Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Web Technologies on the Desktop: An Early Look at Flutter

Elias Müller

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Prof. Dr. Marco Aiello
Commenced:	November 10, 2020
Completed:	May 10, 2021

Abstract

Academic research into cross-platform desktop application frameworks has been fairly limited in recent years due to a lack of innovation within the sector. Businesses and companies prioritise the development of Web applications, or even mobile applications. Flutter is a relatively new tool-kit from Google which allows the development of mobile, Web and desktop applications from a single code base. With Flutter desktop reaching beta stage, we explored the tool-kit on the desktop. We did this by first designing a basic application which we then implemented twice using Flutter and Electron - a competing framework. We automated the operation of these applications, and we recorded system information such as CPU, GPU, and memory usage. Lastly, we considered Flutter's desktop features. Although we encountered some issues during development, the development process with Flutter was nonetheless faster and required about 30% less code. However, our recorded usage statistics highlighted that this product is still very much in its beta stages. Our Flutter application took significantly longer to complete one of our tasks in release mode than in development mode, with CPU usage being constantly above 10% while GPU usage was above 40%. Furthermore, Flutter is still missing some features such as support for multiple windows per application or customizable context menus. Based on our results, we cannot recommend Flutter for productive use on the desktop yet. However, the framework shows great potential.

Keywords: Cross-platform, Desktop, Desktop Applications, Electron, Flutter

Kurzfassung

In den letzten Jahren haben Frameworks für die Entwicklung von plattformunabhängigen Desktop-Anwendungen in der Forschung aufgrund weniger bahnbrechender Neuheiten vergleichsweise wenig Aufmerksamkeit erhalten. Unternehmen entwickeln im Allgemeinen lieber für das Web oder den mobilen Markt. Flutter ist ein vergleichsweise neues Toolkit von Google für die Entwicklung mobiler, Web- und Desktop-Anwendungen aus einer gemeinsamen Code-Basis heraus. Vor Kurzem hat Flutter Beta-Status auf dem Desktop erreicht - ein guter Zeitpunkt, sich das Framework näher anzusehen. Im Rahmen dieser Bachelorarbeit implementierten wir dieselbe Anwendung einmal in Flutter und einmal in Electron - einem alternativen Framework für plattformunabhängige Desktop-Anwendungen. Anschließend automatisierten wir die Bedienung der Anwendungen und zeichneten während der Benutzung die Auslastung von CPU und GPU sowie den Arbeitsspeicherverbrauch auf. Zuletzt warfen wir auch einen Blick auf die aktuelle Situation bezüglich noch fehlender Desktop-Features. Obwohl wir mit Flutter auf kleinere Probleme während der Implementierung stießen, benötigten wir weniger Zeit und etwa 30% weniger Code als für die Electron-Umsetzung. Der Beta-Status macht sich aber spätestens bei der System-Auslastung bemerkbar. Im Release-Modus brauchte unsere Anwendung für die Bewältigung einer Aufgabe erheblich länger als im Development-Modus. Außerdem blieb die CPU-Auslastung dauerhaft jenseits der 10%- und die GPU-Auslastung sogar über der 40%-Grenze. Was Features betrifft, fallen besonders die noch fehlende Unterstützung von mehreren Fenstern pro Anwendung und die noch nicht anpassbaren Kontextmenüs ins Gewicht. Aufgrund unserer Ergebnisse können wir Flutter noch nicht für den Produktiveinsatz auf dem Desktop empfehlen. Trotzdem bewerten wir das Framework als vielversprechend.

Schlüsselwörter: Desktop, Desktop-Anwendungen, Electron, Flutter, Plattformunabhängig

Contents

List of Figures

List of Tables

Listings

Acronyms

CPU Central Processing Unit. 3, 5, 7, 9, 18

CSS Cascading Style Sheets. 22

GPU Graphics Processing Unit. 3, 5, 7, 9, 18

HTML Hypertext Markup Language. 19

HTTP Hypertext Transfer Protocol. 19

IPC Inter-Process Communication. 23

JSON JavaScript Object Notation. 21

PWA Progressive Web Application. 28

UI User Interface. 24

URL Uniform Resource Locator. 19

1 Introduction

Due to the success and dominance of the Web, traditional desktop applications are becoming less and less important for both users and companies. Companies prefer to offer their services through a Web interface, where they can comfortably rely on proven technologies. For users, this simplifies operation by allowing them to access services from anywhere, and it also eliminates the need to install an application on their machines. However, traditional desktop applications are still relevant. Business and professional software, as well as performance-critical applications, are often still developed for the desktop. Mostly, these applications focus on a single platform. In order to target all platforms, developers have to rely on frameworks such as Qt and GTK or other tool-kits. Therefore, developers have to learn languages and concepts that differ from the ones used for Web applications. Due to the Web's dominance, using Web technologies to develop desktop applications makes sense.

However, the Web, and how we access and use it, is also evolving. Today, more people access the Web using mobile devices and apps, rather than desktop computers [Statcounter, 2021]. Therefore, while the Web remains high priority, the mobile platform has become increasingly important. Flutter is a relatively new framework which has its origins in the mobile space. It aims to allow the development of mobile applications, Web applications, and desktop applications from a single code base. Although the concept of writing code once and executing it on multiple platforms is not entirely novel, no one has ever actually achieved it in the way that Flutter does.

Flutter has been the subject of research and analysis. However, the focus tends to be on the mobile market and competing technologies in that field. Only recently, Flutter's desktop target was declared beta and "production-quality support" [Github, 2021d] is expected this year. Since we have not found any articles that explicitly focused on Flutter desktop, we delved further. We also hope that more awareness is brought to Flutter's potential on the desktop platform.

Based on our idea to investigate the desktop state of Flutter, we consider the following research questions:

- **RQ1:** Is Flutter already a viable choice for desktop application development?
- **RQ2:** How does Flutter stack up against technologies comparable to Flutter?

As for the second research question, we identified Electron as comparable technology on the desktop. Electron is a mature framework that allows the use of Web technologies like JavaScript to develop cross-platform desktop applications. A popular application written in Electron is Microsoft Teams [Maguire et al., 2020]. Other alternatives we looked at were React Native, Ionic framework, and Xamarin. However, none of these platforms support Linux or if they do, they use Electron for it [Ionic, 2020].

In order to fully consider the research questions, we implement the same application in Flutter and Electron. We then record system usage such as Central Processing Unit (CPU) and Graphics Processing Unit (GPU) usage during basic tasks. In order to assess the overall maturity of the Flutter framework, we also look at missing features regarding desktop support.

The Web and its evolution are a prerequisite for Flutter's and Electron's existence. Specifically JavaScript's popularity is the reason why Electron exists. For Flutter, we also consider the recent shift towards mobile platforms as the dominant factor for its development as Flutter initially focused on mobile devices as the primary platform of interest. We therefore start with background information about the Web and its core milestones in chapter 2. Next, we define Web technologies before we introduce Electron and Flutter. This is followed by a chapter on the current research situation regarding these frameworks, and the context in which they operate (see chapter 3). We then move on to the practical and experimentation chapters of this thesis. In chapter 4, we consider the functional requirements of our reference application before we describe both implementations and their specifics. In chapter 5, we list our test system and test data, before detailing our testing process. The results and evaluation of our experiments follow in chapter 6. We conclude with an overview about Flutter's packages and missing features (see chapter 7).

This thesis is aimed primarily at developers and secondarily at anyone interested in Flutter's current desktop state.

2 Background Information

Before making any direct comparisons between frameworks, it is important to understand the surrounding context in which these applications operate. We will therefore firstly consider the history of the Web, and key milestones in its history. We will then consider how exactly we define 'Web technologies', and how both of the frameworks fit into this definition.

2.1 Web

"The World Wide Web (WWW, or simply Web) is an information space in which the items of interest, referred to as resources, are identified by global identifiers called Uniform Resource Identifier (URI)" [Berners-Lee, Bray, et al., 2004]. It was invented by Tim Berners-Lee while working at the European Organization for Nuclear Research CERN in 1989. He also was the leading development force behind the three main technologies URL, HTTP, and HTML following long research projects in human-computer interaction, information retrieval, internetworking and more, as pointed out by Aiello [Aiello, 2018].

The Uniform Resource Locator (URL) is a special type of URI that goes beyond just identifying a resource, and also describes its primary access mechanism [Berners-Lee, Fielding, et al., 2005]. Hypertext Transfer Protocol (HTTP) is a request-reply protocol on the application layer of the Open Systems Interconnection model (OSI model). Opening a page identified by a URL in the Web browser typically triggers the browser to send a GET request to a server which then responds with a status, a text describing the status (reason) and possibly the requested resource (body) [Aiello, 2018]. Lastly, Hypertext Markup Language (HTML) is a language for documents that allows to classify the content of a document (title, header) in addition to describing how contents should be displayed [Aiello, 2018]. While HTML was initially designed for text and images, the latest iteration HTML5 also has elements to facilitate the embedding of audio and video.

The combination of these three "simple yet [. . .] effective" [Aiello, 2018] technologies and the Internet formed the earliest version of the Web as we know it today. Often, the term Internet is used interchangeably when referring to the Web. In fact, the Internet describes the network of networks and devices that exchange data using the TCP/IP protocol. The Web then uses the Internet for transporting the data.

Year	Description
1994	Cookies
1995	Java
1995	JavaScript
1998	Web Services

Table 2.1: Big changes to the Web based on [Aiello, 2018]

2.1.1 Changes to the Web

While the Web in its core has since remained, as a space to exchange information, it is nonetheless very different from when it was first introduced. The open nature of the Web, in addition to the growing user base, led to considerable changes. Aiello refers to these changes as ‘patches’, “as they are attempts to fix the original design of the Web” [Aiello, 2018]. Table 2.1 shows the most significant changes to the Web from today’s perspective.

Cookies

The first big change was the introduction of Cookies. Due to the stateless nature of HTTP requests, no page can access information about the pages that preceded or followed that page - unless the information is stored somewhere. While the information can be stored at the server or in the URL, a better solution is to store this type of information on the client itself. Cookies allow exactly that.

Today, cookies are widely used for storing stateful information like session and authentication data. Cookies are also used to track the user’s browsing history [Aiello, 2018].

Java

The second major patch to the Web was the need to perform client-side calculations, that only became possible with the invention of Java and its Web target. Java code could be integrated into Web pages through so-called Applets. These Applets also required a Java Virtual Machine (JVM) on the client’s machine. Applets ran in a sandbox with limited access to the client’s hardware [Aiello, 2018].

With this addition, it became possible to execute code on the client’s machine within a dedicated area of a Web page. Applets were used for resource intensive visualizations [Calegari, 2013], games [Hunter, 2009] and interactive applications, such as remote consoles [Audriusa, 2009].

Major browsers no longer support Java Applets [Oracle, 2021a,b]. Instead, JavaScript has replaced many of Java Applet’s functionalities. In recent years, another approach for high-performance applications on Web pages has emerged: WebAssembly (see section 2.4).

JavaScript

Around the same time as the emergence of Cookies and Java, another patch was crafted. A scripting language called LiveScript, that was later renamed to JavaScript, was integrated into the browser, giving Web pages the ability to dynamically change the content displayed. To begin with, JavaScript was unstandardized, leading to incompatibility across browsers.

While being “a horrible example of an object-oriented language” [Aiello, 2018], JavaScript was well-received due to its simplicity and browser support. Additionally, Asynchronous JavaScript + XML (AJAX) made it possible to load data and page content dynamically and, as the name suggests, asynchronously onto a page.

JavaScript is crucial for building Web applications today. It can be used in the database (MongoDB), the server (Node.js) and the browser. Additionally, JavaScript Object Notation (JSON) is widely used to exchange data between clients and servers [Aiello, 2018].

Web Services

While the first three patches focused on improving the user experience for humans, the last patch was necessary to improve the Web for machines and computers. The Web was no longer considered a space for human-readable documents, rather it was considered a space “of programmatically accessible nodes” [Aiello, 2018]. This was made possible by a standardized language, Web Services Description Language (WSDL), and by decoupling “clients of objects from their containers” [Aiello, 2018]. Additionally, the growing availability of Internet connections and nodes led to a new way of writing systems.

In recent years, especially the Representational State Transfer (REST) manifestation of Web Services in combination with JSON proved to be a good solution. Without the need of a complex interface description, REST is well suited for multimedia or streaming applications because of its performance advantages [Aiello, 2018].

2.1.2 Web Technologies

The term ‘Web technologies’ is used to refer to a whole collection of frameworks, protocols, languages, and formats used in the context of the Web. Many of these were specifically built for the Web including the aforementioned HTTP, HTML and JavaScript. Others were initially not designed with the Web in mind, but changed course because of the Web’s potential.

Java for example was originally designed as an alternative to C/C++ [Southwick, 1999]. Later, the designers added the Web as a target platform for Java. Today, Java can still be considered Web technology. It may very rarely be used on the front-end side as an Applet, but it is still a reliable choice for back-end systems and databases [Jetbrains, 2020]. Another Web focused language is PHP. Like JavaScript, development began in 1995. Today, many popular systems are powered by PHP including WordPress and Moodle [Hills et al., 2014].

Flutter can also be considered Web technology to some degree. It uses Skia for rendering, an open source 2D graphics library which also is used by Chrome and Firefox. Additionally, Flutter can be used to build Web applications by compiling Dart to JavaScript. Google uses Flutter to power the Web and mobile app of Google Ads [Flutter, 2021a; Google, 2019].

2.2 Electron

Simply put, Electron is a framework to create cross-platform desktop applications with Web technologies like Cascading Style Sheets (CSS), HTML and JavaScript. It was developed in 2013 by Cheng Zhao at GitHub as 'Atom Shell' as a native layer for the Atom text editor [Zhao, 2013] and was later renamed Electron in 2015 [Sawicki, 2015]. Many commonly used applications are written in Electron including Microsoft Teams [Maguire et al., 2020], Facebook Messenger and the very popular text editor Visual Studio Code [Electron, 2021a].

2.2.1 Architecture

Electron bundles the Chromium Content Module and the Node.js JavaScript runtime as seen in Figure 2.1, each complementing the other module's missing functionality to form a desktop application tool-kit.

The Chromium Content Module is part of the Chromium project, which itself is the open-source project many modern Web browsers are built upon, including Google Chrome and Microsoft Edge. The module is "the core code needed to render a page" [Project, 2021]. It includes Web platform features, GPU acceleration and the V8 JavaScript engine. However, it does not include any other browser feature.

The introduction of JavaScript in 1995 allowed developers to create Web pages that respond to user input immediately without loading another page. Based on the success of JavaScript, several attempts were made to bring JavaScript to the server. One of the most popular is Node.js. Initially released in 2009, the open-source cross-platform runtime environment uses the V8 engine to interpret JavaScript, just like the Chromium Content Module. Node.js also adds modules for

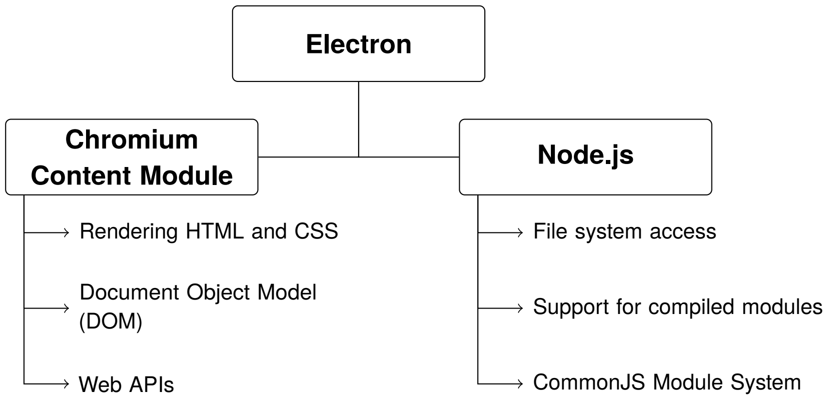


Figure 2.1: Electron's architecture based on [Kinney, 2018]

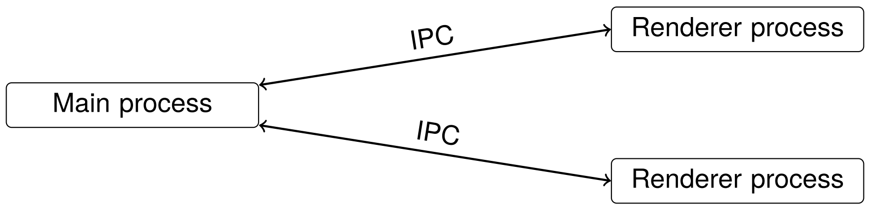


Figure 2.2: Main and renderer processes in Electron based on [Griffith et al., 2017]

working with the underlying operating and file systems, in addition to threading capabilities and support for working with native code [Node.js, 2021]. Node.js also comes bundled with the `npm` package manager, giving developers access to a large catalog of external libraries [Kinney, 2018].

2.2.2 Functional Principle

In order to understand and design an Electron application, it is important to understand the multi-process architecture of the framework. Each Electron application has a main process, and zero or more renderer processes as shown in Figure 2.2. These processes communicate using the Inter-Process Communication (IPC) modules `ipcMain` and `ipcRenderer`.

As the name suggests, a renderer process is responsible for rendering the displayed content. By using Chromium's multi-process architecture each renderer spawns a new thread. Features such as accessing the clipboard are available to both processes, while other functionalities such as working with the file system, are only possible in the main process. The main process also is responsible for the application startup and shutdown routines in addition to creating the application's `BrowserWindow`. Additionally, the main process manages access to the underlying operating system. All renderer processes are controlled by the main process as well [Electron, 2021b].

Both types of processes make use of the IPC modules. Listing 2.1 and Listing 2.2 show how the communication between main and renderer process might look like.

```typescript
import { ipcMain, dialog } from 'electron';

ipcMain.handle('select-file', async (event, ...args) => {
    return dialog.showOpenDialog({properties: ['openFile']});
});
```

Listing 2.1: TypeScript - Registering a channel in the main process

```typescript
import { ipcRenderer } from 'electron';

ipcRenderer.invoke('select-file')
    .then(response => response.canceled ? Promise.reject('No file selected') : response.filePaths[0])
    .then(file => {
        console.log(file)
    })
```

Listing 2.2: TypeScript - Calling a channel in the renderer process

Listing 2.1 needs to be executed as part of the application's startup procedure. In this case we register a function on the channel `select-file`. Later, when our application calls it – in this case by executing the `invoke` function in Listing 2.2 – the main process invokes our previously registered handler.

When it comes to developing an application, it does not differ much from a Web application. The User Interface (UI) can be designed in several ways. It is possible to use frameworks like Angular, React and Vue for the application or plain HTML, CSS and JavaScript.

2.3 Flutter

Flutter is an open-source UI tool-kit that has been developed by Google since 2017. It seeks to build natively compiled applications for multiple platforms from a single codebase. While Flutter initially focused more on mobile operating systems, it has now branched out to supporting desktops as well. With Flutter's last major update, the Web target was declared stable and desktop support moved to beta [Sells, 2021]. Despite Flutter being young, more than 150,000 apps were created in the last two years including: Google Ads, Google Stadia and Philips' Hue Bluetooth and Hue Sync apps [Flutter, 2021a]. Recently, Canonical, the company behind Ubuntu, made Flutter the "default choice for future desktop and mobile apps created by Canonical" [Flutter, 2021d] while Toyota aims at using Flutter for their vehicle infotainment systems [Flutter, 2021d].

2.3.1 Architecture

Figure 2.3 shows Flutter's layered architecture. The Embedder layer represents the code closest to the hardware and handles rendering surfaces, input, the event loop and more. Depending on the platform, it is written in either C++ on Android, Windows and Linux or a combination of Objective-C and Objective-C++ for iOS and macOS.

Text layout, network and file I/O are implemented in the middle layer – the Engine. It is "the low-level implementation of Flutter's core API" [Flutter, 2019] which also includes the Dart runtime and plugin architecture. Additionally, on this layer the rendering is implemented for which Skia is used.

Developers will directly use tools from the Framework layer to write their applications. This includes Widgets, Animations and Gestures. The Flutter framework also comes with the `pub` package manager which can be used to add dependencies to a project. Because the framework is deliberately kept small, even for basic operations like working with file paths or HTTP requests, plugins are required.

When targeting the Web, Engine and Embedder are replaced by a special Browser layer. Part of the upper layers will be compiled to JavaScript, and remaining tasks are handled by the browser. For rendering, there are two different approaches: the HTML or WebGL path. While the former uses HTML, CSS and Canvas to get the smallest code size package, the latter "provides the fastest path to the browser's graphics stack" [Flutter, 2019].

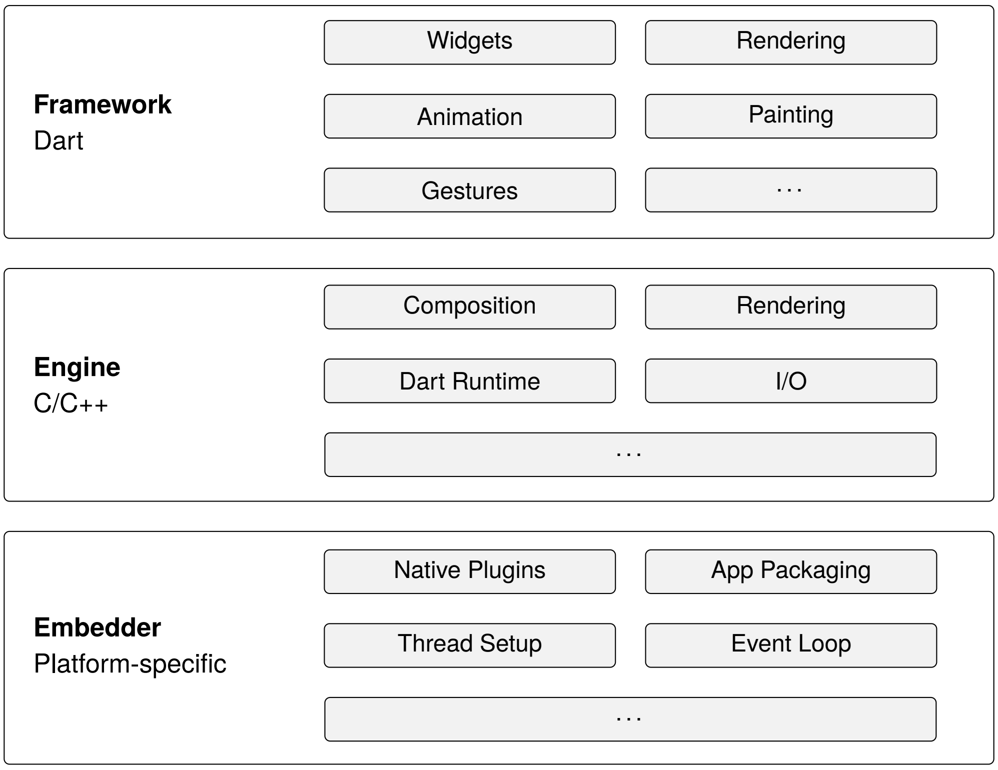


Figure 2.3: Flutter's architecture [Flutter, 2019]

2.3.2 Functional Principle

In contrast to Electron and traditional Web applications, no HTML, CSS, or pure JavaScript is used for the application development. Instead, every component must either be a `StatelessWidget` or a `StatefulWidget`. These widgets are then merged into a tree structure which ultimately results in the application's interface.

```
class MyHomePage extends StatefulWidget {
  MyHomePage({Key key, this.title}) : super(key: key);
  final String title;

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

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

  @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),
      ),
    );
  }

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

Listing 2.3: Dart - Shortened version of Flutter's sample code

The code displayed in Listing 2.3 is part of the boilerplate code when creating a new project. The application features a button as well as a text field which contains how often the button has been pressed. As we are changing the state while pressing the button, a stateful widget is used for the implementation. Every time we press the button, the `setState` function in the `_incrementCounter` method triggers a rebuild of the state's widget by calling the `build` method. Figure 2.4 shows the tree structure the code produces. Contrary to Electron and Web development in general, formatting and even styling happens through a widget. The `Center` widget for example centers its child widget while the `Column` widget positions its children one below the other. Widgets also handle padding and responsiveness.

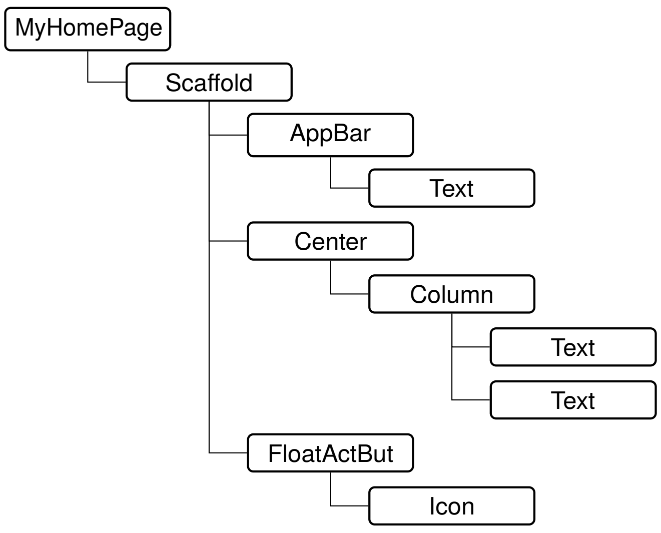


Figure 2.4: Tree structure produced by Flutter's sample code (Listing 2.3)

2.3.3 Dart

Dart is a programming language developed by Google for the development of mobile, Web, server and desktop applications. Bracha, a software engineer from Google who worked on Dart, describes the goal of the language with the following words:

> "Dart is intended to provide a platform that is specifically crafted to support the kinds of applications people want to write today. As such it strives to protect the programmer from the undesirable quirks and low-level details of the underlying platform" [Bracha, 2016]

In order to achieve these goals, Dart is an object-oriented garbage-collected language with a syntax inspired by the C programming language family. It uses a static type system with type interference. The design of the Dart programming language was mainly influenced by Smalltalk, Java and JavaScript. Erlang and C# also had an impact at the concurrency model Dart is using [Bracha, 2016].

```dart
class Person {

  final String name;
  int? age;

  // Person("John Doe", age: 30)
  // Person("Jane Doe")
  Person(this.name, {this.age});

}
```

Listing 2.4: Dart - Dart example

The code in Listing 2.4 shows some of Dart's features. The class `Person`'s constructor takes two arguments, the second being an optional named argument. As a result, the optional argument `age` needs to be explicitly declared nullable. It also requires constructor invocations to always name the argument given, and cannot be used as positional argument.

Originally released in 2014, the language reached version 2.0 in 2018 with the current version 2.12 being released in February 2021. A few of the most notable changes include a core library for calling native C code (Dart version 2.5), extension methods (2.7), a Dart command-line tool (2.10) and sound null safety (2.12) [Dart, 2021].

Dart also comes with a package manager called `pub` which allows managing dependencies and publishing packages to the pub.dev package archive. While there were plans to integrate a Dart VM into Chrome, these plans were dropped in favor of optimizing the compilation of Dart to JavaScript [Bak et al., 2015].

2.4 Related Technologies

Since Flutter and Electron target the desktop (see Figure 3.1), they also compete with traditional cross-platform desktop application tool-kits like Qt, GTK, or wxWidgets. These are mature frameworks with a wide variety of widgets and are available to various programming languages.

GTK applications can even be built with JavaScript. However, these frameworks fall much less strongly into the category of Web technology, if at all, as per our definition in the Web Technologies chapter.

On the other side of the spectrum, there are more recent developments such as Progressive Web Applications (PWA) and WebAssembly. PWAs are responsive, allow offline usage and work with any browser that meets the PWA requirements. While the feature set is getting closer to the scope of traditional desktop applications, support for working with the file system has only recently been added [Liebel, 2020]. "WebAssembly addresses the problem of safe, fast, portable low-level code on the Web" [Haas et al., 2017]. It is a binary format to execute programs as stand-alone or in browser. WebAssembly is not designed to replace JavaScript, but to complement it. While WebAssembly delivers on its performance promise, there are concerns about security [Lehmann et al., 2020].

We also want to mention Single Page Applications (SPA). When a website, which is build as a single page, is called for the first time, the page loads all required resources. Additional content is loaded asynchronously, if needed, but without reloading or loading another page. Thus, using the application feels more like using a traditional desktop application [Flanagan, 2006].

3 State of the Art

With Flutter targeting mobile, desktop, and Web platforms as illustrated in Figure 3.1, Flutter has points of contact and overlaps with these areas. Electron on the other hand focuses on bringing pure Web technologies to the desktop. In technical literature, not all three areas are well covered, especially in a broad-based comparison.

3.1 Related Work

In a study from 2016, however, Vassallo et al. provide an overview of cross-platform development approaches for mobile and desktop applications with a short excursion to Web applications. After listing some categories for mobile development like native applications, Web applications for mobile, hybrid applications, and more, Vassallo et al. compare the possibilities of the desktop space: cross-platform GUI libraries like Qt, GTK, and wxWidgets in addition to a Web-based approach. In the end, the authors recommend a Web-based approach to cross-platform application development including mobile and desktop applications in order to support as many platforms as possible. Other reasons to go for a Web-based approach are code reuse, and the responsiveness of modern Web applications [Vassallo et al., 2016].

Another exploratory study from Scoccia et al. focuses on Web frameworks for desktop applications, as “desktop Web app frameworks have not been studied empirically” [Scoccia et al., 2020]. The study aims to fill this gap by analyzing the usage of Web-based desktop applications including the impact of their pros and cons. In the study, Scoccia et al. found that Web applications are used in three categories in particular: Productivity, Developer Tools, and Utilities. Furthermore, Web application frameworks are often used by projects with a small number of core developers. The authors also found that reusing code written for Web applications in desktop applications is not only a good concept in theory, but is actually applied in practice with many desktop applications

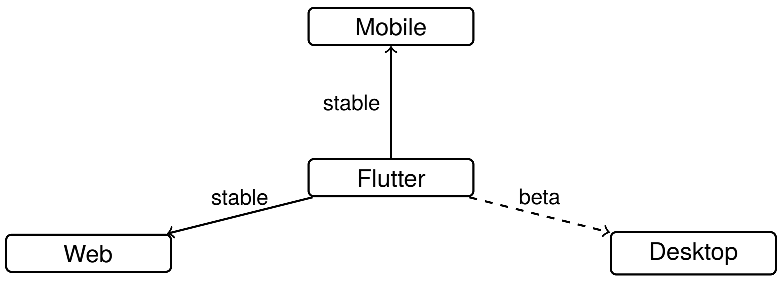


Figure 3.1: Flutter’s target platforms - May, 2021

using dependencies originally written for the Web. However, the study was based on only 453 open-source repositories available on GitHub. It is unclear to what extent the results can be applied to a larger data set and closed-source software [Scoccia et al., 2020].

In a practical approach to Web frameworks, Ivanova et al. introduce SPAs and PWAs before analyzing the three most popular front-end frameworks: React, Angular, and Vue. The comparison features directives and components, programming languages, templates, libraries, performance, testing, and learning curves. The authors decided to use Angular in the end mainly for being a full-featured framework, in addition to having good documentation and enterprise support [Ivanova et al., 2019]. However, it is difficult to make a general recommendation, especially because frameworks and tools develop quickly and dynamically.

In terms of comparing cross-platform desktop application development frameworks, Abeer Alkhars chose to compare Electron with JavaFX. The author starts with an introduction of both of these frameworks. For the comparison, Abeer Alkhars decided to focus on programming language, database access, availability of widgets, availability of APIs, and performance. For the performance measurement, Abeer Alkhars wrote the same application in Electron and JavaFX. Electron wins the performance comparison over JavaFX by being faster while using less memory. However, two separate tools were used for the performance measurements – Netbeans IDE for JavaFX and Chrome DevTools for the Electron application – and more test runs would have been desirable [Abeer Alkhars, 2017].

In the mobile area, Fentaw compared React Native with Flutter. The author wrote a reference application in React Native and Flutter, just like Abeer Alkhars did in the Electron / JavaFX comparison. Again, the performance of these reference applications was measured. Fentaw found, that Flutter had slightly better performance while the overall development was quite similar for both frameworks. However, in order to measure the performance, Fentaw executed the tests only 5 times. For more exact results, again, more runs would have been desirable [Fentaw, 2020].

Cheon et al. documented the process of porting a native Android app to the Flutter framework. While lessons learned are subjective, and the porting itself heavily depends on the app, the document also discusses the differences between Java and Dart. The authors found that their app written in Dart required about 37% less code than their app written in Java. However, the authors also noted that there is no official approach for global state management in Flutter. In addition, the authors complained about the library situation. Many of the libraries are limited in functionality and for the same task multiple libraries exist, which requires a decision-making process for each library. The author concludes that writing an app in Flutter leaves much to be desired because of "a lack of built-in, official, or sort of standard way" [Cheon et al., 2020] in Flutter [Cheon et al., 2020].

Lastly, Biørn-Hansen et al. present a different approach to the development of applications: PWA. Focusing on mobile apps, the authors developed three apps using different frameworks: a hybrid app with the Ionic Framework, an interpreted app with React Native, and a progressive Web application using React.js. In the end, Biørn-Hansen et al. see great potential in PWA, however, possible features of an app are ultimately limited by the wrapping browser [Biørn-Hansen et al., 2017].

3.2 Contribution

Cross-platform desktop application development tool-kits have not received a lot of coverage in the academic field in recent years. With Flutter now enabling support for desktop application development, we expect coverage to increase. We think that Flutter may be a viable choice for desktop application development where previously only Qt and Electron would have been taken into consideration. Therefore, we want to raise awareness for this cross-platform tool-kit in the academic field by looking at the current feature set and performance of the – in our opinion – promising framework.

4 Developing the Reference Application

In order to be able to make meaningful statements about the Flutter and Electron frameworks, we implemented an application in both frameworks. We present the basic design including mockups of the user interface in section 4.1.

Since the desktop target of Flutter was still in alpha status at the time of implementation, we started to implement the application with Electron first. Later, we ported the application from Electron to Flutter. Other from rewriting the user interface, we kept internal changes to the architecture to a minimum. We cover the Electron implementation in section 4.2 and the Flutter implementation right after in section 4.3. Although both frameworks are cross-platform, we opted for Linux as the target platform because of the developer and measurement capabilities.

4.1 The Reference Application

We decided to implement a basic dictionary application with an interface inspired by Material Design. The application has a simple structure: We have a *main view*, which is displayed when a user opens the application. Next, we have a *detail view* to display an entry's detail information and lastly, we have a *settings view* where the user configures the application. Figure 4.1 shows the initial mockup of the application's main view when the user searches the word list. In the finished

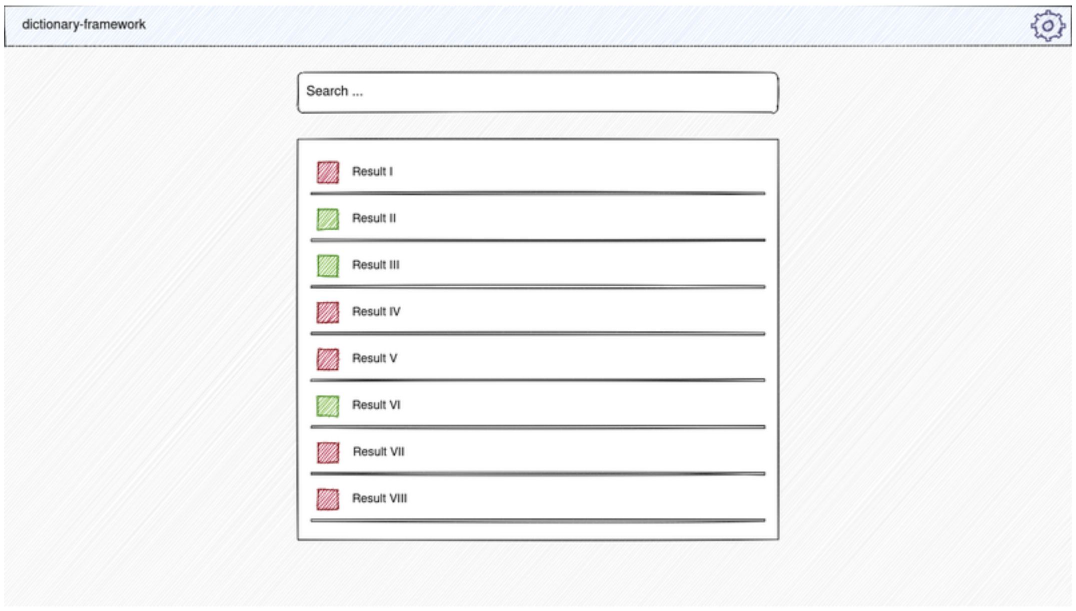


Figure 4.1: Mockup of the main view

application, the red and green areas will be replaced by national flags. As for the entries, we were planning to use a SQLite database. Because we want to keep it simple, we only support at most one word list.

4.1.1 Features

We settled on two main features: importing a list of words and searching it. The former is a stress test for the CPU and system memory during parsing, processing and inserting into the database. The latter will put the GPU to the test through user behavior such as scrolling. The following sections describe these features based on their influence on the design of the application.

Feature A: Import a Word List

If no word list has been imported, users will see a hint on the main page that they need to import one. The hint also includes a button to quickly go to the import wizard. The import consists of four states: *Initial*, *Parsing*, *Inserting* and *Imported*. The import is in the Initial state when the user has not imported a word list yet. In contrast to the initial one, there is the Imported state, which means the user has already imported a word list. Parsing describes the state where the application processes and parses the word list. Inserting means that the entries are currently being inserted into the database. Users are kept up to date with the current status. Additionally, there exists a fifth state, the *Error* state. The only difference between Error and Initial state is the displayed message. Lastly, users should be able to clear the database, if a word list was imported successfully.

As for the word list, we came up with the following requirements: The first line contains the source and target language in the form of ISO 3166–1 alpha-2 codes separated by a minus. A valid first line for example is `US-DE`. Every additional line in the word list must either be blank or follow the entry schema: word/phrase in the source language, tab character, word/phrase in the target language, tab character, optional part of speech. If one line does not satisfy the rule, the whole document is considered invalid, and the error state is assumed.

Feature B: Search the Word List

Only when a word list has been imported, the user should be given the possibility to search it by typing into the search field. The user's input should be processed in the following way: 300 milliseconds after the user typed in the last character (debounce time), the application checks if the query meets the length requirement which is two or more characters. If that is the case, the application searches for phrases that start with the query in the source or target language. Then, the application sorts the results based on the Levenshtein distance between phrase and query. If a phrase cannot be found, the application shows the user a hint. While searching, a circular progress indicator will be displayed. Searching the word list must also be possible on the detail view of an entry.

When the user selects a search result, the application triggers a second query. Depending on the language of the selected result, the application looks for matches that contain the selected phrase in either source or target language.

4.1.2 Packaging

Packaging and distribution of desktop applications heavily depends on the targeted architecture and operating system.

On Apple's macOS, the primary distribution platform is the App Store. Developers build and test their applications before they submit them for review. After passing the review process, developers are free to release the applications to the public. Although Microsoft has also tried to set up a central platform for application distribution on Windows, developers tend to prefer the traditional way: Users are required to visit the developer's website to download an installer or a single-file runnable application.

On Linux, users typically install software from their distribution's repositories with the help of a package manager. Because dependencies are also managed via the package manager, installing the latest version of a software might be difficult. Therefore, alternative distribution methods like `Snap`, `Flatpak` and `AppImage` gained more and more attention in recent years especially for graphical desktop applications. Applications offered by these alternative distribution methods bring their dependencies with them. For the developer, this has the additional advantage that the application does not have to be packaged separately for each Linux distribution. Users can now install multiple versions of a software side by side without being limited to software that is available in their distribution's repositories. Additionally, applications installed by these alternative distribution methods run in a sandbox. The biggest disadvantage of these alternative distribution methods is the package size.

We chose AppImage because it is very easy to package applications with it. Additionally, we do not plan to distribute our applications.

4.2 Electron Implementation

For the most part, writing an Electron application is very similar to writing a Web application. This is especially true when it comes to the user interface. For this reason, we will call the presentation layer *front-end* and everything else *back-end* in this chapter including the database module.

Because we had no experience with either React or Vue and only limited experience with Angular, we decided to use the latter to implement the front-end. We wanted to write the back-end in TypeScript, and with Angular's first class TypeScript support we were able to write the whole application in one language. We started with an Angular Web application. Then we transformed it into an Electron application with the back-end written in JavaScript. Later, we replaced the JavaScript with TypeScript. From this point on, further development corresponded to the development of a Web application including the ability to develop the user interface in the browser and rebuilding on file change. However, this was only true until we started using Electron specific functions, because this required the presence of ipcMain and ipcRenderer (see subsection 2.2.2 Functional Principle).

For the back-end, we adopted the principle of services from Angular. The Angular documentation states, that services are software components "with a narrow, well-defined purpose. It should do something specific and do it well" [Angular, 2020]. We defined a service in the back-end for each

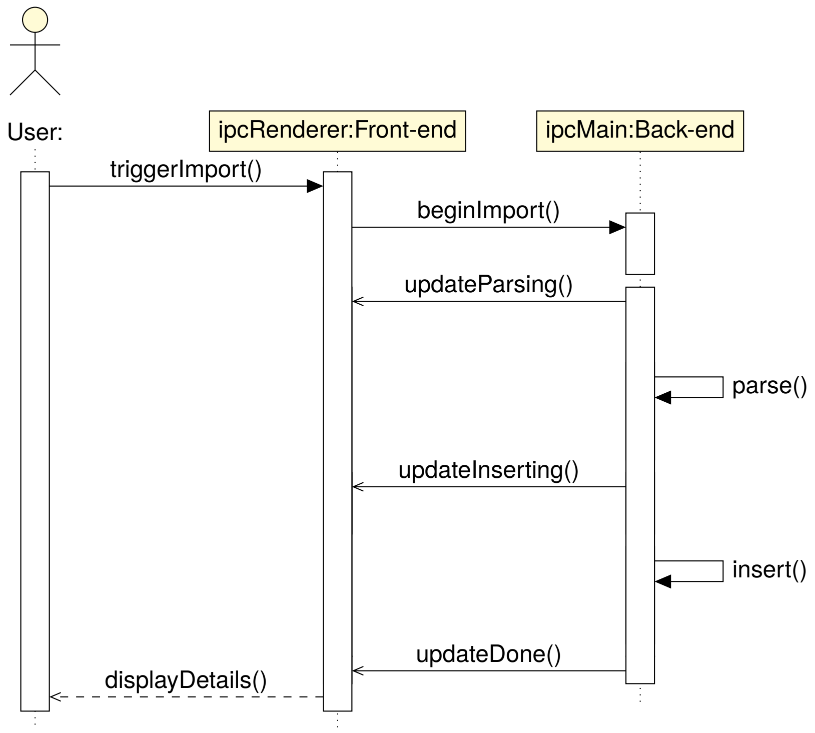
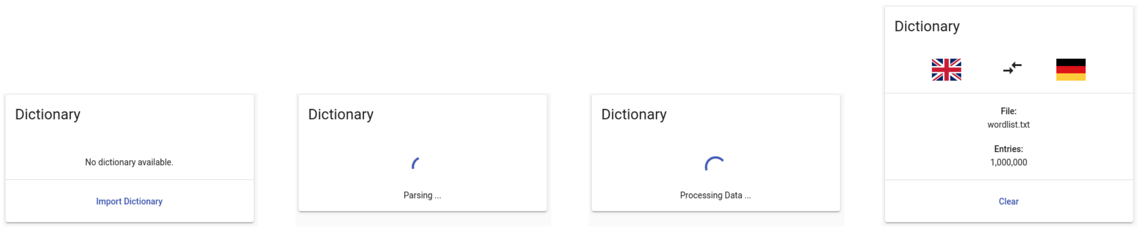


Figure 4.2: Communication between front-end and back-end during import



(a) Initial **(b)** Parsing **(c)** Inserting **(d)** Imported

Figure 4.3: States during import

IPC-channel we used for the communication between front-end and back-end. In our application, the back-end only reacts to changes in the front-end which are ultimately triggered by user behavior. For this reason, all back-end services behave like listeners that are registered at application startup.

Next, we want to look at what is probably the biggest difference between the Electron and the Flutter implementation apart from the different approach to writing the user interface: the import process. The reason for this is once again the separation between back-end and front-end. Figure 4.2 shows the message exchange between ipcRenderer in the front-end and ipcMain in the back-end. Figure 4.3 displays the resulting changes in the user interface. We already showed in listings 2.1 and 2.2, how such a communication might look like at code level.

We start off in the Initial state **(a)**. The user triggers the import with the selection of a file, which we omit in Figure 4.2 for simplicity. The front-end then triggers the import in the back-end. The back-end immediately sends an update to the front-end, that it has begun parsing the file. This puts

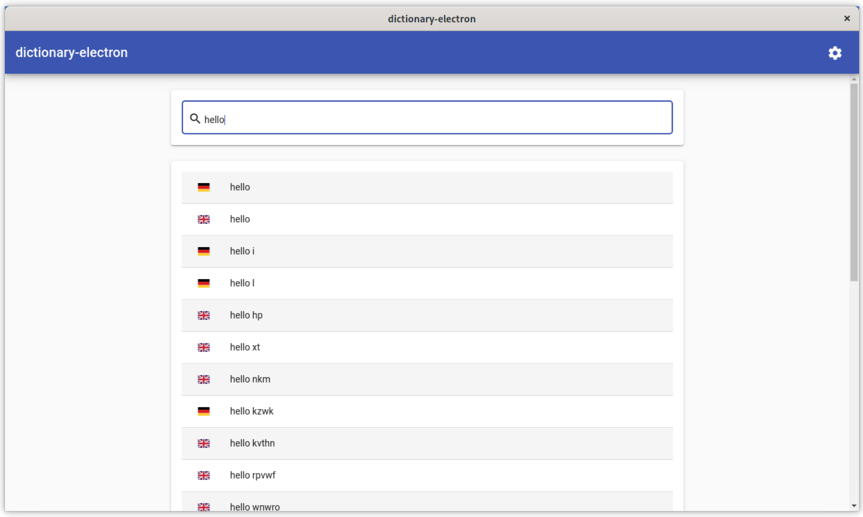


Figure 4.4: Main view in Electron

the import in the Parsing state **(b)**. After the parsing is done, the back-end sends another update to the front-end, which causes the import to change to the Inserting state **(c)**. If nothing went wrong during the parse and insert steps, the back-end sends a last update to the front-end **(d)**.

Overall, we used ten dependencies for the Electron implementation including libraries to work with database and application configuration files. We also used dependencies for embedding national flags and to package our application as an AppImage.

During the implementation phase, we did not experience any surprises as we should expect from a mature framework. Figure 4.4 shows the main view of the Electron implementation while searching for the word `hello`. We describe the generation of the test data used in section 5.2. Appendix A contains more screenshots of the application in the form of a comparison between Flutter and Electron.

4.3 Flutter Implementation

Developing a Flutter application is much more like developing a desktop application than a Web application. While of course Web applications are also possible with Flutter, the response time from changing a line of code to seeing its effects is noticeably less than what we experienced when working with Electron. We also do not have a separation between front-end and back-end by design, which also brings it closer to traditional desktop application tool-kits.

Instead of front and back-end, we have user-interface code and non-user-interface code. As described in the introductory text of chapter 4, we ported the application from Electron to Flutter. We kept the general idea of services that contain the functional bits of our application. Also, the database module almost is a one-to-one copy from the Electron implementation including the queries which we wrote as raw queries. As a replacement for dependency injection Angular has built in, we made use of the dependency `getIt` to achieve something similar.

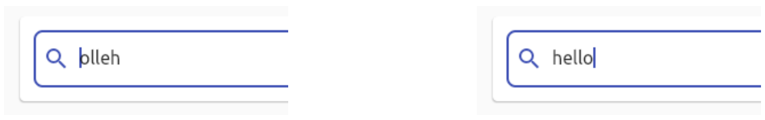
```
1  @override
2  Widget build(BuildContext context) {
3    var results = getIt<SearchService>().searchResults$.valueWrapper.value;
4    return ListView.builder(
5        scrollDirection: Axis.vertical,
6        shrinkWrap: true,
7        itemCount: results.length,
8        itemBuilder: (context, index) {
9          var result = results[index];
10         return Container(
11           color: (index % 2 == 0) ? Colors.grey[200] : Colors.white,
12           child: ListTile(
13               leading: Text(result.flag),
14               title: Text(result.display),
15               onTap: () {
16                 getIt<DictionaryDetailService>()
17                     .loadDetails(result.display, result.isSource);
18                 getIt<SearchDetailToggleService>().showDetailPage();
19               }),
20         );
21       });
22 }
```

Listing 4.1: Dart - Construction of the search result list

In our application, the global state is managed by services. We can see that in Listing 4.1. The code is responsible for constructing the search result list after the user searched for a word. We get the search results from a global service called `SearchService` in line 3 which manages the current state of the search including whether the database is being queried at the moment, whether the search is active, the current search query, and the results. The listing also shows two other services in action: `DictionaryDetailService` in line 16 and `SearchDetailToggleService` in line 18. The former loads the detail data when the user selects a result, the latter loads the detail view. However, we do not believe that this type of state management scales well for large applications. We would therefore recommend moving services into better encapsulated modules or taking a different approach to global state.

Listing 4.1 also shows how lists in the user interface can be processed in Flutter. The `itemBuilder` function in line 8 gets called for each list element to build the element view. Here we use alternating row colors for the list elements. The predefined `ListTile` widget used in line 12 makes it easy to define the layout for each individual result. In contrast to the Electron implementation which used Scalable Vector Graphics (SVG) for the national flags, we use pure text in Flutter (line 13).

However, this became a problem when we tried to package our application. At the time of writing, Flutter only supports packaging Snap applications on Linux officially. Therefore, we packaged our application manually, but ran into problems with embedding fonts. As a result, when we run our AppImage on a second machine, national flags may not work. Another point related to packaging is that we had to hardcode a path for the database due to the sandbox restrictions of AppImage. As we expect Flutter to offer additional distribution methods other than Snap in the future, we have not pursued our packaging problems further.


plleh

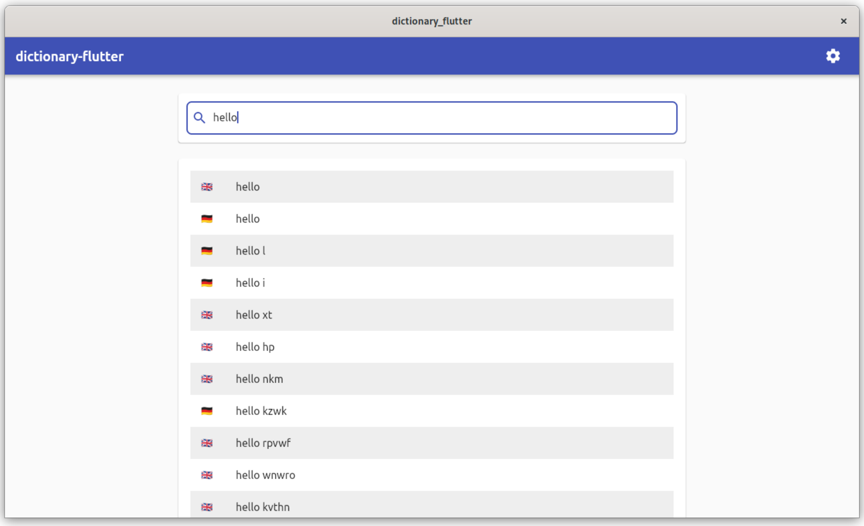
hello

(a) Wrong insertion order **(b)** Fixed insertion order

Figure 4.5: Insertion order in TextFormField

We did not only experience problems during packaging, development also was not as smooth as it was with Electron. While implementing the search input field, we noticed that the character insertion order was reversed (see Figure 4.5). After some research, we found out that the bug was possibly caused by GTK, the tool-kit used to wrap Flutter on Linux [Github, 2021a]. As for performance, we noticed a significant performance difference between `development` mode and `release` mode when working with our test data (see section 5.2 Test Data). In development mode, parsing took about 3.3 seconds, while in release mode the same task took about 8.3 seconds. The time for the insertion remained roughly the same with about 30 seconds for both development and release mode. Additionally, we observed significant frame drops during parsing in both modes. For some functionality in our application such as file selection during import, we had to import modules that were not yet final. We discuss this topic in chapter 7 Flutter's Feature Completeness in more detail.

Overall, we are still quite happy with the experience during implementation. While not everything feels polished at the moment, we think Flutter is on the right track.


dictionary_flutter

dictionary-flutter

hello

hello
hello
hello l
hello i
hello xt
hello hp
hello nkm
hello kzwk
hello rpvwf
hello wnwro
hello kvthn

Figure 4.6: Main view in Flutter

Figure 4.6 shows the main view of the Flutter implementation while searching for the word `hello`. We describe the generation of the test data used in section 5.2. Again, Appendix A contains more screenshots of the application.

5 Experiment Setup

With both applications developed, we were ready for our evaluation process. First, we will list the hardware and software of the test system in section 5.1. Then we will present the data we used to perform the tests in section 5.2. Finally, we will describe our approach to evaluate CPU, GPU, system memory, and startup time in section 5.3.

5.1 Test System

Hardware	**CPU:**	Intel Core i7-7700HQ @ $(4 \cdot 2) \times 3.8$ GHz
	GPU:	Intel HD Graphics 630
	System Memory:	16 GB DDR4 @ 2133 MHz
	Resolution:	2560×1440 pixels
Software	**OS:**	Arch Linux
	Kernel:	5.11.6-arch1-1
	Desktop:	GNOME 3.38.3 / Mutter / X.Org
	Graphics Driver:	Mesa 20.3.4-3

Table 5.1: Hardware and software of the test system

Table 5.1 shows the test system's hardware and software specifications. At the time of testing, Node was at version `15.11.0-1` and Electron on version `11.0.3`, while Flutter had just reached version `2.1.0-12.1.pre` in the development channel. On the day of testing, we updated all software components to their latest versions. We conducted our tests on a laptop connected to the grid with no external devices attached.

5.2 Test Data

We generated our own test data based on the requirements from subsection 4.1.1. First, we set the basic conditions. As described in subsection 4.1.1, each line consists of two phrases and an optional part of speech. Each phrase consists of at least one and at most five words. Each word, in turn, is between one and ten characters long. We first generated one million such lines based on these conditions. Then we agreed on the query `hello`. We randomly picked one entry and replaced either the source or the target phrase with the query. We did it because we wanted to be able to always select the same entry in our test scenario (see subsection 5.3.1). In the last step, we inserted the query into 99 other randomly selected entries by replacing one word in one of their phrases.

5.3 Test Method

In contrast to performance measurements for algorithms, time is secondary for our evaluation approach. Therefore, we have not measured time elapsed for a particular scenario or CPU/GPU time. Instead, we measured CPU, GPU, and system memory usage for the entire system.

5.3.1 Test Scenarios

Based on the two main features presented in subsection 4.1.1, we designed two basic scenarios which we summarize in the following:

- **Import:** We always started without any application data including an empty database. First, we went to the settings view and pressed the import button. Then, we selected the previously prepared word list (see section 5.2) and confirmed the import. After waiting for 50 seconds, we navigated back to the main view.

  In this scenario, we did not do much except waiting. However, this was a stress test for the CPU and system memory during parsing, processing, and inserting into the database.

- **Search:** In this scenario, we started with an already imported word list. After we started the application, we selected the input field on the main view and entered our query. We waited for two seconds before we selected the first result. Then, we waited for four seconds before we moved the mouse to the scrollbar. We scrolled all the way down, waited for one second, and then all the way up again. Lastly, we went back by pressing the back button below the search input field.

  In addition to two database queries executed here, we have rendering work for the GPU in this scenario in particular.

Additionally, we included a third scenario: the idle scenario. Like the name suggests, we waited for a specified time without doing anything while still measuring system usage. This allowed us to finally subtract the idle system usage from the usage during the actual scenario to get data specifically for our metrics.

5.3.2 Test Execution

In order to make any statements at all about CPU, GPU, or system memory usage, we have completely automated the operation of our application with the help of a Python script. For that, we used the `PyAutoGUI` module which allowed us to programmatically control the mouse and keyboard. We packaged our applications as AppImages built in release mode and moved them next to our script. After these preparations, actual testing began: First, we rebooted our system. Next, we opened up a terminal and navigated to the test directory. Then, we executed our script for one combination of scenario and framework.

We executed this procedure twenty times for each combination of framework and scenario including the idle one with system restarts between each execution. We present the results of these executions in chapter 6.

```python
def run(args):
    we = MeasureRun(args)
    we.check_preconditions()

    we.start_application()
    we.wait_for(seconds=3)

    we.begin_measurements()
    we.run_scenario()
    we.stop_measurements()
    we.wait_for(seconds=3)

    we.stop_application()
    we.wait_for(seconds=1)
    we.cleanup()
```

Listing 5.1: Python - Core logic of the experiment (actual code)

Listing 5.1 shows our high-level code for the test execution. First, we check in the `check_preconditions` method if all our utilities to measure CPU, GPU, and system memory (see subsection 5.3.3) are available. Then, we launch the application with a resolution of 1280×720 pixels. The `begin_measurements` method spawns separate processes with the aforementioned utilities, which log their output into separate files as long as the corresponding measure process keeps running. Next, we execute our scenario. After that, we stop the measurement processes before shutting down the application. Lastly, we perform cleanup operations such as removing the database and application's configuration data.

5.3.3 Used Metrics and Tools

For measuring hardware usage, a unified approach does not exist for all metrics required. The available tools depend on the hardware and operating system of the device tested. Because of this, we describe in detail which tools we used and how we measured. We settled on an interval of one second for all usage related measurements.

CPU Usage

CPU time describes how much time a CPU spends processing instructions [Foundation, 2021]. CPU usage on the other hand describes CPU time as a percentage of the total capacity of CPU [Wikipedia, 2021]. To measure CPU usage, we made use of the `mpstat` command line utility. Among a detailed report about how CPU time was spent for specific cores, this utility also reports the idle time of the CPU for all cores. Following the calculation from Chekkilla [Chekkilla, 2016], we subtract the idle portion from 100 to get the CPU usage. We used the following command to obtain the idle usage: '`mpstat -o JSON 1`'.

GPU Usage

Because there is no unified approach to obtain GPU usage in Linux yet, obtaining this data depends on the installed hardware and vendor. As for our GPU, we used the `intel_gpu_top` utility. We used the command ‘`intel_gpu_top -s 1000 -o output.json -J`’ to directly obtain the portion the graphics card was active.

System Memory Usage

In contrast to GPU usage, a unified utility exists for measuring system memory usage: the `free` utility. Besides reporting the total, available, or free system memory, the tool also reports the total memory used of a system. We used the ‘`free --mega -s 1`’ command to obtain this metric.

Startup Time

We measured the startup time during the execution of the import feature. The approach to measuring startup time is the same for both applications, but the implementation differs depending on the framework. We started measuring the earliest time possible and stopped after the main window was created. We did not automate this measurement, instead we used the frameworks’ timer capabilities to log the time. Then, we wrote down the startup time manually for each application start.

Size of the Application

The code style used plays a big part in measuring code length of the projects. For IntelliJ, which we used for Flutter, and WebStorm, which we used for Electron, we left the formatting rules at default. We then used `tokei` to count the lines of the source files. For Electron, we used the ‘`tokei -e package-lock.json -s code`’ command to exclude the automatically generated `package-lock.json` file. For Flutter, we used the ‘`tokei -s code`’ command.

As for the packaged size, we measured the size of the packaged AppImage with the `du` utility, and the command ‘`du -sh <AppImage file>`’.

6 Results of the Experiment

Before examining the results of the experiments in further detail, it's worth emphasising the way in which these results were obtained: the test system is explained in section 5.1, the test data in section 5.2, and the approach to measuring, in addition to the tools used, can be found in section 5.3.

Furthermore, it should be noted that the results do not necessarily reflect on how performant the frameworks are now. The results simply show the state of both frameworks for a given date for a specific hardware composition for a snapshot of a rapidly developing operating system. Nevertheless, trends are clearly discernible.

As far as measurements are concerned, the rough course of events for both frameworks is important to emphasise. This is especially true for the import scenario: A significant difference for the time these applications spent parsing and inserting was observed. The Flutter implementation chapter (see section 4.3), proves that time spent parsing and inserting differs quite significantly for development and release mode in an unexpected way. The same applies for the differences between Flutter and Electron. The Electron application parses the word list in less than four seconds and inserts the entries into the database in about 15 seconds. In contrast, the Flutter application requires almost 10 seconds for parsing, and over 30 seconds to insert entries into the database. In any case, this can also be attributed to the differences in implementation which we discuss in more detail in the CPU Usage section.

We performed all measurements on March 16, 2021.

6.1 CPU Usage

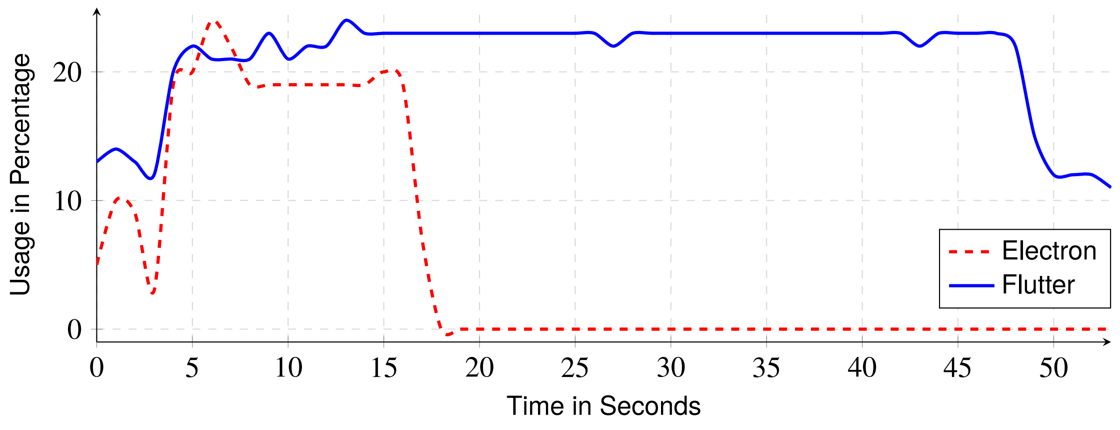


Figure 6.1: CPU usage during import

Figure 6.1 shows the CPU usage while importing. While starting with a lower usage, the Electron application peaks sooner at about 24% usage at the sixth second. The Flutter implementation has a slightly higher CPU usage during parsing and inserting. Additionally, the idle CPU usage is much higher as we can observe from the last three seconds where both applications are in idle state.

As for the time differences we mentioned earlier, we perform batch inserts in both applications. In the Electron implementation, this is a one-liner where a chunk size of 500 was specified. The Flutter dependency, which was used for the database access, requires a little more work in terms of implementation. With a specified chunk size of 500, the application took about a minute to insert entries into the database. With a much bigger chunk size – 10,000 and greater –, we were able to reduce the time to a little under 30 seconds for inserting. We finally went with a chunk size of 1,000 for the Flutter implementation as a compromise.

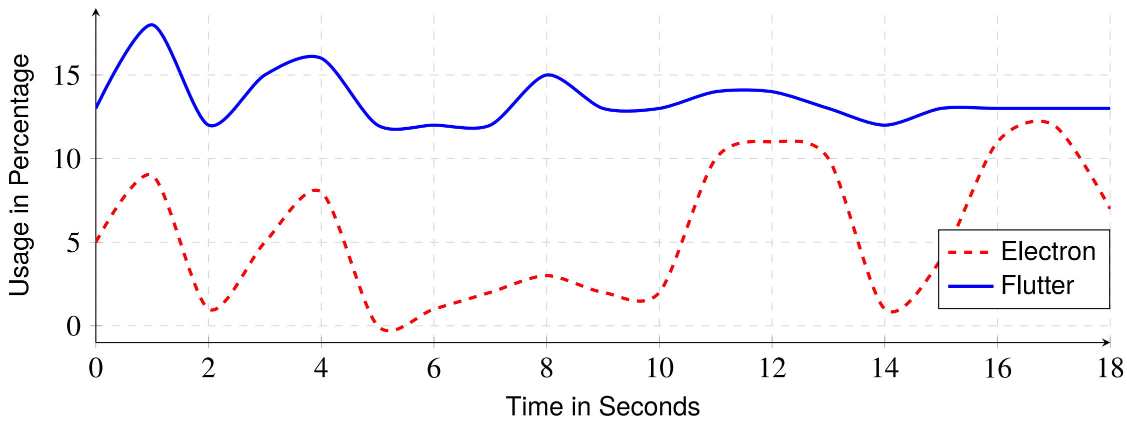


Figure 6.2: CPU usage during search

Figure 6.2 displays the CPU usage during the search scenario. Generally, we observe a higher CPU usage for Flutter with a peak usage of 18%. Both database queries are clearly identifiable for both frameworks until the fifth second - one for searching the query and one for finding detail information for the selected entry. The up and down scrolling was performed in the end of the search scenario and is only clearly visible for the Electron application with two small peaks between the 10 – 14 and 14 – 18 intervals.

In conclusion, considerably higher demands on the CPU were seen on Flutter on our hardware. Therefore, the findings indicate that Flutter's CPU demands are not fully optimized at the moment. This is compounded by the lack of effects, such as up and down scrolling in the end of the search scenario, and the high usage in general. The CPU usage of Electron is within expectations.

6.2 GPU Usage

Figure 6.3 and Figure 6.4 show similar findings in terms of Flutter's GPU usage. In both scenarios, GPU usage stays consistently above the 40% mark regardless of widgets, views, and animations to render. Additionally, scrolling the view in the end of the search scenario (Figure 6.4) is not reflected in the course of GPU usage values of Flutter.

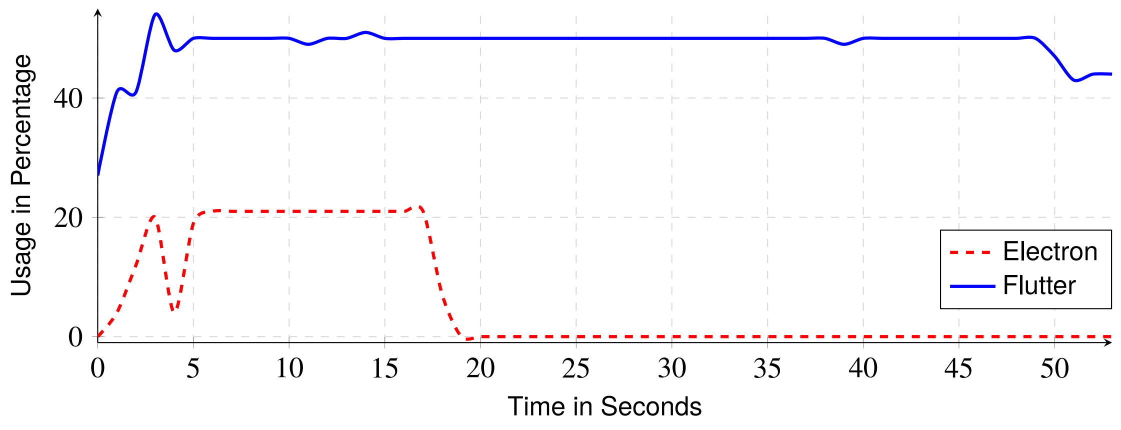


Figure 6.3: GPU usage during import

Conversely, Electron requires the GPU to render at about 21% during the import while displaying progress indicators (see Figure 4.3). During idle operation, Electron does not require any rendering, and the GPU usage settles down to 0%. Electron also shows no difference between rendering the search result list in the beginning of the search scenario and scrolling the view up and down in the end. In both cases, GPU usage stays at around 11% or below.

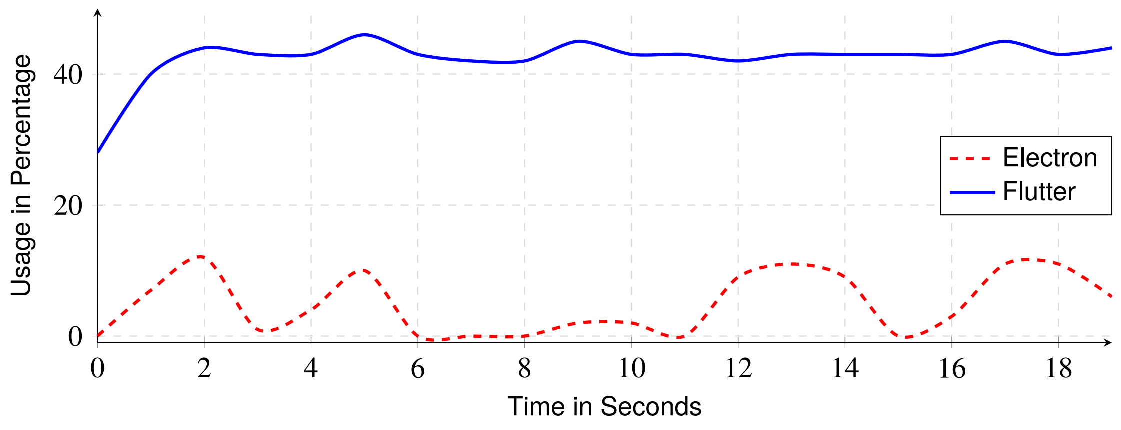


Figure 6.4: GPU usage during search

Based on the generally and consistently higher demands on the GPU by Flutter, in addition to frame drops during importing, considered in section 4.3, a very similar conclusion about GPU usage is reached. Flutter has not been very well optimized yet when it comes to GPU usage. Electron, however, does not show any weaknesses in GPU usage, as would be expected from a mature framework.

6.3 System Memory Usage

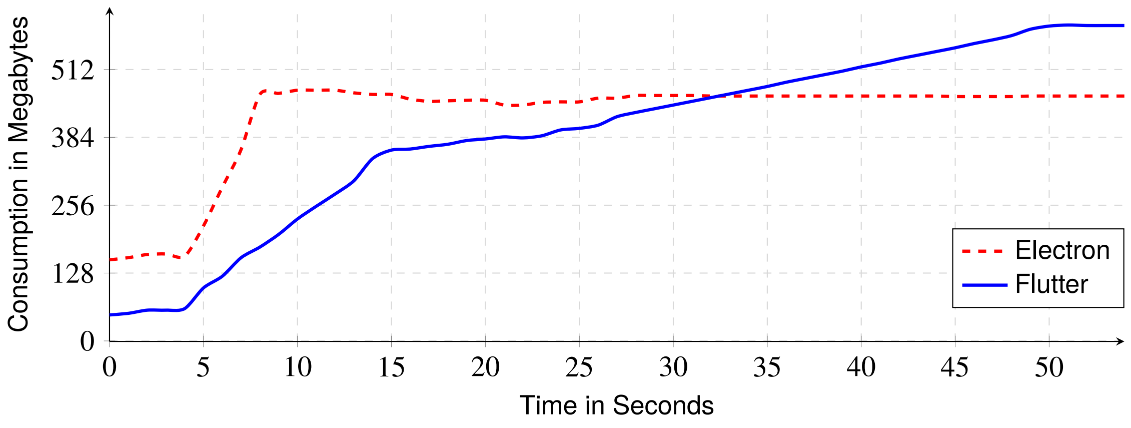


Figure 6.5: System memory usage during import

When it comes to system memory usage, Electron's browser base makes itself noticeable. In both scenarios (see Figure 6.5 and Figure 6.6), Electron requires more memory from the start. As for the parsing of the import, Flutter and Electron consume about the same memory as indicated by the steep increase. While inserting into the database, which starts at around the seventh second, Electron does not require more memory. In contrast, Flutter continues to demand system memory until the insertion is complete. Finally, the data shows a consumption of slightly less than 600 megabytes by Flutter.

However, the more realistic search scenario as seen in Figure 6.6 shows that, in fact, Flutter consistently stays below 100 megabytes, whereas Electron requires a lot more memory peaking at 162 megabytes in the thirteenth second.

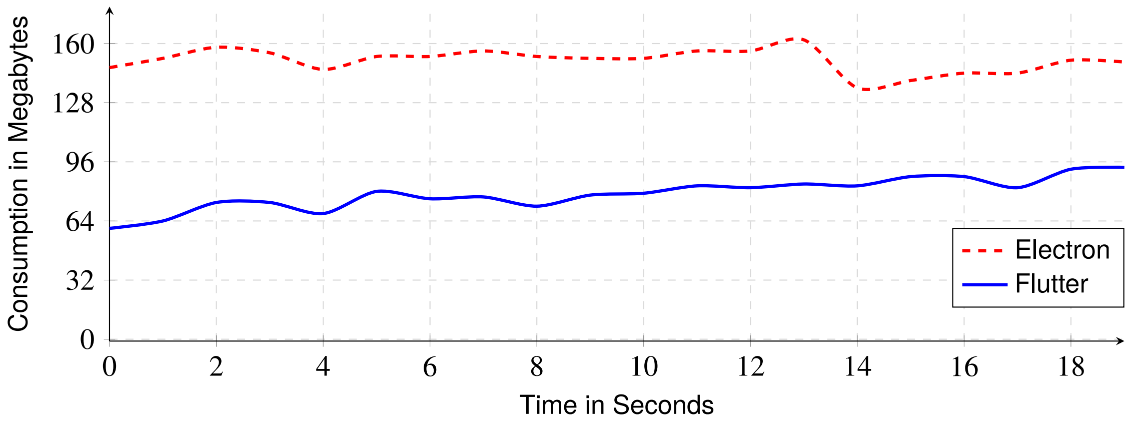


Figure 6.6: System memory usage during search

Since the main difference is in the course of values over time during import, it is assumed that the insertion into the database, and thus a dependency, is responsible for the high memory usage of Flutter. To confirm and possibly improve upon this, the library would need to be substituted. Overall, a confirmation of the criticism of Cheon et al. regarding the dependency situation (see chapter 3) can be seen.

In general, however, Flutter requires less system memory than Electron.

6.4 Startup Time

As for the startup time, Flutter is the clear winner as seen in Figure 6.7. Application starts are more than 11 times faster with Flutter than with Electron on average. After initial testing, the application startup time began to improve even further to about 40 milliseconds for the Flutter AppImage, one month after our initial testing. It should also be highlighted that the package method used, AppImage, has a big influence on application startup time as well. Running the binaries without packaging reduces the time to start for Electron to around 200 and for Flutter to around 10 milliseconds.

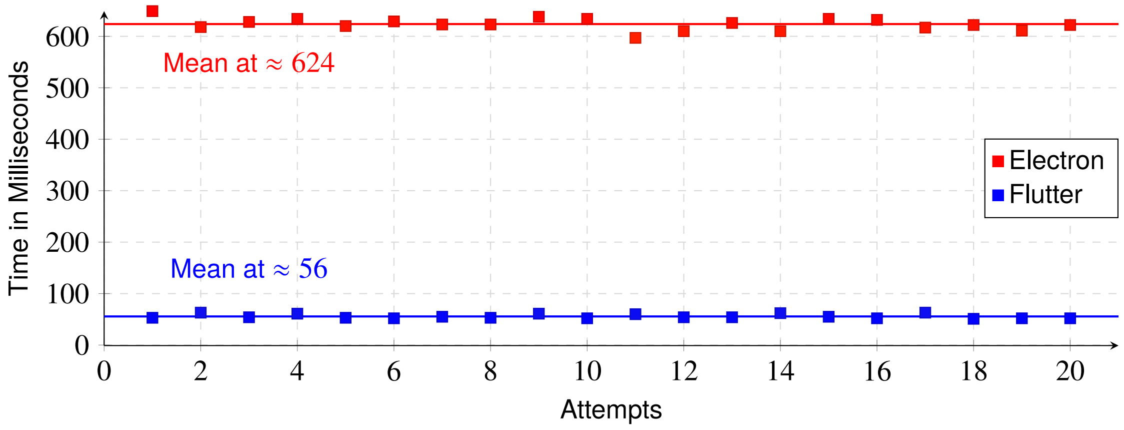


Figure 6.7: Application startup times

6.5 Application Size

Table 6.1 shows the code statistics of both projects. Ostensibly, it may seem that a lot of different programming languages are used. However, code was not required to be written in all of them. For the Electron implementation, we only wrote TypeScript, HTML, Sass, and CSS as seen in Figure 6.8's breakdown. As for the Flutter implementation, we used Dart exclusively (see Figure 6.9). If we only count the lines of code we wrote ourselves, we get 3,239 lines for Electron and 2,316 for Flutter.

As for the AppImage size we measured 166 megabytes for the packaged Electron and 9.1 megabytes for the Flutter AppImage. The Electron version needs to package almost an entire browser while the Flutter one only requires parts of the GTK tool-kit for embedding purposes.

Language	Files	Lines	Code	Comments	Blanks
TypeScript	136	3565	2730	109	726
JSON	9	468	468	0	0
HTML	16	416	335	1	80
Sass	9	118	96	1	21
CSS	7	89	78	0	11
JavaScript	2	69	58	5	6
Markdown	2	27	0	17	10
Total	181	4752	3765	133	854

(a) Electron

Language	Files	Lines	Code	Comments	Blanks
Dart	71	2708	2316	17	375
CMake	3	214	148	33	33
C++	3	128	91	14	23
YAML	1	84	30	42	12
C Header	2	31	12	11	8
Shell	1	14	8	1	5
Markdown	1	3	0	2	1
Total	82	3182	2605	120	457

(b) Flutter

Table 6.1: Code statistics of the implementations

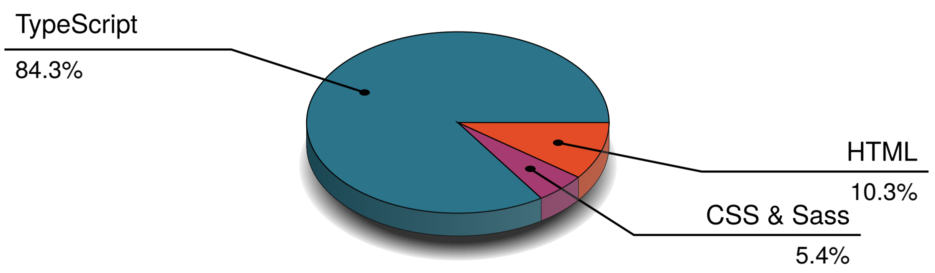


Figure 6.8: Code written for the Electron implementation

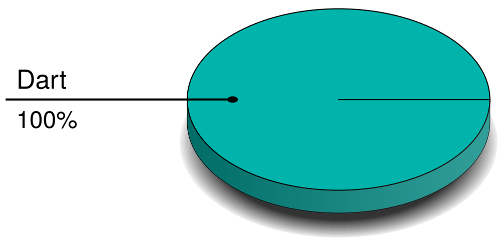


Figure 6.9: Code written for the Flutter implementation

7 Flutter's Feature Completeness

The decision for or against a UI tool-kit depends on whether the tool-kit can meet the minimum requirements of the application that is being developed. With Flutter initially targeting the mobile market, the focus was on meeting the requirements primarily for that platform. Although the desktop platform does have significant differences to the mobile platform, a large part of the requirements that may be needed for a desktop UI tool-kit were already covered when Flutter branched out to support the desktop platform. This includes features such as internationalization, working with application resources and HTTP requests. Differences and missing features therefore exist especially in the areas reserved for the desktop.

During our implementation of the reference applications (see chapter 4), any features which may be missing were not examined. In the following, we therefore present some missing features within Flutter which were encountered during our research. We start with the package situation, in which we examine the availability of the top 100 Dart and Flutter packages by platform. After that, we look at some widgets that are still missing. Finally, we list some generally missing features of the framework that are considered standard in other tool-kits such as Qt or GTK.

7.1 Packages

Figure 7.1 shows the availability of the top 100 packages for Dart and Flutter sorted by popularity. We did not count packages that were related to each other for example `path_provider` and `path_provider_windows`. As would be expected, all of these packages are available for Android and iOS. At the time of writing, macOS has the best support out of the three desktop operating systems. Packages that are missing for macOS are primarily developed for mobile devices such as `image_picker`, `permission_handler` and `device_info`. The other desktop operating systems, Windows and Linux, are almost on par. The only package out of our top 100 that works on Windows

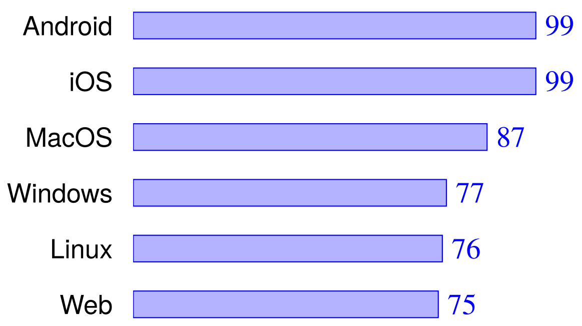


Figure 7.1: Availability of the top 100 most popular Dart packages by platform

and not on Linux yet is `wakelock`. It is a package to keep the device's screen awake. To close the gap to macOS, Windows and Linux mainly miss packages that are required to work with Google's Firebase. In general, the following rule applies: If a package is available on Linux, it is also available on Windows and if a package is available on Windows it is also available on macOS.

What the figure does not tell us about is the availability of libraries that serve a similar purpose. While having multiple libraries competing with each other is beneficial generally, it complicates the decision-making process for dependencies, which is especially true when we consider the fact that Flutter requires developers to use dependencies, even for basic operations such as HTTP requests. As an example, ten dependencies for our Flutter implementation were used. Another point of note is that some desktop related libraries are still under development. As a result, we were required to specify more than just the base dependency: for example `file_selector` and `file_selector_linux`. Other libraries required the addition of another dependency to work on the desktop, for example `sqflite` and `sqflite_common_ffi` for working with the database. We expect these minor issues to be resolved in the near future.

7.2 Widgets

Next, we would like to mention a few widgets that exist in other tool-kits but not officially in Flutter. We also make no claim to completeness. We refer to widgets as official or first party if they are either included in the default widget set or provided by the Flutter development team as a package. We consider a widget missing, if there is no existing official implementation of a concept or widget. For some, it is debatable whether they are really 'missing' because of the difference in approaching and designing applications in Flutter to existing frameworks. For others, third party packages exist.

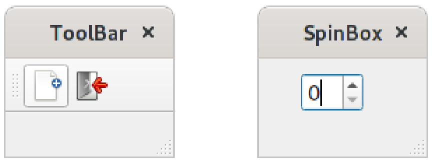


Figure 7.2: ToolBar and SpinBox from the Qt tool-kit

Two widgets we have not found an official implementation are the ToolBar and the SpinBox or SpinButton widgets (see Figure 7.2) available in Qt and GTK. The former typically consists of buttons with an icon and a label. Often, the user can move the ToolBar from the usual place at the top to the left, right, or bottom. A popular application that uses multiple ToolBars by default is LibreOffice. We have not found the same concept in Flutter, neither as a first party package nor as a third party library. The widget that comes closest to the ToolBar is the AppBar. The situation is different with the SpinBox. While we have not found a first party implementation of the SpinBox, we found multiple implementations from external developers: `flutter_spinbox`, `spinner_input`, and `flutter_counter` for example.

Another widget that is also not available as first party is the TreeView widget as shown in Figure 7.3. Again, a whole lot of third party packages exist to fill the gap: `tree_view`, `flutter_tree`, `flutter_treeview`, and `flutter_simple_treeview`.

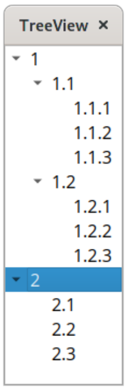


Figure 7.3: TreeView from the Qt tool-kit

A widget that is actively being worked on to bring it to the desktop is the Texture widget. In fact, this widget works already on macOS [Github, 2019b] and Windows [Github, 2020c], only Linux support is missing [Github, 2021b]. This widget enables Flutter to integrate with "host platform video player, camera, or OpenGL APIs, or similar image sources" [Github, 2020d].

7.3 General Remarks

The Flutter framework is developed on GitHub. For this reason, missing features of the tool-kit itself are also treated openly. In the following, we present a selection of missing features in the framework that some developers may consider crucial for their application:

- **Multiple windows support** [Github, 2019d]**:** Currently, only one-window-applications are supported. The Flutter team considers multiple windows to be a core feature. For this reason it is prominently listed in the roadmap for 2021 [Github, 2021d].
- **Tray icon support** [Github, 2019a]**:** This functionality is missing at the moment. The Flutter developers currently plan to add this functionality in the form of a plugin.
- **Full screen mode** [Github, 2020b]**:** At the moment, a mode is missing to display the content in full screen mode.
- **Disable window resizing** [Github, 2020a]**:** Currently, Flutter does not allow forbidding window resize. According to the developers, this would only require a few lines of native code.
- **Context menu** [Github, 2021c]**:** The current implementation of the context menu is limited in terms of layout, positioning, and functionality. For example, only the default buttons 'Cut', 'Copy', 'Paste', and 'Select all' are supported at the moment.
- **Client side window decorations** [Github, 2019c]**:** Traditionally, the window manager is responsible for positioning an application on the desktop. Also window decorations such as the title bar, minimize, and close buttons are also managed by the window manager. In recent years, it has become a trend to make use of the space in the title bar by moving application

controls into the title bar. For this, the application – the client – needs to control the window decorations. At the time of writing, no official support for client side window decorations exists in Flutter.

8 Conclusions and Outlook

Flutter, as an up-and-coming promising technology, is outperformed by Electron in our comparison. This is to be expected as Flutter is only in beta status at the time of writing. However, it is also not about whether Flutter is superior to Electron for desktop applications or visa versa. Flutter has undeniable advantages – write once, run on all platforms and operating systems – making it superior, provided the advantages matter. So, the question is, how far along is Flutter, and how well does it work?

To verify this, we wrote the same application in Flutter and Electron and tested it in an automated fashion. Our biggest surprise is that the Flutter application completed our import scenario faster in debug mode than in release mode. Especially the file reading and subsequent processing of the data was about two times faster in debug than in release mode but still slower than the Electron implementation. This also is surprising because we ported the reading and processing code almost one-to-one from TypeScript to Dart. At this point we can only speculate, but we believe that the performance difference is due to the compilation used: Debug builds are compiled *just in time* (JIT) and profile or release builds are compiled *ahead of time* (AOT) [Flutter, 2021c].

Our recorded usages of CPU and GPU are within the expected range, considering Flutter's beta status. In all our scenarios, CPU usage was above 10% regardless of which action our simulated user was executing. This is even surpassed by the GPU usage, as it almost never fell below the 40% threshold. In general, Flutter has a high base consumption regarding CPU and GPU usage. We are convinced that this will be addressed soon in order to "deliver production-quality support" [Github, 2021d] which is planned for this year. Other metrics we examined, such as system memory usage, startup time, and application size look good already, although there is still room for improvement.

We also considered Flutter's feature completeness. Based on our research of the top 100 packages, we can already certify good desktop support to Flutter. More packages out of our top 100 are available for the desktop than for the Web which already is considered a stable platform. However, we believe that there is still room for improvement for packages. As highlighted by our system memory usage results of the import scenario (see section 6.3), achieving the best results requires more effort in Flutter due to required package evaluation and package maturity. This is also evident from the number of implementations for missing widgets: We count at least three SpinBox and four TreeView implementations which developers would need to evaluate if they need these widgets. We are certain that this will improve over time, as these are signs of a still young ecosystem. As far as missing features are concerned, the most important ones are the support of multiple windows per application and customizable context menus. We believe that this will be taken care of by the time of the stable release, especially since most of the issues are already being worked on.

This leads us back to the research questions mentioned at the beginning of this thesis:

- **RQ1:** Is Flutter already a viable choice for desktop application development?
- **RQ2:** How does Flutter stack up against technologies comparable to Flutter?

We answer the first question with 'Almost'. The official documentation advises against "releasing a desktop application until desktop support is stable" [Flutter, 2021b]. But if all use cases are covered, there is nothing wrong with using Flutter, provided one wants to get involved in a still young ecosystem. Performance and system usage problems will be addressed by the Flutter team. However, if maturity and stability are essential, alternatives should be considered at the moment.

Answering the second question is more difficult. In this thesis, we have considered with the desktop target and can give a partial answer. Unsurprisingly, development was not as smooth or mature as it was with Electron and Angular. However, it was intuitive and required significantly less code. Additionally, the built-in packaging and distribution is a nice bonus. As far as our measured values are concerned, we would not attach too much importance to it. Although they appear underwhelming for CPU and GPU usage, the focus is on adding missing features. Thereafter, performance will be addressed. In any case, this is how the Web target was approached [Github, 2021d]. In order to answer the second question even better, performance measurements would have to be performed on additional operating systems and on additional hardware setups. In addition, other frameworks could be included in the comparison as well – for example React Native for another Web-based approach or Qt for a more native one. It would also have been interesting to see, how well Flutter scales with a really complex or long-running application. Finally, Flutter also targets the Web. This also opens up the possibility, to build a Web application in Flutter and then use Electron to turn it into a desktop application.

Bibliography

[Abe17] W. M. Abeer Alkhars. “Cross-Platform Desktop Development(JavaFX vs. Electron)”. English. Tech. rep. Linnaeus University, Växjö, 2017. 68 pp. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-61313 (visited on 11/10/2020) (cit. on p. 30).

[Aie18] M. Aiello. *The Web Was Done by Amateurs - A Reflection on One of the Largest Collective Systems Ever Engineered*. Springer, 2018, pp. 1–168. ISBN: 978-3-319-90007-0. DOI: 10.1007/978-3-319-90008-7. URL: https://doi.org/10.1007/978-3-319-90008-7 (visited on 11/10/2020) (cit. on pp. 19–21).

[Ang20] Angular. *Introduction to services and dependency injection*. Oct. 22, 2020. URL: https://angular.io/guide/architecture-services (visited on 04/29/2021) (cit. on p. 35).

[Aud09] Audriusa. *Web access to the server console at the hardware level with the help of a Java applet*. Oct. 29, 2009. URL: https://en.wikipedia.org/wiki/Java_applet#/media/File:Remoteconsoleapplet.png (visited on 04/08/2021) (cit. on p. 20).

[BBC+04] T. Berners-Lee, T. Bray, D. Connolly, P. Cotton, R. Fielding, M. Jeckle, C. Lilley, N. Mendelsohn, D. Orchard, N. Walsh, S. Williams. *Architecture of the World Wide Web, Volume One. W3C Recommendation 15 December 2004*. Dec. 15, 2004. URL: https://www.w3.org/TR/webarch/ (visited on 04/06/2021) (cit. on p. 19).

[BFM05] T. Berners-Lee, R. Fielding, L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Jan. 2005. URL: https://tools.ietf.org/html/rfc3986#section-1.1.3 (visited on 04/06/2021) (cit. on p. 19).

[BL15] L. Bak, K. Lund. *Dart for the Entire Web*. Mar. 25, 2015. URL: https://news.dartlang.org/2015/03/dart-for-entire-web.html (visited on 04/06/2021) (cit. on p. 27).

[BMG17] A. Biørn-Hansen, T. A. Majchrzak, T.-M. Grønli. “Progressive Web Apps: The Possible Web-native Unifier for MobileDevelopment”. In: *International Conference on Web Information Systems and Technologies*. Vol. 2. SCITEPRESS. 2017, pp. 344–351 (cit. on p. 30).

[Bra16] G. Bracha. *The Dart programming language*. Boston, 2016 (cit. on p. 27).

[Cal13] D. Calegari. *Mandelbrot set explorer*. May 8, 2013. URL: https://web.archive.org/web/20130508054436/http://math.uchicago.edu/~dannyc/fractals/simple.html (visited on 04/08/2021) (cit. on p. 20).

[CC20] Y. Cheon, C. Chavez. *Creating Flutter Apps from Native Android Apps*. Tech. rep. University of Texas at El Paso, Sept. 1, 2020. URL: https://scholarworks.utep.edu/cs_techrep/1492/ (visited on 03/24/2021) (cit. on pp. 30, 49).

[Che16] A. G. Chekkilla. "Monitoring and Analysis of CPUUtilization, Disk Throughput and Latencyin servers running Cassandra database. An Experimental Investigation". MA thesis. Faculty of ComputingBlekinge Institute of TechnologySE-371 79 Karlskrona Sweden, June 30, 2016 (cit. on p. 43).

[Dar21] Dart. *Dart language evolution*. Apr. 6, 2021. URL: https://dart.dev/guides/language/evolution (visited on 04/06/2021) (cit. on p. 27).

[Ele21a] Electron. *Build cross-platform desktop apps with JavaScript, HTML, and CSS*. Mar. 18, 2021. URL: https://www.electronjs.org/ (visited on 03/18/2021) (cit. on p. 22).

[Ele21b] Electron. *Quick Start Guide*. Mar. 18, 2021. URL: https://www.electronjs.org/docs/tutorial/quick-start (visited on 03/23/2021) (cit. on p. 23).

[Fen20] A. E. Fentaw. "Cross platform mobile application development: a comparison study of React Native VsFlutter". English. MA thesis. UNIVERSITY OF JYVÄSKYLÄFACULTY OF INFORMATION TECHNOLOGY, 2020. 98 pp. URL: http://urn.fi/URN:NBN:fi:jyu-202006295155 (visited on 11/10/2020) (cit. on p. 30).

[Fla06] D. Flanagan. *JavaScript: The Definitive Guide, 5th Edition*. O'Reilly Media, Inc., Aug. 31, 2006. ISBN: 9780596101992 (cit. on p. 28).

[Flu19] Flutter. *Flutter architectural overview*. English. Google. Oct. 12, 2019. URL: https://flutter.dev/docs/resources/architectural-overview (visited on 03/24/2021) (cit. on pp. 24, 25).

[Flu21a] Flutter. *Apps take flight with Flutter. See how customers are using Flutter to make beautiful apps in record time*. Google. Mar. 24, 2021. URL: https://flutter.dev/showcase (visited on 03/24/2021) (cit. on pp. 22, 24).

[Flu21b] Flutter. *Desktop support for Flutter*. Apr. 14, 2021. URL: https://flutter.dev/desktop (cit. on p. 56).

[Flu21c] Flutter. *Flutter performance profiling. Why you should run on a real device:* Flutter. Apr. 27, 2021. URL: https://flutter.dev/docs/perf/rendering/ui-performance (visited on 05/04/2021) (cit. on p. 55).

[Flu21d] Flutter. *Top 10 things you need to know about Flutter Engage*. Video starting from 3:51. Google, Canoncial. Mar. 4, 2021. URL: https://youtu.be/IdrCyS7EF8M?t=231 (visited on 03/24/2021) (cit. on p. 24).

[Fou] F. S. Foundation. *The GNU C Library. The GNU C Library Reference Manual*. URL: https://www.gnu.org/software/libc/manual/html_node/Processor-And-CPU-Time.html#Processor-And-CPU-Time (visited on 04/26/2021) (cit. on p. 43).

[Git19a] Github. *Add plugin for system tray icons*. Oct. 18, 2019. URL: https://github.com/google/flutter-desktop-embedding/issues/595 (visited on 05/01/2021) (cit. on p. 53).

[Git19b] Github. *Add texture support for macOS shell*. Apr. 19, 2019. URL: https://github.com/flutter/engine/pull/8507 (visited on 04/30/2021) (cit. on p. 53).

[Git19c] Github. *Desktop Client Side Decoration support*. Apr. 21, 2019. URL: https://github.com/flutter/flutter/issues/31373 (visited on 05/01/2021) (cit. on p. 53).

[Git19d] Github. *Support multiple windows for desktop shells*. Apr. 8, 2019. URL: https://github.com/flutter/flutter/issues/30701 (visited on 05/01/2021) (cit. on p. 53).

[Git20a] Github. *[window_size] Add an option to prevent all resizing*. July 11, 2020. URL: https://github.com/google/flutter-desktop-embedding/issues/774 (visited on 05/01/2021) (cit. on p. 53).

[Git20b] Github. *[window_size] Allow toggling full screen*. Feb. 20, 2020. URL: https://github.com/google/flutter-desktop-embedding/issues/679 (visited on 05/01/2021) (cit. on p. 53).

[Git20c] Github. *Add windows plugin texture support*. June 30, 2020. URL: https://github.com/flutter/engine/pull/19405 (visited on 04/30/2021) (cit. on p. 53).

[Git20d] Github. *Texture class*. Nov. 2, 2020. URL: https://api.flutter.dev/flutter/widgets/Texture-class.html (visited on 04/30/2021) (cit. on p. 53).

[Git21a] Github. *[Linux][GTK] The text cursor moves to start when typing in textfield on some versions of GTK*. Feb. 19, 2021. URL: https://github.com/flutter/flutter/issues/76383 (visited on 04/22/2021) (cit. on p. 39).

[Git21b] Github. *Linux texture support*. Mar. 10, 2021. URL: https://github.com/flutter/engine/pull/24916 (visited on 04/30/2021) (cit. on p. 53).

[Git21c] Github. *Proposal: desktop context menu fidelity*. Jan. 19, 2021. URL: https://github.com/flutter/flutter/issues/74255 (visited on 05/03/2021) (cit. on p. 53).

[Git21d] Github. *Roadmap*. Feb. 22, 2021. URL: https://github.com/flutter/flutter/wiki/Roadmap (visited on 05/01/2021) (cit. on pp. 17, 53, 55, 56).

[Goo19] Google. *Who uses Dart*. Aug. 15, 2019. URL: https://dart.dev/community/who-uses-dart (visited on 04/12/2021) (cit. on p. 22).

[GW17] C. Griffith, L. Wells. *Electron: From Beginner to Pro: Learn to Build Cross Platform Desktop Applications using Github's Electron*. Berkeley, CA: Apress, 2017. ISBN: 978-1-4842-2826-5. DOI: 10.1007/978-1-4842-2826-5_6. URL: https://doi.org/10.1007/978-1-4842-2826-5_6 (cit. on p. 23).

[HK14] M. Hills, P. Klint. "PHP AiR: Analyzing PHP systems with Rascal". In: *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Feb. 6, 2014, pp. 454–457. DOI: 10.1109/CSMR-WCRE.2014.6747217 (cit. on p. 21).

[HRS+17] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, J. Bastien. "Bringing the Web up to Speed with WebAssembly". In: *SIGPLAN Not.* 52.6 (June 30, 2017), pp. 185–200. ISSN: 0362-1340. DOI: 10.1145/3140587.3062363. URL: https://doi.org/10.1145/3140587.3062363 (cit. on p. 28).

[Hun09] P. Hunter. *ChessApp*. Sept. 7, 2009. URL: https://web.archive.org/web/20090907072956/http://english.op.org/~peter/ChessApp/ (visited on 04/08/2021) (cit. on p. 20).

[IG19] S. Ivanova, G. Georgiev. "Using modern web frameworks when developing an education application: a practical approach". In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. May 24, 2019, pp. 1485–1491. DOI: 10.23919/MIPRO.2019.8756914 (cit. on p. 30).

[Ion20] Ionic. *Deploying a Desktop App*. Apr. 2, 2020. URL: https://ionicframework.com/docs/deployment/desktop-app (visited on 05/07/2021) (cit. on p. 17).

[Jet20] Jetbrains. *The State of Developer Ecosystem 2020. Java*. English. June 11, 2020. URL: https://www.jetbrains.com/lp/devecosystem-2020/java/ (visited on 04/12/2021) (cit. on p. 21).

[Kin18] S. Kinney. *Electron in Action*. Manning Publications, Nov. 1, 2018. ISBN: 1617294144. URL: https://www.xarg.org/ref/a/1617294144/ (cit. on pp. 22, 23).

[Lie20] C. Liebel. *Produktivitäts-PWAs auf Desktop-Niveau dank File System Access und File Handling API*. Nov. 19, 2020. URL: https://www.heise.de/developer/artikel/Produktivitaets-PWAs-auf-Desktop-Niveau-dank-File-System-Access-und-File-Handling-API-4963752.html (visited on 04/23/2021) (cit. on p. 28).

[LKP20] D. Lehmann, J. Kinder, M. Pradel. "Everything Old is New Again: Binary Security of WebAssembly". In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 14, 2020, pp. 217–234. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann (cit. on p. 28).

[MMPB20] D. Maguire, J. Martinez, H. Payne, A. Borys. *How Microsoft Teams uses memory*. Apr. 11, 2020. URL: https://docs.microsoft.com/en-us/microsoftteams/teams-memory-usage-perf (visited on 03/18/2021) (cit. on pp. 17, 22).

[Nod21] Node.js. *Node.js v15.12.0 Documentation*. Feb. 8, 2021. URL: https://nodejs.org/docs/latest/api/ (visited on 03/23/2021) (cit. on p. 23).

[Ora21a] Oracle. *Java and Firefox Browser*. Apr. 8, 2021. URL: https://www.java.com/en/download/help/firefox_java.html (visited on 04/08/2021) (cit. on p. 20).

[Ora21b] Oracle. *Java and Google Chrome Browser*. Apr. 8, 2021. URL: https://java.com/en/download/help/chrome.html (visited on 04/08/2021) (cit. on p. 20).

[Pro21] C. Project. *Content module*. Mar. 18, 2021. URL: https://chromium.googlesource.com/chromium/src/%20/HEAD/content/README.md (visited on 03/18/2021) (cit. on p. 22).

[SA20] G. L. Scoccia, M. Autili. "Web Frameworks for Desktop Apps: An Exploratory Study". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3422171. URL: https://doi.org/10.1145/3382494.3422171 (cit. on pp. 29, 30).

[Saw15] K. Sawicki. *Atom Shell is now Electron*. Apr. 23, 2015. URL: https://www.electronjs.org/blog/electron (visited on 03/18/2021) (cit. on p. 22).

[Sel21] C. Sells. *What's New in Flutter 2. Flutter web and Null Safety move to stable, Flutter desktop moves to beta and so much more!* Google. Mar. 3, 2021. URL: https://medium.com/flutter/whats-new-in-flutter-2-0-fe8e95ecc65 (visited on 03/24/2021) (cit. on p. 24).

[Sou99] K. Southwick. *High Noon: The Inside Story of Scott McNealy and the Rise of Sun Microsystems*. New York : John Wiley, 1999. URL: https://archive.org/details/highnoon00kare/page/120/mode/2up (visited on 04/12/2021) (cit. on p. 21).

[Sta21] Statcounter. *Desktop vs Mobile vs Tablet Market Share Worldwide. Mar 2020 - Apr 2021*. Apr. 30, 2021. URL: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide (visited on 05/03/2021) (cit. on p. 17).

[VG16] K. Vassallo, L. Garg. “Cross-Platform Development Frameworks: Overview of contemporary technologies and methods for cross-platform application development.” English. In: Dec. 1, 2016. URL: https://www.researchgate.net/publication/314680454_Cross-Platform_Development_Frameworks_Overview_of_contemporary_technologies_and_methods_for_cross-platform_application_development (cit. on p. 29).

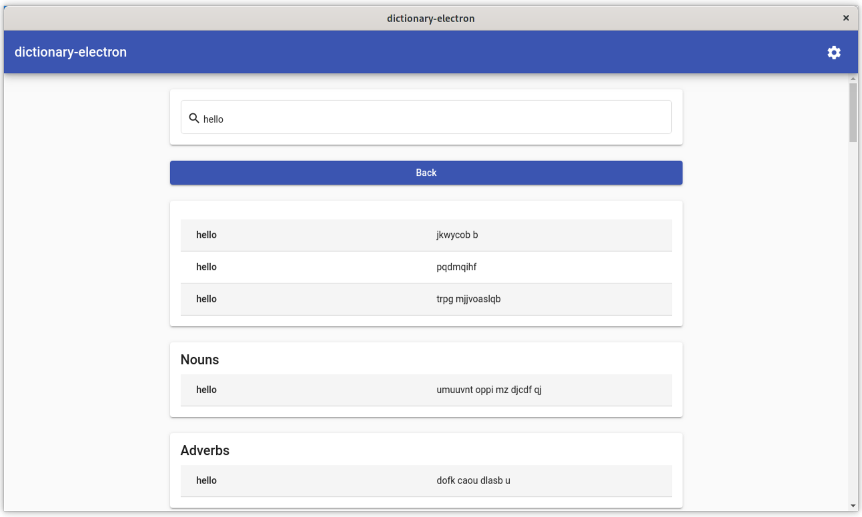
[Wik21] Wikipedia. *CPU time*. Apr. 12, 2021. URL: https://en.wikipedia.org/wiki/CPU_time (visited on 04/29/2021) (cit. on p. 43).

[Zha13] C. Zhao. *Atom Shell v0.1.0*. July 15, 2013. URL: https://github.com/electron/electron/tree/v0.1.0 (visited on 03/18/2021) (cit. on p. 22).

If not otherwise stated, all links were last followed on May 1, 2021.

A Screenshots of the Reference Application

A.1 Detail View



hello	jkwycob b
hello	pqdmqihf
hello	trpg mjjvoaslqb

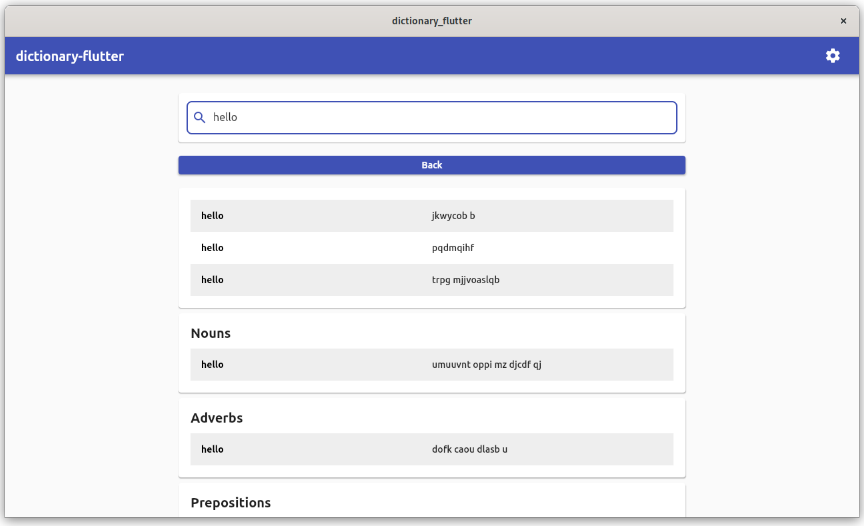
Nouns

hello	umuuvnt oppi mz djcdf qj

Adverbs

hello	dofk caou dlasb u

Figure A.1: Detail view in Electron



hello	jkwycob b
hello	pqdmqihf
hello	trpg mjjvoaslqb

Nouns

hello	umuuvnt oppi mz djcdf qj

Adverbs

hello	dofk caou dlasb u

Prepositions

Figure A.2: Detail view in Flutter

A.2 Settings View

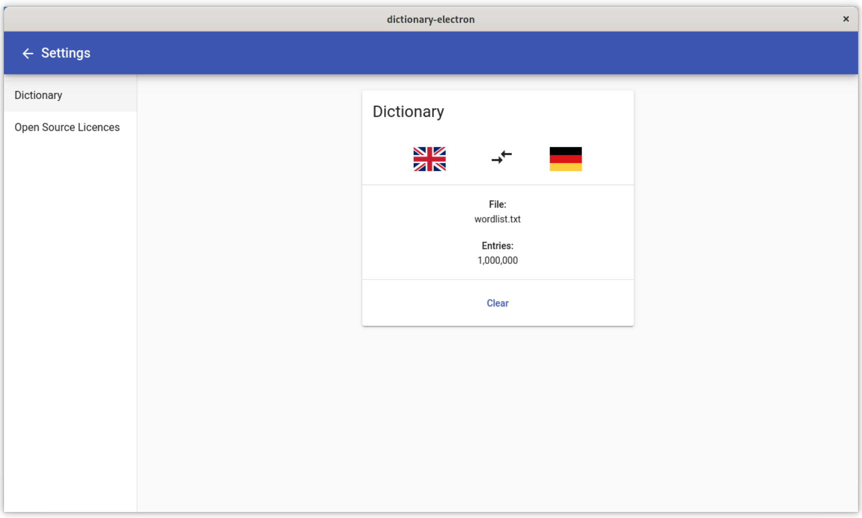


Figure A.3: Settings view in Electron

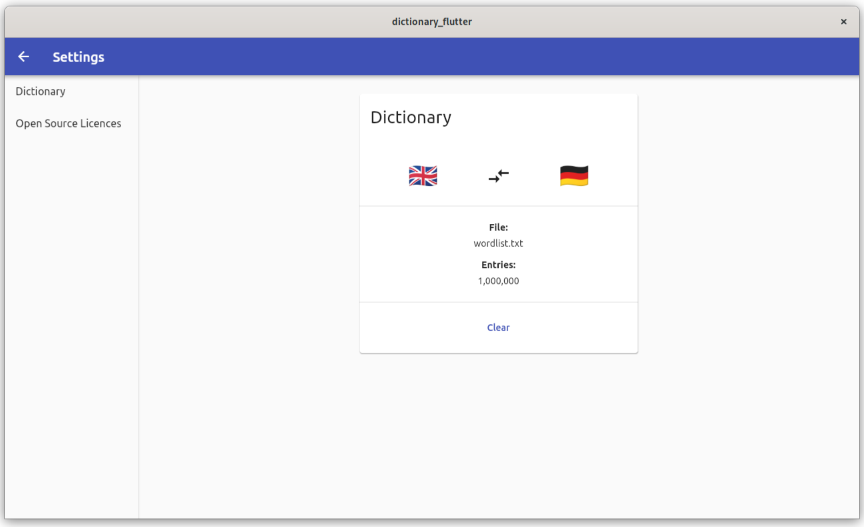


Figure A.4: Settings view in Flutter

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature