null-safety

- **Spread** operator for lists
  
  ```dart
  var list = [1, 2, 3];
  var list2 = [0, ...list]; // list2 is now [0,1,2,3]
  ```

- Asynchronous calls built-in
  
  - `async, await` keywords
    
    [https://dart.dev/codelabs/async-await](https://dart.dev/codelabs/async-await)

    [https://dart.dev/guides/language/language-tour](https://dart.dev/guides/language/language-tour)
Dart Functions

```dart
bool isEvenNumber(int x) {
  return x % 2 == 0;
}

bool isEvenNumber(int x) => x % 2 == 0;

void greet() {
  print("Joe");
}
```

• => shorthand

https://dart.dev/guides/language/language-tour#functions
Dart Functions - arguments

Both positional and named parameters supported

• Positional:
  • optional arguments with „[ ]“:

```dart
void sendMessage(String from, String msg) { }
void sendMessage(String from, String msg, [int priority]) { }
```

• Named arguments (all optional).
  • Define:
    ```dart
    void sendMessage({String from, String msg, int priority}) { }
    ```
  • Invoke:
    ```dart
    sendMessage(from: "Joe", msg:"Hi", priority:1 );
    ```
  • Specify required parameters:
    ```dart
    void sendMessage({String from, @required String msg, int priority}) { }
    ```

https://dart.dev/guides/language/language-tour#functions
Dart Constructors

• No data class like Kotlin.
• However, there is a short-hand for initializing variables

```dart
class Person {
  String name;
  int age;

  Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
}
```

// Same can be written as
```dart
class Person {
  String name;
  int age;

  Person (this.name, this.age);
}
```
Flutter Widgets

• Flutter’s UIs are made up of Widgets
  • widget is an immutable declaration of part of the user interface

• “Everything’s a widget”
  • Even layouts, fonts, colour themes

• Roughly equivalent to View-s in Android
  • Can be Visible (Button, Text) or Invisible (Container)

• Flutter Widgets are immutable!
  • Flutter creates a new tree of widget instances. In comparison, an Android view is drawn once and does not redraw until invalidate is called.

https://flutter.dev/docs/resources/technical-overview#everythings-a-widget
Widget hierarchy

• A widget’s main job is to provide a `build()` method
  • describes how to display the widget in terms of other, lower level widgets.

• Widgets are composed into a tree.
  • The root widget acts as the “application” object.
  • Child objects inherit parents’ properties.

• You can also control the layout of a widget by composing it with other widgets.
  • For example, to center a widget, you wrap it in a `Center` widget. There are widgets for padding, alignment, row, columns, and grids, ...

https://flutter.dev/docs/development/ui/widgets-intro
import 'package:flutter/material.dart';

void main() => runApp(
  Center(child:
    Text("Hello World", textDirection:TextDirection.ltr)
  )
);
Material Design Library

• Material is Google’s standard visual design language for mobile and the web.
  • Flutter offers a rich set of Material widgets.

• MaterialApp – root Widget of the library

• **Scaffold** Widget – provides standard material components
  • a default app bar with title
  • body property - holds the widget tree for the home screen. The widget subtree can be quite complex.

• Similarly, a Cupertino library exists

https://api.flutter.dev/flutter/material/MaterialApp-class.html
Images from: https://pub.dev/packages/flutter_platform_widgets
import 'package:flutter/material.dart';

void main() => runApp(MyApp());

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text("Lecture App"),
        ), // AppBar
        body: Center(
          child: Column(
            children: <Widget>[
              Text("Hello"),
              Text("World"),
              RaisedButton(
                child: Text("Greet!"),
                onPressed: () => print("Hi"),
                color: Colors.amberAccent,
              ), // RaisedButton
            ], // <Widget>[]
          ), // Column, Center, Scaffold, MaterialApp
        ),
      ), // MaterialApp
    );
  }
}
Understanding the widget tree

• Define the unique characteristics of a widget by implementing a `build()` function
  • `Build()` returns a tree (or hierarchy) of widgets.

• With this nesting (composition), you form a widget tree

• Flutter efficiently renders the UI based on the widget descriptions
Flutter Declarative UI

• Flutter uses a declarative approach to program the UI; including updating the UI

• In Native Android, you declare the XML layout, but then programmatically directly modify elements of the inflated layout

• Flutter, by contrast, lets the developer describe the current UI state and leaves the transitioning to the framework.

• Widgets are immutable, lightweight blueprints, not the actual UI rendered objects

https://flutter.dev/docs/get-started/flutter-for/declarative
Declarative vs Imperative UI example

- Assume the user entered an invalid input and we want to show this in UI with color and text:
  - Imperative:
    ```java
    // Referring to existing b, c objects
    b.setBackgroundColor("#FF0000");
    c.setText("Invalid input!");
    ```
  - Flutter:
    ```dart
    return ViewB(
        color: red,
        child: Text("Invalid input!"),
    )
    ```

https://flutter.dev/docs/get-started/flutter-for/declarative
State changes in UI

• Instead of updating the UI when state changes, you only update the state, and that triggers a redraw of the user interface.

• UI is automatically updated when state changes
Stateful vs stateless widgets

- Stateless widgets - immutable
  - their properties can’t change—all values are final.
  - Usually Icon, Button, Text

- Stateful widgets
  - maintain state that might change during the lifetime of the widget.
  - for example, it can change its appearance in response to events triggered by user interactions or when it receives data.
  - E.g. Checkbox, Radio, Slider, InkWell, Form, and TextField

https://flutter.dev/docs/get-started/codelab#step-3-add-a-stateful-widget
Stateful widgets

• Implement two classes:
  1) a StatefulWidget
     • Immutable, re-created
  2) a State class
     • Persists over lifetime of widget

• When the widget’s state changes, the state object calls setState(), telling the framework to redraw the widget.

https://flutter.dev/docs/get-started/codelab#step-3-add-a-stateful-widget
Creating a StatefulWidget

class MyStatefulWidget extends StatefulWidget {
  @override
  State<StatefulWidget> createState() {
    return MyState();
  }
}

class MyState extends State {
  var _gameScore = 0;

  @override
  Widget build(BuildContext context) {
    return Text("Score is $_gameScore");
  }
}

• Widget has to create the State object
• State implements the build() method
• You set the new state values inside the setState() method
Code Example
Managing State

• There are several approaches to who should manage the state of a widget.
  • Parent widget manages state, forwards state values to child widget in build
  • Widget manages its own state

• Basic rule of thumb:
  • If the state is just to do with aesthetics (e.g. icon changes, manage it within the widget itself)
  • If it involves user data, manage it in the parent

• If the tree is large, forwarding state can be cumbersome
  • InheritedWidget, ScopedModel for these cases

https://flutter.dev/docs/development/ui/interactive#managing-state
Stateful Widget A

```
var myState;
void updateAState() {
  ...
}
```

**Button**

- onPressed: updateAState()
- text: myState.toString()

Stateful Widget B

```
var myState;
void updateBState() {
  ..
}
```

pass down pointers to myState, updateBState

Stateless Widget C

```
var myState;
void updateCState() {
  ..
}
```

**Button**

- onPressed: updateBState()
- text: myState.toString()

Widget manages own state

Parent manages state

https://flutter.dev/docs/development/ui/interactive#managing-state
Flutter System Overview

<table>
<thead>
<tr>
<th>Framework</th>
<th>Material</th>
<th>Cupertino</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dart</td>
<td>Widgets</td>
<td>Rendering</td>
</tr>
<tr>
<td></td>
<td>Animation</td>
<td>Painting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Gestures</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Foundation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Engine</th>
<th>Service protocol</th>
<th>Composition</th>
<th>Platform channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/C++</td>
<td>Dart isolate setup</td>
<td>Rendering</td>
<td>System events</td>
</tr>
<tr>
<td></td>
<td>Dart VM management</td>
<td>Frame scheduling</td>
<td>Asset resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frame pipelining</td>
<td>Text layout</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Embedder</th>
<th>Render surface setup</th>
<th>Render surface setup</th>
<th>Render surface setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform specific</td>
<td>Thread setup</td>
<td>Event loop interop</td>
<td></td>
</tr>
</tbody>
</table>

Image: https://flutter.dev/docs/resources/technical-overview
Rendering in Flutter

There’s more than one tree

```
App
  ├── ContainerWidget
  │    └── TextWidget
  │        ├── Color:red
  │        └── Text: „Foo Bar“
  └── Element

Element

Element

Element

AppRenderObject

ContainerWidgetRenderObject

TextWidgetRenderObject
```
Rendering Pipeline

https://www.youtube.com/watch?v=UUFxWzp0-DU
Create beautiful apps faster with Flutter's collection of visual, structural, platform, and interactive widgets. In addition to browsing widgets by category, you can also see all the widgets in the widget index.

- **Accessibility**
  Make your app accessible.
  [Visit](https://flutter.dev/docs/development/ui/widgets)

- **Animation and Motion**
  Bring animations to your app.
  [Visit](https://flutter.dev/docs/development/ui/widgets)

- **Assets, Images, and Icons**
  Manage assets, display images, and show icons.
  [Visit](https://flutter.dev/docs/development/ui/widgets)

- **Async**
  Async patterns to your Flutter application.
  [Visit](https://flutter.dev/docs/development/ui/widgets)

- **Basics**
  Widgets you absolutely need to know before building your first Flutter app.
  [Visit](https://flutter.dev/docs/development/ui/widgets)

- **Cupertino (iOS-style widgets)**
  Beautiful and high-fidelity widgets for current iOS design language.
  [Visit](https://flutter.dev/docs/development/ui/widgets)
Flutter Project Structure

• Projects are generated by the Flutter command-line tool (which IDE uses)
• Each project has a lib folder, which contains .dart files, e.g. main.dart
• Platform-specific sub-folders
  • android, ios
  • Source and compiled code
• pubspec.yaml
  • Defines project dependencies, versions
Adding dependencies, assets

• Flutter uses Dart’s own build system, and the Pub package manager.
  • The tools delegate the building of the native Android and iOS wrapper apps to the respective build systems.

• External dependencies to use in Flutter are generally defined in pubspec.yaml
  • Find Flutter packages in Pub: https://pub.dev/flutter

• Assets are also declared in pubspec.yaml
pubspec.yaml

```yaml
name: flutter_app

description: A new Flutter application.

version: 1.0.0+1

environment:
  sdk: ">=2.1.0 <3.0.0"

dependencies:
  flutter:
    sdk: flutter
  cupertino_icons: ^0.1.2

dev_dependencies:
  flutter_test:
    sdk: flutter

flutter:
  uses-material-design: true

https://dart.dev/tools/pub/pubspe
Example with assets

Specifying in .yaml:

```yaml
# The following section is specific to Flutter.
flutter:
  uses-material-design: true

assets:
  - images/test.png
```

Creating a Widget:

```dart
Image.asset("images/test.png",
  width: 600,
  height: 240,
  fit: BoxFit.cover)
```
Accessing mobile features

• How do I use the Camera, Maps, SMS, ... of the smartphone from Flutter?
• Will it work exactly the same on iOS and Android?
• This functionality is split into plugins
• Repository of packages is available at:
  https://pub.dev/flutter
• E.g., geolocator for GPS
  • https://pub.dev/packages/geolocator
• If a functionality is missing, you can try and implement it yourself
  https://flutter.dev/docs/get-started/flutter-for/android-devs#flutter-plugins
https://flutter.dev/docs/development/platform-integration/platform-channels
Discussion

So is there any point in learning native anymore?

Multi-platform w/ Flutter

• Shared code-base, 2(+1) target platforms
• Dart – fairly new language
• Rapid development
• Some performance overhead
• Same UI engine on older devices
• Still have to track plugins features on all target platforms

Native

• Android TV, Android Wear
• Access to latest APIs immediately
• Kotlin/Java – larger developer pool
• Although Flutter is fast, theoretically fastest result will be native
• Relying on backwardsCompat libraries
• Ecosystem is larger
Next week

Working as a Mobile Developer
Kelian Kaio, Mooncascade

Fog Computing: Concepts, Challenges and Research Scope
Chinmaya K. Dehury, PhD
Research Fellow @ Mobile & Cloud Lab
References & Materials: