Tesinas de Grado

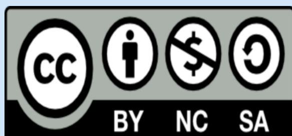
Mauro Omar Rodriguez

Innovación y desarrollo multiplataforma en la industria Fintech

2023

Instituto de Ingeniería y Agronomía

Carrera: Ingeniería en Informática



Esta obra está bajo una Licencia Creative Commons.
Atribución – No comercial – Compartir igual 4.0
https://creativecommons.org/licenses/by-nc-sa/4.0/

Documento descargado de RID - UNAJ Repositorio Institucional Digital de la Universidad Nacional Arturo Jauretche

Cita recomendada:
Rodriguez, M. O. (2023). *Innovación y desarrollo multiplataforma en la industria Fintech* [Práctica Profesional Supervisada, Universidad Nacional Arturo Jauretche]. https://rid.unaj.edu.ar/handle/123456789/2876

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Ingeniería en Informática

Práctica profesional Supervisada
Informe Final

Innovación y desarrollo multiplataforma en la industria Fintech

Rodriguez Mauro Omar

Florencio Varea, diciembre de 2023

PRÁCTICA PROFESIONAL SUPERVISADA (PPS)
Informe Final

DATOS DEL ESTUDIANTE
Apellido y Nombres: Rodriguez Mauro Omar
Nº de Legajo: 18889
Correo electrónico: mauromarod@gmail.com
Cantidad de materias aprobadas al comienzo de la PPS: 45

PROFESOR TUTOR DE LA UNAJ
Apellido y Nombres: Dr. Ing. Morales Martín, Ing. Cabral Leonardo Julian
Correo electrónico: martin.unaj@gmail.com, ljcabral@unaj.edu.ar

PROFESOR TUTOR DEL TALLER DE APOYO A LA PRODUCCIÓN DE TEXTOS ACADÉMICOS DE LA UNAJ
Apellido y Nombres: Lic. Prof. Kelly Carolina
Correo electrónico: kellygcarolina@gmail.com

DATOS DE LA ORGANIZACIÓN DONDE SE REALIZA LA PPS
Nombre o Razón Social: ALAU TECNOLOGIA S.A.U.
Dirección: Nicaragua 4677
Teléfono: 1131417932
Sector: Servicios de financiación y actividades financieras

TUTOR DE LA EMPRESA/INSTITUCIÓN
Apellido y Nombres: Arias Roberts Rodrigo
Correo electrónico: rodrigo.arias@uala.com.ar

FIRMA DEL COORDINADOR DE LA CARRERA

Firma Estudiante:	Firma tutor UNAJ:	Firma tutor TAPTA UNAJ:	Firma tutor Empresa:

Resumen

El presente trabajo forma parte de la Práctica Profesional Supervisada (PPS) de la carrera de Ingeniería en Informática, la cual representa una oportunidad para los estudiantes de abordar problemas reales y desarrollar soluciones prácticas. Actualmente, uno de los principales desafíos que enfrentan las empresas tecnológicas es la duplicidad de esfuerzos en el desarrollo de aplicaciones para diferentes sistemas operativos, en particular iOS y Android. Esta situación no solo implica mayores costos sino también tiempos de desarrollo prolongados. A partir de esta problemática surge la propuesta de este proyecto: establecer un nuevo paradigma de desarrollo basado en la integración de módulos multiplataforma en aplicaciones nativas de iOS y Android. Con esto, se busca no solo mejorar los tiempos de desarrollo sino también optimizar los recursos y esfuerzos en la implementación de lógica de negocio.

Abstract

This work is part of the Supervised Professional Practice (SPP) in the field of Computer Engineering, which offers students the opportunity to tackle real-world problems and develop practical solutions. Currently, one of the main challenges facing technology companies is the duplication of efforts in developing applications for different operating systems, particularly iOS and Android. This situation not only implies higher costs but also extended development timelines. From this problem the proposal of this project arises: to establish a new development paradigm based on the integration of cross-platform modules into native iOS and Android applications. The objective is not only to enhance development timelines but also to optimize resources and efforts in the implementation of business logic.

Dedicatorias y agradecimientos

En primer lugar quiero dedicarle este trabajo a mi familia, en especial a la abuela Tota, quien me apoyó en cada paso que di en mi vida y sé que está feliz por este nuevo logro. A mi mamá, por la paciencia y consejos; a mi papá, por hacerme apostar siempre por más, y a mi hermana y ahijada, por alegrarme y alentarme en los momentos más difíciles de la carrera.

Agradecer a Ualá por permitirme llevar a cabo el proyecto, a Lenny y mis compañeros por interesarse en el día a día, en especial a Rodrigo Arias por darme la confianza necesaria para liderar el desafío de cambiar el paradigma de desarrollo de una Fintech tan importante como Ualá.

Quiero agradecer también a todos los que forman parte de la Universidad Nacional Arturo Jauretche, a mis tutores Carolina y Julián, quienes me guiaron durante esta práctica profesional. A los docentes que brindan no sólo su tiempo y conocimiento, sino oportunidades de crecimiento, en especial a profesores como Oscar Cortes Bracho, quien vio en mí un potencial que yo no sabía que tenía y me animó a postularme a mi primer trabajo, y Pablo Sabatino, quién clase a clase nos brindó, además de los conocimientos técnicos, consejos y lecciones sobre su experiencia laboral. Un agradecimiento en especial a Sabri con quien tuve el privilegio de cursar la mayor parte de la carrera y me dio una mano en más de una caída.

Por último quiero expresar mi más profundo agradecimiento a todas las personas que hicieron posible que, como otros tantos habitantes de Florencio Varela, ciudad muchas veces postergada, pudiéramos cumplir nuestro sueño de ser primera generación de universitarios. Gracias por creer en nosotros y darnos la oportunidad de demostrar que somos capaces de lograr grandes cosas.

Contenido

Resumen 1
Abstract 3
Dedicatorias y agradecimientos 4
1. Introducción 9
2. Objetivos 11
  2.1 Plan de trabajo 15
  2.2 Cronograma de trabajo 16
3. Marco teórico 18
  3.1 Inicio de las aplicaciones móviles 18
  3.2 Evolución del desarrollo de apps móviles 18
  3.3 Influencia de las Fintech 19
  3.4 Tendencias en el desarrollo de aplicaciones 19
  3.5 Desarrollo nativo 20
    3.5.1 Android 20
    3.5.1.1 Java 20
    3.5.1.2 Kotlin 21
    3.5.2 IOS 21
    3.5.2.1 Swift 21
  3.6 Desarrollo Cross-Platform 22
  3.6.1 Xamarin 22
  3.6.2 React Native 22
  3.7 Desarrollo Multiplataforma 23
    3.7.1 Flutter 23
    3.7.2 Kotlin Multiplatform 24
  3.8 Herramientas 24
    3.8.1 Gradle 25
    3.8.2 Maven Publish Plugin 25
    3.8.3 JFrog 26
    3.8.4 GitHub Actions 26
    3.8.5 CocoaPods 27

3.8.6 Split 27
4. Desarrollo 28
4.1 Situación inicial 28
4.2 Análisis de tecnologías 30
4.2.1 Lenguaje 30
4.2.2 Rendimiento 31
4.2.3 Interfaz de Usuario (UI) 32
4.2.4 Otros aspectos 32
4.2.5 Conclusiones 33
4.3 Módulo de Prueba 34
4.3.1 Get platform 34
4.3.2 Dependencias Multiplatform 39
4.3.3 Dependencias nativas 43
4.3.4 CI/CD 50
4.3.4.1 Android 50
4.3.4.2 IOS 56
4.4 Módulo Promociones 59
4.4.1 Análisis y propuestas 59
4.4.2 Desarrollo 62
5. Conclusiones 71
6. Reflexiones sobre PPS como espacio de formación 75
7. Referencias bibliográficas 77

Índice de figuras

FIGURA 1: PLAN DE TRABAJO. 16
FIGURA 2: ENCUESTA EN CANAL ANDROID . 29
FIGURA 3: ENCUESTA EN CANAL IOS . 29
FIGURA 4: WIZARD NEW PROJECT. 35
FIGURA 5: WIZARD KMP PROJECT. 36
FIGURA 6: WIZARD KMP APP. 36
FIGURA 7: PROYECTO INICIAL. 37
FIGURA 8: EJECUCIÓN PROYECTO INICIAL. 38
FIGURA 9: DEPENDENCIAS MULTIPLATAFORMA. 40
FIGURA 10: EJECUCIÓN CONSUMO SPACEX API. 41
FIGURA 11: EXPECT Y ACTUAL CLASS. 43
FIGURA 12: DEPENDENCIAS ANDROID. 44
FIGURA 13: DEPENDENCIAS IOS. 45
FIGURA 14: OBTENER TOKEN MEDIANTE USERCREDENTIALS. 46
FIGURA 15: IMPLEMENTACIÓN NATIVA DE USERCREDENTIALS. 46
FIGURA 16: VIEWMODEL ARCHITECTURE. 47
FIGURA 17: INICIALIZAR KOIN. 48
FIGURA 18: MIS CUPONES DESDE KMP. 49
FIGURA 19: CONFIGURACIÓN DE MAVEN Y JFROG. 51
FIGURA 20: TAREAS DE PUBLICACIÓN DESDE GRADLE. 52
FIGURA 21: REPOSITORIO JFROG. 53
FIGURA 22: INTEGRACIÓN DE DEPENDENCIAS KMP. 54
FIGURA 23: ANDROID WORKFLOW. 55
FIGURA 24: ANDROID ACTIONS. 55
FIGURA 25: IOS WORKFLOW. 58
FIGURA 26: IOS ACTIONS. 58
FIGURA 27: FLUJO DE PROMOCIONES INICIAL. 61

FIGURA 28: PROMOCIONES - DATA. 63

FIGURA 29: PROMOCIONES – DOMAIN. 64

FIGURA 30: PROMOCIONES - USECASE. 64

FIGURA 31: PROMOCIONES - VIEWMODEL. 66

FIGURA 32: PROMOCIONES – APP – INITKOIN. 68

FIGURA 33: PROMOCIONES - ACTIVITY. 69

FIGURA 34: LECTURA DE FEATURE FLAG. 70

FIGURA 35: CONFIGURACIÓN DE SPLIT. 70

FIGURA 36: PROMOCIONES - FLUJO FINAL. 71

1. Introducción

El panorama actual en el sector tecnológico plantea a las empresas retos cada vez más específicos, especialmente en el terreno del desarrollo de aplicaciones móviles. En la búsqueda constante de soluciones innovadoras, las empresas se ven enfrentadas a desafíos en la contratación de desarrolladores experimentados, una dificultad especialmente notoria en plataformas específicas como iOS. Ante esta problemática, se propone incorporar un enfoque de desarrollo multiplataforma, con el objetivo de acelerar el proceso de desarrollo, optimizar la asignación de recursos y la implementación de la lógica empresarial.

De esta iniciativa nace la Práctica Profesional Supervisada (PPS) que se realizará en Ualá, empresa de tecnología que ofrece un ecosistema de soluciones financieras. Ualá fue creada en octubre de 2017 por el emprendedor argentino Pierpaolo Barbieri con el objetivo de impulsar la inclusión financiera en América Latina. Actualmente cuenta con más de 1.200 empleados distribuidos mayormente en Argentina, Colombia y México. A nivel organizativo, la división de la estructura en *streams* revela una evolución estratégica hacia una empresa multiproducto y multinacional. Un *stream* representa una cadena de valor que abarca desde la idea inicial hasta la ejecución de productos financieros. Guiado por un Business Owner y un equipo, establece prioridades estratégicas para cumplir con los Objetivos y Resultados Clave (*OKRs*). Este enfoque tiene el propósito de impulsar la colaboración y el alineamiento, al mismo tiempo que permite a los diferentes productos financieros operar de manera autónoma y ágil. Al minimizar las dependencias externas, cada *stream* de productos puede tomar decisiones más informadas y rápidas, posibilitando una mayor agilidad en la generación de valor.

La organización en cinco *streams* responde a la realidad actual de la organización y apunta a establecer una organización inicial. Sin embargo, puede iterar hacia otros tipos de estructuras para adaptarse a la evolución del negocio:

- **Stream Credit**: Incluye productos crediticios como préstamos.
- **Stream Payments**: Abarca productos transaccionales como transferencias y tarjetas.
- **Stream Bis**: Enfatiza productos de aquerencia como link de pago y mPOS.
- **Stream Wealth Management**: Se encarga de productos de inversión como fondos comunes y compra/venta de moneda extranjera.
- **Stream Apps Ecosystems**: Centra su enfoque en funciones transversales como Onboarding, Home, CX, entre otros.

El desarrollo de la Práctica Profesional se realizará en el equipo de Platform dentro del stream Apps Ecosystems, conocido dentro de la empresa como "Core". El mismo tiene como propósito influir en el desarrollo y la implementación de las mejores prácticas dentro de la compañía, contribuyendo así a la resolución de los desafíos mencionados y aportando a la expansión y evolución de Ualá en el competitivo mercado tecnológico. Debido a la condición de desarrollo multiplataforma (iOS y Android), no existe un responsable exclusivo, por lo que se llevará a cabo en colaboración con los arquitectos de ambas tecnologías.

En el contexto actual, la empresa cuenta con aplicaciones nativas para iOS y Android. Sin embargo, esta configuración implica que la creación o modificación de una funcionalidad requiere la participación de al menos dos programadores que desarrollen, cada uno, el código necesario para su respectiva plataforma. Esta

duplicación de esfuerzos genera la paradoja de resolver un mismo problema con código separado.

La introducción de tecnologías multiplataforma surge como una solución estratégica, precisa, de directrices claras y decisiones bien definidas. Se debe considerar qué aspectos va a abarcar, si es adecuado o no que comparta una lógica de negocio, qué aspectos incluirá de la interfaz gráfica, entre otras consideraciones que pueden variar según la tecnología que se utilizará. Esta estrategia tiene como objetivo fundamental reducir los tiempos de desarrollo, la que, una vez implementada, se traducirá en una disminución de costos y tiempo invertido en el ciclo de desarrollo. Además, al requerir menos líneas de código, simplifica la estructura y, por ende, reduce la probabilidad de errores, lo que se traduce en una mayor eficiencia y mantenibilidad del producto final.

El presente trabajo expondrá el diseño, el desarrollo y la implementación de un módulo multiplataforma dentro de aplicaciones de uso masivo utilizadas por casi 5 millones de personas. Esta nueva estructura permitirá no solo reutilizar y mejorar la calidad del código, sino también maximizar la eficiencia del trabajo para los diversos equipos de desarrollo. Se abordarán diversos aspectos incluyendo la selección de las herramientas apropiadas, las estrategias de implementación propuestas, los desafíos encontrados, y las conclusiones y reflexiones derivadas de la experiencia de la práctica profesional.

2. Objetivos

Considerando la creciente demanda de profesionales con experiencia en el desarrollo de aplicaciones móviles y la necesidad de maximizar la eficiencia y el rendimiento de aplicaciones utilizadas por millones de usuarios, el objetivo de la Práctica Profesional Supervisada (PPS) es establecer un nuevo paradigma de desarrollo móvil en Ualá. Este paradigma se basará en un enfoque de desarrollo multiplataforma que permitirá desarrollar nuevas *features* de manera más eficiente y escalable, sin requerir un aumento significativo en recursos humanos e incluso con la disminución de estos. Por lo tanto, el propósito central de la PPS será implementar un módulo multiplataforma que pueda integrarse fácilmente en las aplicaciones nativas de iOS y Android. Esta integración buscará optimizar los costos, los tiempos de desarrollo y, a su vez, evitar la duplicación de esfuerzos al implementar nuevas funcionalidades en ambas plataformas. Con la implementación exitosa de este módulo, se sentarán las bases para futuras *features* y desarrollos, impulsando así el uso continuo de un *framework* multiplataforma en la organización.

Los propósitos específicos que se buscan alcanzar son los siguientes:

- Evaluar y seleccionar un adecuado *framework* o SDK de desarrollo multiplataforma que se ajuste a las metodologías de desarrollo de la empresa.
- Crear e integrar un módulo en las aplicaciones de ambos S.O., que realice solicitudes a *APIs* públicas como parte de una prueba de concepto (POC).
- Incorporar bibliotecas internas al módulo, como el sistema de autenticación, para obtener los *tokens* necesarios y conectar con el *backend* de Ualá.
- Desarrollar una *feature* con distintos tipos de peticiones al *backend* mediante autenticación provista por librería propia.

- Crear un flujo integral de CI/CD para optimizar el proceso de implementación del módulo multiplataforma.
- Realizar el despliegue del módulo en entornos productivos.

Pasos a seguir

a. Análisis y enfoques de desarrollo multiplataforma

En primer lugar, se llevará a cabo un exhaustivo análisis de la situación actual de las aplicaciones existentes, evaluando su arquitectura, funcionalidades y limitaciones. Además, se investigarán enfoques de desarrollo multiplataforma con el objetivo de encontrar la mejor estrategia para integrar el módulo en cuestión. Se estudiarán detenidamente las opciones más viables y populares, como *Kotlin Multiplatform, Flutter* y *React Native,* analizando las ventajas y desventajas de cada una en términos de rendimiento, escalabilidad, facilidad de implementación y mantenimiento, siempre teniendo en cuenta la problemática inicial y los recursos con los que cuenta la empresa.

b. Definición de alcance y tecnología

En una segunda etapa, se procederá a definir el alcance del módulo y seleccionar el *framework* más adecuado para su desarrollo. Teniendo en cuenta los resultados obtenidos en el análisis previo, se evaluarán cuidadosamente las necesidades y los requisitos específicos de las aplicaciones existentes en las plataformas Android e iOS. Se considerará la compatibilidad del módulo con ambas plataformas, así como la facilidad de implementación y su integración sin problemas. Además, se valorará la capacidad del *framework* seleccionado para satisfacer los objetivos de eficiencia, rendimiento y escalabilidad establecidos para el proyecto, por lo que el *framework* elegido debe permitir aprovechar al máximo las características y funcionalidades de cada plataforma, asegurando una experiencia de usuario fluida y consistente en ambas aplicaciones. También se tomará en cuenta la disponibilidad de recursos y

documentación de apoyo para garantizar un desarrollo eficiente y una fácil mantención a largo plazo.

c. Desarrollo e implementación del módulo

En esta etapa se dará inicio al desarrollo del módulo utilizando la tecnología elegida en la etapa anterior. Se seguirán rigurosamente las buenas prácticas de programación, haciendo uso de las herramientas y recursos necesarios para asegurar la calidad y eficiencia del código desarrollado. El objetivo principal es obtener un módulo funcional, modular y de alto rendimiento que pueda ser integrado de manera sencilla en las aplicaciones existentes. Durante el proceso de desarrollo, se prestará especial atención a la estructura del código, la separación de responsabilidades y la reutilización de componentes, con el propósito de lograr un diseño escalable y mantenible. Asimismo, se llevarán a cabo pruebas exhaustivas para verificar el correcto funcionamiento del módulo en ambos sistemas operativos, asegurando su compatibilidad y adecuado desempeño. Además, se documentará los procesos y decisiones tomadas durante el desarrollo, brindando un respaldo sólido para el futuro desarrollo de nuevos módulos y el mantenimiento de los mismos.

d. Verificación, integración y mantenimiento del módulo

Una vez concluido el desarrollo del módulo, se dará paso a la etapa de verificación, integración y mantenimiento del mismo en un entorno productivo. En esta fase, se llevarán a cabo pruebas exhaustivas para verificar el correcto funcionamiento del módulo en las aplicaciones Android e iOS existentes. Se realizarán pruebas de integración, asegurando que el módulo se integre de manera adecuada y no genere conflictos con las funcionalidades existentes de las aplicaciones. Asimismo, se realizarán pruebas de rendimiento y estabilidad para garantizar que el módulo

cumpla con los estándares de calidad establecidos. En resumen, esta etapa se enfoca en la verificación exhaustiva del módulo, su correcta integración en las aplicaciones existentes y el establecimiento de mecanismos de mantenimiento para garantizar su funcionamiento óptimo y su actualización constante.

e. Conclusiones y redacción del informe final

En esta última etapa, se llevará a cabo la recopilación y el análisis de los resultados obtenidos durante el desarrollo del proyecto. Se evaluará el cumplimiento de los objetivos planteados y se realizará una revisión exhaustiva de los aspectos destacados, lecciones aprendidas y posibles áreas de mejora. Sobre la base de estos análisis, se redactará el informe final de la PPS, el cual incluirá una introducción, una descripción detallada del proyecto, la metodología utilizada, los resultados obtenidos, las conclusiones y las recomendaciones para futuros trabajos relacionados, y servirá además como documentación y referencia para estos.

2.1 Plan de trabajo

En la figura 1 se establece el cronograma de actividades por desarrollar en relación con los tiempos estimados para cada actividad:

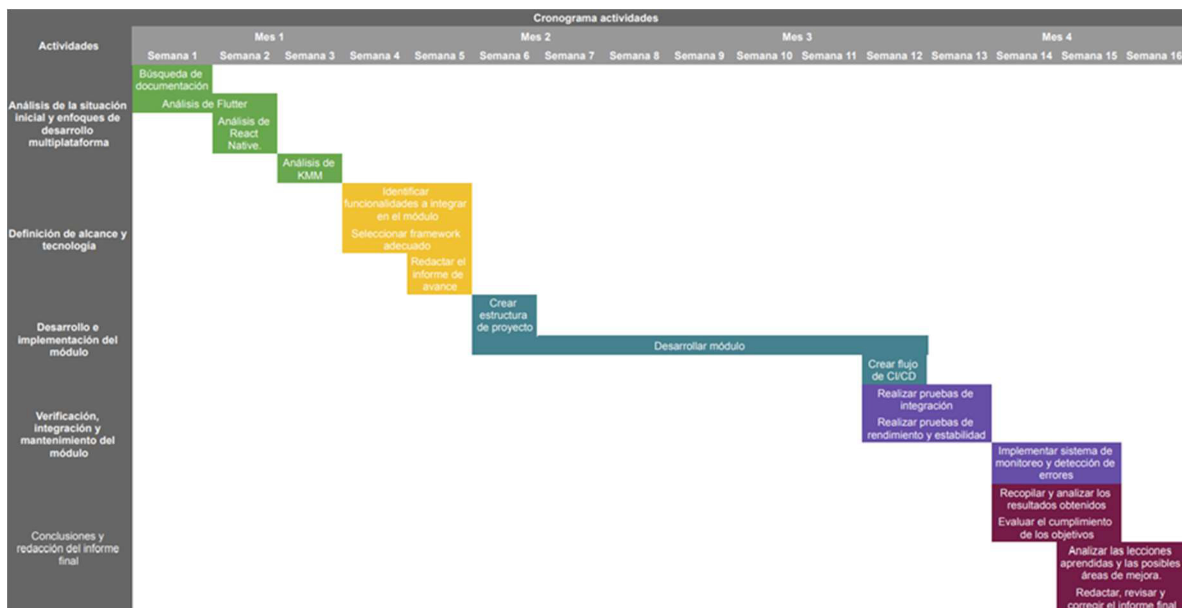


FIGURA 1: PLAN DE TRABAJO
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

2.2 Cronograma de trabajo

A continuación se presenta una tabla con las tareas, subtareas, etapas y duración estimada para llevar a cabo el proyecto. Esta planificación sirve como guía inicial para organizar y gestionar el avance del proyecto de manera eficiente, teniendo en cuenta que dichos plazos y duraciones son estimaciones iniciales y podrán ajustarse a medida que avance el proyecto:

Etapa	Tarea	Duración
Análisis de la situación inicial y enfoques de desarrollo multiplataforma	Búsqueda de documentación	3 semanas
	Análisis de *Flutter*	
	Análisis de *KMM*	
	Análisis de *React Native*.	
Definición del alcance del proyecto y de la tecnología a utilizar	Identificar funcionalidades a integrar en el módulo	2 semanas
	Seleccionar *framework* adecuado	
Desarrollo e implementación del módulo	Crear estructura del proyecto	12 semanas
	Desarrollar módulo	
	Crear flujo de CI/CD	
Verificación, integración y mantenimiento del módulo	Realizar pruebas de integración	2 semanas
	Realizar pruebas de rendimiento y estabilidad	

	Implementar sistema de monitoreo y detección de errores	
Conclusiones y redacción del informe final	Recopilar y analizar los resultados obtenidos	3 semanas
	Evaluar el cumplimiento de los objetivos	
	Analizar las lecciones aprendidas y las posibles áreas de mejora para futuros trabajos relacionados.	
	Redactar, revisar y corregir el informe final	

3. Marco teórico

El desarrollo de aplicaciones móviles abarca diferentes enfoques y estrategias que han evolucionado con el tiempo. En este capítulo, se explorarán los inicios y la evolución de las aplicaciones móviles, los conceptos fundamentales de desarrollo nativo, *cross-platform* y multiplataforma, así como también las ventajas y desventajas inherentes a cada enfoque.

3.1 Inicio de las aplicaciones móviles

Hacia finales de la década de 1990, surgieron las primeras aplicaciones móviles básicas, como la agenda, el calendario y los editores de tonos de llamada, marcando el inicio de las aplicaciones móviles. En ese período, también emergieron aplicaciones de entretenimiento, como los juegos *Tetris* y *Snake*, que evidenciaron el potencial del desarrollo de aplicaciones.

3.2 Evolución del desarrollo de apps móviles

El año 2007 marcó un hito en la evolución de las aplicaciones móviles con la presentación del iPhone por parte de Steve Jobs, seguido un año después por el lanzamiento de la App Store. Esto permitió a los programadores crear y comercializar sus desarrollos de manera accesible lo que abrió un mundo de posibilidades y oportunidades. Paralelamente, en respuesta a estas oportunidades de mercado, Google también se embarcó en el desarrollo de Android, un sistema operativo móvil que revolucionaría la industria. Android, presentado en 2008, inauguró una plataforma abierta y versátil que permitía la creación de aplicaciones para diversos dispositivos. Estas plataformas posibilitaron que el ecosistema de aplicaciones se expandiera a un ritmo sin precedentes.

3.3 Influencia de las Fintech

Las aplicaciones móviles han tenido un impacto significativo en el sector financiero a través de las *fintech*. Estas empresas han capitalizado la conveniencia y accesibilidad de las aplicaciones móviles para ofrecer servicios financieros innovadores, desde pagos móviles hasta inversiones y préstamos. Las *fintech* han cambiado la forma en que las personas gestionan sus finanzas y han contribuido al crecimiento del sector tecnológico.

3.4 Tendencias en el desarrollo de aplicaciones

El ámbito de las aplicaciones móviles sigue en auge, la tendencia apunta a una mayor humanización de la tecnología, con el uso extendido de asistentes virtuales y chatbots. La realidad aumentada se afianza como herramienta de enriquecimiento, mientras que la mejora de la experiencia de usuario y la seguridad siguen siendo prioridades. El pago móvil y la transformación del Internet de las Cosas hacia el Internet de las Personas también dibujan un panorama prometedor. Las aplicaciones se erigen como facilitadoras de tareas y herramientas de productividad en un mundo cada vez más móvil.

3.5 Desarrollo nativo

El enfoque de desarrollo nativo implica la creación de aplicaciones específicas para una plataforma particular, como iOS o Android. Estas aplicaciones se construyen utilizando los lenguajes de programación y las herramientas recomendadas por la plataforma. El enfoque nativo maximiza las características y capacidades únicas de cada sistema operativo, lo que puede derivar en una experiencia de usuario fluida y optimizada. No obstante, este enfoque conlleva la necesidad de escribir un código

separado para cada plataforma, lo que puede aumentar la complejidad y el tiempo requerido en el proceso de desarrollo.

3.5.1 Android

En el contexto del desarrollo nativo, Android ofrece ciertas ventajas notables para los desarrolladores. Debido a su naturaleza de código abierto, la programación en el entorno Android es altamente adaptable y modificable, permitiendo una mayor flexibilidad en la optimización de aplicaciones. Dentro de Android, se utilizan varios lenguajes de programación. A continuación, presentamos los más prevalentes en la actualidad:

3.5.1.1 Java

Históricamente, Android ha sido compatible con el lenguaje de programación *Java*, lo que lo convierte en el lenguaje por defecto para el desarrollo de aplicaciones en esta plataforma. Con una amplia aplicación y velocidad, Java ha mantenido su popularidad entre los programadores de Android.

3.5.1.2 Kotlin

Este lenguaje es un proyecto de código abierto (licencia Apache 2) desarrollado por *JetBrains*, empresa desarrolladora del *IDE IntelliJ IDEA*, entre otros, que se ha vuelto el preferido en la comunidad de desarrollo de Android. Sus ventajas incluyen un código claro e intuitivo sin sacrificar la eficiencia, la interoperabilidad con *Java* y la solución de problemas heredados por parte de *Java* como excepciones por punteros nulos. Basado en inspiraciones de lenguajes como *Scala* y *Java, Kotlin* ha mejorado la velocidad de compilación y otras características. Desde que Android lo adoptó como lenguaje principal en 2019 se ha convertido en una elección recomendada para nuevos proyectos; incluso, es obligatorio para proyectos en los

que se utilice el kit de herramientas para compilar UI nativas Android, *Jetpack Compose*.

3.5.2 IOS

Apple proporciona una plataforma única para crear aplicaciones que funcionan de manera óptima en sus dispositivos. Este enfoque implica la construcción de aplicaciones utilizando lenguajes de programación y herramientas respaldados por Apple.

3.5.2.1 Swift

Uno de los puntos distintivos del desarrollo nativo en iOS es el lenguaje de programación *Swift*, desarrollado por Apple, que se ha convertido en la opción principal para la creación de aplicaciones en esta plataforma debido a sus características, que les otorga a los desarrolladores una vía eficiente y efectiva para crear aplicaciones de alta calidad y rendimiento para dispositivos Apple.

3.6 Desarrollo Cross-Platform

El enfoque de desarrollo *cross-platform* implica la creación de aplicaciones que pueden ser implementadas en múltiples sistemas operativos mediante el uso de tecnologías compartidas. Esta metodología busca optimizar los recursos al compartir una base de código entre distintas plataformas, lo que puede agilizar los procesos de desarrollo y reducir costos. No obstante, es importante considerar que puede haber ciertas limitaciones en términos de rendimiento y acceso a características específicas de cada plataforma.

3.6.1 Xamarin

Es una plataforma originada por los desarrolladores de Mono y posteriormente adquirida por Microsoft; adopta el lenguaje C# para crear aplicaciones nativas con un nivel de rendimiento destacado. Sin embargo, es esencial mencionar que la creación de la interfaz de usuario podría demandar un mayor esfuerzo de desarrollo y la comunidad de usuarios no es tan activa en comparación con otros entornos.

3.6.2 React Native

Otra tecnología prominente en el desarrollo *cross-platform* es *React Native*, impulsada por Facebook, que se basa en el uso de *JavaScript* o *TypeScript* y permite la creación de aplicaciones móviles que comparten una base de código entre sistemas operativos como iOS y Android. Si bien tecnologías anteriores como *Phonegap* o *Cordova* convertían páginas web en aplicaciones móviles, *React Native* adopta un enfoque distinto. No recurre a la simple carga de una página web en una aplicación, sino que utiliza *JS* para convertir componentes nativos en una aplicación. Este enfoque elude las limitaciones que surgían al cargar una página web en una aplicación móvil, proporcionando una experiencia más nativa y fluida.

3.7 Desarrollo Multiplataforma

El desarrollo multiplataforma es una variante avanzada del enfoque *cross-platform*, cuyo objetivo principal es crear aplicaciones que puedan operar en múltiples plataformas manteniendo un alto grado de código compartido. Este enfoque busca maximizar la reutilización de código y garantizar una experiencia de usuario coherente a lo largo de diferentes sistemas operativos. Utilizando tecnologías como *Flutter* y *Kotlin Multiplatform*, el desarrollo multiplataforma puede lograr resultados que se asemejan a aplicaciones nativas, tanto en apariencia como en rendimiento, al mismo tiempo que se aprovechan las características específicas de cada plataforma.

3.7.1 Flutter

Es un *framework* de código abierto respaldado por Google. Destinado tanto a desarrolladores de *frontend* como *fullstack*, permite la creación de interfaces de usuario (UI) de aplicaciones para múltiples plataformas con un único conjunto de código. Una característica distintiva de *Flutter* es su capacidad para generar rápidamente funcionalidades con un enfoque en la experiencia de usuario nativa mediante el uso de *widgets* que encapsulan las diferencias críticas entre plataformas, como el desplazamiento, la navegación, los íconos y las fuentes, lo que proporciona un rendimiento completamente nativo tanto en iOS como en Android. Aunque *Flutter* se introdujo inicialmente en 2018 como una solución principalmente para aplicaciones móviles, ha evolucionado para admitir el desarrollo de aplicaciones en múltiples plataformas, incluidas iOS, Android, web, Windows, MacOS y Linux.

3.7.2 Kotlin Multiplatform

Kotlin Multiplatform (KMP) está diseñado para simplificar el desarrollo de proyectos multiplataforma. Esta tecnología reduce el tiempo dedicado a escribir y mantener el mismo código para diferentes plataformas, al tiempo que conserva la flexibilidad y los beneficios de la programación nativa. KMP es un conjunto de herramientas de desarrollo que permiten ejecutar *Kotlin* en varias plataformas. Entre estas herramientas, se encuentra la interoperabilidad del lenguaje *Kotlin* con los lenguajes de otras plataformas compatibles, lo cual permite crear bibliotecas en el mismo formato que el lenguaje de destino. Por ejemplo, los desarrolladores de iOS suelen utilizar bibliotecas en formato *.framework* escritas en *Swift* u *Objective-C* por lo que pueden utilizarlo de la misma manera que cualquier otra biblioteca, sin siquiera saber que fue escrita en *Kotlin*. Otra innovación de *Kotlin Multiplatform* es la posibilidad de tener diferentes implementaciones de algunas funciones, según la plataforma en la que se ejecute el programa. Para solucionar este problema, KMP propone escribir la función varias veces (una para cada plataforma) y elegir la correcta al compilar el programa.

3.8 Herramientas

A continuación se detallan brevemente las herramientas más importantes utilizadas para el desarrollo del proyecto:

3.8.1 Gradle

Gradle es una herramienta de construcción de código abierto que se destaca por su rapidez, confiabilidad y versatilidad. Su lenguaje de construcción declarativo facilita la lectura y escritura de la lógica de construcción, lo que aumenta la eficiencia del desarrollo. Además, es ampliamente compatible con una variedad de lenguajes y marcos, incluyendo Android, *Java, Kotlin Multiplataforma y C/C++,* fundamentales para el desarrollo del presente trabajo. A través de su amplia comunidad de desarrolladores y su enfoque en la automatización, se ha convertido en la elección predeterminada para proyectos Android, y es conocido por su velocidad, capacidad de manejar proyectos de cualquier tamaño y su confiabilidad, respaldada por características como construcciones incrementales y almacenamiento en caché. *Gradle* ofrece una herramienta adicional llamada *Build Scan* que proporciona información detallada para identificar problemas y depurar construcciones: esto, sumado a su compatibilidad con las principales IDE y la línea de comandos, lo han convertido en una herramienta esencial para la gestión de proyectos de desarrollo de software.

3.8.2 Maven Publish Plugin

El Plugin de Publicación de Maven brinda la capacidad de publicar artefactos de construcción en un repositorio Apache Maven, de esta manera el módulo puede ser consumido por Gradle. Utiliza una extensión en el proyecto denominada *publishing* de tipo *PublishingExtension* que proporciona un contenedor de publicaciones

nombradas y un contenedor de repositorios nombrados. El complemento de *Gradle* para Android genera un componente para cada variante de construcción en la aplicación o módulo de biblioteca.

3.8.3 JFrog

JFrog Artifactory es una solución *DevOps* versátil que automatiza y gestiona binarios y artefactos en todo el proceso de entrega de aplicaciones, mejorando la productividad en el ecosistema de desarrollo. Entre sus características se destacan la compatibilidad con múltiples entornos, una gestión de repositorio universal que admite una amplia variedad de formatos, una alta escalabilidad y opciones de replicación, y una integración sin problemas con herramientas de construcción. Además, ofrece una interfaz *API* personalizable y una potente función de búsqueda. La solución en la nube de *Artifactory* también facilita la distribución de software en múltiples ubicaciones, lo que simplifica la gestión y el control.

3.8.4 GitHub Actions

GitHub Actions es una plataforma de integración y entrega continua (*CI/CD*) para automatizar procesos de construcción, prueba e implementación. Permite crear flujos de trabajo que se activan por eventos en un repositorio, como solicitudes de extracción o creación de problemas. Estos flujos contienen trabajos que pueden ejecutarse en secuencia o en paralelo, cada uno en su propia máquina virtual o contenedor, y se dividen en pasos que pueden ser *scripts* personalizados o acciones reutilizables. *GitHub Actions* proporciona máquinas virtuales Linux, y MacOS, plataformas necesarias para la compilación de los módulos dedicados a iOS y Android. Permite además utilizar *runners* hosteados en su propia infraestructura y cada uno de estos flujos se ejecuta en su propio entorno de máquina virtual, o dentro

de un contenedor, y consta de uno o más pasos que ejecutan un *script* personalizado reutilizable que puede simplificar sus flujos de trabajo.

3.8.5 CocoaPods

Es un gestor de dependencias muy utilizado en el ecosistema de desarrollo de iOS que simplifica la administración de bibliotecas de terceros. Mediante su archivo de configuración *Podfile*, permite especificar y controlar dependencias de manera eficiente. Con una comunidad activa y una sólida resolución de dependencias, garantiza que las bibliotecas sean compatibles entre sí. Esto simplifica la actualización de bibliotecas y fomenta un desarrollo iOS más rápido y confiable, además de ofrecer una estrecha integración con *Xcode*. En resumen, *CocoaPods* es esencial para mantener proyectos iOS actualizados y optimizar la gestión de dependencias, mejorando la eficiencia y la calidad del código.

3.8.6 Split

Split proporciona todas las herramientas necesarias para crear, dirigir y gestionar *flags* de características (*feature flags*). Además, supervisa el estado de cada implementación para que los devs puedan enfocarse en seguir construyendo productos al separar la implementación del lanzamiento. Los *flags feature* permiten controlar de forma remota si se ejecuta o no, sin necesidad de una nueva implementación de código ni lanzamientos. La segmentación individual permite realizar pruebas de manera segura en producción, liberando cambios a equipos internos, testers de QA o clientes beta, antes de exponerlos al resto de usuarios. Ampliar gradualmente el lanzamiento a un subconjunto de usuarios, los cuales pueden ser segmentados por porcentaje o algún atributo, como puede ser ubicación, tipo de cuenta o antigüedad de la cuenta.

4. Desarrollo

Basándonos en los conceptos descritos en el capítulo anterior, este apartado abordará en detalle el proceso de desarrollo llevado a cabo en el proyecto. Se examinarán las etapas de implementación, profundizando en los temas previamente mencionados, y se proporcionará una guía paso a paso de las tareas ejecutadas para alcanzar los objetivos establecidos. Además, se presentará una descripción exhaustiva de las herramientas empleadas, su integración y aplicación en el contexto de este proyecto.

4.1 Situación inicial

Al inicio de este proyecto, Ualá contaba con una aplicación móvil con la cual un usuario puede acceder a un ecosistema financiero digital. Para abarcar el mayor porcentaje de mercado posible, esta App se encuentra desarrollada nativamente para dispositivos iOS y Android, lo que requiere experiencia en *Swift* y *Kotlin*, lenguajes elegidos para cada versión. En una primera instancia, se realizó una encuesta entre los desarrolladores para corroborar sus conocimientos técnicos sobre otros lenguajes de programación. Las alternativas a los lenguajes mencionados fueron *JavaScript/TypeScript* y *Dart*, necesarios para el desarrollo en *React Native* y *Flutter*, respectivamente. El resultado de esta encuesta se refleja en las figuras 2 y 3:

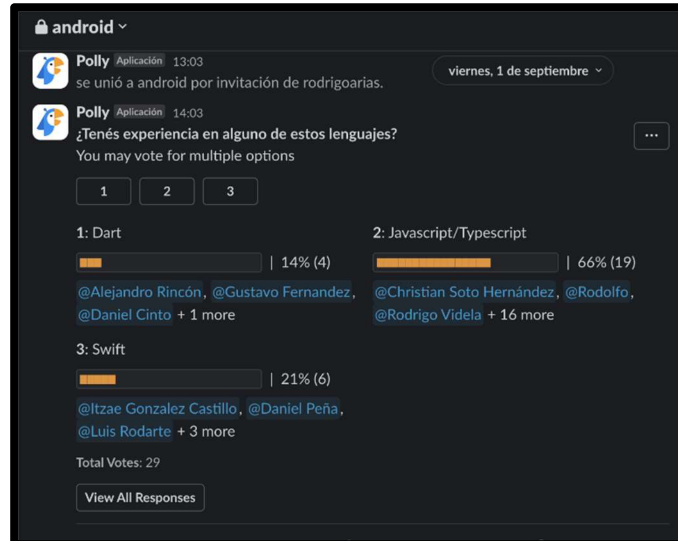


FIGURA 2: ENCUESTA EN EL CANAL ANDROID.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

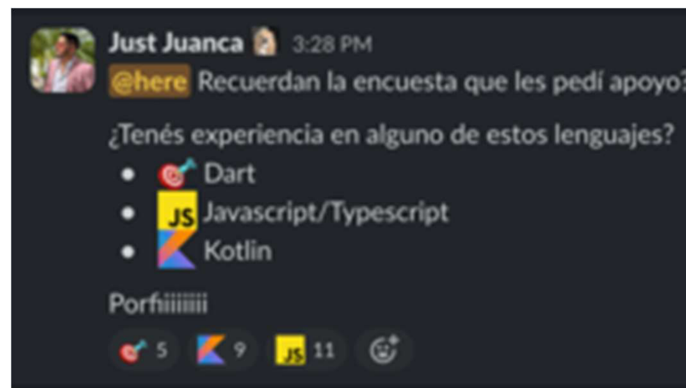


FIGURA 3: ENCUESTA EN CANAL IOS.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Firma Estudiante:	Firma tutor UNAJ:	Firma tutor TAPTA UNAJ:	Firma tutor Empresa:

Estos resultados brindaron una ventaja para el desarrollo multiplataforma con opciones como *React Native o KMP*, pero no fue el único punto evaluado al momento de decidir qué lenguaje y *framework* usar para el proyecto.

4.2 Análisis de tecnologías

Al comenzar la investigación sobre las tecnologías a utilizar, se tomó nota de que Microsoft había finalizado el soporte de *Xamarin* para dar comienzo al desarrollo *cross-platform* mediante *.NET6*. Por lo tanto, se descartó y se inició el análisis con tres posibles tecnologías: *React Native, Flutter* y *KMP*. La decisión tomó en cuenta la mayor cantidad posible de aspectos, algunos de los cuales fueron:

4.2.1 Lenguaje

Kotlin es un lenguaje de programación moderno y conciso que se ha convertido en el lenguaje oficial de desarrollo en Android, respaldado por Google. Una de las ventajas más notables es que ofrece características avanzadas que simplifican el código, como funciones de orden superior, clases de datos e inferencia de tipos. En lo que respecta a Ko*tlin Multiplatform*, en lugar de depender completamente de abstracciones, como sucede con algunas soluciones de desarrollo multiplataforma, permite escribir código específico para cada plataforma cuando sea necesario, brindando mayor control y optimización.
Dart, por otro lado, se destaca por su capacidad de compilación *Just-in-Time* (JIT) y *Ahead-of-Time* (AOT), lo que contribuye a una rápida carga y renderización de la interfaz de usuario en aplicaciones *Flutter*. La sintaxis de *Dart* es similar a la de otros lenguajes populares, lo que facilita su adopción por parte de los desarrolladores.
Por último, *JavaScript* es el lenguaje principal de *React Native (RN)*. En términos simples, *RN* actúa como un puente entre la plataforma de destino y *JS*. Esto permite

que la lógica de la aplicación se comunique con los componentes nativos del dispositivo. Sin embargo, a diferencia de *Kotlin y Dart*, no es un lenguaje nativo para ninguna de las plataformas móviles, lo que puede requerir una capa adicional de abstracción y puede afectar el rendimiento en comparación con soluciones completamente nativas.

4.2.2 Rendimiento

Flutter utiliza su propio motor de renderizado llamado *Skia*, que permite renderizar gráficos de alta calidad y realizar animaciones suaves. No requiere un puente entre el código y los componentes nativos, lo que puede resultar en mejor velocidad y rendimiento en comparación con React Native. Además, Flutter puede trabajar con imágenes pesadas y ofrece un alto rendimiento, incluso en aplicaciones con gráficos intensivos.

React Native opera a través de un puente que comunica el código *JavaScript* con los componentes nativos. Este puente puede causar cierta latencia y afectar el rendimiento de la aplicación, especialmente en comparación con tecnologías que no requieren puentes.

Kotlin Multiplatform (KMP) busca la eficiencia al permitir que partes específicas del código sean compartidas entre plataformas. Esto puede conducir a un rendimiento más optimizado, ya que el código compartido puede ser altamente optimizado para cada plataforma en particular. Sin embargo, la eficiencia exacta dependerá de cómo se implemente y optimice el código en cada plataforma específica.

4.2.3 Interfaz de Usuario (UI)

En cuanto a la interfaz de usuario, *Flutter* utiliza el *canvas* del *SDK* nativo de las diferentes plataformas y dibuja sus propios componentes de interfaz en ese lienzo, siguiendo las especificaciones de *Material Design*. Además, ofrece un conjunto

amplio de *widgets* personalizables que proporcionan una apariencia nativa a la aplicación. Estos *widgets* pueden ser ajustados y personalizados para satisfacer las necesidades específicas de cada aplicación.

React Native utiliza componentes nativos que se asignan al código JS y proporciona un conjunto de componentes de interfaz de usuario predefinidos, aunque no son tan adaptables como los *widgets* de *Flutter*.

Aunque existe una versión *Alpha* que permite la creación de una interfaz de usuario compartida mediante el uso de *Jetpack Compose*, el equipo de *Platform* desaconseja su implementación en etapas tan tempranas de pruebas. Por lo tanto, estas características de *KMP* no se consideran hasta que alcancen un nivel de estabilidad adecuado y, en caso de ser elegido, seguirá siendo necesario escribir el código de UI nativo por separado para Android e iOS.

4.2.4 Otros aspectos

La tecnología impulsada por JetBrains brinda la capacidad de integrarse con cualquier proyecto existente, permitiendo flexibilidad en la incorporación de sus funcionalidades. En contraste, en el caso de *Flutter* o *React Native*, es necesario seguir utilizando su propia infraestructura y estructura de proyecto preestablecida.

En cuanto al tamaño de la aplicación, un primer ejemplo de "Hola mundo" en *KMP* agrega menos de medio megabyte a una compilación de iOS, gracias a la flexibilidad de compartir solo el código deseado. Cabe señalar que en Android, el tamaño agregado es cero, ya que las aplicaciones de Android ya utilizan Kotlin. Por otro lado, *Flutter* agrega casi 5MB en ambas plataformas, esto se debe a que incluye su *framework UI*.

4.2.5 Conclusiones

En primer lugar, el profundo conocimiento que nuestro equipo de desarrolladores tiene Del lenguaje *Kotlin* fue un factor determinante. Esta familiaridad con el lenguaje permitió que la curva de aprendizaje fuera más suave y la sintaxis de *Kotlin,* fácil de aprender, incluso para desarrolladores de iOS, ya que sigue conceptos similares a *Swift*. Además, ya se ha utilizado durante años en aplicaciones grandes y en producción, lo que significa que nuestro código base está listo para ser compartido de manera eficiente entre múltiples plataformas.

Para el caso de nuevas aplicaciones, *Flutter* ofrece un impresionante y poderoso marco *cross-platform*, pero es esencial tener en cuenta las limitaciones respecto a la integración con aplicaciones existentes y el peso que añade al agregar todo el *framework UI,* a diferencia de *Kotlin*, que no agrega un *SDK* en ninguno de los dos Sistemas Operativos.

La facilidad de integración de *KMP* con aplicaciones existentes fue otro factor clave. La capacidad de elegir qué aspectos del código compartir, como las reglas de negocio o *Business Logic*, junto con la posibilidad de reutilizar bibliotecas de componentes UI existentes, nos brindó una ventaja significativa. En contraste, tanto *Flutter* como *RN* requerían la creación de un sistema de diseño desde cero, lo que habría implicado un mayor esfuerzo y tiempo.

Por último, durante nuestro análisis, hemos observado que en el *roadmap* de *Kotlin* se encuentra la planificación de lanzar una versión estable de *Compose Multiplatform*. Este *framework*, basado en *Kotlin* y *Jetpack Compose*, permitiría la reutilización del *Design System* que hoy en día se utiliza en el desarrollo de nuevas *features* dentro del ecosistema Android de Ualá. Dado que también está desarrollado mediante *Compose* y es compatible con esta tecnología, su estabilidad sería un avance importante para facilitar la creación de interfaces de usuario

compartidas en múltiples plataformas. Aun cuando el punto fuerte a desarrollar sea compartir sólo la lógica de negocio, este punto es un plus que vuelve más interesante la elección de *Kotlin Multiplatform* para nuestro desarrollo.

4.3 Módulo de Prueba

Para iniciar el desarrollo multiplataforma, se decidió comenzar con un pequeño módulo que contenga un método para obtener la versión del Sistema Operativo. La intención es integrar este módulo en las aplicaciones de iOS y Android para validar su comportamiento y peso, permitiéndonos iterar para incorporar funcionalidades adicionales, como la realización de peticiones al *backend* y el uso de librerías propias de Ualá, como las relacionadas con la autenticación del usuario, esenciales para la comunicación entre las apps y *backend* a través de peticiones *HTTP*.

4.3.1 Get platform

Para comenzar con la creación del módulo a iterar, se creó un repositorio privado en la cuenta de *Github* de Ualá, lo que permitirá, en el futuro, implementar los flujos de trabajo necesarios para las pruebas de CI/CD. Una vez creado, se realizó la clonación del repositorio localmente y procedimos a crear nuestra aplicación multiplataforma siguiendo los pasos proporcionados en la documentación oficial de *Kotlin Multiplatform*. Se configuró la app ingresando la información requerida sobre el proyecto, como su nombre, ubicación y el SDK mínimo a utilizar para Android. Luego, continuamos con la configuración de la distribución de las librerías o módulos. En el caso de Ualá, se utilizó *CocoaPods* como gestor de dependencias, lo que permitió mantener el ecosistema iOS con cambios mínimos para facilitar la transferencia de conocimientos a los desarrolladores de iOS en el futuro.

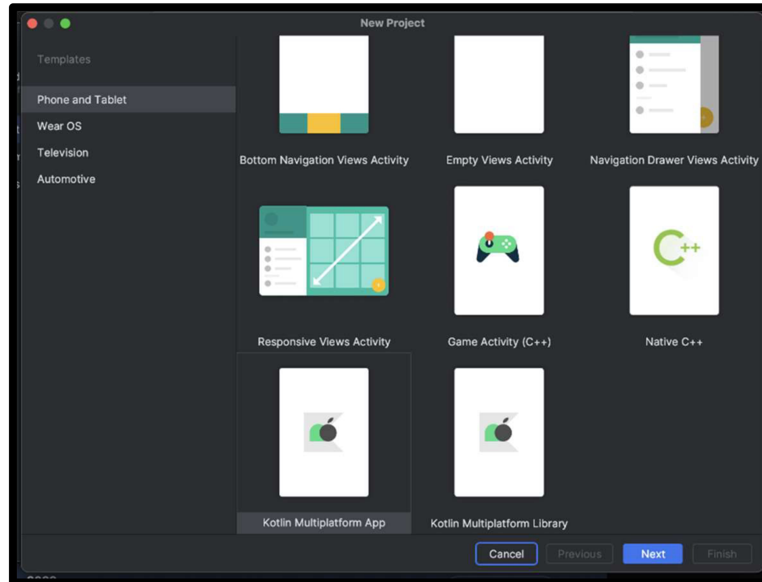


FIGURA 4: WIZARD NEW PROJECT.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

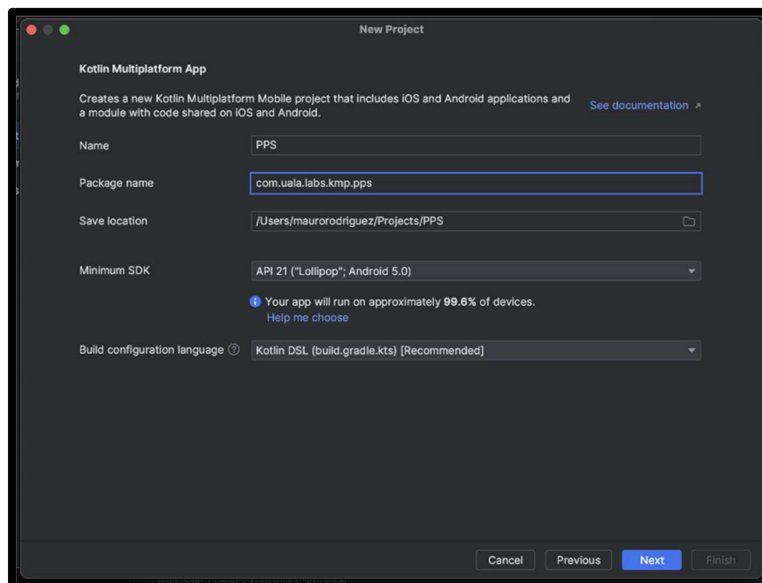


FIGURA 5: WIZARD KMP PROJECT.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

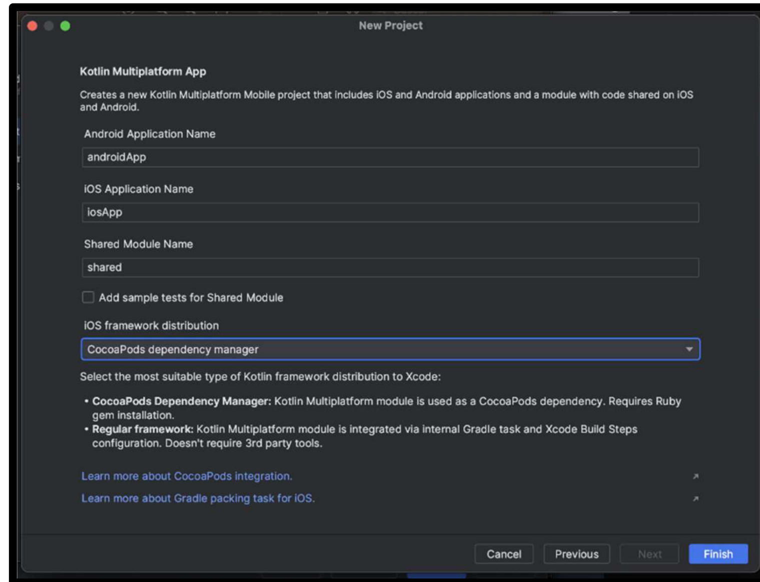


FIGURA 6: WIZARD KMP APP.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Al finalizar la configuración inicial del proyecto, se observó que el proyecto contaba con tres componentes distintos. Por un lado, dos aplicaciones nativas: una para Android y otra para iOS. Ambas aplicaciones consumen un módulo llamado *shared*, el mismo contiene toda la lógica de negocio que iremos iterando para ser consumida por estas aplicaciones que funcionarán como Apps demo. Una vez que estas demos funcionen correctamente, se generarán las versiones locales de este módulo para ser utilizadas por las aplicaciones originales de Ualá. Dentro del módulo *shared* se encuentra la carpeta *commonMain*, donde se creará la lógica de negocio abstracta del sistema operativo a implementar. En caso de necesitar distintas implementaciones de un mismo método, se dispone de carpetas separadas para cada plataforma, donde especificaremos cómo deberá ajustarse el comportamiento según los requisitos y condiciones de cada una, en este caso: *androidMain* e *iosMain*.

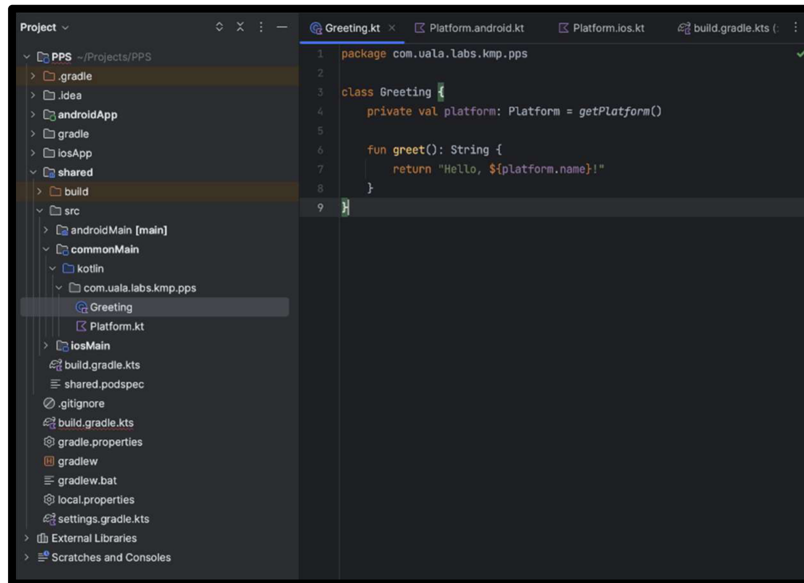


FIGURA 7: PROYECTO INICIAL
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Para realizar la primera prueba, se ejecutaron las demos de cada sistema operativo en su respectivo *IDE.* Al obtener los resultados esperados, procedimos a integrar el módulo con las aplicaciones de Ualá. Para esto, ejecutamos el método *assemble* que genera la versión del módulo *shared* para cada plataforma. En el caso de Android, se generó un archivo con extensión .*aar* que importamos mediante Android Studio. En cuanto a iOS, se creó un paquete llamado *Pod,* que se instala mediante *CocoaPods* y se consume en el *IDE* principal para desarrollos iOS, como *XCode*. Al ejecutar cada aplicación, verificamos que los resultados siguieran siendo exitosos, por lo que consideramos que la integración de los módulos, en principio, funciona sin complicaciones adicionales.

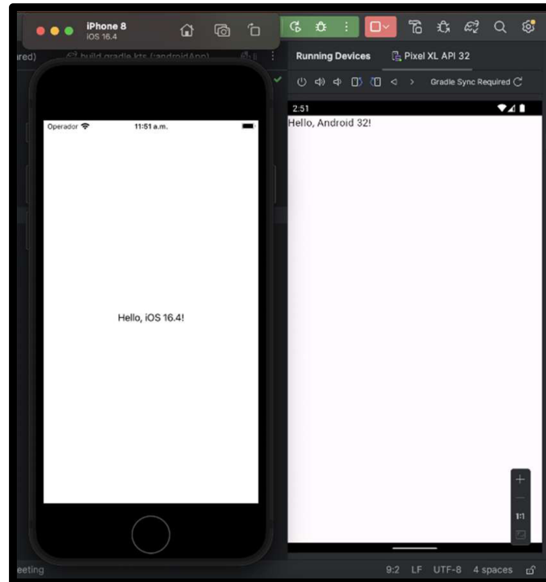


FIGURA 8: EJECUCIÓN PROYECTO INICIAL.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

4.3.2 Dependencias *Multiplatform*

En una segunda iteración, el objetivo fue agregar un método que permitiera consumir una API pública, lo que ayudaría a probar cómo se comporta la tecnología desarrollada con las aplicaciones integradas. A partir de este punto, dichas apps coexistirían con bibliotecas nativas y Multiplataforma para realizar solicitudes HTTP. Para lograrlo, incorporamos bibliotecas recomendadas por Kotlin Multiplatform (KMP), que incluyen:

- **Ktor**: Es un cliente HTTP utilizado para la comunicación con *APIs* y el *backend* de Ualá.
- **Kotlinx.coroutines**: Facilita la escritura de código que se ejecuta de manera asincrónica.

- **Kotlinx.serialization**: Permite mapear las respuestas obtenidas de servicios consumidos mediante *Ktor* en formato *JSON* y convertirlas en objetos manipulables con facilidad usando *Kotlin*.

```
sourceSets {
    val ktorVersion = "2.0.1"
    val koinVersion = "3.2.0"
    val serializationVersion = "1.3.3"

    val commonMain by getting {
        dependencies {
            // Ktor
            implementation("io.ktor:ktor-client-core:$ktorVersion")
            implementation("io.ktor:ktor-client-logging:$ktorVersion")
            implementation("io.ktor:ktor-client-content-negotiation:$ktorVersion")
            implementation("io.ktor:ktor-serialization-kotlinx-json:$ktorVersion")

            // Serialization
            implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:$serializationVersion")

            // koin
            implementation("io.insert-koin:koin-core:$koinVersion")

            implementation(coreCatalogue.kotlinxCoroutinesCore)

        }
    }
```

FIGURA 9: DEPENDENCIAS MULTIPLATAFORMA.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Una vez que sincronizamos el proyecto mediante *Gradle*, comenzamos a desarrollar una arquitectura mínima que nos permitirá aplicar buenas prácticas de desarrollo, como la separación de funciones y la inversión de dependencias, entre otras. Después de crear el modelo de datos que obtuvimos de la API pública seleccionada, *SpaceXApi*, se comenzó a crear los servicios y repositorios dentro del módulo shared, lo que permitió consumir esa información desde cada plataforma.

Para recolectar la información de forma asincrónica en el ecosistema Android, se optó por utilizar *Flow*, una librería propia de *Kotlin* que suele utilizarse en el desarrollo nativo Android. Para mantener un estándar, se creó una clase abstracta que permitió especificar cómo se maneja el contexto de las corrutinas en cada

plataforma. De esta manera, comenzamos a utilizar las carpetas específicas para la implementación nativa dentro de nuestro módulo Multiplataforma: *iosMain* y *androidMain.* En *commonMain* creamos una clase abstracta que contiene un *mainDispatcher* una clase particular de KMP denominada *expect*. Esta clase nos obligó a crear la implementación específica de esta clase mediante el uso de *actual class* dentro de cada plataforma. Por lo tanto, creamos, respetando la estructura de carpetas, una clase *Dispatcher* para cada sistema operativo e indicamos cómo se implementaría respetando las especificaciones nativas. En el caso de Android, se utilizó el *Dispatchers* de corrutinas *Main* mientras que iOS utiliza *MainLoopDispatcher.* Esto representa la primera diferencia en la implementación de una biblioteca Multiplataforma dentro de nuestro módulo *shared*.

Para realizar las pruebas se repitió el proceso anterior: primero se validaron los cambios utilizando las demos que se encuentran dentro del proyecto *KMP*. Una vez que se realiza la prueba y se obtienen resultados exitosos, se procede a integrar esta segunda iteración en las aplicaciones nativas visualizando los resultados en el emulador Android y el simulador iOS.

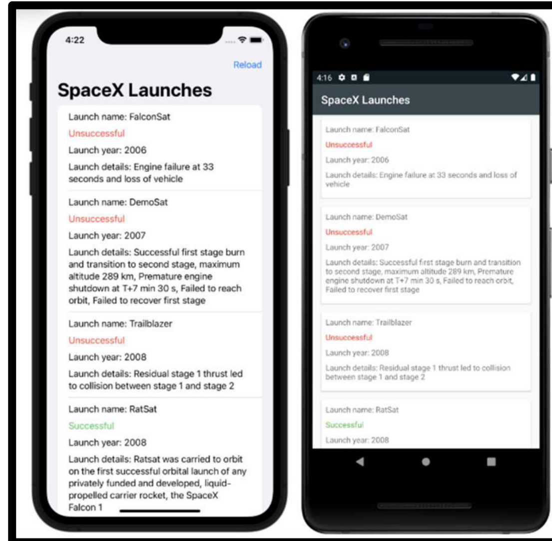


FIGURA 10: EJECUCIÓN CONSUMO SPACEXAPI.
FUENTE: WWW.JETBRAINS.COM/HELP/KOTLIN-MULTIPLATFORM-DEV/MULTIPLATFORM-KTOR-SQLDELIGHT

Dado el éxito de los resultados y a fin de facilitar desarrollos futuros, decidimos agregar una nueva dependencia: *Koin*. Esta biblioteca, desarrollada en *Kotlin*, es compatible con proyectos Multiplataforma y proporciona una manera fácil y eficiente de incorporar inyección de dependencias en nuestras aplicaciones. Se agregó entonces la dependencia mediante *Gradle*, utilizando nuevamente el sector de dependencias Multiplataforma en el archivo *build.gradle*. Sincronizado el proyecto, se continuó con la configuración de *Koin* dentro del módulo *shared* y *commonMain*, ya que la lógica se utilizaría para inyectar dependencias en ambos S.O.: en este caso, el repositorio correspondiente al consumo de datos desde *SpaceXApi*. Se concluyó la configuración inicial de los módulos a inyectar y, una vez completado el flujo, iniciamos *Koin* desde cada aplicación. En una primera iteración destinada a la inyección de dependencias, modificamos los archivos que devuelve el sistema operativo de cada aplicación e implementamos las facilidades que *Koin* ofrece. Creamos los módulos y componentes siguiendo la documentación oficial del

inyector de dependencias multiplataforma y, una vez finalizado, realizamos las pruebas correspondientes.

```
// platform Module
val platformModule = module {
    singleOf(::Platform)
}

// KMP Class Definition
expect class Platform() {
    val name: String
}

// iOS
actual class Platform actual constructor() {
    actual val name: String =
        UIDevice.currentDevice.systemName() + " " + UIDevice.currentDevice.systemVersion
}

// Android
actual class Platform actual constructor() {
    actual val name: String = "Android ${android.os.Build.VERSION.SDK_INT}"
}
```

FIGURA 11: EXPECT Y ACTUAL CLASS.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Con los resultados a favor, continuamos con el refactor del flujo correspondiente a las peticiones mediante *Ktor* a la *API* antes mencionada y realizamos las pruebas correspondientes. Con éxito en el proceso, culminamos la segunda iteración del módulo *Shared*.

4.3.3 Dependencias nativas

La siguiente iteración en el módulo de pruebas consistió en realizar consultas al *backend* propio de Ualá. En este caso, se busca obtener los cupones canjeados por un usuario en el programa de beneficios denominado Ualá+. Por cuestiones de seguridad, todas las *API* propias de Ualá utilizan un sistema de autenticación basado en diversos tipos de *tokens*. Antes de realizar las peticiones, debemos integrar el submódulo de *Core* que tiene como una de sus responsabilidades más

importantes la generación y validación de *tokens*, llamado *Authorization*. Este submódulo cuenta con su versión nativa tanto para iOS como para Android. A diferencia de las dependencias anteriores, esta debe integrarse siguiendo las pautas específicas de cada plataforma.

En el caso de Android, se mantuvo el uso de *Gradle* y *Maven* para identificar la librería a integrar. Dado que forma parte de un repositorio privado, antes se deben configurar las credenciales que permitirán, al sincronizar el proyecto, obtener las dependencias nativas de Android. Estas dependencias facilitan la obtención del *token* que se utiliza al realizar peticiones al *backend* de Ualá.

```
sourceSets { this: NamedDomainObjectContainer<KotlinSourceSet>
    val commonMain by getting {...}
    val commonTest by getting {...}
    val androidMain by getting { this: KotlinSourceSet!
        dependencies { this: KotlinDependencyHandler
            implementation( dependencyNotation: "io.ktor:ktor-client-okhttp:$ktorVersion")

            //Uala
            implementation( dependencyNotation: "uala-android-authorization:authorization:$ualaAuthorizationVersion")
            implementation( dependencyNotation: "uala-android-authorization:device-id-provider:$ualaDeviceProviderVersion")
            implementation( dependencyNotation: "uala-android-core:core:$ualaCoreVersion")
            implementation( dependencyNotation: "uala-android-localization:localization:$ualaLocalizationVersion")
            implementation( dependencyNotation: "uala-android-sign-in:sign-in:$ualaSignInVersion")

            //Firebase
            implementation(coreCatalogue.firebaseBOM)
            implementation(coreCatalogue.firebaseMessaging)
            implementation(coreCatalogue.firebaseAnalytics)
            implementation(coreCatalogue.firebaseCrashlytics)
            implementation(coreCatalogue.firebasePerf)
            implementation(coreCatalogue.firebaseConfig)
        }
    }
```

FIGURA 12: DEPENDENCIAS ANDROID.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Por otro lado, en iOS, el gestor de dependencias es *CocoaPods*, y las credenciales se gestionan mediante SSH. Esta configuración es propia del ecosistema de desarrollo de iOS, lo que hizo que la configuración fuera compleja en los primeros intentos de sincronizar el proyecto para validar la integración de dependencias.

```
kotlin { this: KotlinMultiplatformExtension
    androidTarget {...}
    iosX64()
    iosArm64()
    iosSimulatorArm64()

    cocoapods { this: CocoapodsExtension
        version = "1.2.0"
        ios.deploymentTarget = "14.0"
        podfile = project.file("../iosApp/Podfile")
        framework {...}

        specRepos { this: CocoapodsExtension.SpecRepos
            url("https://github.com/CocoaPods/Specs.git")
            url("git@github.com:Bancar/uala-core-ios.git")
        }
        pod( name: "UalaCoreInterfaces") { this: CocoapodsExtension.CocoapodsDependency
            source = git( url: "https://github.com/Bancar/uala-core-ios") { this: CocoapodsExtension.CocoapodsDependency.PodLocation.Git
                branch = "POC/tokenProviderInterface"
            }
        }
    }
```

FIGURA 13: DEPENDENCIAS IOS.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Una vez integradas las dependencias, se puede continuar con la incorporación de las mismas al módulo. Para ello, se hace uso de la potencia de *Kotlin Multiplatform* (KMP) mediante clases *expect*. Estas clases actúan como interfaces que obligan a cada plataforma a declarar la implementación correspondiente mediante clases del tipo *actual*. De esta manera, logramos utilizar *API* específicas de cada plataforma dentro del módulo compartido *shared* y emplearlas sin problemas al momento de implementar la lógica de negocios.

```
publish.gradle    KMMUserCredentials.kt ×
1  package com.uala.labs.kmm.base
2
   Mauromarod
3  expect class KMMUserCredentials {
       Mauromarod
4      fun getTokenSynchronously(handleNoSession: Boolean = false): String
5  }
6
```

FIGURA 14: OBTENER TOKEN MEDIANTE USERCREDENTIALS.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Este tipo de declaraciones permite que, al integrar librerías nativas, cada plataforma utilice las clases correspondientes. Por ejemplo, en Android, se logró implementar el método para obtener el *token* mediante el uso de una interfaz propia de la librería *Core.Authorization* llamada *UserCredentialsInterface*. En el caso de iOS, para implementar el método para obtener el *token*, se utilizó la librería nativa de iOS a través del protocolo *TokenProviderProtocol*.

```
IOSUserCredentials.kt ×
3  import cocoapods.UalaCoreInterfaces.TokenProviderProtocolProtocol
4
   Mauromarod
5  actual class KMMUserCredentials(private val tokenProvider: TokenProviderProtocolProtocol) {
6
       Mauromarod
7      actual fun getTokenSynchronously(handleNoSession: Boolean) : String = tokenProvider.getToken()
8  }

AndroidUserCredentials.kt ×
3  import ar.com.bancar.uala.authorization.data.UserCredentialsInterface
4
   Mauromarod
5  actual class KMMUserCredentials(private val userCredentials: UserCredentialsInterface) {
       Mauromarod
6      actual fun getTokenSynchronously(handleNoSession: Boolean): String {
7          return userCredentials.getTokenSynchronously(handleNoSession)
8      }
```

FIGURA 15: IMPLEMENTACIÓN NATIVA DE USERCREDENTIALS.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Para avanzar con la integración de las librerías nativas de autenticación, se hizo uso de estas clases que posibilitan obtener el *token*. Este viaja en las llamadas al *backend* como parte de los *headers* de cada petición. Para mantener coherencia con el resto de los módulos desarrollados dentro de Ualá, se procede a crear una arquitectura escalable que tiene en cuenta que contamos con un inyector de dependencias.

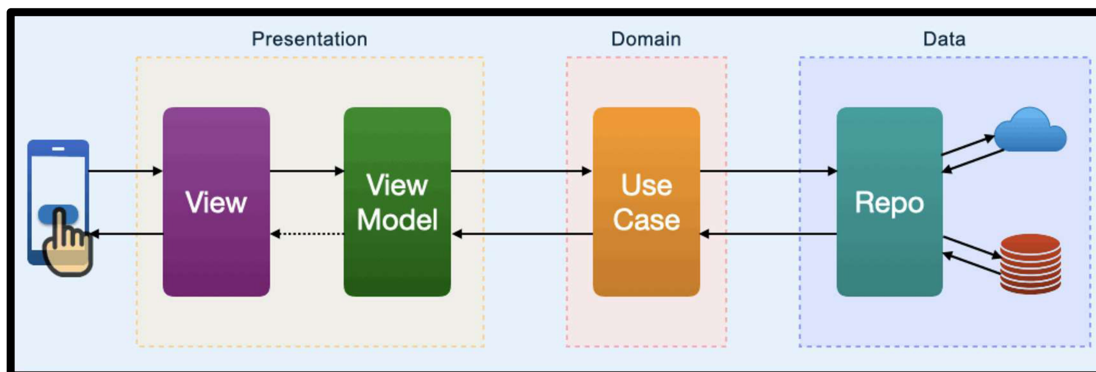


FIGURA 16: VIEWMODEL ARCHITECTURE.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Koin facilita el uso de los distintos módulos en diferentes partes de la aplicación, incluso la posibilidad de crear una arquitectura multimódulo en caso de ser necesario. Para actualizar sus beneficios se modificó la estructura de la configuración inicial de *Koin*, ya que es necesario que, al inicializarse en cada plataforma, estas envíen la implementación de nuestra interfaz en Android y el protocolo en iOS. De esta manera, se obtiene un único punto de entrada para todos los servicios que incluyen peticiones.

```
Mauromarod *
fun initKoin(koinInput: KoinInput, appDeclaration: KoinAppDeclaration = {}): KoinApplication {
    stopKoin()

    return startKoin { this: KoinApplication
        appDeclaration()
        modules(getSharedModules(koinInput))
    }
}

new *
fun getSharedModules(koinInput: KoinInput): List<Module> {
    val (countryId, environment) = koinInput

    val environmentConfig = KmpConfigurations.getConfiguration(countryId, environment)

    return listOf(platformModule(), domainModule, getDataModule(koinInput, environmentConfig), dispatcherModule, viewModelModule)
}
```

FIGURA 17: INICIALIZAR KOIN.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

La última clase a desarrollar fue el *ViewModel*: esta es una clase que tiene una responsabilidad particular, ya que conectará la interfaz de usuario con la lógica de negocios incluida en los repositorios. Este *ViewModel* (VM), como los creados en cada *feature* a desarrollar, usará el tipo de *Kotlin open,* que permite aplicar herencia en la aplicación nativa de iOS, justamente en la clase similar a la del *VM* de Android que cuenta actualmente con una implementación basada en la arquitectura *ObservableObject.* En esta clase, se definieron los diferentes estados de la pantalla. Así, al iniciar el flujo, se mostrará un efecto *shimmer* mientras se realiza la petición al backend. Al obtener los datos, se ocultarán los componentes de cargas para mostrar los datos obtenidos en las pantallas correspondientes. Es importante destacar que, al compartir la lógica en ambas plataformas, el comportamiento será el mismo. Solo será necesario definir qué componentes visuales se utilizarán para mostrar los datos.

Luego de la configuración inicial de la integración de las librerías, se continuó con la implementación del módulo en la aplicación utilizada como *demo*. También se

agregó la librería propia de Ualá que se encarga del inicio de sesión para obtener los parámetros mencionados previamente, que *Koin* utilizará para inyectar nuestros repositorios. Una vez que la inyección de dependencias ha iniciado, la aplicación dispone de las credenciales suficientes para realizar peticiones al *backend*. Comenzamos las pruebas en la aplicación Android, donde contamos con la interfaz de usuario correspondiente para listar los cupones utilizando *Compose* y obtenemos los resultados esperados obteniendo una respuesta correcta por parte de nuestro módulo KMP.

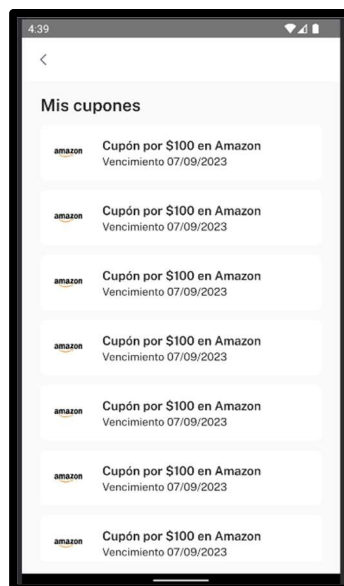


FIGURA 18: MIS CUPONES DESDE KMP.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Dentro de la demo iOS inicializamos el módulo de *KMP* mediante una clase del tipo *struct*, propia de *Swift*, en la cual implementamos un método estático que recibe los parámetros del *environment* y procede a generar la *configuración* inicial del inyector de dependencias. Se desarrolló el viewmodel nativo, que se utiliza en Swift, para vincular la pantalla de cupones con los servicios que realizan la llamada al *backend,*

de forma tal que nuestra clase herede de nuestro VM multiplataforma y probamos la aplicación obtuvimos el mismo comportamiento que en la demo de Android.

4.3.4 CI/CD

Automatizar las pruebas, la compilación, la validación del código y el despliegue desde el comienzo de un proyecto de desarrollo, especialmente en uno multiplataforma, permite una mayor eficiencia al centrarse en la creación y mejora de características, en lugar de realizar tareas manuales repetitivas. Además, asegura que cada cambio en el código se someta a pruebas y validaciones de manera sistemática. Esta automatización no sólo acelera la detección de errores, sino que también garantiza que el *software* permanezca en un estado funcional en todo momento.
La automatización también contribuye a mejorar la calidad del *software*. Al ejecutar pruebas de manera continúa y desplegar automáticamente las versiones probadas y validadas, se reduce significativamente el riesgo de errores y problemas en producción. Esto conduce a un *software* más confiable y robusto, lo que facilita la gestión de versiones y correcciones. Además, permite una rápida y coherente implementación de soluciones a medida que el proyecto evoluciona, promoviendo así un desarrollo más eficiente y seguro.

4.3.4.1 Android

En una primera iteración del proceso de automatización, se optó por realizar un despliegue local de la versión Android del módulo multiplataforma. Para esto, se utilizaron las librerías nativas de *Maven* y *JFrog*, que permiten distintos tipos de despliegues a través de la configuración específica de ambientes en los archivos *Gradle* del proyecto.

Para iniciar la configuración, se agregaron las dependencias al proyecto y al módulo compartido desde los archivos *build.gradle* respectivos.

```
pluginManagement {
    repositories {
        google()
        gradlePluginPortal()
        mavenCentral()
        maven { url = uri("https://plugins.gradle.org/m2/") }

        val artifactoryUser: String by settings
        val artifactoryPassword: String by settings
        val artifactoryUserSnapshot: String by settings
        val artifactoryPasswordSnapshot: String by settings

        maven {
            url = uri("https://uala.jfrog.io/uala/plugins-release")
            credentials {
                username = artifactoryUser
                password = artifactoryPassword
            }
        }

        maven {
            url = uri("https://uala.jfrog.io/uala/libs-snapshot")
            credentials {
                username = artifactoryUserSnapshot
                password = artifactoryPasswordSnapshot
            }
        }
    }
}
```

FIGURA 19: CONFIGURACIÓN DE MAVEN Y JFROG.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Una vez que se integraron las librerías, se configuró *Maven* para el uso correcto de credenciales, repositorios y ambientes a desplegar desde el archivo *Settings.gradle*. Una configuración adicional incluyó la creación de un archivo *Publish.gradle*. A través de la combinación de los plugins *maven-publish* y *jfrog-artifactory*, se estableció el *path* donde se llevaría a cabo el *deploy*. Se declararon los tipos de *build* a generar, incluyendo una versión *debug* para desarrollo y una versión *release* para producción.

Al comenzar con las pruebas, se generó una versión local que llevó a cabo la compilación y publicación del módulo multiplataforma, simulando el proceso de publicación al repositorio remoto de JFrog configurado.

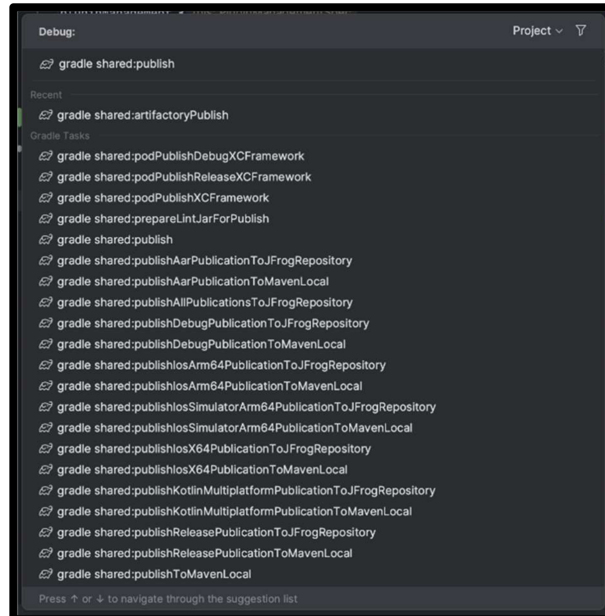


FIGURA 20. TAREAS DE PUBLICACIÓN DESDE GRADLE.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Al ejecutar la tarea *publishDebugPublicationToMavenLocal* se obtiene un Android Archive (*.aar*) que contiene recursos de Android, clases en *Kotlin/Java* y bibliotecas *C/C++* del proyecto *shared*. De esta manera, la biblioteca de recursos resultante se puede compartir e integrar de manera sencilla en proyectos nativos de Android.

El siguiente paso consistió en replicar este proceso para publicar una versión remota de la librería. Esto permitió probar la integración más allá del entorno local mediante los repositorios de *JFrog*. Se ejecutó la tarea correspondiente a la publicación remota *publishDebugPublicationToJFrogRepository* sin agregar mayor complejidad, ya que esta se creó automáticamente al momento de establecer la configuración de los plugins.

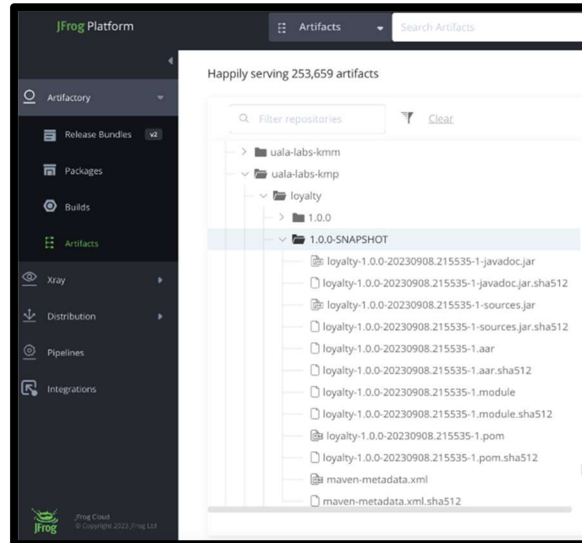


FIGURA 21: REPOSITORIO JFROG.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Una vez creada la versión remota dentro del repositorio Ualá de *JFrog*, se continuó con su integración en la aplicación nativa de Android mediante *Gradle*, como cualquier otra librería nativa. Esto permitió verificar que la librería se publicó de manera exitosa.

```
//Braze
implementation(coreCatalogue.appBoy)

//Google play library
implementation(coreCatalogue.googleGMSAuth)
implementation(coreCatalogue.googleGMSAdsIdentifier)
implementation(coreCatalogue.googleGMSMaps)
implementation(coreCatalogue.googleGMSLocation)

//Loyalty
implementation "uala-labs-kmp:loyalty:$ualaLoyaltyVersion"

//KMP
implementation("io.insert-koin:koin-android:$koinVersion")
implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:$serializationVersion")

//Firebase Libs
api platform(coreCatalogue.firebaseBOM)
api(coreCatalogue.firebaseMessaging)
api(coreCatalogue.firebaseAnalytics)
api(coreCatalogue.firebaseCrashlytics)
api(coreCatalogue.firebasePerf)
api(coreCatalogue.firebaseConfig)
```

FIGURA 22: INTEGRACIÓN DE DEPENDENCIAS KMP.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

El último paso para mejorar el flujo de *CI/CD* en el proyecto Android fue crear un *workflow*. Se denomina así al grupo de pasos o tareas asociadas a cualquier flujo correspondiente a la construcción de un módulo o aplicación. En este caso el *workflow* a crear será el encargado de automatizar el despliegue del módulo cada vez que se realizará un *Pull Request* (PR) o *Push* al *branch* principal (*main*) del proyecto. Para esto, se utilizó la herramienta proporcionada por *GitHub* llamada *Actions*. Esta herramienta facilita la creación de flujos mediante archivos *.yml*, donde se configura el nombre del *workflow*, cuándo se ejecutará, el sistema operativo a utilizar (en este caso, Ubuntu para un mejor rendimiento costo-beneficio) y los pasos a ejecutar. En este caso el flujo se activa cuando se realiza un *push* en el *branch* *main* y los pasos a ejecutar para esta versión incluyen la configuración de *Java* necesaria para la compilación del proyecto y la ejecución de la tarea de *Gradle*, utilizada anteriormente para publicar el módulo en los repositorios remotos de *JFrog*

una vez que el módulo se hubiera compilado. De esta manera, concluyó la primera iteración de la automatización del módulo multiplataforma para ser integrado como una librería nativa de Android.

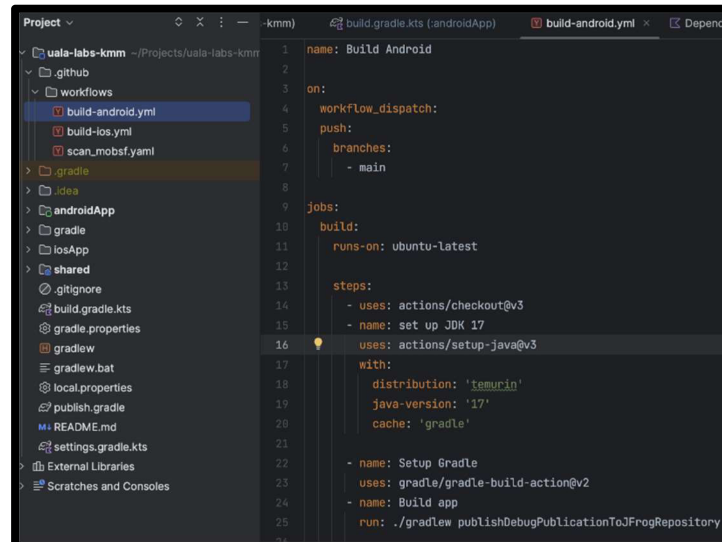


FIGURA 23: ANDROID WORKFLOW.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

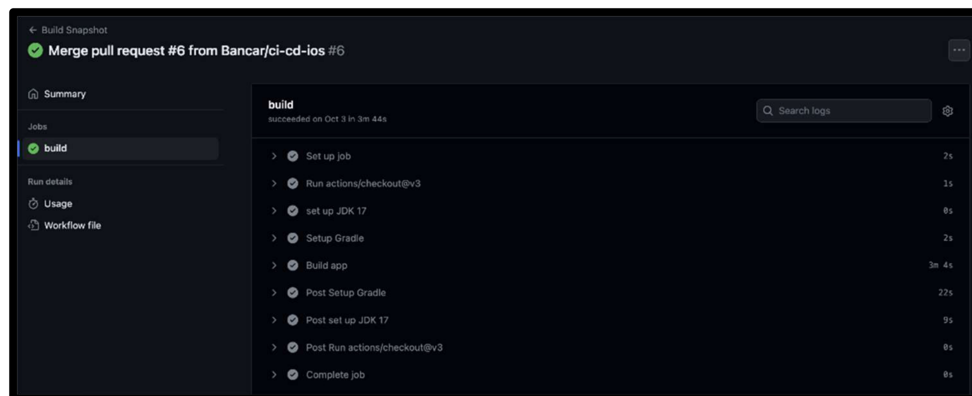


FIGURA 24: ANDROID ACTIONS.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

4.3.4.2 IOS

Para dar continuidad a la automatización, se buscó replicar los pasos previamente ejecutados para el entorno Android, enfocado en el desarrollo de una versión local de la librería multiplataforma. El objetivo era integrar dicha librería en aplicaciones nativas de iOS, lo que implicaba mantener la misma metodología que utilizan los desarrolladores de iOS de Ualá para exportar módulos a la App publicada en la *AppStore* de Apple.

Un análisis detallado de la estructura del proyecto permitió asegurar que el camino a seguir era utilizar *XCFrameworks* como formato de salida de nuestro módulo. El resultado de este proceso debió publicarse en un repositorio privado de Ualá para luego ser integrado como una dependencia a través de *Cocoapods*, siguiendo un enfoque similar al empleado al integrar la librería de *UalaCoreInterfaces* dentro de nuestro módulo *shared*.

En contraste con el *workflow* de Android, la configuración inicial del *plugin* de *Cocoapods* resultó suficiente para continuar con la publicación del *framework*. Este plugin se integró a *Gradle* y, una vez sincronizado el proyecto, generó las tareas necesarias para la compilación y publicación local del *XCFramework* junto con su archivo *podspec* correspondiente. La ejecución de la tarea *podPublishXCFramework* creó tanto una versión depurada (*debug*) como una productiva, ambas listas para ser exportadas al repositorio privado.

Para automatizar la operación de exportación del *XCFramework* al repositorio de *Pods* de Ualá, se implementó un nuevo flujo en *GitHub Actions*, teniendo en cuenta las particularidades de la compilación de módulos en el entorno de iOS. Esto incluyó el requisito de utilizar una máquina virtual con sistema operativo MacOS; de lo contrario, la compilación no generaría el *framework* necesario. Esta configuración se realizó en el archivo *.yml*, que define los pasos a ejecutar cuando se genera un

pull request o *push* en la rama principal del proyecto (*main*), al igual que en el flujo de Android.

Dentro del archivo *.yml*, se desarrolló el *step* utilizando ac*tion/checkout@v2* para clonar el repositorio en la máquina virtual; se instaló la versión requerida de *Java* (Java 17) mediante *setup-java@v3* y se agregó la conexión *SSH* utilizando las credenciales de los repositorios privados de Ualá. El último paso para completar la configuración del entorno fue llevar a cabo el *setup* de *Gradle*, una herramienta clave para ejecutar las tareas de publicación del módulo, lo cual se realiza a través de *Gradle-build-action@v2.*

Finalizada la configuración, se agregó un siguiente *step* para compilar el módulo utilizando la tarea *build* de *Gradle*. Esto, a su vez, genera los archivos necesarios para que el plugin de *Cocoapods* pueda publicar el *XCFramework* correctamente a través de la tarea *podPublishXCFramework*, misma *task* que se utilizó en la prueba local. Con este último paso, el flujo de trabajo produce los archivos requeridos para su publicación en el repositorio privado de Pods de Ualá. Esto implica añadir un paso que clone el repositorio, configure las credenciales necesarias y cambie al directorio del repositorio clonado. Luego, en un paso subsiguiente, se crea una carpeta con el nombre de la versión actual del módulo y se copian tanto el *XCFramework* como el *podspec* al nuevo directorio. Finalmente, se ejecuta un último paso encargado de realizar el *commit* y el posterior *push* de los cambios relacionados con la creación de la nueva versión del módulo multiplataforma.

De esta manera, el proyecto cuenta con tareas destinadas a realizar publicaciones locales para corroborar el funcionamiento normal de la publicación de módulos, así como facilitar la instalación en distintos proyectos locales que lo requieran. Además, se incluyen dos flujos de trabajo destinados a automatizar el proceso de publicación remota del módulo multiplataforma en sus respectivos repositorios privados. En una

próxima iteración, se podrían agregar *steps* para ejecutar las pruebas del proyecto durante la compilación de cada una de estas versiones, e incluso implementar algún corrector de código como *Lint Actions*.

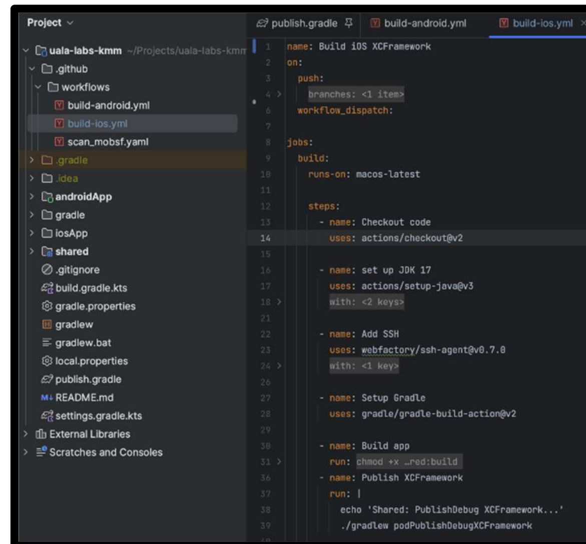


FIGURA 25: IOS WORKFLOW.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

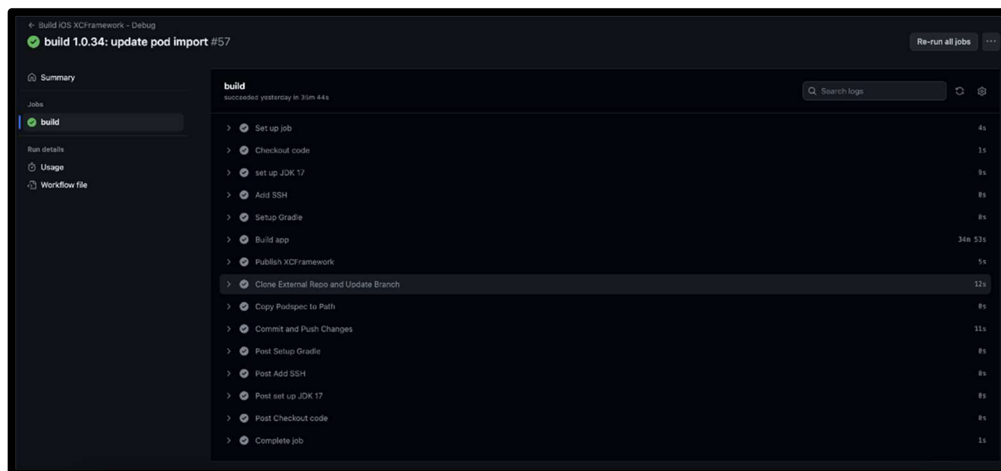


FIGURA 26: IOS ACTIONS.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

4.4 Módulo Promociones

4.4.1 Análisis y propuestas

Para comenzar con la elección de la *feature* a desarrollar se realizó un análisis previo de flujos en los que se pudiera implementar un módulo multiplataforma que beneficiara de alguna manera tangible a los usuarios mejorando su experiencia, ya sea agregando alguna nueva funcionalidad, o mejorando la performance de la app, y se brindó al *Product Owner* dos posibilidades para que priorizara cual debería ser el módulo a desarrollar dentro de esta práctica profesional.

La primera opción consistía en agregar persistencia de datos a ciertos componentes de la pantalla de *Home* como accesos directos, movimientos, tarjetas del usuario, entre otros componentes. Esta *feature* permitiría evitar la realización de peticiones constantes al *backend* cada vez que el usuario ingresa a la app para obtener datos que generalmente no varían y suelen ser constantes durante grandes plazos de tiempo. Además, al guardarlos en una base de datos local, dentro del celular, el proceso de lectura de estos datos sería mucho más rápido que en caso de obtenerlos realizando peticiones *HTTP* a servicios *cloud*.

La otra opción era reemplazar uno de los flujos más utilizados en las dos plataformas: Promociones. Esta *feature* se basa en mostrarle al usuario las diferentes promociones a las cuales puede acceder por ser usuario de Ualá, pero la característica principal de este flujo era que en ninguna de las dos plataformas eran parte propia de la app nativa, sino que al acceder desde cualquiera de los accesos al módulo de promociones, la app lanzaba a un navegador externo que navegaba automáticamente a la *url* de Ualá destinada a mostrar las promociones. La problemática particular de que el flujo no estuviera desarrollado en forma nativa en ninguno de las dos plataformas no sólo se basa en que perjudicaba la experiencia

de usuario, sino que de algún modo este salía de la aplicación, con el riesgo de dejar de usarla. Otro de los problemas era que el equipo de marketing perdía la posibilidad de aplicar métricas desde los SDK que se utilizan para medir tiempos, procesos, usabilidad, errores, por mencionar algunas de las métricas en las cuales se basan las decisiones y la prioridad de los desarrollos o campañas destinadas a captar nuevos usuarios o fidelizar a los existentes. Las ventajas de optar por esta opción brindaría a MKT la posibilidad de aplicar estudios incluso sobre con qué marcas desarrollar campañas o renovar las promociones teniendo en cuenta cuáles son las más solicitadas por los usuarios midiendo incluso aspectos específicos del flujo como la búsqueda de categorías, o el acceso al detalle de cada marca.

Una vez que desde el equipo de Producto analizaron los pros y contras de cada una de las posibilidades, estos eligieron desarrollar de forma nativa la *feature* de Promociones por la posibilidad de aplicar métricas al flujo completo, el consejo por parte de *Core* que evidenciaba la mejora en el proceso de integración y *deploy* en el *AppStore* que suele lanzar advertencias por tener flujos que necesitan de navegadores incrustados dentro de la app para utilizar una funcionalidad, y el consenso por parte de UX/UI de que brindar una funcionalidad nativa iba a mejorar sustancialmente la experiencia de navegación de la app.

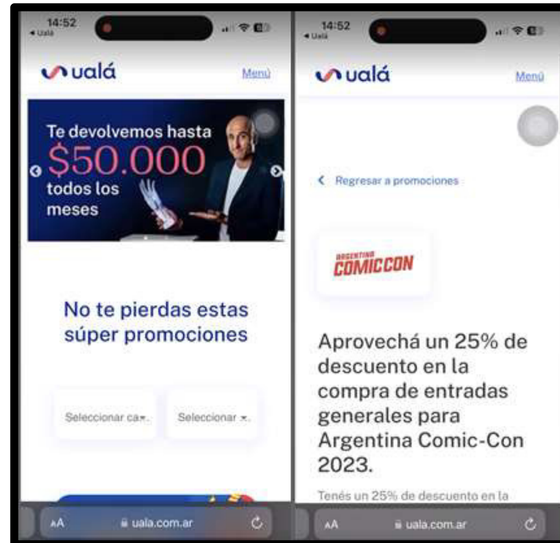


FIGURA 27: FLUJO DE PROMOCIONES INICIAL.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

De esta manera el módulo a desarrollar debía mostrar las promociones vigentes para los usuarios de Ualá brindando la posibilidad de filtrar, por categoría e ubicación, además ofrecer una pantalla de detalle donde se especifique la información importante sobre cada promoción, como puede ser los días de vigencia, los términos y las condiciones, o los códigos para uso, y, por último, desde la misma pantalla de detalle se agregarían accesos a webs destinadas a las marcas participantes.

A nivel técnico, uno de los requerimientos más importantes, sin mencionar que el desarrollo sea nativo, era que toda la información se debía actualizar de forma automática sin necesidad de lanzar nuevos *releases* de las apps, por lo que se estableció que la información a consultar tenía que ser provista por un sistema de gestión de contenidos utilizado por marketing, como es *Contentful*. Al igual que todo desarrollo nuevo era necesario hacer uso de las librerías destinadas al uso de componentes propios del *Design System* de Ualá denominado Abra, disponible para ambas plataformas.

4.4.2 Desarrollo

Teniendo en cuenta que, al ser una feature nueva, se debía definir cómo se iban a organizar no solo el código en términos de su arquitectura sino también resolver otras dudas técnicas. ¿Cómo se dividirían los repositorios? ¿Se debía utilizar un repositorio para el desarrollo KMP y otros dos para la parte visual de cada plataforma? ¿Se podrían integrar los componentes visuales al repositorio general de cada app? ¿Se deberían generar nuevos repositorios por *feature* donde convivan los módulos multiplataforma junto con los de Android e iOS? Esta decisión debía tener en cuenta que al crear nuevos repositorios, estos iban a necesitar automatización para los procesos que no pertenecieran exclusivamente al proyecto por lo que se tomó la decisión en conjunto, con arquitectura de ambas plataformas, de crear un repositorio propio para los módulos multiplataforma donde se modificaría la lógica de negocio a compartir y, además, todo evolutivo a nivel interfaz de usuario debía realizarse dentro de los repositorios existentes y, en caso de que actualmente no contarán con uno, se podría crear uno por S.O. si podría separarlo de la app integrada. Distribuir las responsabilidades de esta manera facilitó el comienzo del desarrollo pero, una vez finalizado el proceso, se realizó una evaluación de cómo podría seguir iterando este proceso al implementar más *features* utilizando la tecnología de Kotlin Multiplatform a medida que se distribuyera el conocimiento en los equipos de desarrollo.

Sobre la base de lo desarrollado en el módulo de prueba, se pudo reutilizar el diseño de arquitectura separando en capas nuestro *feature*. Dentro del *package Data* se agregaron las clases que mapean los *request* y *response* necesarios para obtener las promociones realizando peticiones a *GraphQL* mediante *Ktor*, al igual que se implementó anteriormente. Las clases destinadas a la conexión entre las apps y la

API también se ubicaron dentro de Data; estas implementan los métodos del *package Domain*:

```
package com.uala.labs.kmm.data.repository

import ...

Mauromarod *
internal class PromosRepositoryImpl(
    private val remoteData: PromosRemoteData,
    private val token: String
) : PromosRepository {
    Mauromarod
    override suspend fun getPromos(): PromotionData {
        val limit = 25
        val maxBatches = 4
        var currentSkip = 0

        val availablePromotions = mutableListOf<PromotionItem>()

        while (currentSkip < maxBatches * limit) {
            val promotions = getPromos(limit, currentSkip)

            availablePromotions.addAll(promotions)

            if (promotions.size < limit) break

            currentSkip += limit
        }

        return mapToPromotions(availablePromotions)
    }
```

FIGURA 28: PROMOCIONES - DATA.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Dentro de la capa dominio se definieron los métodos necesarios para implementar toda la lógica de negocio mediante *interfaces repository*. En esta capa además de las *interfaces,* se crearon los *models,* clases que mapean las propiedades más significativas para definir las entidades que se usarán dentro del flujo mediante la capa *UI* de promociones. Por último se crearon los *UseCase que* facilitan la comunicación entre cada *repository* con el ViewModel que consumirá cada app para mostrar la información mediante sus módulos UI.

```
package com.uala.labs.kmm.domain.repository

import ...

Mauromarod
interface PromosRepository {
    Mauromarod
    suspend fun getPromos(): PromotionData
    Mauromarod
    suspend fun getBanners(): List<PromotionBanner>
    Mauromarod
    suspend fun getTitles(): PromotionTitle
    Mauromarod
    suspend fun getPromotionDetails(promoId: String): PromotionDetails
}
```

FIGURA 29: PROMOCIONES – DOMAIN.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

```
package com.uala.labs.kmm.domain.usecase

import ...

Mauromarod
class GetPromosUseCase(private val repository: PromosRepository) {

    Mauromarod
    @Throws(Throwable::class)
    operator fun invoke(): Flow<Result<PromotionData>> = flow { this: FlowCollector<Result<PromotionData>>
        emit(
            try {
                Result.success(repository.getPromos())
            } catch (ex: Exception) {
                Result.failure(ex)
            }
        )
    }
}
```

FIGURA 30: PROMOCIONES - USECASE.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

En la capa de presentación del módulo *multiplatform* solo se creó un *ViewModel*, que tiene como responsabilidad exponer y ejecutar los métodos necesarios para realizar las peticiones a la *API* de *contentful* que brindarán la información que viaja mediante las capas Domain y Data al módulo UI. Con el desarrollo de Loyalty funcionando, solo se adaptaron los métodos a los nuevos casos de uso, con la

particularidad de que en el desarrollo anterior los métodos eran privados: al ingresar a la pantalla de Cupones, se inicializaba el *ViewModel* y ejecutaba automáticamente el flujo de peticiones y recolección de datos mediante Flows; en este caso, la decisión fue que los métodos para obtener las promociones fueran públicos y, de esta manera, que fuera más flexible para la comunicación por parte de la UI de cada plataforma. Además de los métodos, el *ViewModel* es el encargado mantener el estado de cada pantalla y, por ejemplo, al iniciar las peticiones a *Contentful*, comunicarle a las pantallas que el estado es de carga, por lo tanto estas mostrarán componentes de carga como *Shimmers* o *CircleLoading*, según el caso, siguiendo las indicaciones del equipo de Diseño.

```
Mauromarod *
open class PromotionsKmpViewModel : MainIoExecutor() {
    Properties

    Mauromarod *
    fun getPromotionsData() {
        collect(flow = getBannersUseCase()) {...}
        collect(flow = getTitlesUseCase()) {...}
        collect(flow = getPromosUseCase()) { resource ->
            resource
                .onSuccess { data ->
                    if (data.promotionList.isEmpty()) {...} else {
                        _promotionsData.update { currentData ->
                            currentData.copy(
                                featuredList = data.featuredList,
                                promotionList = data.promotionList,
                                categoryFilters = data.categoryFilters,
                                localizationFilters = data.locationFilters,
                                filteredPromotions = filterPromotions(data.promotionList)
                            )
                        }
                        _promotionsState.value = PromotionsStateUI.Success(_promotionsData.value)
                    }
                }
                .onFailure { error -> _promotionsState.value = PromotionsStateUI.Error(error) }
        }
    }
```

FIGURA 31: PROMOCIONES - VIEWMODEL.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Con la experiencia reciente del desarrollo multiplataforma del flujo Mis cupones, crear la lógica de negocio de Promociones no generó mayor dificultad más que la de entender cómo realizar los *request* hacia *Contentful* y poder modelar las respuestas de la *API* hacía clases que faciliten la representación visual mediante

los componentes visuales de cada Design System. Este proceso de adaptar los *reponse* de la *API* de *GraphQL* evidenció muchos errores en el modelado de datos al momento de generar la *API*. Uno de los más evidentes es que, para obtener los datos para listar todas las promociones, hay que realizar 4 peticiones distintas que devuelven por separado los datos de, por ejemplo, títulos, categorías, locaciones y promociones. La ventaja de unificar el proceso de obtención, mapeado y devolución de datos permite que este error se siga replicando por cada plataforma. Más allá de la ventaja de identificar y describir el error, en un futuro, mejorando la API, es posible que pueda solucionarse.

Por cuestiones relacionadas a los equipos de Diseño y Producto se dio prioridad al desarrollo Android para continuar con el proceso de incorporar de forma nativa el flujo de Promociones a las apps. Utilizando los componentes de Abra, sistema de diseño desarrollado mediante Compose para apps android, se siguieron los lineamientos actuales para desarrollar cada uno de los componentes visuales necesarios para asimilar el flujo creado para la Web de Ualá, pero adaptándolo a las características propias de una *app mobile*. Como se mencionó anteriormente, al crear la interfaz gráfica, se hizo uso del repositorio general de Ualá donde se encuentra la aplicación que integra cada uno de los módulos, divididos generalmente por *feature*, que conforman la aplicación de cada sistema operativo. Se agregó un package *Promotions* en el cual se fueron creando los componentes visuales mediante Compose, la *activity* que contendrá el flujo completo mediante la estrategia de *Compose*, conocida como Coordinador, mediante la cual se pasan como parámetros el *ViewMode*l, el contexto y se van creando los métodos para generar toda la navegación entre las distintas *Screens* del flujo, con sus respectivos estados de inicio (*Init*), carga (*Loading*), error (*Error*) y datos (*Success*). De esta manera, según la interacción del usuario, tenemos una única fuente de verdad

mediante la cual se pueden disparar los distintos procesos del flujo, como pueden ser filtrar datos, navegar a las webs de las marcas o navegar a pantallas de detalle. Para conectar la *UI* con el *ViewModel* del módulo multiplataforma repetimos la configuración de *Koin*, por lo que como paso previo agregamos las dependencias necesarias como *Koin* y *Kotlinx.Serialization*. La dependencia de Promociones multiplataforma utiliza la misma configuración de *Dagger* que nos permitió, anteriormente, exportar módulos a los repositorios remotos de Ualá en JFrog por lo que solo es necesario *pushear* al *branch* dedicado a promociones y, al generar el *PR*, se dispara el flujo automático de *deploy* en *JFrog* que nos permite importarlo con facilidad en la app integrada. Una vez sincronizado el proyecto, comenzamos a desarrollar los métodos necesarios para la inicialización del ambiente de *Koin*. Para esto, agregamos al inicio de la app un método que obtuviera los datos del ambiente, como son el país, el *environment* (*debug, stage, prod*) y los *tokens* necesarios para cada *environment*. Una vez culminada la configuración de *Koin*, se prosiguió a inyectar el *PromotionViewModel* al *PromotionActivity* y utilizar los *models* del dominio para llenar cada componente con la información correcta.

```
dencies.gradle    UalaApplication.java ×
public void initEnvironment(EnvironmentModule module) {
    moduleComponents.clear();
    Printer.info("Init Environment");
    environmentComponent = DaggerEnvironmentComponent.builder()
            .environmentModule(module)
            .coreModule(new CoreModule(module.getFactory().getCoreEnvironmentFactory()))
            .mFAModule(new MFAModule(module.getFactory().getMFAFactory()))
            .cardModule(new CardModule(module.getFactory().getCardFactory()))
            .transactionsModule(new TransactionsModule(module.getFactory().getTransactionsFactory()))
            .transfersModule(new TransfersModule(module.getFactory().getTransfersFactory()))
            .screenProviderModule(new ScreenProviderModule(module.getFactory().getScreenProviderFactory()))
            .selectableListModule(new SelectableListModule(module.getFactory().getSelectableListFactory()))
            .deviceIdSetterModule(new DeviceIdSetterModule(module.getFactory().getDeviceIdSetterFactory()))
            .gAcquiringModule(new GAcquiringModule(module.getFactory().getAcquiringFactory()))
            .signUpModule(new SignUpModule(module.getFactory().getSignUpFactory()))
            .helpModule(new HelpModule(module.getFactory().getHelpFactory()))
            .profileModule(new ProfileModule(module.getFactory().getProfileFactory()))
            .creditsModule(new CreditsModule())
            .appComponent(appComponent).build();

    ApplicationUtils.Companion.initKoinForKMP(
            application: this,
            module.provideEnvironment().getCountryId(),
            BuildConfig.ENVIROMENT,
            environmentComponent.getUserCredentials(),
            BuildConfig.PROMOTIONS_GRAPHQL_TOKEN
    );
```

FIGURA 32: PROMOCIONES – APP – INITKOIN.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

```
package ar.com.bancar.uala.ui.promotions

import ...

Mauromarod *
class PromotionsActivity : VisualBaseComponentActivity() {

    private val promotionsKmpViewModel: PromotionsKmpViewModel by inject()

    Mauromarod *
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            val scaffoldState = rememberScaffoldState()

            UalaAbraTheme {
                Scaffold(
                    modifier = Modifier.fillMaxSize(),
                    scaffoldState = scaffoldState,
                    snackbarHost = { PromotionsSnackBarHost(hostState = it) },
                    topBar = {...},
                ) { paddingValues ->
                    Box(
                        modifier = Modifier.padding(top = paddingValues.calculateTopPadding())
                    ) { this: BoxScope
                        PromotionsNavHost(
                            coordinator = rememberPromotionsCoordinator(
                                promotionsViewModel = promotionsKmpViewModel
```

FIGURA 33: PROMOCIONES - ACTIVITY.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Firma Estudiante:	Firma tutor UNAJ:	Firma tutor TAPTA UNAJ:	Firma tutor Empresa:

Culminada la etapa de desarrollo, se comenzaron a ejecutar las pruebas en distintos ambientes y, al obtener éxito en cada una de estas, se continuó con la generación de un *Split* que permitiera generar un público específico para que la salida a producción fuera solo a un porcentaje para facilitar el control de cualquier contingencia que pudiera llegar a suceder. La implementación de *Split* como herramienta para segmentar público hace que no sea necesario depender de la salida de un *release* para cambiar un flujo, como puede ser prender o apagar una *feature*, por lo que agregarla al desarrollo de promociones nos dió la posibilidad de mantener el flujo que utiliza un navegador externo para acceder a la funcionalidad desde la web y, en la misma versión de la app, direccionar a una proporción de los usuarios hacia el desarrollo nativo de promociones. Se agregó la lógica necesaria para obtener el estado del Split antes de acceder a las pantallas y se realizaron las pruebas con distintos usuarios para validar que cada uno viera el estado correspondiente y navegara según la configuración indicada para cada tipo de población.

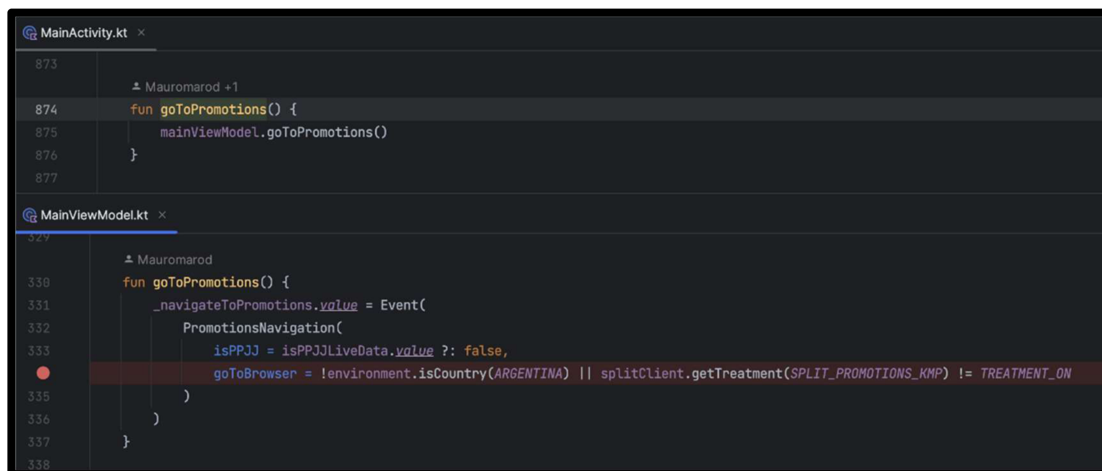


FIGURA 34: LECTURA DE FEATURE FLAG.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

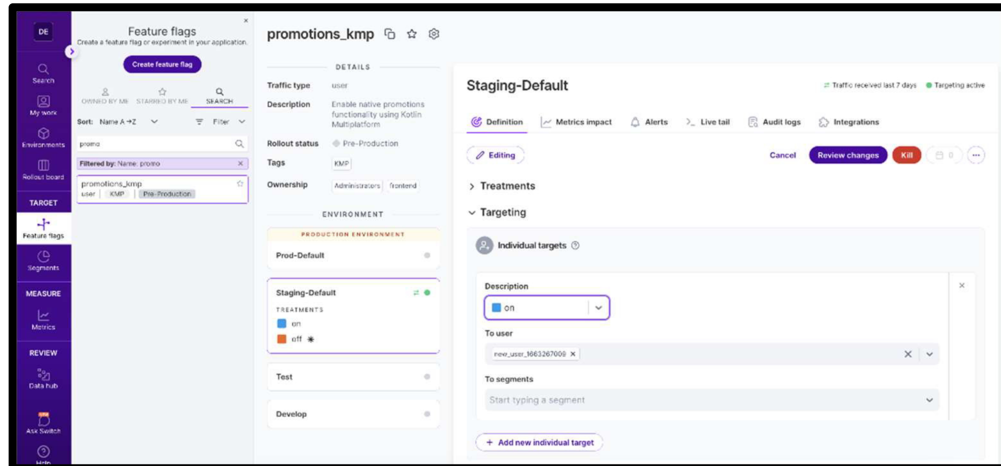


FIGURA 35: CONFIGURACIÓN DE SPLIT.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

Terminado el desarrollo, se armó el PR necesario a fin de integrarlo en la app. Al utilizar el repositorio general de Ualá no hubo necesidad de agregar o modificar el *workflow* de CI/CD y se pudo generar una versión de pruebas para ser aprobada por el equipo de QA, antes de subir los cambios a cada Store.

FIGURA 36: PROMOCIONES - FLUJO FINAL.
FUENTE: ELABORACIÓN PROPIA BASADA EN LA PRÁCTICA

5. Conclusiones

Teniendo en cuenta que el presente trabajo surgió como parte del programa de mejoras continúas que aplica Ualá para mejorar los distintos procesos que se realizan dentro de la organización, se puede evidenciar que el trabajo de investigación y desarrollo fue exitoso. Se logró crear, automatizar e integrar un módulo multiplataforma a las apps nativas Android e iOS, lo que generó un nuevo modelo de desarrollo en la empresa que reducirá los tiempos no sólo de codificación sino de *time to market,* plazo de tiempo que transcurre desde la definición de una *feature* o producto hasta que está disponible en las *store* para el usuario final. Según las métricas obtenidas en el desarrollo del módulo Loyalty se redujo el tiempo de desarrollo en un 30%, teniendo en cuenta que el desarrollo nativo en cada plataforma promedio 15 días frente a los 10 días consumidos para realizarlo mediante Kotlin Multiplatform, teniendo en cuenta además que el trabajo realizado por el desarrollador iOS se redujo a menos de la mitad.

A pesar de que el resultado final fue satisfactorio, durante el desarrollo se encontraron diversos bloqueos o situaciones que resultaron difíciles de resolver, como por ejemplo la escasa documentación para casos específicos, como son la integración de librerías nativas para iOS a través de *Cocoapods*, la compatibilidad entre versiones de las distintas dependencias utilizadas en ambas plataformas o mi escaso conocimiento del lenguaje *Swift* al momento de integrar los módulos en iOS. Al momento de integrar librerías nativas android, este proceso no varió del que se utilizaba previamente por lo que no agregó complejidad al proyecto, El caso de iOS, por el contrario, demoró varios días el proyecto al no contar con mensajes claros por parte del *IDE* y *framework*: la documentación mencionaba ejemplos simples pero no brindaba información para casos donde se utilizan credenciales, SSH o

submódulos para acceder a las dependencias, como fue en el caso del módulo de *Core* encargado del manejo de credenciales necesarias para inicializar los ambientes mediante *Koin.* Una cuestión que por momentos puso en riesgo el proyecto fue la incompatibilidad entre versiones de las apps integradas y las dependencias utilizadas para generar los módulos *multiplatform*. Al momento de generar el primer módulo se intentaron utilizar las versiones estables más actuales, tanto de *Kotlin (1.9.0)* como de las librerías necesarias para la integración con iOS como *Koin (3.4.0)* y *Cocoapods (1.9.0)*. El porqué de esta decisión era buscar la mayor estabilidad posible para una tecnología que crece día a día pero no cuenta con años de estabilidad como podría ser el ejemplo de *Futter*. El Problema era que, dentro del proyecto multiplataforma, coexisten sin problemas las distintas librerías pero, al momento de integrarlas a las apps, surgen conflictos con otras librerías, incluidos con *SDK* importantes como los encargados de transaccionar pagos. Esto obligó a realizar distintos análisis de dependencias, identificar los conflictos y adecuar las versiones necesarias para solucionarlos mediante el uso de la herramienta *dependencies* de *Gradle* lo que derivó en reemplazar las versiones de librerías que generaban el conflicto por lo que se optó utilizar las siguientes:

- *Kotlin* 1.6.21.
- *Koin* 3.2.0.
- *Cocoapods* 1.6.21.

Por otra parte, uno de los beneficios más significativos fue la implementación de los *workflows* destinados a *CI/CD* que permitieron, además, convertir un repositorio en un *template* reutilizable por cualquier equipo que desee comenzar a migrar su lógica de negocio hacia *multiplatform.* La posibilidad de contar con un repositorio donde conviven tanto la lógica de negocio como las demos destinadas a probar las

distintas *features* en cada S.O., incluso en diferentes niveles de ambientes, sin necesidad de integrarlo a las apps integradas, facilita el traspaso del desarrollo plenamente nativo a uno multiplataforma, sin cambiar el modo de trabajo de los programadores, lo que repercutirá en una mejor adaptación de estos al nuevo paradigma. Con la finalización de este proyecto se sientan las bases para unificar la lógica de negocio de cada plataforma en una única fuente de verdad por lo que facilita que cada app obtenga el mismo comportamiento sin importar el *S.O.* donde se esté ejecutando, reduciendo no sólo la cantidad de desarrolladores necesarios para la creación y/o modificación de cualquier *feature* sino también el código relacionado a cada funcionalidad. lo que evita duplicidad de código y disminuye la posibilidad de aparición de *bugs.*

En relación con los aspectos que podrían mejorar se pueden mencionar varios, como puede ser la migración de los componentes gráficos a repositorios propios de cada módulo multiplataforma, lo que reduciría los tiempos de pruebas, teniendo en cuenta que el proceso de sincronización, compilación y ejecución de las apps integradas es mucho mayor al de las demos. Para esto se podrían integrar las dependencias propias de Ualá relacionadas al sistema de diseño de cada tecnología y así mantener todo aspecto relacionado al módulo en un solo repositorio, por lo que importarlo desde otros módulos o apps sería aún menos costoso. Otra de las propuestas que se hicieron al equipo de Arquitectura fue la migración de herramientas de gestión de público, análisis de performance o eventos e incluso librerías relacionadas con la persistencia de datos como son *Firebase Analytics, Firebase Performance*, *Split* y *SQLite*. Agregar *traces* para obtener métricas de rendimiento, o la cantidad de usuarios que ingresan a cierta sección de las apps, son procesos que se deben unificar para que los resultados sean accesibles, caso similar a la segmentación de público: hoy en día se realizan en cada plataforma

utilizando distintas estrategias, lo que se debería simplificar. Por último, durante el desarrollo de este proyecto se realizó en paralelo una *POC* en android utilizando la persistencia de datos como estrategia de mejora de rendimiento y es algo que, según las métricas obtenidas en la app android, ha mejorado sustancialmente los tiempos de carga e inicio, por lo que replicarlo usando KMP sería una ventaja significativa para ambas plataformas. Por último, una de las mejoras más importantes a implementar será la de empezar a compartir componentes visuales utilizando *Compose Multiplatform*, tecnología que al momento del desarrollo de este proyecto se encuentra en fase de desarrollo y pruebas por lo que su utilización queda momentáneamente bloqueada hasta que logre la estabilidad necesaria para la salida a producción de un producto utilizado por más de 5 millones de usuarios. La posibilidad de reutilizar el *Design System Abra* desarrollado en *Compose* reduciría aún más los tiempos de desarrollo y el resto de los beneficios antes mencionados, sin cambiar el lenguaje ni formas de desarrollo para, por lo menos, la mitad de los programadores de la organización.

6. Reflexiones sobre PPS como espacio de formación

Durante el transcurso de la carrera obtenemos conocimientos técnicos que nos facilitan la incorporación al campo laboral con la posibilidad de trabajar en áreas tan distintas como puede ser Redes, Seguridad, o incluso Desarrollo, siendo esta última la que mayor salida laboral tiene para los estudiantes, incluso en los primeros años de la carrera cuando la formación nos permite elegir puestos dentro de distintas áreas o equipos como puede ser *backend*, *frontend*, *mobile* o más orientados a analistas funcionales. En mi caso, la combinación de materias como "Proyecto de Software", "Redes" y "Base de datos" me dieron la confianza necesaria para iniciar mi camino como desarrollador *mobile*, por lo que al iniciar esta práctica profesional cuento con varios años de experiencia y un *seniority* avanzado, por lo que en un principio no logré dimensionar lo que podría beneficiarme realizar esta experiencia. Al comenzar a pensar en el proyecto y llevarle mi propuesta a mi tutor organizacional, arquitecto de Ualá con el que no tenía contacto hasta ese momento, empecé a darme cuenta de que estaba frente a la posibilidad de aplicar todos los conocimientos adquiridos durante la carrera: ya no solo debía desarrollar un código, sino planificar y gestionar cuestiones relacionadas a tiempos de desarrollo, recursos e, incluso, incursionar en el diseño de arquitecturas o procesos de automatización en los que nunca había trabajado por no pertenecer a áreas o niveles de jerarquía encargadas de procesos tan importantes y centrales para aplicaciones de uso masivo. Esta práctica me dio la oportunidad de mostrarme como un referente y llevar adelante un cambio de paradigma en una empresa tan importante como es Ualá. Además, trabajar a la par de los arquitectos android e iOS me hizo crecer como profesional, adquirir habilidades blandas que son difíciles de aprender por el solo

hecho de estudiar una carrera.
Al concluir esta práctica y dado lo exitoso de la misma, no sólo logré un cambio en mi *seniority*, sino también el respaldo por parte de colegas que vieron la importancia del trabajo realizado y mi capacidad para estar al frente de iniciativas de investigación, lo que además abrió puertas hacia otros desarrollos de este tipo en los que ya estoy trabajando. En resumen creo que la experiencia de atravesar esta instancia siempre será beneficiosa para el estudiante, haya tenido o no una experiencia profesional al momento realizarla, porque nos obliga a mejorar, incorporar nuevas habilidades y enfrentar nuevos desafíos a cambio de una experiencia única en la que debemos ser protagonistas, a sabiendas de que, gracias a nuestra formación, tenemos todas las herramientas para conseguirlo, seguir creciendo y, sobretodo, aprendiendo.

7. Referencias bibliográficas

- Cocoapods.org (2023), *CocoaPods*. Disponible en https://cocoapods.org/ (Consultado el 28/11).
- Create First App (2023), *Multiplatfom development*. Disponible en https://kotlinlang.org/docs/multiplatform-mobile-create-first-app (Consultado el 28/11).
- Docs.Gradle.org (2023), *Gradle User Manual*. Disponible en https://docs.gradle.org/current/userguide/userguide (Consultado el 28/11).
- Flutter.dv (2023), *Flutter*. Disponible en https://flutter.dev/ (Consultado el 28/11).
- Github.com (2023), *Github Actions*. Disponible en https://docs.github.com/en/actions (Consultado el 28/11).
- Gradle Build Tool (2023), *Gradle Build Tool*. Disponible en https://gradle.org/ (Consultado el 28/11).
- Gradle.org (2023), *Maven Publish Plugi*. Disponible en https://docs.gradle.org/current/userguide/publishing_maven (Consultado el 28/11).
- HiBob.com (2023), *Onboarding a Ualá*. Disponible en https://hibob.com/uala/docs/onboarding (Consultado el 28/11).
- JFrog.com (2023), *JFrog Artifactory.* Disponible en https://jfrog.com/help/r/jfrog-artifactory-documentation/jfrog-artifactory (Consultado el 28/11).
- KMP Dependency Injection framework (2023). *Koin DI*. Disponible en https://insert-koin.io/ (Consultado el 28/11).
- KotlinLang.org (2023). *Kotlin Multiplatform*. Disponible en https://kotlinlang.org/docs/multiplatform (Consultado el 28/11).
- ReactNative.dev (2023). React Native. Disponible en https://reactnative.dev/ (Consultado el 28/11).
- Oficcial IDE for Android Apps (2023), *Android Studio*. Disponible en https://developer.android.com/studio (Consultado el 28/11).

- Oficcial IDE for Apple Apps (2023), *Xcode 15*. Disponible en https://developer.apple.com/xcode/ (Consultado el 28/11).
- Uala.com.ar (2023), *Somos Ualá*. Disponible en https://www.uala.com.ar/nosotr@s (Consultado el 28/11).