

FLUTTER TECHNOLOGY AND MOBILE SOFTWARE APPLICATIONS

Alexandru TĂBUȘCĂ¹

Cristina COCULESCU²

Mironela PÎRNĂU³

Abstract

Nowadays, many companies that develop mobile software applications have to develop the same application for the iOS and Android operating systems in parallel. Thus, the costs for creating the application will be higher because there have to be two teams of programmers. They have to work in parallel on the two platforms and to collaborate with each other, so that the developed application complies with requirements of design, planning and operation. The use of cross-platform frameworks allows the simultaneous development of an application for both the iOS and Android systems, generally using a platform-specific programming language, and the application code will be transposed and compiled into a dedicated code for each individual platform. Flutter technology is successfully used for the development of mobile applications, which contain a very good user interface and a technology which is simultaneously adapted to several platforms. Basically, the same code can be used for a web application that runs in a browser and can adapt itself to iOS, Android, Windows, MacOS or Linux platforms. Flutter is a development framework for front end, web development, UI, implemented by Google, being completely independent of the platform and can run both on IOS and Android systems, both as a web application and as a Windows application.

Keywords: Flutter, Dart, mobile software

JEL Classification: C88

1. Introduction

Flutter is a software development kit (SDK) created by Google under an open-source license. Initially, it was presented at the Dart Conference in 2015, and was called "Sky". It was designed as the main method of developing applications for the Fuchsia operating system (the operating system that later and gradually merged Android and ChromeOS). The latest Flutter version has three main components:

- Embedder – specific to the platform (iOS or Android)

¹ PhD Associate Professor, Romanian-American University, School of Computer Science for Business Management, tabusca.alexandru@rau.ro

² PhD Associate Professor, Romanian-American University, School of Computer Science for Business Management, coculescu.cristina@rau.ro

³ PhD Associate Professor, Titu Maiorescu University, Faculty of Informatics, mironela.pirnau@prof.utm.ro

- Engine (the Flutter rendering engine)
- Framework - the foundation library and widgets.

Applications developed in Flutter are written using the object-oriented programming language known as Dart. This programming language runs in a virtual machine written in C/C++. For choosing the programming language, Google considered the following criteria:

- developers' productivity
- using an object-oriented programming language
- predictability of the language, for a high performance but also a fast memory allocation, which is why Flutter is based on the fast and efficient allocation of small portions of memory [1-2].

A unique feature of Flutter technology is that it draws each pixel independently. Compared to React Native technology, it has internal widget collections – Cupertino for iOS and Material for Android but does not use OEM widgets [3].

Framework Dart	Material		Cupertino
	Widgets		
	Rendering		
	Animation	Painting	Gestures
	Foundation		
Engine C/C++	Dart	Skia	Text
Embedder Platform specific	Render Surface Setup	Native Plugins	Packaging
	Thread Setup	Event Loop Interop	

Fig 1. Flutter components

The rendering engine is written primarily in C/C++ and provides support for the Skia graphics library, but also links to development kits specific to both platforms, iOS and Android. In Flutter, Skia is an open source 2D graphics library that contains APIs common to a large number of hardware and software platforms.

The core library in Flutter is written in Dart and contains the core classes and functions used to build applications. Each element of the graphical interface is represented by a widget or a group of widgets.

2. Usage of Flutter Technology

To develop a Flutter application, we need to first install the software development kit (SDK) from its home web address [4] - the archive taken into account at the moment of the article elaboration is flutter_windows_3.3.8-stable.zip. The installation is done through the flutter_console.bat file inside the flutter folder. Also, to run Flutter commands in the windows console we need to add the path to the ".bat" script in the PATH system variable, see Fig 2.

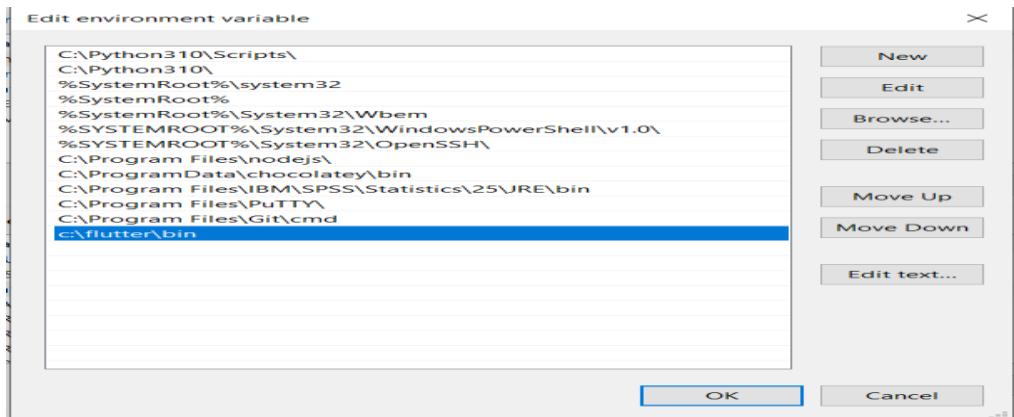


Fig 2. Adding of Flutter path

Flutter depends on the Android Visual Studio system through dependencies, so its installation becomes mandatory. The latest versions of Android SDK, Android SDK Platform-Tools and Android SD Build-Tools are installed with Android Visual Studio. Android Studio is created by Google and is an IDE used to develop Android applications [3]. In order to develop Flutter applications with Android Studio, it is necessary to install Flutter (see Fig. 3) and Dart (see Fig. 4) plugins.

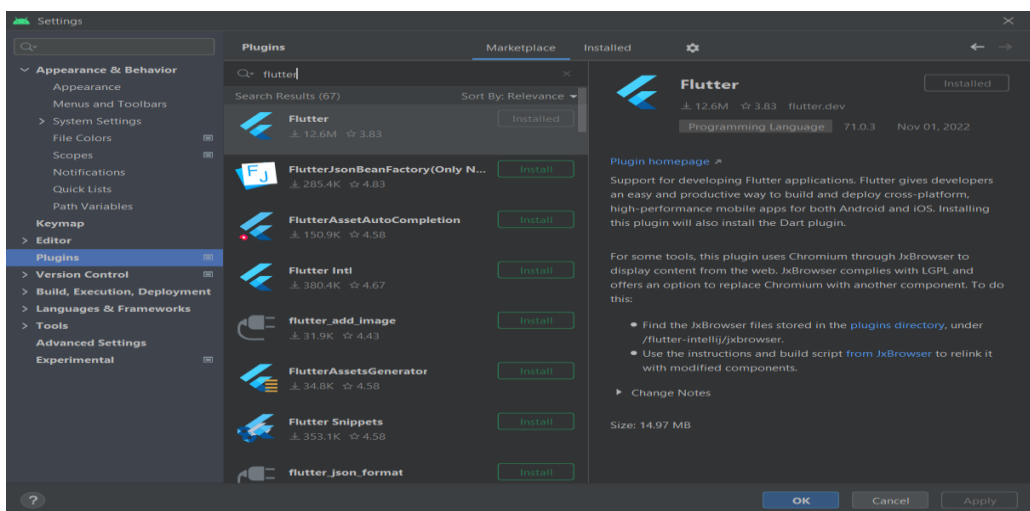


Fig. 3 Adding Flutter plugin to Android Studio

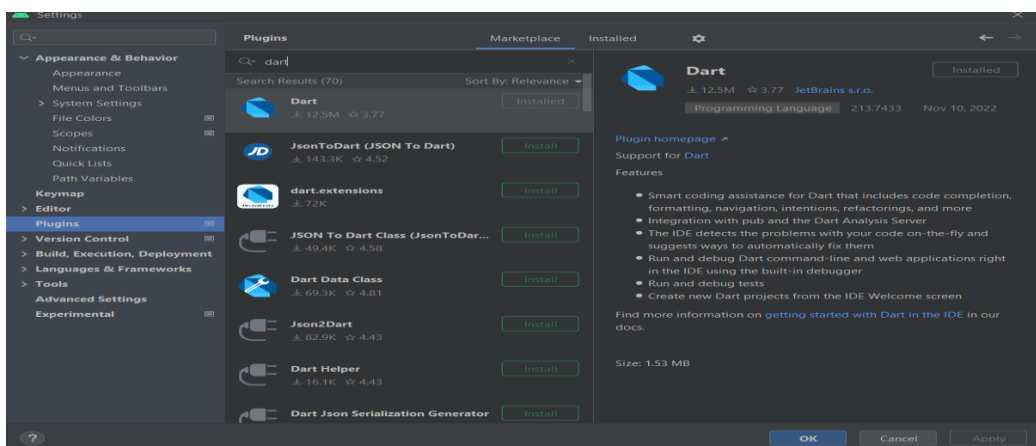


Fig. 4 Adding Dart plugin to Android Studio

After installing the Android Studio Application and adding the plugins (Fig 3. and Fig. 4.), the flutter doctor command is run in the console. This command will perform an analysis of the requirements and create a report with the information obtained, according to Fig. 5.

```
C:\Users\Mironela Pirnau>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.3.8, on Microsoft Windows [Version 10.0.19044.1889], locale en-US)
[!] Android toolchain - develop for Android devices (Android SDK version 33.0.0)
    X cmdline-tools component is missing
      Run `path/to/sdkmanager --install "cmdline-tools;latest"`
      See https://developer.android.com/studio/command-line for more details.
    X Android license status unknown.
      Run `flutter doctor --android-licenses` to accept the SDK licenses.
      See https://flutter.dev/docs/get-started/install/windows#android-setup for more details.
[✓] Chrome - develop for the web
[✓] Visual Studio - develop for Windows (Visual Studio Build Tools 2019 16.11.10)
[✓] Android Studio (version 2021.3)
[✓] VS Code (version 1.73.1)
[✓] Connected device (3 available)
[✓] HTTP Host Availability

! Doctor found issues in 1 category.

C:\Users\Mironela Pirnau>
```

Fig 5. Flutter doctor

With the help of the command *flutter create name_project* (see Fig 6.) a new project is created, and Flutter generates all the necessary files for a new application inside the directory where this command is run. The new directory structure created contains the files needed to generate the code in both Android and iOS.

```
C:\Users\Mironela Pirnau>flutter create flutter_project
Creating project flutter_project...
Running "flutter pub get" in flutter_project... 1,518ms
Wrote 127 files.

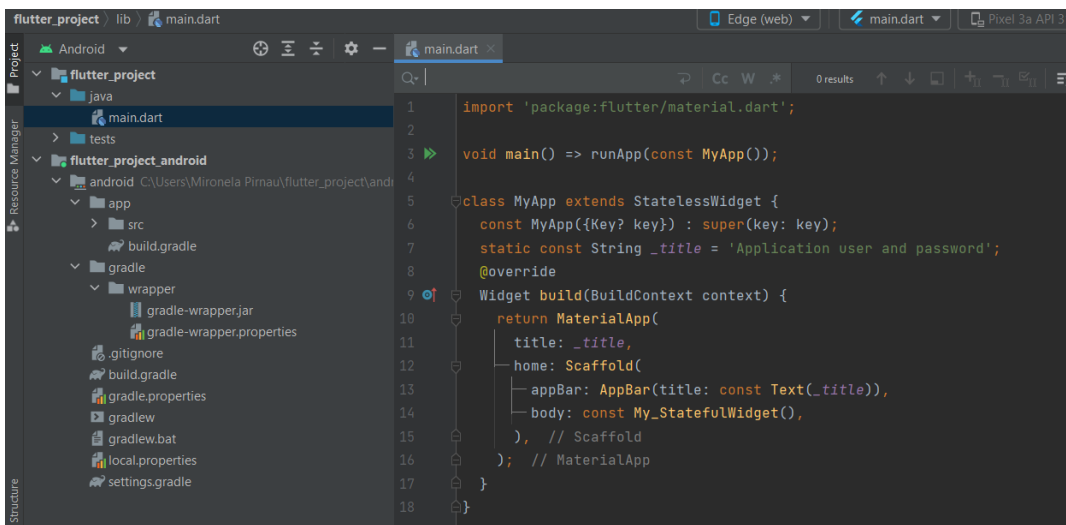
All done!
In order to run your application, type:

  $ cd flutter_project
  $ flutter run

Your application code is in flutter_project\lib\main.dart.
```

Fig. 6. Project creation within Flutter environment

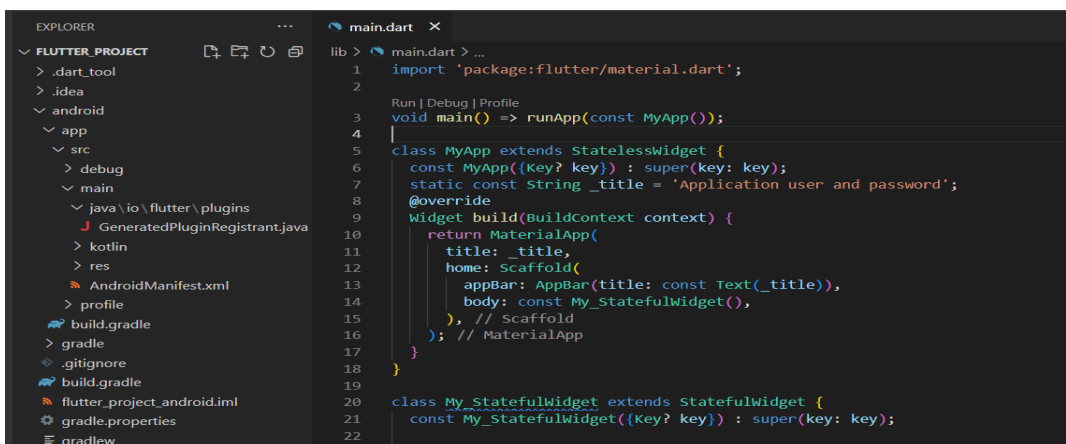
The project can be open with Visual Studio Code (Fig. 7) or Android Studio (Fig. 8).



The screenshot shows the Visual Studio Code interface. On the left, the 'Project' pane displays the file structure of the Flutter project, including folders for 'flutter_project', 'flutter_project_android', and various configuration files like 'build.gradle' and 'gradlew'. The main editor area shows the 'main.dart' file with the following Dart code:

```
1 import 'package:flutter/material.dart';
2
3 void main() => runApp(const MyApp());
4
5 class MyApp extends StatelessWidget {
6   const MyApp({Key? key}) : super(key: key);
7   static const String _title = 'Application user and password';
8   @override
9   Widget build(BuildContext context) {
10    return MaterialApp(
11      title: _title,
12      home: Scaffold(
13        appBar: AppBar(title: const Text(_title)),
14        body: const My_StatefulWidget(),
15      ), // Scaffold
16    ); // MaterialApp
17  }
18 }
```

Fig 7. Flutter Project opened with Visual Studio Code



The screenshot shows the Android Studio interface. The 'EXPLORER' pane on the left displays the project structure, including folders for 'FLUTTER_PROJECT', 'android', and 'app'. The main editor area shows the 'main.dart' file with the following Dart code:

```
lib > main.dart > ...
1 import 'package:flutter/material.dart';
2
3 Run | Debug | Profile
4 void main() => runApp(const MyApp());
5
6 class MyApp extends StatelessWidget {
7   const MyApp({Key? key}) : super(key: key);
8   static const String _title = 'Application user and password';
9   @override
10  Widget build(BuildContext context) {
11    return MaterialApp(
12      title: _title,
13      home: Scaffold(
14        appBar: AppBar(title: const Text(_title)),
15        body: const My_StatefulWidget(),
16      ), // Scaffold
17    ); // MaterialApp
18  }
19
20 class My_StatefulWidget extends StatefulWidget {
21   const My_StatefulWidget({Key? key}) : super(key: key);
22 }
```

Fig 8. Flutter Project opened with Android Studio

Testing the application on Android devices can be done by using a virtual machine created in Android Virtual Manager (Fig. 9) or by using a physical device connected via USB.

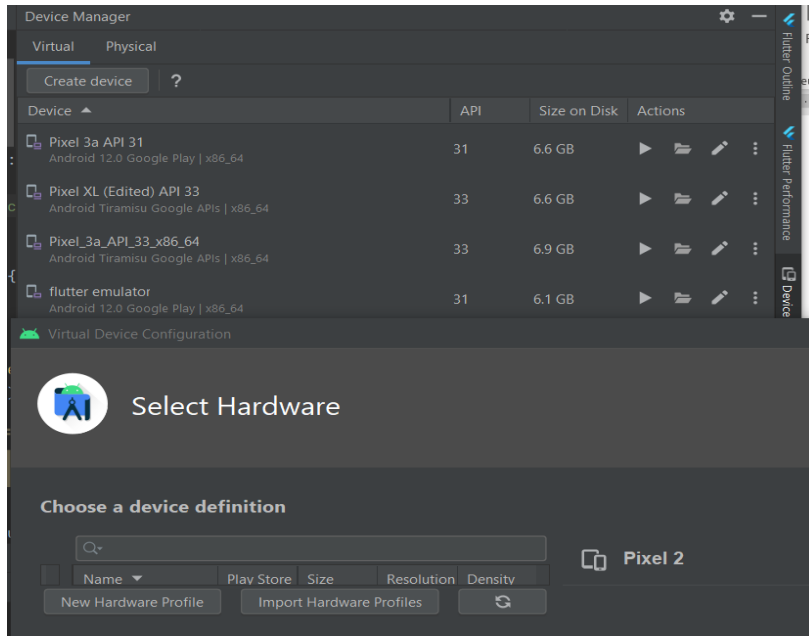


Fig 9. Create a new device within Android Virtual Manager

Android Virtual Manager is an emulation platform for devices using the Android operating system. It allows the creation of virtual machines of any version of the operating system, which will then run inside a Microsoft Windows window. Testing the application on the physical device is done by activating USB debugging in the phone's settings and connecting it using the USB cable. The user interface, after the correct running of the application, looks like in Fig 10.

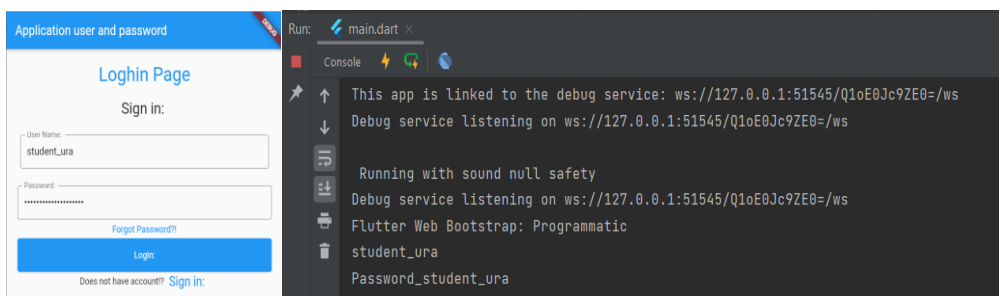


Fig 10. Interface of the current application

3. Widgets of Flutter

Each element of the graphical interface is represented by a widget or a group of widgets. In Flutter, a widget represents [3] a description of a graphical element that can be:

- a structural element: a text, an image, a shape, a button
- a styling element: font, color
- a schematic element: border, padding

Widgets form a composition-based hierarchy. Each widget is placed inside another, from which it inherits properties. There is no separate "application" object, this role being taken by the root widget. In Flutter, interaction with the user is given by updating the widget hierarchy, this is done by comparing the new widget with the old one, only the different elements being modified. Complex Widgets can also be created by combining several simple Widgets. In Flutter, the used widgets are not the native ones from the platforms the app runs on (Android or iOS) but they are specific representations, because Flutter contains a rich set of widgets, layouts and themes for each platform (such as Material Design for Android and Cupertino for iOS). In Flutter, widgets are divided into two categories:

- StatelessWidget
- StatefulWidget.

The **StatelessWidget** widgets are those widgets whose state does not change (such as for example: Text, Icon, IconButton) and do not depend on other components in the interface. A StatelessWidget widget has the role of describing part of the user interface by recursively building other widgets that form the user interface. The build process continues recursively, as shown in Fig. 11. In any Flutter application, the entry point is the *void main()* function in the *main.dart* file. This function calls, using the shorthand syntax, the *runApp* function with the constructor of the root widget class as a parameter. Flutter comes with the package called *material.dart*, which allows rapid development of an application because it contains all the basic widgets. Flutter can also import external packages that can be found at the link: <https://pub.dartlang.org/flutter>. In our case, *MyApp* is the root widget of type StatelessWidget that overrides the build method (see Fig. 11).

```
import 'package:flutter/material.dart';
void main() => runApp(const MyApp());
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  static const String _title = 'Application user and password';
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: _title,
      home: Scaffold(
        appBar: AppBar(title: const Text(_title)),
        body: const My_StatefulWidget(),
```

```
),  
);  
}  
}
```

Fig. 11 Listing of code

Building widgets is done by implementing the build function, as shown in Fig. 12, which returns a hierarchy of widgets. When building the user interface, Flutter will recursively call the build function of each widget returned by the initial function. The structure of an application's component elements in Flutter is tree-like, where each node in the tree is represented by a widget. As these do not store location information in the tree, the concept of *BuildContext* arose. Widgets rebuild each time the app's internal state changes. To accomplish this, the *build* method is called, which receives a parameter of type *BuildContext*, where context is an instance of this class. Determining the position in which a certain widget is found in the interface, but also the position where it should end up, is done using *BuildContext*, a context that helps to position it correctly in the application stack. The child parameter is often used in Flutter widgets and represents the element or list of elements inside the widget whose parameter it is.

```
Widget build(BuildContext context) {  
  return Padding(  
    padding: const EdgeInsets.all(10),  
    child: ListView(  
      children: <Widget>[  
        Container(  
          alignment: Alignment.center,  
          padding: const EdgeInsets.all(10),  
          child: const Text(  
            'Loghin Page',  
            style: TextStyle(  
              color: Colors.blue,  
              fontWeight: FontWeight.w500,  
              fontSize: 30),  
            )),  
        Container(  
          alignment: Alignment.center,  
          padding: const EdgeInsets.all(10),  
          child: const Text(  
            'Sign in: ',  
            style: TextStyle(fontSize: 24),  
            ...));  
      ]  
    )  
  );  
}
```

Fig. 12 Listing with partial code of the *build* function

The **StatefulWidget** widgets are widgets whose state is changed by the user's interaction with the application, but also by changing other parameters of the application. Each stateful widget has an initial state that contains information about the widget. For a user-created widget to have dynamic content, it must inherit the StatefulWidget widget, according to Fig. 13.

```
class My_StatefulWidget extends StatefulWidget {  
  const My_StatefulWidget({Key? key}) : super(key: key);  
  
  @override  
  State<My_StatefulWidget> createState() => _MyStatefulWidget_State();  
}
```

Fig. 13 Listing of code

If the elements that make up a widget change their properties, so that the framework identifies the difference between the states of the widget, automatically all the elements that have undergone a change will be rendered. If in a StatefulWidget widget there are widgets that do not change their state, such as *Text*, we must use the *const* suffix before it, because the application will remove constant elements from the rendering process and thus increase its performance, according to Fig. 14.

```
class _MyStatefulWidget_State extends State<My_StatefulWidget> {  
  TextEditingController nameController = TextEditingController();  
  TextEditingController passwordController = TextEditingController();  
  @override  
  Widget build(BuildContext context) {  
    return Padding(  
      ...  
      child: const Text( 'Forgot Password?!', ), ), Container( height: 50,  
padding: const EdgeInsets.fromLTRB(10, 0, 10, 0),  
child: ElevatedButton(  
  child: const Text('Login:'),  
  ...  
  )  
  )  
}
```

Fig. 14 Listing of code

Once built, all widgets are stored in the tree that stores the logical structure of the user interface. The tree stores state in the case of stateful widgets, this being necessary because widgets cannot store relationships to the parent element or child elements. During widget

construction, Flutter avoids traversing the parent chain using *InheritedWidgets*. This widget maintains a hash table for each element, thus avoiding repeated traversal of the same widgets. This hash table only changes when a new element is inserted into the tree.

4. Dart Programming Language – main features

Dart is an optimized programming language that allows the rapid development of a solution that can be implemented on several platforms. Being a type-safe language, it uses type checking to ensure that a declared variable corresponds to the data type for which it was initialized. By using interfaces, declaring the type of a variable became optional, and declaring it in a dynamic way reverted to the dynamic reserved word, which ensures that a variable is checked and validated at runtime, when the code is executed [5]

This scenario provides the possibility for a variable to be null on declaration, by using the sound null safety paradigm. Using this paradigm at runtime, Dart performs static code analysis, thus filtering and eliminating the possibility of throwing a null exception. The technologies used by Dart for compilation offer two possibilities for running the code:

- Native Platform - for applications that are intended to run on mobile and desktop platforms
- Web Platform - for applications that run in a web browser.

For web applications, both compilation modes translate to JavaScript code. Regardless of the method used for compilation and execution, code execution needs the Dart runtime. The runtime is responsible for the following critical tasks:

- memory management
- aggressive use of the variable validation system
- isolated process management so that Dart controls the main application process in an isolated process.

Multithreaded programming in Dart is done by using *async-await*, *isolated* constructs, as well as dedicated classes, like for example *Future* and *Stream* classes. The execution of the code within an application is done inside a hybrid execution thread, which at first glance looks like a thread, and after a more detailed analysis we will find that it looks like a process. This method of abstraction is called *isolate*. In dart, each *isolate* has a single encapsulated execution thread, i.e., a safe thread.

5. Flutter vs. React

The main competing cutting-edge technologies for cross-platform development are Flutter – launched by Google in 2018 - and React Native launched in 2015 by Facebook. Although React Native supports most APIs for iOS and Android platforms, it does not provide the ability to create custom elements as Flutter does by using widgets. There are numerous statistics that show that (as of May 2021), Flutter is closing in on React Native in terms of

popularity and usage [6]. A look at the Google Trends results shows that starting from 2020, Flutter has a global search frequency above React Native. Flutter is the most popular cross-platform mobile framework used by global developers, according to a survey conducted in 2021 [7]. According to this survey, Flutter was used by about 42% of software developers. The survey also identified that 33% of mobile developers use cross-platform technologies while the rest of mobile developers use native tools. A check in google trends of user searches between Flutter and React Native technologies identified that worldwide, the interest of users' searches for Flutter technology exceeded the interest of Internet users' searches for React Native technology (see Fig. 15).

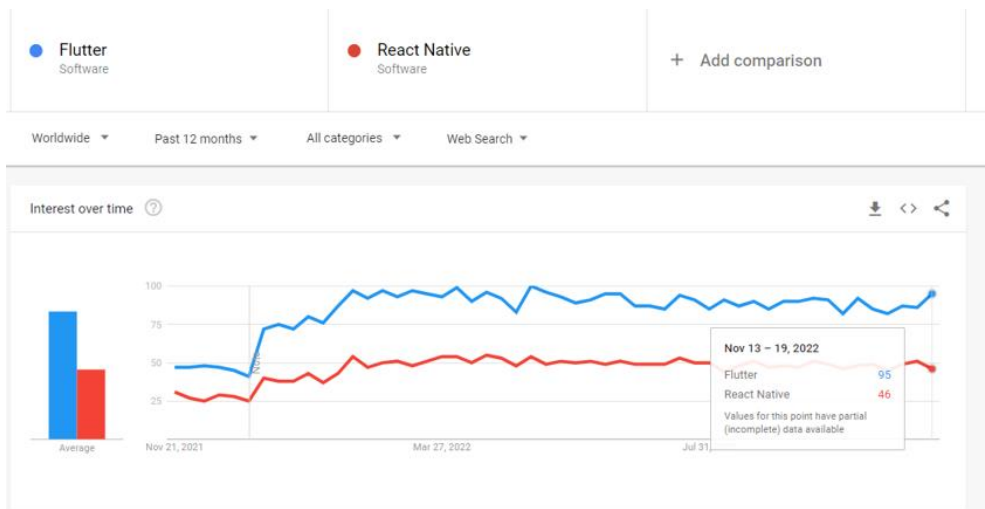


Fig. 15 Comparison of internet searches for Flutter vs React Native technology between 11/21/2021 – 11/20/2022

If we analyze user searches, but for the period 2018-2022, it can be observed that, from April 2020, the interest in Flutter exceeds the interest of searches for React Native (see Fig. 16).

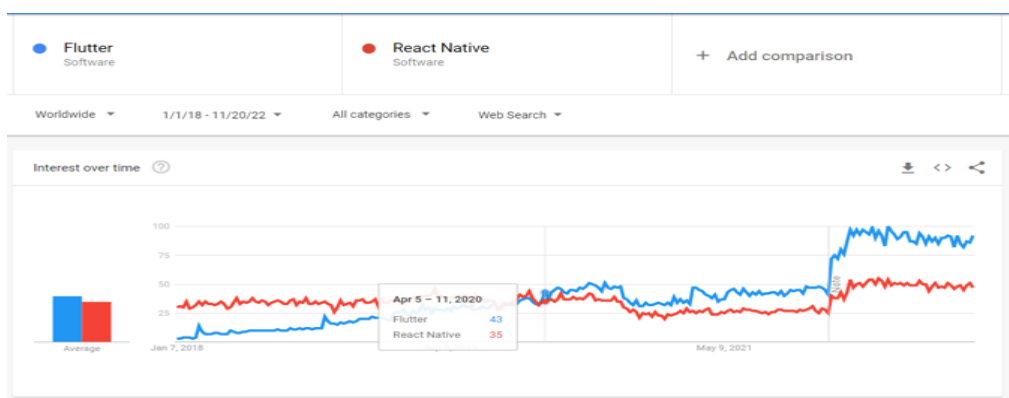


Fig 16. Comparison of internet searches for Flutter vs React Native technology between 01/01/2018 – 11/20/2022

6. Conclusion

In our case-study application, discussed and partially presented in the above code-listing figures, Flutter allowed code to be written only once and compiled simultaneously for both iOS and Android. Flutter technology continues numerous facilities for the development of interactive multi-platform applications based on a complete package of tools for creating complex elements from simple elements. In Flutter any structure is a widget, including the application is represented by a widget containing the other widgets. However, the space occupied by the Flutter application is relatively large, because it is necessary to wrap the Flutter library in it. In order to be able to keep the same design, layout and widgets on mobile platforms, Flutter does not use the native elements of the mobile operating system but renders them using the Skia library. This while facilitating cross-platform development adds extra storage space occupied by the application. Flutter is fully integrated with Google services, so it makes it easy to use the Firebase Realtime Database solution for NoSQL databases. Regarding the performance of an application made in Flutter, the following factors must be taken into account:

- Speed
- Memory usage
- App size
- Energy consumption [8].

References

- [1] Napoli, M.L. *Beginning Flutter: A Hands-on Guide to App Development*; John Wiley & Sons: Hoboken, NJ, USA, 2019.
- [2] Biessek, Al., *Flutter for Beginners: An introductory guide to building cross-platform mobile applications with Flutter and Dart 2*. Packt Publishing Ltd, September 2019.
- [3] <https://docs.flutter.dev/development/ui/widgets/material> - last access: 11.2022
- [4] <https://docs.flutter.dev/get-started/install/windows> - last access: 11.2022
- [5] <https://dart.dev> - last access: 11.2022
- [6] <https://insights.stackoverflow.com/survey/2021#most-popular-technologies-misc-tech> - last access: 11.2022
- [7] <https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours> - last access: 11.2022
- [8] <https://docs.flutter.dev/perf/> - last access: 11.2022

Bibliography

Sameha Rahman, <https://www.freecodecamp.org/news/https-medium-com-rahman-sameeha-whats-flutter-an-intro-to-dart> - last access: 06-12-2022

--, <https://www.javatpoint.com/flutter-dart-programming> - last access: 06-12-2022

TYAGI P, *Pragmatic Flutter*, ISBN 0367612097, Taylor & Francis Ltd. 2021

MARBURGER M, *Flutter and Dart*, ISBN 3836281465, Rheinwerk Verlag Gmbh 2021

HOSSEINI P, *Flutter: Zero to App*, ISBN 1080745076, Independently Published 2019

ROSE R, *Flutter & Dart Cookbok*, ISBN 1098119517, O'Reilly Media 2022

MEILLER D, *Modern App Development with Dart and Flutter*, ISBN 3110721279, De Gruyter 2021

BELCHIN M, JUBERIAS P, *Web Programming with Dart*, ISBN 148420557X, APress 2014

PAYNE R, *Beginning App Development with Flutter*, ISBN 1484251806, APress 2019

ALESSANDRIA S, *Flutter Projects*, ISBN 1838647775, Packt Publishing Ltd. 2020