Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Migración de aplicaciones Android hacia Flutter, un framework para desarrollo de apps multiplataforma

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Cristian Zazo Millán

Tutor: Patricio Letelier Torres

2018-2019

Dedicado a mi familia, mi pareja y mis amigos por apoyarme incondicionalmente durante mi formación y mi crecimiento personal.

Resumen

El desarrollo de aplicaciones móviles mantiene un importante crecimiento año tras año. Dos sistemas operativos son los más usados en el ecosistema móvil: Android y iOS. Para empresas que desarrollan software y que son relativamente pequeñas es difícil abordar el desarrollo de aplicaciones móviles en nativo, con desarrolladores especializados en ambas plataformas o con equipos distintos desarrollando en paralelo. Por esto se han popularizado entornos de desarrollo que permiten realizar un desarrollo y mantenimiento en un código común, pero generar código nativo hacia distintas plataformas. Sin embargo, cuando una empresa decide apostar por un entorno de desarrollo multiplataforma, a menos que se decida volver a desarrollar desde cero una aplicación existente, le resultará interesante poder migrar al menos parcialmente la aplicación existente a dicho nuevo entorno de desarrollo.

Android es la plataforma móvil que cuenta con mayor cantidad de aplicaciones desarrolladas de forma nativa. Por otra parte, Flutter es un framework para desarrollo multiplataforma recientemente lanzado por Google (y con el soporte que esto supone). Así, en este TFG, se establecerán pautas para la migración de una aplicación nativa desarrollada en Android hacia Flutter. Las pautas se aplicarán a un caso de estudio real, que ilustrará la migración de una aplicación de una *startup* en la cual el autor de este TFG ha realizado sus prácticas.

Palabras clave: Android, Flutter, migración de aplicaciones, desarrollo multiplataforma.

Abstract

Mobile app development keeps an important growth year over year. Two operating systems are the most used in the mobile ecosystem: Android and iOS. For companies that develop software and are relatively small, it is difficult to approach the development of native mobile applications, with specialized developers on both platforms or with different teams developing in parallel. This is the why development environments that allow development and maintenance of a common code but generate native code to different platforms have been popularized. However, when a company decides to opt for a multiplatform development environment, unless it is decided to redevelop an existing application from scratch, it will be interesting to be able to, at least, partially migrate the existing application to this new development environment.

Android is the mobile platform that hast the most applications developed natively. On the other hand, Flutter is a framework for multiplatform development recently launched by Google (and with the support that this implies). Thus, in this work, guidelines will be established for the migration of a native application developed in Android to Flutter. The guidelines will be applied to a real case study, which will illustrate the migration of an application of a startup in which the author of this work has carried out his practices.

Keywords: Android, Flutter, app migration, cross-platform development.

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

1.1. Motivación

A lo largo de los últimos 10 años, la cantidad de dispositivos móviles inteligentes ha crecido significativamente, al igual que los ingresos relacionados con las aplicaciones móviles. Además, la tendencia creciente continuará durante los próximos años. **[1]** De esta forma, desarrollar una aplicación para móviles es algo indispensable para muchas empresas dado que es donde pueden llegar a una mayor cantidad de usuarios para que utilicen sus servicios.

Actualmente, desarrollar una aplicación móvil para los dos sistemas operativos móviles mayoritarios, iOS y Android, puede resultar muy costoso especialmente para equipos de desarrollo de pequeñas o medianas empresas. Tener presencia en la tienda de aplicaciones de ambos sistemas operativos es muy importante para así poder tener un mayor alcance, pero optar por un desarrollo nativo para cada una de las plataformas puede ser muy costoso y de un riesgo bastante elevado, especialmente para empresas pequeñas como *startups*.

A nivel mundial, en 2017 se publicaron más aplicaciones nuevas en la tienda de aplicaciones de Android que en la de iOS. Sin embargo, los desarrolladores suelen optar por desarrollar inicialmente la aplicación para iOS. La decisión sobre para qué plataforma desarrollar primero depende de muchos factores, como la distribución de usuarios en Android y iOS en una determinada región, el público al que se quiera llegar inicialmente y los recursos disponibles de la empresa. En caso de optar por desarrollar inicialmente la aplicación en Android por tener una mayor cantidad de usuarios posibles y un menor coste económico de desarrollo (no se necesita *hardware* específico y la licencia de desarrollador es más barata), el siguiente paso es realizar una migración de la aplicación de Android a iOS para así poder llegar a muchos más usuarios. En 2017, dado que inicialmente las aplicaciones se suelen publicar en iOS, se migraron más de 17 mil aplicaciones de iOS a Android, mientras que unas 7.500 lo hicieron de Android a iOS. Además, de todas las aplicaciones disponibles en ambas plataformas, solamente 450 mil están disponible en ambas, de un total de 1,6 millones en iOS y 3,2 millones en Android. Se puede observar por tanto que la cantidad de aplicaciones presentes en Android y no en iOS es muy elevada, acercándose a los 3 millones. De todas estas aplicaciones, en 2017 se produjo un decrecimiento de las aplicaciones desarrolladas de forma no nativa tanto en iOS como en Android, aunque tras la reciente aparición de Flutter[1] y todas las ventajas que aporta a este tipo de desarrollo, es de esperar que el crecimiento de aplicaciones desarrolladas de forma no nativa crezca en los próximos años. **[2]**

[1] https://flutter.dev/

En la comunidad de desarrolladores, el interés por Flutter ha crecido muy rápidamente, hecho que se puede contrastar observando la cantidad de preguntas relacionadas con el *framework* en el sitio web Stack Overflow[2]. La cantidad de preguntas relacionadas con Flutter actualmente es similar o superior a la del resto de *frameworks* de desarrollo de aplicaciones móviles multiplataforma, como se puede observar en la Figura 1.

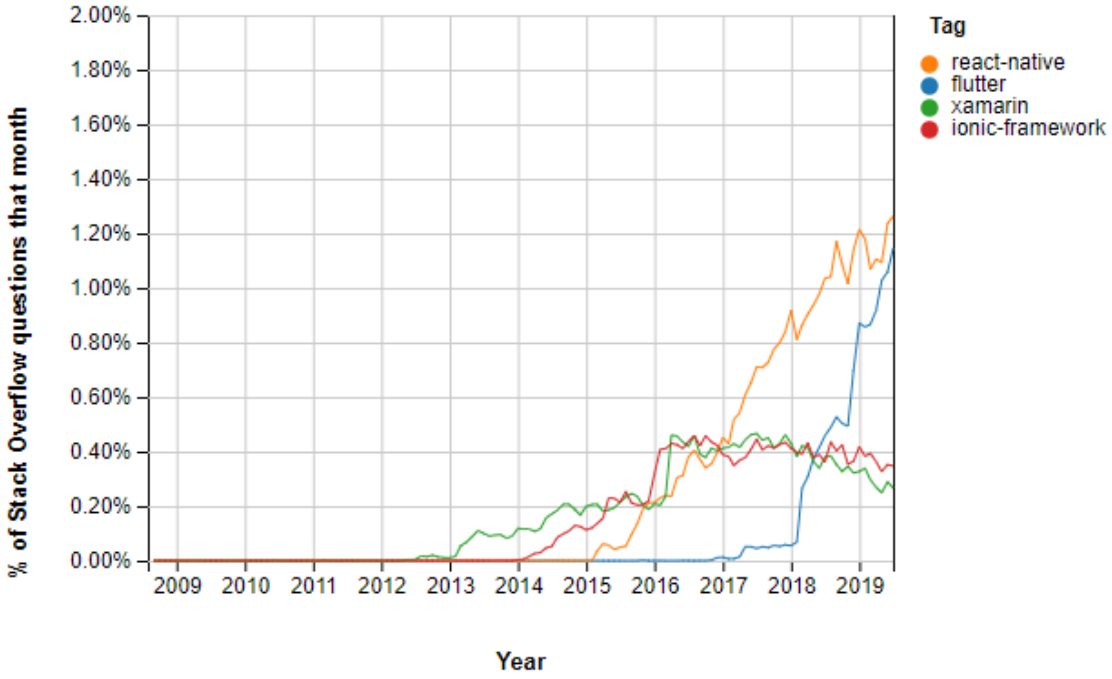

Figura 1: Cantidad de preguntas de cada framework en Stack Overflow

En este trabajo se establece un procedimiento a seguir a las empresas que, teniendo una aplicación en Android, quieran tener una en iOS sin aumentar los recursos necesarios, dado que el mismo equipo de Android puede encargarse de realizar la migración y posteriormente el mantenimiento de la aplicación realizada en Flutter. Además, de esta forma, todos los desarrolladores que porten sus aplicaciones de Android a iOS utilizando las herramientas expuestas en este trabajo, podrán tener sus aplicaciones disponibles en el futuro también para otras plataformas como la web, escritorio (Windows, macOS, Linux), o sistemas operativos en desarrollo como Fuchsia[3], de Google.

[2] https://es.stackoverflow.com/
[3] https://fuchsia.dev/

1.2. Objetivo

El objetivo principal de este proyecto es definir una serie de patrones a seguir a la hora de realizar una migración de una aplicación nativa desarrollada en Android a Flutter, un *framework* de desarrollo de aplicaciones multiplataforma.

El impacto esperado tras finalizar la migración es, en el caso de tener solamente una aplicación para Android, tener la aplicación disponible para ambas plataformas, y en el caso de tener la aplicación disponible para iOS y Android, fusionar ambos equipos de desarrollo en uno especializado en Flutter para así poder aumentar la velocidad en la que se realiza la entrega continua (*Continuous Delivery*).

1.3. Estructura

A continuación, se presenta la estructura de la memoria:

El capítulo 1 presenta la introducción a la memoria, en la que se encuentra la motivación del trabajo, objetivo a cumplir, impacto esperado, metodología, estructura y convenciones.

En el capítulo 2 se exponen las ventajas y desventajas del desarrollo móvil multiplataforma, se introducen algunos *frameworks* de desarrollo multiplataforma (Xamarin, React Native y Flutter), y se realiza una comparativa de Flutter con el resto.

En el capítulo 3 se presenta brevemente el estado del arte relacionado con la migración de una migración nativa en Android a un *framework* multiplataforma y se realiza una crítica compuesta por los beneficios que puede aportar Flutter con respecto al estado del arte actual.

En el capítulo 4 se introducen los puntos clave de Flutter: su proceso de instalación en Windows, su lenguaje de programación (Dart), sus componentes visuales (*widgets*), la navegación entre las distintas vistas de una aplicación, la animación de elementos y el uso de *platform channels* para suplir las posibles carencias actuales en el *framework* o implementar código específico de cada plataforma.

En el capítulo 5 se presenta la arquitectura base de un nuevo proyecto en Flutter y se propone una arquitectura escalable a utilizar en el proyecto Flutter resultante de la migración basada en la experiencia del caso de estudio.

En el capítulo 6 se presentan una serie de patrones acerca de cómo migrar distintos componentes de la interfaz de usuario de una aplicación nativa en Android a sus *widgets* equivalentes en Flutter.

En el capítulo 7 se presentan una serie de patrones acerca de cómo migrar algunos fragmentos comunes de código presentes en una aplicación Android que usa Java a Flutter, donde se utiliza el lenguaje Dart.

En el capítulo 8 se introduce el caso de estudio en el que, siguiendo las pautas y patrones presentados en los capítulos anteriores, se consigue migrar con éxito la aplicación existente en Android a Flutter, obteniendo así las aplicaciones de Android y iOS desde un mismo código fuente.

El capítulo 9 presenta las conclusiones del trabajo realizado respecto al objetivo planteado inicialmente.

En el capítulo 10 se comenta el trabajo futuro surgido a partir del desarrollo de este documento: mantenimiento de patrones, añadir nuevos patrones de migración tanto de interfaz como de lógica y el desarrollo de una herramienta que, aplicando los patrones expuestos, permita la automatización parcial de la tarea de migración.

1.4. Convenciones

- El código fuente (nombres de métodos, variables, clases, etc.) se muestra en letra `Fira Code`. Y sólo se empleará esta tipología para este tipo de contenido.
- Las palabras extranjeras se remarcarán en cursiva.

2. Desarrollo móvil multiplataforma

El desarrollo móvil multiplataforma es una tendencia que crece año tras año y que permite, a partir de un código común, generar aplicaciones para los sistemas operativos móviles más usados actualmente: Android y iOS.

Esto permite a una empresa, especialmente a las que son relativamente pequeñas, desarrollar su aplicación móvil y publicarla en la tienda de aplicaciones de ambos sistemas operativos sin la necesidad de dos equipos distintos, cada uno especializado en una de las plataformas, reduciendo así los costes. Además, se reduce también el tiempo de desarrollo necesario y se reduce la cantidad de código duplicado, dado que las funciones sólo se programan una vez y funcionan en ambas plataformas.

No obstante, optar por un desarrollo móvil multiplataforma también tiene dos desventajas principales: el rendimiento, el cual en la mayoría de los casos se ve afectado por el uso de componentes no nativos, dado que la comunicación de éstos con los distintos componentes nativos de cada plataforma suele ser inconsistente; y la experiencia de usuario, dado que los componentes de la interfaz no pueden aprovechar el potencial que podrían al no ser componentes nativos de la propia plataforma.

Actualmente, algunos de los *frameworks* de desarrollo móvil multiplataforma más utilizados son React Native[4] y Xamarin[5]. A este conjunto se ha unido recientemente Flutter[6], que soluciona los principales problemas del resto de *frameworks* de desarrollo móvil multiplataforma y el cual se analizará más adelante.

2.1. Xamarin

Xamarin **[3]** es uno de los *frameworks* de desarrollo móvil multiplataforma más populares. Se presentó oficialmente en 2011, y desde 2016 es propiedad de Microsoft.

Utiliza C# como lenguaje de programación para su desarrollo y posee dos formas distintas de compilar a código nativo: en iOS, por limitaciones del sistema operativo, utiliza AOT[7] (*ahead-of-time*), mientras que en Android utiliza JIT[8] (*just-in-time*).

Xamarin también permite realizar llamadas a código específico desarrollado para una plataforma, permitiendo así ejecutar bloques de código previamente desarrollados (por ejemplo, Java en Android o Swift en iOS). De esta forma se puede reutilizar

[4] https://facebook.github.io/react-native/
[5] https://docs.microsoft.com/es-es/xamarin/
[6] https://flutter.dev/
[7] https://es.wikipedia.org/wiki/Compilación_anticipada
[8] https://es.wikipedia.org/wiki/Compilaci%C3%B3n_en_tiempo_de_ejecución

código desarrollado anteriormente o bien realizar funciones específicas de cada plataforma.

Como se puede observar en la Figura 2, una aplicación desarrollada con Xamarin posee una gran cantidad de lógica e interfaz compartida, aunque también permite la ejecución de código específico de cada plataforma para realizar funciones más específicas, como comprobar el nivel de la batería o acceder al almacenamiento del dispositivo.

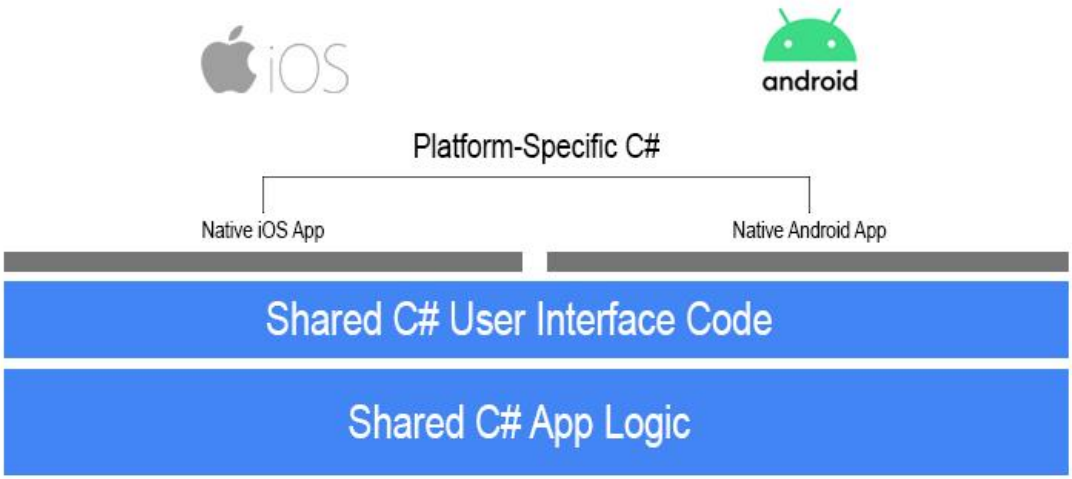

Figura 2: Funcionamiento de Xamarin

Para la interfaz, Xamarin permite utilizar controles genéricos que se mostrarán igual en ambas plataformas, o controles nativos específicos para cada plataforma, para así parecerse visualmente más a una aplicación desarrollada de forma nativa.

2.2. React Native

React Native **[4]** es otro de los *frameworks* de desarrollo móvil multiplataforma más utilizados en la actualidad. Desarrollado por Facebook, React Native es una versión de React especializada en compilar aplicaciones móviles en Android y iOS.

Utiliza JavaScript como lenguaje de programación, y React para la parte de la interfaz de usuario. React es una librería para crear interfaces de usuario desarrollada por Facebook, cuyos elementos clave para crear la interfaz de usurario se llaman componentes.

En React Native, al igual que ocurre en React, un componente puede estar compuesto de otros componentes y de primitivos. En el caso de React Native, estos componentes se convierten en componentes nativos dependiendo de la plataforma en la que estemos ejecutando la aplicación. Por ejemplo, en React Native encontramos

un componente primitivo `Text`, que en Android será traducido como un `TextView` y en iOS como un `UIView` con el texto.

En React Native contamos con dos hilos de ejecución principales: en el hilo de la interfaz, se renderizan los componentes de la interfaz de usuario definidos en React Native, como ocurre en una aplicación nativa; y en el otro hilo se ejecuta la lógica de la aplicación, escrita en JavaScript. Estos hilos se comunican entre sí haciendo uso de un elemento intermedio llamado "Bridge" (puente), que gestiona las llamadas asíncronas entre hilos para asegurar que no se producen interbloqueos, como se puede observar en la Figura 3.

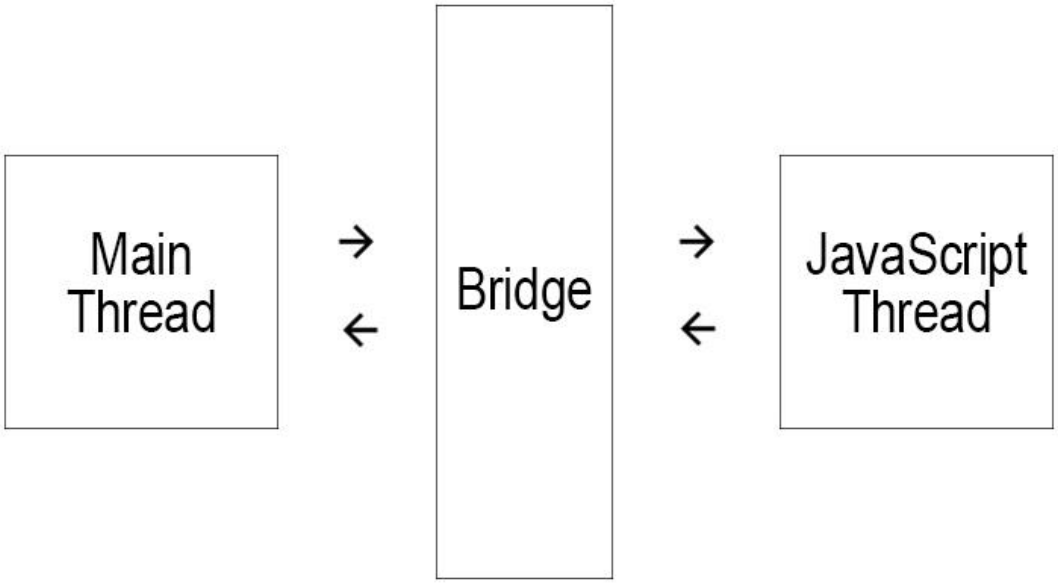

Figura 3: Funcionamiento de React Native

2.3. Flutter

Flutter **[5]** es un nuevo *framework* de desarrollo móvil multiplataforma cuyos focos principales están el rendimiento y la interfaz y experiencia de usuario. Está desarrollado por Google y su primera versión estable fue publicada a finales de 2018.

Utiliza el lenguaje de programación Dart[9], también desarrollado por Google, tanto para la interfaz de usuario como para la lógica de la aplicación. Todo el código escrito en Dart se compila utilizando AOT a código máquina en ambas plataformas para así obtener el mayor rendimiento posible.

En la parte de la interfaz de usuario, los elementos que componen tanto la interfaz como la interacción se llaman *widgets*. En Flutter, todo es un *widget*: un botón, un

[9] https://dart.dev/

menú, un margen, un texto, una fuente o esquema de color, una vista/actividad, incluso la propia aplicación en sí. Estos *widgets* pueden ser añadidos, modificados, reemplazados o eliminados dinámicamente de la interfaz. Muchos de ellos incorporan animaciones automáticas para cuando se añaden, modifican o eliminan, o se les pueden añadir sin demasiado trabajo, lo que mejora la experiencia de usuario de la aplicación desarrollada.

Actualmente Flutter permite compilar la aplicación a Android y iOS, aunque en el futuro también permitirá compilar aplicaciones de escritorio (Windows, macOS, Linux), aplicaciones web, aplicaciones para dispositivos embebidos (Raspberry Pi), e incluso en el Google I/O de 2019 se mostró una aplicación desarrollada en Flutter ejecutándose en un televisor (Android TV) y en un reloj (Wear OS). **[6]** Esto aporta una gran versatilidad al desarrollar una aplicación utilizando Flutter dado que se podrá llegar a un público mucho más amplio y en contextos mucho más variados.

En la Figura 4 se puede observar los componentes que permiten el funcionamiento de Flutter, en los que se observan dos bloques principales: *Framework* y *Engine*. El primero, escrito en Dart, es el que permite al programador desarrollar la aplicación utilizando Flutter y sus componentes. El segundo bloque es el motor que permite ejecutar estas aplicaciones, escrito en C++ y de código abierto, con el que el programador no necesita interactuar directamente dado que Flutter gestiona automáticamente la interacción con dicho motor del *framework*.

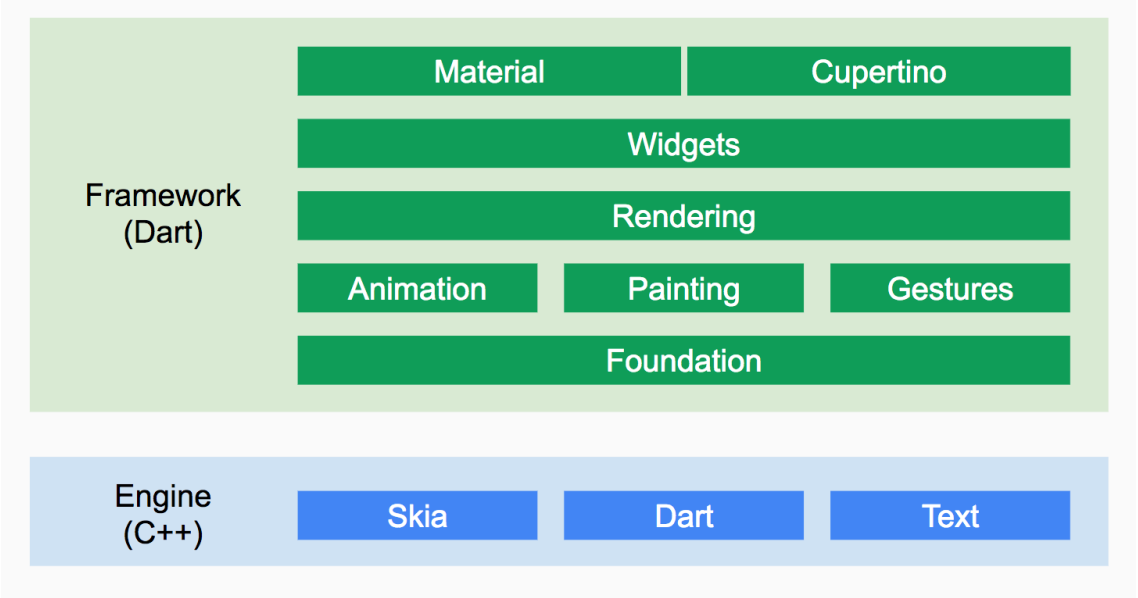

Figura 4: Funcionamiento de Flutter

2.4. Comparativa

En comparación con Xamarin y React Native, Flutter ofrece varias ventajas y mejoras que hacen decantar la balanza a su favor y que se profundizarán en el capítulo 4. La información expuesta en esta comparativa está basada tanto en la experiencia personal como en publicaciones en blogs. **[7] [8] [9]**

- Ofrece un mejor rendimiento gracias a utilizar una compilación AOT en lugar de JIT. Además, utiliza Skia, el motor de renderizado de gráficos en 2D con aceleración por *hardware* que encontramos en Chrome y Android.
- Permite aumentar la productividad gracias a una funcionalidad llamada *Hot Reload*, que permite ver de forma instantánea los cambios que realices en el código o interfaz sin necesidad de reiniciar la aplicación o perder su estado.
- Posee una librería de componentes visuales más avanzada y extensa, gracias a las librerías Material[10] y Cupertino[11].
- Permite añadir animaciones y transiciones a 60 fotogramas por segundo de una forma rápida y sencilla, bien usando código o bien utilizando animaciones de Flare[12].
- Es compatible con herramientas de integración continua optimizadas específicamente para Flutter, como Codemagic[13].
- Actualmente puede compilar aplicaciones a Android y iOS, pero en el futuro permitirá hacerlo también a web, Windows, macOS, Linux y sistemas embebidos.

[10] https://flutter.dev/docs/development/ui/widgets/material
[11] https://flutter.dev/docs/development/ui/widgets/cupertino
[12] https://www.2dimensions.com/about-flare
[13] https://codemagic.io/

3. Estado del arte

3.1. Migración de Android a multiplataforma

Para realizar una migración completa de una aplicación en Android a un *framework* multiplataforma se deberá realizar la migración tanto de la interfaz de usuario como de la lógica. Esto puede ser una operación de riesgo elevado a corto plazo pero que aportará una reducción de costes y recursos necesarios a medio y largo plazo para la empresa que migre sus aplicaciones, dado que sólo necesitarán programar las funcionalidades e interfaces una vez y tanto los usuarios de Android como los de iOS se podrán beneficiar más rápidamente de dichos cambios.

Antes de comenzar la migración, se deberá valorar, por ejemplo, si aportará beneficios a la aplicación a migrar. Si la aplicación que se desea migrar de Android a un *framework* multiplataforma tiene como público objetivo exclusivamente usuarios de Android, realizar esta migración a corto plazo representa un riesgo evitable, dado que no se va a exprimir todo el potencial que aportan este tipo de *frameworks*. Sin embargo, si el público objetivo mayoritario es Android, pero se desea incluir también a los usuarios de iOS y así poder llegar a más usuarios, este tipo de migración resulta muy interesante y beneficiosa.

Para una empresa pequeña realizar una migración de su app nativa a un *framework* multiplataforma puede tener un alto riesgo al no disponer de los recursos suficientes para crear, por ejemplo, un equipo que se dedique exclusivamente a realizar la migración, como sí puede ocurrir en una empresa de mayor tamaño. Tras finalizar la migración, la aplicación estará disponible también para iOS, llegando así a más usuarios. Si la empresa que realiza la migración sólo tenía la aplicación disponible para Android, ahora la tendrá disponible tanto en Android como iOS sin necesidad de aumentar los costes por aumentar el equipo. Si la empresa ya tuviese su aplicación disponible en ambas plataformas, podrá reducir considerablemente los costes al no necesitar dos equipos distintos para las dos aplicaciones, sino que un solo equipo será el encargado de ambas aplicaciones.

La forma más óptima para llevar a cabo esta migración es realizar ingeniería inversa de la aplicación nativa existente para identificar sus requisitos y garantizar así que durante la migración se cubrirá la totalidad de los requisitos funcionales existentes en la aplicación inicial. Una vez se obtiene un listado de los requisitos, se puede abstraer el diseño que se deberá implementar en la aplicación migrada final.

3.1.1. Migración de Android a Xamarin

Uno de los mensajes más relevantes para realizar este trabajo se encuentra en StackOverflow, en el que el usuario Lex Li responde a un usuario que pregunta qué puede hacer para migrar su aplicación Android a iOS usando Xamarin. En la respuesta, Lex Li expone que la interfaz de usuario de Android puede ser reutilizada con facilidad dado que la interfaz de usuario de los proyectos realizados con Xamarin.Android comparten el mismo formato XML que los proyectos desarrollados en Android nativo, por lo que se podrá reutilizar la interfaz existente. Se menciona además que las librerías de terceros se pueden importar en Xamarin, aunque algunas ya incluyen directamente sus componentes de Xamarin. Finalmente, añade que el código Java general se puede transformar en código C# haciendo uso de la herramienta Sharpen[14]. **[10]**

El otro mensaje referente a la migración de Android a Xamarin se encuentra en el foro Reddit, escrito por el usuario “unndunn”, quien responde a un usuario que quiere migrar su aplicación de Android a iOS usando Xamarin. En este mensaje, se expone que el usuario que quiere migrar la aplicación deberá aprender C# y .NET, pero que, para un programador familiarizado con Java, la adaptación a C# no debería ser demasiado costosa. Se añade además que, al querer realizar un desarrollo plataforma, se debería aprender a utilizar un *framework Model-View-ViewModel* (MVVM), como por ejemplo MvvmCross[15], para así poder compartir la lógica de la aplicación entre las aplicaciones Android y iOS. Además, se deberá refactorizar la aplicación Android existente, dado que la interfaz de usuario puede permanecer intacta pero las actividades y fragmentos existentes en la aplicación se deberán adaptar a la nueva arquitectura MVVM. Finalmente, se explica que se deberá aprender a desarrollar interfaces de usuario para iOS, o aprender a usar Xamarin.Forms, el *framework* de desarrollo de interfaces multiplataforma de Xamarin. **[11]**

3.1.2. Migración de Android a React Native

En el caso de querer realizar la migración a React Native, Kevin Pelgrims expone el proceso de migración de su aplicación Imagine[16] a React Native. En su caso, ya disponían de aplicaciones nativas en producción desarrolladas con Java (Android) y Swift (iOS), pero al tener que incorporar nuevas funcionalidades en la aplicación, se encontraba con el problema de tener que desarrollarlas dos veces, una para cada sistema operativo, creando así una gran cantidad de código duplicado.

En dicha publicación, se expone que hay dos posibles caminos para migrar una aplicación nativa en Android a React Native: añadir las nuevas funcionalidades usando React Native e ir reemplazando, poco a poco, las funcionalidades existentes por sus equivalentes en React Native; o reescribir completamente la aplicación para así poder sacar el máximo rendimiento posible a React, opción por la que opta el autor de la publicación. **[12]** Aunque el autor expone su experiencia durante el proceso de

[14] https://github.com/mono/sharpen
[15] https://www.mvvmcross.com/
[16] https://getimagine.io/

migración, ventajas e inconvenientes, no aporta demasiada información acerca de las acciones realizadas para completar dicha migración más allá del “comenzar el proyecto desde 0”.

Existe también una pregunta relacionada en StackOverflow en la que un usuario quiere migrar su aplicación de Android a React Native. El usuario “Akhi” responde que, aunque la mejor opción es crear un nuevo proyecto desde 0, se pueden convertir las vistas de la aplicación nativa en Android a componentes de React Native y crear un *Bridge* que permita la comunicación de la aplicación React Native comunicarse con el código nativo de Android. **[13]** El problema de esta opción es que no se podría ejecutar la aplicación en iOS dado que la lógica de la aplicación está únicamente en Android, por lo que habría que migrar ese código nativo de Android a React Native o abstraer la mayor parte del código que pueda ser compartido entre ambas plataformas e implementar de forma nativa tanto en Android como en iOS el código específico de cada sistema operativo.

3.2. Crítica al estado del arte

Actualmente no existe una documentación clara a seguir para realizar una migración de una aplicación existente desarrollada en nativo en Android hacia Flutter, ni en los trabajos finales de grado de alumnos de la Escuela Técnica Superior de Ingeniería Informática (ETSINF) ni en blogs disponibles en internet. De igual forma, la documentación de cómo realizar una migración de una aplicación nativa existente en Android hacia otros *frameworks* de desarrollo multiplataforma como Xamarin o React Native es prácticamente inexistente, dado que la poca información que hay se basa en un “comenzar el proyecto completo desde 0”.

Por este motivo, en este trabajo de fin de grado se propondrá una arquitectura para estructurar de forma escalable un proyecto en Flutter y se establecerán una serie de patrones de migración tanto de la interfaz como de la lógica con la que se pretende guiar al programador que lea el documento durante la migración de su aplicación.

De entre todos los *frameworks* disponibles a los que se podría migrar una aplicación nativa en Android, Flutter es el más interesante dado que ofrece claras mejoras frente a otras alternativas. Además, si se opta en la actualidad por utilizar Flutter, en el futuro la aplicación migrada ganará mucha versatilidad al poder ser ejecutada no solo en Android y iOS sino también en la web o sistemas operativos de escritorio.

3.3. Propuesta

La solución para el caso de estudio que se planteará más adelante pasará por utilizar Flutter como *framework* para el desarrollo de la app resultante de la migración.

Al elegir Flutter como *framework* destino al que se migrará la aplicación, se obtendrán beneficios como un mejor rendimiento de la aplicación, mayor productividad para el equipo de desarrollo, más rapidez y versatilidad a la hora de implementar las interfaces y, además, la aplicación resultante estará disponible en iOS, con el beneficio de que en el futuro, con pocas modificaciones, la aplicación también podría estar

también disponible en más plataformas como en la web, en sistemas operativos de escritorio (Windows, macOS, Linux) o en Fuchsia, un sistema operativo de código abierto actualmente en desarrollo y creado por Google.

Además, gracias al conjunto de patrones de migración, tanto de interfaz como de lógica, se espera guiar al programador que va a realizar la migración sobre cómo proceder para migrar los componentes visuales de Android más comunes y algunas de las situaciones más comunes en la lógica, mostrando ejemplos de las equivalencias.

4. Flutter

Flutter es un *framework* de desarrollo de aplicaciones móviles multiplataforma desarrollado por Google. Su primera versión, conocida como “Sky”, fue presentada en 2015 durante el Dart Developer Summit. La primera versión estable fue presentada al público el 4 de diciembre de 2018 durante el evento Flutter Live en Londres (10 meses después de publicar la primera versión beta). **[14]** Como lenguaje de programación utiliza Dart, también desarrollado por Google, aunque se pueden implementar funciones específicas haciendo uso de otros lenguajes como Java o Swift (en Android y iOS, respectivamente). A continuación, se introducen las principales funciones y características disponibles actualmente en el Flutter.

4.1. Instalación

En primer lugar, se tendrá que descargar e instalar Android Studio [17] , independientemente de la plataforma (Windows, macOS, Linux). Una vez instalado, durante el proceso de instalación, se deberá instalar la última versión de *Android SDK*, *Android SDK Build-Tools*, y *Android SDK Platform-Tools*. Es imprescindible disponer de la última versión para desarrollar correctamente para Android con Flutter. Este paso es común para todos los sistemas operativos desde los que se puede desarrollar para Flutter.

El proceso de instalación que se describirá a continuación es para el sistema operativo Windows, aunque se puede instalar de forma similar en macOS, Linux o ChromeOS siguiendo la guía de instalación oficial **[15]**, accediendo a través de los botones que se muestran a continuación en la Figura 5:

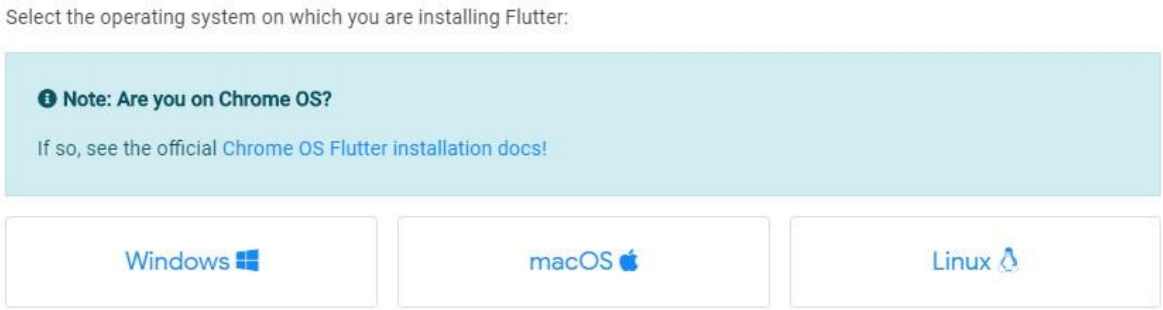

Figura 5: Instalación de Flutter en distintos sistemas operativos

[17] https://developer.android.com/studio

4.1.1. Windows

Para comenzar la instalación en Windows, primero se deberá descargar el Flutter SDK[18] desde la web oficial de Flutter para así descargar la última versión estable disponible del *framework*. Una vez descargado el SDK comprimido en formato ZIP, se deberá descomprimir y se extraerá una carpeta llamada "flutter" que contiene el SDK. Esta carpeta se deberá colocar en la ruta que queramos para el SDK (por ejemplo, en "`C:\src\flutter`"), pero hay que tener en cuenta no moverla a un directorio que requiera privilegios (por ejemplo, "`C:\Program Files`").

A continuación, dentro de la carpeta "flutter" se deberá buscar y ejecutar un archivo llamado "flutter_console.bat". Al ejecutar dicho archivo se abrirá la consola de Flutter, como se muestra en la Figura 6, desde la que podremos ejecutar todos sus comandos.

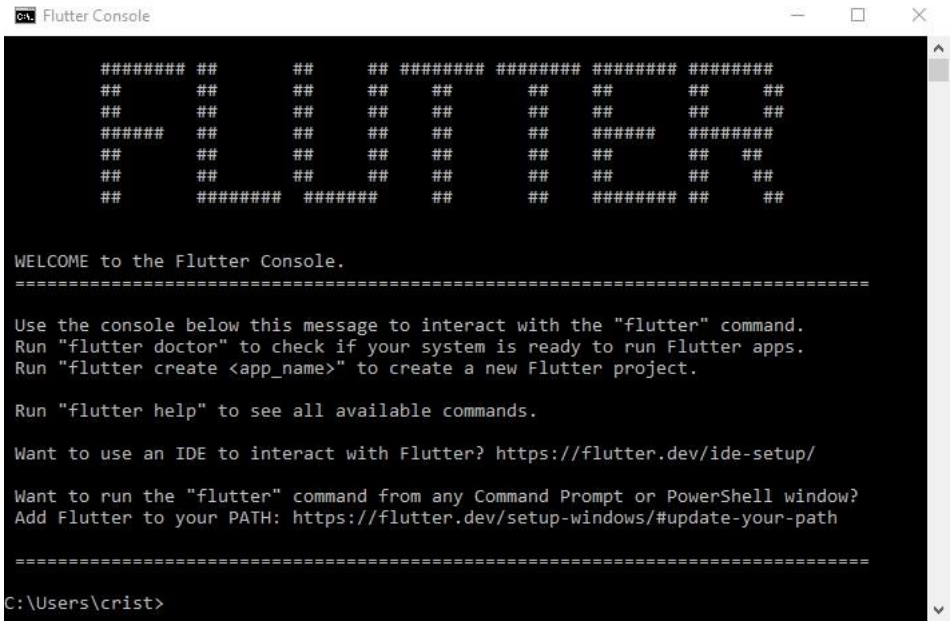

Figura 6: Consola de Flutter en Windows

Si se quieren utilizar los comandos de Flutter desde cualquier terminal, hay que añadir Flutter a la variable de entorno "PATH". Para ello, desde la barra de búsqueda del menú de inicio de Windows, se escribirá "env" y se seleccionará la opción "Editar las variables de entorno de esta cuenta". Aparecerá una ventana en la que, en el bloque llamado "Variables de usuario para <usuario>", se deberá seleccionar la variable "PATH" y hacer clic en el botón "Editar…" (en caso de que no exista dicha variable, se deberá crear haciendo clic en el botón "Nueva…"). A esta variable se le añadirá un nuevo valor con la ruta completa de la carpeta "`flutter\bin`" (por ejemplo, "`C:\src\flutter\bin`"), añadiendo el separador "`;`" entre las posibles rutas existentes previamente y la nueva ruta que se añada. Tras guardar los cambios, se deberán cerrar todas las consolas que haya abiertas y ya se podrá abrir una nueva en la que se podrán utilizar los comandos de Flutter.

[18] https://flutter.dev/docs/get-started/install/windows

4.2. Dart

Dart **[16] [17]** es un lenguaje de programación de código abierto, orientado a objetos, basado en clases y con recolector de basura (*garbage collector)*, desarrollado por Google y presentado por primera vez en 2011, aunque su primera versión estable no llegó hasta 2013. Su principal objetivo en su nacimiento era ofrecer una alternativa más moderna a JavaScript como principal lenguaje de programación web. Utiliza una sintaxis similar a C que puede ser compilado a código fuente de JavaScript gracias al compilador "dart2js". Además de JavaScript, se pueden desarrollar aplicaciones compiladas con AOT, como las realizadas utilizando Flutter; o aplicaciones que pueden ejecutarse en la línea de comandos, como un servidor web.

Para desarrollar aplicaciones utilizando Dart, se pueden utilizar varios entornos de desarrollo distintos, como Dart Editor, Eclipse (utilizando el *plugin* de Dart), Android Studio, Visual Studio Code o Atom, entre otros. Además, existe un editor online para experimentar con las funciones de Dart y ejecutar el código llamado DartPad.

A continuación, se muestra un ejemplo de código donde se calcula el enésimo número de Fibonacci.

```
void main() {
  int i = 0;
  print('fibonacci($i) = ${fibonacci(i)}');
}

/// Calcula el número de Fibonacci
int fibonacci(int n) {
  return n < 2 ? n : (fibonacci(n - 1) + fibonacci (n-2));
}
```

Figura 7: Función Fibonacci en Dart

Como se puede observar, de una forma similar a lo que se encuentra en Java, el punto de entrada de la ejecución de una aplicación escrita en Dart es el método `main()`. Dentro de este método se puede ver cómo se declara una variable llamada "`i`", precedida de la palabra reservada "`var`", por lo que esta variable infiere el tipo dependiendo del valor que se guarda en ella (en este caso, un número entero). El siguiente método que se puede observar es el método `fibonacci(int n)`, que recibe un número entero y devuelve como resultado de su computación otro número entero. En el interior del método se puede observar cómo se utiliza un operador ternario y recursividad para calcular el resultado de la operación.

4.2.1. Programación asíncrona

Dart soporta programación asíncrona de una forma muy sencilla. El código Dart se ejecuta sobre un único hilo de ejecución, aunque se puede ejecutar también de forma concurrente haciendo uso de la librería `Isolate`[19].

Para convertir una función escrita en Dart en asíncrona basta con hacer dos sencillos cambios: marcarla con la palabra reservada "`async`" y cambiar el tipo de objeto que devuelve "T" por un "`Future<T>`". De esta forma, al llamar a esta función, se pone en la cola de ejecución, por lo que devuelve un objeto `Future` incompleto. Cuando finaliza su ejecución, este objeto `Future` se completa con un valor o error.

Dentro de una función asíncrona se puede hacer uso también de la palabra reservada "`await`" antes de realizar la llamada a una función asíncrona, lo que pausará la ejecución hasta que el `Future` sea resuelto con un valor o error.

Si no se quiere pausar la ejecución para hacer la llamada asíncrona, también se puede hacer uso de del método "`then((result) {})`". que dado un valor o error "`result`", resultante de completar el `Future`, se ejecutará el código que se declare entre las llaves en el que se puede hacer uso de dicho valor.

4.3. Widgets

En Flutter la interfaz de usuario se construye mediante un árbol de *widgets*. Todo lo que ve el usuario es un *widget*: la propia aplicación en sí, una nueva ventana, un campo de texto o un detector de gestos, entre otros. Flutter tiene una gran variedad de *widgets* primitivos que se pueden modificar y personalizar según las especificaciones requeridas en la aplicación. Además, Flutter integra dos extensas librerías de *widgets* modificados: Material, que sigue las directrices de diseño de *Material Design* **[18]** especificadas por Google; y Cupertino, que provee una forma simple de integrar *widgets* que sigan las *Human Interface Guidelines* **[19]** especificadas por Apple. También existen una serie de *widgets* adaptables que, dependiendo de la plataforma en la que se estén ejecutando (Android o iOS), se renderizarán siguiendo las guías de diseño de su plataforma (Material o Cupertino, respectivamente).

4.3.1. Material

En el caso de la librería Material, se puede acceder a una gran variedad de *widgets* que siguen las directrices de diseño de *Material Design*. Estos *widgets* ya incorporan todos los requisitos especificados por Google (elevación, interacción, movimiento…) y pueden ser incorporados en una app de una forma muy sencilla tras importar la librería "`package:flutter/material.dart`".

[19] https://api.dart.dev/stable/dart-isolate/dart-isolate-library.html

```dart
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter TFG App'),
      ), // AppBar
      floatingActionButton: FloatingActionButton(
        child: Icon(Icons.add),
        onPressed: () => null,
      ), // FloatingActionButton
      body: Center(
        child: RaisedButton(
          child: Text('¡Púlsame!'),
          onPressed: () => null,
        ), // RaisedButton
      ), // Center
    ); // Scaffold
  }
}
```

Figura 8: Ejemplo de uso de widgets Material

4.3.2. Cupertino

Existe además la librería Cupertino, que provee, de forma similar a la librería Material, *widgets* que siguen las directrices de diseño de Apple especificadas en las *Human Interface Guidelines*. Estos *widgets* de la librería Cupertino permiten incorporar en la aplicación a desarrollar de forma rápida y sencilla la interfaz nativa que puede esperar un usuario que utilice el sistema operativo iOS, incluyendo animaciones, navegación, selectores, etc. Para poder utilizar los *widgets* de la librería Cupertino, debemos importar la librería "`package:flutter/cupertino.dart`" y ya se tendrá acceso al catálogo completo de Cupertino.

```dart
}

class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return CupertinoPageScaffold(
      navigationBar: CupertinoNavigationBar(
        middle: Text('Flutter TFG App'),
      ), // CupertinoNavigationBar
      child: Center(
        child: CupertinoButton.filled(
          child: Text('¡Púlsame!'),
          onPressed: () => null,
        ), // CupertinoButton.filled
      ), // Center
    ); // CupertinoPageScaffold
  }
}
```

Figura 9: Ejemplo de uso de widgets Cupertino

4.3.3. Adaptive Widgets

Además de las librerías Material y Cupertino, existen algunos *widgets* que dependiendo del sistema operativo en el que se están ejecutando (Android o iOS) son renderizados como *widgets* Material o Cupertino. Actualmente no todos los *widgets* tienen una implementación utilizando el constructor `adaptive()`, pero cada vez son más los que soportan este tipo de implementación.

```
double _value = 0;

Slider.adaptive(
  value: _value,
  min: 0.0,
  max: 10.0,
  onChanged: (double value) {
    setState(() {
      _value = value;
    });
  },
);
```

Figura 10: Ejemplo de uso de Adaptive Widgets

Dado que la cantidad de *widgets* que soportan este tipo de constructor todavía no es muy amplia, se puede obtener un efecto similar comprobando en tiempo de ejecución el sistema operativo donde se está ejecutando la aplicación. De esta forma, podemos renderizar un *widget*, texto, imagen o incluso una app completamente distinta en cada sistema operativo dependiendo de las necesidades del proyecto a desarrollar. Como actualmente Flutter solo soporta Android o iOS de forma estable, se puede realizar la comprobación del sistema operativo en el que se está ejecutando la aplicación utilizando un condicional u operador ternario. Para ello, se deberá importar la librería "`dart:io`", con la que se podrá comprobar el sistema operativo en el que se está ejecutando la aplicación.

```
class HomePageAdaptive extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Platform.isAndroid ? HomePageMaterial() : HomePageCupertino();
  }
}
```

Figura 11: Ejemplo de comprobación del sistema operativo actual

4.4. Stateless Widgets

En Flutter, un *stateless widget* está compuesto por uno o más *widgets* y sus propiedades son inmutables, es decir, las variables declaradas dentro de un *stateless widget* deberán ser declaradas finales. Este tipo de *widgets* son muy útiles a la hora de componer interfaces personalizadas y reducir la cantidad de código duplicado. Si la interfaz que se desea componer no depende de nada más que los valores que se especifican al componerla y los posibles valores del contexto en el que se va a componer la interfaz (por ejemplo, los colores especificados en el tema de la aplicación), la mejor opción será utilizar un *stateless widget*.

```
class ProductRow extends StatelessWidget {
  final String name;
  final double price;

  ProductRow({this.name, this.price});

  @override
  Widget build(BuildContext context) {
    return ListTile(
      title: Text(this.name),
      subtitle: Text('Precio: $price'),
    ); // ListTile
  }
}
```

Figura 12: Ejemplo de un `StatelessWidget`

4.5. Stateful Widgets

Al contrario que ocurre con los *stateless widgets*, los *stateful widgets* sí pueden mutar respondiendo así a cambios en el contexto o sus propiedades. Para inicializar los recursos que se usarán al componer el *widget*, se deberá sobrescribir el método `initState()`. Cuando se desee actualizar el estado de alguna de las propiedades del *widget*, se deberá realizar una llamada al método `setState(() {})`, en el que se actualizarán los valores que cambiarán en el estado del *widget*. A continuación, se muestra un ejemplo en el que, al pulsar el botón, se actualizará el texto mostrando la cantidad total de pulsaciones realizadas.

```dart
class StatefulSample extends StatefulWidget {
  @override
  _StatefulSampleState createState() => _StatefulSampleState();
}

class _StatefulSampleState extends State<StatefulSample> {
  int _counter;

  @override
  void initState() {
    _counter = 0;
    super.initState();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter TFG App'),
      ), // AppBar
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            RaisedButton(
              child: Text('¡Púlsame!'),
              onPressed: () => _add(),
            ), // RaisedButton
            Text('Pulsado $_counter veces'),
          ], // <Widget>[]
        ), // Column
      ), // Center
    ); // Scaffold
  }

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

Figura 13: Ejemplo de un `StatefulWidget`

4.6. Inherited Widgets

Cuando una aplicación desarrollada en Flutter comienza a introducir árboles de *widgets* más extensos y complejos, puede resultar excesivamente tedioso realizar el paso de parámetros desde un *widget* dado hasta otro que puede estar varios niveles más abajo. Para realizar ese paso de parámetros, hay que enviar dicho parámetro en el constructor de todos los *widgets* que se encuentren en el árbol en el camino hasta llegar al *widget* que utilizará dicho parámetro. Gracias al uso de *inherited widgets*, esta operación es mucho más sencilla, dado que únicamente se deberá rodear el árbol de *widgets* de uno de tipo `InheritedWidget`.

Al declarar los parámetros que se utilizarán más adelante en el `InheritedWidget` inicial, que además deberán ser declarados finales por tratarse de un *widget* inmutable, se podrá acceder a dichos parámetros en cualquiera de los *widgets* que se encuentren en un nivel inferior en la jerarquía del árbol de *widgets*.

Por ejemplo, en Flutter, se puede obtener cualquiera de los valores del tema de la aplicación desde cualquier parte de ella, dado que el tema es un `InheritedWidget` y se puede acceder a sus valores de forma sencilla. Si se quiere obtener el color primario de la aplicación, simplemente se deberá realizar una llamada a `Theme.of(context).primaryColor`, que devolverá un *widget* de tipo `Color` que se podrá utilizar, por ejemplo, para poner de fondo en un botón.

4.7. Navegación

La navegación en Flutter se basa en una pila de rutas a la que se pueden añadir y quitar rutas para navegar por las distintas vistas de la aplicación. Para ello, se hace uso del *widget* `Navigator`, que será el encargado de gestionar las rutas de la pila. Además, en el *widget* de la app, se pueden especificar rutas con nombre, para que sea más sencillo realizar las llamadas, especialmente cuando en una aplicación pueden existir un gran número de rutas distintas. Para acceder al objeto `Navigator` de la aplicación, lo haremos mediante una llamada a `Navigator.of(context)`, en el que se puede realizar llamadas a las distintas funciones según se desee. Algunas de las más comunes son:

- **`pop()`:** Elimina la ruta actual de la pila.
- **`popAndPushNamed()`:** Elimina la ruta actual de la pila y la reemplaza por una nueva ruta con nombre.
- **`popUntil()`:** Elimina las rutas de la pila hasta que se cumple una condición dada.
- **`push()`:** Añade una ruta nueva a la pila y la muestra.

4.8. Animaciones

Flutter permite añadir animaciones de forma sencilla a las aplicaciones, renderizándolas a 60 fotogramas por segundo para obtener una experiencia de usuario agradable y suave. Ya incluye de forma nativa muchas animaciones en algunos *widgets* como puede ser `FloatingActionButton`, en el que, si el usuario navega entre dos rutas que contienen uno, se animará entre las distintas posiciones y cambiará su tamaño para adaptarse a la posible diferencia de tamaño dependiendo de si su contenido cambia (por ejemplo, en la ruta inicial es un icono y en la ruta destino es un icono con texto).

Además, existen paquetes como Animator[20], que permite añadir de forma rápida y sencilla animaciones a los *widgets*, reduciendo el esfuerzo y las líneas de código necesarias para realizar la misma animación sin hacer uso del paquete.

[20] https://pub.dev/packages/animator

```
child: IconButton(
  tooltip: 'Tienda',
  icon: Animator(
    tween: Tween<double>(begin: -0.2, end: 0.2),
    duration: Duration(milliseconds: 300),
    curve: Curves.easeInOut,
    cycles: 0,
    builder: (Animation anim) => Transform.rotate(
      angle: anim.value,
      child: Icon(Icons.shopping_basket),
    ), // Transform.rotate
  ), // Animator
  color: Colors.amber,
  onPressed: () => null,
), // IconButton
```

Figura 14: Ejemplo de animación de agitar usando Animator

También se pueden incorporar en la aplicación fácilmente animaciones realizadas utilizando la herramienta Flare desarrollada por la empresa 2Dimensions. Estas animaciones se pueden controlar vía código e incluso se pueden realizar animaciones con las que el usuario pueda interactuar para realizar alguna acción.

```
child: FlareActor(
  "animations/LogoSwapp.flr",
  alignment: Alignment.center,
  fit: BoxFit.cover,
  animation: 'Untitled',
), // FlareActor
```

Figura 15: Ejemplo de animación usando Flare

4.9. Platform Channels

Dado que Flutter es un *framework* para desarrollar aplicaciones multiplataforma, puede ocurrir que se necesite hacer llamadas a código específico de cada plataforma, por ejemplo, para obtener el nivel de la batería. Para realizar esta función, se deberá crear un objeto `MethodChannel` con un identificador único creado, por ejemplo, por el identificador de la app junto con el nombre del canal.

```dart
class _HomePageState extends State<HomePage> {
  static const platform = const MethodChannel('com.czazo.tfg/api');

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Flutter TFG App'),
      ), // AppBar
      body: Center(
        child: RaisedButton(
          child: Text('¡Púlsame!'),
          onPressed: () async {
            try {
              final int result = await platform.invokeMethod('getAPILevel');
              print('API Level: $result');
            } on PlatformException catch (e) {
              print('Error al obtener la versión de la API');
              print(e.toString());
            }
          },
        ), // RaisedButton
      ), // Center
    ); // Scaffold
  }
}
```

Figura 16: Creación de un Platform Channel en Dart (Flutter)

Una vez se ha creado el Platform Channel en Dart, se deberá implementar en las plataformas específicas en las que funcionará nuestra aplicación. Si se compilará una versión para Android, la implementación se hará en Java o Kotlin, mientras que, si se compilará una versión para iOS, la implementación se hará en Objective-C o Swift.

```
private static final String CHANNEL = "con.czazo.tfg/api"

 •••

new MethodChannel(getFlutterView(), CHANNEL).setMethodCallHandler(
    new MethodChannel.MethodCallHandler() {
        @Override
        public void onMethodCall(MethodCall call, MethodChannel.Result res) {
            if (call.method.equals("getAPILevel")) {
                int apiLevel = android.os.Build.VERSION.SDK_INT;
                if (apiLevel > 0) {
                    res.success(apiLevel);
                } else {
                    res.error("ERROR", "Ha ocurrido un error", apiLevel);
                }
            } else {
                res.notImplemented();
            }
        }
    }
);
```

Figura 17: Implementación de un Platform Channel en Java (Android)

En el proceso de migración de una aplicación existente en Android, esta funcionalidad puede ayudar a tener lista una primera versión migrada a Flutter, pero únicamente disponible para Android. Esto es posible dado que, tras migrar la interfaz de usuario a Flutter, se puede aprovechar la mayoría del código Java o Kotlin sin mucho esfuerzo haciendo uso de los *Platform Channels*, e ir posteriormente realizando la migración de la lógica poco a poco.

Si se quiere tener la aplicación también disponible para iOS, no bastará con implementar únicamente los *Platform Channels* de Android, sino que se deberán implementar también los de iOS. Sin embargo, esto puede crear un problema de código duplicado y la recomendación es migrar toda la lógica posible de Java o Kotlin a Dart y dejar los *Platform Channels* únicamente para código específico para cada plataforma.

5. Arquitectura propuesta para un proyecto en Flutter

Al crear un nuevo proyecto en Flutter, se crea una estructura compuesta por carpetas y archivos que otorgan una base simple junto con una aplicación básica de ejemplo y un test para dicha aplicación. En la arquitectura que se crea por defecto al crear un nuevo proyecto encontramos varias carpetas y archivos importantes:

- **`android/`:** Es la carpeta que contiene todo el código necesario para compilar la aplicación para Android y en la que se puede modificar, por ejemplo, el icono, el manifiesto y la pantalla de inicio; o añadir código específico del sistema operativo haciendo uso de los *Platform Channels*.
- **`ios/`:** De forma similar a la carpeta anterior, se crea una carpeta que contiene el subproyecto encargado de posibilitar la compilación de la aplicación para el sistema operativo iOS. En ella, además, se pueden declarar *Platform Channels* para el código específico de esta plataforma, o cambiar el icono, pantalla de inicio parámetros de la propia aplicación como la versión mínima de iOS que se soportará.
- **`lib/`:** En esta carpeta es donde se deberá añadir todo el código escrito en Dart y es la carpeta principal del proyecto. Por defecto, se crea un archivo `main.dart` con un ejemplo simple de una aplicación en Flutter.
- **`test/`:** Es la carpeta en la que se deberán incluir todos los *tests* que se desarrollen para la aplicación, cuyo nombre de archivo deberá ir precedido del sufijo "`_test`" y ser archivos con extensión "`.dart`", por ejemplo, "`homescreen_test.dart`".
- **`pubspec.yaml`:** Este archivo en formato YAML será el encargado de gestionar el nombre, descripción, autores, versión, dependencias, imágenes, etc. del proyecto.

A continuación, se propone una arquitectura para estructurar el proyecto a desarrollar obtenida tras semanas de iteraciones y mejoras en el proyecto de migración que se detallará más adelante en el caso de estudio. Dado que al comienzo del proyecto de migración no existía prácticamente información ni buenas prácticas relacionadas con la arquitectura, se optó por iterar desde la arquitectura inicial que crea un proyecto vacío, obteniendo la arquitectura expuesta a continuación. De forma adicional, se puede consultar la página web Flutter Architecture Samples[21], en la que se exponen distintas arquitecturas posibles para realizar una misma aplicación de ejemplo, para observar su funcionamiento y ayudar en la decisión sobre cuál sería la mejor para un determinado proyecto.

[21] https://fluttersamples.com/

Primeramente, en la carpeta raíz del proyecto, se crearán tres carpetas llamadas "`animations`" para, en caso de utilizarse, añadir los archivos creados con la herramienta Flare; "`fonts`" para, en caso de ser necesario, añadir fuentes personalizadas al proyecto; e "`images`", donde se añadirán las imágenes que se necesiten en proyecto. Si el proyecto va a necesitar una gran cantidad de imágenes, se pueden estructurar en subcarpetas, por ejemplo, para los iconos o creando varias subcarpetas para separar las imágenes que se necesiten en las distintas rutas que haya en la aplicación. Dado que para un determinado proyecto es posible que no se usen todas las carpetas (animaciones, fuentes e imágenes), su creación será opcional dependiendo de las necesidades del proyecto. Se deben añadir estas nuevas rutas de fuentes e imágenes creadas al archivo "`pubspec.yaml`" para importar sus recursos como se muestra a continuación:

```yaml
flutter:
  uses-material-design: true
  assets:
    - animations/
    - images/
    - images/onboarding/
  fonts:
    - family: IndieFlower
      fonts:
        - asset: fonts/IndieFlower.ttf
```

Figura 18: Recursos importados en el archivo `pubspec.yaml`

Después, dentro de la carpeta `lib/`, se deberán crear las siguientes carpetas:

- **`model/`:** La estructura de los diversos objetos que se vayan a necesitar en la aplicación se deberán incluir en esta carpeta. Por ejemplo, se puede crear un archivo "`user.dart`" en el que se declaren los parámetros, constructor y funciones específicas de este tipo de objeto.
- **`screens/`:** Esta carpeta será probablemente una de las más extensas, puesto que todas y cada una de las vistas (rutas) de la aplicación se deberán declarar aquí. Por ejemplo, la vista principal puede estar declarada en el archivo "`home.dart`". Si las vistas son ligeramente complejas y necesitan funciones o *widgets* específicos, en lugar de crear un archivo para la vista, se creará una subcarpeta con el nombre de la vista, por ejemplo "`home/`". con los siguientes archivos, según se necesite:
  - **`index.dart`:** En este archivo se declarará únicamente el árbol de *widgets* de la interfaz de la aplicación.
  - **`utils.dart`:** Aquí se deberán declarar todas las posibles funciones que se vayan a utilizar exclusivamente en la vista declarada en el archivo "`index.dart`" de su misma carpeta, por ejemplo, la validación de un formulario o la gestión de los distintos estados de la vista.
  - **`widgets.dart`:** Todos los *widgets* que se vayan a utilizar únicamente en la vista de la carpeta en la que se encuentra el archivo se deberán declarar en este archivo. Si en algún momento uno de los *widgets* se va a utilizar en otra

vista, se podrá realizar una refactorización para adaptar mejor el nombre del *widget* y se deberá mover a la carpeta "`widgets/`" para conservar la consistencia de la estructura de la aplicación.

- **`utils/`:** Las funciones que se utilicen en varias partes de la aplicación se deberán incluir aquí. Se puede diferenciar según el tipo de utilidad, por ejemplo, y crear un archivo "`api.dart`" en el que se añadan las llamadas al servidor y un archivo "`utils.dart`", en el que se añadan funciones comunes como formatear una fecha, comprobar si el usuario ha iniciado sesión, obtener la versión actual de la aplicación, etc.
- **`widgets/`:** En esta última carpeta se deberán incluir todos los *widgets* que se reutilicen en varias partes de la app, como puede ser un botón o un campo de texto. De esta forma, se puede garantizar que los *widgets* que se utilizan en la aplicación son iguales y consistentes.

6. Migración de la interfaz

A continuación, se van a exponer una serie de patrones para mostrar al programador qué *widgets* son los equivalentes para poder migrar con mayor facilidad y rapidez la interfaz de la aplicación en Android, siguiendo las directrices de diseño de Material Design (y, por tanto, la librería Material de Flutter) a Flutter.

6.1. Activity

En Android, un `Activity` es una vista específica de la aplicación. En Flutter, estos componentes se llaman rutas. Para navegar entre las distintas vistas de una aplicación en Flutter, se hace uso del objeto `Navigator`. Dado que este objeto funciona como una pila de rutas, para navegar a una nueva ruta se añadirá un *widget* `MaterialPageRoute` a la pila. Dentro de este *widget* que representa la nueva ruta se construirá un *widget* llamado `Scaffold`, que representa la estructura básica visual de una vista en Flutter. En este *widget* se pueden incorporar una gran cantidad de *widgets* fácilmente, como un botón flotante, una barra de navegación o un menú deslizable lateral. Este *widget* en sí no contiene ningún elemento visual más allá del color de fondo correspondiente al tema de la aplicación (gris claro por defecto).

```
onPressed: () {
  Navigator.push(
    context,
    MaterialPageRoute(
      builder: (context) => MainRoute(),
    ), // MaterialPageRoute
  );
},
```

Figura 19: Ejemplo de uso del widget `MaterialPageRoute` en Flutter

6.2. TextView

Los componentes de interfaz `TextView`, utilizados para mostrar texto dentro de la aplicación, son unos de los componentes más fáciles de migrar a Flutter. En estos componentes se puede distinguir entre dos partes principales: el texto a mostrar y el estilo del texto (fuente, tamaño, etc.).

Para mostrar texto en Flutter, se deberá hacer uso del *widget* `Text`, que recibe como primer parámetro una cadena de texto con el texto a mostrar y, opcionalmente, una serie de parámetros que permitirán modificar el estilo y cómo se muestra el texto en la aplicación: número máximo de líneas que ocupará el texto, cómo mostrar visualmente que hay más texto del que se puede mostrar (por ejemplo, haciendo uso de puntos suspensivos o un degradado), la alineación, la fuente, el tamaño de la letra o el color del texto, entre otros. La lista completa de parámetros se puede consultar al pulsar las teclas “Ctrl + Barra espaciadora”, combinación que activará el autocompletado y mostrará todos los parámetros que acepta el *widget*, junto con su

documentación. A continuación, se muestra un ejemplo de cómo usar este *widget* en Flutter modificando varios de sus parámetros. Nótese que, en el parámetro `style`, el valor que se le pasa al parámetro es un *widget* `TextStyle` que contiene muchos más parámetros y es el encargado de, por ejemplo, cambiar el color o el tamaño de la fuente.

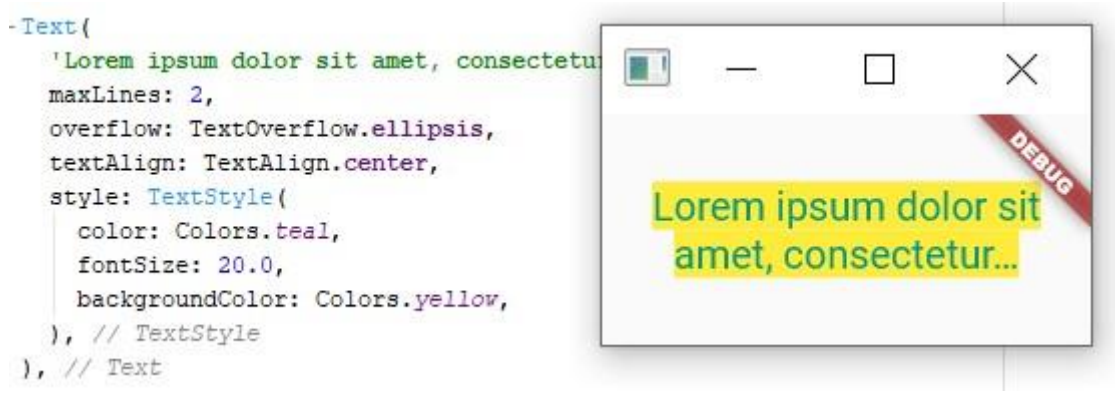

Figura 20: Ejemplo de uso del widget `Text` en Flutter

6.3. ImageView

En Android, el componente `ImageView` permite añadir una imagen incluida en el proyecto a la interfaz de la aplicación. En el caso de que se quiera mostrar una imagen almacenada en un servidor en internet, se necesitarán librerías de terceros como Glide[22] para poder mostrar la imagen.

Añadir imágenes es mucho más sencillo, especialmente imágenes que se encuentran en un servidor en internet. Para ello, existe el *widget* `Image`, en el que se especificará en su parámetro `image` la imagen que se desea añadir. Por ejemplo, si se desea incluir una imagen presente en el proyecto (que debe ser añadida previamente al archivo "`pubspec.yaml`"), el valor que se le pasará al parámetro es un *widget* `AssetImage`, cuyo primer parámetro será una cadena de texto con la ruta donde se encuentre la imagen. Si por el contrario se quiere añadir una imagen disponible en internet, el valor que se le pasará al parámetro `image` del *widget* `Image` deberá ser un *widget* `NetworkImage`, cuyo primero parámetro será una cadena de texto con el enlace donde se encuentra alojada la imagen.

También existen constructores que permiten realizar esta función de una forma mucho más rápida, como por ejemplo `Image.asset()` o `Image.network()`, a los que se les pasará como primer parámetro la ruta de la imagen (dependiendo de si es una imagen almacenada en el proyecto o en internet, respectivamente).

Las imágenes que se incluyan en la aplicación desarrollada con Flutter también pueden ser personalizadas gracias a los distintos parámetros que contiene el *widget* `Image`, como la altura, el ancho, el ajuste o pintar la imagen de un color específico.

[22] https://bumptech.github.io/glide/

Existe además un *widget* llamado `FadeInImage` que es especialmente útil cuando las imágenes que se desean cargar son muy grandes. Este *widget* permite mostrar una imagen mientras se carga la imagen principal (por ejemplo, se puede mostrar una imagen de "Cargando…") haciendo uso del parámetro `placeholder`, mientras que la imagen se pasará como valor al parámetro `image` haciendo uso de los *widgets* comentados anteriormente. Cuando la imagen principal se cargue completamente, reemplazará a la imagen de carga realizando una suave animación de desvanecimiento.

Figura 21: Ejemplo de uso del widget `Image` en Flutter

6.4. ImageButton

El componente `ImageButton` en Android sirve para añadir botones de acción más personalizados, mostrándolos como una imagen que puede ser pulsada para realizar una determinada acción.

En Flutter hay varias formas de conseguir un resultado similar, aunque todas ellas pasan por utilizar un *widget* `GestureDetector`, al que se le pasará como hijo mediante el parámetro `child` el *widget* al que se le quiere añadir una determinada interacción. Esto permite mucha más versatilidad dado que cualquier *widget* que se cree podrá ser utilizado como botón, incluyendo en este caso el *widget* `Image`. Al *widget* detector de gestos (GestureDetector), además del *widget* que se le pasará como hijo, se le podrán añadir muchos tipos distintos de gestos para añadir interacción. Como en este caso se tratará de un botón, el parámetro donde se deberá incluir la lógica es `onTap`.

Si se desea añadir un efecto visual similar al que se puede encontrar al pulsar en un botón cualquiera que siga las guías de estilo de Material Design, basta con reemplazar el *widget* `GestureDetector` por un `InkWell`.

```
InkWell(
└─child: Image.network(
    'https://www.inf.upv.es/www/etsinf/wp-content/uploads/2017/02/ETSInf_PRINCIPAL_V-horizontal.png',
    color: Colors.pink,
  ), // Image.network
  onTap: () => print('ImageButton pulsado'),
), // InkWell
```

Figura 22: Ejemplo de uso del widget InkWell en Flutter

6.5. Button

El botón es uno de los componentes más importantes dado que permite la interacción y navegación dentro de la aplicación. Desde enviar un formulario hasta completar una compra, una aplicación puede necesitar muchos tipos distintos de botones dependiendo del contexto en el que se vaya a utilizar.

Flutter posee varios botones predefinidos, y en caso de que ninguno satisfaga el resultado visual esperado, se podrá crear un *widget* desde cero y añadirlo como hijo de un `GestureDetector` o `InkWell`.

Los tres tipos principales de botones funcionan de forma similar, varían su estilo y, por tanto, el contexto en el que deberán ser utilizados siguiendo las guías de estilo de Material Design. Todos ellos son implementaciones más específicas del *widget* `MaterialButton`, el cuál puede ser usado como punto de partida cuando se desee personalizar un botón. Los tres tipos de botones predefinidos en Flutter en la librería Material son `FlatButton`, `OutlineButton` y `RaisedButton`.

```
FlatButton(
└─child: Text('FlatButton'),
  onPressed: () => null,
), // FlatButton
OutlineButton(
└─child: Text('OutlineButton'),
  onPressed: () => null,
), // OutlineButton
RaisedButton(
└─child: Text('RaisedButton'),
  onPressed: () => null,
), // RaisedButton
```

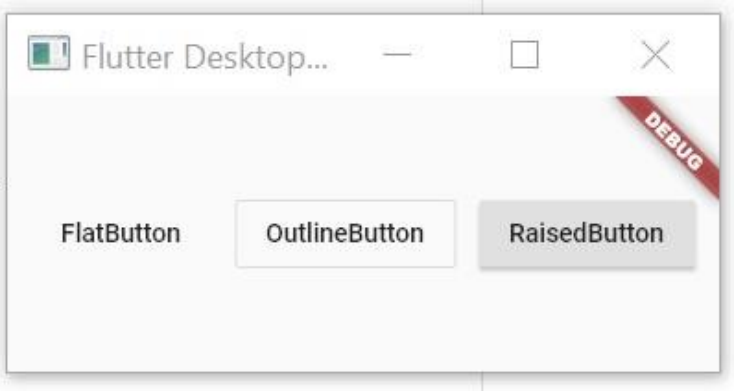

Figura 23: Ejemplo de uso de botones en Flutter

Además de estos botones, también existen dos tipos de botones predefinidos diseñados para la navegación entre vistas, puesto que ambos vuelven a la vista anterior al ser pulsados. El primero de ellos, `BackButton`, mostrará una flecha hacia atrás que se renderizará de forma distinta dependiendo de la plataforma en la que se

esté ejecutando la aplicación (Android o iOS). El segundo, `CloseButton`, actúa de forma similar, pero en este caso el icono mostrado es una cruz de cerrar.

También existe la clase `IconButton`, que permite añadir un botón que se muestra como un icono que puede ser personalizado.

6.6. RecyclerView

Cuando se desea mostrar una lista de elementos en una aplicación nativa en Android, lo más común es implementar un `RecyclerView`. Este elemento permite crear una lista optimizada con elementos específicos, aunque su implementación es algo tediosa.

En Flutter, se puede conseguir el mismo resultado con mucho menos esfuerzo y líneas de código gracias al *widget* `ListView`. Este *widget* puede ser utilizado de tres formas distintas dependiendo del resultado que se quiera obtener: utilizar su propio constructor para crear una lista de *widgets* que se pasan directamente como hijos, utilizando el constructor `builder()`, que permite crear una lista de elementos bajo demanda (por ejemplo, para crear un *widget* para cada elemento de una lista de objetos); o utilizando el constructor `separated()`, que actúa de forma similar al constructor `builder()`, pero además permite insertar *widgets* entre cada uno de los elementos de la lista (por ejemplo, una línea divisoria o un anuncio cada cierta cantidad de elementos).

Además, modificando sus parámetros, se puede modificar la dirección en la que se podrá deslizar para explorar la lista (vertical u horizontal), invertir el orden en el que se muestran los elementos o cambiar las físicas en los extremos de la lista (por ejemplo, en Android, al llegar al final de la lista se muestra una onda, mientras que en iOS los elementos crean una animación de rebote), entre otros parámetros.

En el caso de que se aniden distintos tipos de listas (por ejemplo, un `ListView` dentro de un *widget* `Column`), puede ser necesario añadir al ListView el parámetro `shrinkWrap` con un valor de "`true`".

```
ListView.separated(
  scrollDirection: Axis.vertical,
  itemCount: vocales.length,
  itemBuilder: (context, i) {
    return Text(
      vocales[i],
      textAlign: TextAlign.center,
    ); // Text
  },
  separatorBuilder: (context, i) => Divider(),
), // ListView.separated
```

Figura 24: Ejemplo de uso del widget `ListView` en Flutter

6.7. WebView

El uso del componente `WebView` es muy importante en el caso de que se quiera mostrar al usuario una página web sin salir de la propia aplicación.

Flutter no posee un *widget* de forma nativa que permita implementar la funcionalidad del WebView, pero existe la librería `webview_flutter`, desarrollada por Google, que, aunque actualmente se encuentra en fase de desarrollo (*Developer Preview*), permite una funcionalidad correcta de este componente. Para poder utilizar esta librería, se deberá añadir la dependencia en el archivo "`pubspec.yaml`" y posteriormente importarla dentro del proyecto.

Una vez se importe la librería, se podrá utilizar el *widget* WebView donde se desee de una forma muy simple, añadiendo al parámetro `initialUrl` la cadena de texto con el enlace que se quiera cargar. Además, se puede añadir una función que se ejecutará cuando la página web termine de cargarse mediante el parámetro `onPageFinished`, o desactivar o restringir el uso de JavaScript en la página web que se vaya a cargar haciendo uso del parámetro `javascriptMode`.

```
WebView(
  initialUrl: 'https://www.inf.upv.es/www/etsinf/es/',
), // WebView
```

Figura 25: Ejemplo de uso del widget WebView

6.8. SeekBar

En determinados casos de uso, el componente `SeekBar` es clave para la interacción del usuario con la aplicación, como puede ser la barra de progreso de un vídeo o de una canción.

En Flutter, el equivalente de este componente es un *widget* llamado `Slider`, que además posee un constructor `adaptive()`, el cuál renderizará un `Slider` de la librería Material si la aplicación se ejecuta en Android, y un `Slider` de la librería Cupertino si se ejecuta en iOS.

Para implementar correctamente este *widget*, se deberá hacer dentro de un `StatefulWidget`, dado que, para actualizar el valor seleccionado, se deberá actualizar el estado del *widget*. En este *widget*, los parámetros más importantes son: `min`, `max`, `value` y `onChanged`. Los dos primeros serán los encargados de especificar el rango de valores que tendrá el Slider, mientras que el parámetro `value` contendrá el valor seleccionado. Para actualizar el valor al deslizar, en el parámetro `onChanged`, se deberá actualizar el estado para reflejar correctamente el cambio en el valor, como se muestra a continuación.

```dart
double _value = 0.5;

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Slider.adaptive(
      min: 0,
      max: 1,
      value: _value,
      onChanged: (value) {
        setState(() {
          _value = value;
        });
      },
    ), // Slider.adaptive
  ); // Scaffold
}
```

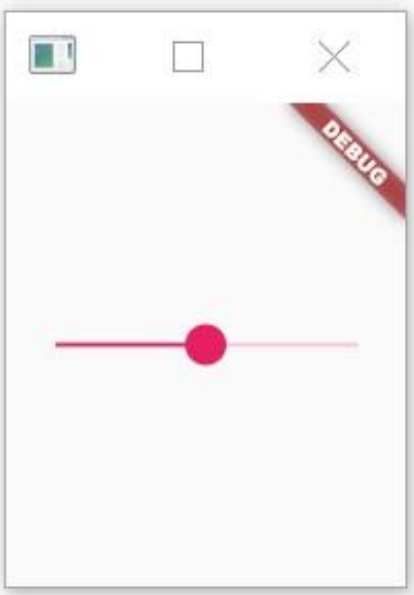

Figura 26: Ejemplo de uso del widget `Slider` en Flutter

6.9. Switch

Uno de los componentes más comunes, especialmente en los ajustes de una aplicación en Android, es el uso del componente `Switch`. Gracias a este componente, se pueden activar y desactivar opciones de una forma rápida y clara.

Este componente tiene una implementación similar a la del *widget* Slider, e incluye también un constructor `adaptive()` con el que se podrá renderizar un `Switch` distinto dependiendo de la plataforma donde se ejecute la aplicación. En este caso, los parámetros principales serán `value`, que contendrá el valor actual del `Switch` (activado o desactivado); y `onChanged`, en el que se deberá modificar el valor actual del *widget*.

```
bool _value = true;

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Switch(
          value: _value,
          onChanged: (value) {
            setState(() {
              _value = value;
            });
          },
        ), // Switch
      ), // Center
  ); // Scaffold
}
```

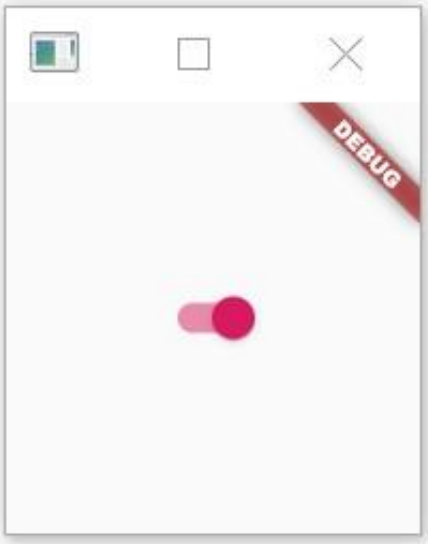

Figura 27: Ejemplo de uso del widget `Switch` en Flutter

6.10. Chip

El componente `Chip` en Android es utilizado en diversos contextos, como puede ser un filtro de búsqueda o representación de los atributos de un elemento. Esto significa que puede existir un componente `Chip` pasivo, que simplemente muestre información, o un `Chip` activo, que, al pulsarlo, abra una vista mostrando artículos con ese atributo mostrado; o bien active o desactive un filtro de búsqueda.

Para incorporar este componente de Android en la aplicación en Flutter, dado que existen distintos tipos de Chip según su función, se debe añadir un *widget* `Chip`, cuyo parámetro principal será `label`, al que se debe pasar una cadena de texto. Opcionalmente, se pueden especificar también algunos parámetros como por ejemplo `avatar`, con el que añadir algo más de información a este *widget*. De esta forma, se puede crear un `Chip` pasivo, dado que no se ha añadido ninguna interacción. Si se quiere convertir este *widget* en un botón, basta con reemplazarlo por un `ActionChip`, `ChoiceChip`, `FilterChip` o `InputChip`, dependiendo del contexto y la función que se desee realizar.

```
Chip(
  label: Text('Hola mundo'),
  avatar: CircleAvatar(
    child: Text('Z'),
  ), // CircleAvatar
), // Chip
```

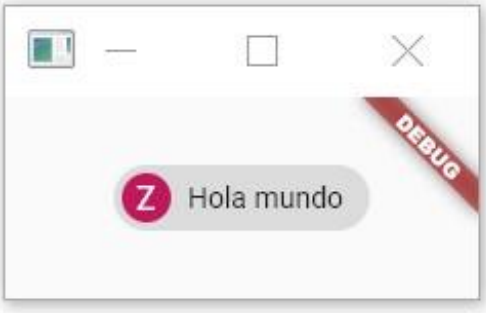

Figura 28: Ejemplo de uso del widget `Chip` en Flutter

6.11. CheckBox

Un `CheckBox` puede cumplir una función similar a un `Switch` en determinados contextos, aunque su uso más común sea el de seleccionar elementos de una lista.

A efectos prácticos, su implementación es prácticamente igual a la que se puede observar en la implementación del *widget* `Switch` de la Figura 29, pero sustituyendo el *widget* `Switch` por el *widget* `CheckBox`. El parámetro `value` contiene el valor actual del `CheckBox`, que, en el caso de que se especifique el parámetro `tristate` con un valor de "`true`", podrá ser nulo, además de "`true`" o "`false`". En el parámetro `onChanged` se deberá actualizar el valor para renderizar correctamente el estado del `CheckBox`.

```
bool _value = true;

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Checkbox(
        value: _value,
        onChanged: (value) {
          setState(() {
            _value = value;
          });
        },
      ), // Checkbox
    ), // Center
  ); // Scaffold
}
```

Figura 29: Ejemplo de uso del widget `CheckBox` en Flutter

6.12. RadioButton

En el caso del componente `RadioButton`, su uso principal es el de elegir uno de los elementos de una lista de forma excluyente. Puede haber un elemento preseleccionado en la lista, pero al seleccionar un elemento distinto, debe cambiar la selección de uno a otro.

La implementación de este componente en Flutter se realiza mediante el *widget* `Radio`, cuyos parámetros principales son: `groupValue`, cuyo valor será una variable que almacene el valor del `Radio` seleccionado; `value`, cuyo valor se podrá elegir de forma arbitraria siempre y cuando sea un número entero que identifique a ese *widget*; y `onChanged`, que deberá actualizar el valor de la variable asignada en el parámetro `groupValue` al valor del *widget* seleccionado.

```
int _value = -1;

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          Radio(
            groupValue: _value,
            value: 0,
            onChanged: (value) {
              setState(() {
                _value = value;
              });
            },
          ), // Radio
          Radio(
            groupValue: _value,
            value: 1,
            onChanged: (value) {
              setState(() {
                _value = value;
              });
            },
          ), // Radio
        ], // <Widget>[]
      ), // Column
    ), // Center
  ); // Scaffold
}
```

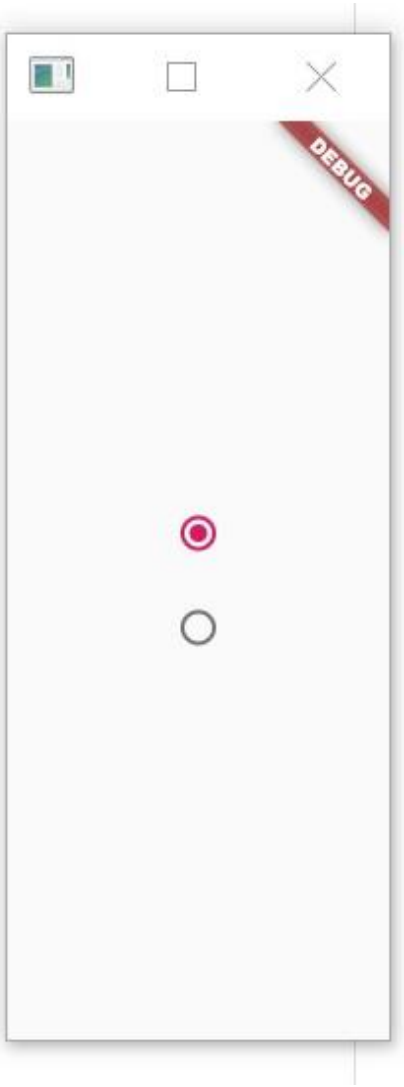

Figura 30: Ejemplo de uso del widget `Radio` en Flutter

6.13. FloatingActionButton

El componente `FloatingActionButton` es uno de los elementos más característicos de las aplicaciones Android. Su función es presentar una acción principal contextualizada a una vista concreta.

En Flutter, este *widget* se puede implementar de forma sencilla gracias al parámetro `floatingActionButton` del *widget* `Scaffold`, al que se le pasará como valor el *widget* `FloatingActionButton`. Además, se puede especificar su posición dependiendo cómo se quiera posicionar en una vista concreta gracias al parámetro `floatingActionButtonLocation` del *widget* `Scaffold`. Los parámetros más importantes de este *widget* son: `child`, al que se le pasará un *widget* `Icon` con el icono a mostrar dependiendo de la acción principal; y `onPressed`, en el que se implementará la acción a realizar cuando se pulse el botón.

También existe la posibilidad de utilizar el constructor `extended()`, en cuyo caso el parámetro `child` será reemplazado por otros dos parámetros: `label`, que

contendrá el texto correspondiente a la acción; y `icon`, al que se le pasará como parámetro un *widget* `Icon` con el icono correspondiente a la acción del botón.

Además, en Flutter, si el usuario navega entre dos vistas que contienen un `FloatingActionButton`, el botón realizará una animación de transición de color, posición y tamaño de la vista inicial a la vista destino sin necesidad de programar la animación explícitamente.

```
return Scaffold(
  floatingActionButton: FloatingActionButton.extended(
    label: Text('Iniciar chat'),
    icon: Icon(Icons.chat_bubble),
    onPressed: () => null,
  ), // FloatingActionButton.extended
  floatingActionButtonLocation: FloatingActionButtonLocation.endFloat,
```

Figura 31: Ejemplo de uso del widget `FloatingActionButton` en Flutter

6.14. ProgressBar

Durante la carga de datos en las aplicaciones, es muy recomendable añadir un indicador de progreso, determinado o indeterminado, para mostrar al usuario que se está cargando algo. Cuando se sabe el progreso total de la carga, se puede mostrar un indicador de progreso determinado con este valor, mientras que, si el progreso total es desconocido, se puede hacer uso de la versión indeterminada del indicador de progreso.

En Flutter existen dos tipos de indicadores de progreso: `CircularProgressIndicator` y `LinearProgressIndicator`. Dependiendo del contexto en el que se vaya a utilizar, será más recomendable uno u otro, dado que ambos pueden ser tanto determinados como indeterminados. Por defecto, si no se especifica un valor en el parámetro `value`, se mostrará como un indicador de progreso indeterminado. Además, se pueden modificar los parámetros `valueColor` y `backgroundColor` para dar el estilo deseado a estos *widgets*.

```
CircularProgressIndicator(
  valueColor: AlwaysStoppedAnimation(Colors.green),
), // CircularProgressIndicator
LinearProgressIndicator(),
```

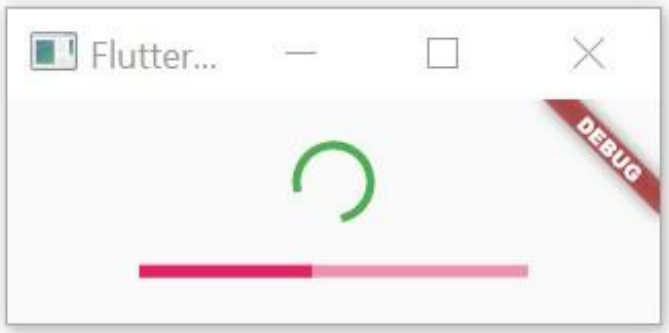

Figura 32: Ejemplo de uso de indicadores de progreso en Flutter

6.15. Toolbar

Uno de los elementos más importantes para la navegación en Android es el componente `Toolbar`, en el que se muestra, en caso de ser necesario, un botón para volver a la vista anterior; el nombre de la vista y, opcionalmente, algunas acciones o menú.

En Flutter, gracias a los parámetros `appBar` y `bottomNavigationBar` del *widget* `Scaffold`, se puede añadir esta barra en la parte superior o inferior de la aplicación, haciendo uso de los *widgets* `AppBar` y `BottomAppBar`, respectivamente. Estos *widgets* poseen distintos parámetros, aunque la composición visual que se puede conseguir es similar en ambos. Elegir uno u otro dependerá del tipo de aplicación que se esté desarrollando.

En el *widget* `AppBar`, por ejemplo, se pueden especificar parámetros para el título, acciones contextuales de la vista o una acción principal (normalmente, cerrar la vista o volver atrás). Por otra parte, en el *widget* `BottomAppBar`, solo se permite especificar un *widget* hijo que, dependiendo de las acciones que se quieran incluir y su organización, se podrá implementar de una forma u otra. Si solo se quiere una acción a la izquierda, bastaría con incluir directamente un `IconButton` como hijo del *widget* `BottomAppBar`. Si por el contrario se quieren agregar acciones a ambos lados, se puede optar por incluir un *widget* `Row` (fila) que, como se muestra en la Figura 33, separa los hijos que se le añadan, por lo que quedan repartidos a lo largo del *widget* `BottomAppBar`.

```
Scaffold(
  appBar: AppBar(
    leading: BackButton(),
    title: Text('Título'),
    actions: <Widget>[
      IconButton(
        icon: Icon(Icons.add),
        onPressed: () => null,
      ), // IconButton
      IconButton(
        icon: Icon(Icons.search),
        onPressed: () => null,
      ), // IconButton
    ], // <Widget>[]
  ), // AppBar
  bottomNavigationBar: BottomAppBar(
    child: Row(
      mainAxisAlignment: MainAxisAlignment.spaceBetween,
      children: <Widget>[
        IconButton(
          icon: Icon(Icons.menu),
          onPressed: () => null,
        ), // IconButton
        IconButton(
          icon: Icon(Icons.more_vert),
          onPressed: () => null,
        ), // IconButton
      ], // <Widget>[]
    ), // Row
  ), // BottomAppBar
); // Scaffold
```

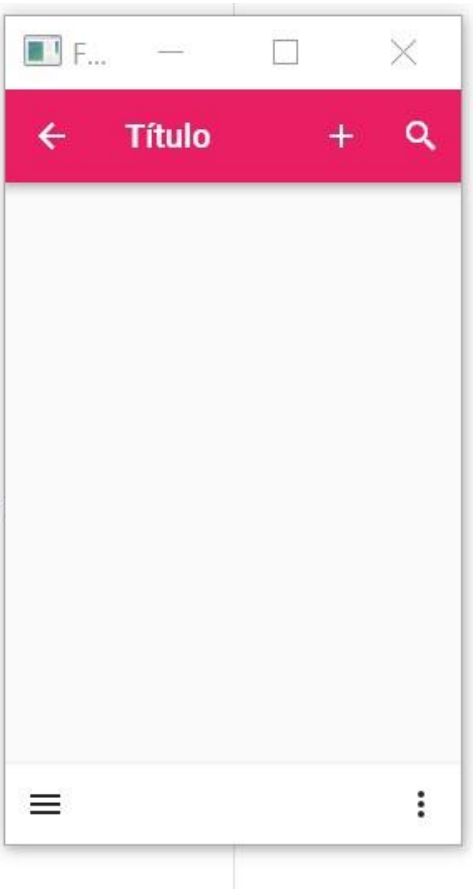

Figura 33: Ejemplo de uso de los widgets `AppBar` y `BottomAppBar` en Flutter

6.16. SnackBar

El componente `SnackBar` es muy común en las aplicaciones Android dado que sirve para informar brevemente al usuario de una acción realizada. Se muestran en la parte inferior de la pantalla y puede mostrarse flotando sobre otros elementos y puede contener una acción relacionada con el mensaje.

Cuando se quiera implementar el *widget* SnackBar en Flutter, hay dos formas de hacerlo dependiendo de dónde se quiera añadir. Si se va a añadir el *widget* en el mismo método `build()` en el que se compone el *widget* `Scaffold`, se deberá rodear la creación del `SnackBar` con un *widget* `Builder`. En caso contrario, se producirá un error y no se mostrará. Por el contrario, si se va a mostrar en un lugar distinto al que construye el `Scaffold` de la vista, se podrá mostrar sin necesidad de un `Builder` dado que se podrá acceder al `Scaffold` del contexto en el que se mostrará el `SnackBar`.

En el parámetro `content` se debe añadir un *widget* `Text` para mostrar el mensaje. De forma adicional, se puede añadir una acción relacionada con el mensaje añadiendo el parámetro `action`, cuyo valor deberá ser un *widget* `SnackBarAction`, cuyos parámetros `label` y `onPressed` sirven para mostrar el texto de la acción y la acción a realizar respectivamente. El *widget* `SnackBar` también posee parámetros útiles como `backgroundColor`, que permite cambiar el color; `behavior`, que permite establecer

si el SnackBar será flotante o no; o `duration`, que permite definir el tiempo durante el que se mostrará el mensaje.

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Builder(
        builder: (builderContext) => RaisedButton(
          child: Text('Mostrar SnackBar'),
          onPressed: () {
            Scaffold.of(builderContext).showSnackBar(
              SnackBar(
                content: Text('Texto de ejemplo'),
                action: SnackBarAction(
                  label: 'ACCIÓN',
                  onPressed: () => null,
                ), // SnackBarAction
                behavior: SnackBarBehavior.floating,
              ), // SnackBar
            );
          },
        ), // RaisedButton
      ), // Builder
    ), // Center
  ); // Scaffold
}
```

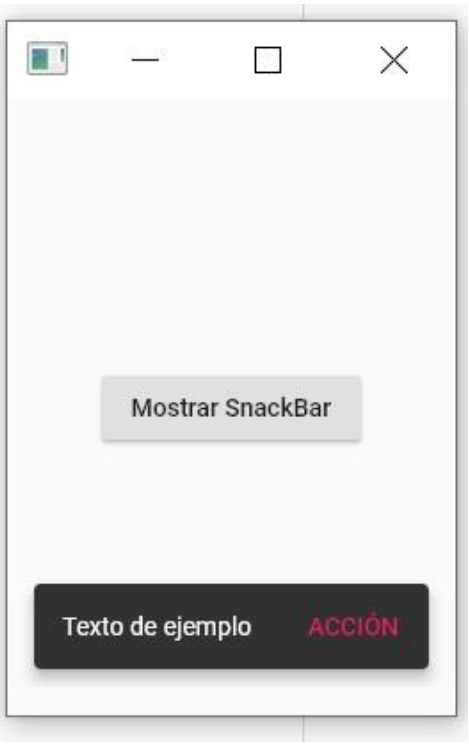

Figura 34: Ejemplo de uso del widget `SnackBar` en Flutter

6.17. BottomNavigationView

Uno de los modelos de navegación más comunes en las aplicaciones nativas en Android es mediante el uso de componentes `Fragment`, que permiten, mediante el uso de una barra de navegación en la parte inferior de la aplicación, cambiar entre los distintos `Fragment` que existan.

Para migrar este tipo de navegación, es imprescindible que la vista posea un `Scaffold` y sea un `StatefulWidget`. Para añadir esta navegación se hará uso del *widget* `BottomNavigationBar`, que se pasará como valor al parámetro `bottomNavigationBar` del `Scaffold`. Este *widget* posee dos parámetros principales: `items`, en el que se incluirá la lista de elementos que tendrá la barra de navegación; y `onTap`, donde se deberá gestionar el cambio entre las distintas vistas.

Para gestionar el cambio de vista, se deben crear un mínimo de dos elementos de navegación y componer dos vistas distintas, que serán las vistas equivalentes a los `Fragment` de la aplicación Android. Dado que el equivalente de los Fragment en Flutter será una composición cualquiera de *widgets*, cuyo *widget* padre será pasado como valor al parámetro `body` del `Scaffold`, en el parámetro `onTap` se deberá actualizar el estado para, dependiendo del elemento pulsado en la barra de navegación, cargar un *widget* u otro en el parámetro `body` del `Scaffold`.

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    bottomNavigationBar: BottomNavigationBar(
      onTap: (index) => print('Pulsado: $index'),
      items: <BottomNavigationBarItem>[
        BottomNavigationBarItem(
          icon: Icon(Icons.home),
          title: Text('Inicio'),
        ), // BottomNavigationBarItem
        BottomNavigationBarItem(
          icon: Icon(Icons.search),
          title: Text('Buscar'),
        ), // BottomNavigationBarItem
        BottomNavigationBarItem(
          icon: Icon(Icons.person),
          title: Text('Perfil'),
        ), // BottomNavigationBarItem
      ], // <BottomNavigationBarItem>[]
    ), // BottomNavigationBar
  ); // Scaffold
}
```

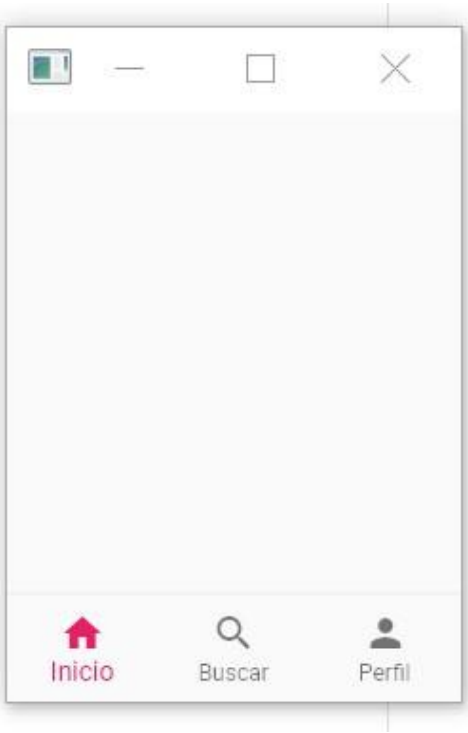

Figura 35: Ejemplo de uso del widget `BottomNavigationView` en Flutter

6.18. ViewPager

Otra forma de navegación a través de componentes `Fragment` en Android es mediante el uso de un `ViewPager`, que permite navegar entre los distintos componentes `Fragment` mediante uso de gestos de deslizar.

En Flutter, el *widget* `PageView` permite conseguir el mismo efecto que en Android. Si se pasa un *widget* `PageView` como valor al parámetro `body` de un `Scaffold`, ocupará el total de la pantalla del dispositivo. El parámetro más relevante de este *widget* es `children`, al que se le pasará una lista de *widgets* compuestos, en el que cada *widget* será el equivalente a un `Fragment` en Android.

Además, se podrán especificar otros parámetros como `scrollDirection`, para especificar si la navegación se realizará en horizontal o en vertical; o `onPageChanged`, que ejecutará una función especificada cada vez que se navegue de un *widget* a otro.

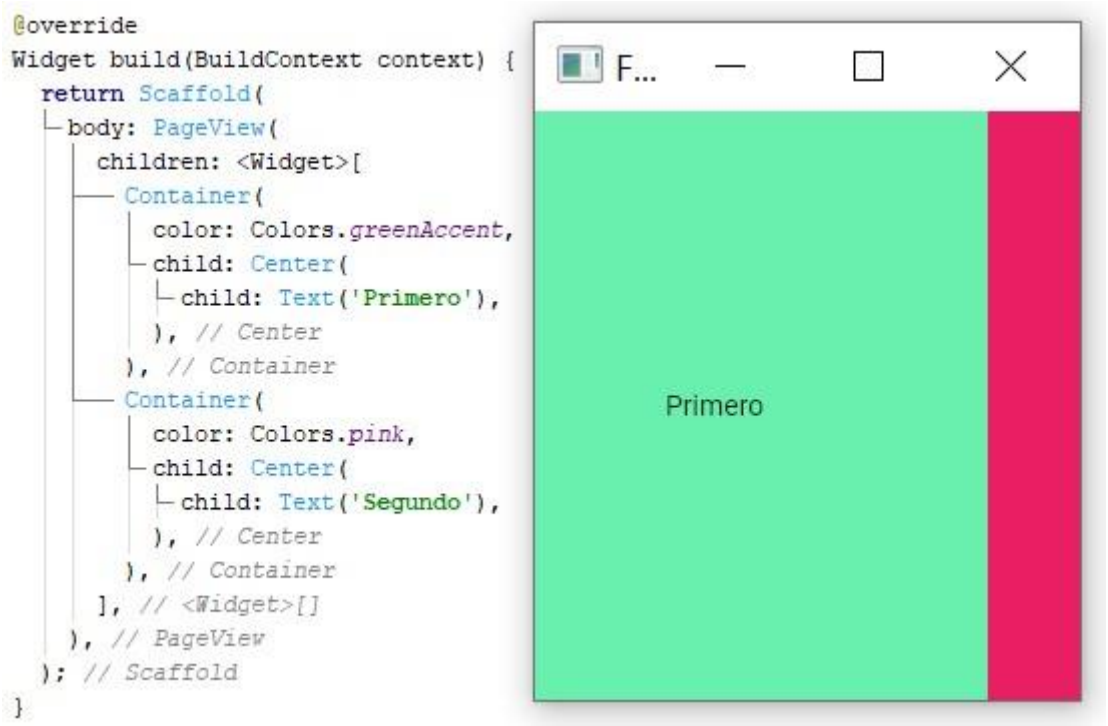

Figura 36: Ejemplo de uso del widget `PageView` en Flutter

6.19. DrawerLayout

El menú lateral es un componente en Android muy común para gestionar la navegación entre las distintas vistas de una aplicación. Basta con deslizar desde el lateral de la pantalla para arrastrar este menú y mostrarlo sobre el resto de los componentes visuales de la aplicación.

Para incorporar este menú en Flutter, bastará con añadir un *widget* `Drawer` al parámetro `drawer` del *widget* `Scaffold`. De esta forma, al deslizar desde el lateral izquierdo de la aplicación, aparecerá este menú que se podrá componer con los *widgets* que se desee haciendo uso del parámetro `child` del *widget* `Drawer`.

Además, en caso de que se quiera añadir también un menú similar, pero deslizando desde el lateral derecho de la aplicación, basta con añadir el *widget* `Drawer` al parámetro `endDrawer` del `Scaffold`.

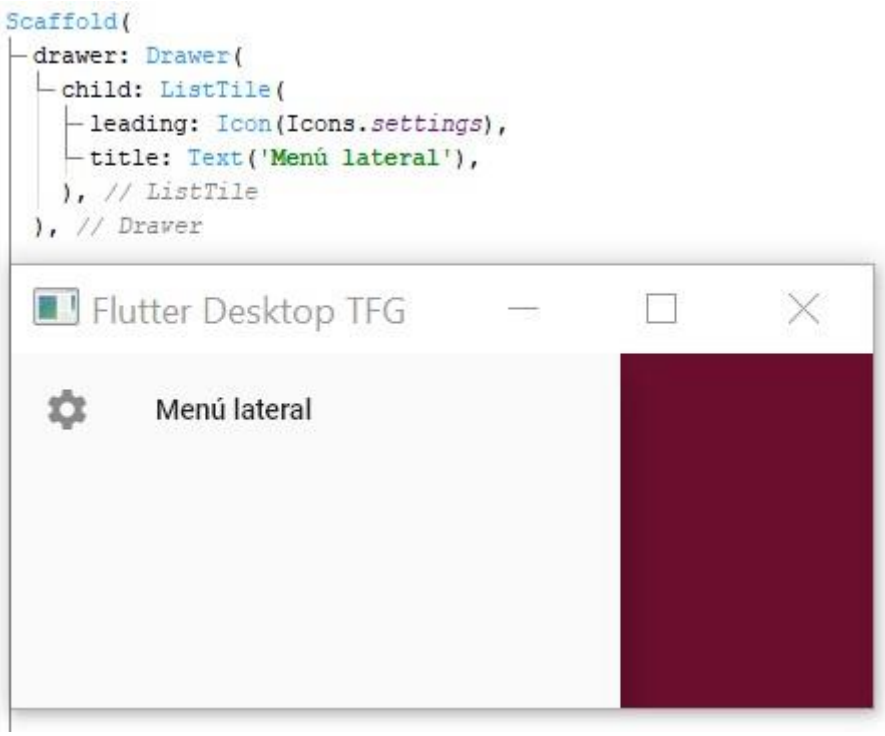

Figura 37: Ejemplo de uso del widget `Drawer` en Flutter

6.20. LinearLayout

El componente `LinearLayout` es muy útil cuando se desea componer una interfaz que contiene una sucesión vertical u horizontal de elementos. De esta forma, los distintos componentes que se añadan dentro de este componente se mostrarán ordenados de forma vertical u horizontal.

En Flutter hay dos *widgets* distintos dependiendo de la orientación, vertical u horizontal, en la que se quieran mostrar los *widgets*.

Si se quieren mostrar de forma vertical, el *widget* a utilizar será un `Column`, que recibirá los distintos *widgets* que se quieran mostrar siguiendo un orden vertical, simulando una columna, como valor en el parámetro `children`.

De forma similar, si lo que se quiere es mostrar una sucesión de *widgets* en una fila horizontal, se deberá utilizar el *widget* `Row`.

Ambos *widgets* permiten alinear los elementos en sus dos ejes mediante los parámetros `mainAxisAlignment` y `crossAxisAlignment`. El primero se encargará de la alineación del eje principal (eje vertical en la columna, eje horizontal en la fila) mientras que el segundo se encargará de la alineación del eje transversal (eje horizontal en la columna, eje vertical en la fila). Gracias al uso de estos dos parámetros se puede, por ejemplo, centrar los elementos o añadir un hueco entre ellos.

6.21. ConstraintLayout

Cuando se desea componer una vista, el componente `ConstraintLayout` es uno de los más populares en el ecosistema Android, dado que permite componer la interfaz añadiendo restricciones entre los distintos componentes visuales de la interfaz.

Lamentablemente, en Flutter no existe un *widget* que sea capaz de componer una interfaz de la misma forma que lo hace un `ConstraintLayout`. No obstante, mediante la composición de distintos *widgets* como Column, Row, Stack o Flexible, se puede obtener un resultado visual similar al que se podría obtener en Android mediante el uso del componente `ConstraintLayout`.

6.22. TabLayout

El uso de pestañas para gestionar la navegación Android es una forma simple y visual de gestionar la navegación dentro de una aplicación en Android. Esta navegación es posible tanto deslizando horizontalmente para navegar entre las distintas pestañas como pulsando en las propias pestañas, lo que llevará al usuario a la vista de la pestaña seleccionada.

Para incorporar este tipo de navegación en una aplicación Flutter, el primer paso será pasar el *widget* `Scaffold` de la vista a la que se quiera añadir la navegación por pestañas como valor al parámetro de un *widget* `DefaultTabController`. A este *widget* será necesario también añadir el parámetro `length`, cuyo valor será el número total de pestañas que se deseen añadir.

Para añadir las pestañas a la vista, se hará uso del *widget* `AppBar`. En este *widget*, haciendo uso del parámetro `bottom`, se añadirá un *widget* `TabBar` que se mostrará en la parte inferior del *widget* `AppBar`, debajo del título de la vista. Finalmente, se deberán añadir tantos *widgets* `Tab` en el parámetro `tabs` como se hayan indicado previamente en el parámetro `length` del *widget* `DefaultTabController`. Cada *widget* `Tab` deberá ir acompañado de un icono, un título o ambas opciones. El icono se añadirá pasando un *widget* `Icon` como valor al parámetro `icon` del *widget* `Tab`, mientras que para añadir un título se pasará un *widget* `Text` como valor al parámetro `child`.

Una vez se hayan añadido las pestañas a la barra superior de la vista, se deben crear los *widgets* que se mostrarán en cada una de las pestañas. Para ello, mediante el uso del *widget* `TabBarView`, que se podrá pasar como valor al parámetro `body` del *widget* `Scaffold`, se añadirán tantos *widgets* como pestañas se hayan especificado previamente. Cada *widget* que se pase como valor al parámetro `children` del *widget* `TabBarView` representará, por orden, una vista distinta de las pestañas especificadas.

```
DefaultTabController(
  length: 4,
  child: Scaffold(
    appBar: AppBar(
      title: Text('TabLayout'),
      bottom: TabBar(
        tabs: <Tab>[
          Tab(icon: Icon(Icons.home)),
          Tab(icon: Icon(Icons.shopping_cart)),
          Tab(icon: Icon(Icons.person)),
          Tab(icon: Icon(Icons.settings)),
        ], // <Tab>[]
      ), // TabBar
    ), // AppBar
    body: TabBarView(
      children: [
        Icon(Icons.home),
        Icon(Icons.shopping_cart),
        Icon(Icons.person),
        Icon(Icons.settings),
      ],
    ), // TabBarView
  ), // Scaffold
); // DefaultTabController
```

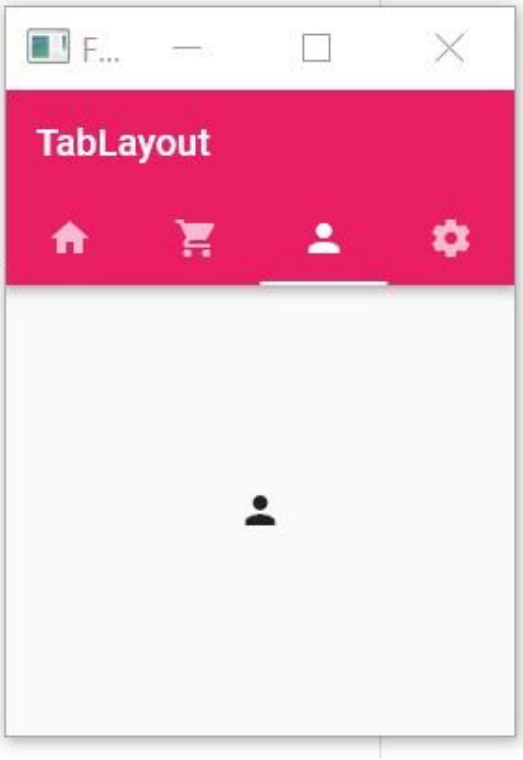

Figura 38: Ejemplo de uso de la navegación mediante pestañas en Flutter

6.23. Resumen

A continuación, se muestra una tabla resumen en la que se indicará cada patrón según su número de índice en este documento, el componente visual de Android a migrar, el o los *widgets* equivalentes en Flutter, y la categoría del patrón. Existen más componentes visuales más en Android cuyos patrones de migración no se han podido definir en este documento, dado que se han especificado los más comunes y populares.

Índice	Android	Flutter	Categoría
1	Activity	MaterialPageRoute	Navegación
2	TextView	Text	Visualización
3	ImageView	Image	Visualización
4	ImageButton	Image + InkWell, GestureDetector	Interacción
5	Button	FlatButton, OutlineButton, RaisedButton	Interacción
6	RecyclerView	ListView	Estructura
7	WebView	WebView (librería externa)	Visualización
8	SeekBar	Slider	Interacción
9	Switch	Switch	Interacción
10	Chip	Chip	Interacción, Visualización
11	CheckBox	CheckBox	Interacción
12	RadioButton	Radio	Interacción
13	FloatingActionButton	FloatingActionButton	Interacción
14	ProgressBar	CircularProgressIndicator, LinearProgressIndicator	Visualización
15	Toolbar	AppBar, BottomAppBar	Navegación
16	SnackBar	SnackBar	Interacción, Visualización
17	BottomNavigationView	BottomNavigationBar	Navegación
18	ViewPager	PageView	Navegación
19	DrawerLayout	Drawer	Navegación
20	LinearLayout	Column, Row	Estructura
21	ConstraintLayout	Ø	Estructura
22	TabLayout	TabBar	Navegación

Tabla 1: Resumen de patrones de migración de la interfaz

7. Migración de la lógica

La lógica de la aplicación nativa en Android será la parte más costosa de migrar, dado que se deberá reescribir en su mayoría. No obstante, se detallan a continuación algunos patrones de migración para algunos de los casos más comunes en las aplicaciones Android. La lógica a la que se hace referencia en este capítulo es a la presente dentro de la aplicación móvil (por ejemplo, validación de formularios o almacenamiento de información), la lógica correspondiente al servidor es completamente reutilizable.

7.1. Llamadas al servidor

Las llamadas al servidor para obtener o enviar datos son una funcionalidad presente en la mayoría de las aplicaciones, sean hechas para el ecosistema Android o no: obtener un listado de productos, guardar el perfil del usuario o procesar un pedido con su respectivo pago son algunas de las posibilidades que se pueden conseguir de esta forma.

Para conseguir esta funcionalidad en Android, se pueden utilizar librerías externas que simplifiquen la cantidad de código necesario para realizar peticiones a un servidor. Una de las más populares es Retrofit[23], que permite crear una interfaz en Java que posteriormente Retrofit convierte en una implementación de dicha interfaz.

También existe la posibilidad de realizar las peticiones implementando una clase que extienda la clase `AsyncTask`. De esta forma, la petición al servidor se realizará en otro hilo de ejecución y no bloqueará el hilo principal, encargado de renderizar la interfaz de usuario, por lo que se evitará que la aplicación quede congelada.

En Flutter, gracias a que el lenguaje de programación Dart permite la ejecución asíncrona de funciones de una forma muy simple, estas peticiones se pueden implementar de una forma rápida y sencilla gracias a la librería `http`[24]. Tras añadir la librería al fichero "`pubspec.yaml`" e importarla en el código (preferiblemente con un alias, como "http"), bastará con llamar a los métodos `get()`, `post()`, `put()` o `delete()`, dependiendo de la deseada. Estos métodos recibirán como primer parámetro el enlace del que se quieren obtener o enviar datos, pudiendo especificar además la cabecera de la petición en el parámetro `headers`. En el caso de que se quiera enviar información, podrá hacerse especificando su codificación en el parámetro `encoding` y la información a enviar se pasará como valor al parámetro `body`.

[23] https://square.github.io/retrofit/
[24] https://pub.dev/packages/http

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;

Future<Game> fetchGame(String id) async {
  final response = await http.get('$url/games/$id');
  if (response.statusCode == 200) {
    return Game.fromJson(json.decode(response.body));
  } else {
    return null;
  }
}
```

Figura 39: Ejemplo de petición GET en Dart

7.2. Formularios

Una función muy común a la hora de desarrollar una aplicación es el uso de formularios. Esta funcionalidad permite, por ejemplo, registrar un usuario o realizar un pedido. Por ello, es importante gestionar correctamente la validación de los distintos campos de los formularios que existan en la aplicación.

En Flutter, para crear un formulario que pueda ser validado de forma sencilla, se debe hacer uso del *widget* `Form`. Para poder acceder este formulario desde el código, se deberá crear una variable con un objeto `GlobalKey<FormState>`, que posteriormente se pasará como valor al parámetro `key` del *widget* `Form`. Después, y teniendo en cuenta que para validar un formulario será imprescindible que se trate de un `StatefulWidget`, se podrá acceder al estado del formulario para poder validarlo. Para ello, cuando se necesite validar el formulario antes de realizar la acción (por ejemplo, pulsando un botón ubicado al final de los campos de texto), se accederá a su estado mediante el parámetro `currentState`, desde el que se podrá acceder al método `validate()`, que ejecutará la validación de los distintos campos del formulario. Si al ejecutar dicho método se devuelve un valor "`true`", los campos del formulario son correctos y por tanto se puede realizar la acción del formulario.

Para realizar la validación individual de cada campo cuando se llame al método `validate()`, se deberá especificar la validación en el parámetro `validator` de cada *widget* `TextFormField` que requiera validación. Si existe un error en la validación, se devolverá como resultado una cadena de texto con el mensaje de error. Si la validación del campo es positiva, se deberá devolver "`null`", indicando así que no existe un error en ese campo.

```
validator: (value) {
  if (value.length != 9) {
    return 'El teléfono debe tener 9 dígitos';
  } else if (int.tryParse(value) != null) {
    return 'El teléfono introducido no es válido';
  } else {
    return null;
  }
},
```

Figura 40: Ejemplo de validación de un formulario en Flutter

7.3. Almacenamiento local

En Android existen dos formas principales de almacenar localmente información de la aplicación: mediante el uso de una base de datos local SQLite o mediante el uso de SharedPreferences. El primero de ellos está más enfocado a crear una base de datos local en la que almacenar, de forma relacional, información como usuarios o productos mientras que el segundo está más enfocado a almacenar valores como la configuración de la aplicación.

Ambas formas de almacenamiento local están disponibles para usar en Flutter mediante librerías externas que permiten obtener un funcionamiento equivalente tanto en Android como en iOS. Para utilizar la SQLite en Flutter, la librería más popular es `sqflite` [25], mientras que para el uso de SharedPreferences la librería es shared_`preferences`[26], desarrollada por Google. Para comenzar a usar cualquiera de estas librerías, basta con añadirlas al fichero "`pubspec.yaml`" e importarlas desde un archivo con extensión "`.dart`". Dado que ambas librerías se basan en la lectura y escritura de ficheros en el almacenamiento del dispositivo, sus funciones se basarán principalmente en el uso de objetos `Future` y funciones asíncronas.

```
Future<String> getUserNotificationsToken() async {
  final SharedPreferences prefs = await SharedPreferences.getInstance();
  return prefs.getString('tokenNotif') ?? null;
}
```

Figura 41: Ejemplo de almacenamiento local con SharedPreferences en Flutter

[25] https://pub.dev/packages/sqflite
[26] https://pub.dev/packages/shared_preferences

8. Caso de Estudio

Swapp[27] es una *startup* que, mediante una suscripción mensual, permite a los usuarios solicitar videojuegos para la PlayStation 4 en formato físico . Para solicitar los videojuegos, y dependiendo del tipo de suscripción de suscripción escogida, el usuario se debe apuntar a una cola virtual y, cuando llegue a la primera posición, podrá solicitar el videojuego. En caso de que el usuario ya tenga un videojuego en casa, cuando reciba el que ha pedido a través de la aplicación, debe enviar de vuelta el que ya tiene en casa.

8.1.1. Migración

En este apartado se expondrán los detalles relacionados con la migración de algunas de las vistas de la interfaz de la aplicación nativa en Android a la aplicación nativa multiplataforma en Flutter, en los que se mostrará el resultado de la migración de algunas de las vistas más importantes de la aplicación. Se podrá observar que las vistas no han sido migradas componente a componente, sino que aprovechando la gran capacidad de Flutter con respecto a la composición de interfaces de usuario, se han rediseñado la gran mayoría de las vistas de la aplicación. No se mostrarán detalles relativos a la migración de la lógica de la aplicación por motivos de confidencialidad.

En la Figura 42 se muestra la vista principal de la aplicación. En la aplicación en Android (izquierda) se muestra una imagen principal junto con tres listas horizontales de videojuegos, en las que únicamente se muestra la carátula. Por el contrario, en la aplicación en Flutter (derecha), además de la imagen principal, se muestra una lista horizontal con las distintas categorías de los videojuegos. Además, en las posteriores listas donde aparecen las próximas y últimas novedades junto con varias sagas, se puede observar, además de la carátula del videojuego, su título e información sobre su disponibilidad.

[27] https://lanzadera.es/proyecto/swapp/

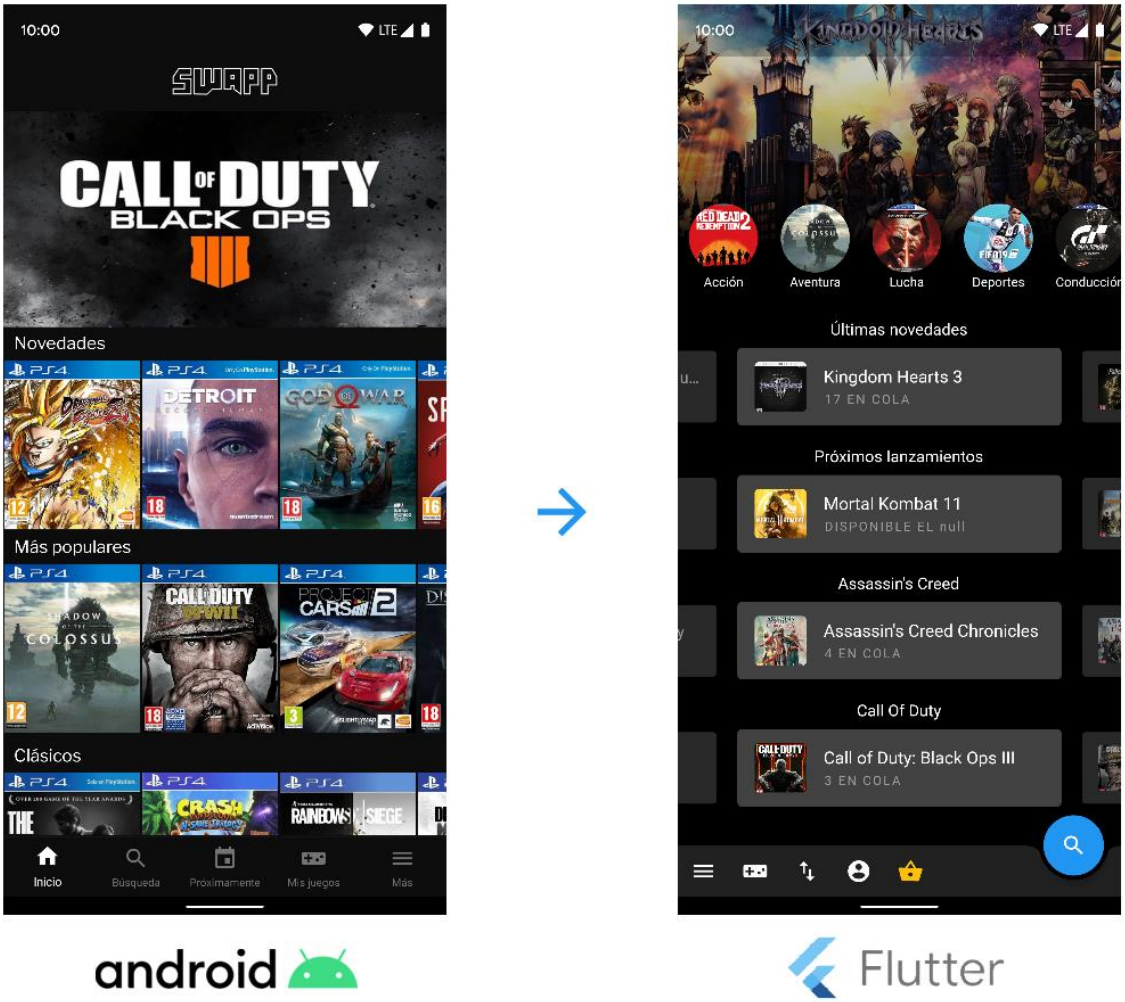

Figura 42: Migración de la vista principal de Android a Flutter

Otras de las vistas que han sido rediseñadas aprovechando la gran cantidad de *widgets* de Flutter para mejorar la interfaz de usuario es la vista de la lista de videojuegos a pedir por el usuario y la información de un pedido concreto.

En la vista de la lista de videojuegos a pedir, se incorporaron *widgets* de mayor tamaño en los que poder mostrar más información y reducir el espacio vacío de la pantalla, tal y como se puede observar en la Figura 43.

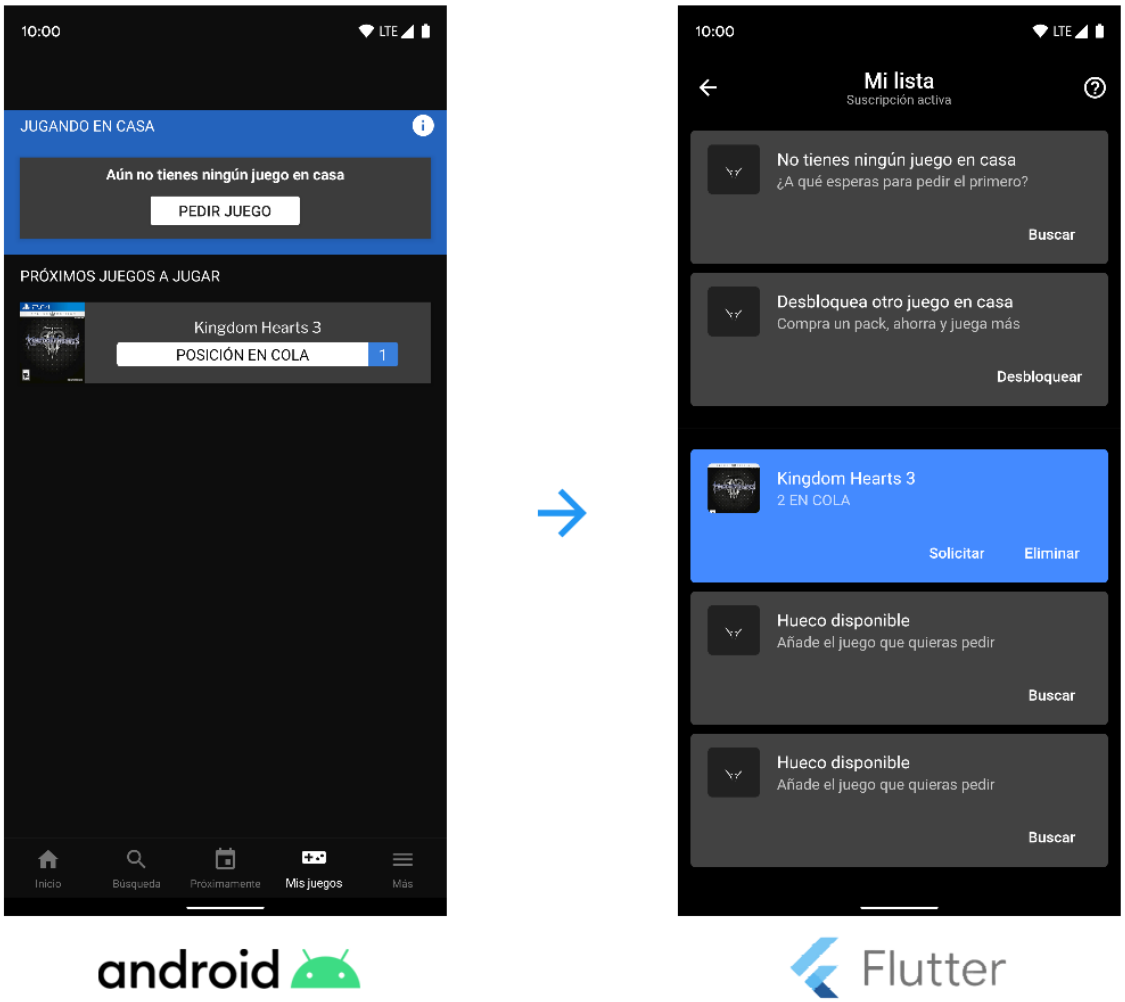

Figura 43: Migración de la vista de lista de videojuegos a pedir de Android a Flutter

Además, tras el rediseño, se incorporó una nueva vista a la aplicación al pulsar sobre uno de los pedidos de la lista de pedidos del usuario. En esta nueva vista, como se muestra en la Figura 44, el usuario puede obtener información acerca de los videojuegos que recibe y envía, así como un listado detallado de todos los estados por los que puede pasar su pedido, mostrando en cuál de ellos se encuentra y añadiendo además la posibilidad de editar o cancelar el pedido antes de que sea enviado.

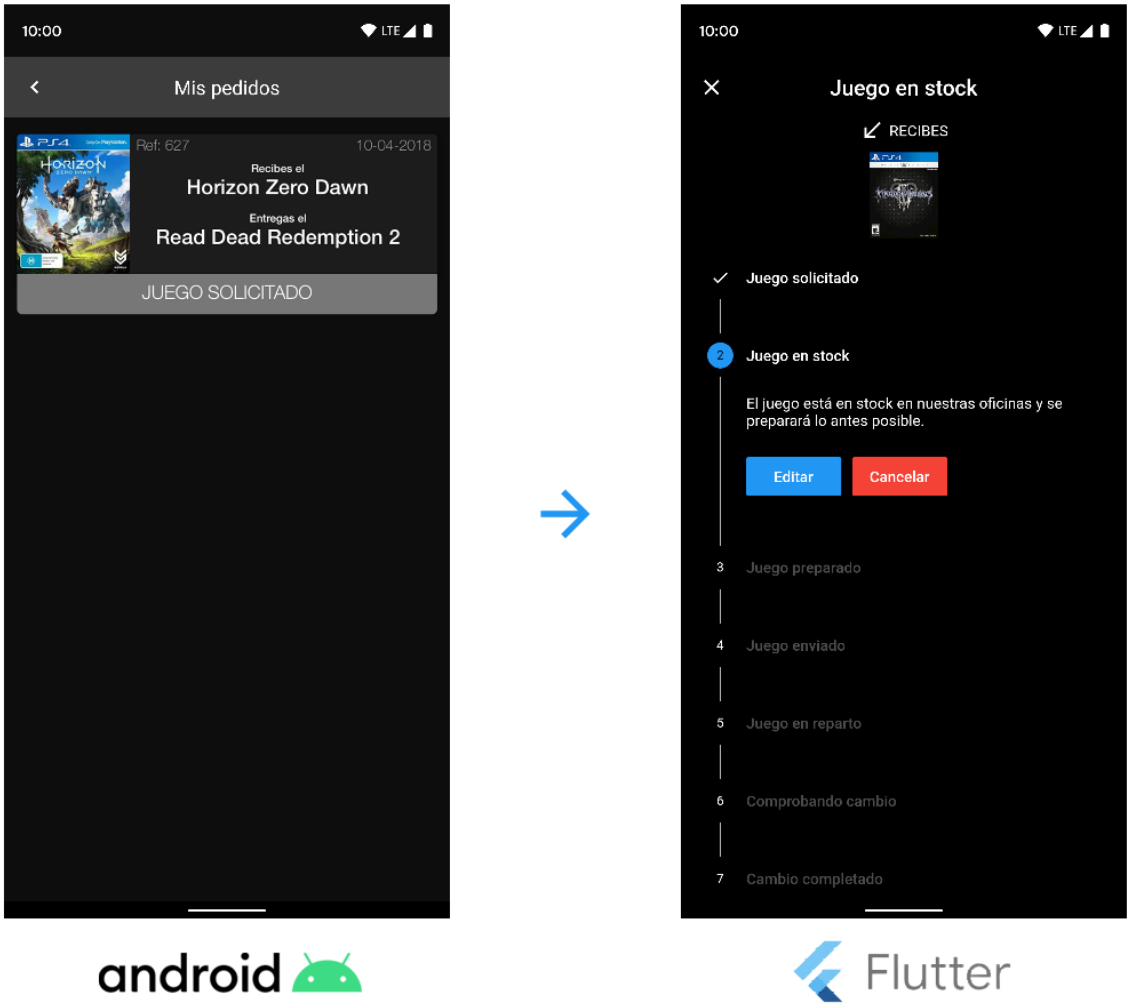

Figura 44: Migración de la vista de detalle de un pedido de Android a Flutter

8.1.2. Métricas

Una vez finalizada la migración se puede realizar una comparativa de distintas métricas, como la cantidad total de líneas de código, tamaño del archivo instalable, tamaño de la aplicación instalada en el dispositivo o el tiempo de inicio de la aplicación.

Las métricas “Tamaño app instalada”, “Tiempo de inicio (4G)” y “Tiempo de inicio (WiFi)” han sido tomadas utilizando un dispositivo de referencia Google Pixel 3 XL con el sistema operativo Android 10.

Para obtener la métrica “Líneas de código” se utilizó el *plugin* Statistics[28]. Para las líneas de código en Android, se contabilizaron las líneas de código Java (lógica)

[28] https://plugins.jetbrains.com/plugin/4509-statistic/

presentes en el directorio “`app/src/main/java`” y las líneas de código XML (interfaz) presentes en el directorio “`app/src/main/res/layout`”.

Para obtener las métricas “Tamaño del archivo instalable (APK)” y “Tamaño de la aplicación instalada”, se ha compilado un archivo instalable (APK) y posteriormente se ha instalado en el dispositivo de referencia mencionado anteriormente utilizando una versión de producción (*release*) de la aplicación. En Flutter, la versión utilizada para obtener estas métricas es `1.7.8+hotfix.2` (versión estable).

Para obtener las métricas “Tiempo de inicio (4G)” y “Tiempo de inicio “(WiFi)” se ha utilizado la aplicación con la sesión iniciada en la misma cuenta, realizando la media de 10 muestras para cada una de las métricas, desde que se pulsa el icono de la aplicación para abrirla hasta que carga por completo la vista principal. La métrica de “Tiempo de inicio (4G)” se realizó utilizando el dispositivo de referencia mencionado anteriormente utilizando la red 4G de Movistar. La métrica de “Tiempo de inicio (WiFi)” se realizó utilizando la red WiFi de Movistar bajo la banda de 5GHz con un ancho de banda de 50Mbps.

Métrica	Android	Flutter
Líneas de código	19782	12800
Tamaño del archivo instalable (APK)	9,3 MB	24,2 MB
Tamaño de la aplicación instalada	13,36 MB	53,32 MB
Tiempo de inicio (4G)	10 s	1,82 s
Tiempo de inicio (WiFi)	9,79 s	1,51 s

Tabla 2: Comparativa de métricas de la aplicación de Swapp en Android y Flutter

Como se puede observar en la Tabla 2, las líneas de código se han visto reducidas en un 35,3%. Concretamente, en Android, de las 19782 líneas de código contabilizadas, 13228 corresponden a Java (lógica) y 6554 corresponden a XML (interfaz). Se puede observar además que el tamaño tanto del archivo instalable como de la aplicación instalada han aumentado un 160,2% y un 299,1%, respectivamente. Teniendo en cuenta que los teléfonos inteligentes actualmente poseen mayor cantidad de almacenamiento, el aumento del tamaño de la aplicación es asumible para obtener un aumento de rendimiento. En el caso de la aplicación de Swapp, la diferencia de tiempo al abrir la aplicación en 4G o WiFi es bastante grande, reduciéndose en un 81,8% y 84,58%, respectivamente.

8.1.3. Proceso de migración

Debido al reducido tamaño del equipo de desarrollo, compuesto por 3 personas (dos de ellas a tiempo parcial), incorporar nuevas funciones en las distintas aplicaciones de la plataforma (web, Android y iOS) suponía una gran cantidad de código duplicado y plazos de entrega elevados, dado que había una única persona encargada de desarrollar las funciones para cada plataforma. Posteriormente se decidió dejar de dar soporte a la aplicación web, por lo que todos los recursos se dedicaban a dos plataformas en lugar de tres, lo que permitió acelerar los tiempos de entrega para las nuevas funcionalidades.

Cuando las aplicaciones de Android y iOS eran lo suficientemente estables y tenían las funciones que iban a ser necesarias durante los siguientes meses, se decidió migrar ambas aplicaciones a Flutter. De esta forma, todos los recursos del equipo de desarrollo se dedicarían a un único código, lo que permitió incorporar nuevas y mejores funcionalidades en la aplicación de una forma más rápida y sin necesidad de desarrollar las mismas funciones varias veces.

La migración comenzó oficialmente en diciembre de 2018 y terminó en abril de 2019, momento en el que todas las funciones principales estaban desarrolladas completamente en Flutter. El proceso de migración se puede dividir en dos bloques principales: la migración de la interfaz y la migración de la lógica. Por el flujo de trabajo utilizado, cuando se finalizaba la migración de una de las vistas, se migraba seguidamente la lógica de esa vista, completando poco a poco todas las vistas de la aplicación. Después de publicar la primera versión estable, tanto en Android como en iOS, el ritmo de desarrollo de las nuevas funcionalidades era notablemente superior, por la facilidad de uso del framework Flutter y su lenguaje Dart. Además, poco a poco se fueron incluyendo mejoras en la interfaz y animaciones para mejorar la interfaz y experiencia del usuario.

Para validar la aplicación, dado que el tiempo y los recursos disponibles eran escasos, no se pudo validar mediante pruebas unitarias de la lógica, pruebas de integración o pruebas de la interfaz de usuario. Antes de publicar la primera versión estable, se realizó una validación manual de todas las vistas y todas las funcionalidades de la aplicación en varios dispositivos Android y iOS, para así garantizar un correcto funcionamiento de la aplicación en ambas plataformas.

Como anécdotas en el transcurso de la migración, surgieron algunos problemas, especialmente con las librerías de terceros. Por ejemplo, cuando comenzó la migración, no existían librerías para integrar fácilmente la plataforma de pagos Stripe[29] o la plataforma Intercom[30] para el chat de soporte. Pese a contactar con ambas empresas para consultar acerca de sus planes de soporte de sus plataformas en Flutter, la respuesta siempre era que, al menos a corto plazo, no tenían planes para desarrollar librerías oficiales. Algunas semanas más tarde se publicaron librerías no

[29] https://stripe.com/
[30] https://www.intercom.com/

oficiales que permitían utilizar tanto Stripe[31] como Intercom[32] en Flutter. Además, actualmente colaboro con el desarrollo de la librería de Intercom para Flutter, a la que he añadido pruebas y corregido errores.

Finalmente, en el proyecto completo de migración, aunque aprovechando el potencial de Flutter se rediseñaron la mayoría de las vistas, sí se utilizaron algunos de los patrones especificados en este documento. Los componentes de Android y fragmentos de la lógica que se migraron siguiendo los patrones especificados previamente en los capítulos 6 y 7 son:

- **Activity:** Cada una de las vistas de la aplicación se reemplazó por los respectivos *widgets* `MaterialPageRoute`.
- **TextView:** Para mostrar los distintos textos en la aplicación, todos los `TextView` fueron reemplazados por *widgets* `Text`.
- **ImageView:** Para mostrar imágenes se utilizó el *widget* `Image` con los constructores `asset()` y `network()` dependiendo de su procedencia (memoria o servidor, respectivamente).
- **Button:** Para interactuar en la aplicación, los *widgets* utilizados fueron, en su gran mayoría, `RaisedButton`.
- **RecyclerView:** Para mostrar listas con diversos elementos, se hizo uso del *widget* `ListView`.
- **Chip, CheckBox, RadioButton:** Para mostrar información o modificar ajustes del usuario se hizo uso de *widgets* como `Chip`, `CheckBox` y `Radio`, entre otros.
- **FloatingActionButton:** Tras el rediseño, en muchas de las vistas, la acción principal pasó a ubicarse en un *widget* `FloatingActionButton` que se animaba y mostraba más claramente la acción, en lugar de botones normales.
- **ProgressBar:** Para mostrar el progreso en distintas acciones en la aplicación, los *widgets* utilizados tras la migración fueron `CircularProgressIndicator` y `LinearProgressIndicator`, dependiendo del contexto.
- **Toolbar:** El resultado de aplicar este patrón fue incluir el *widget* `AppBar` en los distintos *widget* `Scaffold` de la aplicación, añadiendo además distintas acciones contextuales dependiendo de la vista.
- **SnackBar:** Para mostrar mensajes informativos o de error al realizar una determinada acción, al igual que ocurría en la aplicación nativa en Android, se hizo uso del *widget* `SnackBar`.
- **ViewPager:** Para hacer una introducción del funcionamiento de la aplicación la primera vez que se abría, se utilizaron una serie de imágenes dentro de un *widget* `PageView`, que permitía su desplazamiento horizontal.
- **LinearLayout, ConstraintLayout:** Para estructurar la mayoría de las vistas de la aplicación se usó una combinación de *widgets* `Column` y `Row` que permitieron obtener resultados similares.

---

[31] https://pub.dev/packages/stripe_api
[32] https://pub.dev/packages/intercom_flutter

- **Llamadas al servidor:** Las llamadas al servidor para obtener o enviar información se migraron haciendo uso de la librería `http` y los métodos `get()`, `post()` y `put()` de dicha librería.
- **Formularios:** Los formularios de la aplicación en Flutter se migraron para ser validados usando la lógica de la propia aplicación (mediante el método `validate()` del estado actual del *widget* `Form`) para evitar errores, además de ser validados una segunda vez por parte del servidor.
- **Almacenamiento local:** Para el almacenamiento local, dado que en la aplicación nativa se hacía uso de `SharedPreferences`, en Flutter se utilizó la librería `shared_preferences`.

9. Conclusiones

Con respecto al objetivo principal del documento, se ha elaborado una extensa lista de patrones que un desarrollador Android puede seguir para convertir su aplicación nativa en Android a Flutter. De esta forma, el desarrollador puede acudir a este documento para observar, dependiendo de qué componentes visuales utiliza en su aplicación, cuáles son los mejores equivalentes a utilizar para migrar su interfaz de usuario a Flutter. Dado que el proceso de migración de la lógica será el más costoso por tener que reescribir gran parte del código, se han proveído algunos de los casos de uso más comunes para que el desarrollador pueda observar cómo realizar dichas funciones de la mejor forma posible en Flutter.

La mayoría de estos patrones han sido validados gracias al caso de estudio, donde no se migró exactamente la misma interfaz, sino que se aprovechó el potencial de Flutter para construir una mejor interfaz de usuario.

Por tanto, se puede concluir que el objetivo principal del TFG ha sido cumplido de forma satisfactoria, tras haber elaborado una lista de patrones de migración tanto de interfaz de usuario como de lógica, que además han sido utilizados y validados en un caso de estudio que ha desarrollado exitosamente el proyecto de migración de la aplicación nativa en Android a Flutter.

El uso de Flutter a corto plazo puede suponer una carga de trabajo que determinados equipos de desarrollo pueden no ser capaces de asumir, especialmente si ya tienen disponibles sus aplicaciones en Android y iOS y simplemente quieren unificar el desarrollo. Sin embargo, las opciones de personalización de la interfaz de usuario que ofrece Flutter son mucho mayores, por lo que se podrá conseguir una interfaz y experiencia de usuario mucho mejor. Además, una vez se disponga de la aplicación migrada a Flutter, en un futuro próximo se podrá generar una versión web de la aplicación, o versiones de escritorio que funcionarán utilizando el mismo código existente, por lo que se ganará mucha más versatilidad y se podrá llegar a un público más amplio. Aunque actualmente Flutter para escritorio y web se encuentra en desarrollo, ya se puede experimentar con él. De hecho, la mayoría de las imágenes de este documento que muestran una pequeña ventana ejemplificando un fragmento de código están realizadas compilando esa aplicación para el sistema operativo Windows. Las posibilidades que habilitará este *framework* en el futuro lo harán mucho más atractivo, si cabe, de lo que es ahora mismo.

Desde el punto de vista personal, el desarrollo de este TFG me ha permitido desarrollarme profesionalmente dado que, en el transcurso de la migración, he aprendido a utilizar un nuevo *framework*, Flutter, junto con su lenguaje de programación Dart, que me será de gran utilidad para mi carrera profesional en el desarrollo de aplicaciones multiplataforma.

En el Grado en Ingeniería Informática, cursado en la Escuela Técnica Superior de Ingeniería Informática, las asignaturas que más me han resultado útiles para el desarrollo de e TFG son:

- **Interfaces persona computador:** Gracias a los conocimientos obtenidos en esta asignatura, rediseñamos la aplicación haciéndola más fácil de utilizar y más visual, basándonos en los datos de uso de la aplicación de los usuarios existentes.
- **Gestión de proyectos:** El proyecto de migración llevado a cabo se realizó exitosamente, en parte, por una correcta gestión de las tareas y recursos dedicadas al proyecto, tal y como se enseña en esta asignatura.
- **Proyecto de ingeniería del software:** Para comenzar el proyecto, inicialmente se realizó una abstracción de las partes principales de la aplicación junto con sus casos de uso, lo que permitió definir correctamente las tareas a realizar y su prioridad para así llevar a cabo una migración completa de todas las funcionalidades existentes en la aplicación nativa en Android a la aplicación nativa multiplataforma en Flutter.

10. Trabajo futuro

Como trabajo futuro de este TFG queda la ampliación de patrones de migración de la interfaz y profundizaciones en distintos casos de uso posibles. De esta forma, y con una base lo suficientemente extensa de patrones de migración de la interfaz de usuario, se podría trabajar en el desarrollo de una herramienta que permita la automatización parcial de este trabajo de migración. Mediante una serie de equivalencias entre los componentes de la interfaz de usuario de una aplicación nativa en Android y los respectivos *widgets* de Flutter, junto con los posibles parámetros de cada uno de ellos, se podría convertir, al menos parcialmente (existen componentes o parámetros en Android sin equivalencias directas en Flutter), un archivo XML que defina una interfaz de usuario en Android a su equivalente en Flutter en un archivo Dart.

Además, debido a que el *framework* puede cambiar a lo largo del tiempo, los patrones de migración pueden llegar a quedar desactualizados en algún momento, por lo que será necesario adaptarlos a los cambios que reciba el *framework* Flutter, además de añadir nuevos patrones que permitan ampliar la lista de patrones de migración para más componentes de Android.

Finalmente, la lista de patrones de migración, tanto de interfaz como de lógica, puede verse alterada cuando se incorpore un nuevo *widget* que represente una equivalencia mejor con su respectivo componente en Android. Esto, junto con las posibles futuras funcionalidades del *framework* (como el soporte para aplicaciones de escritorio o web, entre otras plataformas), harán necesario un mantenimiento a largo plazo de este documento que permita a los desarrolladores migrar sus aplicaciones nativas en Android a Flutter en cualquier fase del desarrollo del *framework*.

11. Bibliografía

[1] Parker, J. (consultado el 15 de agosto de 2019). *10 years of growth of Mobile App Market*. Obtenido de Knowband: https://www.knowband.com/blog/mobile-app/growth-of-mobile-app-market/

[2] Ariel. (consultado el 15 de Agosto de 2019). *iOS Developers Ship 29% Fewer Apps In 2017, The First Ever Decline – And More Trends To Watch*. Obtenido de AppFigures: https://blog.appfigures.com/ios-developers-ship-less-apps-for-first-time/

[3] *Understanding the Xamarin Mobile Platform*. (consultado el 4 de Junio de 2019). Obtenido de Microsoft Docs: https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/building-cross-platform-applications/understanding-the-xamarin-mobile-platform

[4] Evkoski, B. (consultado el 4 de Junio de 2019). *React Native: What it is and how it works*. Obtenido de Medium: https://medium.com/we-talk-it/react-native-what-it-is-and-how-it-works-e2182d008f5e

[5] *Flutter Technical Overview*. (consultado el 4 de Junio de 2019). Obtenido de Flutter Docs: https://flutter.dev/docs/resources/technical-overview

[6] *Beyond Mobile: Material Design, Adaptable UIs, and Flutter (Google I/O'19)*. (consultado el 18 de Junio de 2019). Obtenido de YouTube: https://www.youtube.com/watch?v=YSULAJf6R6M

[7] *Why Android developers should pay attention to Flutter in 2019*. (consultado el 15 de Agosto de 2019). Obtenido de Codemagic: https://blog.codemagic.io/why-android-developers-should-pay-attention-to-flutter-in-2019/

[8] Bellinaso, M. (consultado el 15 de Agosto de 2019). *Flutter: the good, the bad and the ugly*. Obtenido de Medium: https://medium.com/asos-techblog/flutter-vs-react-native-for-ios-android-app-development-c41b4e038db9

[9] Skuza, B., Mroczkowska, A., & Włodarczyk, D. (consultado el 15 de Agosto de 2019). *Flutter vs React Native – what to choose in 2019?* Obtenido de Droids On Roids: https://www.thedroidsonroids.com/blog/flutter-vs-react-native-what-to-choose-in-2019

[10] Li, L. (consultado el 12 de Junio de 2019). *Porting an Android application to Xamarin?* Obtenido de StackOverflow: https://stackoverflow.com/a/37221856

[11] Mensaje en el foro Reddit (consultado el 12 de Junio de 2019). *Migrate Android project to Xamarin*. Obtenido de Reddit: https://www.reddit.com/r/xamarindevelopers/comments/60fof5/migrate_android_project_to_xamarin/df6o8ej/

[12] Pelgrims, K. (consultado el 12 de Junio de 2019). *Converting an app to React Native — Why and how*. Obtenido de Medium: https://medium.com/leoilab/converting-an-app-to-react-native-why-and-how-b56c02c07b96

[13] Akhi. (consultado el 13 de Junio de 2019). *How to convert android existing app to react-native android*. Obtenido de StackOverflow: https://stackoverflow.com/a/42086006

[14] *Flutter Live - Flutter Announcements and Updates (Livestream)*. (consultado el 15 de Agosto de 2019). Obtenido de YouTube: https://www.youtube.com/watch?v=NQ5HVygq1Qc

[15] *Flutter Install guide*. (consultado el 25 de Junio de 2019). Obtenido de Flutter Docs: https://flutter.dev/docs/get-started/install

[16] *A tour of the Dart language*. (consultado el 26 de Junio de 2019). Obtenido de Dart Docs: https://dart.dev/guides/language/language-tour

[17] *Dart*. (consultado el 26 de Junio de 2019). Obtenido de Wikipedia: https://es.wikipedia.org/wiki/Dart

[18] *Material Design Guidelines*. (consultado el 26 de Junio de 2019). Obtenido de Material Design: https://material.io/

[19] *Human Interface Guidelines*. (consultado el 26 de Junio de 2019). Obtenido de Apple Developer Docs: https://developer.apple.com/design/human-interface-guidelines/