Facultad de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO TUTELADO TEORÍA PS
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN INGENIERÍA DE COMPUTADORES

Paradigmas de desarrollo Android - iOS

Estudiante: Mauro A. de los Santos Nodar

A Coruña, febrero de 2021.

Índice general

Capítulo 1

Introducción

En el siguiente trabajo tutelado se expondrán las diferentes opciones que se nos presentan para trasladar nuestra app *Android* a la parte de desarrollo móvil no vista en la asignatura, ***iOS***. Para ello, se analizará como desarrollar código para *Apple* en dos supuestos, el primero, desde una plataforma cruzada, es decir, las llamadas *cross-platforms*, para obtener tanto código *Android* como *iOS* a partir de un sólo desarrollo, y en segundo lugar, las posibilidades que tenemos a la hora de querer transformar un código propio a *iOS*, habiéndolo desarrollado ya en *Android* nativo, como es el caso de nuestro trabajo tutelado de la asignatura. La otra opción restante consistiría en el desarrollo nativo de *iOS*, de la misma manera que se ha hecho en la asignatura con *Java* y *Android*, solo que con *Swift/Oriented C* en *iOS*. Esta posibilidad no se menciona ya que primero, no tiene relación directa con *Android*, principal temática de la asignatura, y segundo, no presenta ninguna dificultad o interés al consistir simplemente en codificar desde 0 nuestro proyecto en otro lenguaje. Por lo que una vez sabemos esto, vamos a comenzar:

Primero de todo, deberemos plantearnos el por qué es útil tener nuestra aplicación en ambas plataformas: *Android* e *iOS*. En primer lugar, sabemos que cubriendo ambas plataformas estamos cubriendo más del 95% de mercado móvil del mundo [1], lo cual sin duda es una ventaja. Por otro lado, para ajustar más esta información, podremos fijarnos también en la tasa de usuarios en cada plataforma dependiendo de nuestra localización o dispositivos en [2], así como otros factores externos como la disposición de los usuarios a gastar dinero en apps (que según *Statia*, es mayor en *Apple*) por si nuestra app incorpora pagos, entre otros.

Obviamente, tendremos que hacer balance con los aspectos negativos, como el precio que supone migrar a *iOS*, bien traducido en coste económico, en retraso de la fecha de publicación de la *release* o la significativa diferencia de tasas entre *Play Store* y *App Store* (25€ de uso ilimtado vs 90€ anuales, respectivamente), por ejemplo.

Una vez analizado esto y decidido que queremos nuestra app en *iOS*, vemos que las opciones que se nos presentan son fundamentalmente tres, de las cuales tenemos código *Android*

de por medio en dos de ellas: un desarrollo híbrido en una *cross platform* (2.1), o bien una traducción de nuestro proyecto *Android* a *iOS* (2.2, 2.3). Normalmente se debería considerar la primera, o incluso la opción de desarrollo simultáneo de la app en ambos lenguajes en el caso de no dominar uno multiplataforma, aunque también podremos adoptar medidas de traducción, las cuales no serán del todo prácticas debido a la diferencia entre componentes nativos, sistemas operativos, lenguajes de programación, etcétera. Estas diferencias serán las que traigan consigo el principal problema a la hora de cambiar entre plataformas, y aunque podamos encontrar un gran listado en [3], se mencionan a continuación de forma resumida algunas de las más importantes:

- *Android* es *open-source code*, mientras que *iOS* no.
- Diferentes enfoques a la hora de acceder a datos y transferir ficheros.
- Navegación muy diferente: con los 3 botones de *Android*: *Home, Back and Multitasking* frente al *Main button* de *iOS*.
- Interfaces de usuario basadas en dos principios opuestos: el *Material Design* contra el *Flat Design* de *iOS*.
- Muchísimos otros como son librerías y *frameworks*, fragmentación de versiones, tamaños de dispositivos, lenguajes de programación, fuentes, iconos, etcétera.

Por lo que para finalizar esta introducción, vemos como hay múltiples aspectos positivos a considerar para desarrollar nuestra app también en *iOS*, los cuales deberemos contrastar junto a los costes que ésto supone. Si nuestra decisión es seguir adelante, deberemos saber que tenemos diferentes opciones, las cuales veremos en detalle en los siguientes capítulos.

Comentar que esta problemática es muy habitual en el mundo de desarrollo para dispositivos móviles ya que no todos los desarrolladores comienzan elaborando su proyecto pensando en ello, y como ejemplo de ésto tenemos decisiones adoptadas por gigantes del mercado como *AirBnB*, la cual comenzó en *iOS* y necesitó 14 meses para migrar a *Android*, o la propia *Instagram*, la cual tardó 4 años en estar en ambas plataformas desde que se lanzó inicialmente en *iOS*.

Por lo que vamos ahora a hablar de manera más práctica de las diferentes herramientas que nos permiten llevar a cabo las soluciones mencionadas.

Capítulo 2

Aproximaciones

2.1 Cross-Platforms

Vamos entonces a comenzar con el primer tipo de solución: el desarrollo en plataformas cruzadas o *cross-platforms*. Esta solución consiste en codificar toda nuestra aplicación en un lenguaje y entorno neutro a ambos sistemas, que a través de mecanismos propios sepa generar dos resultados finales diferentes desde el mismo código, uno para nuestra aplicación *Android* y otra para *iOS*. Esta solución es indudablemente una de las que menos trabajo y coste conlleva ya que sólo nos hará falta un desarrollo para obtener dos resultados, pero tendremos el inconveniente de la necesidad de conocer a fondo la plataforma donde lo vamos a realizar y además tener la solución decidida antes de comenzar el proyecto. Vamos a ver algunos ejemplos de esto, comenzando por el que hoy en día, más se está expandiendo y mayor potencial ofrece, ***Flutter***.

2.1.1 *Flutter*

Flutter es un SDK móvil multiplataforma de código abierto creada en 2017 por Google que se puede utilizar para crear aplicaciones *iOS* y *Android* con el mismo código fuente. *Flutter* no depende de la plataforma, porque tiene sus propios *widgets* y diseños, a diferencia de otros *frameworks* multiplataforma como ***React Native*** y ***Xamarin*** los cuales usan los *widgets* nativos y *WebViews*. El desarrollo es llevado a cabo con el lenguaje de programación ***Dart***, donde el código es compilado de forma nativa usando la GPU para renderizar una interfaz de usuario uniforme para ambas plataformas, así como brindando también la capacidad de acceso a características nativas de cada plataforma como *GPS*, *Bluetooth*, etcétera, eliminando por tanto los problemas de las diferencias entre plataformas comentadas en la introducción 1.

Aunque la documentación y las posibilidades que nos ofrece *Flutter* son inmensas, vamos a mencionar de forma resumida una serie de ellas:

- **Desarrollo** de aplicaciones móviles de forma más **rápida y dinámica**. Podemos hacer cambios en el código y verlos de inmediato en la aplicación gracias a su recarga en caliente, que generalmente solo toma (mili)segundos y ayuda a los equipos a agregar funciones, corregir errores y experimentar de forma más rápida.
- Diseñado para facilitar la **creación de sus propios *widgets*** o personalizar los ya existentes. Nuestra app tendrá prácticamente el mismo aspecto, incluso en versiones anteriores, tanto en los sistemas *Android* como en *iOS*.
- La arquitectura técnica de ambas plataformas es completamente diferente. *Flutter* usa **Dart** como lenguaje de programación, abstrayéndose así de Java/Kotlin en el caso de *Android* y de Swift/ObjectiveC en el caso de *iOS*.

Vemos por lo tanto el gran potencial que nos ofrece, siendo una solución relativamente nueva, pero que nos brinda innumerables ventajas y sin duda hay que tener en cuenta a la hora de planificar el desarrollo de nuestra app ya que nos puede ahorrar mucho tiempo y recursos si queremos una aplicación multiplataforma.

2.1.2 Otras

Como hemos visto, *Flutter* es una solución extramademente reciente, ya que nació en 2017 y sus primeras versiones realmente estables datan de 2018 y 2019, por lo que obviamente no es la única opción que se nos presenta. También tenemos otras como: [4]

- **React Native**: gigante multiplataforma hasta la llegada de *Flutter*, donde permite usar *JavaScript* para desarrollar una aplicación en ambos SO, pudiendo crear controles nativos, con los cuales se obtiene máxima velocidad y rendimiento. Sin embargo, al igual que otras soluciones, debe modicarse el código nativo si queremos acceder a las API nativas. Por otro lado, cuando se crean estos controles, se necesita personalizar su apariencia para *Android* e *iOS* por separado, lo que significa que solo se obtiene un 80% de reutilización del código, aproximadamente.
- **Apache Cordova**: permite escribir aplicaciones en *JavaScript* (o cualquier lenguaje que se compile en *JavaScript*), y cuenta con una serie de macros y servicios integrados, incluidos *Ionic* o *Telerik*, entre otros. Posee una baja curva de aprendizaje.
- **Xamarin**: permite escribir código multiplataforma en *C#*. Es muy estable y al contar con *C#* y con acceso a controles nativos, es una buena opción con gran rendimiento, aunque no recomendada para juegos. Es de pago.
- **Más**: hay otras opciones como *Meteor* (*JavaScript*), *Kivy* (*Python*), etcétera.

Aún existiendo todas estas posibilidades, merece la pena destacar a *Flutter* ya que hoy en día es la solución multi-plataforma más respaldada por las cifras. Solo basta con ver en la siguiente gráfica las consultas en *StackOverflow* sobre los principales entornos de desarrollo *cross-platform* para apps móviles durante los últimos años, para observar que si queremos una solución de este tipo, es la primera que deberíamos considerar [5]:

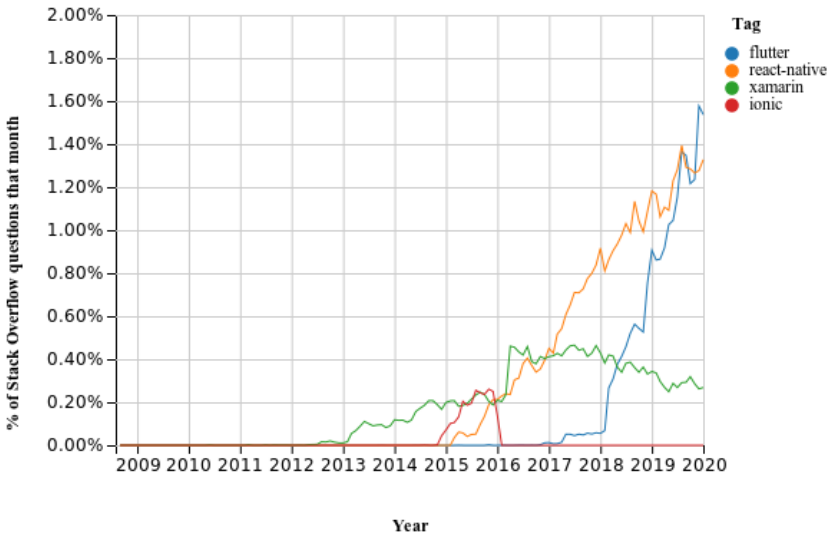

Figura 2.1: % de consultas al mes en Stack Overflow vs. Año

2.2 Kotlin

Vamos ahora con el segundo tipo de soluciones que hemos comentado, aquellas adoptadas una vez que tenemos nuestro código *Android* y queremos convertirlo a *iOS*. Dentro de estas, tenemos diferentes enfoques y puntos de vista por lo que vamos a comenzar hablando del reciente lenguaje en auge de *JetBrains*: **Kotlin**.

Como bien hemos visto, por cada plataforma, destacan dos lenguajes: en *Android*, **Java y Kotlin**, y en *iOS* **Objective C y Swift**. En ambos casos tenemos una tendencia similar, y es aquella que deja progresivamente de lado el uso de los lenguajes más antiguos como Java y Objective C para adoptar aquellos más modernos y funcionales como son **Kotlin** en el caso de *Android* y **Swift** en el de *iOS*. Las razones son múltiples, y una de ellas es la interoperabilidad y la capacidad de adaptación y traducción entre estos lenguajes, ya que después de dominar los fundamentos del lenguaje, el código se puede traducir fácilmente entre ellos. Por otro lado, estos lenguajes expresivos han aliviado la barrera del aprendizaje multiplataforma, por lo que estas ventajas, entre otras, hacen que esta tendencia sea predominante.

2.2.1 Traductores Kotlin-Swift y viceversa

Por lo que en esta sección, vamos en primer lugar, a analizar la posibilidad de desarrollar nuestro código *Android* en Kotlin, con la finalidad de después hacer una traducción a Swift para expandir nuestra app al mundo *iOS*. Para ésto, tanto como para el caso opuesto, contamos con diversas herramientes *open-source* de terceros como son:

- **Kotlift** [6]: "el primer compilador de lenguaje fuente a fuente de Kotlin a Swift".
- **SwiftKotlin** [7]: "Una herramienta para convertir código Swift a Kotlin de una manera fácil y rápida".
- **Gryphon** [8]: "Escribe la lógica del código de *iOS* de tu aplicación en Swift, luego usa Gryphon para convertirlo en Kotlin y usarlo en *Android*".

Aunque su resultado no pueda está garantizado al 100%, debemos ver como son herramientas que nos pueden ayudar de gran manera a la hora de cambiar entre plataformas si hemos elegido desarrollar en las nuevas opciones de lenguajes de ambas plataformas.

2.2.2 KMM

Aunque si hablamos de Kotlin, no nos podemos quedar tan sólo en su capacidad de fácil adaptación a Swift, si no que deberemos mencionar la otra gran opción a la hora de usar Kotlin con la finalidad de llegar a *iOS*, y esta será el recientemente conocido como **KMM** o ***Kotlin Multiplatform Mobile***.

Kotlin Multiplatform Mobile, la estrella en ascenso en el espacio multiplataforma, es una función experimental (en *Alpha* desde mediados de 2020) que permite ejecutar Kotlin en *JavaScript*, *iOS*, aplicaciones de escritorio nativas y más. Es un SDK para el desarrollo móvil multiplataforma suministrado por JetBrains. Utiliza las capacidades multiplataforma de Kotlin e incluye varias herramientas y funcionalidades diseñadas para que la experiencia de crear aplicaciones móviles multiplataforma sea todo lo agradable y eficiente posible de principio a fin.

Con **KMM**, se puede disfrutar de la flexibilidad de desarrollo entre plataformas así como de mantener las ventajas de la programación nativa. Se utiliza una base de código única para la lógica empresarial de las aplicaciones de *iOS* y *Android*, y se escribe código específico de plataforma sólo cuando es necesario, es decir, para implementar una interfaz nativa o para trabajar con APIs específicas de la plataforma. Como bien vemos en la siguiente imagen:

Figura 2.2: KMM

Decir que es una solución todavia más reciente que *Flutter*, pero que de igual manera, se presenta como otra de las grandes opciones del futuro a la hora del desarrollo de apps para *Android* e *iOS*, por lo que deberemos comentar también una serie de características:

- El código compartido, escrito en Kotlin, se compila a código de bytes de JVM con Kotlin/JVM y a binarios nativos con Kotlin/Native, de modo que puede utilizar sus módulos de lógica empresarial de KMM como cualquier otra biblioteca móvil normal.
- Se trabaja con ambas plataformas desde un solo IDE (ya existe un complemento para la integración con dispositivos *iOS* y simulador directamente en *Android Studio*)
- Para *Android*, se puede utilizar Kotlin en cada parte de los proyectos. Al mismo tiempo, KMM ofrece sólida integración con el proceso de desarrollo de *iOS*, gracias a la interoperabilidad de Kotlin/Native con Objective-C/Swift, y la capacidad de utilizar APIs específicas de la plataforma.
- Kotlin/Native ofrece interoperabilidad bidireccional con Objective-C/Swift. Los módulos de Kotlin pueden utilizarse por completo en Swift/Objective-C. Puede utilizar marcos de trabajo Objective-C y bibliotecas en código Kotlin, así como bibliotecas Swift si su API se ha exportado a Objective-C. Por lo que el desarrollo es bidireccional.
- Integración con el gestor de dependencias *CocoaPods*: Se pueden añadir dependencias en bibliotecas *Pod* almacenadas en el repositorio *CocoaPods* o localmente, y también puede utilizar proyectos multiplataforma con destinos nativos como dependencia de *CocoaPods (Kotlin Pod)*. Aunque este concepto es bastante complejo y extenso, su importancia es clave y merece la pena mencionarlo.

Vemos entonces después de analizar sus características como **KMM** realmente se dispone a ser una solución de futuro y la principal competidora de **Flutter**. Dentro de todas las soluciones comentadas hasta el momento, han sido las dos a las cuales hemos dedicado más tiempo y espacio, por lo que antes de pasar al siguiente y último tipo de solución, vamos a dejar una imagen comparativa de ambas: [9]

Technology	Kotlin Multiplatform	Flutter
Programming Language	Kotlin	Dart
Components Library	Small very new support	Quickly growing, non-inclusive
Learning Curve	Easy to pick up, Kotlin syntax is similar to Java and Swift	Dart makes the barrier for entry higher and reactive programming isn't all intuitive.
Created By	JetBrains	Google
EcoSystem	Very new, is also supported by Google and growing.	Become mature and quickly growing. Still missing big apps in production.
Hot Reload	Not supported	Supported
Github Stars	33,500+ stars	102,000+ stars
First Release	Aug 2018	May 2017

Figura 2.3: *Flutter* vs KMM

2.3 Traducción directa

Para acabar, el último tipo de soluciones que nos quedan son aquellas que denomino de traducción o conversión directa. Son una de las soluciones más buscadas, debido a la sencillez y facilidad que traerían consigo, al ofrecer un programa o framework, que pulsando un par de botones nos tradujeran todo nuestro proyecto *Android* a *iOS*. Pero por desgracia, hay que decir que estas soluciones de traducción no existen, o al menos, en el sentido literal de la palabra. Son soluciones muy cómodas y fáciles de utilizar, pero también las más complicadas de implementar y llevar a cabo debido a las diferencias *inter-plataforma* ya comentadas. Se ha intentado alguna vez desarrollar este modelo de forma literal, así como se han intentado hacer aproximaciones, por lo que vamos a comentar dos de los casos más significativos.

2.3.1 Mechdome

Aunque si *googleamos* por soluciones de conversión de proyectos *Android* a *iOS* aún nos sigue apareciendo en diversas páginas y foros, decir que en 2018, dos años después de su lanzamiento, esta idea desapareció sin dejar rastro. Se postulaba como una solución ideal

para la problemática tratada en este trabajo: útil porque permitía la conversión automática de cada elemento de una aplicación *Android* en aplicaciones de *iOS* nativas de alta fidelidad, para que pudieran ser empleadas en dispositivos de Apple, sin necesidad de tener que codificar de nuevo, y por encima de todo, sin necesidad del código fuente, de binario a binario.

Para convertir una app de *Android* en una aplicación para los dispositivos Apple sólo era necesario enviar el archivo APK a MechDome. Automáticamente, la aplicación sería compatible con *iOS* y OS X, sin compartir el código fuente, estando en un breve período de tiempo nuestra app en *iOS* lista para publicarse en la App Store.

Los pasos que se indicaban a seguir en su documentación eran sencillos:

- Subir nuestra aplicación compilada a su página web.
- Seleccionar si queríamos una conversión para dispositivos reales o emuladores.
- Pagar por el servicio y esperar a que en un breve período de tiempo tuvieramos nuestro resultado.

Aunque tampoco merece la pena analizar a fondo esta idea ya que, hoy por hoy en 2021 no está operativa, y por otro lado, no la he podido probar personalmente y sólo me baso en opiniones encontradas por Internet, merece la pena comentar su funcionamiento e idea por encima ya que es una solución muy buscada y perseguida.

En cuanto a sus aspectos técnicos, destacar que permitía utilizar las API estándar de Java de *Android*, las bibliotecas de *Android* de terceros y las bibliotecas de Java de terceros. Además, daba soporte también lenguajes basados en *Java Virtual Machine* (JVM), como por ejemplo el visto 'Kotlin'. Las funciones estándar de *iOS* y el soporte de hardware estaban totalmente disponibles: cámara, sensores, *Touch ID*, GPS, *AirDrop*... Y el pilar de todo esto era la compilaciónde la aplicación de *Android* directamente en un código *iOS* completamente nativo, donde el código convertido estaba ya optimizado.

El funcionamiento de este framework se basaba en que su compilador de aplicaciones de *Android* tomaba la aplicación que se había enviado, convertía su código de bytes de *Android* en un ejecutable optimizado y creaba un paquete de *iOS*, donde empaquetaba todos los recursos y metadatos. Se obtenía así una aplicación *iOS* estándar con el contenido de su APK de *Android* completamente reproducido y la conversión de código era de binario a binario.

Las ventajas a la hora de usar este framework eran obvias ya que nos permitía ahorrarnos el tiempo de desarrollo *iOS* reutilizando todo nuestro codigo *Android*, tanto para despliegues como para posteriores actualizaciones, aunque obviamente tenía sus contras ya que bien es cierto que determinados paquetes y funcionalidades no eran compatibles, como por ejemplo funcionalidades equivalentes para *widgets*, accesos como root, fondos de pantalla, etc. Tampoco lo eran algunas APIs de *Android* como *RenderScript*, Bluetooth o *Battery Manager* y algunos

formatos de anuncios como *Native Ads Advanced* tampoco eran soportados. Por otro lado algunas de las API de Google Firebase, como los informes de fallos, tampoco tenían soporte. [10]

En cuanto a las tarifas, convertir sus aplicaciones de *Android* existentes a *iOS*, se podía realizar a través de diferentes paquetes con diferentes precios cada uno según el servicio que se requisiera. De forma gratuíta podíamos convertir una app, pero después teníamos que contratar algún plan mensual, donde por ejemplo con el plan básico, se podían convertir 3 aplicaciones de *Android* al mes.

Al esfumarse del mercado en 2018, desde entonces no ha vuelto a aparecer algo parecido, o al menos que llegara a ser tan usado y conocido como fue **MechDome**, lo que invita a pensar que este modelo de negocio de traducción nativo a nativo es practicamente utópico y las otras soluciones mostradas en el trabajo son realmente en las que habría que poner el foco de atención, aunque como vemos, nunca podremos dar del todo por muerta a esta opción, después de saber que existió, aunque por degracia, no pudieramos probarlo y fuera por poco tiempo.

2.3.2 J2ObjC

Aunque parecido en el sentido de traducción, este enfoque difiere de Mechdome ya que consistirá en una herramienta de línea de comandos de código abierto llamada ***J2ObjC*** para la migración de código de *Android* a *iOS*. Esta herramienta, creada por Google, traduce el código fuente de Java a Objective-C, que como bien hemos venido comentando, es uno de los lenguaje de la plataforma *iOS*. Si se desea utilizar este enfoque para nuestro proyecto, se debe contar con un conocimiento del desarrollo en *iOS*, ya que la herramienta no nos brinda una traducción literal, operativa y sin errores, si no que requerirá un conocimiento en desarrollo de software *iOS* para tras llegar al código de Objective C, conseguir hacerlo operativo.

J2ObjC es compatible con la mayoría de las funciones de ejecución y lenguaje Java, lo cual incluye excepciones, clases, tipos, etc. También puede tener la traducción de tests JUnit. Sin embargo, la herramienta no incluye ningún kit de herramientas de IU independiente de la plataforma, es decir, se necesita escribir el código de la interfaz de usuario de *iOS* desde cero.

Hay que tener en cuenta que, a diferencia de soluciones del estilo de MechDome, que convertía binario en binario, **J2ObjC** no puede realizar esta conversión, si no que se necesita acceder al código fuente de Java desde su aplicación de *Android*, como la mayoría de traductores entre lenguajes que hemos comentado a lo largo de la memoria.

Como se puede ver, esta herramienta ayudará a migrar la lógica de la aplicación y el código relacionado con el modelo de datos, sin embargo, se debe probar a fondo y remodelar para que sea totalmente operativo, por lo que aunque no una solución inmediata, no deja de ser un apoyo o ayuda en la fase de desarrollo *iOS* de nuestro proyecto *Android*.

Capítulo 3

Conclusiones

Por lo tanto, después de repasar todas estas opciones, debemos sacar varias cosas en conclusión. Primero de todo, que las opciones para convertir un proyecto *Android* a *iOS* son:

- **Desarrollar simultáneamente y desde 0**, dos aplicaciones separadas en los lenguajes nativos de ambas plataformas que elijamos, Java o Kotlin en caso de *Android*, y Swift u Objective C en el caso de *iOS*.
- **Desarrollar un sólo proyecto** en las denominadas ***cross-platforms***, a partir del cual, se crearán dos aplicaciones diferentes, una para cada entorno. En este apartado, deberemos destacar a Flutter y KMM, basadas respectivamente en los lenguajes Dart y Kotlin y vistas en detalle en la memoria.
- **Traducciones directas**: Aquí podemos realizar otras subdivisiones, teniendo en primer lugar, traductores de los **lenguajes funcionales y modernos**: Kotlin y Swift, en segundo lugar, traductores de los **lenguajes más antiguos**: Java y Objective C, y en tercero, **traductores de binarios**, como Mechdome.

Por lo que, en conclusión, si se busca una herramienta que una vez que tengamos nuestra app *Android* acabada y desarollada de forma nativa, nos la convierta a *iOS* en un par de clicks, como mi caso particular del trabajo tutelado de la asignatura, deberemos saber que la respuesta es no, a día de hoy no la hay. Pero deberemos saber también, que dependiendo de si tenemos nuestra aplicación en Java o Kotlin, y de nuestro dominio de *cross-platforms* como Flutter o KMM (realmente las que a día de hoy tienen más proyección), tenemos diferentes opciones al alcance de nuestra mano, para con eso si, un cierto grado de esfuerzo, convertir nuestro código *Android*, pasando por una fase de desarrollo intermedia, bien en lenguaje multiplataforma o bien en lenguaje nativo de *iOS*, conseguir una solución totalmente válida para que nuestra aplicación esté también en la plataforma de la manzana.

Bibliografía

[1] "How to convert an android app to ios or vice versa," https://mlsdev.com/blog/how-to-convert-android-app-to-ios, accedido en enero de 2021.

[2] "Statcounter: Global stats," https://gs.statcounter.com/os-market-share/mobile/worldwide, accedido en enero de 2021.

[3] "Android vs. ios," https://www.diffen.com/difference/Android_vs_iOS, accedido en enero de 2021.

[4] "How do i create an app for both android and ios mobile devices?" https://www.quora.com/How-do-I-create-an-app-for-both-Android-and-iOS-mobile-devices, accedido en enero de 2021.

[5] "¿es flutter el framework del futuro?" https://www.bbvanexttechnologies.com/es-flutter-el-framework-del-futuro/, accedido en enero de 2021.

[6] "Kotlift," https://github.com/studo-app/Kotlift, accedido en enero de 2021.

[7] "Swiftkotlin," https://github.com/angelolloqui/SwiftKotlin, accedido en enero de 2021.

[8] "Gryphon: The swift to kotlin translator," https://vinivendra.github.io/Gryphon/, accedido en enero de 2021.

[9] "Comparison between flutter vs kotlin multiplatform: Which one should you prefer?" https://kodytechnolab.com/flutter-vs-kotlin-comparison, accedido en enero de 2021.

[10] "How to convert an android app to ios?" https://www.devteam.space/blog/how-to-convert-an-android-app-to-ios/#:~:text=This%20approach%20uses%20an%20open,a%20skilled%20iOS%20development%20team, accedido en enero de 2021.

[11] "Kotlin multiplatform mobile ahora es alpha," https://blog.jetbrains.com/es/kotlin/2020/09/kotlin-multiplatform-mobile-ahora-es-alpha/, accedido en enero de 2021.