Encina, Fernando Sebastian

Modularización Mobile Android con CI/CD

2021

Instituto: Ingeniería y Agronomía

Carrera: Ingeniería en Informática

Universidad Nacional Arturo Jauretche

Instituto de Ingeniería y Agronomía

Carrera de Ingeniería en Informática

PRÁCTICA PROFESIONAL SUPERVISADA

Informe final

Modularización Mobile Android con CI/CD

Sebastián Encina

Florencio Varela, abril de 2021

Estudiante
Sebastián Encina
DNI: 37.819.397
Nº de Legajo: 4070
Encina.sebastian94@gmail.com
Cantidad de materias aprobadas al comienzo de la PPS: 44
Práctica Profesional Supervisada (PPS) enmarcada en artículo (4 ó 7) de la Resolución (CS) 103/16.

ORGANIZACIÓN DONDE SE REALIZA LA PPS
FluxIT S.A
Camino Centenario 2565, La Plata, (1897) Buenos Aires, Argentina
221-4453480
Sector: Desarrollo de software

Tutor por la organización
Ing. Matías Navarro
Matias.navarro@fluxit.com.ar

Docente supervisor por UNAJ
Dr. Ing. Patricio Gross
pgross@unaj.edu.ar

Docente tutor por el Taller de Apoyo para la Producción de Textos Académicos (UNAJ)
Prof. Lía Lavigna
llavigna@unaj.edu.ar

Coordinador de la carrera de Ingeniería en Informática
Dr. Ing. Martín Morales
martin.morales@unaj.edu.ar

Resumen en español

El presente informe forma parte de la práctica profesional supervisada (PPS) de la carrera de Ingeniería en informática que consiste en la migración de una aplicación *mobile* Android con arquitectura monolítica hacia una con arquitectura modular, estableciendo, además, un esquema de integración continua y despliegue continuo con el fin de automatizar y mejorar los procesos de desarrollo. En el mismo, se describe como fue el paso a paso de la migración, la elección de las herramientas para llevar a cabo el proyecto y la descripción de las implementaciones realizadas para cumplir los objetivos planteados en el alcance de esta PPS. Para concluir, se reflejan los resultados, las simplezas y dificultades encontradas durante el desarrollo del trabajo y las conclusiones y reflexiones sobre el desarrollo de software enmarcado en las prácticas profesionales.

Resumen en inglés

This report is part of the supervised professional practice (PPS) of the Computer Engineering career that consists of the migration of an Android mobile application with monolithic architecture to one with modular architecture, also establishing a continuous integration and a continuous deployment scheme, in order to automate and improve development processes. In it, it describes how the migration step by step was, the choice of tools to carry out the project and the description of the implementations carried out to meet the objectives set out in the scope of this PPS. To conclude, the results, the simplicities and difficulties encountered during the development of the work and the conclusions and reflections on the development of software framed in professional practices are reflected.

Índice

Índice de figuras

1. Introducción

La Práctica Profesional Supervisada (PPS) nace en un marco de trabajo entre la contratación de FluxIT S.A y un tercero, con quienes se crea un equipo de trabajo denominado "CORE" conformado por distintos perfiles (arquitectos de software, desarrolladores back-end, front-end, dev-ops y referentes mobile) con el objetivo de poder tomar decisiones que impacten directamente en el desarrollo y en la aplicación de buenas prácticas por parte de la empresa contratante. Esto implica la definición de lineamientos, la propuesta de mejoras, investigación del uso de nuevas herramientas, análisis y soporte, entre muchas tareas a cargo.

Una de las responsabilidades y tareas que surgieron fue la de modularizar una APP nativa que, al momento de realizarse la presente PPS, se encuentra en pleno desarrollo y sumarla a un plan de integración continua, automatizando los procesos y facilitando el desarrollo de la misma.

En la empresa, las compilaciones de las aplicaciones se realizaban en una computadora on-site dentro del edificio, esto debido a que se necesitaba una computadora con sistema operativo Mac Os para poder compilar las aplicaciones iOS (en el mercado generalmente existen aplicaciones para sistemas Android y otras para iOS). Sin embargo, en medio de un contexto de pandemia, los empleados se vieron obligados a realizar trabajo remoto de forma indeterminada quedando la computadora encargada de las compilaciones automáticas en el edificio, sin nadie a cargo y pronta a desconectarse. Esta situación motivó a la búsqueda de una herramienta en la nube que permitiera automatizar los procesos de compilación y publicación para las aplicaciones tanto Android como iOS y de esta forma, facilitar los desarrollos.

La decisión de modularizar la aplicación actual se debe al gran número de equipos de desarrollo que existen trabajando sobre un mismo código fuente. Por este motivo, la empresa requiere del trabajo por verticales de negocio en una aplicación y, que cada vertical, tenga su propio equipo de desarrollo. Es clave que las aplicaciones nativas soporten este requerimiento, ya que muchas veces los equipos de desarrollo pertenecen a distintas empresas tercerizadas y deberían poder trabajar de forma independiente, es decir, cada una con su código fuente y luego

realizar una integración de los trabajos ejecutados dando como resultado una aplicación robusta.

Otro punto fundamental de efectuar la modularización es la creación de un módulo Core, que resuelva necesidades comunes a todos los módulos y que pueda ser utilizado como dependencia para cualquiera de las aplicaciones en construcción.

El presente trabajo expondrá el diseño, desarrollo e implementación de la migración de una aplicación móvil nativa y monolítica hacia una aplicación modular acoplada a un esquema de integración continua. 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 permitiendo la autogestión e independencia de los mismos.

1.1 Objetivos de la Práctica Profesional Supervisada

Los objetivos específicos de la PPS se encuentran detallados a continuación.

- **Lograr independencia de células** para que cada equipo pueda trabajar de manera independiente, de forma que se simplifique la división de trabajo y se consiga tener autonomía para la construcción de la app. Para alcanzar este objetivo es fundamental modularizar la aplicación y asignar responsabilidades por parte de los distintos equipos sobre los módulos correspondientes.

- **Desarrollar un módulo Core** que resuelva necesidades comunes no solo para los módulos de la aplicación en cuestión, sino también para futuras aplicaciones. Este módulo debe incluir: clases de networking, eventos internos y analíticos, logs, configuración de sharedpreferences e implementación de librerías de uso común (Crashlytics, Analytics, etc.)

- **Preparar un esquema de modularización** para en un futuro poder detectar y generar módulos funcionales. Esto implica plasmar las mejores prácticas para la creación de módulos, la publicación de los mismos en versiones abiertas (SNAPSHOT) o cerradas para su futuro consumo y el manejo adecuado de las dependencias en la aplicación.

- Armar CI/CD utilizando Bitrise: un esquema de integración continua que permita la construcción y publicación automática no solo de módulos, sino también, de aplicaciones.

1.2 Tareas a realizar

El desarrollo de la PPS estará dividido en 5 etapas.

- **Relevamiento (análisis)**

En primer lugar, se realizará un análisis en profundidad de la aplicación mobile "autogestión" para entender cómo funciona actualmente y lograr que se capten las necesidades que tiene el equipo desarrollador y la organización en sí.

- **Definición de requerimientos funcionales y no funcionales (diseño)**

A partir de la información recolectada en el relevamiento, se desarrolla la descripción de los requerimientos principales tomando como puntos principales la comprensión de la metodología de trabajo que tienen los desarrolladores. De este modo, poder definir una integración continua de la aplicación de acuerdo a las necesidades encontradas y la definición de un diagrama de funcionalidades, que permitan identificar las tareas a realizar, para la migración que corresponde a la modularización.

- **Definición de herramientas y tecnologías.**

Elección de las herramientas y tecnologías necesarias para poder llevar a cabo las tareas definidas. Esto implica reutilizar las ya disponibles por la organización, pero a su vez, definir nuevas herramientas y tecnologías que sirvan como complemento y ayuden a cumplir los objetivos planteados para el proyecto.

- **Desarrollo e implementación.**

Con las etapas anteriores finalizadas, se procede a efectuar la implementación de las tareas correspondientes a la integración continua para la aplicación y a ejecutar la migración y refactorización de código para ser utilizado por los módulos que se van a crear y utilizar.

Para la integración continua, se establecerán distintos flujos de trabajo (dependiendo los ambientes bajos de desarrollo y la puesta en producción) con tareas a ejecutar de acuerdo a la información obtenida en etapas anteriores.

Para la modularización, se definirán esquemas de versionado que estarán acoplados a la integración continua para que los equipos integren los cambios de manera segura y eficiente en la aplicación.

- **Verificación y mantenimiento.**

A medida que se finalicen las tareas, se informará a los referentes a cargo de la aplicación para que se realicen las integraciones para, posteriormente, ser probadas por los QA (Quality assurance) a cargo.

Se realizará un seguimiento de estas pruebas para poder resolver conflictos, en caso de que los haya, y se establecerán nuevas tareas si se requieren cambios.

1.3 Cronograma

En esta sección, se mostrará una tabla con la planificación del trabajo a realizar. Se destacan cuatro secciones principales que agrupan las actividades a ejecutar:

- Relevamiento y diseño inicial (en color naranja)
- Desarrollo e implementación (en color verde)
- Validación y conclusiones sobre el proyecto (en color violeta)
- Redacción de informe final PPS (en color amarillo)

Tareas	ETAPAS				
	Etapa uno	Etapa dos	Etapa tres	Etapa cuatro	Etapa cinco
Relevamiento y diseño inicial					
Analisis de aplicación autogestion					
Definicion de requerimientos para integracion continua					
Definición de requerimientos para la construcción de módulo Core y diseño del mismo (Diagrama de					
Definición de esquema de versionado y publicación de módulos					
Configuración de Artifactory6					
Desarrollo e implementación					
Configuración bitrise y diseño de workflows de app para integración continua					
Creación de módulo Core					
Configuración de workflows para construcción y publicación de módulos.					
Migración y refactorización de capa de networking (Core)					
Migración y refactorizacion de librerías analíticas y de logs (Core)					
Publicación de primeras versiones de Core e integración con app					
Integración de módulo Core en módulo Login					
Migración y refactorización de Feature Flags (Core)					
Manejo de dependencias entre módulos					
Publicación de nuevas versiones de todos los módulos e integración con app					
Validacion y conclusiones sobre el proyecto					
Redacción de informe final PPS					

2. Marco teórico

El presente capítulo, que se desarrolla a continuación, permite conocer los conceptos base sobre los que se desenvuelve el desarrollo de esta Práctica Profesional Supervisada.

En primer lugar, se abordará la definición de “CI/CD” con el fin de entender la importancia y los beneficios que otorga este modelo y cómo impacta en los procesos de desarrollo de software.

En segundo lugar, se describirá la modularización para aplicaciones Android, con el objeto de comprender la forma en que este tipo de proyecto logra adecuarse a las metodologías y procesos que llevan las organizaciones a la hora de realizar desarrollo de software.

Con estas descripciones teóricas se podrá comprender el desarrollo de la Práctica Profesional Supervisada que se expone más adelante.

2.1 CI/CD

El desarrollo de las aplicaciones modernas tiene como objetivo contar con múltiples desarrolladores que trabajen de forma simultánea en distintas funciones de la misma aplicación. No obstante, si en el proyecto se decide fusionar todo el código fuente, que fue ramificado en un solo día, podría resultar en tareas tediosas y manuales, que además pueden llevar mucho tiempo. Esto ocurre, porque cuando los desarrolladores que trabajan de forma aislada sobre el código fuente (pudiendo ser sobre un Branch específico) implementan un cambio en la aplicación, existe la posibilidad de que esta modificación entre en conflicto con los distintos cambios implementados simultáneamente por otros desarrolladores.

La integración continua o CI, un modelo informático propuesto inicialmente por Martin Flower, ayuda a que los desarrolladores fusionen los cambios que introducen en el código para incorporarlos en un Branch con mayor frecuencia, inclusive dando la posibilidad de que esto ocurra diariamente. Cuando finalmente los cambios implementados por los desarrolladores son fusionados, se validan con el desarrollo automático de la aplicación y disparan distintas tareas que ejecutan pruebas automatizadas (pueden ser test de unidad o de integración generalmente) para verificar que los cambios en la aplicación no hayan generado errores. Este procedimiento implica probar todo lo que compone la aplicación, desde clases y el funcionamiento hasta librerías y los distintos módulos que componen toda la aplicación. Si en el proceso de ejecución de las tareas (pruebas automáticas y demás acciones) alguna falla, entonces se detecta un conflicto entre el código nuevo y el actual, la integración continua facilita la lectura de logs, el análisis y la resolución de esos errores con frecuencia y rapidez.

Imagen 01 *– Proceso estándar de CI.*

Fuente: Recuperado de https://www.campusmvp.es/recursos/image.axd?picture=/2019/3T/ProcesoCI.png (2019)

Una vez que el proceso de integración continua fue validado y no se detectaron errores, se procede a la etapa de "CD" que incluye el *continuous delivery* y el *continuous deployment*. El código fusionado se carga en un repositorio (puede ser por ejemplo GitHub, Bitbucket, etc.) para que finalmente se pueda poner en entorno de producción (proceso que también se puede automatizar).

La etapa final consolidada de la integración y distribución continua es la implementación continua, que automatiza el lanzamiento de una aplicación a producción. Esto, depende en gran medida, del correcto diseño de la automatización de pruebas y tareas a ejecutar.

La implementación continua implica que los cambios implementados por uno o varios desarrolladores en una aplicación, se puedan poner en marcha unos minutos después de

escribirlo (habiendo pasado todas las pruebas automatizadas y las tareas programadas). Estas prácticas de CI/CD hacen que la implementación de una aplicación minimice riesgos y que se puedan realizar entregas con cambios en las aplicaciones en fragmentos pequeños y no hacerlo todo de una vez. No obstante, implica la realización de muchas inversiones iniciales, como lo son el diseño de las pruebas automatizadas, la realización de los *scripts* que ejecuten las diversas tareas particulares que se adapten a las distintas etapas de construcción, pruebas y liberación en el canal de la CI/CD.

La CI/CD puede especificar desde las prácticas relacionadas de integración y distribución continua solamente, o las tres prácticas vinculadas de integración continua, distribución continua e implementación continua. Cabe destacar, que el término "continuous delivery" también abarca los procesos de la implementación continua o "continuous deployment". Solo se debe recordar que la CI/CD son procesos que suelen percibirse como una canalización e implica incorporar un alto nivel de automatización permanente y supervisión constante al desarrollo de las aplicaciones, estas colaboran en ordenar y gestionar procesos para solucionar o evitar diferentes errores.

Imagen 02 *– Proceso estándar de CI/CD.*

Fuente: Recuperado de https://www.redhat.com/es/topics/devops/what-is-ci-cd (2021)

Principales beneficios del CI/CD:

- **Mejorar la calidad de los productos**: el proceso de pruebas automatizadas por el que pasa cada cambio realizado en una nueva construcción de la aplicación, conlleva que los desarrolladores implementen buenas prácticas y traten siempre de tener la mayor calidad posible en el diseño y desarrollo del producto.

- **Riesgo de implementación reducido:** al estar implementando cambios más pequeños, hay menos problemas y es más fácil solucionar los que aparezcan.

- **Detección de funcionamientos anómalos:** los reportes y los resultados finales que se ofrecen al final de cada integración, traen en consecuencia un análisis en detalle y derivan en la determinación de bugs o funcionalidades nuevas a agregar.

- **Facilita la comunicación entre equipos distintos:** los equipos comerciales y de negocio pueden tener mayor visibilidad y comunicación con los de desarrollo, ya que pueden ver un producto tangible y recibir continuamente entregas del mismo para poder realizar pruebas y dar el visto bueno sobre lo realizado, o también establecer nuevas decisiones respecto al desarrollo de nuevas funcionalidades o mejoras.

- **Progreso creíble:** generalmente se hacen los seguimientos del proceso mediante el seguimiento del trabajo realizado. Si una tarea marcada como “done” significa “los desarrolladores terminaron su trabajo”, no es tan creíble como si se implementará el trabajo en un entorno de producción (o similar a producción) ya que, en el último caso, hay mayor visibilidad de lo realizado.

- **Comentarios del usuario:** Uno de los mayores riesgos en el desarrollo de software es que se construya algo que no es útil. Cuanto antes y con más frecuencia se tenga el software en funcionamiento frente a usuarios reales, más rápido se obtendrá comentarios para descubrir cuan valioso es realmente.

2.3 Modularización

La modularización consiste en separar la aplicación en distintas unidades denominadas módulos que permite encapsular de una forma más estricta todo el código que se incluya dentro de ellos.

El término de módulo en Android proviene de la propia configuración de Gradle que permite dividir el código en módulos y luego interrelacionarlos y realizar comunicaciones entre ellos mediante la declaración de los mismos como dependencias.

Siempre que se crea una aplicación Android desde cero, el proyecto se construye con un módulo principal denominado APP donde generalmente se coloca todo el código dentro del mismo. Una estructura de esta forma es denominada monolítica, dado que todos los aspectos funcionales están acoplados y sujetos en un mismo programa.

Este tipo de aplicación implica que, las reglas de encapsulación llevadas a cabo en el desarrollo no queden demasiado claras y todo el código queda disponibilizado y accesible para cualquier desarrollador, ya que se encuentra en un mismo módulo sobre el cual todos trabajan.

Hay distintos enfoques de modularización que deben ser elegidos dependiendo el proyecto, equipo y organización. El primer enfoque es el de modularización por capas, donde todos los módulos de funciones se combinan en el módulo de aplicación. Es decir, se crean separaciones por capas dentro del proyecto en donde se agrupan clases relacionadas a un dominio específico. Algunos ejemplos claros son los siguientes: la separación en una capa de red, una capa de datos y una de commons. Estas capas son denominadas módulos, pero se encuentran dentro de la aplicación modulo, por lo que la APP sigue siendo monolítica solo que contiene una mayor organización y estructuración. Este enfoque se suele escoger en aplicaciones pequeñas o con pocos desarrolladores.

El segundo enfoque más conocido es el de la modularización por *features*. Las funcionalidades de la aplicación son separadas en “módulos funcionales”, a través de la estrategia de Gradle en donde cada módulo se construye como una “library” de Android. Esto da como resultado, paquetes de código independiente, que se interrelacionan como dependencias entre sí y pueden ser utilizadas en varias aplicaciones.

Un beneficio importante en la modularización por *features* es la mejora en tiempos de compilación. Esto se da gracias a una *feature* en Gradle llamada “build caché”, que permite cachear (almacenar en memoria temporal) todas las compilaciones. Si, por ejemplo, una aplicación tiene diez módulos, cuando se compila por primera vez, se compilan todos los módulos, pero a la siguiente vez, solo compilan aquellos módulos en los cuales haya código

modificado. Si no hay código modificado no se vuelve a compilar y, por lo tanto, los tiempos de compilación son mucho menores.

Otro punto en este enfoque es la creación de un módulo Core, que debe tener funcionalidades e implementaciones de uso común para el resto de los módulos o aplicaciones que lo utilicen. Es fundamental colocar solo el código que va a ser reutilizado y que no deba ser implementado para distintas soluciones, de otra forma generaría que el módulo crezca cada vez más y perdería el sentido de su uso.

La idea de abstracción por capas también puede ser implementada desde el enfoque de *features*. Se pueden usar estrategias y herramientas como, por ejemplo, los "Visibility modifiers" de Kotlin, que permiten determinar qué clases son públicas, para todo acceso o limitadas a un paquete o módulo, etc. De esta forma, se limitaría el acceso a ciertas clases y se ocultarían ciertas implementaciones dentro del proyecto.

3. Desarrollo

Teniendo en cuenta los conceptos vistos en el capítulo 2, este apartado describirá el desarrollo que se aplicó en el trabajo, profundizando el detalle de los temas tratados e indicando el paso a paso de las tareas que fueron ejecutadas para la concreción de los objetivos buscados.

Esto implica la descripción de las herramientas empleadas, su integración y utilización en este proyecto.

3.1 Tecnologías y herramientas

A lo largo del trabajo se utilizarán diferentes herramientas y tecnologías. A continuación, se detallan las principales:

Bitrise

Bitrise es una herramienta que brinda una solución PaaS para el continuous integration y continuous deployment (CI/CD) de aplicaciones móviles.

Permite simplificar al máximo las tareas que se relacionan con la compilación del código fuente, ejecuciones de prueba de calidad de código e integración con los Stores. Estas tareas se encuentran asociadas en cualquier flujo correspondiente a la construcción de una aplicación y son agrupadas en lo que se llama Workflows.

La utilización de esta herramienta apunta a la transformación digital de un proceso que, hasta el momento, se realizaba de manera manual en pos de evitar errores humanos y aumentar en *time-to-market* de las APPS. Se entiende como *time-to-market*, al plazo de lanzamiento que referencia el tiempo comprendido desde que un producto o servicio es concebido hasta que está disponible para el usuario final.

Bitbucket

Bitbucket es una herramienta de alojamiento de código y colaboración basada en Git diseñada para equipos. Esta presenta integraciones con Jira y Trello de tal forma que, en su conjunto, logran unir a todo el equipo de software con el fin de poner en práctica un proyecto. Además, facilita la colaboración por parte de los equipos, en lo que respecta al código, desde el concepto hasta la nube, la creación de código de calidad mediante pruebas automatizadas e implementación de código con total seguridad.

Jira Software

Jira es una herramienta en línea que ayuda a los equipos a planificar, asignar, supervisar y gestionar el trabajo, así como a elaborar informes al respecto. Uno de los puntos importantes a la hora de planificar es la posibilidad que otorga la plataforma de crear historias de usuario e incidencias, permitir la planificación de *sprints* y distribuir tareas entre el equipo de software. De esta forma, se crean lazos entre los equipos en todos los ámbitos, desde el desarrollo de software ágil y la atención al cliente hasta la gestión de empresas emergentes y de grandes empresas.

Confluence

Confluence es un software de colaboración en equipo en donde se permite crear y organizar el trabajo en un solo lugar desde, virtualmente, cualquier sitio. La plataforma se organiza por páginas y espacios, y está estructurada de una manera intuitiva facilitando la configuración, la creación y el descubrimiento.

Por una parte, se entiende por páginas a los documentos en los que las personas crean, editan y debaten sus trabajos. Por otra parte, los espacios son áreas que contienen páginas para personas, equipos y proyectos estratégicos.

Además de crear una base de conocimientos de documentación y requisitos de producto con el fin de ser compartida, también se destaca la posibilidad de expandir las funcionalidades de la herramienta por el soporte de plugins de terceros que tiene.

Kotlin

Es un lenguaje de programación pragmático, que fue diseñado en un principio para superar a Java, pero sin dejar de ser interoperable con el código Java, inclusive sin la necesidad de capas de adaptación. El propósito de estas funcionalidades no es otro que facilitar una migración de este último hacia Kotlin.

Puede ser utilizado en cualquier tipo de desarrollo, basado en servidor web de cliente; aunque, actualmente su uso está muy extendido en el desarrollo de aplicaciones móviles en Android.

Como principales características, este lenguaje de programación elimina el código repetitivo, distinguiéndose por su exactitud y claridad, lo que permite reducir notablemente los errores comunes de código. Posee funcionalidades como delegaciones, inicializaciones tardías, eliminación de NullPointerException y, también, aborda la seguridad de tipos en las listas, entre otras funcionalidades importantes que corrigen viejos problemas de Java.

Cabe destacar, que es un proyecto gratuito y de código abierto registrado bajo la licencia de Apache 2.0. El código del proyecto se desarrolla abiertamente en GitHub y está a cargo principalmente del equipo empleado en JetBrains.

Un punto importante es que fue declarado por Google como lenguaje oficial en Android, por lo que es el lenguaje recomendado para el desarrollo de aplicaciones nativas de este tipo.

JFrog Artifactory6

JFrog Artifactory es un repositorio de artefactos universal, que posibilita la administración de todos los formatos de empaquetado (en cualquier tipo de lenguaje), herramientas de compilación y servidores de CI principales.

Su principal ventaja es centralizar en único sitio y en único espacio de almacenamiento todas las dependencias de un proyecto y su gestión durante los ciclos de desarrollo, incluyendo los diversos ambientes de desarrollo que pueda tener. Esto permite que los equipos de trabajo eviten usar diversos repositorios, con distintas versiones o distintas fuentes.

Al ser un repositorio universal, se puede integrar con el entorno deseado y brinda la libertad de elegir la pila de herramientas. Esto ofrece la utilidad de unir un ecosistema de CI/CD, aumentando la productividad del desarrollador y evitar bloqueos de proveedores.

Gradle

Es una herramienta que permite la automatización de compilación de código abierto, que está diseñada para ser lo suficientemente flexible para compilar diversas tecnologías y lenguajes, pero en especial Android ya que es su sistema de compilación oficial. Los *scripts* de compilación de Gradle se codifican utilizando Groovy o Kotlin DSL (Domain Specific Language).

Una característica importante es la gestión de dependencias que ofrece, en donde admite repositorios y sistema de archivos compatibles con Maven e Ivy, encargándose de descargar y administrar las dependencias transitivas. De esta misma forma, también permite publicar Artifacts en los repositorios nombrados con diseños de directorios completamente personalizables.

3.2 Implementación CI/CD

El desarrollo de la aplicación en Android es llevada a cabo por un conjunto de equipos denominados PODS. Cada POD es un equipo que con diversas capacidades trabaja en colaboración para lograr las necesidades planteadas por el cliente, de forma autoorganizada, colaborativa y autónoma.

Cada equipo está enfocado en un área funcional específica de la aplicación, trabajando en conjunto sobre un mismo repositorio y siguiendo los lineamientos del esquema de *branching*.

TBD (Trunk Based Development).

Para que los equipos puedan entregar cambios de código de manera más frecuente y fiable, se integró el repositorio de la aplicación, manteniendo el esquema de *branching* utilizado por los equipos, a una CI/CD implementada en Bitrise.

Para realizar esta implementación, primero se efectuó un exhaustivo análisis sobre los procesos de calidad que se llevaban a cabo a la hora de ejecutar los desarrollos en la aplicación y sobre cómo eran los procesos de prueba a la hora de integrar cambios o construir nuevas versiones de la APP.

Teniendo cuenta los puntos anteriores, con el proyecto ya subido en Bitrise, se realizaron todas las configuraciones necesarias para poder construir los distintos Workflows que permitieran automatizar las distintas tareas y cumplir con todos los requisitos de calidad, integración y distribución en los distintos ambientes en que la APP puede ser ejecutada.

Los ambientes en que la APP puede ser ejecutada son:

- Release: Ambiente productivo.
- Dev: Ambiente de desarrollo.
- QA: Ambiente de testing.
- UAT: Ambiente de UAT (User Acceptance Testing).

3.2.1 Bitrise como herramienta PaaS

Para poder aplicar un esquema de CI/CD es importante tomar la decisión de escoger una herramienta acorde a las necesidades que tiene la empresa. Esto implica realizar un análisis y comparación entre los distintos productos que hay en el mercado, costos y demás variables.

La elección de Bitrise como servicio PaaS para la implementación del proceso de CI/CD viene solventada por el enfoque que tiene en el desarrollo de aplicaciones móviles (iOS, Android, React Native, Flutter, etc.). Su ventaja sobre herramientas como Jenkins y Fastlene, el último con un alto costo de mantenimiento para servidores Mac OS X a la hora de realizar compilaciones para iOS y su simplicidad en la plataforma (buen trabajo de UI/UX) inclinaron la balanza en la elección de esta herramienta. Otro punto, no menos importante, que sirvió para la elección de esta plataforma fue la experiencia en proyectos anteriores, donde demostró ser eficiente y, además, tuvo la participación de varias personas del proyecto actual.

3.2.2 Editor de flujo de trabajos (Workflows)

Workflows

Un Workflow o flujo de trabajo en Bitrise es una colección de *steps* que, al realizar una compilación de una aplicación, se ejecutan en el orden en que fueron definidos. Es el sector del proyecto en donde se configuran las compilaciones, se prueban e implementan las subidas a las tiendas (AppStore, Google Play Store, etc.). Además, se permite agregar certificados para firmado (provisioning profile en el caso de iOS y keystores para Android), configurar variables de entorno, *secrets* e inclusive implementar *triggers*.

Los Workflows se pueden crear, definir y modificar de dos formas:

- Utilizando el editor de Workflow gráfico, que se provee en la plataforma de Bitrise.
- Editando directamente el archivo YML, que fue generado en el proyecto (se crea por defecto al subir una aplicación en Bitrise).

Cabe destacar que cualquier cambio que se realice desde el editor de Workflow gráfico impacta directamente en el archivo YML. El editor de Workflow es solo una manera más

amigable de hacerlo, por lo que las ediciones se pueden realizar desde cualquiera de los dos caminos.

De forma predeterminada, una sola compilación es un solo Workflow, pero también es posible encadenar Workflows para que se ejecuten en sucesión. De esta manera, se pueden atomizar las tareas por flujos de trabajo privados y encadenarlos en un Workflow primario para que se pueda ejecutar. Esto permite reutilizar los *steps* y realizar mínimos cambios para la ejecución de pipelines de diferentes ambientes bajos.

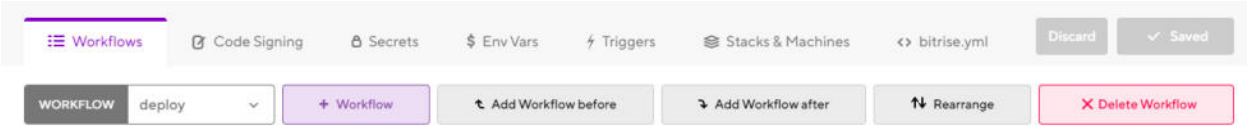

Imagen 03. *Editor de Workflow con todas las opciones disponibles.*

Fuente: Elaboración propia, basada en la práctica.

Code signing

Las aplicaciones móviles necesitan ser firmadas digitalmente como una forma de garantizar que el código no se haya alterado desde que se firmó. La firma de código es el proceso donde justamente se realiza una firma de la aplicación, proporcionando seguridad para la implementación e identificando al autor.

Para que las aplicaciones puedan ser subidas en una tienda en línea, como Google Play o App Store, el firmado de código es fundamental. Por esto, se encuentra la pestaña “code signing” donde se puede cargar el archivo **Android keystore file**.

Una vez que se carga el archivo del almacén de claves, automáticamente se genera una URL que hace referencia a la ruta donde se aloja. Bitrise asigna una variable de entorno (BITRISEIO_ANDROID_KEYSTORE_URL) a la URL de descarga del archivo como valor (que es una URL de descarga de solo lectura por tiempo limitado).

Finalmente se cargan las configuraciones correspondientes al Android keystore file (password, alias y key password) y se exportan automáticamente como variables de entorno. De esta forma quedan disponibles las entradas para la configuración del *step* de Bitrise

(**Android sign**) que se encarga de firmar la aplicación cuando se realiza la construcción y compilación.

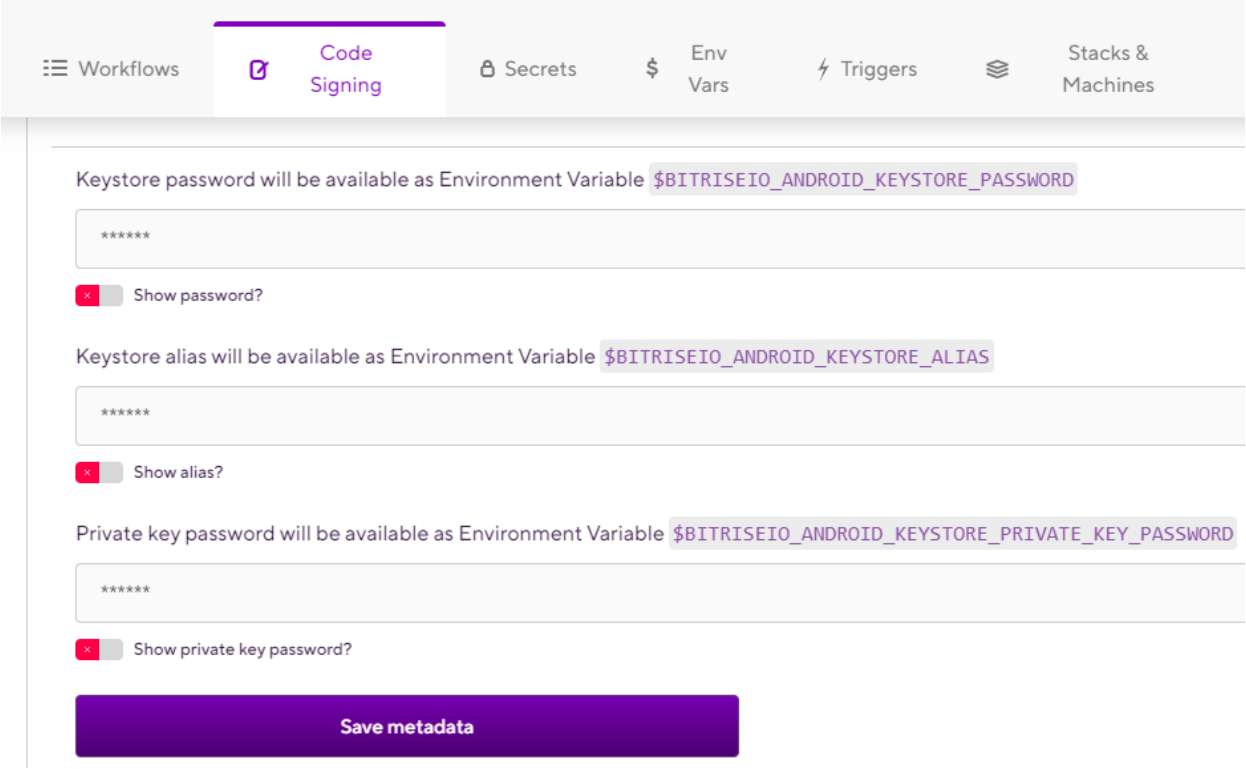

Imagen 04. *Pestaña "Code Signinig".*

Fuente: Elaboración propia, basada en la práctica.

Secrets

En esta sección se permiten agregar *secrets*, que son variables que contienen información sensible y no se desean exponer en el archivo de configuración de compilación (*bitrise.yml*).

Las variables definidas en los *secrets* se pueden usar como cualquier otra variable de entorno con la diferencia que, a la hora de realizar la compilación, el CLI de Bitrise filtra automáticamente los *secrets* e imprime [REDACTED] con nuevas líneas después de la clave. Todo esto para que los *secrets* no sean visibles en el registro de compilación.

Un claro ejemplo de uso de estas variables puede ser para los valores de usuarios y contraseñas, tokens, etc.

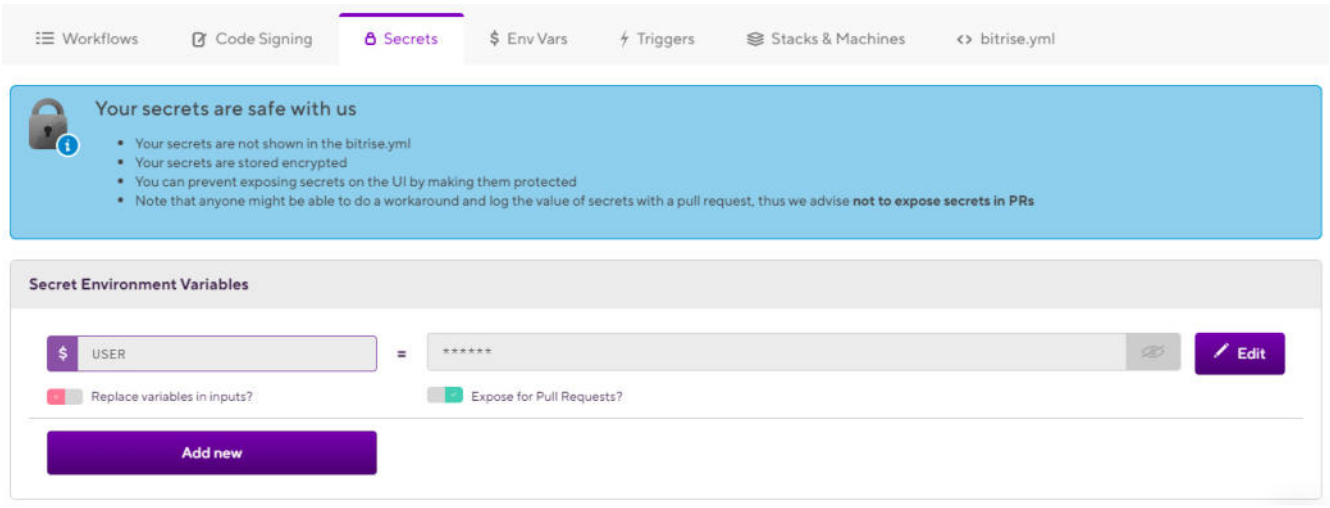

Imagen 05. *Pestaña de configuración "Secrets".*

Fuente: Elaboración propia, basada en la práctica.

La propiedad "Expose for Pull Requets?" permite que cuando una compilación se dispara por medio de un *trigger*, en donde se ejecutó un Pull Request, Bitrise pueda leer esta información.

Env vars

Las variables de entorno son datos almacenados a través de una clave y un valor. En esta sección del proyecto en Bitrise, se permite mantener una colección de todas las variables de entorno registradas para la aplicación.

Pueden ser registradas a nivel aplicación (se pueden utilizar en cualquier Workflow y hacer referencia tantas veces como se desee) o a nivel Workflow. En este último caso, es una buena práctica crear Workflows por ambientes bajos (producción, uat, test, dev, etc.) y configurar las variables de entorno para cada Workflow con la misma clave, pero distinto valor. Esto significa que, en tiempo de compilación, se va a utilizar la misma variable, pero con el valor correspondiente para el ambiente que se quiera apuntar.

Una vez registradas las "Env vars", pueden utilizarse para cualquier campo de entrada de los *steps* del Workflow elegido. Se escoge la opción insert variable y se coloca la variable correspondiente (anteponiendo siempre el signo "$").

Al contrario de los *secrets*, las variables de entorno quedan completamente expuestas en compilaciones desencadenadas por pull request, además de que quedan a la vista en el archivo *bitrise yml* y en los registros. Por esto, no es recomendable agregar ningún tipo de información sensible en Env Vars.

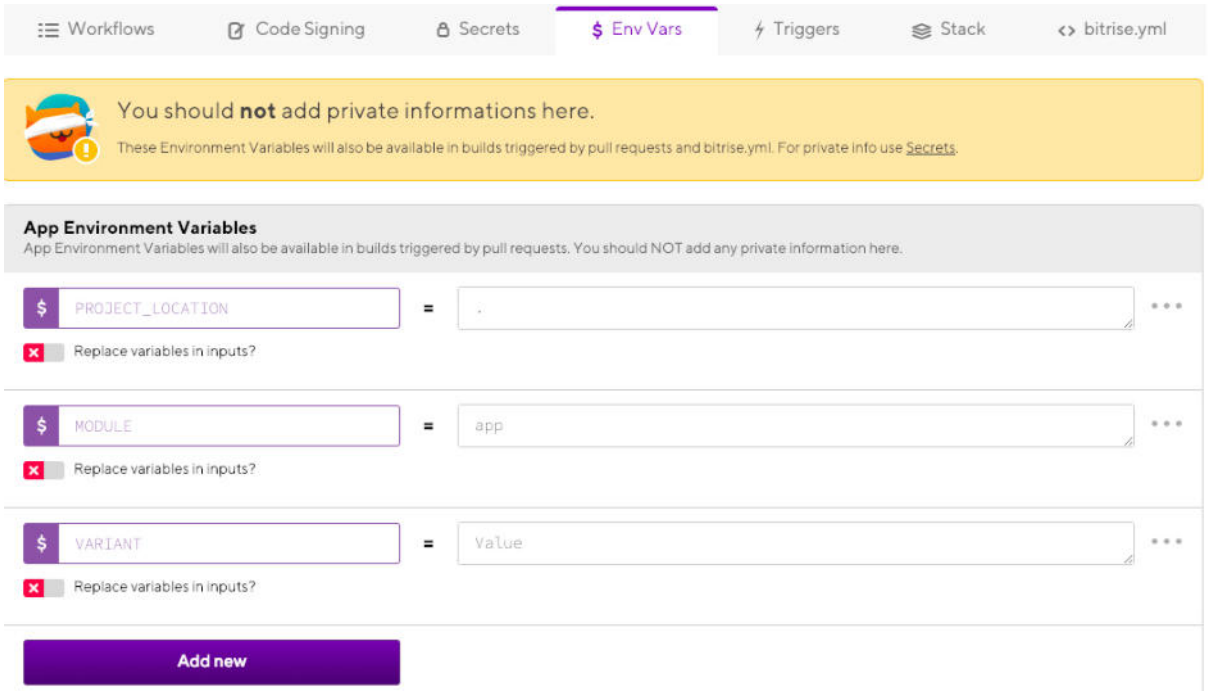

Imagen 06. *Variables de ambiente o entorno.*

Fuente: Elaboración propia, basada en la práctica.

Triggers

Generalmente, cuando se registra un Webhook para un evento o varios eventos (por ejemplo, para un push de código y para pull request de eventos), el servicio de alojamiento de código fuente (Gitlab, Bitbucket, etc.) va a llamar al Webhook cada vez que ocurra el evento relacionado.

En Bitrise, estas llamadas de Webhook se denominan *triggers* y se pueden asignar a diferentes Workflows, o no asignarse en ninguno.

Las posibilidades de *triggers* que ofrece Bitrise son las siguientes:

- Push: se puede indicar un Branch desde donde se haga un push del código fuente. Esto desencadena el Workflow indicado.
- Pull Request: se indica un *source* Branch (código fuente desde el cual se parte) y un Branch target (hacia donde se va a mergear el código fuente estipulado). Esto desencadena el Workflow indicado.
- Tag: se puede indicar un tag específico que desencadene el Workflow indicado.

En todos los casos, se pueden configurar los *triggers* utilizando el "*" símbolo como comodín en los nombres de los branchs o tags.

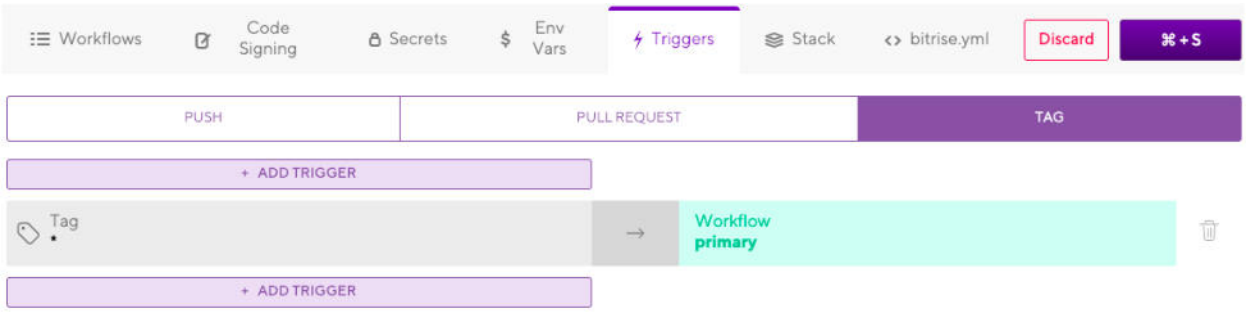

Imagen 07. *Ejemplo de Triggers.*

Fuente: Elaboración propia, basada en la práctica.

Stack

El stack indica la versión de la máquina virtual que se usará para ejecutar la compilación. Cuando se sube la aplicación, se selecciona una pila adecuada para el proyecto. Desde esta pestaña, en editor de Workflow, se puede modificar.

Generalmente para los proyectos de Android se escoge la pila "Android & Docker, on Ubuntu 20.04" aunque, como se nombró anteriormente, se puede modificar dependiendo la infraestructura que se necesite y el plan de pagos que se haya contratado con la plataforma Bitrise.

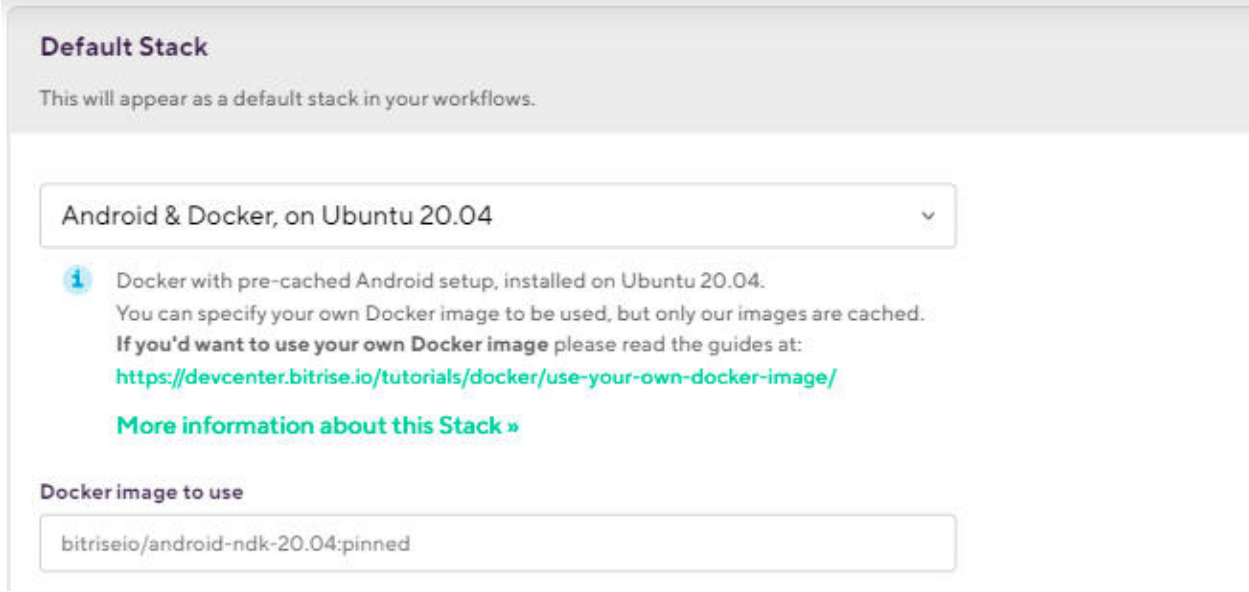

Imagen 08. *Representación de la configuración del stack.*

Fuente: Elaboración propia, basada en la práctica.

File Bitrise.yml

Es la representación de la configuración de la aplicación. Es un archivo YAML que define todos los *steps*, las variables configuradas, los Workflows creados, los *triggers*, básicamente todo lo referido a la construcción y compilación del proyecto. En el editor de Workflow se puede editar de forma visual a través de la interfaz de usuario web o desde este archivo donde se puede ver la configuración en un formato YAML, así como también se puede editar la configuración.

```
format_version: 5
default_step_lib_source: https://github.com/bitrise-io/bitrise-
steplib.git
project_type: android
app:
  envs:
  - MY_NAME: My Name
workflows:
  test:
    steps:
    - script@1.1.5:
        inputs:
        - content: echo "Hello ${MY_NAME}!"
```

Imagen 09. *Ejemplo de archivo yml.*

Fuente: Recuperado de https://devcenter.bitrise.io/bitrise-cli/basics-of-bitrise-yml/ (2021)

3.2.3 Carga de APP como proyecto en Bitrise

Crear la aplicación y dar acceso al repositorio de código

Al tener una cuenta en Bitrise, se permite añadir una aplicación por medio de la web UI o por la CLI siempre y cuando se haya iniciado sesión en la plataforma.

La aplicación fue subida por medio de la web UI desde el panel de Bitrise, seleccionando el repositorio de código con el cual se trabaja (se puede elegir entre Github/ Bitbucket /GitLab, o agregar el repositorio manualmente) y brindándole el acceso correspondiente.

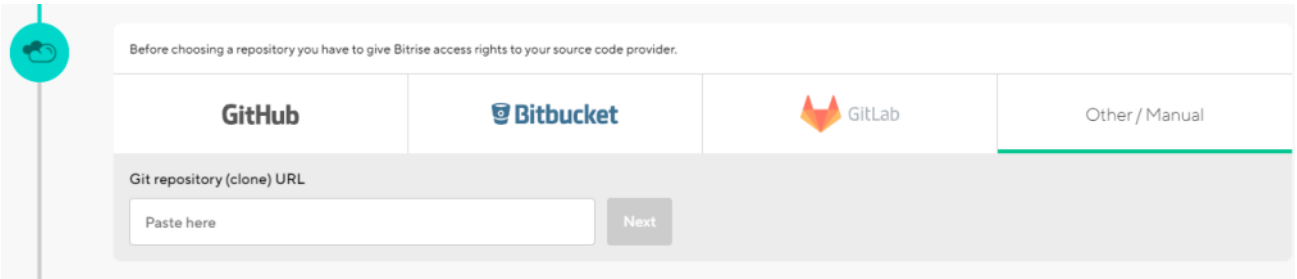

Imagen 10. *Configuración de repositorio de código para subida de APP.*

Fuente: Elaboración propia, basada en la práctica.

Al no haber una cuenta de servicio aún creada para la plataforma, y ante la previsión de una migración de repositorios desde Bitbucket a GitLab, la única manera de conectar el código fuente con Bitrise era por medio de una clonación HTTPS desde Bitbucket.

Esto representó un problema, ya que no era posible conectar el repositorio de Bitbucket mediante la URL HTTPS en el paso nombrado. Para solventar esta dificultad, se debió cargar un repositorio público de un proyecto Android y luego, una vez creado el proyecto, modificado la URL del repositorio por el que realmente correspondía (el de la aplicación autogestión), agregando ciertas tareas en el Workflow que permitieran autenticar con el repositorio (posteriormente se explicará este paso).

Especificar la rama

Luego, se escogió la rama principal (master), y la plataforma realizó un escaneo para validar el proyecto y detectar la mejor configuración de compilación para la pila (configuraciones de máquinas) y las plantillas de *steps* que se colocan por defecto para la construcción del proyecto.

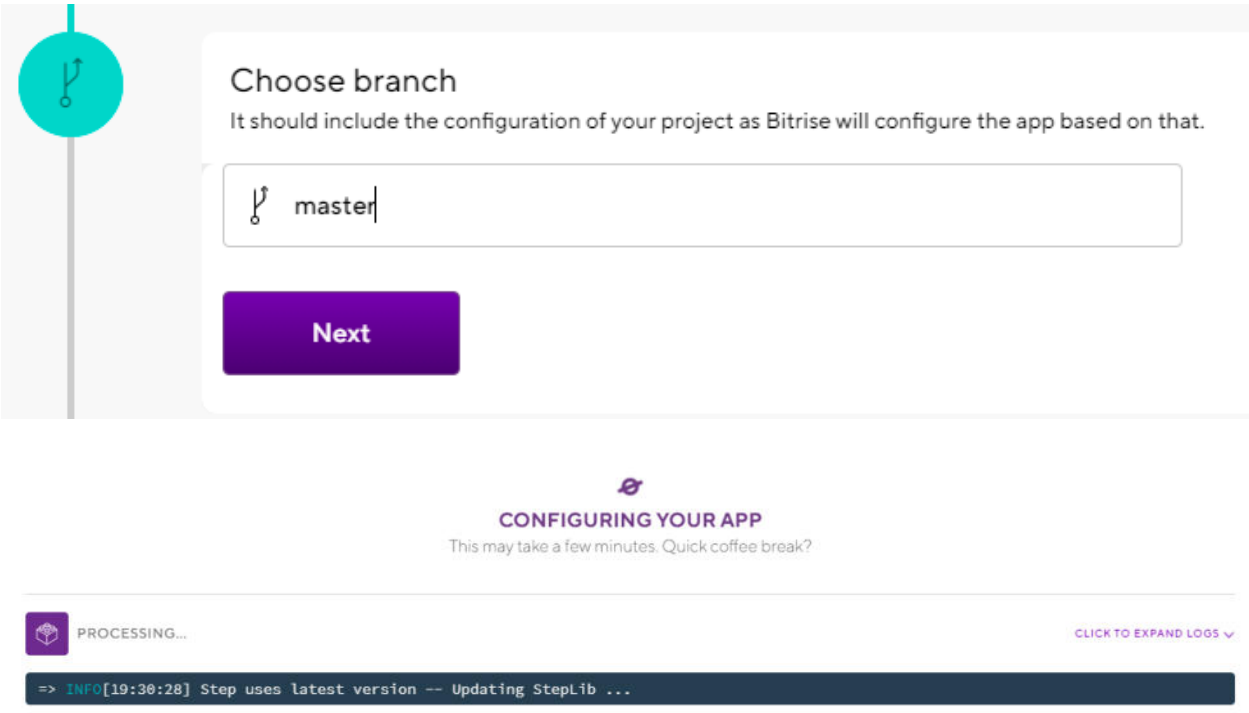

Imagen 11. *Selección de Branch de repositorio de código fuente.*

Fuente: Elaboración propia, basada en la práctica.

Al ser una aplicación desarrollada en Android nativo, la plataforma detectó correctamente la pila y estableció las configuraciones necesarias. Cabe destacar que en caso de que no se detecte, se puede intentar elegir manualmente.

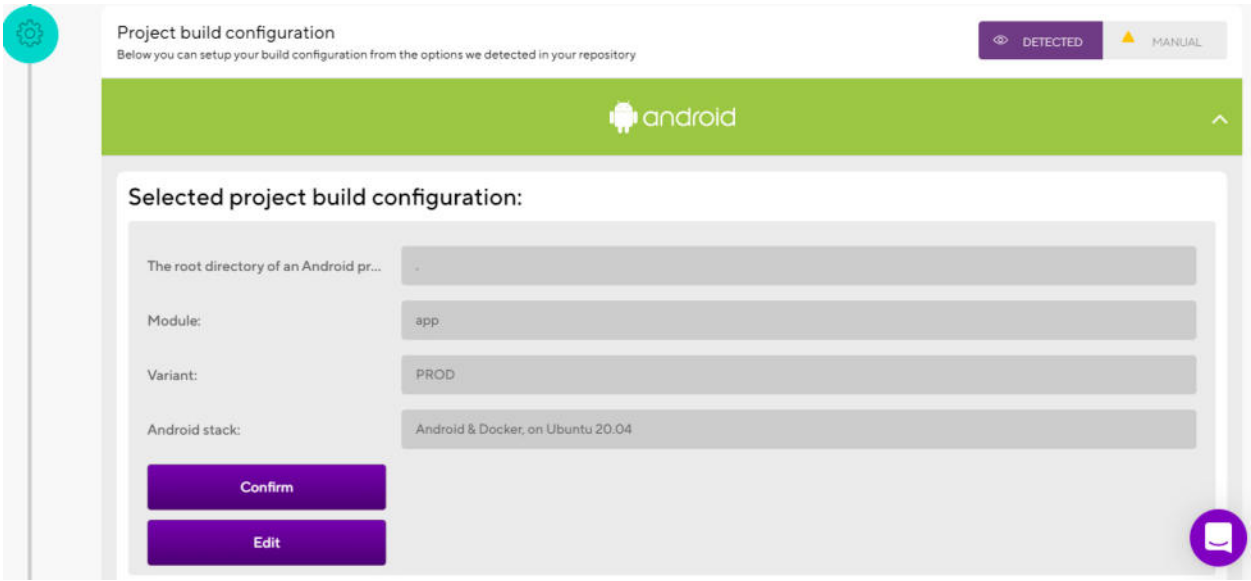

Imagen 12. *Configuración inicial de la APP subida.*

Fuente: Elaboración propia, basada en la práctica.

Una vez que se concretaron todos estos pasos, se seleccionó una imagen para que figure un ícono para la APP, concluyendo con la subida de la aplicación para ponerla a disposición en Bitrise y poder comenzar el proceso de CI/CD.

Con el proyecto ya integrado en Bitrise, se inició con las distintas configuraciones de CI/CD desde el dashboard:

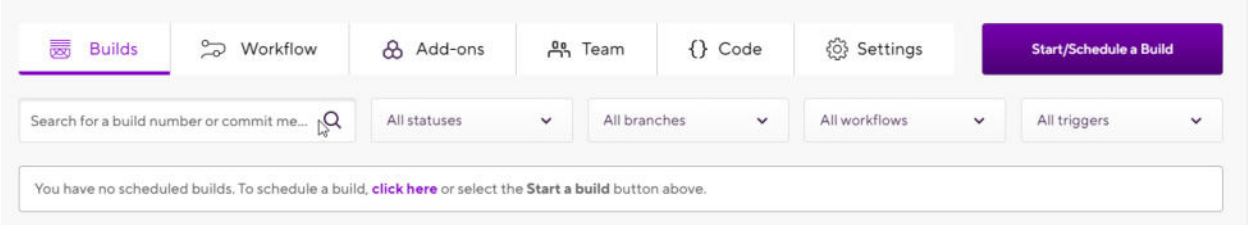

Imagen 13. *Dashboard del proyecto con todas sus opciones de configuración.*

Fuente: Elaboración propia, basada en la práctica.

3.2.4 Configuración base de proyecto de la APP

- **Webhook**

Siguiendo la estrategia de TBD (Trunk Based Development), fue necesario realizar la configuración de un Webhook para que Bitrise sea capaz de capturar los eventos de Pull Request que se realizaban desde un branch hacia otro en el repositorio donde se alojaba el código fuente de la aplicación. En consecuencia, a este evento, ejecutar un pipeline (con un Workflow específico dependiendo las necesidades) de manera automática.

Desde la pestaña "**{} Code**" dentro del proyecto, se encuentra la opción "INCOMING WEBHOOKS", que es la que permite obtener manualmente la URL del Webhook para Bitbucket.

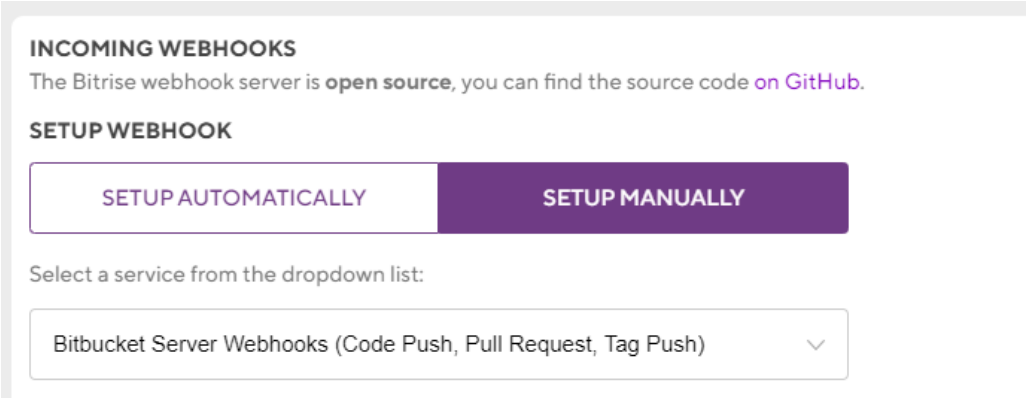

Imagen 14. *Opción para obtener URL de Webhook para Bitbucket.*

Fuente: Elaboración propia, basada en la práctica.

Esta URL obtenida fue agregada por la persona con permisos de Administrador en Bitbucket (DevOps) en el repositorio de la APP. Además, una vez que se agregó esta configuración también se tuvieron que especificar ante qué eventos se podrían disparar acciones.

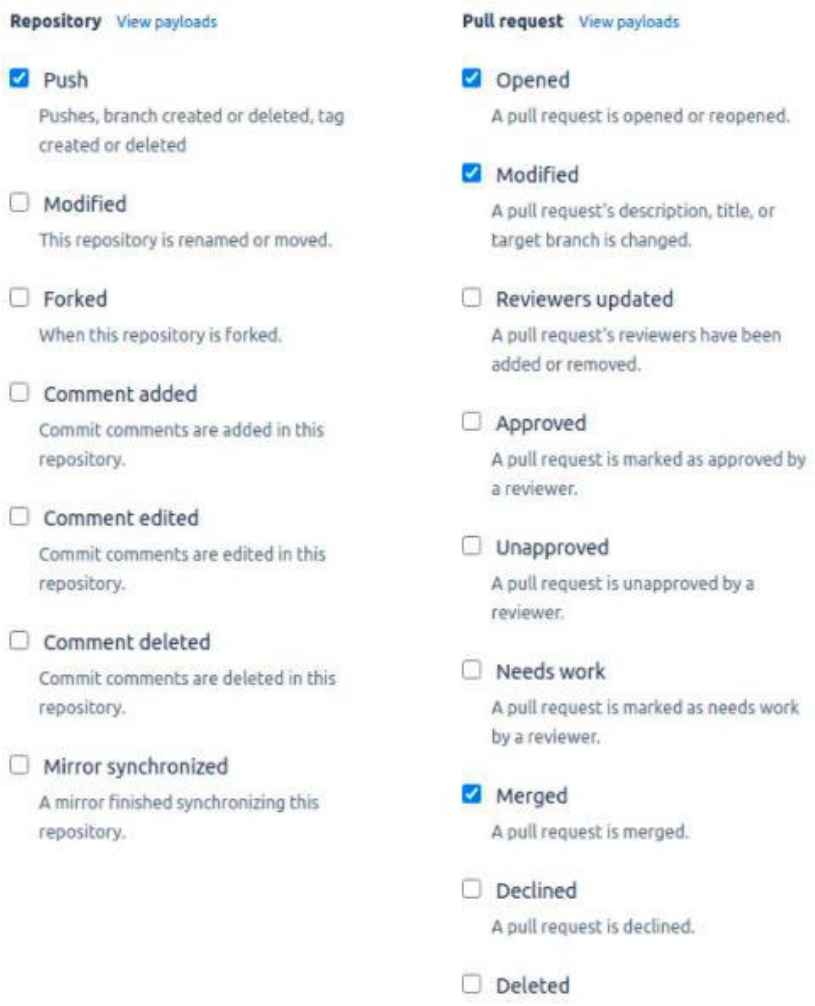

Imagen 15. *Configuración de Webhook de repositorio de APP.*

Fuente: Elaboración propia, basada en la práctica.

De esta forma, el repositorio de la APP alojado en Bitbucket quedó enlazado con el proyecto armado en Bitrise y a la espera de ejecutar cualquier tarea que se especifique ante la ejecución de los eventos seleccionados.

- **Groups & Team Members**

Todas las personas involucradas en el desarrollo (developers) y testeo (QA) se registraron en Bitrise y se los añadió a un grupo creado (Team Member) llamado

“Autogestion”. De esta forma, se permitió restringir los permisos y accesos a los distintos proyectos que fueron agregados en la plataforma.

El Team Member “Autogestión” fue asignado con el rol de tipo **Developer**. Esto permite que las personas pertenecientes a este equipo pueden asignarse a aplicaciones, ejecutar compilaciones y ver registros de compilación. No obstante, no pueden cambiarse los roles de los miembros del equipo, agregar nuevos miembros, eliminar miembros o crear, editar o eliminar Workflows.

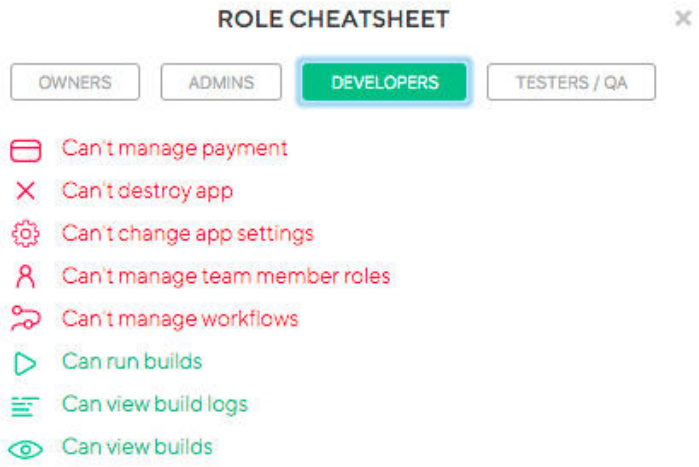

Imagen 16. *Permisos de rol DEVELOPR.*

Fuente: Elaboración propia, basada en la práctica.

La persona encargada de realizar el mantenimiento del proyecto en Bitrise (manejar los Workflows, establecer las configuraciones y velar porque todo funcione correctamente) tiene el rol de tipo **Owner**. Pero, también puede ser asignado un usuario **Admin**, además, cabe destacar que los owners tienen acceso ilimitado a las aplicaciones.

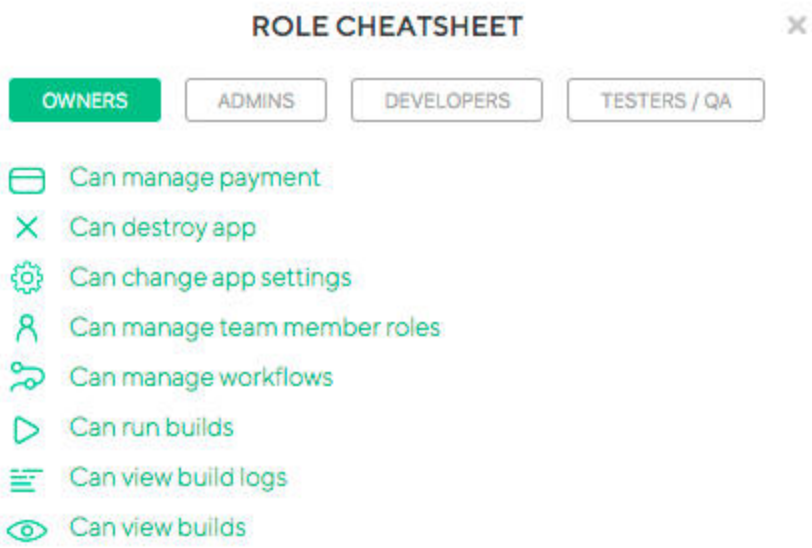

Imagen 17. *Permisos de rol OWNER.*

Fuente: Elaboración propia, basada en la práctica.

3.2.5 Editor de flujo de trabajos en el proyecto

Según los procesos de desarrollo y calidad establecidos con el esquema de *branching* de la aplicación, se decidieron crear distintos Workflows con el objetivo de construir la aplicación ante cualquier cambio que se quiera integrar. De este modo, poder distribuirlo en los distintos ambientes disponibles y también para lograr realizar una subida de la APP de forma automática al Google Play Store.

Para poder realizar esto, se crearon Workflows genéricos o privados, que tienen *steps* o tareas comunes y que pueden ser reutilizados en los Workflows que van a ser expuestos y manejados.

Workflows genéricos o privados

En el archivo *bitrise.yml* se declaran los Workflows privados con el signo "_" y luego el nombre descriptivo finalizando con los ":" para poder empezar a agregar los *steps* correspondientes que van a ser ejecutados secuencialmente.

Los Workflows genéricos declarados para la aplicación de Autogestión fueron los siguientes:

- **_install-java11:**

  Las máquinas virtuales provistas por Bitrise (configuradas en la opción "Stack"), donde se ejecutan los pipelines, contenían una versión vieja de Java, por lo que surgió la necesidad de instalar Java 11, como un primer paso de todos los Workflows, mediante el *step* provisto por Bitrise "Script" donde se escribió un *script* personalizado para cumplir con dicha necesidad.

```
_install-java11:
  steps:
  - script:
      inputs:
      - content: |
          #!/usr/bin/env bash
          # fail if any commands fails
          set -e
          # debug log
          set -x

          # write your script here
          sudo update-alternatives --set javac /usr/lib/jvm/java-11-openjdk-amd64/bin/javac
          sudo update-alternatives --set java /usr/lib/jvm/java-11-openjdk-amd64/bin/java

          export JAVA_HOME='/usr/lib/jvm/java-11-openjdk-amd64'
          envman add --key JAVA_HOME --value '/usr/lib/jvm/java-11-openjdk-amd64'
      title: Install Java 11
```

Imagen 18. *Workflow privado "_install-java11" detallado desde la vista bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

Este Workflow en un futuro será eliminado dado que la plataforma va a actualizar los stacks de máquinas virtuales ya con Java 11.

- **_clone:**

  Contiene los pasos comunes para realizar una autenticación al repositorio remoto alojado en bitbucket y clonarlo en la máquina virtual.

```yaml
_clone:
  steps:
  - authenticate-host-with-netrc:
      inputs:
      - username: "$GIT_USER"
      - password: "$GIT_USER_TOKEN"
      - host: "$BITBUCKET_URL"
  - git-clone:
      inputs:
      - clone_depth: '1'
```

Imagen 19. *Workflow privado "_clone" detallado desde la vista bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

El primer *step* es el de "Authenticate host with netrc", que es el encargado de realizar la autenticación con el repositorio remoto tomando como entradas el host (en este caso la URL de Bitbucket), el Username (un User con permisos de accesos al repositorio) y un Password o Token de autenticación para poder realizar esta acción.

El segundo *step* es el de "Git Clone Repository", que comprueba el estado del repositorio definido. Opcionalmente actualiza los submódulos del repositorio y exporta las propiedades de estado del repositorio de git logradas.

El *step* toma la URL del repositorio Git de la configuración inicial de "Settings" del proyecto y toma por defecto una ruta de Bitrise para clonarlo.

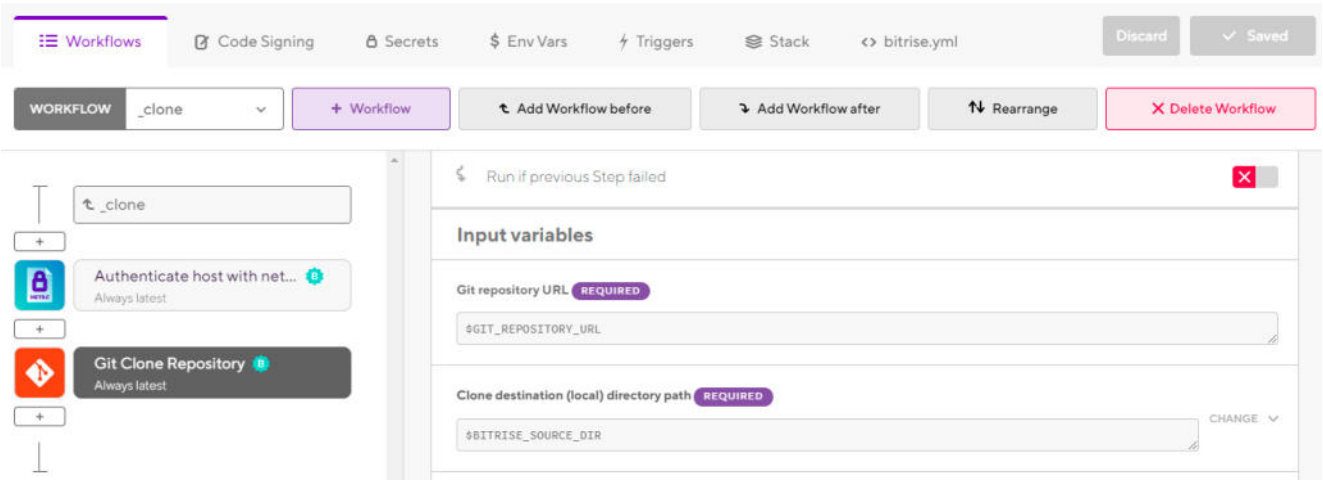

Imagen 20. *Step "Git Clone Repository" detallado desde el Workflow Editor UI.*

Fuente: Elaboración propia, basada en la práctica.

- **_write-global-variants:**

  La aplicación tiene, en su repositorio de código, un archivo llamado "gradle.properties" donde coloca un usuario y contraseña de Artifactory (repositorio centralizado que almacena distintos artefactos y dependencias), que le sirve para realizar la autenticación correspondiente y poder descargar ciertas dependencias necesarias.

  Al no ser una buena práctica colocar información sensible en el código fuente, lo que se hizo fue cargar el User, Pass y URL necesarios para la autenticación como variables secretas en Bitrise a través de la opción "Secrets".

  Mediante el *step* "Script", se creó un *script* personalizado que toma el valor de estas variables y las agrega en el archivo "gradle.properties" para que la aplicación pueda descargar las dependencias y construirse adecuadamente.

```
_write-global-variants:
  steps:
  - script:
      title: Escribir variables de configuracion para gradle
      inputs:
      - content: |-
          echo "
                MAVEN_USER=$MAVEN_USER
                MAVEN_PASS=$MAVEN_PASS
                MAVEN_REPO=$MAVEN_REPO
                " >> "main/dev/gradle.properties"
```

Imagen 21. *Workflow privado "_write-global-variants" detallado desde la vista de bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

- **_validate-password:**

  Es un Workflow que fue creado con la idea de permitir solo a los usuarios autorizados poder ejecutar las compilaciones y los despliegues de las APPS productivas.

  Tiene un único *step* que es el de "Script", el cual se personalizó para que cuando el usuario quiera ejecutar un Workflow que implica un despliegue, deba ingresar una

clave específica y un valor que coincida con una contraseña previamente definida y almacenada en los *secrets* de Bitrise.

Si este password coincide, entonces se imprime un "SUCCESS" en la consola de logs y el Workflow permite continuar con el flujo. En caso contrario, si el valor no coincide o el nombre de la variable no es la correcta, entonces se dispara un error en la ejecución, se imprime un error en la consola de logs y se corta el flujo.

- **_build:**

  Contiene todos los pasos necesarios para que la aplicación se construya en el ambiente correspondiente. Esto incluye: instalación de dependencias, ejecución de linters (comprobadores de código), test unitarios, construcción y publicación de estado de compilación.

  El primer *step* es el de "Bitbucket server post build status" que se encarga de publicar el estado de compilación en el repositorio bitbucket. El *step* toma como entradas el username y password de git para autenticar y la URL de Bitbucket (tomadas desde las Env vars). En este caso, se setea el estado como "INPROGRESS" y de esta manera, se publica el estado de la compilación con un ícono que representa el build en proceso.

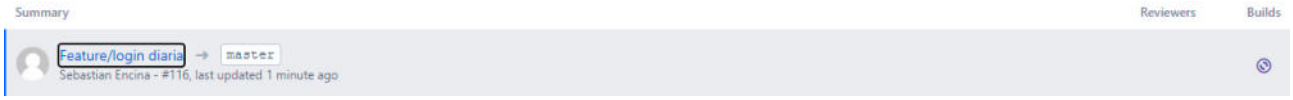

Imagen 22. *Ejemplo de acción del step "Bitbucket server post build status" con estado INPROGRESS sobre Bitbucket.*

Fuente: Elaboración propia, basada en la práctica.

El segundo *step* es el de "Install missing Android SDK components". Este paso se asegura que los componentes requeridos del SDK de Android (plataformas y herramientas de compilación) estén instalados. Para hacerlo, toma como input la ruta del archivo gradlew (que se encuentra en el proyecto de la aplicación) y ejecuta la task "gradlew dependencies", que instala todas las dependencias.

Imagen 23. *Step "Install missing Android SDK components".*

Fuente: Recuperado de https://www.bitrise.io/integrations/steps/install-missing-android-tools. (2021)

El tercer *step* es el de "Android Lint". Toma por defecto la ruta del proyecto y se configura colocando como entradas el nombre del módulo (que en el caso de esta aplicación es app) y el ambiente correspondiente.

Ejecuta la tarea de *lint*, la cual da como resultado un archivo HTML o XML de informe en el que se destacan las líneas de código donde se encuentran errores, explica los tipos de errores y sugiere correcciones. El *step* no hace que el build falle si detecta algún error estructural en el código.

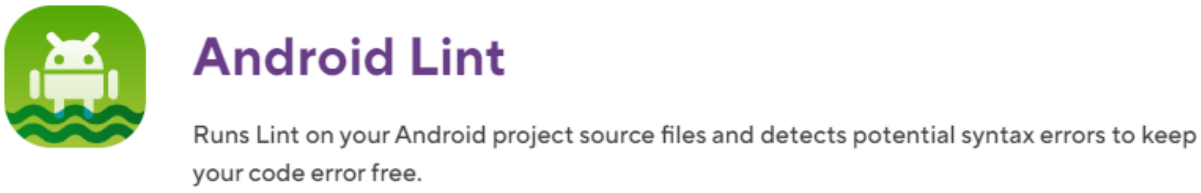

Imagen 24. *Step "Android Lint".*

Fuente: Recuperado de https://www.bitrise.io/integrations/steps/android-lint. (2021)

El cuarto *step* es "Gradle Runner". Este se encarga de ejecutar una tarea de Gradle específica, tomando como entradas: la ruta del archivo "build.gradle" del proyecto, la ruta del archivo gradlew y la tarea que se desea ejecutar.

El nombre del *step* fue personalizado como "Lint Kotlin", debido a que la tarea a ejecutar es la de "lintKotlin", que corresponde a la ejecución de un complemento de Gradle añadido en el proyecto de la aplicación. Esta tiene como objetivo *lintear* y

formatear archivos fuente de Kotlin. En caso de que se detecte algún error estructural en el código, el *step* hace que el build falle.

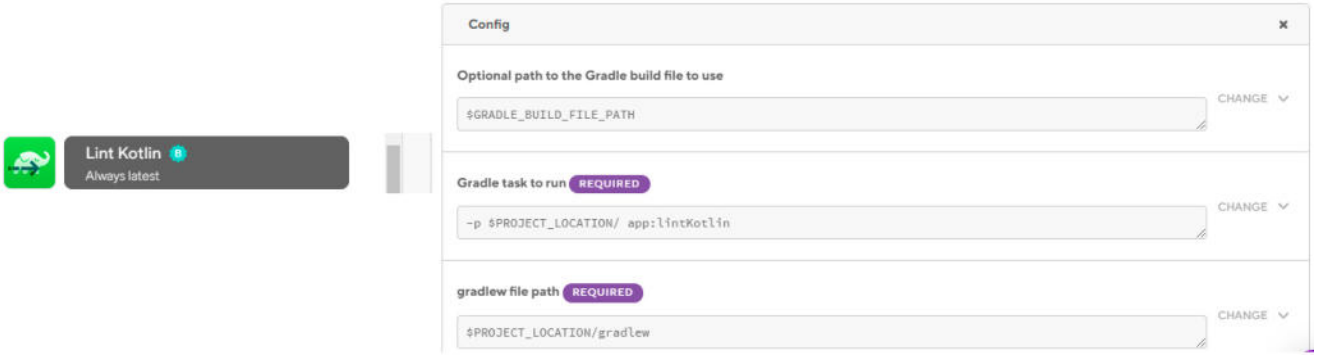

Imagen 25. *Step "Lint Kotlin".*

Fuente: Elaboración propia, basada en la práctica.

El quinto *step* es "Android Unit Test". Teniendo en cuenta que la aplicación debe tener implementado test unitarios, este *step* los ejecuta y da como resultado un reporte que muestra cómo salieron. Para poder realizar esta tarea, toma como entradas: la ruta del proyecto, el módulo y el ambiente correspondiente.
En caso de que alguno de los test unitarios falle, el build falla.

Imagen 26. *Step "Android Unit Test".*

Fuente: Recuperado de https://www.bitrise.io/integrations/steps/android-unit-test. (2021)

El sexto *step* es el de "Android Build". Se encarga de crear el proyecto de Android en Bitrise con comandos de Gradle: instala todas las dependencias que se enumeran en el archivo *build.gradle* del proyecto, además, crea y exporta un APK o AAB. Una vez que se exporta el archivo, queda disponible para otros *steps* en el flujo de trabajo.
Esto causa que, por defecto, tome la ruta del proyecto y se configure colocando como entradas el nombre del módulo y el ambiente correspondiente.

Imagen 27. *Step "Android Build".*

Fuente: Recuperado de https://www.bitrise.io/integrations/steps/android-build. (2021)

El sexto *step* es nuevamente el de "Bitbucket server post build status", pero en este caso se setea el estado de compilación como "AUTO", lo que indica que, si el build se construyó correctamente, se publica un estado "SUCCESSFUL" y si falla, un estado de "FAILED" en el repositorio de bitbucket.

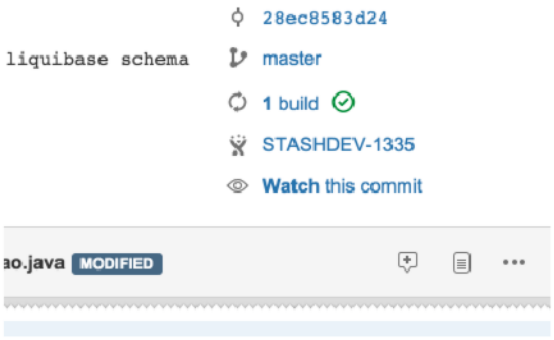

Imagen 28. *Ejemplo de acción del step "Bitbucket server post build status" con estado SUCCESFUL sobre Bitbucket.*

Fuente: Recuperado de https://developer.atlassian.com/server/bitbucket/how-tos/updating-build-status-for-commits/ (2019)

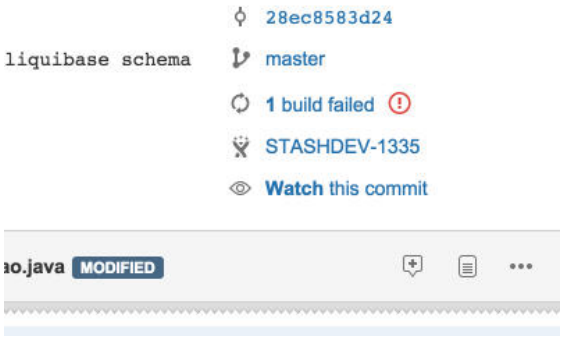

Imagen 29. *Ejemplo de acción del step "Bitbucket server post build status" con estado FAILED sobre Bitbucket.*

Fuente: Recuperado de https://developer.atlassian.com/server/bitbucket/how-tos/updating-build-status-for-commits/ (2019)

Workflows expuestos o públicos

- **build:**

  Es un Workflow que se encarga de construir la aplicación en el ambiente "debug" y distribuirla a los miembros del equipo "autogestión", que está compuesto por los devs y algunos testers.

  A través de la palabra clave "before_run", ejecuta secuencialmente una serie de Workflows genéricos y una vez que termina, ejecuta el *step* "deploy-to-bitrise-io".

```
build:
  before_run:
  - _clone
  - _write-global-variants
  - _build
  steps:
  - deploy-to-bitrise-io:
      inputs:
      - notify_user_groups: devs
```

Imagen 30. *Workflow público "build" detallado desde la vista de bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

El *step* “deploy-to-bitrise-io” accede a los artefactos de compilación construidos durante los *steps* anteriores y los dispone en la pestaña “APPS & ARTIFACTS” de cualquier compilación. Para los artefactos instalables como IPA o APK, el *step* puede crear una página de instalación publica que permite a los evaluadores instalar la aplicación en sus dispositivos. También, el *step* puede notificar a los usuarios sobre la compilación, como en este caso donde se notifica a los usuarios pertenecientes al equipo “autogestión”.

- **distribution_dev:**

  Es un Workflow que se encarga de construir la aplicación en el ambiente “debug” y distribuirla a distintos equipos a través del uso de **Firebase App Distribution**.

  Se ejecutan previamente los Workflows genéricos que permiten la construcción de la aplicación en el ambiente dicho y luego ejecuta los *steps*, que permiten hacer la distribución de la apk.

```
distribution_dev:
  before_run:
  - _clone
  - _write-global-variants
  - _build
  steps:
  - script:
      run_if: true
      inputs:
      - content: "#!/usr/bin/env bash\n# fail if any commands fails\nset -e\n# debug
          log\nset -x\n\n# write your script here\nset FIREBASE_TOKEN=$FIREBASE_TOKEN "
      title: Set variable firebase_token
  - gradle-runner:
      run_if: true
      inputs:
      - gradlew_path: "$PROJECT_LOCATION/gradlew"
      - gradle_task: "-p $PROJECT_LOCATION/ app:appDistributionUpload$VARIANT"
      title: Firebase distribution task
  - deploy-to-bitrise-io:
      inputs:
      - notify_user_groups: devs
  - cache-push: {}
  envs:
  - opts:
      is_expand: false
    VARIANT: debug
```

Imagen 31. *Workflow público "distribution_dev" detallado desde la vista de bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

El *step* "Script" es un *script* personalizado que se encarga de setear como variable global en la máquina virtual el token de firebase que fue almacenado en las Env Vars. Este token es necesario para poder autenticar con los proyectos de Firebase de la aplicación (un proyecto por cada ambiente) y así poder distribuir la apk.

El *step* "Gradle Runner", en este caso personalizado como "Firebase distribution task", ejecuta la tarea de Gradle "appDistributionUpload" que, a través del complemento de Firebase y con el ambiente definido, se autentica con los proyectos de Firebase y distribuye la apk a los grupos de testers definidos en el archivo *build.gradle*. Los miembros de estos grupos son asignados desde la consola de Firebase, en cada proyecto, a través de sus correos electrónicos.

- **distribution_qa:**

  Es un Workflow que se encarga de construir la aplicación en el ambiente “quality” y distribuirla a distintos equipos a través del uso de **Firebase App Distribution**.

```
distribution_qa:
  before_run:
  - _clone
  - _write-global-variants
  - _build
  steps:
  - script:
      inputs:
      - content: "#!/usr/bin/env bash\n# fail if any commands fails\nset -e\n# debug
          log\nset -x\n\n# write your script here\nset FIREBASE_TOKEN=$FIREBASE_TOKEN "
      title: Set variable firebase_token
  - gradle-runner:
      inputs:
      - gradlew_path: "$PROJECT_LOCATION/gradlew"
      - gradle_task: "-p $PROJECT_LOCATION/ app:appDistributionUpload$VARIANT"
      title: Firebase distribution task
  - deploy-to-bitrise-io:
      inputs:
      - notify_user_groups: devs
  - cache-push: {}
  envs:
  - opts:
      is_expand: false
    VARIANT: quality
```

Imagen 32. *Workflow público “distribution_qa” detallado desde la vista de bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

- **distribution_uat:**

  Es un Workflow que se encarga de construir la aplicación en el ambiente “uat” y distribuirla a distintos equipos a través del uso de **Firebase App Distribution**.

```
distribution_uat:
  before_run:
  - _clone
  - _write-global-variants
  - _build
  steps:
  - script:
      inputs:
      - content: "#!/usr/bin/env bash\n# fail if any commands fails\nset -e\n# debug
          log\nset -x\n\n# write your script here\nset FIREBASE_TOKEN=$FIREBASE_TOKEN "
      title: Set variable firebase_token
  - gradle-runner:
      inputs:
      - gradlew_path: "$PROJECT_LOCATION/gradlew"
      - gradle_task: "-p $PROJECT_LOCATION/ app:appDistributionUpload$VARIANT"
      title: Firebase distribution task
  - deploy-to-bitrise-io:
      inputs:
      - notify_user_groups: devs
  - cache-push: {}
  envs:
  - opts:
      is_expand: false
    VARIANT: uat
```

Imagen 33. *Workflow público "distribution_uat" detallado desde la vista de bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

- **deploy:**

  Es un Workflow que está armado para realizar el despliegue de una aplicación productiva. Se encarga de construir la aplicación apuntando al ambiente productivo, firmando y subiéndola al Google Play Store. Esta serie de pasos deriva en que, posteriormente, cualquier usuario pueda descargarla y utilizarla.

  Como parte de la ejecución del pipeline, previamente se ejecutan secuencialmente distintos Workflows genéricos.

```yaml
deploy:
  before_run:
  - _validate-password
  - _clone
  - _write-global-variants
  - _build
  steps:
  - sign-apk: {}
  - google-play-deploy:
      inputs:
      - service_account_json_key_path: "$BITRISEIO_BITRISEIO_SERVICE_ACCOUNT_JSON_KEY_URL_URL_URL"
      - package_name: **********
  - deploy-to-bitrise-io:
      inputs:
      - notify_user_groups: devs
  envs:
  - opts:
      is_expand: false
    VARIANT: release
```

Imagen 34. *Workflow público "deploy" detallado desde la vista de bitrise.yml.*

Fuente: Elaboración propia, basada en la práctica.

En primer lugar, se ejecuta el Workflow "_validate-password", que funciona como filtro de validación. Es decir, se realiza la validación de que quien esté ejecutando el Workflow sea un usuario autorizado y, a partir del resultado, determinar si se ejecutan los pasos siguientes o no.

La ejecución de los Workflows posteriores corresponden a todos los pasos necesarios para poder construir la aplicación, pero apuntando a los ambientes productivos, es decir, utilizando la "Env Var" $VARIANT con el valor de "release" en cada input necesario de los *steps* correspondientes a la construcción de la APP.

Una vez construida la app, se ejecuta el *step* "Android Sign" o también llamado "sign-apk". Este paso se encarga de tomar el archivo del almacén de claves (keystore) y las credenciales proporcionadas en la pestaña "Code Signinig" del editor de Workflows para firmar la APK digitalmente. Una vez ejecutado, se produce una APK o paquete de aplicación firmado que se utiliza como valor de entrada de campo para el *step* siguiente que es el de "Google Play Deploy".

Imagen 35. *Step "Android Sign".*
Fuente: Recuperado de https://www.bitrise.io/integrations/steps/sign-apk. (2021)

Finalmente, se ejecuta el *step* de "Google Play Deploy", el cual se encarga de subir la aplicación Android construida y firmada a Google Play Store.

Este paso utiliza la API de Google, por lo que previamente se debe configurar el acceso de la API de Google. Esto incluye:

- Vinculación de la Consola de desarrollador de Google a un proyecto de API.
- Configurar el acceso a la API mediante una cuenta de servicio.
- Otorgar los derechos de acceso necesarios a la cuenta de servicio.
- Cargar la clave JSON de la cuenta de servicio en Bitrise. Esto se realiza descargándola y adjuntándola en la solapa "Code Signing". Lo que permite mantenerla segura y accesible para el *step*.

La cuenta de servicio utilizada debe tener al menos los permisos de: "Editar la lista de PlayStore, los precios y la distribución", "Administrar APK de producción", "Administrar APK alfa y beta", "Administrar usuarios Alfa y Beta".

En lo que respecta a la configuración en sí del *step*, se debe señalar en un input la ruta del archivo de clave JSON de la cuenta de servicio. Además, se debe establecer la ruta del archivo de la aplicación (APK y/o ABB). Por otra parte, agregar el nombre del paquete de la aplicación y en el input Track, se debe colocar el track donde se quiere subir la aplicación (Alfa, Beta o un track custom).

Imagen 36. *Step "Google Play Deploy".*
Fuente: Recuperado de https://www.bitrise.io/integrations/steps/google-play-deploy. (2021)

Triggers

Con la idea de mantener calidad en el código y agilidad ante cada cambio en el repositorio, se estableció la ejecución de un *trigger* ante cada Pull Request, que se realiza desde cualquier Branch fuente a cualquier Branch destino.

Teniendo en cuenta el esquema de *branching* utilizado por los equipos (TBD), generalmente la ejecución automática del pipeline se dispara ante cada PR, que se realiza desde un Branch en desarrollo hacia el Branch Master; pero, se dan los casos en donde se ejecuta el mismo procedimiento hacia branches de fixing o de releases.

Desde Bitrise, se configuró para que el Workflow a ejecutar ante cada Pull Request sea el de **build**. Un Workflow que se encarga de detectar errores estructurales de código, ejecutar tests unitarios y construir la aplicación apuntando al ambiente dev, siempre y cuando cada paso haya sido ejecutado correctamente y no se haya disparado ninguna excepción.

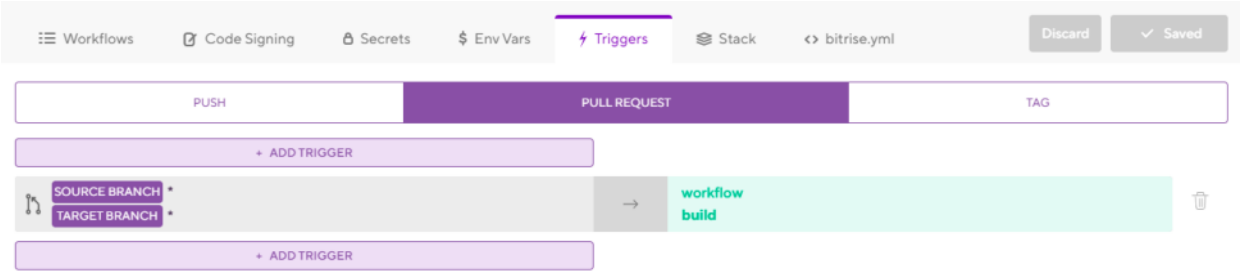

Imagen 37. *Configuración de Trigger en el proyecto.*
Fuente: Elaboración propia, basada en la práctica.

Esta estrategia de ejecutar el pipeline ante la finalización de cada feature o corrección de bugs, y previo a la fusión con master, permite que los devs encuentren problemas y no integren código erróneo o incorrecto.

3.3 Módulo Core

Para tomar la decisión sobre qué enfoque de modularización se aplicaría sobre el proyecto en el cual se está trabajando en esta PPS, se realizó un fuerte análisis sobre el desarrollo realizado hasta el momento en la APP y la forma de trabajo que tenían los equipos.

La aplicación tiene diferentes secciones, todas desarrolladas sobre la misma base de código, pero por distintos equipos o squads. Esto imposibilita la reutilización de ese código en otros productos. Como, por ejemplo, el manejo de eventos analíticos o el manejo de feature *flags* que pueden ser utilizadas como llaves de corte. Además, imposibilita hacer foco sobre las funcionalidades específicas del negocio a las cuales el equipo esté asignado. Todos estos puntos llevaron a tomar la decisión de migrar de una arquitectura monolítica hacia una modularizada por *features*.

Este enfoque basado en *features* permite sentar las bases para la creación de módulos funcionales y la creación de un módulo Core, el cual da como ventaja la reutilización de código, la mejora en tiempos de compilación y la posibilidad de utilizarlo en diversas aplicaciones de la organización.

El objetivo de la creación del módulo Core parte de una necesidad de reutilizar implementaciones y librerías de una manera homogénea y compartiendo los mismos lineamientos de buenas prácticas para diversas aplicaciones de la compañía. Teniendo en cuenta esta premisa, se tomó como base el código construido en la APP y se migraron distintas clases e implementaciones hacia un nuevo proyecto construido sobre un repositorio de código independiente. De esta forma, la construcción y despliegue del módulo Core es independiente de cualquier aplicación que la utilice o la quiera utilizar.

El análisis realizado sobre la APP en cuestión y para la construcción del módulo Core, dio como resultado la migración de todas las clases base que conllevaban la implementación de:

- Networking
- Eventos analíticos
- Logs
- Constantes de uso común
- Uso de librerías de Firebase
- Shared Preferences

La migración de clases sobre el proyecto Core implicó la refactorización de las mismas y también una refactorización y ordenamiento de la APP, que contenía estas implementaciones base. Debido a esto, se necesitó remodelarlas de una forma más abstracta para que cumpliera con las necesidades no solo de la aplicación en cuestión, sino también del resto que se sume a utilizar dicho módulo.

3.3.1 Capa de Networking

El módulo Core tiene una capa de abstracción denominada networking, la cual contiene las implementaciones necesarias para establecer las conexiones HTTP y utilizar servicios APIREST de manera segura. La implementación de esta capa, está basada en el uso de la librería Retrofit2, en conjunto con OkHttp para el uso de interceptores y seguridad en las solicitudes HTTP.

Retrofit2

Retrofit es un cliente HTTP de tipo seguro para Android y Java. Este permite conectar a un servicio web Rest (peticiones GET, POST, PUT, PATCH y DELETE), de manera sencilla, traduciendo la API a interfaces Java y obteniendo el resultado estructurado gracias al uso de conversores.

Además, hace utilización de OkHttp para manejar las peticiones de red y tiene soporte para librerías de conversión JSON para manejar las estructuras de los objetos, tanto de request como de response. Los conversores más usados son Gson, Jackson y Moshi.

Okhttp

OkHttp es un cliente HTTP creado por Square que tiene soporte HTTP/2 y permite que todas las solicitudes al mismo host compartan un socket. Por una parte, presenta una agrupación de conexiones que reduce la latencia de las solicitudes (si HTTP/2 no está disponible). Por otra parte, contiene un GZIP transparente, que reduce el tamaño de las descargas y contiene un almacenamiento en caché de respuesta que evita la red por completo para las solicitudes repetidas.

Su Api de request/response está diseñada con constructores fluidos e inmutabilidad. También, admite llamadas de bloqueo síncronas y llamadas asíncronas con devoluciones de llamada.

Implementación

Dado que el módulo Core está pensando para ser un módulo reutilizable, no solo por distintos módulos de una misma aplicación sino también para distintas APPS, se decidió realizar una implementación de Retrofit, que permita construir una instancia única desde la APP con los conversores que se prefieran y con una implementación personalizada de Okhttp. De esta manera, la aplicación principal hereda la clase “RetrofitFactory” de Core, sobreescribe el método que permite la inicialización de la instancia de Retrofit con los parámetros indicados, y finalmente, se construye para su uso en cualquier módulo siguiendo los lineamientos de un patrón de diseño Singleton.

CoreConfig

CoreConfig es una clase basada en el patrón Singleton, que sirve para realizar una inicialización del módulo Core estableciendo variables necesarias para la construcción del cliente Okhttp y la capa de Networking en sí.

Las variables que van a ser seteadas desde la APP a través de esta clase son las detalladas a continuación:

- *baseUrl:String* - Es la parte coherente de las solicitudes que se van a realizar al servidor, es decir, la sección de la URL que siempre se repite. Este valor es utilizado por Retrofit

para generar la instancia Singleton que será utilizada para manejar las solicitudes al servidor.

- *versionName:String* - Es el número de versión de la APP. Sirve para agregarlo como encabezado en las solicitudes que se realizan al servidor.
- *requestDataProvider: IRequestDataProvider* - Es la implementación de una interfaz que permite al módulo Core solicitar un Token al módulo de Login a través de la APP.

Estas variables son seteadas al iniciar la APP una única vez y son necesarias para que la capa de Networking funcione.

OkhttpFactory

Se presenta como una clase abstracta que tiene como fin la construcción de un cliente Okhttp utilizando parámetros y configuraciones base ya seteadas. Además, posibilita la modificación de algunos métodos con el fin de agregar nuevas configuraciones y construir un cliente con características personalizadas.

Como base, la clase OkhttpFactory construye un cliente seguro utilizando SSLPinning, que es el proceso de verificar que el certificado que ha enviado el servidor sea solo del servidor deseado y no de cualquiera válido. De esta manera, la APP envía un listado de certificados que considera seguros, con los cuales van a interactuar las peticiones y dichos certificados se agregan al cliente para validarse. Por lo tanto, cada petición que se hace valida que sea un certificado seguro comparando la lista recibida con el certificado del servidor al que apunta la petición. Si coinciden, la petición es segura y se realiza. En caso de no coincidir, quiere decir que alguien está haciendo una suplantación o una acción maliciosa, por lo que el cliente Okhttp no permite finalizar la petición.

Otro punto importante del cliente es la construcción de un interceptor. Ante cada petición que se realiza a través de Okhttp, se ejecuta un interceptor que se encarga de añadir encabezados ya predefinidos y necesarios para todas las solicitudes. Uno de estos encabezados es, por ejemplo, el token; considerado un *header* importante que permite autenticar las solicitudes y establecer un enlace seguro entre el cliente y el servidor, que es añadido a través de la interfaz *IRequestDataProvider*.

En caso de que la APP quiera realizar una implementación personalizada del cliente Okhttp, lo que se establece una implementación de una nueva clase efectuando una herencia de la clase OkhttpFactory y sobrescribiendo el método “makeOkhttp”. Este método es de tipo abstracto, pero establece la obligación de retornar un objeto de tipo OkhttpFactory, lo que permite a la APP utilizar los métodos protegidos que realizan la construcción base del cliente y agregarle nuevas configuraciones para retornar un nuevo objeto Okhttp basándose en el patrón Builder.

```
class OkHttpFactoryImpl : OkHttpFactory() {
    override fun makeOkHttpClient(context: Context, hostname: String, pinsList: List<String>):
            OkHttpClient =
        getOkHttpClientBuilder(context, hostname, pinsList).apply {  this: OkHttpClient.Builder
            addInterceptor(HttpLoggingInterceptor().apply {  this: HttpLoggingInterceptor
                level = HttpLoggingInterceptor.Level.BODY
            })
        }.addNetworkInterceptor(StethoInterceptor())
            .build()
}
```

Imagen 38. *Ejemplo de implementación personalizada de OkhttpFactory para la construcción de objeto Okhttp.*

Fuente: Elaboración propia, basada en la práctica.

RetrofitFactory

Una vez que el cliente Okhttp fue construido, entra en juego la clase RetrofitFactory. Una clase que se encarga de construir una instancia de Retrofit utilizando el cliente Okhttp ya generado y agregando las configuraciones necesarias para realizar las solicitudes, como lo son el *baseUrl* previamente configurado.

Esta clase se instancia por única vez y luego es utilizada en todo momento por la APP para realizar cualquier solicitud al servidor manejando las distintas clases “services” (interfaces que indican como van a ser las distintas solicitudes HTTP).

RetrofitFactory consta de dos variables fundamentales que deben ser inicializadas:

- *cliente:Okhttp* - Es el cliente Okhttp construido previamente a través de la clase OkhttpFactory.

- *converteferFactory:Converter.Factory* - Al ser una aplicación RestFul, es necesario tener un converter Factory ya que es el encargado de convertir los objetos JSON en objetos modelo java (POJO) o dataclass en Kotlin, de tal forma que puedan ser consumidos y utilizados por el proyecto. La conversión o serialización se hace en ambos sentidos, de JSON a objeto y de objeto a JSON.

Estas variables son agregadas por la APP sobrescribiendo un método de la clase RetrofitFactory utilizando la herencia.

```
class RetrofitFactoryImpl: RetrofitFactory() {
    override fun initializateRetrofit(converterFactory: Converter.Factory, client: OkHttpClient) {
        setServiceRetrofit(converterFactory,client)
    }
}
```

Imagen 39. *Inicialización de instancia Retrofit desde la APP.*

Fuente: Elaboración propia, basada en la práctica.

Una vez seteadas las variables, la clase RetrofitFactory se encarga de construir la instancia cuando es solicitada y disponerla para futuros usos.

```
private val instance: Retrofit by lazy {
    Retrofit.Builder()
        .baseUrl(CoreConfig.baseUrl)
        .addConverterFactory(converterFactory)
        .client(client)
        .build()
}

fun <T> getServiceRetrofit(service: Class<T>): T = instance.create(service)
```

Imagen 40. *Instancia de Retrofit construida con los parámetros y variables seteadas.*

Fuente: Elaboración propia, basada en la práctica.

La inicialización de todas las variables y clases se realizan desde la clase “Application” en la APP, clase que permanece instanciada a lo largo del ciclo de vida de Android. De esta manera, queda disponer la instancia de Retrofit para su uso posterior.

```
private fun initModules(context: Context) {
    CoreConfig.baseUrl = BuildConfig.BASE_URL
    CoreConfig.versionName = BuildConfig.VERSION_NAME
    CoreConfig.requestDataProvider = RequestDataProviderImpl()
    okHttpClient = okHttpFactory.makeOkHttpClient(context, Constants.hostName, Constants.pinList)
    retrofitFactory.initializateRetrofit(converter, okHttpClient)
}
```

Imagen 41. *Inicialización de variables y clases necesarias para capa de Networking.*

Fuente: Elaboración propia, basada en la práctica.

3.3.2 Analytics

Analytics es una capa construida en el módulo Core, que, agrupa un conjunto de clases e implementaciones que permiten realizar el tracking de la aplicación a través de una interfaz con la que se implementa una librería analítica. Desde el módulo en cuestión, se provee una implementación de esta interfaz basada en la SDK de Firebase Analytics (la cual se recomienda utilizar), pero es decisión de los desarrolladores utilizarla o realizar una implementación personalizada con una librería distinta.

Firebase Analytics

Es una solución gratuita de medición de apps, que proporciona estadísticas sobre el uso de las apps y la participación de usuarios. Analytics se integra a distintas funciones de Firebase y proporciona una capacidad ilimitada de generar informes sobre un total de hasta 500 eventos distintos, que se pueden definir con el SDK de Firebase. Los informes de Analytics permiten entender claramente cómo se comportan los usuarios para que se puedan tomar decisiones fundamentadas en relación con el marketing de las APPS y las optimizaciones del rendimiento.

Clases e implementación de capa Analytics

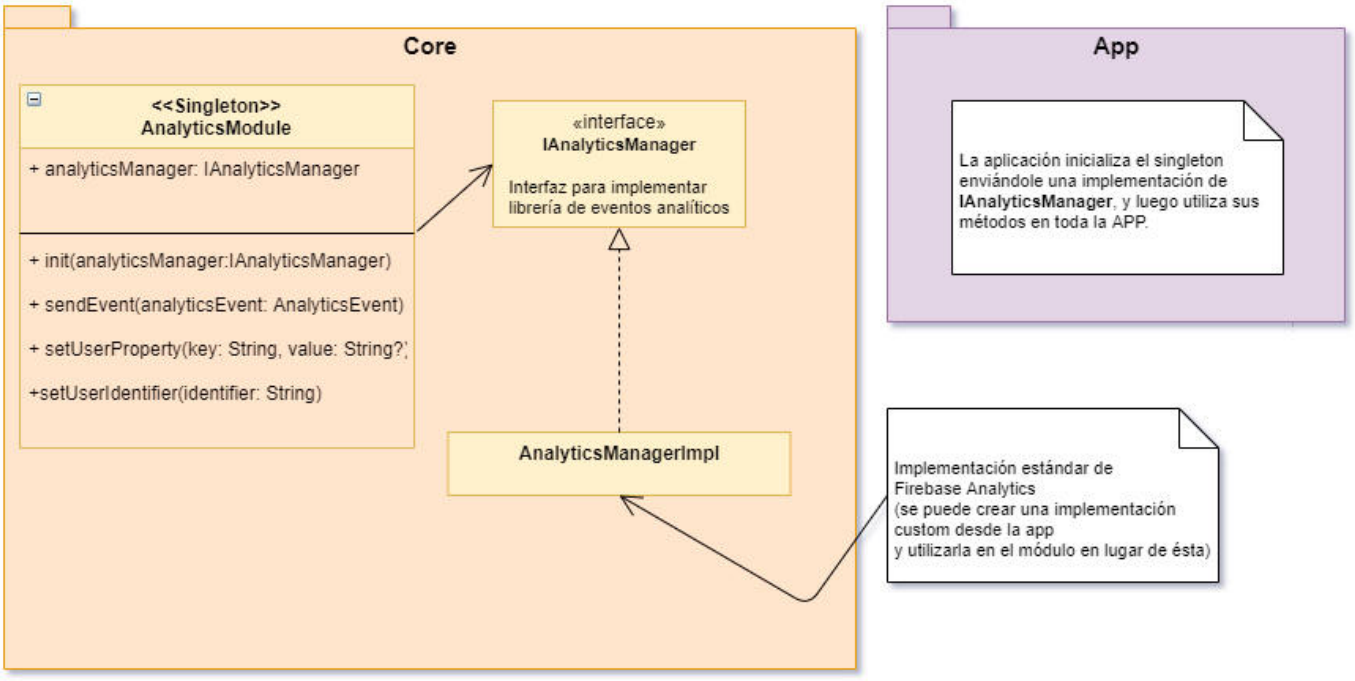

Imagen 42. *Diagrama de clases de Analytics en Core.*

Fuente: Elaboración propia, basada en la práctica.

La clase AnalyticsModule es un Singleton que necesita ser inicializado por la APP antes de utilizar sus métodos. Esto se realiza mediante el método init(), que recibe como parámetro una implementación de la interfaz IAnalyticsManager, la cual puede ser la ya construida desde el módulo Core "AnalyticsManagerImpl" o una construida desde la aplicación de manera personalizada.

Un ejemplo de inicialización sería la siguiente línea de código agregada desde la APP:

```
AnalyticsModule.init(AnalyticsManagerImpl(applicationContext))
```

Donde la clase AnalyticsManagerImpl es la implementación de la interfaz para la librería analítica provista por Core que utiliza la SDK de Firebase Analytics.

Una vez inicializado el Singleton, se puede tener acceso a sus distintos métodos que funcionan como un wrapper de la implementación de IAnalyticsManager utilizada:

- *setUserIdentifier (identifier: String)* - Envía el ID de usuario de la persona que utiliza la APP.
- *setUserProperty(@Size(min = 1L, max = 24L) key: String, @Size(max = 36L) value: String?)* - Envía un atributo que describe al usuario de la aplicación.
- *sendEvent(evento: AnalyticsEvent)* - Envía un evento de tipo AnalyticsEvent (objeto en el que se coloca el nombre del evento y un bundle con todos los parámetros necesarios para trackear).

Envío de eventos desde la APP

La lógica de envío de eventos se realiza a través de la clase AnalyticsModule utilizando una clase intermedia (Interactor) desde los ViewModels de Android.

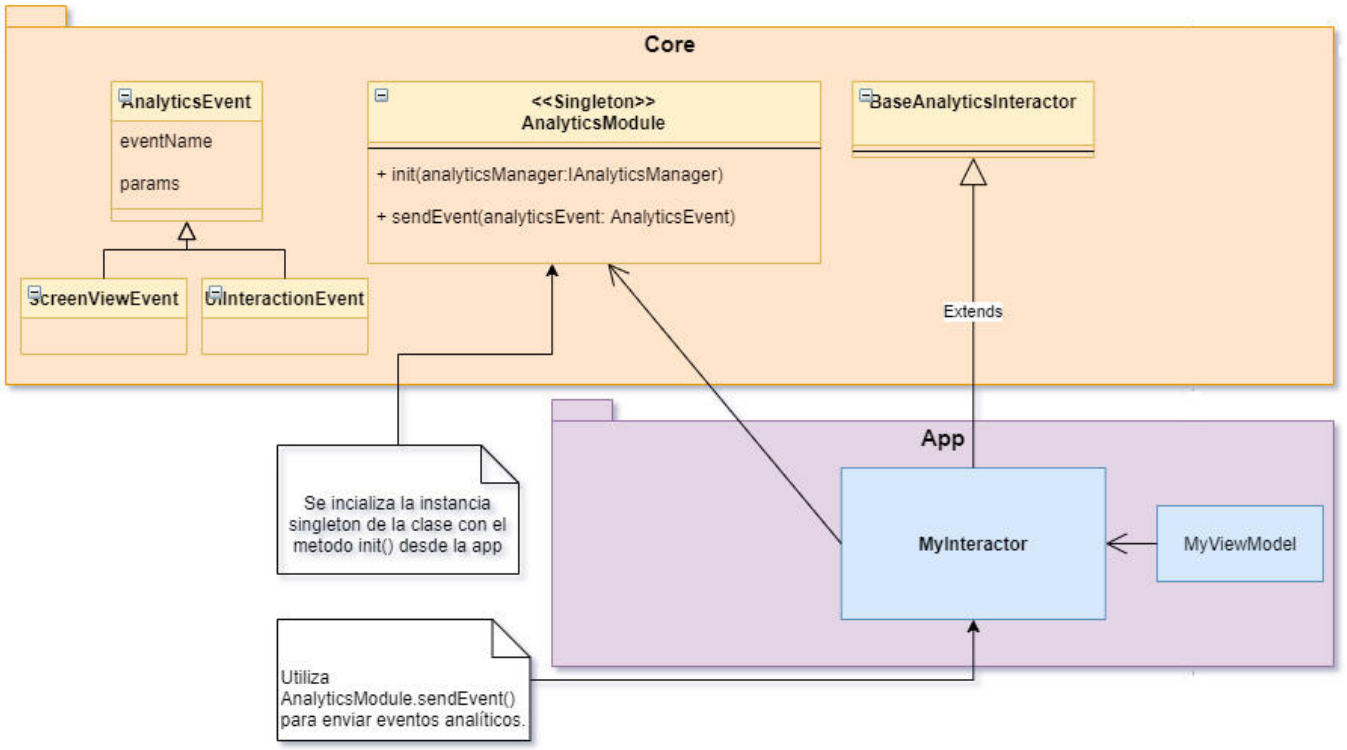

Imagen 43. *Diagrama de envío de eventos utilizando la APP y las clases de Core.*

Fuente: Elaboración propia, basada en la práctica.

La aplicación en cuestión crea una clase que extiende de BaseAnalyticsInteractor() (una clase Interactor definida desde Core). En ella, se crean los métodos que van a utilizar al Singleton AnalyticsModule y que van a encargarse de enviar los eventos, los cuales son objetos de tipo AnalyticsEvent definidos en Core o personalizados por parte de la APP.

Esta definición de crear distintas clases Interactor, que sean usadas por los ViewModels, surgen con el fin de agrupar los eventos por funcionalidades de la APP.

En el ViewModel se recibe por constructor una instancia del Interactor creado y, de esta manera, se crea un método que a su vez termina llamando al método encargado de enviar los eventos (esto a través de la instancia del Interactor recibida).

Finalmente, cualquier Activity o Fragment que consuma el ViewModel en cuestión, puede utilizar los métodos creados para poder realizar el envío de eventos.

3.3.3 Logger

Logger es una capa construida desde el módulo Core, que permite integrar diferentes modos de trazabilidad y de registración de logs para la aplicación, que pueden ser tanto locales como remotos (Crashlytics). El módulo Core provee distintas interfaces para las implementaciones, que son utilizadas en la clase LoggerModule, un Singleton que dispone las variables de instancia y los métodos de logs correspondientes.

Crashlytics

Firebase Crashlytics es una herramienta liviana para provisión de informes de fallas en tiempo real, que permite hacer un seguimiento de los problemas de estabilidad que afectan la calidad de la APP, de tal manera que se puedan priorizar y corregir. Crashlytics agrupa las fallas de forma inteligente y destaca las circunstancias en las que se produjeron, lo que permite ahorrar tiempo en la solución de problemas.

Stetho

Stetho es un bridge de depuración sofisticado para aplicaciones Android. Cuando está habilitado, los desarrolladores tienen acceso a la función de herramientas de desarrollo de Chrome, que forma parte de manera nativa del navegador de escritorio Chrome. En combinación con la popular librería OkHttp, puede utilizar los interceptores para para tener una vista precisa del tráfico de red y ver los logs del mismo.

Firebase Performance

Firebase Performance es un servicio que permite obtener información valiosa sobre las características de rendimiento de las APPS.

El SDK de Firebase Performance Monitoring permite recopilar datos de rendimiento de la APP y, luego, revisa y analiza esos datos en Firebase console. Además, ayuda a comprender en tiempo real donde se puede mejorar el rendimiento de la aplicación, de manera que se pueda utilizar esa información para solucionar problemas.

Interfaces provistas por Core:

- *IRequestLogger*: Es una interfaz que permite de manera genérica establecer una implementación para registrar los logs de los requests de la aplicación y las transacciones. La librería recomendada para la implementación de dicha interfaz es Sthetho, que permite conectar la APP a un navegador Chrome en una computadora y analizar los requests de la aplicación.
  La implementación de RequestLogger queda a criterio de la aplicación, pero siempre se debe tener en cuenta que se tiene que habilitar únicamente para ambientes bajos y no productivos.
- *ITraceLogger*: Es una interfaz que permite a la aplicación realizar traces (rastros) que ayudan a medir los tiempos de los flujos de la APP.
  La librería recomendada para implementación es Firebase Performance, el cual recopila datos para varios procesos comunes de la aplicación estableciendo informes de rendimiento para luego revisar y analizar desde Firebase console.
- *ILoggerType*: Provee los métodos que permiten realizar los distintos tipos de logs. En caso de ser logs locales lo recomendable es implementar una clase que se ejecuta en

los ambientes bajos y, si son productivos, lo recomendado es utilizar una librería puntual como Crashlytics.

Estas implementaciones dependen de la necesidad de cada APP, aunque desde Core se proveen dos implementaciones puntuales para estos casos: *LocalLogger* y *RemoteLogger*.

- *ILoggerTypeFactory*: Otorga la estructura para poder utilizar el patrón Factory. La implementación de esta interfaz queda a cargo completamente de la aplicación.

Clases e implementación de capa Logger

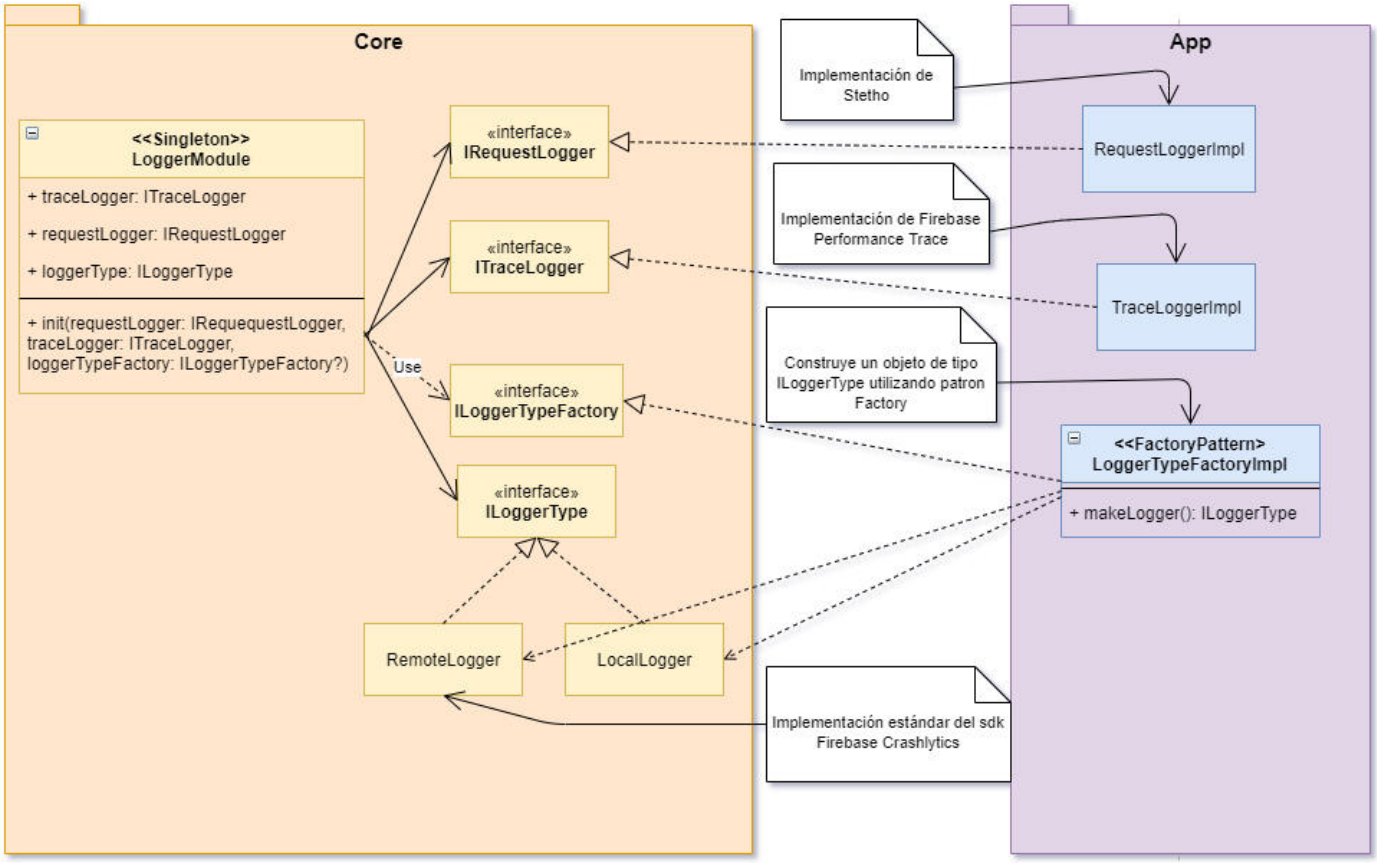

Imagen 44. *Diagrama de clases de Logger en Core.*

Fuente: Elaboración propia, basada en la práctica.

La clase LoggerModule es un Singleton, que es inicializado por la APP antes de utilizar sus métodos. Esta inicialización se realiza mediante el método init().

Un ejemplo de inicialización sería la siguiente línea de código agregada desde la APP:

```
LoggerModule.init(requestLogger: IRequestLogger, traceLogger: ItraceLogger,
loggerTypeFactory: ILoggerTypeFactory?)
```

Donde requestLogger es la implementación que permite realizar logs de los requests a nivel capa de Red de la aplicación; por otra parte, traceLogger es la implementación que utiliza FirebasePerformance y, por último, loggerTypeFactory remite a la implementación basada en el patrón Factory que retorna un tipo de ILoggerType. Este último, puede ser una implementación basada en los logs locales (LocalLogger) o los logs remotos (RemoteLogger, donde se utiliza la librería Crashlytics).

En el caso de loggerTypeFactory, se ejecuta el método *makeLogger(),* que termina seteando un LocalLogger o RemoteLogger y, de esta forma, se proporcionan los métodos de uno u otro para su uso. Si no se recibe ningún Factory, desde el Singleton se provee por defecto el LocalLogger.

Una vez inicializada la clase LoggerModule, la aplicación puede acceder a sus distintos métodos, que funcionan como un wrapper de la implementación de ILoggerType utilizada:

- logUserId(identifier: String)
- error (tag: String?, msg: String, code: Int = -1, e: Exception? = null) value: String?)
- info (tag: String?, msg: String?)
- logParams(params: Bundle)

Los parámetros enviados en lo métodos nombrados anteriormente, varían según la implementación de ILoggerType utilizada: LocalLogger o RemoteLogger.

Cuando la implementación utilizada es ***LocalLogger*** se tiene que:

- tag = definir dónde se imprime el log.
- Msg = mensaje para mostrar en el log.

Cuando la implementación utilizada es ***RemoteLogger***, se tiene que:

- tag = URL del servicio web.
- msg = método HTTP (GET, POST, PUT, DELET, PATCH)
- code = código HTTP (500, 200, 209, 404, etc.)
- e = excepción para loguear en Crashlytics.

En los casos de *requestLogger* y *traceLogger*, cuando la clase LoggerModule fue iniciada, se acceden directamente como variables de instancia, esto quiere decir, que quedan disponibles y de acceso público para que puedan utilizar los métodos que contienen cada uno.

3.3.4 Feature Flags

Firebase Remote Config es un servicio de nube que permite cambiar el comportamiento y el aspecto de la APP sin que los usuarios tengan que descargar una actualización. Cuando se usa Remote Config, se crean valores predeterminados en la app que controlan el comportamiento y el aspecto de la aplicación. Luego, desde Firebase Console o las API de backend de Remote Config se pueden anular los valores predeterminado es en la APP para todos los usuarios o segmentos de la base de usuarios.

La aplicación controla y verifica si hay disponibles actualizaciones y, de esta manera, aplicarlas, todo con un impacto insignificante en el rendimiento.

Implementación

Desde el lado de Core, se creó la clase "FirebaseRemoteConfingImpl" que contiene toda la implementación de la SDK de Firebase Remote Config y que debe ser inicializada antes de utilizar los métodos que expone, los cuales se encargan de actualizar los *flags* o comprobar individualmente cada uno de ellos por su tipo.

El método init de la clase recibe como parámetro la ruta de un archivo xml, el cual va a contener todos los *flags* que fueron definidos desde la consola de Firebase, en conjunto con cada uno de los valores que se especificaron por defecto. Además, recibe una interfaz llamada

“RemoteConfigCallback”, que puede ser opcional, y se encarga de corroborar si la solicitud de actualización de *flags* fue correcta o errónea. Esto último se realiza, ya que la actualización se ejecuta de forma asíncrona, y la aplicación tiene que saber si falla o no para poder tomar una acción.

Con estos parámetros recibidos, la clase se encarga de, a través la SDK, setear los valores definidos en el xml que se recibió y ejecutar un método que se encarga de consultar con los datos remotos (que fueron seteados en la consola de Firebase). De esta manera, utilizando la interfaz recibida, se le informa a la aplicación si dicha consulta tuvo éxito o no.

Una vez realizado esto, la APP puede utilizar un método expuesto llamado “fetch”, que sirve para corroborar si algunos de los datos fueron actualizados y “traer” esos cambios en cualquier momento.

El resto de los métodos tiene como objetivo enviar el nombre de un *flag* determinado y realizar una consulta a la base remota para que se devuelva el valor definido. De esta manera, la APP compara ese resultado con el valor que tiene por defecto en el xml que se utilizó en el paso de la inicialización y, a partir de esta comparación, determina si hubo un cambio en el valor del *flag* determinado y realiza una acción.

```
override fun isAvailable(key: String): Boolean {

    return Firebase.remoteConfig.getBoolean(key)
}

override fun isAvailableString(key: String): String {

    return Firebase.remoteConfig.getString(key)
}

override fun isAvailableJson(key: String): JSONObject {

    return JSONObject(Firebase.remoteConfig.getString(key))
}
```

Imagen 45. *Métodos que reciben un flag determinado y retornan el valor asignado en la base remota de Firebase Remote Config.*

Fuente: Elaboración propia, basada en la práctica.

3.4 Generación y publicación de módulos

La modularización tiene como uno de sus objetivos principales, la construcción de módulos funcionales que puedan ser separados de la APP principal y que puedan interconectarse y comunicarse de manera efectiva, sin perder su independencia.

Cada módulo debe crearse de forma individual sobre un repositorio independiente al del resto. La idea es que cada módulo sea considerado como un proyecto individual. Es decir, a nivel estructura de Android, cada proyecto va a tener una APP de pruebas y una librería, que va a ser el módulo funcional en cuestión y será la expuesta en Artifactory para que pueda ser consumida como dependencia. Teniendo en cuenta esto, lo primero que se realiza es crear un proyecto Android desde cero utilizando Android Studio.

FASE 1- CREACIÓN DE LIBRERÍA ANDROID.

Una vez que se creó el proyecto, se siguieron los siguientes pasos desde Android Studio:

1- Se realiza clic en ***File > New > New Module.***

2- En la ventana ***Create New Module*** que aparece, se hace clic en ***Android Library*** y, luego, en ***Next***.

3- Se otorga un nombre a la biblioteca, se revisa el nombre del package (en caso de que no sea el correspondiente se lo cambia), se selecciona una versión mínima del SDK y la versión de Java que se va a utilizar (lo ideal es tener las mismas configuraciones que tiene la APP que va a consumir esta librería). Luego se hace clic en ***Finish***.

Finalizado estos pasos, se realiza un Sync del proyecto de Gradle y se mostrará el módulo de la biblioteca en el panel Project de la izquierda.

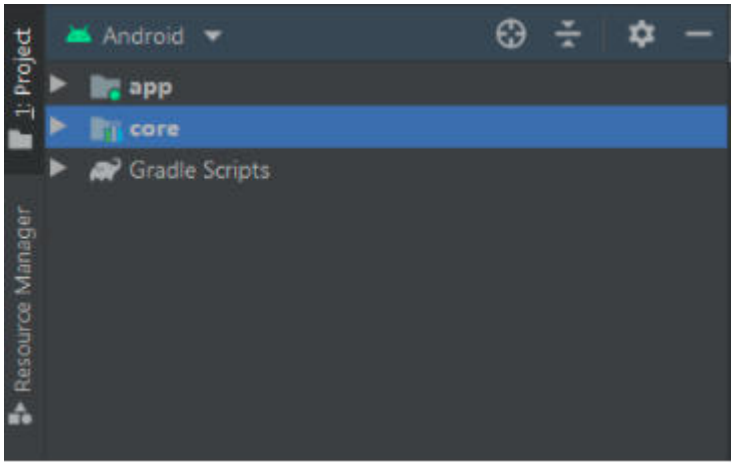

Imagen 46. *Estructura de un proyecto creado con su APP de prueba y el módulo en cuestión.*

Fuente: Elaboración propia, basada en la práctica.

La estructura del proyecto tiene a la aplicación de prueba como “app” y la librería que va a ser consumida tendrá el nombre que se le indicó, en el caso de la figura x se lo configuró con el nombre de “core”.

FASE 2 - CONFIGURACIÓN DE LIBRERÍAS PARA COMPILACIÓN Y PUBLICACIÓN EN ARTIFACTORY

1- Configuración de variable para versión de librería.

Gradle permite crear propiedades de extensión, las cuales son expuestas para lectura y escritura. Esto permite acceder externamente y utilizarlas en los pasos de generación de la librería (para construir el nombre de la misma, por ejemplo), leerlas para los pasos de la integración continua y también para la publicación en Artifactory.

En el *build.gradle* perteneciente al módulo se crea una extensión de propiedad que será la que enmarque si la versión creada es abierta (SNAPSHOT) o cerrada (release). Esto sirve para, posteriormente, integrarse al esquema de publicación de módulos.

Esta property es creada como una clave-valor tomando el nombre de 'versionNumber' y con un valor que va a ser la versión de forma semántica 'x.y.z' (correspondiente al esquema de *branching* utilizado) si es la versión cerrada, o con el agregado de la palabra al final '-SNAPSHOT'. La property quedaría de la siguiente forma:

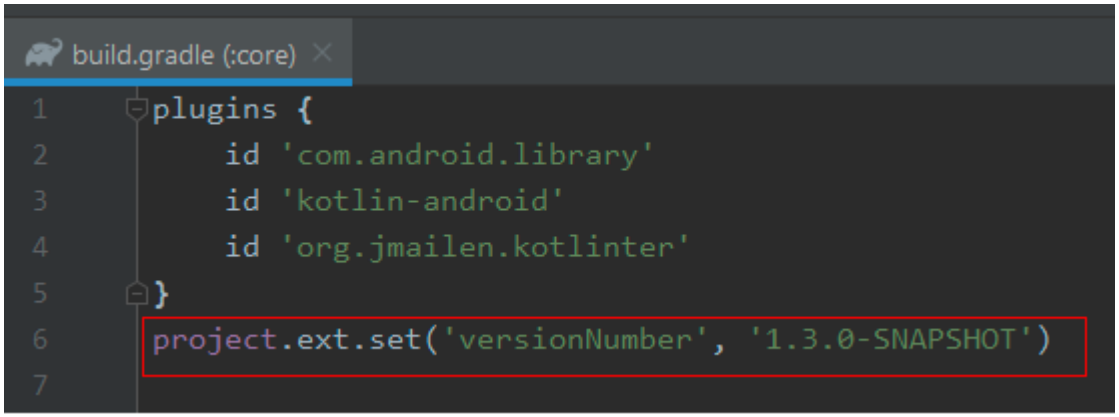

Imagen 47. *Ejemplo de asignación de versión en el build.gradle del módulo..*

Fuente: Elaboración propia, basada en la práctica.

Esta propiedad es utilizada en el defaultConfig del *build.gradle* del módulo para señalar el versionName.

```
defaultConfig {
        minSdkVersion 21
        targetSdkVersion 30
        versionCode 1
        versionName project.ext.versionNumber
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
        consumerProguardFiles "consumer-rules.pro"
    }
```

Imagen 48. *Utilización de versionName en las configuraciones del build.gradle del módulo.*

Fuente: Elaboración propia, basada en la práctica.

2 - Creación de archivo publish.gradle

Para poder realizar la publicación del módulo compilado en Artifactory se necesita crear un archivo, que tenga todas las configuraciones de publicación. Es decir, que se encargue de subir el aar del módulo (la compilación) al repositorio indicado y que, además, sume un archivo pom.xml en donde se indiquen todas las dependencias que se consuman.

Un modelo de objeto de proyecto o POM es la unidad fundamental de trabajo en Maven. Es un archivo XML que contiene información sobre el proyecto y los detalles de configuración utilizados por Maven para construir el proyecto.

Los parámetros más importantes que deben ser usados en el archivo son:

- VARIANT_PUBLISH: indicará si la versión a publicar es 'release' o 'debug.
- groupId: nombre del grupo de Artifactory que va a contener el módulo.
- versión: contenerá la versión a publicar del módulo. Generalmente se utiliza la extensión de propiedad declarada previamente ("project.ext.versionNumber")
- artifact: contiene la ruta absoluta de donde se encuentra el aar (módulo ya compilado).

Otro punto importante en este archivo es declarar el repositorio maven donde se va a publicar. Esto incluye la colocación de credenciales y la URL del repositorio de artifactory.

El archivo *publish.gradle* generado queda de la siguiente manera:

Primera parte: Comienzo de script y parámetros de repositorio y rutas.

```
apply plugin: 'maven-publish'

publishing {
    publications    { PublicationContainer it ->
        teco(MavenPublication) {
            /* Release version has to be implicitly compiled with variant release
             SNAPSHOT version has to be implicitly compiled with variant debug */
            def VARIANT_PUBLISH = 'release'
            if("${project.ext.versionNumber}".contains('SNAPSHOT')){
                VARIANT_PUBLISH = 'debug'
            }
            groupId ${groupId}
            artifactId ${module}
            version project.ext.versionNumber
            artifact("./build/outputs/aar/core-${VARIANT_PUBLISH}-${project.ext.versionNumber}.aar")
```

Imagen 49. *Primera parte del script de publicación.*

Fuente: Elaboración propia, basada en la práctica.

Segunda parte: Construcción de archivo pom.xml con dependencias.

```
        pom.withXml {
            final dependenciesNode = asNode().appendNode('dependencies')
            ext.addDependency = { Dependency dep, String scope ->
                if (dep.group == null || dep.version == null || dep.name == null || dep.name == "unspecified")
                    return // ignore invalid dependencies
                final dependencyNode = dependenciesNode.appendNode('dependency')
                dependencyNode.appendNode('groupId', dep.group)
                dependencyNode.appendNode('artifactId', dep.name)
                dependencyNode.appendNode('version', dep.version)
                dependencyNode.appendNode('scope', scope)
                if (!dep.transitive) {
                    final exclusionNode = dependencyNode.appendNode('exclusions').appendNode('exclusion')
                    exclusionNode.appendNode('groupId', '*')
                    exclusionNode.appendNode('artifactId', '*')
                } else if (!dep.properties.excludeRules.empty) {
                    // Otherwise add specified exclude rules
                    final exclusionsNode = dependencyNode.appendNode('exclusions')
                    dep.properties.excludeRules.each { ExcludeRule rule ->
                        final exclusionNode = exclusionsNode.appendNode('exclusion')
                        exclusionNode.appendNode('groupId', rule.group ?: '*')
                        exclusionNode.appendNode('artifactId', rule.module ?: '*')
                    }
                }
            }
            configurations.api.getDependencies().each { Dependency dep -> addDependency(dep, "compile") }
            // List all "implementation" dependencies (for new Gradle) as "runtime" dependencies
            configurations.implementation.getDependencies().each { Dependency dep -> addDependency(dep, "runtime") }
        }
```

Imagen 50. *Segunda parte del script de publicación.*

Fuente: Elaboración propia, basada en la práctica.

Tercera parte: declaración de repositorio y credenciales para autenticar con Artifactory6.

```
repositories { RepositoryHandler it ->
    maven {
        credentials { PasswordCredentials it ->
            username = MAVEN_USER
            password = MAVEN_PASS
        }
        url MAVEN_REPO
    }
}
```

Imagen 51. *Declaración de repositorio y credenciales para autenticar con Artifactory6*

Fuente: Elaboración propia, basada en la práctica.

Con el archivo creado, se tiene que importar en el *build.gradle* del módulo en la última línea de código. Por ello, Gradle tiene acceso a las instrucciones y puede ejecutarlas para realizar una publicación.

Siguiendo esta línea, cuando se tiene un módulo completamente desarrollado, la CI en uno de sus Workflows, que fueron creados específicamente para la construcción y publicación de módulos, ejecuta en uno de sus *steps* una tarea de Gradle llamada "gradlew publish". Dicha tarea lo que hace es ejecutar el *script* enmarcado en el archivo publish.gradle, que se observó anteriormente, encargándose de autenticar con Artifactory6 y publicar el módulo como un archivo aar y su respectivo pom.xml para que pueda ser descargado y utilizado como dependencia por cualquiera APP.

3.5 Esquema para publicación de módulos

Desde la plataforma Bitrise se definieron dos Workflows base que se encargan de construir un módulo y publicarlo en Artifactory. Estos Workflows son utilizados en cada proyecto que compile un módulo dentro de Bitrise, ya sea funcional o Core, con el fin de automatizar el proceso de construcción y publicación de los mismos en Artifactory.

El primero se encarga de construir el módulo en un artefacto aar (formato de las librerías y módulos Android) y publicarlo como una versión snapshot en el repositorio que se le haya indicado en Artifactory. El segundo cumple la misma función, pero publica una versión cerrada.

Estos procesos se ejecutan de manera automática ante un evento particular, que fue considerado desde Bitrise. En el caso del Workflow "Build_and_snapshot" se ejecuta de forma automática ante cada "Pull Request" que se haga desde un Branch a otro. En el caso de

“build_and_release”, se ejecuta de forma automática cuando se realiza un TAG de una versión indicada desde GIT.

Para que estos procesos automáticos se ejecuten de forma adecuada, se propuso un esquema de publicación de módulos. Estos son lineamientos que tienen que seguir todos los devs encargados de la construcción de las librerías, para poder realizar una publicación de las mismas en Artifactory de una manera adecuada.

Los lineamientos de publicación y pasos a seguir son los siguientes:

1 – Nuevo feature

Cuando se inicia un nuevo desarrollo, que tiene como objetivo sumar una nueva funcionalidad al módulo en cuestión, entonces desde el Branch Master se crea un Branch *feature* que será el que contenga el nuevo desarrollo. Cabe destacar, que este nuevo Branch se debe mantener siempre actualizado (Master puede sufrir modificaciones por trabajos que se estén realizando en paralelo).

En el archivo *build.gradle* del módulo, se verifica que se esté trabajando en una nueva versión del módulo. Si, por ejemplo, desde el Branch Master se partió con la versión 1.4.3, entonces el módulo debe aumentar en un dígito su versión (siempre teniendo en cuenta el esquema de *branching* y versionado que se esté utilizando) y se debe agregar la palabra clave “-SNAPSHOT” al final. Esto se realiza siempre en un feature que comienza a desarrollarse.

```
project.ext.set('versionNumber', '1.4.4-SNAPSHOT')
```

2 – Publicación de versión SNAPSHOT

Una vez finalizada la feature, se suben todos los cambios al repositorio (al Branch remoto) y se realiza un PULL REQUEST a MASTER (o al Branch de fixing en caso de que sea desarrollo de un fix). Esta acción deriva en la ejecución automática del pipeline “build_and_snapshot” en Bitrise, que construye y publica el módulo en artifactory como un artefacto “SNAPSHOT”, o sea en desarrollo.

3 – Publicación de versión RELEASE

Una vez que se aprueba el Pull Request, se mergean los cambios del Branch feature a Master (o al Branch de fixing si ese fuera el caso) y, una vez posicionado sobre el Branch fusionado, se debe ir al *build.gradle* del módulo y quitar la palabra clave "-SNAPSHOT" de la versión. De esta manera, se indica que ya no hay más cambios que realizar y, por ende, se procede a publicar una versión cerrada.

```
project.ext.set('versionNumber', '1.4.4')
```

Al ser una versión cerrada, se debe actualizar el archivo "release-notes.txt" (en caso de no existir se debe crear) en donde se indican cuáles son los nuevos cambios con los que va a salir la nueva versión del módulo, o sea, las nuevas *features* o *bugfixing*.

Una vez realizado todos los cambios, se realiza un commit y un push sobre el branch con los cambios hacia el repositorio remoto, indicando también la versión a través de un mensaje. En este caso de ejemplo sería "Versión 1.4.4"

Con los cambios ya reflejados en el Branch remoto, se debe crear un tag para enmarcar la nueva versión. Si, por ejemplo, el reléase es con la versión 1.4.4, entonces el comando que se ejecuta es el de ***git tag -a v1.4.4 -m "reléase 1.4.4"***, que genera la etiqueta de manera local. Para publicar el tag en el repositorio remoto se ejecuta el comando ***git push origin --tags***.

Estas acciones derivan en la ejecución automática del pipeline "build_and_release", que se encarga de compilar el módulo y publicarlo en Artifactory como una versión cerrada, la cual será descargada y utilizada como dependencia por la APP que lo requiera.

4. Conclusiones

Como se mencionó previamente, en el caso de este proyecto, la empresa contratante presentó la necesidad de mejorar los procesos de desarrollo de su aplicación *mobile*,

mejorar su arquitectura y lograr la automatización de las tareas implementando una CI/CD que provea autonomía y autogestión a los distintos equipos involucrados. Por consiguiente, se trazaron una serie de objetivos por parte del equipo con el fin de satisfacer las necesidades del cliente que se fueron cumpliendo durante el desarrollo del proyecto.

A partir de la APP monolítica, se realizó un exhaustivo análisis con el fin de detectar puntos claves para el desarrollo del módulo Core, donde se observaron implementaciones y clases que podían ser migradas. Es aquí donde comenzaron los mayores contratiempos. Las clases a migrar debieron ser refactorizadas de tal forma que pudieran ser escalables y reutilizables para los distintos módulos. Para concretar este objetivo, fue necesario coordinar con los distintos equipos y desarrolladores involucrados, estableciendo un diseño y arquitectura del módulo que permita no solo la inserción de dichas implementaciones sino también sentar las bases para el desarrollo de futuras funcionalidades. Siguiendo este procedimiento, se construyó un módulo Core base, que se fue nutriendo de las clases ya creadas y permitió que se puedan ir agregando nuevas funcionalidades e implementaciones de una manera más sencilla. Esto dio como resultado la concreción de uno de los objetivos principales: la construcción de un módulo robusto que pudiera resolver necesidades comunes y que fuera consumido por distintos módulos y varias APPS.

Asimismo, con la construcción de un módulo base Core, se estableció un esquema de modularización, definiendo lineamientos sobre cómo construir módulos, cómo versionarlos y cómo seguir un proceso adecuado de desarrollo utilizando un esquema de *branching* bien definido. Estas definiciones permitieron que los distintos equipos pudieran responsabilizarse por la construcción y el mantenimiento de cada módulo funcional que se construyese, logrando independencia y autogestión en cada uno de ellos.

Un hito importante durante la ejecución del proyecto fue lograr la automatización de la construcción y despliegue de la APP. A partir de la implementación de la CI/CD de la APP en Bitrise, se logró minimizar la cantidad de *bugs* que surgían durante el desarrollo. Esto fue gracias a las tareas establecidas en los *pipelines*, encargadas de revisar la calidad y estructura del código (funcionan como “checks” para detectar errores). Sumado a la

calidad de código, la CI/CD se adaptó al esquema de *branching* establecido proporcionando la rápida integración de cambios y nuevas funcionalidades en la APP por parte de los Devs. Además, el hecho de tener distintos *pipelines* divididos por ambientes, permitieron proveer APPS construidas específicamente para pruebas, lo que facilitó también el trabajo de los Testers ante cada cambio, *fix* o nueva funcionalidad creada. Todos los beneficios nombrados, en conjunto, derivaron en despliegues de la APP más eficientes y controlados a producción, mejorando considerablemente el *time-to-market*.

Con la CI/CD ya implementada y con el esquema de modularización establecido, cada módulo creado pasa por un proceso de compilación y publicación a través de la automatización de tareas definidas en los *pipelines*. Es decir, cada módulo cuenta con *pipelines* definidos para cada uno de ellos y, de esta manera, son fácilmente integrados a la APP luego de pasar por todos los procesos de construcción, pruebas y testeos.

5. Reflexión sobre la Práctica Profesional Supervisada como espacio de formación

Teniendo en cuenta el proyecto que dio origen a este trabajo, se ha logrado desarrollar todos los objetivos propuestos. En lo personal, creo que esta Práctica Profesional Supervisada ha servido para potenciar mi formación como profesional, integrando muchos de los conceptos vistos en las materias de toda la carrera. Por otro lado, siento que el proyecto canalizó todos los años de experiencia que llevo trabajando en el área ya que fue un nuevo desafío a superar y también, me permitió adquirir experiencia en cuanto a la planificación y la gestión de un proyecto, tomando un rol de referente y logrando la coordinación y comunicación con los distintos equipos que se fueron involucrando.

Otro punto importante, fue el desafío de desenvolverme en un lado técnico más orientado a la arquitectura. Una cualidad que aún no había desarrollado completamente y donde, generalmente, necesitaba acompañamiento de desarrolladores de mayor *seniority*.

Sin embargo, en este proyecto, se me brindó la confianza para poder demostrar mis capacidades en ese aspecto, permitiéndome realizar diseños desde cero, proponer ideas y reestructurar implementaciones ya creadas con el fin de que el sistema pudiera ser escalable y mantenible en el tiempo.

En conclusión, siento que el desarrollo de la PPS fue una experiencia muy enriquecedora que me permitió crecer tanto en lo personal como en lo profesional. Disfruté del trabajo que fui realizando y me sentí orgulloso de lo conseguido hasta el momento. Se vieron resultados muy buenos y recibí un buen *feedback* por parte de mis compañeros y de todos los equipos involucrados, incluyendo al Product Owner perteneciente al cliente. Todos estos hechos generan una motivación en mí que me permiten seguir creciendo y potenciándome en esta hermosa profesión que elegí.

6. Bibliografía

Atlassian. (2019). Ilustración de ejemplo de *step* “Bitbucket server post build status” con estado SUCCESFUL [Ilustración 28]. Recuperado de https://developer.atlassian.com/server/bitbucket/how-tos/updating-build-status-for-commits/

Atlassian. (2019). Ilustración de ejemplo de Step “Bitbucket server post build status” con estado FAILED [Ilustración 29]. Recuperado de https://developer.atlassian.com/server/bitbucket/how-tos/updating-build-status-for-commits/

Atlassian., guía de Jira Software. (2021). *Empezar con Jira Software.* Recuperado de https://www.atlassian.com/es/software/jira. [Consulta: 22 de marzo de 2021]

Atlassian., guía de Bitbucket. (2021). *Breve presentación de Bitbucket.* Recuperado de https://www.atlassian.com/es/software/bitbucket/guides/getting-started. [Consulta: 22 de marzo de 2021]

Atlassian., guía de Confluence. (2021). *Empezar con Confluence.* Recuperado de https://www.atlassian.com/es/software/confluence/guides/get-started/confluence-overview. [Consulta: 22 de marzo de 2021]

Bitrise. (2021). Ilustración de ejemplo de archivo yml [Ilustración 9]. Recuperado de https://devcenter.bitrise.io/bitrise-cli/basics-of-bitrise-yml/

Bitrise. (2021). Ilustración de Step “Install missing Android SDK components” [Ilustración 23]. Recuperado de https://www.bitrise.io/integrations/steps/install-missing-android-tools

Bitrise. (2021). Ilustración de Step “Android Lint” [Ilustración 24]. Recuperado de https://www.bitrise.io/integrations/steps/android-lint

Bitrise. (2021). Ilustración de Step “Android Unit Test” [Ilustración 26]. Recuperado de https://www.bitrise.io/integrations/steps/android-unit-test

Bitrise. (2021). Ilustración de Step “Android Build” [Ilustración 27]. Recuperado de https://www.bitrise.io/integrations/steps/android-build

Bitrise. (2021). Ilustración de Step “Android Sign” [Ilustración 35]. Recuperado de https://www.bitrise.io/integrations/steps/sign-apk

Bitrise. (2021). Ilustración de Step “Google Play Deploy” [Ilustración 36]. Recuperado de https://www.bitrise.io/integrations/steps/google-play-deploy

Bitrise, documentación para desarrolladores. (2021). Bitrise platform. Recuperado de https://www.bitrise.io/. [Consulta: 28 de marzo de 2021]

Chike Mgbemena (2016, Diciembre). *Sending Data With Retrofit 2 HTTP Client for Android*. Recuperado de https://code.tutsplus.com/es/tutorials/sending-data-with-retrofit-2-http-client-for-android--cms-27845

Facebook Open Source. (2021). Stetho. Recuperado de http://facebook.github.io/stetho/. [Consulta: 22 de septiembre de 2021]

Google. (2021). Google Analytics. Recuperado de https://firebase.google.com/docs/analytics. [Consulta: 10 de agosto de 2021]

Google. (2021). Firebase Crashlytics. Recuperado de https://firebase.google.com/docs/crashlytics. [Consulta: 15 de agosto de 2021]

Google. (2021). Firebase Performance. Recuperado de https://firebase.google.com/docs/perf-mon. [Consulta: 31 de agosto de 2021]

Google. (2021). Firebase Remote Config. Recuperado de https://firebase.google.com/docs/remote-config. [Consulta: 02 de septiembre de 2021]

Gradle Inc., manual de usuario (2021). *What is Gradle?* Recuperado de https://docs.gradle.org/current/userguide/what_is_gradle.html. [Consulta: 15 de marzo de 2021].

Jfrog. (2021). Recuperado de https://jfrog.com/artifactory/. [Consulta: 18 de marzo de 2021]

Turrado, J (2019). *Integración continua: qué es y por qué deberías aprender a utilizarla cuanto antes*. Recuperado de https://www.campusmvp.es/recursos/post/integracion-continua-que-es-y-por-que-deberias-aprender-a-utilizarla-cuanto-antes.aspx

Kotlin, guía para desarrolladores. (2021). Recuperado de https://kotlinlang.org/docs/home.html. [Consulta: 16 de marzo de 2021]

Okhttp. (2019). Recuperado de https://square.github.io/okhttp/. [Consulta: 18 de agosto de 2021]

Proceso estándar de CI. (2019). Ilustración de Proceso de CI. [Ilustración 1]. Recuperado de https://www.campusmvp.es/recursos/image.axd?picture=/2019/3T/ProcesoCI.png

RedHat. (2021). Ilustración de Proceso de CI/CD [Ilustración 2]. Recuperado de https://www.redhat.com/es/topics/devops/what-is-ci-cd