Developing a DVB-I Parser Library in Dart and GUI App in Flutter

1st Luis Hebendanz
M. Sc. Computer Science
TU Berlin
Berlin, Germany

2nd Nicklas
M. Sc. Information Systems Management
TU Berlin
Berlin, Germany

3rd Balint
M. Sc. Information Systems Management
TU Berlin
Berlin, Germany

***Abstract*—In this project, we aimed to develop an efficient DVB-I parser library in Dart and a GUI app in Flutter to present TV services on Android devices.**

We used the Dart programming language to write the DVB-I parser library and Google's cross-platform Flutter framework to develop the GUI app.

However, we faced challenges in developing the GUI app in Flutter due to multiple bugs in the libraries we used and sparse documentation. Despite these challenges, we were able to develop a working app to present TV services on Android devices.

Our project demonstrates the feasibility of using Dart to write an efficient DVB-I parser library and Flutter to develop a GUI app that presents TV services on Android devices. However the challenges we faced in developing the GUI app highlight the importance of mature libraries and documentation to support developers in using these technologies. The lack of maturity in the Flutter ecosystem has compelled us to not recommend it for further projects.

***Index Terms*—IP-TV, DVB-I, Flutter, Dart, Cross Platform**

I. Introduction

In this project, we aimed to develop an efficient DVB-I parser library in Dart and a GUI app in Flutter to present TV services on Android devices. The DVB-I standard is a standards-based solution for delivering television via the internet and offers a discovery mechanism to signal and discover television services, using a set of REST APIs allowing clients to retrieve a list of services in an XML-based format. Our primary objective was to create a parser library that can efficiently handle the DVB-I service list registry and provide all the necessary information required to present the TV service in the client app. Additionally, we aimed to create an Android GUI app using Flutter that uses the DVB-I parser library to present the TV services to the user.

The development of the DVB-I parser library involved reading the DVB-I standard and manually emulating REST requests as specified by the specification. We also familiarized ourselves with the Dart programming language and experimented with simple coding examples to gain proficiency with the language. Once we had a good understanding of the standard and the language, we designed and developed the DVB-I parser library using Dart, implementing the required REST APIs and XML parsing.

The development of the Android GUI app using Flutter involved building an intuitive user interface to present the TV services to the user, as well as incorporating the DVB-I parser library to retrieve and display the information for each service. We faced challenges while developing the app, including multiple bugs in libraries used and sparse documentation, which affected the app's functionality and usability.

In this evaluation, we will assess the success of our project in achieving its goals and evaluate the performance and usability of the DVB-I parser library and Android GUI app developed. We will also discuss the challenges encountered during development and recommend future improvements to enhance the overall functionality and usability of the app.

II. Scientific Background

This section will provide an overview of the utilized technologies and their respective functionalities.

III. Flutter Architecture Overview

Flutter is a cross-platform framework for building mobile and desktop applications. Its architecture consists of four main layers: the Dart app layer, the framework layer, the engine layer, and the platform layer. Shown in figure 1.

The Dart app layer is responsible for composing widgets into the desired UI and implementing business logic. It is owned by the app developer.

The framework layer provides a higher-level API for building UI apps, including widgets, hit-testing, gesture detection, accessibility, and text input. It composites the app's widget tree into a scene.

The engine layer is responsible for rasterizing composited scenes and provides low-level implementation of Flutter's core APIs, including graphics, text layout, and the Dart runtime. It exposes its functionality to the framework using the dart:ui API and integrates with a specific platform using the platform layer.

The platform layer "is the native OS application that hosts all Flutter content and acts as the glue between the host operating system and Flutter" [1]. Flutter includes platform embedders for each of the target platforms, and you can also create a custom platform embedder.

In summary, Flutter's architecture provides a robust and efficient rendering pipeline, bypassing system UI widget libraries and using its own widget set and Skia 2D library [2] for rendering. This results in a high-performance, cross-platform framework with minimal abstractions and overhead.

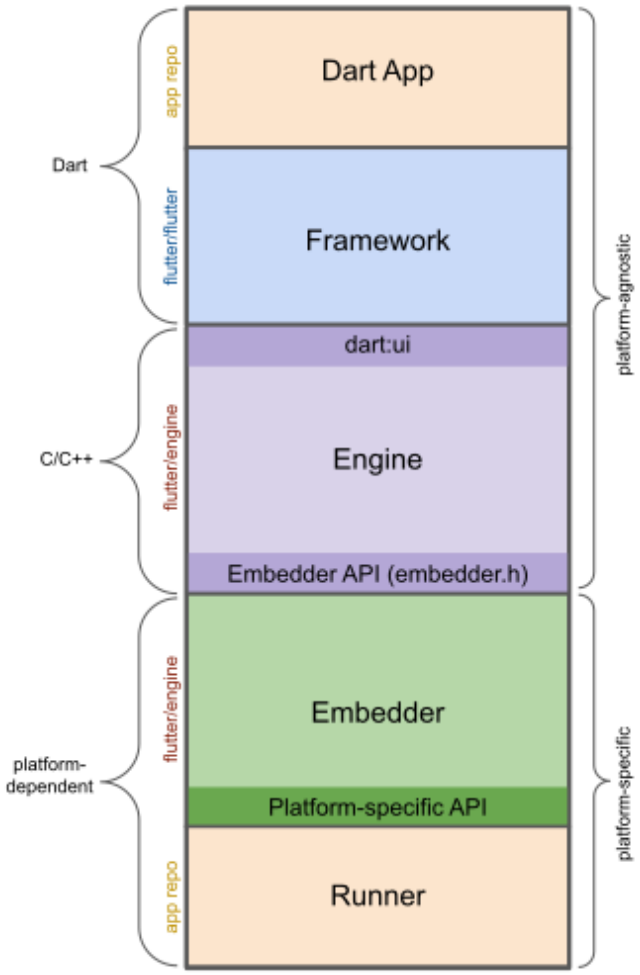

Fig. 1. Flutter architectural overview [1]

IV. Integrating Flutter with other code

Flutter and Dart provide several ways to integrate with other code written in different programming languages.

For mobile and desktop apps, Flutter provides platform channels, which are a mechanism for communicating between Dart code and the platform-specific code of the host app. This allows developers to call custom code written in languages like Kotlin, Swift, or C-based APIs, including those generated for code written in modern languages like Rust or Go. The platform channel mechanism serializes Dart types into a common message format, which is sent to the receiving code that then deserializes it into a programming language-specific object displayed in figure 2.

For C-based APIs, Dart provides a direct mechanism for binding to native code using the dart:ffi library, which can be considerably faster than platform channels since no serialization is required to pass data.

For web apps, the js package serves a similar purpose to dart:ffi for C-based APIs, allowing developers to bind to native code directly from Dart. Additionally, since web apps run in the browser, developers can also use Dart's interop capabilities to integrate with JavaScript libraries, either by importing JavaScript libraries into Dart or by calling Dart code from JavaScript.

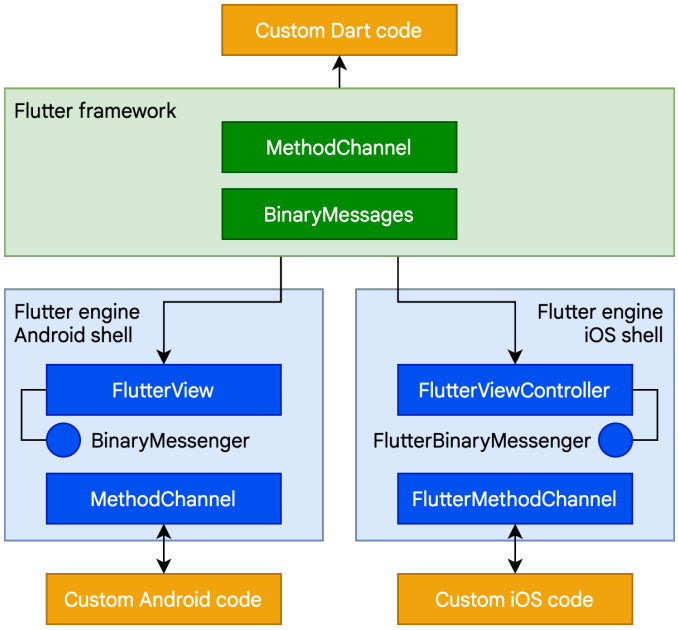

Fig. 2. Platform Channels Overview [1]

Overall, Flutter and Dart provide a variety of interoperability mechanisms that allow developers to integrate their code with other programming languages and platforms, providing flexibility and extensibility to their applications.

V. Dart Language

Dart is a class-based, object-oriented programming language with syntax that is similar to Java-style languages. It is optimized for client-side web and mobile app development, but can also be used for server-side programming. Some key concepts are:

1) Everything in Dart is an object, and every object is an instance of a class.
2) Dart has a garbage collector that automatically frees memory that is no longer in use.
3) Type annotations are optional in Dart because the language can infer types.
4) If you enable null safety in Dart, variables cannot contain null values unless you explicitly make them nullable with a question mark (?).
5) Dart does not have keywords for public, protected, and private access, but an identifier starting with an underscore (_) is private to its library.
6) Dart supports concurrent programming with async-await, Future, and Stream objects.
7) A Future in Dart represents the result of an asynchronous operation and can be completed with a value or an error.

VI. Dart Asynchronous Programming

In Dart, async/await is a powerful mechanism that allows for writing asynchronous code in a synchronous style. With

async/await, developers can write code that doesn't block the application's main event loop while waiting for I/O operations, network requests, or other time-consuming tasks to complete.

To use async/await in Dart, developers mark functions as "async" and use the "await" keyword to wait for the result of an asynchronous operation. When a function is marked as "async", it returns a Future object that can be used to obtain the result of the computation when it completes. The "await" keyword is used to wait for the Future to complete, and once it does, the result is returned as if it were a normal synchronous operation.

Async/await is particularly useful in situations where multiple asynchronous operations need to be performed in sequence, and the result of each operation is dependent on the completion of the previous one.

A. DVB-I Standard

The DVB-I (DVB-Internet) standard is a specification developed by the Digital Video Broadcasting (DVB) organization for delivering linear television services over the internet.The standard defines the mechanisms to be used to find sets of linear television services delivered through broadband or broadcast mechanisms as well as methods to retrieve electronic programme data for those services.[ref to document]

The core of the specification is the Service List Registry (SLR) which provides a set of REST APIs allowing clients (TVs, Mobile, Browser) to retrieve a list of services (ServiceList) in an XML-based format including all information required to present the TV service in the client. The DVB-I standard, published by ETSI, expands upon the existing DVB Broadcast-based delivery methods, such as Terrestrial, Satellite, and Cable, to provide simpler and more accessible options for viewing both Linear and VOD Streaming services on any internet-connected device. With this open standard, users can enjoy a seamless viewing experience across multiple devices without any limitations.

VII. Implementation

Our implementation is split into four big parts: the development setup VII-A, a general overview of the architcture VII-B, the dvb-i parser libary, which contains all the xml parsings and http requests VII-C and finally the user interface which the client interacts with. VII-D. The code for this implementation can be found on *github*[1].

A. Development Setup

Development Setup: Using Nix for a development environment TODO: Setup for Windows describe important libs that were used

B. Architecture

1) General Overview: General Architecture: walkthrough of how a user interacts with client and how parser requests more info graphic of architecture

[1]https://github.com/AWT-DVBI/dvbi_client

C. Dvb-I Parser

libraries used XML to Dart object mapping Features: Lazy loading. Only request data you need. Request more on obj access Async http request for parallel fetching XML to JSON parsing

D. Gui Application

libraries used(?) layout of GUI/Screenshots Content Guide Page features: breaks everything Channel Browsing features: subtitles, volume control, next/prev channel, show channel logo and name

VIII. Challenges

Dart Streams, Dart async idea: reduce loading times by asynchr loading channels. State Management in Flutter: Unaccounted for Dart language features in Riverpod Riverpod: Providers + Dart streams = StreamProviders out-of-the-box state management using State widgets

DVB-I Server wrong response data with official endpoints (use of different staging Endpoints for correct data) start and end time flexibility in standard leads to many edge cases Bugs in libraries: videoplayer, chewie o(n2) performance issue due to the structure of the given xml tree and our used xml lib

IX. Evaluation

scalability include screenshots cross platform framework flutter ....does it work on android and apples ios? only android in combination with exoplayer as mpeg dash is not supported by hls and the current flutter videoplayer libs

X. Conclusion and Future Work

subtitles, channel info button

A. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. 3", even at the beginning of a sentence.

TABLE I
Table Type Styles

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a]Sample of a Table footnote.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization

Fig. 3. Example of a figure caption.

{A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

Please number citations consecutively within brackets [3]. The sentence punctuation follows the bracket [4]. Refer simply to the reference number, as in [5]—do not use "Ref. [5]" or "reference [5]" except at the beginning of a sentence: "Reference [5] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [6]. Papers that have been accepted for publication should be cited as "in press" [7]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [8].

REFERENCES

[1] Flutter, "Flutter architectural overview", [Online] https://docs.flutter.dev/resources/architectural-overview
[2] Skia 2D, "Skia 2D Rendering Library", [Online] https://skia.org/
[3] G. Eason, B. Noble, and I. N. Sneddon, "On certain integrals of Lipschitz-Hankel type involving products of Bessel functions," Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.
[4] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
[5] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
[6] K. Elissa, "Title of paper if known," unpublished.
[7] R. Nicole, "Title of paper with only first word capitalized," J. Name Stand. Abbrev., in press.
[8] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
[9] M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.