Design And Implementation Of Mobile Applications

Theory

Christian Rossi

Academic Year 2024-2025

Abstract

The course is organized into five distinct parts.

The first part sets the stage by framing the problem and exploring the numerous opportunities that mobile devices present. It also provides a brief overview of various alternatives and competing solutions in the field.

In the second part, we focus on mobile application design, aiming to identify guidelines and recurring patterns that can facilitate the design process and contribute to producing high-quality solutions.

The third part introduces key innovations and features of important frameworks, specifically Flutter and React Native, which are essential for developing cross-platform applications.

The fourth part delves into Android development, covering how to create applications for a range of Android devices, including phones, tablets, watches, and TVs, using Kotlin and Jetpack Compose.

Finally, the fifth part addresses the development of applications for iOS-based devices, utilizing Swift and SwiftUI to create efficient and effective mobile applications.

Contents

CHAPTER 1

Mobile applications

1.1 Introduction

The history of mobile devices began in 1973 when Martin Cooper at Motorola made the first-ever mobile phone call using a prototype, marking a pivotal moment in telecommunications. Over the decades, mobile devices have evolved from simple communication tools to highly advanced, multifunctional systems equipped with a wide array of sensors. Modern smartphones now integrate technologies such as accelerometers, gyroscopes, digital compasses, GPS, barometers, ambient light sensors, and proximity sensors, enabling a range of applications from navigation to augmented reality.

Alongside hardware advancements, mobile programming has grown to support a variety of platforms, each with its own preferred development languages. Key programming languages used in mobile app development include:

- Objective-C and Swift: primarily for iOS development.
- Java and Kotlin: widely used for Android development.
- C#: common for cross-platform development, particularly with frameworks like Xamarin.
- HTML5 and JavaScript: popular for web-based and cross-platform apps, especially with frameworks like React Native.
- Python and other languages: increasingly used for cross-platform solutions through frameworks such as Kivy or BeeWare.

This diverse ecosystem of languages supports the ever-expanding capabilities of mobile devices and the broad range of applications they power.

1.2 Design principles

Effective mobile app design revolves around several core principles that prioritize usability, clarity, and user engagement. These principles guide the creation of intuitive and enjoyable user experiences:

1. *Keep it brief*: present information concisely to avoid overwhelming the user.

2. *Pictures are faster than words*: visual elements communicate more efficiently than text, improving comprehension and engagement.

3. *Decide for me but let me have the final say*: offer smart defaults and automation to simplify tasks, while still allowing users to override settings if needed.

4. *I should always know where I am*: ensure users have a clear sense of navigation and context within the app, reducing confusion and frustration.

To successfully implement these principles, designers should adopt the following approaches:

- *Mobile mindset*: design with a mobile-first approach, focusing on creating experiences that are streamlined, unique, and user-centered. Apps should be functional, visually appealing, and tailored to mobile usage patterns.

- *Identify different classes of users*: clearly define and understand the needs of your target audience. Users can be categorized into groups such as those who are bored, busy, or lost, each with distinct behaviors and expectations.

- *First Impressions Matter*: capture user interest within the first few seconds of interaction. To make a lasting impact:
  - Provide minimal or no help text, relying instead on intuitive design.
  - Create a distinctive and captivating look and feel.
  - Ensure the app quickly conveys its purpose and value.

CHAPTER 2

Flutter

2.1 Introduction

Flutter has emerged as a crucial framework for mobile and web development, offering a range of advantages that make it a popular choice for cross-platform applications:

1. *Widespread adoption*: Flutter has the highest number of users among cross-platform frameworks, making it a leading solution for multi-platform development.

2. *Google support*: as an open-source framework backed by Google, Flutter benefits from regular updates, community contributions, and long-term stability.

3. *Dart language*: Flutter uses Dart, a programming language similar to Java, making it easy to learn and adopt for developers familiar with object-oriented languages.

4. *Extensive platform support*: Flutter provides a wide codebase solution, supporting development across multiple devices and platforms, including mobile (iOS, Android), web, and desktop.

Flutter's architecture can be understood through three distinct abstraction layers, each playing a key role in the app development process:

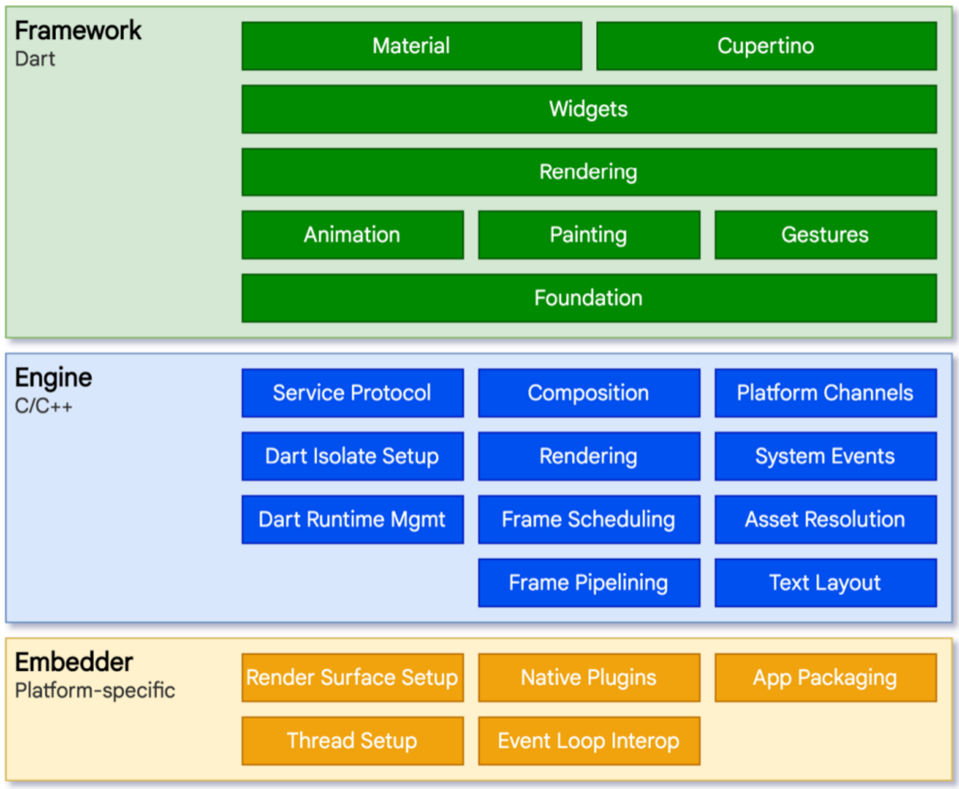

Figure 2.1: Flutter architecture

The architecture consists of the following layers:

- *Embedder*: the embedder is responsible for integrating Flutter's source code into specific platforms, such as iOS, Android, or desktop environments. This layer, developed by Google, ensures that Flutter can operate seamlessly across diverse platforms.
- *Engine*: the engine manages the execution of the application on various devices, handling the compilation process and ensuring that the code is optimized for each target platform.
- *Framework*: written in Dart, the framework is the layer that developers interact with the most. It provides the tools and libraries necessary to create the application's user interface, manage state, and implement core functionality.

2.2 Dart

Dart is an object-oriented programming language developed by Google, designed to be easy to learn and highly efficient for building applications across various platforms. Key features of Dart include:

- *Object-oriented design*: Dart is a class-based language with single inheritance, utilizing a familiar C-style syntax that makes it accessible to developers with backgrounds in languages like Java or C++.
- *Rich type system*: Dart supports interfaces, abstract classes, generics, and both optional and strong typing, providing flexibility while ensuring type safety.

- *Everything is an object*: in Dart, everything is treated as an object, including numbers, functions, and even null values. This uniformity simplifies programming and enhances code consistency.
- *Optional type annotations*: while Dart supports type annotations, they are optional, allowing developers to choose how explicitly they want to define types.
- *Function and variable declarations*: Dart allows for the declaration of top-level functions as well as functions associated with classes. Similarly, it supports both top-level variables and class-bound variables.
- *Private identifiers*: any identifier that starts with an underscore (_) is considered private to its library, encapsulating functionality and enhancing modularity.

These features make Dart a powerful choice for developers working with Flutter, enabling them to create robust and efficient applications.

2.3 Dart applications

Creating a Flutter application can be done easily through the command line interface. To initiate a new application, use the following command:

```
flutter create app_name
```

This command generates a directory structure containing various subdirectories and files. Some of these subdirectories contain auto-generated code for different platforms, while others are critical for development:

- `lib` directory: this folder holds all the handwritten code for your application.
- `pubspec.yaml` file: this file serves as the application's table of contents. It is used to import external libraries and declare dependencies.

To run the application, execute the command:

```
flutter run
```

This command allows you to choose the platform on which you want to run the application. If certain platforms are unavailable on your development device, Flutter will emulate the desired device.

Initially, the `lib` directory contains a file named `main.dart`, which includes the following code:

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

void main() {
    runApp();
}
```

Depending on the desired look and feel, you can import either `material.dart` for Material Design or `cupertino.dart` for iOS-style components. The `main` function contains the `runApp` method, which initializes the application.

The `runApp` function takes a given Widget and sets it as the root of the widget tree. This tree typically consists of two widgets: the Center widget and its child, the Text widget. The Flutter framework ensures that the root widget covers the entire screen.

2.3.1 Functions

In Dart, a function can have any number of required positional parameters, which may be followed by either named parameters or optional positional parameters, but not both simultaneously. Named parameters are optional unless explicitly marked as required. You can also define default values for both named and positional parameters using the equals sign (=). These default values must be compile-time constants, and if no default value is provided, it defaults to null.

2.3.2 Commas

It is a best practice to always add a trailing comma at the end of a parameter list—this applies to functions, methods, and constructors where maintaining formatting is important. Adding trailing commas helps the automatic formatter insert the appropriate amount of line breaks, ensuring your code adheres to Flutter's styling guidelines.

2.3.3 Widgets

The structure of a Flutter application is fundamentally composed of widgets, which are the building blocks of the user interface. The following figure illustrates the material design structure used in Flutter applications:

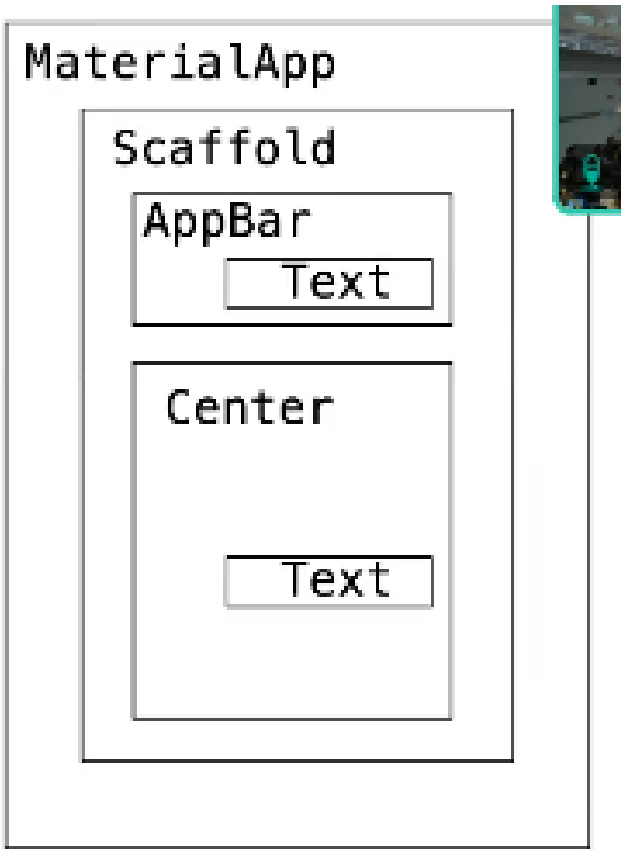

Figure 2.2: Material structure

In Flutter, graphical user interfaces (GUIs) are created entirely through code, where nearly everything is represented as a widget. Key characteristics of widgets include:

- *Immutability*: a widget is an immutable object, meaning that once it is created, its properties cannot be changed. Instead, to modify a widget's appearance or behavior, a new widget must be created.
- *Composability*: widgets are composable, allowing developers to combine existing widgets to build more complex interfaces. This enables the creation of new widgets by composing smaller, reusable ones.
- *Descriptive nature*: widgets describe what the GUI should look like. They provide a visual representation of the interface and its components.

When the state of a widget changes, the Flutter framework responds by rebuilding the affected widget. The framework performs the following steps:

1. *Diff computation*: it computes the difference between the current widget state and the previous rendering, determining what has changed.
2. *Minimal changes*: the framework identifies the minimal set of changes needed to update the interface, optimizing performance.
3. *Selective re-rendering*: only the parts of the UI that have changed are re-rendered, which enhances efficiency and responsiveness.

This architecture of widgets allows for flexible and dynamic user interfaces that can easily adapt to user interactions and data changes.

All widgets have a state, which can be categorized as either mutable or immutable. The state of a stateful widget is changeable; for example, it may represent the current position of a slider or whether a checkbox is checked. This state is stored in a State object, allowing for a clear separation between appearance and content. To update the appearance of the widget in response to changes, we need to call `setState`, which informs the framework to redraw the widget.

In contrast, a stateless widget does not manage any internal state. Stateful widgets maintain state that can change, allowing them to alter their appearance based on user events or incoming data. Implementing a stateful widget requires two classes: one that extends `StatefulWidget` and another that defines the mutable values and includes the `build` method to render the widget.

Const and final In Dart, the keyword `const` is used for values known at compile time (e.g., `const a = 1`), while `final` is appropriate for values that are determined at runtime. Anything not known at compile time should be declared as `final` instead of `const`.

State management The `setState` method notifies Flutter of a state change, prompting the widget to be re-rendered. This triggers the re-execution of the `build` method, ensuring the GUI reflects any updates. If the state variable (e.g.,`_counter`) is modified without invoking `setState`, the `build` method will not be called, and the GUI will remain unchanged.

Flexible The `Flexible` widget allows for resizable elements within the layout. Unlike inflexible widgets, which are fixed in size, flexible widgets adjust according to their properties: `flex` and `fit`. The `fit` property dictates whether the widget occupies extra space, with `FlexFit.loose` using the widget's preferred size by default, and `FlexFit.tight` enforcing the widget to fill all available space. The `flex` property defines the resize ratio relative to other widgets.

Other options The `Expanded` widget can wrap a widget, compelling it to fill any available extra space. Conversely, `SizedBox` can wrap a widget and resize it using specified height and width properties; it can also create empty space by setting these dimensions. The `Spacer` widget generates space between widgets, utilizing the flex property, while `SizedBox` provides space based on a specific number of logical pixels.

2.3.4 Assets

To incorporate external assets, they must be explicitly declared in the project. To include all assets within a directory, use the directory name followed by a slash (e.g., `assets/`). If assets are located in subdirectories, each directory must have a separate entry in the asset declaration.