

A Real-Time Visualization Tool of Hardware Resources for Flutter Applications

Felipe Bedinotto Fava and Claudio Schepke(✉)

Laboratory of Advanced Studies in Computation, Federal University of Pampa (UNIPAMPA), Alegrete, RS, Brazil
{felipefava.aluno,claudioschepke}@unipampa.edu.br

Abstract. Mobile development is a constantly evolving field. Several options are available for developing cross-platform applications, such as Flutter, React Native, Ionic, and Xamarin. These options have the potential to achieve excellent performance compared to native applications, and they can also facilitate the development process by reducing the need for separate projects for each platform. One of the main challenges in developing mobile applications is finding potential changes that could affect the application's performance. To address this topic, we develop a Flutter plugin. It helps identify performance issues for developers in their applications while developing and testing them. The plugin increases the speed of detecting bugs and keeps the application running smoothly. We develop the performance tool in Dart. It features three main aspects to be visualized by the developer: frames rendered per second, number of bytes sent and received through the network, and RAM's information of total available, occupied, and free memory. We validated the tool on three Dart applications running on a physical smartphone, displaying the performance results in real time. The results show variations in FPS, RAM, and network collected values during the execution of the applications.

Keywords: Flutter · Plugin · Network Monitor · FPS · Memory

1 Introduction

Mobile application development is a field that is constantly evolving [3,4]. Several options are available to developers, each with its advantages and disadvantages. Choosing the best option for a project will depend on its specific needs. One popular option is native development, which involves platform-specific languages and tools like Swift and Objective-C for iOS [2], and Java and Kotlin for Android [9]. That often results in high-performing applications but can also be more laborious and expensive due to the need to create and maintain two different source codes. Alternatively, there are development tools such as Flutter, React Native, Ionic, and Xamarin, which allow the development of multiplatform applications with the potential to achieve excellent performance compared

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
O. Gervasi et al. (Eds.): ICCSA 2024, LNCS 14814, pp. 17–30, 2024.
https://doi.org/10.1007/978-3-031-64608-9_2

to native applications [5,6,12]. These tools also facilitate process development by reducing the need for different teams and projects for each platform.

To be accepted by users, a system must work efficiently and meet their needs [1]. It can be frustrating when users open an application and notice that it drains their device battery faster than usual. Poor system performance can also be a crucial turnoff for new users. An application that performs poorly can cause issues such as slowness, crashes, and errors, which lead to user lack of satisfaction, application uninstallation, and even financial loss. Therefore, the system design needs optimal performance to ensure a positive user experience.

Flutter [8] provides developers with a helpful set of tools called DevTools [11] for debugging code and analyzing performance. It is powerful and allows developers to understand how their application is performing. However, this process is still complex and, in some ways, costly, as it requires external application monitoring. To access the DevTools interface, the developer needs to use a link provided by the Flutter SDK when launching the application through the SDK or access using a new tab of Visual Studio Code if all dependencies were already correctly configured. Moreover, the tool's interface has a steep learning curve, making it difficult for many developers to use.

After considering all the aspects presented, we propose a Flutter plugin to help developers identify possible performance issues and detect bugs more efficiently in real-time. Thus, it can used during the development of Flutter applications. The specific objectives pursued during the plugin's development are as follows:

- Be easy for the developers to use;
- Make the interface as less invasive as possible;
- Display an indication of the number of frames generated per second;
- Show values of memory usage;
- View indicators of network traffic.

The contributions of the work are:

(i) the description of a simple, non-invasive plugin alternative to help Flutter developers identify performance problems in their applications while developing them;
(ii) an extension of Flutter's bibliography, given that it is still a topic that is not yet very present in academic works.

The organization of this paper is in four sections. Section 2 points the related work. Section 3 shows aspects of methodology like implementation and parameters for the modules. Section 4 details the results obtained through the plugin execution. Section 5 presents the conclusion of the paper and future works.

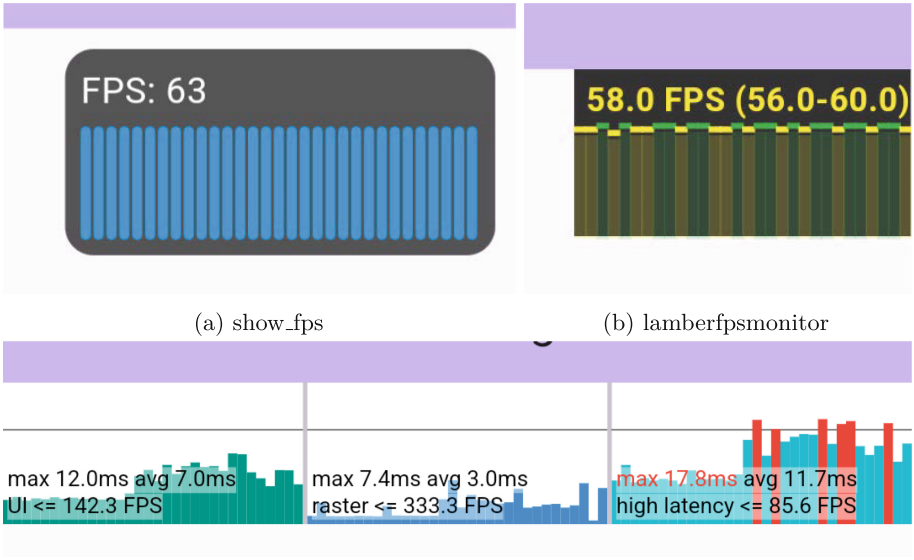
2 Related Work

We discuss in this section the packages and plugins found in the official Dart and Flutter repository [10]. We consider these packages to be related work due

to their relevance to our work. Our research criteria include the terms Flutter, Android, or iOS that address CPU, memory, or network traffic monitoring. We used the base application created by executing the `flutter create {name_for_flutter_project}` command to test these packages and plugins. We have listed the packages that are most suitable for our work below.

1. **show_fps**[1] - The package provides a simple FPS viewer as shown in Fig. 1a. We encounter some issues while using the code example on the official page. The frame counter didn't appear in the application despite following the instructions. It only displays a text reading *FPS* in the top right corner of the page. However, changing the *showChart* parameter from *false* to *true* resolved the issue, and the counter started working alongside the chart. It's worth noting that the chart updates very quickly, which makes it challenging to follow events.
2. **performance_fps**[2] - This package displays the FPS rate of an application. However, the package did not include interface elements and was only accessible to the developer. The developer can call a function that will return two values: the FPS rate and the number of dropped frames, referred to as *dropCount*. It is worth noting that this plugin uses specific mechanisms of the target platforms to obtain data. But it stood discontinued.
3. **lemberfpsmonitor**[3] - It provides the developer a chart as shown in Fig. 1b. At the upper right corner of its interface to the left, it displays the frame rate per second, the lowest and the highest frame rates recorded within the sample window. The visualization of these two last data is inside a parenthesis, split by a hyphen. By default, it has a *maxFPS* variable set to 60. It locks the maximum displayed value of FPS in the chart, The package's *widget* provides an option to change the maximum displayed value of FPS.
4. **statsfl**[4] - The previous *lemberfpsmonitor* package and its official documentation inspired this package. It has the same interface as its predecessor. However, this one does not display any graphs. Instead, it only shows the achieved frame rate values and a small line indicating which frames reached the target of 60 FPS.
5. **memory_info**[5] - This package does not have a pre-built data presentation interface for developers. However, it provides some methods to retrieve information about the device's RAM and internal storage. The available information includes the total RAM of the device, the amount of free RAM, the total size of internal storage, and the amount of free space on it.
6. **performance**[6] - This particular package served as an inspiration for the development of our project. It generates three charts on the primary application that display, from left to right, the frames produced by the UI isolate,

[1] https://github.com/mantreshkhurana/show_fps.
[2] https://github.com/allenymt/flutter_fps.
[3] https://github.com/lember-ecu/fpsmonitoring.
[4] https://github.com/gskinnerTeam/flutter-stats-fl.
[5] https://github.com/MrOlolo/memory_info/tree/master/memory_info.
[6] https://github.com/creativecreatorormaybenot/performance.



(a) show_fps (b) lamberfpsmonitor

(c) performance

Fig. 1. Related work interfaces

raster, and the total time interval between the beginning of sync and the completion of rasterization, as shown in Fig. 1c and Fig. 2. This package has an important feature that updates the graphics only when there is an update on the screen. Nothing appears in the graphics if there is no need for screen refreshment. However, when there is interaction, the graphics get updated.

The **performance** package uses Flutter's resources to gather data to generate charts to achieve this behavior. It motivates us to find ways to capture other data types for monitoring the device while developing an application. The evolution of the tool became the focus of our work. Table 1 presents FPS rate/info, RAM, and Network as monitoring aspects and compares the evaluated packages and our proposal over these aspects.

Table 1. Related Plugins and Packages

Packages	FPS rate	Other FPS info	RAM	Network
show_fps	x			
performance_fps	x	x		
lemberfpsmonitor	x	x		
statsfl	x	x		
memory_info			x	
performance	x	x		
Our Work	x		x	x

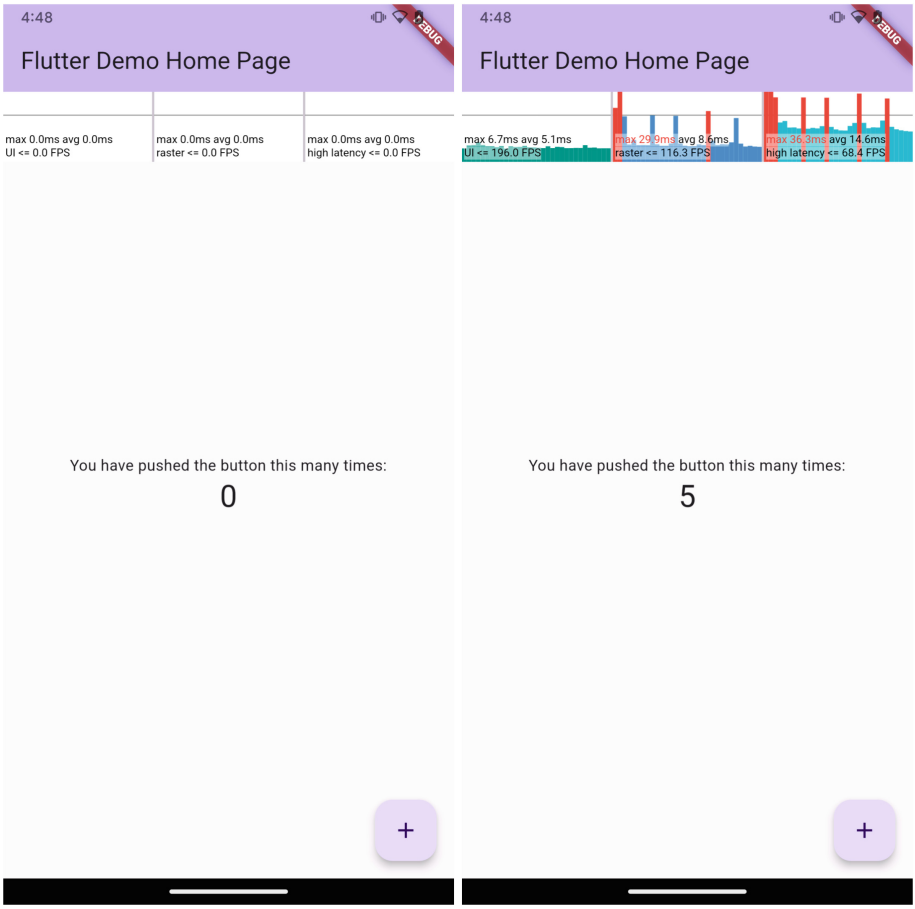


Fig. 2. performance package test

3 Methodology

This section describes the creation process of a performance tool application. The objectives of the tool were defined based on the following factors considered:

- CPU usage: The amount of processing power the application requires and the impact on device performance.
- Memory usage: The amount of memory the application uses while running.
- Battery usage: The influence of the app has on the device's battery life.
- Network usage: The amount of data the system transfers over the network and the effect on network performance.
- Startup time: The time it takes for the application to start and be ready for use.

- Frame rate: The number of frames per second that the application can display that can impact the smoothness of animations and scrolling.
- Experience latency: The time it takes for the system to respond to user input.
- Crashes rate: The number of crashes per user or session.

All these points are significant and analyzed during the development and life of an application. However, to identify one or more problems all of the time, several of these factors are observed and analyzed to find ways to fix them.

It's important to remember that most factors will depend on the application and its intended use. It is also worth evaluating application performance on different devices and under conditions to understand its performance. Among all these factors, we chose three for the first implementation of the tool: memory usage, network usage, and frame rate. Using these three factors together, it is now possible for the developer to understand what is the regular behavior of their application while developing. We can see the impacts in real time, analyzing these three factors with each new code change.

The developed tool allows developers to evaluate the performance of their applications quickly and easily. It provides valuable information to improve application performance and provide a better experience for users.

3.1 Frame per Second (FPS) Counter

During the development of the tool, we reached a stage where we needed to find references and implementation examples. In Sect. 2, we had previously presented the requirement of displaying frames per second. After researching, we found five packages that could provide the desired functionality. However, after analyzing their operation and implementations, we choose only the **performance** package. We studied this package in-depth to understand how it is possible to obtain and manage data until its presentation on the screen.

As shown detailedly in [7], there is a `mixin` available in the `scheduler` library, the `SchedulerBinding` that is a crucial mechanism in Flutter's **performance** package and which is native to Flutter. Therefore, there's no need to install any external modules.

Using `SchedulerBinding`, we can acquire a list of `FrameTiming`, which is a class available in `dart:ui`. This library provides low-level services for the Flutter framework when initializing applications. The `FrameTiming` class stores metrics related to the timing of a frame, such as `vsyncStart`, `buildStart`, `buildFinish`, `rasterStart`, and `rasterFinish`. By combining reactivity, `SchedulerBinding`, and callbacks, we can readily and effectively collect the necessary data to present in the frame counter. Figure 3a shows how the frames per second counter interface looks.

3.2 Memory Usage

While Flutter excels at managing frame rates entirely within Dart code, accessing memory hardware details necessitates a platform-level approach due to limitations in Dart itself. Due to the inclusion of code specific to the Android platform,



(a) FPS Counter (b) RAM Information (c) Network Profile

Fig. 3. Performance Aspects

the functionality presented in this paper is called a plugin rather than a package, which solely relies on Dart code. Using Kotlin as a programming language for specific development for Android, the `MemoryInfo` class was used to return the information to the Dart code through `MethodChannel` and thus present the memory data on the screen. The *plugin* shows three memory data information, as presented in Fig. 3b, which are total, used, and free device memory in *gigabytes*.

The executed routine does not use Flutter's internal reactivities. It fetches information from the platform. It is invoked every second within a function in Dart code. Because of this behavior, when the frame monitor is run alone in a newly created application, it behaves the same as the **performance** package, for example. There are updates to the graph only when there are changes to the interface. However, when the memory or network traffic monitoring module runs together with the frame monitor, there is an interface update every second.

3.3 Network Traffic

In today's world, where many mobile applications act as mere interfaces to APIs, constantly sending and receiving data, traffic monitors play a crucial role. These tools allow developers to measure the exact amount of data flowing in and out of their applications via the Internet during each request-response cycle. This granular insight is invaluable for identifying anomalies. Imagine an API request programmed to receive a specific amount of data (X) but instead returns a significantly different amount (Y). A traffic monitor can flag such discrepancies, helping developers diagnose and fix issues before they impact user experience.

Flutter does not have direct access to the network layer of devices. This layer is only available through system APIs. We sought a solution for the Android operating system since it is not possible to acquire data through Flutter or Dart The TrafficStats class provides the capture of values through its methods. We use the getTotalTxBytes and getTotalRxBytes methods to get the number of bytes sent and received throughout the system, respectively. Figure 3c shows the results of this invocation. As this class collects data at the network layer, it does not differentiate between TCP and UDP packets.

3.4 Customization Parameters

There are several parameters to customize plugins, making it easier for developers to manipulate the modules on the screen. Here are the parameters and their default values that each one indicates:

- **bool disable**: Whether the plugin will be disabled or not, preventing it from being initialized. Its default value is *false*.
- **bool activateFPS**: Whether the frame generation module will be or not activated. Its default value is *true*.
- **double scaleFPS**: The rendered scale of the frame generation module. Its default value is *1*.
- **Alignment alignmentFPS**: The frame generation module alignment within the device screen. Its default value is *Alignment.topRight*.
- **int sampleSizeFPS**: The extrapolated number of 'frame time' samples to the frame generation module. Its default value is *32*.
- **Duration targetFrameTimeFPS**: The maximum time to build a frame. All frames that take longer than this parameter will appear in red. Otherwise, they will draw in the *chartBarsColor* parameter color. This parameter also tells the plugin where to draw the horizontal black line present in the frame generation module. Its default value is *16.7 milliseconds*. This value is equivalent to 60 frames per second.
- **Color backgroundFPS**: The background color of the frame generation module. Its default value is *Color(0x00ffffff)*. 
- **Color textColorFPS**: The text color of the frame generation module. Its default value is *Color(0xff000000)*. 
- **Color textBackgroundColorFPS**: The background color of the text of the frame generation module. Its default value is *Color(0xffffffff)*. 
- **Color chartBarsColor**: The rectangles color representing the frames superior to the *targetFrameTimeFPS* of the frame generation module. Its default value is *Color(0xff4caf50)*. 
- **bool activateMemory**: Whether the memory module will be or not activated. Its default value is *true*.
- **Alignment alignmentMemory**: The memory module alignment within the device screen. Its default value is *Alignment.topRight*.
- **double scaleMemory**: The rendered scale of the memory module. Its default value is *1*.
- **Color backgroundMemory**: The background color of the memory module. Its default value is *Color(0x00ffffff)*. 
- **Color textColorMemory**: The text color of the memory module. Its default value is *Color(0xff000000)*. 
- **Color textBackgroundColorMemory**: The background color of the memory module text. Its default value is *Color(0xffffffff)*. 
- **bool activateNetwork**: Whether the network module will be or not activated. Its default value is *true*.

- **Alignment alignmentNetwork**: The network module alignment within the device screen. Its default value is *Alignment.topLeft*.
- **double scaleNetwork**: The rendered scale of the network module. Its default value is *1*.
- **Color backgroundNetwork**: The background color of the network module. Its default value is *Color(0x00ffffff)*.
- **Color textColorNetwork**: The color of the network module text. Its default value is *Color(0xff000000)*.
- **Color textBackgroundNetwork**: The background color of the network module text. Its default value is *Color(0xffffffff)*.
- **int timeBox**: The seconds shown in the network chart. Its default value is *60*.

4 Experimental Results

This section provides an overview of the interface of the developed plugin, tested on two real applications named *Wonderous* and *OpenLeaf*, both of which are available for download on the *Play Store*. Additionally, both of these applications have public repositories on GitHub[7,8]. During the development of this plugin, we also created a test application.

We obtain the results presented in this section by running the applications on a physical smartphone, a Motorola Z3 Play. It has a Qualcomm SDM636 Snapdragon 636 processor and 4GB of RAM. By doing so, we can evaluate the impact of applications on hardware that is somehow outdated and considered with characteristics of a low-level device.

Although the plugin is available in debug mode, we recommend analyzing the application's behavior in profile or even release mode to obtain results consistent with the actual performance of the application. Debug mode includes functions that can impact and harm the values found during the execution of the application.

We publish the first version of the plugin in the official package repository for Dart and Flutter applications[9]. An open repository is available to track the plugin's entire development[10].

4.1 Sample Test Application

We developed a sample test application, a plugin that performs simple actions such as sending HTTP requests to an online API, incrementing variables through button interaction, and generating a list of 1000000 objects displayed on a different application page. Figure 4 illustrates the difference in the application's behavior before (left screen) and after (right screen) an HTTP request.

[7] https://github.com/JideGuru/FlutterEbookApp.
[8] https://github.com/gskinnerTeam/flutter-wonderous-app.
[9] https://pub.dev/packages/perf_view.
[10] https://github.com/bedinotto/perf_view.

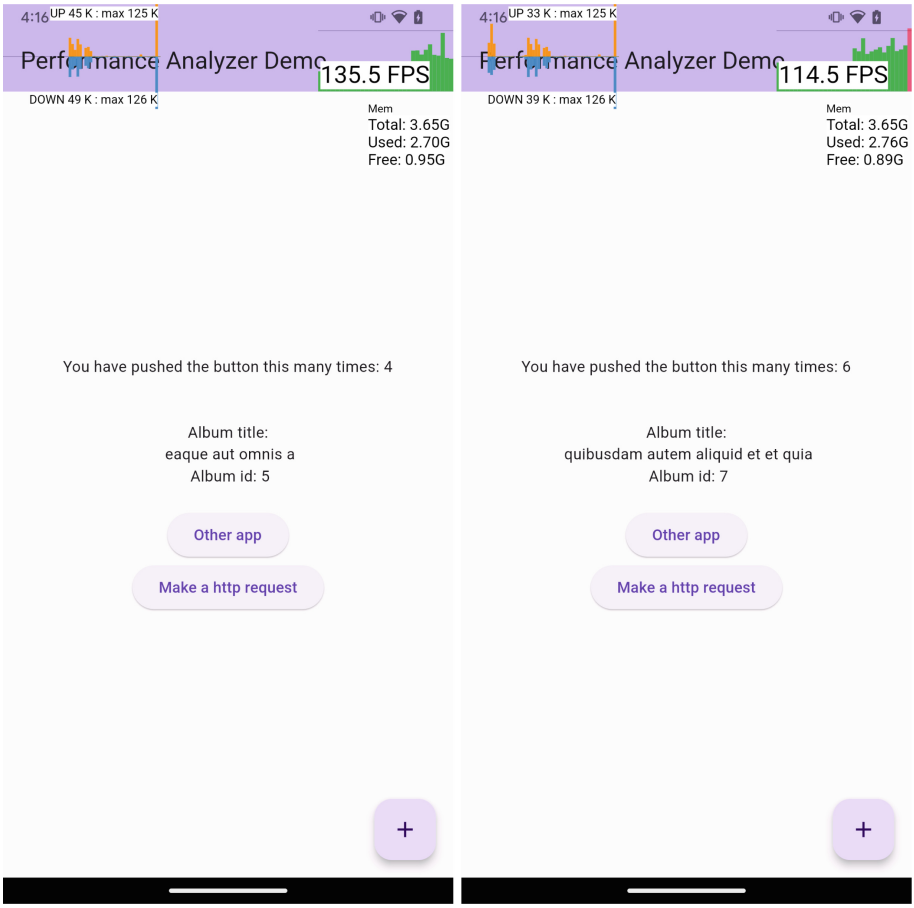


Fig. 4. Comparison of application behavior when making a request to the Internet

4.2 Wonderous

This application is a resource for information on some of the most famous structures in the world, including the Great Wall of China, Christ the Redeemer, and the Pyramids of Giza. The application uses various layout features in Flutter, showcasing the platform's impressive styling and animation capabilities.

In Fig. 5, we can see the Wonderous application page, which provides information, specifically on Christ the Redeemer. At the top of each screen, we can see the value in real-time of network traffic (UP and DOWN), the Frame Per Second Counter, and the memory usage (Total, Used, and Free). Unfortunately, the page experiences a significant drop in performance in one section (right screen). There are many animations and visual effects. Specifically, the frame rate drops to one quarter (87.6 to 19.6 FPS) of what is present on the rest of the page (as the left screen).

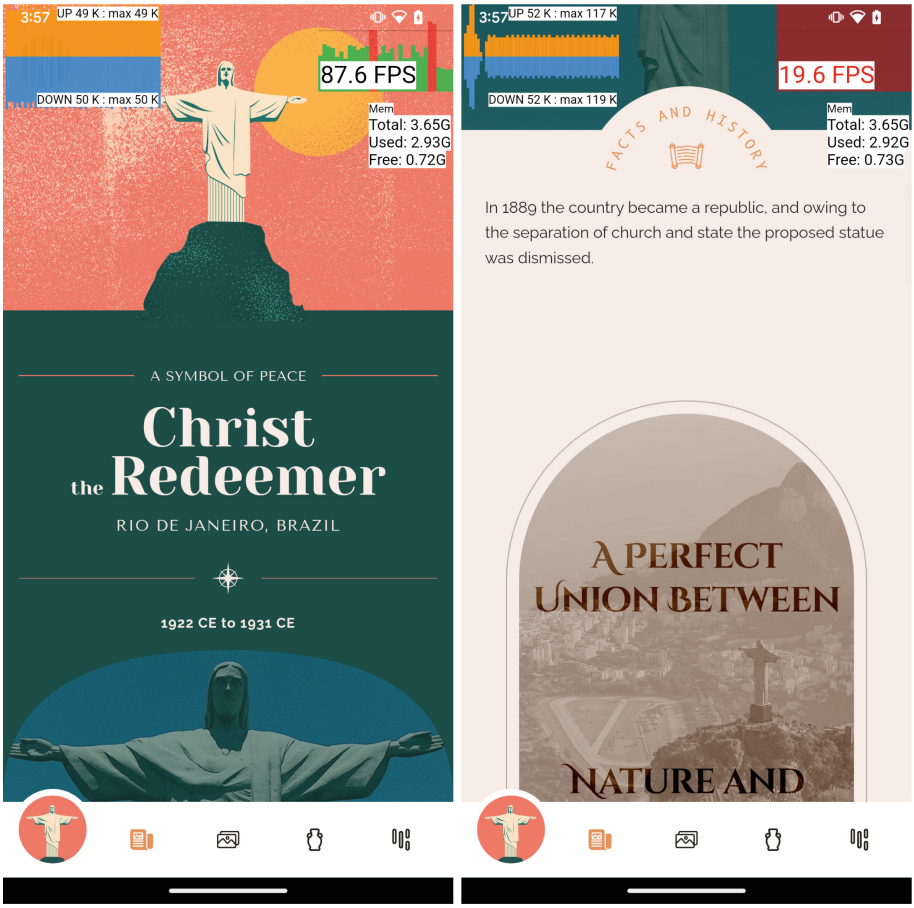


Fig. 5. Two pages about Christ the Redeemer with active plugin on the top of the screen

We notice that the application's other pages, which have the *layout* component filled with animations, exhibited the same behavior. We observe this behavior in Fig. 6, which also shows customized alignment settings for the plugin components.

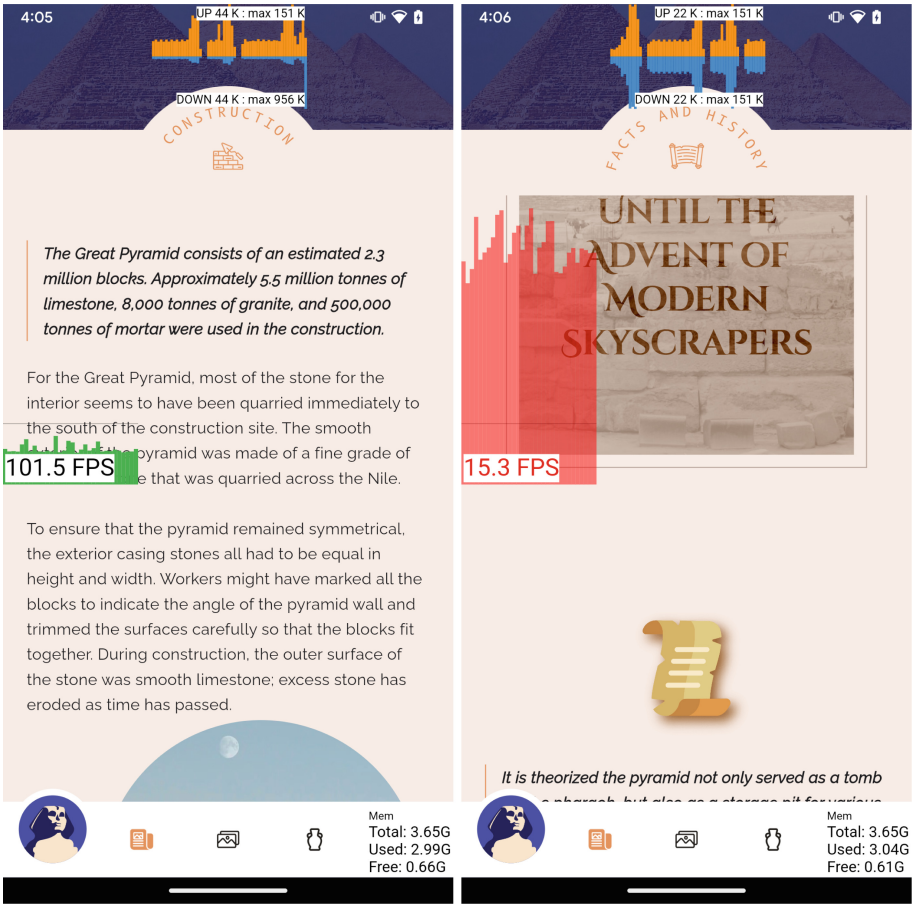


Fig. 6. Page about The Great Pyramid with active plugin

4.3 OpenLeaf

This application functions as a virtual library and offers a range of public domain books that are readily accessible. It also doubles up as a PDF file reader. Figure 7 illustrates the difference between the application's regular performance (left screen) while the interface is loading (right screen). An exceptional frame generation of the test device results when a new page is loaded. We also see more DOWN network traffic and usage of memory resulting from the requested information on the second screen.

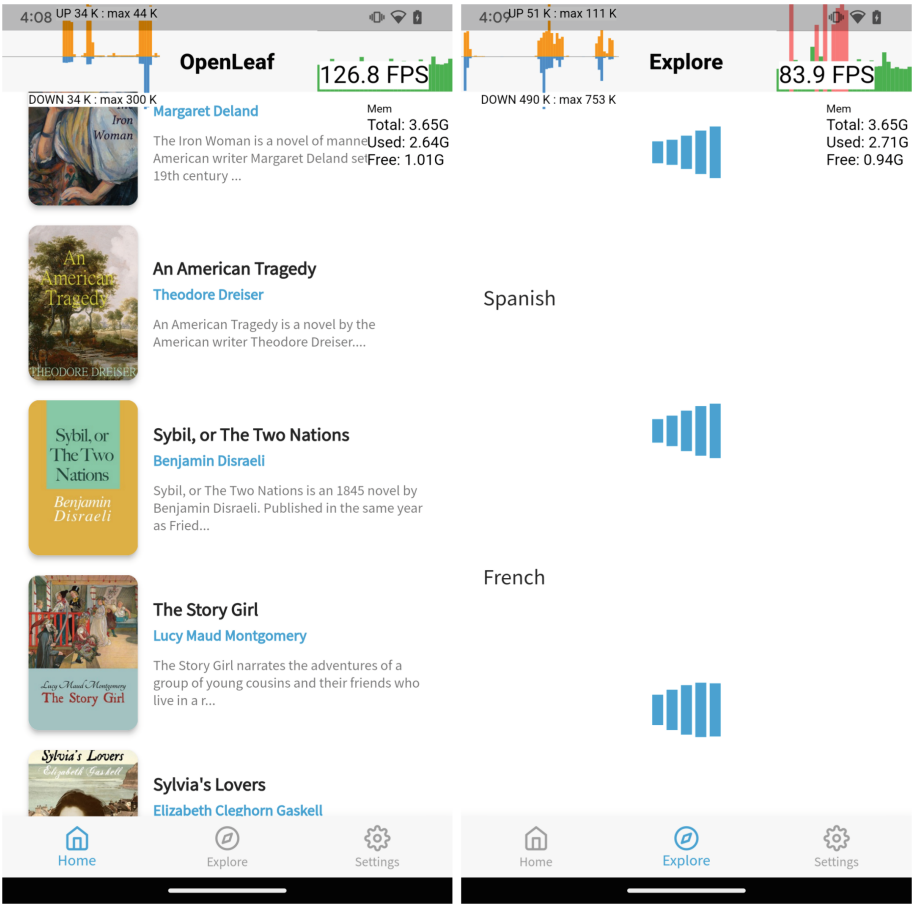


Fig. 7. Comparison between loaded and loading page of the OpenLeaf application

5 Conclusion and Future Works

In this article, we discuss the development of a plugin that measures the performance of applications built using Flutter. We explain the steps taken during the software development, including how the tool provides the visualization for frames generated per second, memory total, free and occupied, and network data transmission. We evaluate the plugin over a sample test application and two applications: Wonderous and OpenLeaf.

With the tool developed, we created a new option for Flutter developers to use while working on their Flutter application, not completely chained to DevTools. The plugin is free and available in the official repository of Dart and Flutter packages and plugins. Additionally, we host the source code in the platform's versioning Github repository. It is also available for free. By making this

work available to the Flutter community, we hope to encourage other developers to use and disseminate it.

In future work, we plan to evaluate the feasibility of using the features introduced in Flutter version 3.16. It recently made the DevTools feature extension available to package authors. We also intend to assess the plugin's computational impact on application performance and explore the possibility of creating similar projects for other application development tools to help developers who use those tools. It includes the application communication in a distributed environment.

Acknowledgment. The authors would like to thank the National Council for Scientific and Technological Development - CNPq (Process Number 407827/2023-4) and Fundação de Amparo à Pesquisa do Estado do Rio Grande do Sul - FAPERGS (07/2021 PqG project N° 21/2551-0002055-5).

References

1. Amaravadi, C., Lessard, Z.L.: The characteristics of good systems. Lond. J. Res. Comput. Sci. Technol. **17** (2017). https://journalspress.com/the-characteristics-of-good-systems/
2. Apple Inc: Objective-C and C++ interoperability with Swift (2024). https://developer.apple.com/swift/#objective-c
3. Caminade, J., von Wartburg, M.: The Success of Third-Party Apps on the App Store (2022). https://www.apple.com/newsroom/pdfs/the-success-of-third-party-apps-on-the-app-store.pdf
4. Ceci, L.: How many apps are there in the play store? (2024). https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/
5. Demedyuk, I., Tsybulskyi, N.: Flutter vs native vs react-native: examining performance (2020). https://inveritasoft.com/blog/flutter-vs-native-vs-react-native-examining-performance
6. Ebone, A., Tan, Y., Jia, X.: A performance evaluation of cross-platform mobile application development approaches. In: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 92–93 (2018)
7. Flutter. Schedulerbinding (2024). https://github.com/flutter/flutter/blob/master/packages/flutter/lib/src/scheduler/binding.dart
8. Google. Flutter documentation (2024). https://docs.flutter.dev
9. Google. Learn Kotlin for Java developers (2024). https://developer.android.com/kotlin/learn-kotlin-java-pathway
10. Google. The official package repository for Dart and Flutter apps (2024). https://pub.dev
11. Google. What is DevTools? (2024). https://docs.flutter.dev/tools/devtools/overview#what-is-devtools
12. Zahra, H.A., Zein, S.: A systematic comparison between flutter and react native from automation testing perspective. In: 2022 International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT), pp. 6–12 (2022). https://doi.org/10.1109/ISMSIT56059.2022.9932749