УДК 004.4: 004.4

DOi 10.30837/ bi.2022.1(98).06

Ilona Revenchuk[1], Yevhen Ostashko[2]

[1] Kharkiv National University of Radio Electronics, Kharkiv, Ukraine,
ilona.revenchuk@nure.ua, ORCID: 0000-0002-5188-9538

[2] Kharkiv National University of Radio Electronics, Kharkiv, Ukraine,
yevhen.ostashko@nure.ua

MINIMIZING THE COSTS OF BUILDING COMPLEX CROSS-PLATFORM APPLICATIONS WITH FLUTTER AND FIREBASE

In our time one of the most if not the most common types of services is client-server ones. There are many frameworks written in many programming languages for both client applications and server ones, especially considering many different popular operating systems that are used by people today. With this many options, the question of which technologies to use to achieve the goal with minimum sacrifices is always relevant and in this article, we research what might be one of the most convenient technologies for client-server development not only for applications with relatively small amount of users but huge ones as well that allows for 1-2 developers to relatively quickly develop, release and maintain big systems with multiple client applications.

As a result of the work, there has been performed a research on the technology for developing cross-platform applications – Flutter, the easiness of its use, and its advantages compared to alternatives, on Firebase services for minimization of the costs to customize the server and on the combination of these technologies for modern client-server application development.

INFORMATION SYSTEM, SERVERLESS, FIREBASE, FLUTTER, DART, CLEAN ARCHITECTURE.

І.А. Ревенчук, Е.В. Осташко. Мінімізація витрат на створення складних кросплатформових додатків за допомогою Flutter та Firebase. У наш час одним із найпоширеніших, якщо не найпоширенішим видом сервісів є клієнт-серверні. Існує багато фреймворків, написаних багатьма мовами програмування як для клієнтських програм, так і для серверних, особливо враховуючи багато різних популярних операційних систем, якими користуються на сьогодняшній день. З такою кількістю варіантів питання про те, які технології використовувати для досягнення мети з мінімальними витратами, завжди актуальне і в цій статті ми досліджуємо технології, які можуть бути одними із найзручнішими для клієнт-серверної розробки не лише для додатків з відносно невеликою кількістю користувачів, але і для великих, що дозволяють 1-2 розробникам відносно швидко розробляти, випускати та підтримувати великі системи з кількома клієнтськими програмами.

За результатами роботи було проведено дослідження технології розробки кросплатформних додатків – Flutter, простоти її використання та переваг перед альтернативами, сервісів Firebase для мінімізації витрат на налаштування серверної частини та взаємодії цих технологій для розробки сучасних клієнт-серверних програм.

ІНФОРМАЦІЙНА СИСТЕМА, БЕЗСЕРВЕРНА, FIREBASE, FLUTTER, DART, ЧИСТА АРХІТЕКТУРА.

Introduction

Cross-platform development is the most effective when it comes to client applications since you use one code base for multiple applications simultaneously increasing amount of people the app is available for and amount of people that are needed to develop it. Fewer code results in fewer bugs, their faster resolution, and quicker implementation of new features. Respectively fewer hours of work are required for the whole cycle of development that results in the same goal being achieved with a lot fewer resources than if each application had been developed by separate teams using native development frameworks. This approach has been spreading rapidly since its first introduction and is one of the most popular ones at the time of the article.

Today many such cross-platform frameworks exist and so the problem of choosing the most suitable one arises again. Different frameworks support different operating systems, use different programming languages, and have communities of different sizes. That is where the Flutter framework, that we are researching in this article, shines in theory, not only is it the most modern cross-platform framework at the time of the article, which means it incorporates the most modern approaches in software development it also supports 6 of the most popular operating systems with the single codebase (Android, iOS, Web, Windows, Linux, MacOS) [1]. The practical side of this framework is what needs to be researched to find out if the framework can be used in most use-case scenarios.

For the client-server application, the client alone is obviously not enough so the question arises of what technology to use for the server side. Just like with a client application, many options are available for back-end development, but the simplest one of all is to use the so-called “serverless” approach. That approach means using existing services such as Amazon web services or, in our case, Firebase, that do most of the work, such as scaling, security, and open API for us. Not having to set up and program the server site has its disadvantages, which we will talk about later, but in most cases, such an approach saves a lot of time and money speeding up the development process and moving release deadlines earlier. Since configuring the server using the “serverless” approach does not require coding, minimum training and education are

required so the same experienced developers that are developing the client application can take on this job as well subsequently making the need for front-end and back-end teams to use the time to communicate and sync their work non-existent.

1. How Flutter works

During Flutter development, applications run on a virtual machine, which allows you to reflect changes in the codebase without the need to completely recompile the application, which speeds up the development significantly, especially compared to native mobile development (Android/iOS). For the release version, Flutter applications are compiled directly into machine code, regardless of whether it is on Intel x64 or ARM, or even on JavaScript if we compile for the web, and as a result, we get an installation file specific for the target platform. The framework is open access and has an active ecosystem of third-party packages that add functionality to the basic framework [2][3].

1.1. Structure

Flutter exists as a series of libraries divided into different layers (pic. 1) that can be replaced if needed [4]. No layer has privileged access to the layers below it, and every part of the framework is designed to be replaceable or completely removed as needed.

The lowest layer, called Embedder, is the layer that procures the cross-platform capabilities of the Flutter framework. It's written using the programming language corresponding to the target platform (Java and C++ for Android, Objective-C for iOS, etc.), so each platform that Flutter supports has its own embedder. This layer ensures the native thread setup for Flutter to work and contains native plugins to allow the framework to use the platform's native capabilities, such as a camera for mobile and many more. Also, this layer converts the application interface configuration, provided by the higher layers, into native drawn elements.

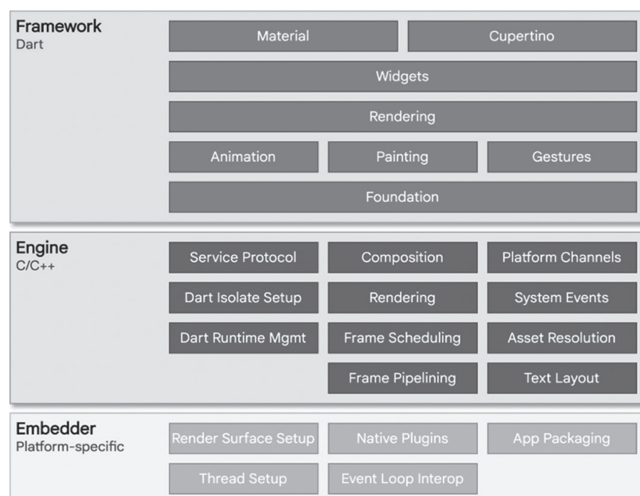

Fig. 1. Flutter's architectural layers

The next layer above the Embedder is a Flutter Engine that is written in C++ and supports primitives, required for the Flutter applications. The engine rasterizes the composite scenes each time a new frame needs to be rendered. It provides a low-level implementation of Flutter's core interface, including graphics, text composition, file system access, plugin architecture, and a Dart compilation toolchain. The engine is available for frameworks through the corresponding package – dart:ui, which wraps the underlying C++ code in Dart classes.

The top-level layer is the one developers interact with the most – the Framework layer. This layer contains various libraries for interface build that are abstractions to the lower-level components to ease the development process, speeding it up consequentially. If going from the lower level of the Framework to the higher the following structure is present:

– fundamental classes and building blocks such as animations, drawing, and gestures;

– rendering layer, which provides abstractions for working with the layout;

– widget layer, which is an abstraction of the composition. Each rendering object from the rendering layer has a corresponding widget class, and this layer also allows you to define combinations of widgets that can then be reused;

Material and Cupertino libraries that provide widgets typical of Material and iOS design languages.

1.2. Rendering

When analyzing cross-platform frameworks, not the last thing they pay attention to is its performance. Surprisingly, Flutter's rendering technology gives it a level of performance comparable to a single-platform framework. In the example of mobile cross-platform frameworks, an abstraction is usually created over the interface libraries of specific platforms. These native libraries usually convert classes to a view on a Canvas object. The interaction of cross-platform framework code with native creates many additional processes that slow down the application. Flutter, in turn, minimizes such abstractions by not using native rendering libraries in favor of its widget structure. The Dart code that draws the Flutter app view is compiled into native code that directly uses drawing on the canvas. Flutter uses Skia technology for rendering, just like Android. The framework engine also includes its own private Skia repository, allowing developers to update their applications with the latest improvements without having to update the native operating system version.

The principle of Flutter is "what simple is fast" [5]. The flow of data from the event to the rendering instructions can be seen in fig. 2.

So, the rendering process goes like this:

– when Flutter wants to redraw its fragment in the corresponding widgets, a special "build" method is called;

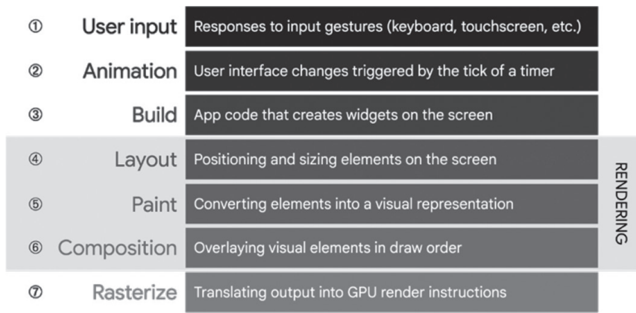

Fig. 2. Rendering pipeline

– during the build phase, Flutter converts the widget structure into a corresponding tree of elements (fig. 3) with one element per widget. Elements are divided into component elements, which simply contain other elements, and elements of render objects, which participate in the drawing phase;

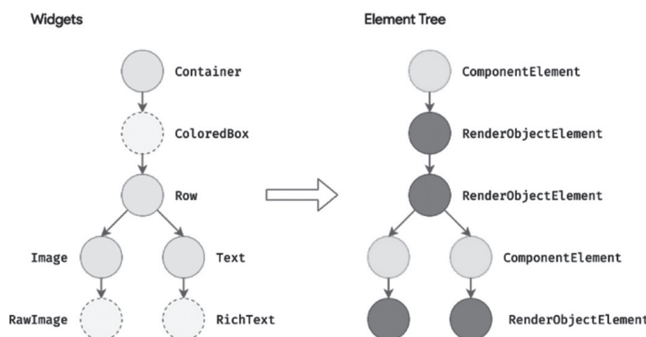

Fig. 3. Transformation of the widget's structure into a tree of elements

– during the construction phase, Flutter creates or updates an object that inherits from RenderObject for each element of the render object in the tree (fig. 4);

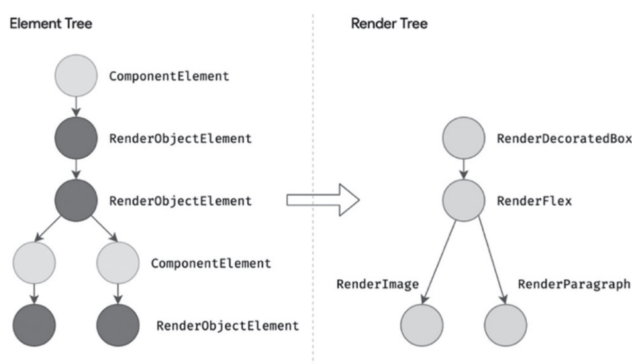

Fig. 4. Transformation of the elements' tree into a tree of render objects

Next, the framework traverses the render tree depth-first and passes down size constraints from parent to child. Then the descendants respond by passing their sizes to the parents, which must be within the received limits (fig. 5). After that, each object is ready for drawing.

The root of all render objects is the RenderView, which represents the full output of the render tree.

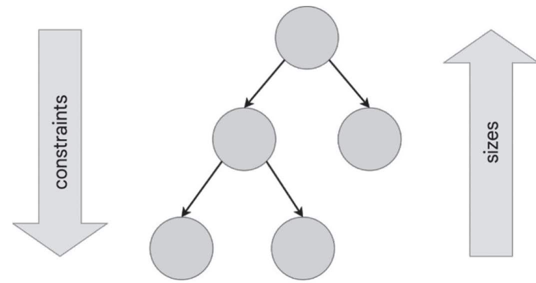

Fig. 5. Passing constraints and dimensions in the render object tree

2. Firebase services

Firebase contains many special services such as Machine Learning, that are not typically used in an average service so they will be omitted and the most useful and common ones will be described. The Firebase architecture consists of several components that interact with each other to provide a framework for developing and hosting web and mobile applications [6]. The following are useful for our purpose:

– Firebase Cloud Storage is an object storage that is available on Google Cloud platforms. When this service is added to Firebase, the application gets access to Google's security measures and the ability to safely download or upload data;

– Cloud Firestore is a scalable, flexible database service for server, mobile, and web development. It serves as a NoSQL document database. Provides "out of the box" access to web sockets, that is, the ability to reactively update data in the client application on database changes. It also has its own system of security rules, which is written in a JavaScript-like programming language and controls access to collections and documents according to user needs;

– Authentication – can be used for simple user authentication through login and password, mail, phone number, and various social networks;

– Hosting – hosting service for web content. It can be combined with cloud functions to develop and host your own microservices;

– App Distribution – allows you to distribute pre-release versions of applications and configure a list of testers for them;

– Cloud Functions – provides an opportunity to create both HTTP endpoints and trigger functions that respond to prescribed events, such as creating a document in a certain collection. Useful when services that are needed to implement some functionality in the application have webhooks, for example, the Stripe payment management service;

– Remote Config is a key-value type cloud data storage for managing some flags and parameters necessary for the client application, for example, the minimum version of the application to which the user will be forced to update the application;

– In-App Messaging allows you to create a system for sending push messages to device tokens, usually stored in Cloud Firestore in the user's document. It is especially useful on the mobile platform, where push notifications are most common;

– Crashlytics – allows you to track errors in applications connected to the service. May contain error codes and messages, and also tracks app crashes;

– Performance – allows you to see how expensive certain processes in the application are, on which platforms, and with which data. Useful to use for complex functionality to test the impact of changes on performance;

– Google Analytics – allows you to create many events that can happen in the application and analyze them with the help of real users. In this way, it is possible to highlight the priority functionality, which is used by almost no one, and plan the development based on real data;

– Dynamic Links – allows you to develop so-called "deep" links that will send the user to, for example, a corresponding mobile application, where it will be processed, instead of a web page;

– AdMob allows you to add integrated Google advertising to the application, immediately providing a method of monetizing the service.

All of the Firebase services listed above would be useful for a typical service providing not only the means of implementation of the basic functionality but supporting services to help improve the user experience as well.

3. Flutter-Firebase communication

Since both Flutter and Firebase belong to the same company – Google, the interaction between these technologies is made as accessible as possible [7]. It is also important that any updates and changes in Firebase will be adapted to the Flutter context in priority, which accelerates the improvement of the service thanks to new functionality and bug fixes.

Flutter has all the necessary libraries for a comfortable interaction of the client application with Firebase services on the website, which distributes both custom and official packages for the framework (pub.dev). They abstract the work with the Firebase API, which makes it much easier to use these services without creating a mechanism for working with Rest, as you would have to do with a regular server. The same packages can also be found for most modern frameworks such as React, View.js, Java Spring, etc.

4. Flutter's performance

Before jumping into using the Flutter framework with all of its stated incredible capabilities a company should still research all of the problems the framework may produce and the most common question is if the productivity tradeoff is small enough to warrant the use of the framework instead of the native solutions, that has been present on the market for decades and has been tested and improved with time more than any other solution for the platforms.

The Flutter framework makes the whole development process many times easier, faster, and consequentially – a lot cheaper, but if the result product's productivity is interfering with users' experience it can make this product very unsuccessful on the market, making all of the spent recourses on the development go to waste and the saving of a big part of them, choosing a cheaper development process, a bad decision that led to nothing but losses. That risk is the reason why relatively new technologies usually take years to be incorporated into big companies and are mostly used by small companies that do not have a lot to lose. In a sense, these small companies are stress-testing these technologies on real projects for big companies to incorporate if proven effective.

Flutter is not an exception, though its rapid growth in popularity is superior to most, not least because its origin is one of the biggest companies in the world – Google, which also used it to rewrite some of their popular, extensively used products, such as Google Pay. These products are performing incredibly in production and have been proven to be as effective as their native counterparts with the benefit of a lot simpler codebase, which inclined many to try Flutter for their projects.

Real big projects written using Flutter prove its effectiveness by themselves, but knowing exactly how different Flutter's productivity is compared to the native apps is essential, especially if the project has to be as fast as possible, for example, a video game would need every frame per second it can get for the most optimal experience of the user even on the old devices.

The precise comparison of the Flutter and native Android apps has been done and published in an article before [8] and it presents a relatively accurate representation of the productivity differences between the Flutter and other platforms' native SDKs since the technology, that Flutter uses, is practically the same for every platform and boils down to drawing elements on the platform provided canvas.

In the mentioned article the study concludes with the native app performing faster than a Flutter one in every aspect, except for the threads rendering, the fact that there is an aspect of the cross-platform framework that works faster than the native counterpart is amazing, but the rest of tests resulted in the Flutter's loss. But that does not mean that Flutter's productivity is an issue, since looking closely into the collected by the author data the only big difference in performance, that can be spotted by the naked eye is app startup, where Flutter seems to be performing heavy tasks, the rest of tests show a minuscule difference of average 50ms difference and the most common tasks that users will face perform with the difference in a smaller range of 10-30ms.

In conclusion, the Flutter apps perform efficiently enough for the absolute majority of cases in the app development sphere, since most apps tend to not perform complex user interface tasks.

It is also important to note that the huge Flutter popularity led to a rapid increase in the corresponding team's funding and the fact that it's an open-source project allowed countless developers to improve different aspects of the framework themselves. Many of those changes have been officially incorporated into the framework's stable version after the review, so Flutter's efficiency increases very fast compared to most other frameworks which means the productivity may increase further in the future.

5. Cloud Firestore

Since Cloud Firestore has a role of a database, which would be storing most of the data of the absolute majority of apps using it it's important to know how it works and how productive it is compared to the alternatives [9][10].

5.1. Servers' locations

When it comes to the productivity of a server the first thing that comes to mind is its location. The closer the server is to the user the faster the response time will be. When choosing the location for the Google Cloud Platform services, which include Cloud Firestore, the developer is presented with two options: a multi-region location or a regional location.

Multi-regional location maximizes the availability and durability of your database. It stores data replicas on multiple servers across multiple regions, some regions, called witnesses only participate in the process of replication. This method allows having consistent data and request times across multiple regions, which is recommended for global apps. At the time of the article, 2 multi-regions are available: Europe and the United States.

Single region location maximizes the efficiency and cost of the data transfer for the specific region. Using a single region location would be more effective for the apps, that are not designed to be global and are used in a specific region, since request time increases the farther you are stationed from the selected region. Multiple regions are available across America, Europe, Asia, and Australia.

Since Firebase charges the user by the number of documents you read, write and delete obviously a multi-regional location would charge more for every operation since the data needs to be replicated to support effective data accessibility across multiple regions.

5.2. NoSQL Advantages and Limitations

Since Cloud Firestore is a NoSQL document-oriented database it is important to understand the principles behind such a data-storing approach since it would not be effective for every service to use it.

The main characteristics of NoSQL Databases architectures are [11]:

– schema-less structure;

– permitting data representations to grow effectively and dynamically;

– horizontal scaling, by data replication of the collections and sharding over massive clusters.

The main advantages of NoSQL Databases:

– volume: data at rest – terabytes to exabytes of existing data to process;

– velocity: data in motion – streaming data, milliseconds to seconds to respond;

– variability: data in many forms – structured, unstructured, text, etc.;

– veracity: data in doubt – uncertainty due to latency, deception, ambiguities, etc.;

– not built on tables and does not employ SQL to manipulate data;

– can handle unstructured, messy, and unpredictable data;

– helpful for working with large sets of distributed data.

The disadvantages are the following:

– disorganized data, it's harder to query needed data;

– the lack of JOINs – the lack of relations between data makes the developer send multiple requests to access some data;

So, in conclusion, if the data for the service needs a strict structure for maximum efficiency it would be recommended to use a conventional relational SQL database, but that does not mean that Cloud Firestore cannot be used for such cases. Even though the documents' structures are not strict and inside one collection documents with different structures are allowed to occur the structure can be enforced manually through the developers' efforts.

In the Cloud Firestore on the top level are only collections, each collection can contain only documents, but each document can contain a subcollection. The structure can be achieved through the client application's effort by defining strict models and paths for data to be stored. That does not solve the problem of fetching multiple different documents through one request but it makes it possible to predict the documents' contents and locations. For most tasks that may occur in a typical service that should be enough.

6. Experiment

Any theory needs to be tested in practice to have value as research. In our case, a test service with a client application written using the Flutter framework that uses most Firebase services to function needs to be developed and tested.

6.1. Experiment steps

The following tasks need to be performed:

1. basic Flutter application setup that includes app architecture, presentation layer architecture for state management, navigation, and dependency injection setups;

2. check how native applications for supported platforms look and feel;

3. create a Firebase project and connect it to the Flutter app;

4. add registration and authentication using the Firebase authentication service since it's the most common use-case;

5. add database usage with Cloud Firestore, Cloud Storage and Real-time database, test their security, add rules;

6. add Cloud functions: HTTP ones and triggers for collections' documents' states changes;

7. add Remote Config usage, push notifications using In-App Messaging, dynamic links and adds using AdMob;

8. add analytics using Google Analytics, Crashlytics, and Performance;

9. distribute mobile apps using app distribution and host web apps using Hosting.

Every listed development task needs to be tested on the speed of implementation, its complexity, and if the result is satisfactory in terms of performance and security.

6.2. Experiment results

As a result of the experiment, a Flutter test application has been developed with Firebase services covering the need for a database, file storage, authentication, etc. Below the experience from the development process of each experiment step in the order presented in the previous section is listed.

6.2.1. Flutter application architecture

The Flutter community has created many libraries that make initial project setup and things like navigation, dependency injection, and interface state management easier. For the test project, a flutter_modular package has been chosen as it makes implementing a modern app architecture easier. The package allows us to divide our app into different, separate modules, each module has its own injected dependencies and they are injected and disposed of depending on the navigation state, so if there are no screens, that belong to a specific module, in the screens stack the dependencies, if were injected prior, are disposed of.

During development, there were discovered 2 things that flutter_modular lacks, that needed to be implemented separately by downloading package and changing it locally.

First, nested navigation, which allows us to open screens inside other screens, that we use for the main screen with the bottom navigation bar does not have caching. That means each tab is loaded every time the user navigates to it, which is a bad practice as many unnecessary requests are sent.

Second, there was no ability to push several screens at the same time. This is crucial when it comes to opening dynamic links and push notifications, as when we lead the user to the corresponding screen the screens back-stack needs to be filled with the correct items.

The basic app structure in a final solution is as follows: an app is divided into features, and each feature has its own package, so it's physically separated from the other features. The feature has 3 packages inside for each layer according to the clean architecture principles. The presentation layer also has a navigation package, that contains the required info for other features to navigate into it if needed.

For the presentation layer architecture for state management, a BLoC (business logic component) was chosen since it's the most popular solution at the moment and perfectly solves the problem of separating the UI components and business logic.

The version of Dart at the time of the article does not have so-called sealed classes like, for example, Kotlin programming language does. But surprisingly it could be solved by the code-generation package called freezed. It automatically generates boilerplate code for models, such as serialization/deserialization, and comparison, and allows the addition of a similar to sealed classes behavior to a class. This has been extensively used for data transfer objects and navigation information so that we can pass needed parameters to a screen through such a class. Other code generation packages were also used, for example, to implement localization only JSON files with strings needed to be filled manually, and the needed code was generated with a corresponding command.

6.2.2. Result client applications

The Flutter framework out of the box contains most if not all widgets for both consistent Android and iOS development in the material and Cupertino packages respectively. Many animations and interactions, such as scroll, are native by default and are performed differently depending on the platform, on which the application is launched. So it is very easy to make the application feel natural on different platforms. Sometimes however behavior needs to be implemented manually. We can use different widgets and animations depending on the platform, so to achieve the maximum native feel some understanding of Android and iOS platforms' design conventions needs to be present, so the cases that are not handled automatically can be implemented manually (often easily with already pre-defined widgets from the corresponding package).

A similar situation is with desktop applications, however, the ability to build Flutter for desktop is fairly new at the time of the article so even though there are already several desktop applications in a production written in Flutter many features remain absent. For our needs, the firebase support has not yet been implemented for the desktop and only MacOS firebase support is in beta now so we do not test those.

For the web application, some changes need to be implemented in the code. Both mobile and web have platform-specific packages, using which on an unsupported platform will lead to an error, so if their usage is required a condition, that checks the platform needs to be used. Apart from that launching an app on the web is easy, however, at the time of the article, the Flutter web is not ideal for every use. The Flutter web feels like a mobile application on the web, even text selection is a fairly recent feature, so there are only the following use cases, when it can be useful:

– Progressive Web Apps;
– Single Page Apps;
– Existing Flutter mobile apps.

Flutter is not suitable for static websites with text-rich flow-based content.

6.2.3. Firebase setup

The firebase project is easily created on the corresponding website. It's important to know that by default the project is restricted and you need to connect a bank card to it so its full capabilities, like Cloud Functions, are unlocked. The charges are not made until the specified threshold of usage, since this is a test app and the threshold is quite high we do not need to worry about it.

Next comes the connection of the firebase to our Flutter application. At the time of the article only Android, iOS, and Web Platforms are supported with the MacOS platform in Beta. Each platform needs its separate configuration, but the usage in the Flutter app will be with the same code.

Finally, for us to be able to connect to databases and authentication services we need to set up them. For databases and storage, the server location needs to be chosen, the west Europe multi-regional location has been chosen. For the authentication, we need to choose providers, the email/password and google has been chosen.

6.2.4. Authentication

For every firebase service, a corresponding official Flutter package has been made. For the authentication the package is firebase_auth. This package handles the whole authentication process, and user caching and provides a stream of authentication state changes. It also supports password reset, where an email, defined in the Firebase console, will be sent to the user.

The usage of this package is simple, for email/password authentication only a simple method with email and password needs to be sent, if the user does not exist he is created automatically. To store the user's info a Cloud Firestore needs to be used, the user's document has the same id as returned from the authentication method.

For the google sign in an additional package google_sign_in is required, it provides a native google authentication process and returns necessary credentials, that are later used with the firebase authentication.

6.2.5. Cloud Firestore, Storage and Real-time database usage

For the cloud firestore the cloud_firestore package is used. To gain access to the document or collection the path can be inputted as a string or built with a builder-like pattern. Cloud firestorm support web socket out of the box, so access to any document or collection of documents can be responsive to changes, reflecting any changes in the database in the application without the need for data refresh requests. It makes the development process faster and more efficient since any change to the document is reflected automatically, without the need to handle such cases manually.

Such a responsive approach has a disadvantage in the case of data pagination. When we request a page of data using a web socket loading the next page can be done in one of two ways:

1. creating a new connection to the next page of data and saving it separately. Such an approach makes it burdensome to handle all of the open connections and also it can handle items addition and removal incorrectly;
2. disposing of a previous connection and creating a new one with the data amount restriction increasing by a one-page count. Such an approach works perfectly and has no incorrect behavior, however, it re-requests previously loaded pages of data again, and any operations on documents are charged.

So, in terms of cost-effectiveness, it would be better to request data statically and reflect its changes only with a refresh sequence. In that case, we trade of user experience for the lowering cost of the system to run. The decision of whether the data should be loaded using web sockets or an HTTP request should be decided for each situation separately.

The security rules for the operations with a Cloud Firestore database are made in a JavaScript-like language [12]. To add a rule, you add a path to the document you want to add a rule to (for example "/users/{userId}") and define conditions for read and write operations. The rules use the authenticated user's id so you need to use the Firebase authentication service to able to establish flexible security.

Cloud Storage works in a similar way to the Cloud Firestore, only access to the files is made through the public URL, which is acquired with a special method of the corresponding package – firebase_storage. The URL will grant complete access to the file that it leads to. The rules, which are defined the same way as for Cloud Storage, only affect the method, that returns a public URL for the file, so if, for example, you need to fetch multiple pictures to show the user you would need to call the URL method multiple times for each file for the rules to be applied. That is an ineffective and long process, so saving the URL of the file in the corresponding document and generating this URL with the file upload would be a more effective,

scalable approach. If a file needs to be protected it would be better to encrypt the URL than use the first method.

A real-time database (RTDB) is a predecessor of a Cloud Firestore. It's an efficient, low-latency solution that requires synced states across clients in real-time [13]. Cloud Firestore scales better and has richer and faster queries, so using it would be better to use it instead of RTDB. The only viable use for an RTDB that has been implemented in the test application is controlling user presence. In cooperation with cloud functions, we can track when the user is online since unlike Cloud Firestore RTDB has an onDisconnect option. When the user's internet connection disappears the cloud function changes the user's state in the corresponding document, allowing us to know the user's presence. That can be used, for example, to know if we need to send the user a push notification for, for example, a new message in a chat. If the user is already present in a chat the notification should not be sent.

6.2.6. Cloud Functions

The Cloud Functions are a convenient way of adding side effects to operations via triggers and creating HTTP endpoints if needed. For example, we can send a push notification on document creation without the need to perform this operation by hand from the client and spend the user's traffic and time. The triggers for document creation, update, and deletion are available. Also, there is a trigger that is called on all 3 of those events – write trigger.

As to HTTP cloud function, one was used in our case – a webhook for stripe operations. To know if the payment or any other stripe event was completed successfully or with an error a webhook, which is basically an HTTP endpoint, needs to be defined in the Stripe console. This webhook will be called when defined in the Stripe console events occur. In our case, a user's document contains info on an ongoing payment process and the status changes based on the data received in the webhook.

Important to know that Cloud Functions have unrestricted access to all Firebase data and security rules do not apply to operations inside them.

6.2.7. Remote config, In-App Messaging and AdMob

Remote config is just a cloud key-value pairs storage, that lets you change the app's behavior without the need to release an update. For example, the app's theme change depending on the season can be handled that way.

Push notifications have been implemented in the following way: a user's device token is fetched and stored in their document on the authorization. When an event, that needs a push notification to be sent occurs we create a document in a notifications collection in Cloud Firestore with all the needed info for the notification, such as the sender and receiver id and the notification type. The Cloud Functions trigger on notifications document creation uses an In-App messaging API to send a push to the device token of the receiver [14].

For AdMob, the google_mobile_ads package is used. The following advertisement types are available:

– banner – a rectangle above or below the screen;

– interstitial – full-screen ads that need to be closed by the user;

– native – flexible type, that allows placing ads wherever the developer wants, so ads are effectively integrated into the app's UI;

– rewarded – an ad that rewards users for watching short videos and interacting with playable ads and surveys.

Advertisements are a great way of monetizing your app as long as they do not severely interfere with the user's experience.

6.2.8. Analytics

Google Analytics, Crashlytics, and Performance are all simple to implement via their corresponding Flutter packages. The only thing needed for them is a Firebase project setup, which has already been done earlier.

Google Analytics allows us to send any kind of event with some simple key-value data with them. For example, when the user enters a screen, we can send a corresponding event with the specific screen data. When a sufficient amount of data is gathered the app usage can be analyzed and development priorities set based on users' experiences [15].

Crashlytics service is Google Analytics for errors and crashes. It tracks both developer-defined errors, that are sent manually through the API and app crashes. The errors then can be seen in the corresponding tab in a Firebase console and dealt with accordingly. In the process of testing this service, it was discovered that there is sometimes not enough info on an error or crash, which makes fixing it hard, so maybe it would be better to use a separate service. Also, Crashlytics does not allow for user reports, which is a minus.

Performance is as simple as sending a request on some operation start, and another one on its end. Useful for complex operations to check their performance on many different users' devices.

6.2.9. App distribution and web hosting

App distribution allows to conveniently distribute of mobile iOS and Android applications without the need to go through their respective console setups. You can invite testers through their emails so that they can gain access to the application. The advantages of using Firebase App Distribution compared to platforms' consoles are the following:

– managing both iOS and Android pre-release distributions from the same place;

– early releases can be delivered to testers quickly, with fast onboarding, no SDK to install, and instant app delivery;

– combined with Crashlytics we can get insights into the stability of test distributions.

Hosting your web application can be done on a custom domain, which is usually bought, or on a domain provided by Firebase. Firebase also allows the creation of Beta channels, allowing you to host test applications on a temporary domain. The hosting needs some simple console application to set up your project after which you can easily deploy your application with a command.

Conclusions

As a result of the study, research has been conducted on Flutter and Firebase technologies and their synergy.

Flutter has been tested for the satisfaction of modern application development standards and the speed and ease of the development cycle.

A test service has been developed with the client side on Flutter and Firebase as a backend. Most Firebase services useful in a typical service have been used. It required 1 developer with only Flutter expertise to develop in a short period of time a fully functioning service with most typical features, that can be distributed for iOS, Android, and Web platforms.

In conclusion, a Flutter and Firebase combination has been proven a very cost-effective approach for developing client-server applications, especially for simple ones, compared to a more conventional approach with multiple native and server applications.

References

[1] *Vaibhav Patil.* Flutter-Modern and Easy Technology to Build Applications // ResearchGate. – 2023. – Vol. 11, No. 2, P. 458-459.

[2] *Artem Velykyy.* FLUTTER: A FULL INTRODUCTION TO THE FRAMEWORK // Axon. – 2020. – P. 2-3.

[3] Flutter architectural overview [Electronic resource] – Resource access mode: https://docs.flutter.dev/resources/architectural-overview.

[4] *Damian Białkowski, Felipe Diniz Dallilo.* FLUTTER UM FRAMEWORK PARA DESENVOLVIMENTO MOBILE // ResearchGate. – 2022. – P. 3-5.

[5] *Slavimir Stošović, Dušan Stefanović, Milan Bogdanović,* Nikola Vukotić. THE USE OF THE FLUTTER FRAMEWORK IN THE DEVELOPMENT PROCESS OF HYBRID MOBILE APPLICATIONS // ResearchGate. – 2022. – P. 5-6.

[6] *Anil Trimbakrao Gaikwad.* FIREBASE – OVERVIEW AND USAGE // ResearchGate. – 2022. – P. 2-4.

[7] *Jashandeep Singh, Swapnil Srivastva, Dipanshu Raj, Shubhampreet Singh, Mir Junaid Rasool.* FLUTTER AND FIREBASE MAKING CROSS-PLATFORM APPLICATION DEVELOPMENT HASSLE-FREE // IRJMETS. – 2022. – Vol. 4, No. 4, P. 3-5.

[8] *Damian Białkowski, Jakub Smołka.* Evaluation of Flutter framework time efficiency in context of user interface tasks // ResearchGate. – 2022. – P. 4-5.

[9] *Omar Almootassem, Syed Hamza Husain, Denesh Parthipan, Qusay H. Mahmoud.* A Cloud-based Service for Real-Time Performance Evaluation of NoSQL Databases // Arxiv. – 2017. – P. 3-4.

[10] *Azad, Avi Chaudhary, Jatin Chauhan, Basant Soam, Mr. Ashwini Kumar.* ANDROID APPLICATION USING FLUTTER AND FIREBASE WITH LBRS TO FIND PEOPLE OF THE SAME INTEREST AND COMMUNICATION PLATFORM // IRJMETS. – 2022. – Vol. 4, No. 5, P. 4773-4775.

[11] *Wisal Khan, Teerath Kumar, Zhang Cheng, Kislay Raj, Arunabha M Roy, Bin Luo.* SQL and NoSQL Databases Software architectures performance analysis and assessments – A Systematic Literature review // 3. – 2022. – P. 3.

[12] Structuring Cloud Firestore Security Rules [Electronic resource] – Resource access mode: https://firebase.google.com/docs/firestore/security/rules-structure.

[13] *Shayan Bagchi.* Firebase-A Cloud Hosted NoSQL Database // ResearchGate. – 2022. – P. 8-9.

[14] *Bhavin M. Mehta, Nishay Madhani, Radhika Patwardhan.* Firebase: A Platform for your Web and Mobile Applications // IJARSE. – 2017. – Vol. 6, No. 4, P. 46-47.

[15] *Julian Harty, Haonan Zhang, Lili Wei, Luca Pascarella, Mauricio Aniche, Weiyi Shang.* Logging Practices with Mobile Analytics: An Empirical Study on Firebase // Arxiv. – 2021. – P. 1-3.

The article was delivered to editorial stuff on the 27.05.2022