Push Notifications HandBook

A step-by-step implementation for iOS and Android

2nd Edition

Updated to Swift, Java, and Kotlin.

Yair Carreno

Push Notifications Handbook for iOS and Android

Yair Carreno

This book is for sale at http://leanpub.com/push-notifications-ios-android

This version was published on 2021-10-15

Contents

Preface

About this book

Dear reader. It is a pleasure to communicate with you and begin this adventure in the world of **Push Notifications**.

Perhaps many agree with me when considering push notifications as a fantastic mechanism to deliver messages to the user. I think it to be one of the most versatile capabilities that mobile applications have. But let's be honest, sometimes, during the implementation process of a feature that includes this ability, it can get more complicated than it should, either because the notification is not displayed, or that they suddenly stopped showing, or that they report a very strange bug and difficult to detect.

Thinking about it, I have good news for you, fellow readers. At the end of this book, you will agree with me that Push Notifications are an excellent mechanism to reach the user and that it is not so difficult after all to understand how they work and are implemented. In many cases, the answers are insight in the official documentation, but we get lost when putting all pieces together.

After many work sessions on this topic in mobile applications, I decided to share my knowledge. The two most important operating systems in mobile applications, iOS, and Android, are related. After all, *what mobile application nowadays does not use Push Notifications in any of its functionalities?*

Surely there will be a considerable percentage that still does not use these capabilities. Still, there is no doubt that it is a matter of time before they need to do so when they notice the multiple advantages they incorporate into an application.

Based on information collected from Firebase Cloud Messaging, Apple, Blogs, eBooks, and implementations in productive projects, I have organized each content so that the reader progressively acquires the fundamental concepts to develop later and implement each of these concepts.

If I can get the reader to include this manual in their reference documents, I feel that I have done my job on this topic. Also, I encourage the reader to constantly check for updates and improvements in this field that are constantly evolving like the other fields.

I hope, reader friend, that you like this work and not being more, "**Let's start!**".

About the source code

To develop the content of this book, I have created two public repositories with the source code for iOS and Android. The iOS project is implemented with *Swift*, and the Android project has *Java* implementation.

In the Push Notifications Implementation chapter, the implemented strategies are detailed, and the reader will be able to find the following repositories for the source code of the APPs:

- Push Notifications Handbook iOS[1].
- Push Notifications Handbook Android[2].
- Push Notifications Handbook Android-Kotlin[3].

Content development

The content has been divided into three main blocks of topics. The first block is about concepts, the second block is dedicated to configurations, and the third block is used to show the implementations in the APPs.

The **first block** of concepts contains the Generalities chapter, where basic information is presented to understand the mechanisms for sending and receiving push notifications fully. Concepts such as types of messages, state of the application at the time of receiving the message, the types of handling that can give to a notification, design patterns, and strategies, among others, are presented in this general chapter.

The Push Notifications Implementation chapter is also part of this block. This chapter presents the necessary steps to implement the design patterns that are presented in the overview chapter. Each of the strategies is described with its pros and cons so that the reader, in his excellent judgment, chooses one he considers most suitable for his solutions.

This block is the basis for developing the topics of the subsequent two blocks.

The **second block** is dedicated to configurations, that is, the activities that the reader must carry out concerning tools and consoles to successfully send messages.

The first chapter of this block is Project configuration in Firebase, where instructions are provided to create a project in Firebase for the cases of message distribution with FCM.

Later, iOS APP Configuration in Apple Developer Portal chapter is presented. It describes the information required and filled out to support push notifications to iOS clients.

And this block ends with Integrating APP to Firebase chapter, which is nothing more than integrating iOS and Android APPs to Firebase message distribution platforms.

The **third block** is about only implementation chapters. Based on the basic concepts and configurations completed, the source code is displayed in the mobile applications to receive and manage the messages from the FCM or APNs message distribution platforms. That is shown in the chapter Configuring the APP to receive Push Notifications.

[1] https://github.com/yaircarreno/Push-Notifications-Handbook-iOS
[2] https://github.com/yaircarreno/Push-Notifications-Handbook-Android
[3] https://github.com/yaircarreno/Push-Notifications-Handbook-Android-Kotlin

Moving to the *backend side*, Creating the Push Notifications Server chapter presents the implementation of microservices responsible for sending messages through FCM or APNs. These services are built with a *Serverless* design and through cloud services on *GCP*.

Finally, Testing Tools for Push Notifications chapter is presented, which is crucial for verifying the correct implementation of the push notifications system in APPs and detecting possible failures, wrong configurations, or possible *bugs* in handling messages.

In addition to these three blocks, I also wanted to share with the reader an extra chapter of recommendations, *tips*, and techniques that could be useful and complement the content. This chapter is called Bonus Track.

Glossary

During the development of the topics in the book, the reader may find some terms described below for clarity:

FCM: Firebase Cloud Messaging. Firebase service for message distribution to mobile and web clients..

APNS: Apple Push Notifications Service. Apple service for message distribution to iOS clients.

Local notifications: Notifications are generated from a *frontend* scope, that is, from the same client application.

Remote notifications: Notifications are generated from a *backend* scope, that is, from a remote server.

Audience

I recommend this book for any actor designing, implementing, and validating software components for mobile clients on iOS or Android operating systems.

Although Push Notifications are only one of the topics in the development of mobile solutions, it is vital to know in detail these mechanisms to use them in in-house implementations, optimizations, or improvements in user loyalty processes, among other uses and applications based on these types of messages.

Questions and contact

This work aims to guide the reader in searching for good design practices is solutions that involve *push notifications*.

It has been developed from my perspective and taking into account the recommendations of expert sources cited in the bibliography. The reader could find alternatives to the exposed techniques that may vary a bit from those listed here, which is precisely the present work's idea.

If the reader finds any aspect in the book that deserves to be reviewed, the comments and feedback given in this regard are welcome. For this and any questions or concerns, the following channels are available.:

- Email: yaircarreno@gmail.com
- Twitter: @yaircarreno[4]
- Blog: yaircarreno.com[5]

About image texts language

By standard, the texts and the description of the images presented throughout this book contain texts in English.

Idioma en Code Snippets

By standard, the code samples that accompany this book are in English. Also, some primary texts are in English to maintain consistency with the official documentation.

Versions of IDEs and technologies used

Android Environment

- **IDE**: Android Studio Arctic Fox 2020.3.1
- **Deployment SDK**: API 30 - Android 11.
- **Minimum SDK**: API 21 - Android 5.0 (Lollipop)
- **Languaje**: Java
- **Interface**: Activities

iOS Environment

- **IDE**: Xcode 12.5
- **iOS Deployment Target**: 14.5
- **Language**: Swift 5
- **Interface**: Storyboard

[4] https://twitter.com/yaircarreno
[5] https://www.yaircarreno.com/

Change log

This chapter is provided to keep the reader informed about updates and changes in this book. Changelog.

Future editions

With the advent of new technologies with declarative UIs in mind, an upcoming edition includes the projects with SwitfUI, Jetpack Compose, and Flutter. So, stay tuned, my readers.

Generalities

Introduction

One feature in mobile applications that I like the most is sending the user a message without needing the application to be running in the foreground. Either for alert purposes, for informational purposes, for advertising offers, marketing, or to draw the user's attention to a message that is to be communicated.

Push Notifications are messages in real-time; this adds a key immediacy value for certain functionalities in an application.

Remote Notifications Vs Local Notifications

Remote notifications are those messages sent from an external server, that is, in a context external to the application. *Local notifications* are those messages generated within the application and managed by the event system API of the device's operating system.

Before going into the detail of push notifications implementation, the following sections present those concepts that I consider to be key to understanding the complete operation from the moment notification is sent until it is received by the application for being presented to the user or managed by the application.

Having these concepts clear is essential for design and implementation phases, concepts such as the state of the application at the time of the notification's arrival, if the notification is *silent* or *alert*, if the server notifications send the message as *notifications*, as *data*, as both.

These considerations are relevant when implementing Push Notifications and avoiding possible errors that are difficult to detect.

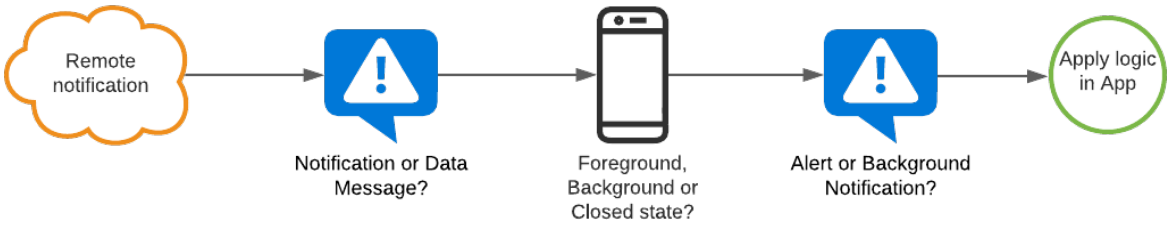

Figure 1.1 Flow receiving notification

Message types

The notification messages that the mobile application receives can come through two platforms, **FCM** (Firebase Cloud Messaging) or **APNS** (Apple Push Notification Service). One of the differences between one or the other case is the **JSON** message delivered by these platforms.

According to the selected strategy, the APP could receive messages from the notification server through FCM, APNS, or both. These patterns and strategies for sending notifications are discussed in the patterns section below.

It is essential to know the structure of the message sent by the notification server through **FCM** or **APNS** since depending on the type of message received, the application status will determine the behavior of the notification. We will demonstrate these conclusions later in the iOS notification test conclusions and Android notification test conclusions.

Let's analyze the message delivered by each of these platforms below:

Message from FCM

In **Firebase Cloud Messaging**, the message can contain only the Notification object, only the Data object, or it can contain both, as shown in the following code:

Example message FCM

```
{
  "message":{
    "token":"bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1...",
    "notification":{
      "title":"Argentina vs. Brasil",
      "body":"Great match!"
    },
    "data" : {
      "Nick" : "Mario",
      "Room" : "ArgentinaVSBrasil"
    }
  }
}
```

The *notification* and *data* objects are also often referred to as *notification payload* and *data payload*, respectively. The reader needs to keep the following in mind:

- The *notification* object is sent when you want FCM to handle the application's notification automatically.
- The *data* object is sent when you want the application to manage and take control of the notification.

These rules are summarized by FCM as follows:

	Use scenario	How to send
Notification message	FCM automatically displays the message to end-user devices on behalf of the client app. Notification messages have a predefined set of user-visible keys and an optional data payload of custom key-value pairs.	1. In a trusted environment such as Cloud Functions or your app server, use the Admin SDK or the FCM Server Protocols: Set the **`notification`** key. May have optional data payload. Always collapsible. See some examples of display notifications and send request payloads. 2. Use the Notifications composer: Enter the Message Text, Title, etc., and send. Add optional data payload by providing Custom data.
Data message	Client app is responsible for processing data messages. Data messages have only custom key-value pairs with no reserved key names (see below).	In a trusted environment such as Cloud Functions or your app server, use the Admin SDK or the FCM Server Protocols: Set the **`data`** key only.

Table 1.1 General rules in FCM: Image taken from Firebase Documentation

Message sent by FCM contains fields that are common and can be interpreted equally on iOS, Android, or Web. These fields are precisely those of the *notification* and *data* objects.

- message.notification.title
- message.notification.body
- message.data

For example, that means would interpret the following message sent from a server with Node in the same way on iOS, Android, or Web:

Example message in Node sending to FCM

```
const message = {
  notification: {
    title: 'Hi notification',
    body: 'Testing my push notification.'
  }
};
```

It is also possible to send information in a message that is intended only for a specific platform. For example, the following message sends information for the Android, iOS, and Web platforms through the **android**, **apns**, **webpush** objects:

Example message in Node sending to FCM with specific information

```
const message = {
  notification: {
    title: 'Sparky says hello!'
  },
  android: {
    notification: {
      imageUrl: 'https://foo.bar.pizza-monster.png'
    }
  },
  apns: {
    payload: {
      aps: {
        'mutable-content': 1
      }
    },
    fcm_options: {
      image: 'https://foo.bar.pizza-monster.png'
    }
  },
  webpush: {
    headers: {
      image: 'https://foo.bar.pizza-monster.png'
    }
  },
  topic: topicName,
};
```

In the previous example, the *title* field is the same for all platforms, but the presentation of the image is particular for each platform. The result of the configuration could be similar to the one shown in the following image:

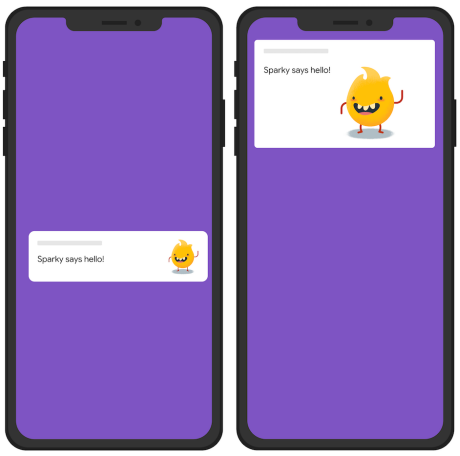

Figure 1.2 Example setting image notification. Image taken from Firebase documentation

Find more examples of these configurations in the following Firebase Cloud Messaging documentation: *Build send requests*[^1].

In addition to these two main objects, *notification* and *data*, the message could also include more information about distributing it. That is done through the exclusive *token*, *topic*, or *condition* fields.

In the following example, the configuration of a *notification* type message distributed to **multiple** devices through its tokens is shown:

Example message in Node sending to FCM by tokens

```
const message = {
  notification: {
    title: 'Sparky says hello!'
  },
  tokens: ['ABCDFE...', 'OPQRSTUV...'],
};
```

The following example shows the configuration of a *notification* type message distributed to a **specific** device through its token:

Example message in Node sending to FCM by only one token

```
const message = {
  notification: {
    title: 'Sparky says hello!'
  },
  token: 'ABCDFE...',
};
```

The following example shows the configuration of a *notification* type message distributed to subscribers through a **topic**.

Example message in Node sending to FCM by topic

```
const message = {
  notification: {
    title: 'Sparky says hello!'
  },
  topic: 'all-team',
};
```

The notification server uses the **Firebase Admin SDK** and the **HTTP protocol v1**[^2] to implement and deliver push notifications. This server can be designed with microservices written in *Node*, *Java*, *Python*, *C#*, or *Go*. It will provide more details with example codes about these implementations in Creating the Push Notifications Server.

Interpreting the FCM message on Android

I mentioned earlier that it is essential to know the type of message sent from the notification server through FCM. That is because depending on the types of objects sent in the message, that is, *notification*, *data*, or both, it will be determined what kind of behavior the remote notification will have, also taking into account the state of the application. Application's state are the topic covered in the next section.

In **Android**, the following conditions are:

Use case 1: If the remote notification requires to be presented only when the application is in a *foreground state*, it is recommended to use a message like the following:

Example message FCM for Foreground State Android

```
{
  "message":{
    "token":"bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1...",
    "notification":{
      "title":"Argentina vs. Brasil",
      "body":"Great match!"
    }
}
```

In this case, only the *notification* object is sent, and the notification will be managed automatically by FCM showing the *title* and the *body* with a standard notification UI component.

However, this case is rare in applications, as generally, the goal is to present the notification regardless of whether the application is in *foreground*, *background*, or *closed*. That brings us to the second case shown below.

Use case 2: If the remote notification requires to be managed and to process information sent in the message when the application is in the *foreground*, *background* or *closed* states, it is recommended to use a message like:

Example message FCM for Foreground, Background or Closed State Android

```
{
  "message":{
    "token":"bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1...",
    "data" : {
      "title":"Argentina vs. Brasil",
      "body":"Great match!"
    }
  }
}
```

In this case, only the *data* object is included. The *notification* object cannot be included since otherwise would only manage the message for a state in foreground.

In summary, and taking into account the FCM documentation, the following are the right conditions for **Android**:

App state	Notification	Data	Both
Foreground	• Doesn't show a notification. • **onMessageReceived**: called.	• Doesn't show a notification. • **onMessageReceived**: called.	• Doesn't show a notification. • **onMessageReceived**: called.
Background	• Show notification. • **onCreate**: called if the user clicks on the message.	• Doesn't show a notification. • **onMessageReceived**: called.	• Show notification. • **onCreate**: called if the user clicks on the message.
Closed	• Show notification. • **onCreate**: called if the user clicks on the message.	• Doesn't show a notification. • **onMessageReceived**: called.	• Show notification. • **onCreate**: called if the user clicks on the message.

Table 1.2 FCM rules for Android

Will verify these conclusions and cases through a set of tests on *push notifications* in the implementation section.

When does onMessageReceived run?

From the previous table 1.2, we can deduce that in Android, the operation *onMessageReceived* will be executed when the application is in *Background* if the message received is only *Data* type. That is important to keep in mind for Android deployments. We will say more on the subject in later sections.

Interpreting the FCM message on iOS

In the iOS case, all messages delivered by FCM are made through APNS. That is one reason why, despite using FCM, you must also configure the sending of the notification in APNS.

Is it possible to use only FCM to distribute messages to iOS clients without using APNS?

No. Distributing messages to iOS clients will always require APNS integration.

In the previous section of Android, emphasis was placed on the concept of message types (*notification* or *data*), and how depending on these types of messages, the behavior of the notification could be influenced when it is received in the APP.

On the other hand, in iOS, the concept of notification type is more relevant, that is, *Alert push notifications* and *Background push notifications*. These concepts will be the subject of an upcoming section called Types of notifications.

However, in iOS, these concepts are interpreted similarly. A *notification* or *notification + data* type message will be interpreted as an *Alert push notification*, and a data type message will be interpreted as a *Background push notifications*. We will demonstrate that later.

For now, I will show examples of these two types of messages in FCM:

Example message in Node sending to FCM for Alert notification

```
const registrationToken = 'bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1...';

const message = {
  notification: {
    title: 'Urgent action needed!',
    body: 'Urgent action is needed to prevent your account from being disabled!'
  },
  data: {
      key1: "some_value",
      key2: "another_value",
      key3: "one_more"
  },
  token: registrationToken
};

admin.messaging().send(message)
  .then((response) => {
    console.log('Successfully sent message:', response);
  })
  .catch((error) => {
    console.log('Error sending message:', error);
  });
```

Example message in Node sending to FCM for Background notification

```
const registrationToken = 'bk3RNwTe3H0:CI2k_HHwgIpoDKCIZvvDMExUdFQ3P1...';

const message = {
  notification: {
    title: 'Urgent action needed!',
    body: 'Urgent action is needed to prevent your account from being disabled!'
  },
  data: {
      key1: "some_value",
      key2: "another_value",
      key3: "one_more"
  }
};

const options = {
```

```
  priority: 'high',
  contentAvailable: true
};

admin.messaging().sendToDevice(registrationToken, message, options)
  .then((response) => {
    console.log('Successfully sent message:', response);
  })
  .catch((error) => {
    console.log('Error sending message:', error);
  });
```

Limitations of FCM messages

- For *notification* type messages, the key-value values predefined in the standard must be used. Changing these names could generate errors.
- For *data* type messages, the key-value values can be customized and must be considered when processing the client application notification.
- The *notification* type message can optionally contain a *data* payload. Do not confuse this payload with the other type of message that we have called *data*.
- Maximum size of 4KB per message is allowed, whether only the *notification*, *data*, or both are sent.
- For testing and sending messages through the Firebase console, up to 1024 characters are allowed.

Message from APNS

If the platform chosen for the distribution of messages to iOS users has been APNS, then the APP must be prepared to receive messages with the following types of structures[^3]:

This one is an Alert Push Notifications type message:

Example message APNS for Alert notification

```
{
   "aps" : {
      "alert" : {
         "title" : "Game Request",
         "subtitle" : "Five Card Draw",
         "body" : "Bob wants to play poker"
      },
      "category" : "GAME_INVITATION"
   },
   "gameID" : "12345678"
}
```

This one is a Background Push Notifications type message, also known as silent messages:

Example message APNS for Background notification

```
{
   "aps" : {
      "content-available" : 1
   },
   "acme1" : "bar",
   "acme2" : 42
}
```

This one is a *Sound notification* type message:

Example message APNS for playing sound notification

```
{
   "aps" : {
      "badge" : 9,
      "sound" : "bingbong.aiff"
   },
   "messageID" : "ABCDEFGHIJ"
}
```

This one is a *With Localized Content* type message:

Example message APNS for notification with localized content

```
{
   "aps" : {
      "alert" : {
         "loc-key" : "GAME_PLAY_REQUEST_FORMAT",
         "loc-args" : [ "Shelly", "Rick"]
      }
   }
}
```

Application's state

As mentioned in previous sections, within the strategies used by the APP for the administration of *push notifications*, it is essential to know the state in which the application is when the message is received. These states are described below.

Foreground State

An application is said to be in a *Foreground* state when at least one of its screens or services is visible to the user; that is, it runs in the foreground.

Background State

An application is in the Background state when none of its views are visible to the user; that is when the application runs all its services in the background.

This state has certain restrictions to operate. For example, we have limited memory access or depend on the device's battery level.

Also, due to privacy and transparency policies, some operations are restricted in the background state, such as accessing the location.

It is recommended to avoid using these states to execute prolonged tasks; instead, other alternatives can be used, such as processes with jobs that work in the background and do not interfere with the interaction with the user.

Closed State

If the application is not running in the foreground or running in the background, it is said to be a closed application with all its processes turned off.

Determine the status of the application

One of the functions performed by message distribution platforms (FCM or APNS) is to determine the APP's state when it delivers the message and accordingly decides the behavior of the notification delivered.

However, the reader could add additional validations on the application's state when the message is received—for example, this to enforce a particular flow or a specific use case.

The programmatic way to check the status of the application on both iOS and Android is described below.

In iOS

To determine the *foreground* or *background* states of the App in iOS, one can use the following function in Swift:

Checking App status in iOS

```
1  func isForeground() -> Bool {
2      return UIApplication.shared.applicationState == .active
3  }
```

The delegated class for the implementation of this check must have a dependency:

Dependency included

```
import UIKit
```

In Android

For Android, we will use Google's recommendation[^4] to use *ProcessLifecycleOwner*[^5] class to determine the *foreground* and *background* states of the APP like this:

Checking App status in Android

```
private boolean inForeground() {
    return ProcessLifecycleOwner.get().getLifecycle().getCurrentState()
            .isAtLeast(Lifecycle.State.STARTED);
}
```

The following dependencies[^6] need to be installed in the application:

Dependencies included

```
def lifecycle_version = "2.3.1"

// Lifecycles only (without ViewModel or LiveData)
implementation "androidx.lifecycle:lifecycle-runtime:$lifecycle_version"

// alternately - if using Java8, use the following instead of lifecycle-compiler
implementation "androidx.lifecycle:lifecycle-common-java8:$lifecycle_version"

// optional - ProcessLifecycleOwner provides a lifecycle for the whole application process
implementation "androidx.lifecycle:lifecycle-process:$lifecycle_version"
```

What about the closed/kill state?

At the application level, it will make sense to determine if the state is *foreground* or *background*. For that cases in which the application is completely closed, the handling of the messages will be managed by the operating system as follows.

In the Android case, see the previous *Table 1.2 FCM rules for Android*. And for the iOS case see the following table:

App state	Alert push	Background push
Foreground	• Show notification. • **willPresent**: called. • **didReceive**: called if the user clicks on the message	• Doesn't show a notification • **didReceiveRemoteNotification**: called.
Background	• Show notification. • **didReceive**: called if the user clicks on the message	• Doesn't show a notification • **didReceiveRemoteNotification**: called.
Closed	• Show notification. • **didFinishLaunchingWithOptions**, **didReceive**: called if the user clicks on the message	• Doesn't show a notification. • **didFinishLaunchingWithOptions**, **didReceiveRemoteNotification**: called if the user clicks on the message

Table 1.3 APNs rules for iOS

In the Implementing in iOS chapter, a set of verifications that conclude will show the data presented in the previous table.

Types of notifications

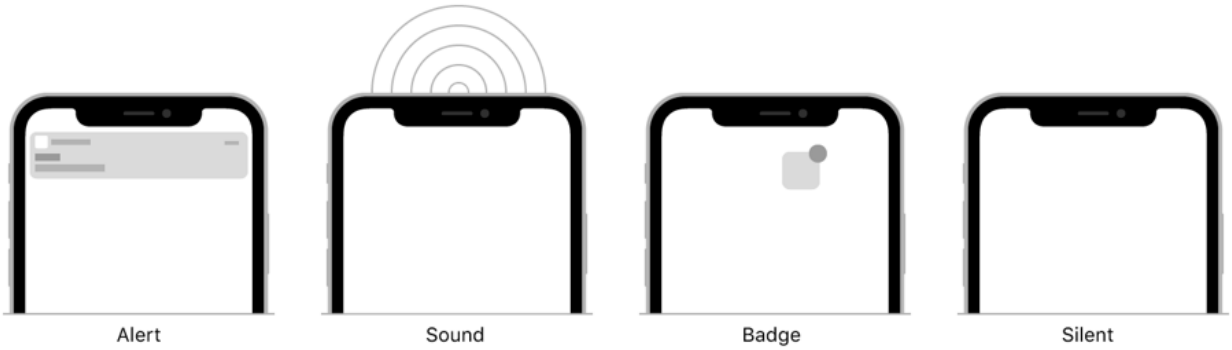

Figure 1.3 Notifications type: Image taken from Apple Documentation

Alert Push Notifications

Alert Push Notifications are messages visible to both the user and the application. They allow the user to interact with the notification and perform specific actions defined by the user experience. These types of notifications can have a standard or custom UI layout.

These notifications can be executed when the application is in Foreground, Background, or Closed status.

Background Push Notifications

They are also known as *silent notifications*. They are messages visible only to the application. The user is not notified about these messages for which they are silent notifications that are only known by the application in the background.

These types of notifications can be used to activate the execution of background tasks that do not require the user to be informed, such as updating a component, querying the cache for optimization, among other use cases.

Like Alert Push Notifications, they are notifications that can be executed when the application is in *Foreground*, *Background*, or *Closed* status. However, a series of restrictions could interrupt or prevent its normal operation, such as a low battery level or a configuration by the user that prevents background processes. These limitations must be taken into account when designing functionalities with this type of notification.

Although the concept of *alert push notifications* and *background push notifications* is documented for iOS, it is a concept that can also be found on Android and applied similarly. The chapter Configuring the APP to receive Push Notifications will be presented the details of implementing this type of notification on both iOS and Android.

Patterns for Push Notifications implementation

Can use multiple strategies and patterns to implement a *push notification* system. This chapter presents the most common patterns, which may vary depending on the needs of the solution.

After analyzing these patterns, the implementation detail of said patterns exposed in this chapter is presented:

- Https triggers with FCM and APNS approach.
- Https triggers with APNS approach (only iOS).
- Https triggers with FCM approach (only Android).

The implementation of these strategies can be taken as a reference for implementing the other exposed uses.

HTTPS Triggers with FCM and APNS Approach

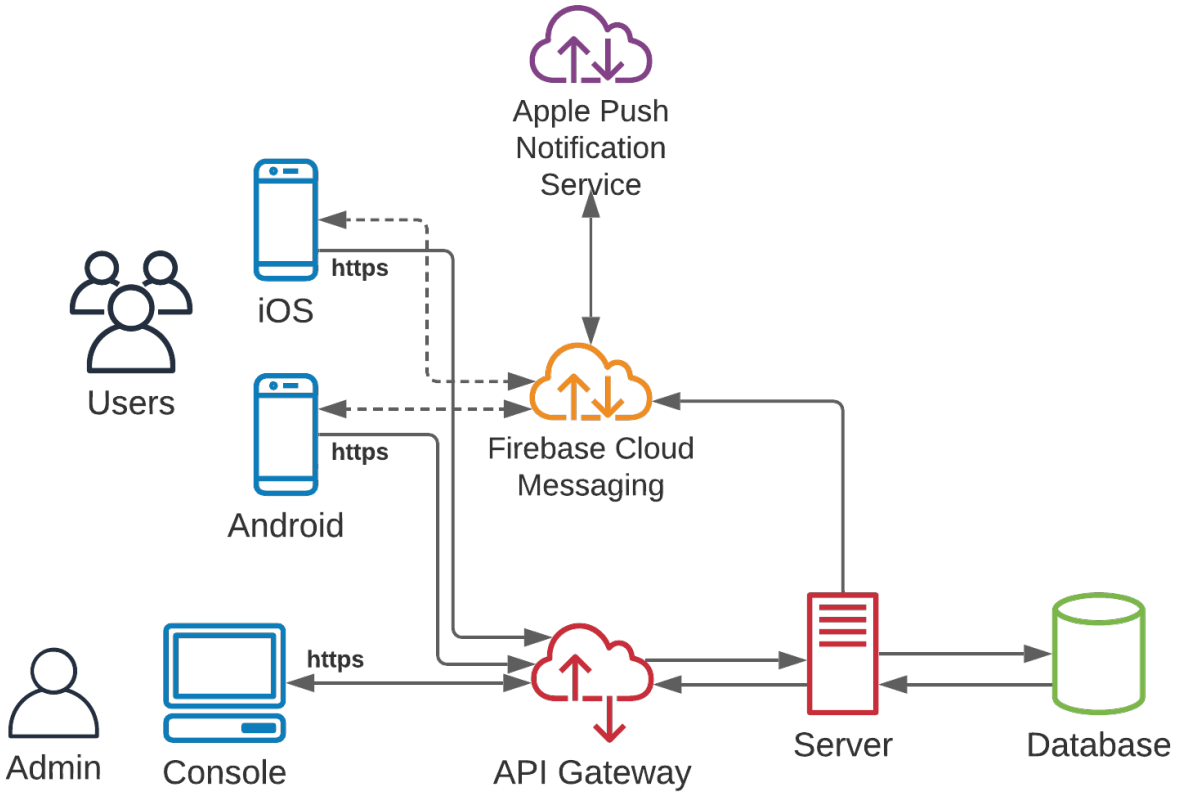

Figure 1.4 HTTPS Triggers with FCM and APNS Approach

This pattern uses the **FCM** (Firebase Cloud Messaging) platform to distribute messages to iOS, Android, and Web clients.

Additionally, FCM communicates with **APNS** (Apple Push Notification Service) to guarantee the delivery of notifications to iOS clients in a secure manner and authorized by Apple.

Server notifications provide through *Microservices* and *API Gateway* the integration for clients to register their devices to receive notifications, update tokens of one or more devices, generate requests to send notifications. Also, configure notifications and provide other functionalities that the solution requires at the Push Notifications level through, for example, a console.

The Server will also access the **Database** persistence mechanism that will store the tokens of the registered devices to be notified with the messages.

A *database* could also store particular settings for notification and required by *Server*.

HTTPS Triggers with Independent FCM and APNS Approach

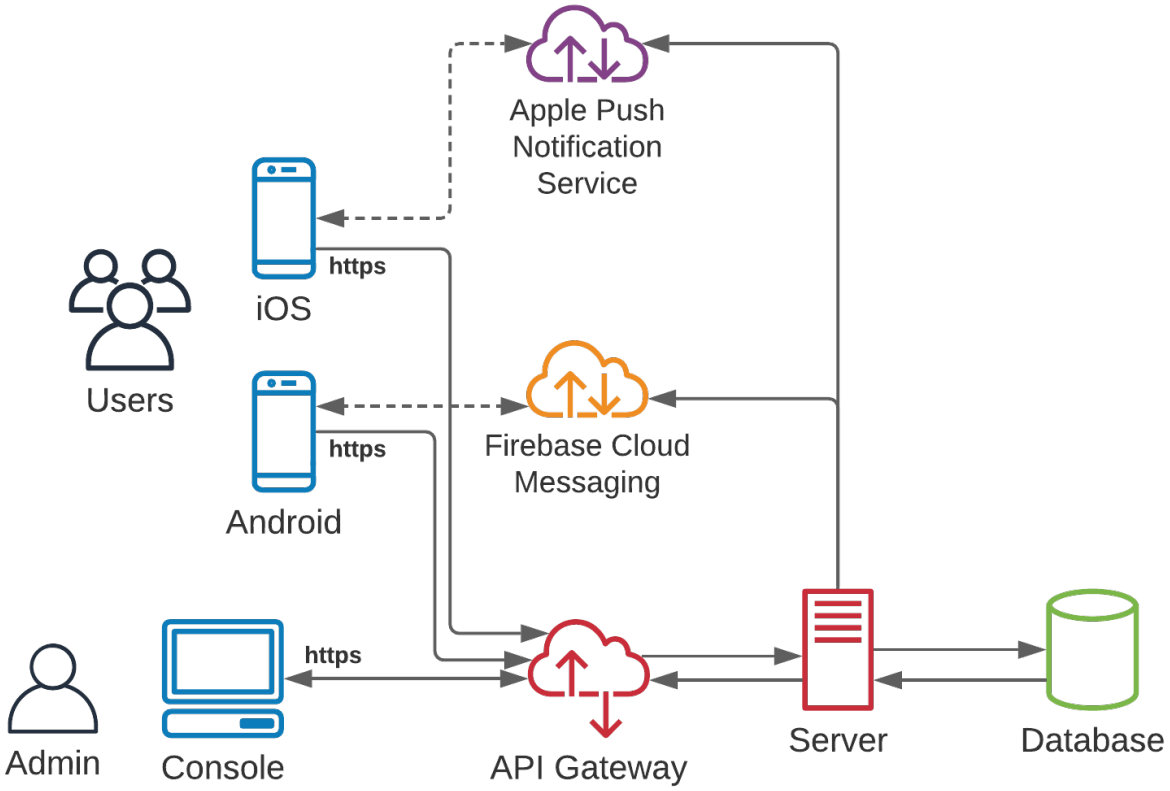

Figure 1.5 HTTPS Triggers with Independent FCM and APNS Approach

This pattern uses the **FCM** platform to distribute messages to Android clients and uses the **APNS** platform to broadcast messages to iOS clients. Push Notifications distribution is designed using both platforms independently, unlike the previously exposed pattern.

In this pattern, the *API Gateway*, *Server*, and *Database* components fulfill the same functions described in the previous design, with the difference that in this strategy, *Database* will store both the tokens generated by FCM and the tokens generated by APNS.

HTTPS Triggers with APNS Approach

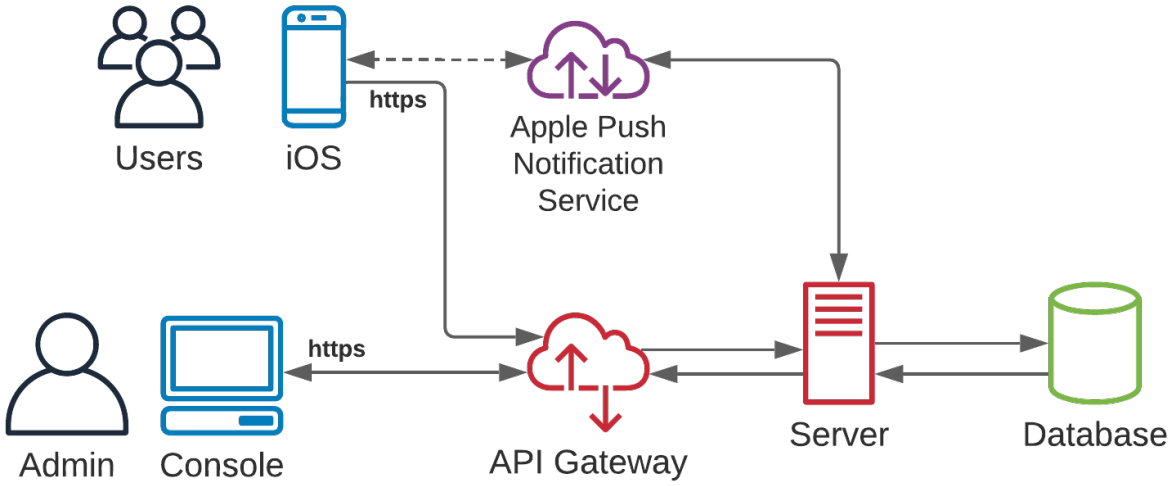

Figure 1.6 HTTPS Triggers with APNS Approach

This pattern is appropriate when it is required to provide notifications only to iOS users. In this case, only **APNS** is used as a platform to manage and distribute the messages.

To keep in mind

Remember that also through *FCM* can provide notifications to iOS clients. However, if you do not have short- and long-term plans to serve Android users, more successful use is to use *APNS*.

HTTPS Triggers with FCM Approach

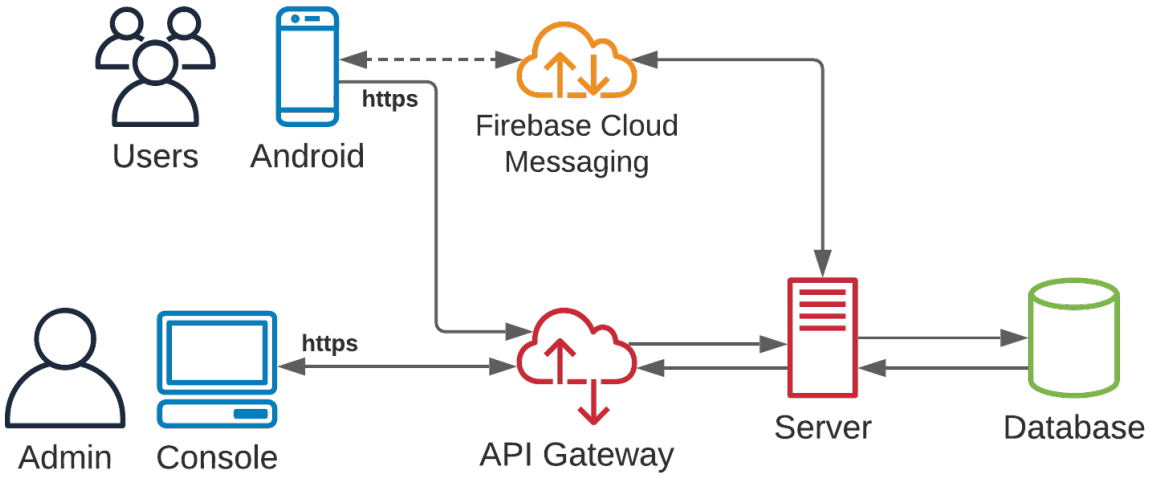

Figure 1.7 HTTPS Triggers with FCM Approach

This pattern is appropriate when it is required to provide notifications only to Android users. In this case, only **FCM** is used as a platform to manage and distribute the messages.

Event-Driven Approach

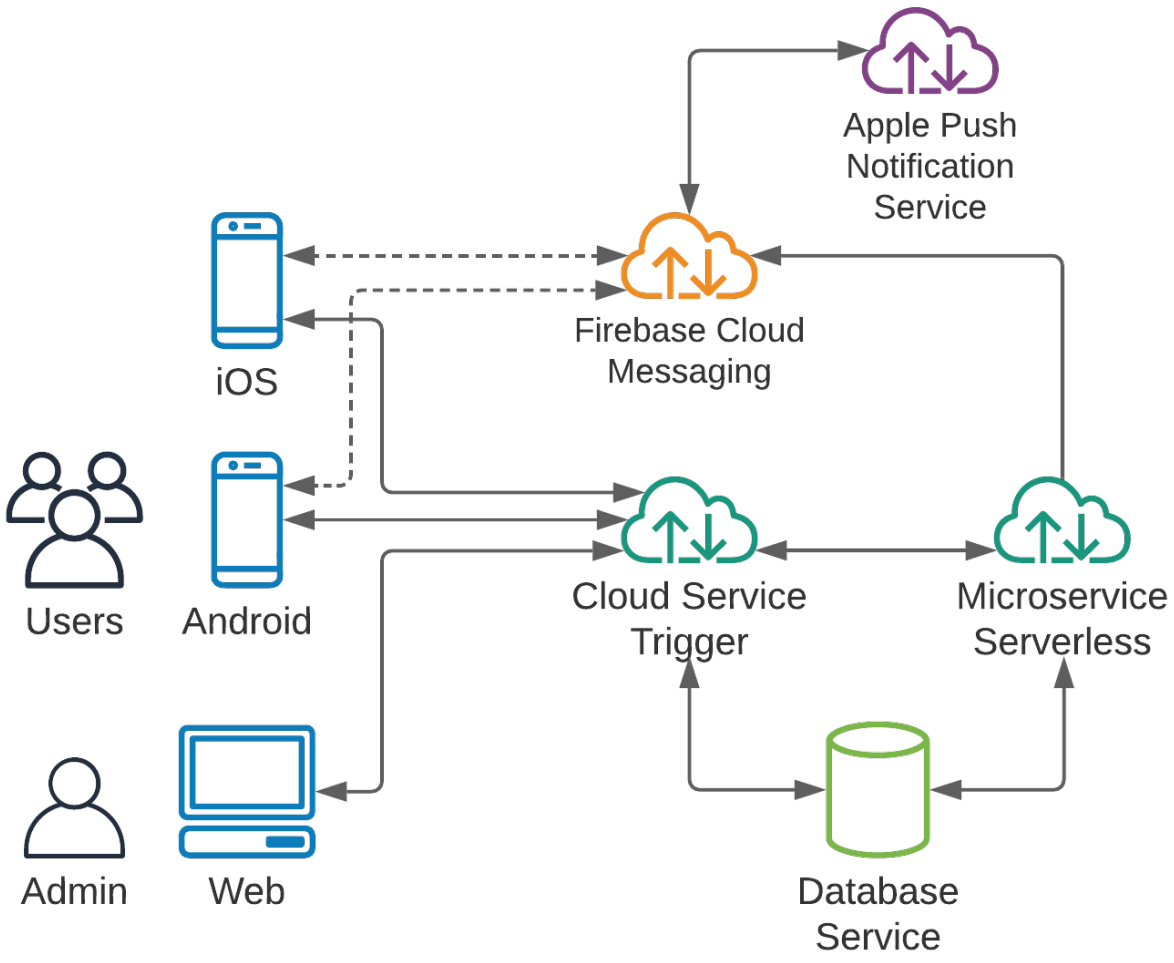

Figure 1.8 Event-Driven Approach

This pattern is a design with a *Serverless approach*. The execution actions come from *HTTP triggers* and triggers originating from databases, serverless functions (*Cloud Functions* or *Lambdas*), or programmed and automated batch processes.

In this pattern, **Database Service** is used to store the tokens and the required notification system settings. These services could be databases such as *Firestore*, *DynamoDB*, *Realtime Database*, *RDS*, and *Cloud SQL*.

As **Microservice Serverless**, you could use services available with *FaaS* such as *Cloud Functions* or *Lambdas* or serverless containers such as *Cloud Run* or *Fargate*.