

**Universidad de Los Andes**

**Facultad de Ingeniería**

**Minería de repositorios: Benchmark Kotlin Multiplatform, React Native, Flutter**

**Asesor:**

**Camilo Andrés Escobar Velásquez**

**Trabajo de Pregrado 2024-1**

**Estudiante:**

**Santiago Hernández Facio Lince**

**201922432**

**Repositorio del proyecto: https://github.com/shernandezf/Tesis2024**

**Resumen**

En el intrincado y vasto universo del desarrollo de aplicaciones móviles, este documento pretende servir como una guía, más conocida como benchmark, esencial para los desarrolladores de aplicaciones móviles, enfocándose en tres de los frameworks de programación más usados para el desarrollo de aplicaciones híbridas. El objetivo es realizar una comparativa exhaustiva y evaluativa de Kotlin Multiplatform, React Native y Flutter, basada en una serie de métricas relevantes que permitan identificar las fortalezas y debilidades inherentes a cada lenguaje. A través de este análisis, buscamos facilitar la selección del framework de programación más adecuado para futuros proyectos, asegurando una decisión informada y estratégica que responda a las necesidades específicas del desarrollador.

**Palabras clave**

*Benchmark, Aplicaciones Móviles, Framework, React native, Kotlin, Flutter.*

**Introducción**

Ante la creciente diversificación de fabricantes de dispositivos y sistemas operativos para dispositivos móviles, los desarrolladores se encuentran ante un dilema tecnológico significativo al elegir entre desarrollo nativo, web o híbrido. Esta decisión inicial es crucial, ya que sienta las bases para seleccionar posteriormente un lenguaje de programación que no solo cumpla con los requisitos funcionales del proyecto, sino que también optimice los atributos de calidad deseados. En un mercado saturado de opciones, la elección de un lenguaje de programación puede ser abrumadora, especialmente para quienes están a punto de embarcarse en nuevos proyectos o incluso para aquellos que buscan adentrarse en el mundo de la programación. Afortunadamente, es posible clasificar los lenguajes de programación según sus puntos fuertes; mientras algunos se destacan en el desarrollo de videojuegos, otros, como los que nos ocupan, están especialmente diseñados para el desarrollo de aplicaciones móviles híbridas.

A pesar de reducir la elección en tecnologías específicas para desarrollo móvil híbrido, la abundancia de opciones sigue presentando un desafío considerable. En respuesta a esto, desarrolladores de todo el mundo han comenzado a realizar evaluaciones comparativas utilizando diversas métricas, tales como velocidad, usabilidad, mantenimiento y capacidad de respuesta. Los resultados de estas comparativas se analizan para identificar ventajas y limitaciones específicas, lo que culmina en la publicación de informes de referencia o benchmarks que buscan orientar a la comunidad de desarrolladores.

Sin embargo, a pesar de la creciente demanda en el sector de aplicaciones móviles híbridas, aún se observa una escasez de referencias y estudios comparativos profundos que evalúen los lenguajes disponibles en el mercado actual. Esta carencia de información constituye una barrera significativa para la toma de decisiones informadas en un campo tan dinámico y en constante evolución como lo es el desarrollo de aplicaciones móviles.

**Solución**

Ante el desafío de seleccionar eficazmente entre los diversos frameworks disponibles para el desarrollo de aplicaciones móviles híbridas, esta tesis se propone desarrollar un Benchmark confiable y exhaustivo. Este Benchmark comparará tres de los frameworks híbridos más populares en la actualidad: Kotlin Multiplatform, React Native y Flutter. La comparación se realizará desde múltiples perspectivas, utilizando métricas diseñadas específicamente para evaluar los lenguajes de programación subyacentes de cada framework:

* Kotlin: a través de Kotlin Multiplatform.
* JavaScript: utilizando React Native.
* Dart: con Flutter.

Además de las métricas de lenguaje, se establecerán criterios adicionales para comparar directamente los frameworks en sí, así como las aplicaciones finales desarrolladas con cada uno. Estos criterios incluirán la evaluación de la funcionalidad, la eficiencia y la escalabilidad de los frameworks, junto con la experiencia del desarrollador y la calidad de la interfaz de las aplicaciones resultantes. Se analizará también la facilidad de integración con otros sistemas y servicios, además, se considerarán aspectos como la robustez del soporte comunitario y la disponibilidad de recursos de aprendizaje y de documentación, que pueden influir significativamente en la velocidad de desarrollo y en la resolución de problemas. Esta evaluación busca ofrecer una visión completa que no solo mida el rendimiento técnico, sino también la practicidad y eficiencia de los frameworks en escenarios de uso real.

**Objetivos**

* Implementar métricas de calidad básicas y generales que apliquen a todos los frameworks.
* Desarrollar métricas de calidad específicamente adaptadas al entorno del desarrollo móvil.
* Realizar una comparación objetiva de los lenguajes mediante el desarrollo de un proyecto común, permitiendo así evaluar su funcionalidad en condiciones uniformes.
* Facilitar un entorno donde otros desarrolladores puedan replicar o ampliar las comparativas realizadas, promoviendo una colaboración continua y el enriquecimiento del estudio.
* Conducir pruebas controladas para asegurar la precisión y relevancia de los resultados obtenidos.
* Ejecutar pruebas en ambientes de producción reales para validar la aplicabilidad de los frameworks en escenarios operativos auténticos.

**Metodología**

Como se mencionó anteriormente, el objetivo es evaluar la eficiencia tanto del framework como del lenguaje de programación base. Para ello, dividiremos el benchmark en dos partes esenciales. La primera parte se enfocará exclusivamente en los lenguajes de programación. Se llevarán a cabo pruebas de velocidad, monitoreando el tiempo que cada lenguaje requiere para resolver un problema matemático complejo y computacionalmente exigente. Posteriormente, se registrarán y analizarán estos tiempos para determinar qué lenguajes exhiben un rendimiento superior.

En la segunda parte, el análisis se centrará completamente en los frameworks. Esta evaluación abarcará una gama más amplia de variables, incluyendo el tiempo necesario para crear y compilar un nuevo proyecto, así como la calidad y estructura de la documentación y los proyectos internos de los frameworks. También se examinará la integración con servicios web externos, la implementación y gestión de sensores, la precisión en la exhibición de colores, el impacto en el rendimiento de la batería de las aplicaciones y, finalmente, la capacidad de cada framework para desarrollar una aplicación híbrida que sea visualmente atractiva y que incorpore todos los elementos mencionados anteriormente. Este enfoque integral pretende proporcionar una evaluación detallada que refleje la utilidad y la eficacia de los frameworks en condiciones prácticas y operativas reales.

**Primera Etapa**

Antes de empezar con las pruebas es necesario hacer una breve explicación acerca de cómo funcionan los 3 lenguajes de programación y cuál es su uso principal.

Kotlin:

Kotlin es un lenguaje de programación de código abierto de tipo estático dirigido a JVM, Android, JavaScript, Wasm y Native. Está desarrollado por JetBrains. El proyecto comenzó en 2010 y fue de código abierto desde el principio. El primer lanzamiento oficial 1.0 fue en febrero de 2016. (JETBRAINS, 2024)

JavaScript:

JavaScript es un lenguaje de programación que los desarrolladores utilizan para hacer páginas web interactivas. Desde actualizar fuentes de redes sociales a mostrar animaciones y mapas interactivos, las funciones de JavaScript pueden mejorar la experiencia del usuario de un sitio web. Como lenguaje de scripting del lado del servidor, se trata de una de las principales tecnologías de la World Wide Web. (Amazon Web Services, 2023)

Dart:

Dart es un lenguaje optimizado para el cliente para desarrollar aplicaciones rápidas en cualquier plataforma. Su objetivo es ofrecer el lenguaje de programación más productivo para el desarrollo multiplataforma, junto con una plataforma de ejecución flexible para marcos de aplicaciones. (Dart, 2024)

Para realizar una comparación con respecto al rendimiento de los 3 lenguajes de programación se decidió optar por una métrica bastante usada en el mundo de la programación, esta se basa en medir el tiempo de ejecución de un programa, normalmente

un programa computacionalmente exigente para obtener tiempos de medición significativos.

Con esto en mente, definimos los siguientes 3 problemas que involucran cálculos matemáticos extensos:

El Conjunto de Mandelbrot:

La ecuación de Mandelbrot describe un conjunto de números complejos que generan un patrón fractal, conocido como el conjunto de Mandelbrot. Este patrón es una imagen muy famosa que parece un objeto con formas repetitivas y detalles infinitos. La idea básica es que tomas un número complejo 𝑐c y lo usas en una fórmula iterativa: 𝑧𝑛+1=𝑧𝑛2+𝑐zn+1​=zn2​+c. (Coordinador Comunicaciones UniSabaneta, 2024)

En nuestro programa, inicialmente establecemos dos números que representan las dimensiones (altura y anchura) de nuestro plano cartesiano, correspondientes a los ejes Y y X, respectivamente. A continuación, comenzamos a iterar sobre cada punto de este plano. Al llegar a un punto, fijamos un límite máximo de iteraciones y procedemos a determinar si este punto, y los subsiguientes generados por las iteraciones, pertenecen al conjunto de Mandelbrot. Este proceso se repite hasta alcanzar el número máximo de iteraciones establecido. Luego, avanzamos al siguiente punto inicial y repetimos el procedimiento.

Como se puede ver, este algoritmo propone varias iteraciones y cálculos, presentando una complejidad de: . Los 3 lenguajes comparten el mismo algoritmo por ende el tiempo de ejecución recae únicamente en el desempeño del lenguaje.

Criba de Eratóstenes: Este algoritmo se emplea para identificar los números primos dentro de un intervalo específico, X. Comienza con una iteración a partir del número 2 y continúa hasta que cada número ha sido evaluado. La metodología consiste en crear una lista o tabla booleana inicialmente establecida en valores verdaderos. Durante la iteración, utilizando una variable 𝑖, se marcan como falsos todos los múltiplos de 𝑖 que sean menores que X. De esta manera, se van excluyendo progresivamente los números compuestos del conjunto de posibles primos. La eficiencia del algoritmo, cuando se implementa en los tres lenguajes de programación, presenta la siguiente complejidad: .

Secuencia Fibonacci:

Es una secuencia infinita de números naturales cuyos dos primeros términos son 1 y 1 y tal que, cualquier otro término se obtiene sumando los dos inmediatamente anteriores. Por tanto, se cumple la relación: (Universidad de Almería, s.f.)

El algoritmo que hemos implementado está su forma tradicional y recursiva para encontrar el trigésimo quinto número primo no incluye optimizaciones avanzadas, como las que proporcionaría la programación dinámica. Este enfoque básico sigue un método directo sin almacenar los resultados de cálculos previos, lo cual podría reducir significativamente el número de operaciones necesarias al reutilizar estos resultados. Debido a la ausencia de tales mejoras, el algoritmo se enfrenta a una escalada exponencial en el número de operaciones requeridas a medida que aumenta el valor de entrada, lo que se refleja en su complejidad computacional de: . Decidimos ir por la implementación más básica ya que proporcionará tiempo de ejecución mayores, lo cual puede ser beneficioso para la comparación.

Resultados:

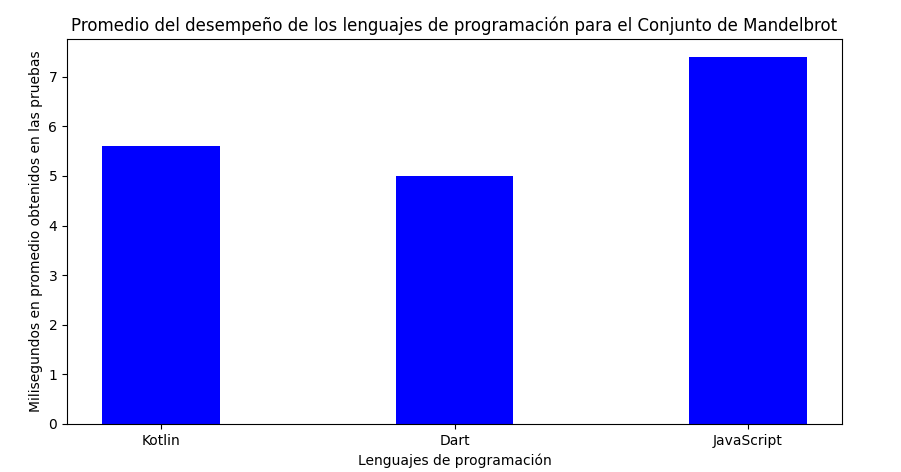
Teniendo en mente que la idea es que cualquier usuario pueda ejecutar estas pruebas en sus propios dispositivos y entornos locales, todos los algoritmos descritos anteriormente, para cada uno de los 3 lenguajes, fueron convertidos en imágenes Docker y posteriormente subidos a Docker Hub, de esta forma, cualquier usuario que cuente con Docker instalado en su respectivo dispositivo puede correr las pruebas de ejecución.

|  |  |  |  |
| --- | --- | --- | --- |
|  | El Conjunto de Mandelbrot | Criba de Eratóstenes | Secuencia Fibonacci |
| Kotlin | pelucapreb/tesis2024-dk:k.c.1 | pelucapreb/tesis2024-dk:k.c.2 | pelucapreb/tesis2024-dk:k.c.3 |
| JavaScript | pelucapreb/tesis2024-dk:d.c.1 | pelucapreb/tesis2024-dk:d.c.2 | pelucapreb/tesis2024-dk:d.c.3 |
| Dart | pelucapreb/tesis2024-dk:js.c.1 | pelucapreb/tesis2024-dk:js.c.2 | pelucapreb/tesis2024-dk:js.c.3 |

Para poder ejecutar las pruebas desde una terminal, es necesario poner el comando “Docker run” y añadir la imagen que se quiere ejecutar, en la tabla se pueden ver los nombres de las imágenes de cada lenguaje y de cada problema a resolver:

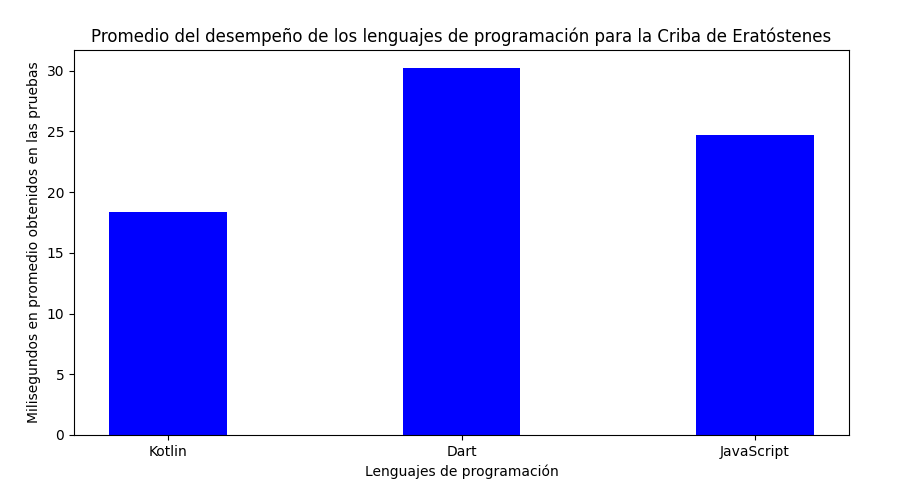
**Tabla #1 nombre imágenes Docker.**

Posteriormente, implementamos tres scripts, uno para cada problema, escritos en Python. Estos scripts se encargan de ejecutar cada imagen cinco veces y calcular un promedio de los tiempos de ejecución para cada lenguaje de programación. Adicionalmente, cada vez que se ejecuta el script de Python, el orden de los lenguajes se selecciona de manera aleatoria. Esto se hace para mitigar cualquier latencia que pueda surgir del uso continuo de recursos computacionales, considerando que el rendimiento del último programa ejecutado podría verse ligeramente afectado. Finalmente, usando una librería llamada Matplotlib, se realiza una gráfica comparativa del promedio de los tiempos de ejecución de los 3 lenguajes.

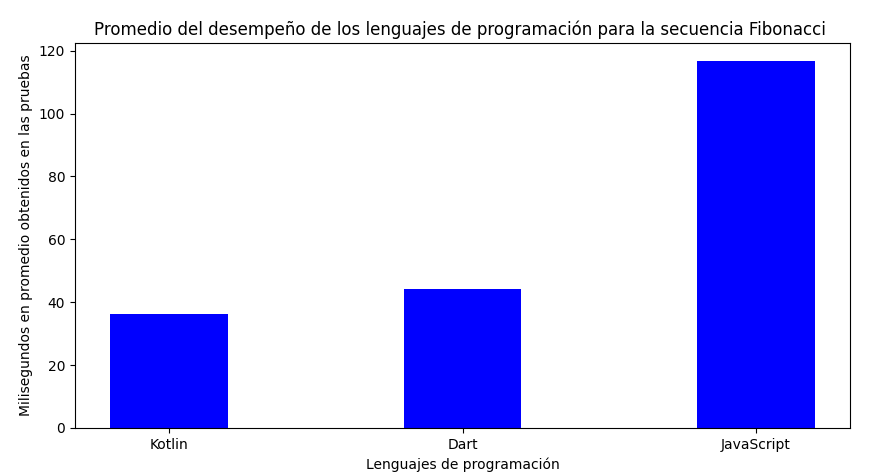
El Conjunto de Mandelbrot:

**Grafica #1 comparación tiempos de ejecución conjunto de Mandelbrot.**

Criba de Eratóstenes:



**Grafica #2 comparación tiempos de ejecución de la criba de Eratóstenes.**

Secuencia Fibonacci:

**Grafica #3 comparación tiempos de ejecución de la secuencia Fibonacci.**

Como se puede ver en las gráficas de los resultados, Kotlin es el lenguaje que mayor velocidad tiene en la ejecución de los programas, seguido de Dart y, finalmente, de JavaScript. Esta jerarquía en la velocidad de ejecución puede atribuirse a varios factores propios de cada lenguaje. Kotlin, diseñado para ser tanto conciso como eficiente, aprovecha la Java Virtual Machine (JVM) para optimizar la ejecución, lo cual resulta en tiempos de procesamiento más rápidos y un mejor rendimiento en comparación con otros lenguajes que no utilizan compilación a bytecode. Por su parte, Dart, aunque inicialmente interpreta el código en tiempo de desarrollo, compila este a código nativo cuando se despliega la aplicación, lo que le permite superar en rendimiento a lenguajes puramente interpretados como JavaScript. JavaScript, siendo un lenguaje de scripting interpretado y con un enfoque en flexibilidad más que en velocidad, tiende a ser más lento en comparación con Kotlin y Dart, especialmente en tareas computacionalmente intensivas. Estas diferencias resaltan la importancia de elegir el lenguaje adecuado basado en las necesidades específicas de rendimiento y eficiencia para cada proyecto de software.

**Segunda Etapa**

Empecemos con una descripción de como es la estructura y funcionamiento de cada uno de los frameworks:

Kotlin Multiplatform:

React Native:

Flutter:

La primera métrica que evaluamos consiste en medir el tiempo que cada uno de los frameworks requiere para compilar el proyecto inicial que generan. Para ello, construimos el APK utilizando el archivo gradlew para el caso de Kotlin Multiplatform y para React Native, mientras que Flutter ya cuenta con un comando para construir el APK: “flutter build apk”. Para cronometrar la duración de estos comandos, empleamos la función Measure-Command de Windows, la cual proporciona el tiempo exacto de ejecución de un comando. Adicionalmente, desarrollamos un script de Windows PowerShell (.ps1) que ejecuta automáticamente estas acciones cinco veces y registra todos los tiempos obtenidos en un archivo externo. Esta metodología nos permite recopilar datos de manera sistemática, facilitando significativamente el análisis comparativo del rendimiento de los diferentes frameworks.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Tiempo 1 | Tiempo 2 | Tiempo 3 | Tiempo 4 | Tiempo 5 | Promedio |
| (Segundos) | (Segundos) | (Segundos) | (Segundos) | (Segundos) | (Segundos) |
| *Kotlin Multiplatform* | 9,68 | 9,08 | 10,57 | 10,07 | 9,48 | 9,78 |
| *React Native* | 11,13 | 8,87 | 10,11 | 11,51 | 9,71 | 10,27 |
| *Flutter* | 4,36 | 3,07 | 3,09 | 3,02 | 2,99 | 3,31 |

Resultados:

**Tabla #2 Desempeño compilación frameworks.**

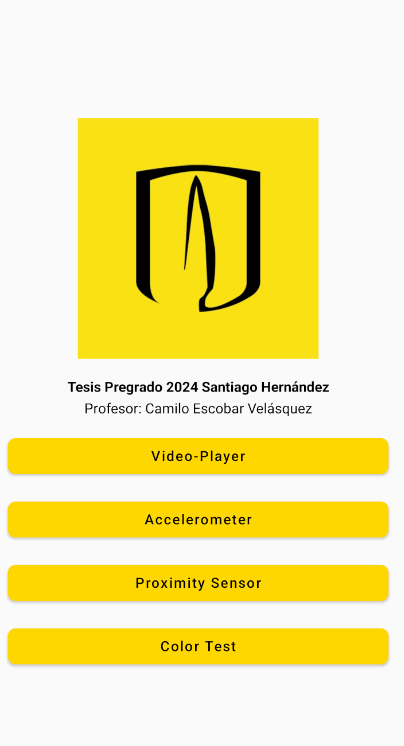
**Grafica #4 comparación desempeño compilación frameworks.**

En la comparación de tiempos para la creación de APK e instalación en un dispositivo Android, se observa que Kotlin Multiplatform y React Native presentan un rendimiento inferior en comparación con Flutter. Una de las razones principales de esta diferencia radica en los procesos de compilación que cada uno emplea. Tanto Kotlin Multiplatform como React Native requieren el uso de los comandos ./gradlew clean assembleRelease para la generación del APK. Este proceso implica pasos adicionales de limpieza y ensamblaje del proyecto, lo que puede aumentar significativamente el tiempo total de compilación. Por otro lado, Flutter ofrece un comando integrado flutter build apk, que está optimizado específicamente para compilar aplicaciones de manera más eficiente y rápida. Esta integración y optimización en Flutter reduce la cantidad de pasos necesarios y acelera el proceso de compilación, lo que resulta en un mejor rendimiento para la creación del APK y su instalación en dispositivos Android.

Continuando con las pruebas, como se mencionó anteriormente, se definió una aplicación en común para desarrollar en los 3 frameworks. La idea es que las 3 aplicaciones sean lo más similares posibles para evaluar y comparar las herramientas y facilidades que cada framework entrega a sus desarrolladores y comparar, haciendo un resultado final en común, como es el proceso de desarrollo de cada framework. Adicionalmente, dentro de la aplicación se encuentran 4 elementos que generan una comparación más completa que abarca distintos ámbitos del desarrollo móvil:

* Integración con servicios web: Esta prueba se realiza mediante la extracción de un video alojado en la web y posteriormente la visualización de este. Durante este proceso se mide el tiempo que le toma al programa ir a extraer y posteriormente a mostrar el video.
* Integración con sensor acelerómetro: Se realiza la integración del acelerómetro y se muestra en pantalla los valores X,Y,Z obtenidos por el sensor.
* Integración con sensor de proximidad: Se realiza la implementación del sensor de proximidad y se muestra en pantalla si el usuario está cerca o lejos del sensor.
* Prueba de visualización de color: Se hace una visualización de 4 colores definidos por su código hexadecimal y se realiza la comparación entre los frameworks.

La página principal de la aplicación se ve de la siguiente forma:

****

Para la creación de una pantalla inicial en tres frameworks distintos, se tomaron en cuenta diversas consideraciones técnicas que influyen en la metodología y la eficiencia del desarrollo.

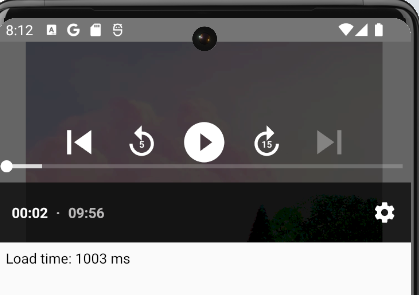
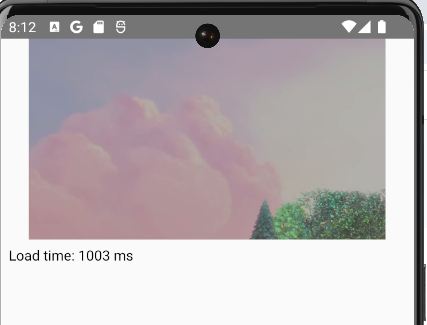
Kotlin Multiplatform: Utilizamos Jetpack Compose, que es la herramienta moderna de Kotlin para el desarrollo de interfaces gráficas. Jetpack Compose reemplaza la antigua metodología basada en XML, ofreciendo un enfoque más declarativo y reactivo para construir UIs. Este cambio no solo simplifica el código, sino que también mejora el rendimiento al minimizar la complejidad del árbol de vistas y reducir la carga de inflado de XML en tiempo de ejecución.

React Native: Este framework aprovecha tecnologías de desarrollo web para la creación de aplicaciones móviles. Utiliza HTML para estructurar la interfaz, CSS para el diseño y estilos, y JavaScript para agregar interactividad. En nuestro proyecto, empleamos JavaScript, aunque versiones recientes de React Native soportan también TypeScript. La principal ventaja de React Native es su capacidad para permitir a los desarrolladores web migrar fácilmente al desarrollo móvil, reutilizando conocimientos y código entre plataformas.

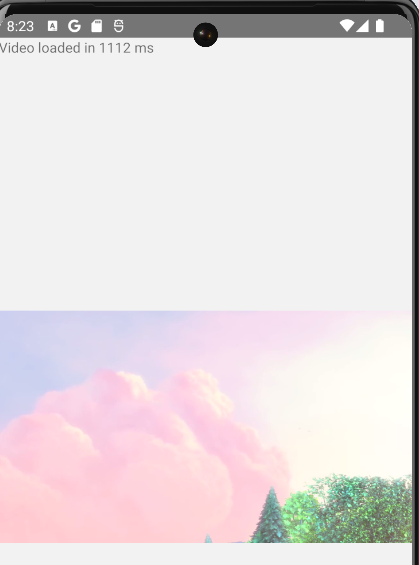
Flutter: Este framework se basa en la creación de interfaces a través de una extensa biblioteca de widgets, cada uno con propiedades y métodos específicos para configurar su comportamiento y apariencia. La arquitectura de widgets de Flutter es similar en algunos aspectos a Jetpack Compose, en tanto ambos proporcionan un modelo altamente modular y reutilizable para el diseño de UI. Flutter facilita la creación de interfaces fluidas y responsivas que se compilan en código nativo, ofreciendo un alto rendimiento en múltiples plataformas.

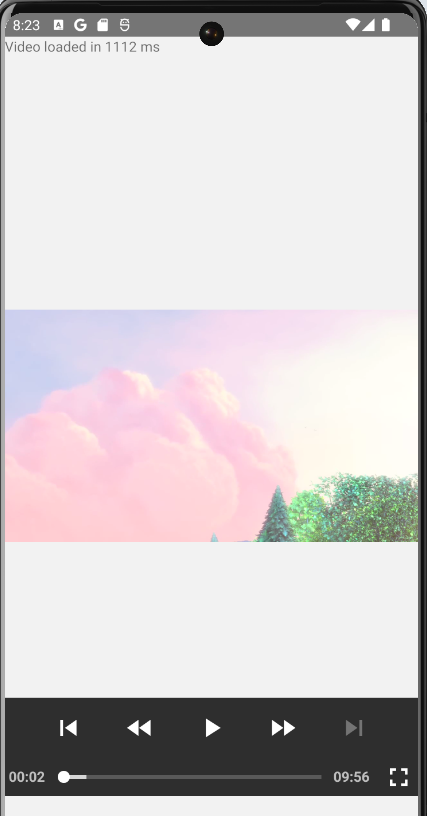
En resumen, cada framework presenta un enfoque distinto: Kotlin Multiplatform y Jetpack Compose optimizan el desarrollo nativo de Android con una moderna API de UI, React Native integra tecnologías web para una experiencia de desarrollo cruzado, y Flutter ofrece un robusto sistema de widgets que soporta un desarrollo de UI altamente dinámico y multiplataforma.

Integración con servicios Web:

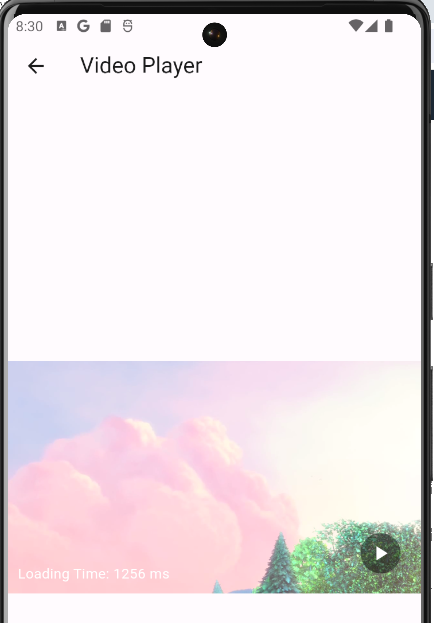
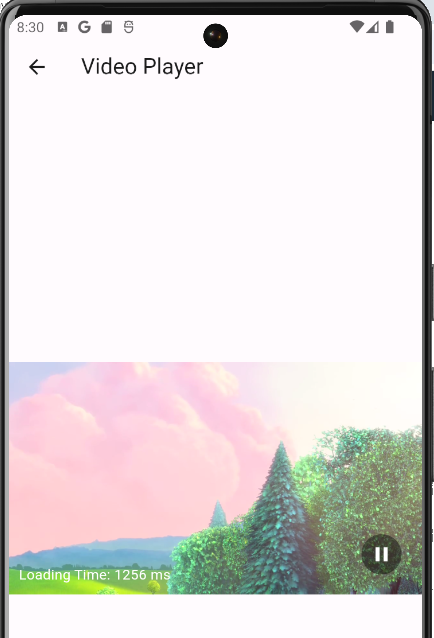
Kotlin Multiplatform: Para la extracción e integración de un video alojado en la web, se utilizó la librería Media3 ExoPlayer, la cual es desarrollada directamente por Android y cuanta con una gran documentación y varios ejemplos de su funcionamiento. La librería provee con todos los elementos de manipulación del video para el usuario: Adelantar el video, pausar, reanudar, reiniciar y avanzar. Lastimosamente la librería no funciona de forma híbrida por ende solo funciona en la sección exclusiva de desarrollo Android, en este caso, tocaría realiza 2 implementaciones por separado, una para Android y otra para IOS con alguna otra librería . Así se ve la implementación:

React Native: React también ofrece su propia librería para la extracción e integración de contenido multimedia como videos, se “llama React-native-video”, su manejo es bastante sencillo y provee al usuario también todas las funciones de manejo del video, sin embargo estás no están integradas de forma visualmente atractiva como en Kotlin. Esta librería funciona de manera híbrida por ende solo toca hacer una implementación:





Flutter: Este framework también proporciona una librería nativa al igual que en los casos anteriores, sin embargo, a pesar de contar con una buena documentación y ejemplos de uso, su implementación es un poco más compleja, así como los pasos necesarios previos a utilizar la librería. Esta librería, a diferencia de las anteriores, solo provee al usuario un botón de pausar y reanudar el video, si se deseará implementar las demás herramientas, sería tarea del desarrollador implementarlas.



Como se puede notar, la distribución gráfica cambia con respecto a cada framework, esto se hizo así para mostrar las diferencias entre los mismos. En Kotlin, el video se ubica por defecto en la parte superior de la pantalla, las herramientas de manipulación se ubican encima del video y el tiempo requerido para extraer el video va en la parte inferior por fuera del video. En React Native, el video se ubica automáticamente en la mitad de la pantalla y las herramientas de manipulación, se ubican en la parte inferior finalmente el tiempo requerido se ubica en la parte superior. Con flutter, el video también se ubica en la mitad de la pantalla del dispositivo, pero tanto el tiempo de extracción como el botón de pausa y reanudar se ubican sobre el video, dificultando así su visualización.

Resultados:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Tiempo 1 (Milisegundos) | Tiempo 2 (Milisegundos) | Tiempo 3 (Milisegundos) | Tiempo 4 (Milisegundos) | Tiempo 5 (Milisegundos) | Promedio  (Milisegundos) |
| Kotlin Multiplatform | 1002 | 922 | 707 | 876 | 712 | 843,8 |
| React Native | 2040 | 980 | 634 | 859 | 733 | 1049,2 |
| Flutter | 954 | 912 | 1429 | 939 | 793 | 1005,4 |

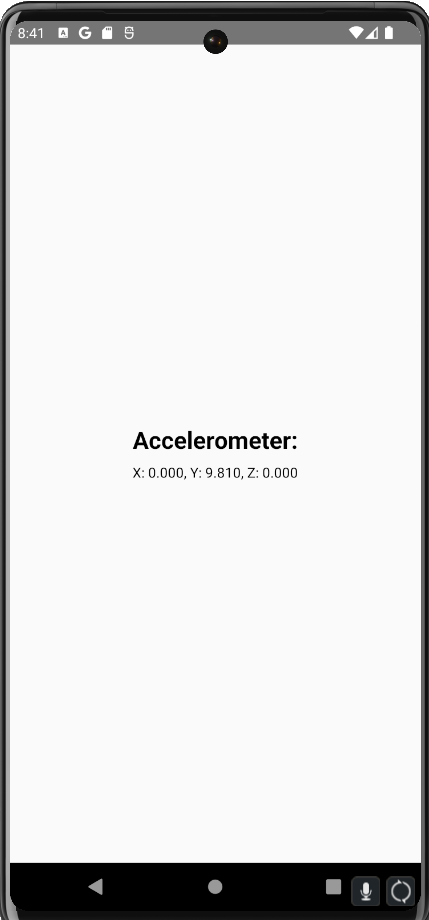
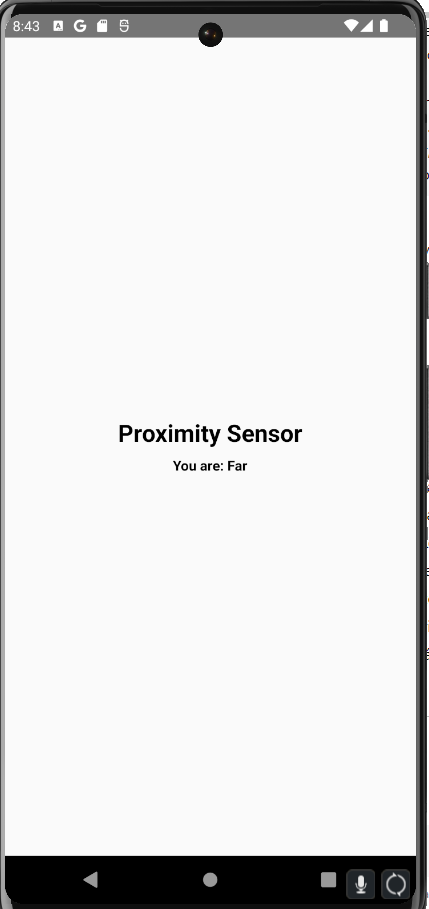
**Tabla #4 Desempeño de interacción con servicios web.**

**Grafica #5 comparación desempeño de interacción con servicios web.**

En esta prueba, los resultados obtenidos indican que no existe una diferencia significativa en los tiempos de extracción y visualización de contenido audiovisual entre los tres frameworks evaluados. Cada uno de los frameworks demostró tener tiempos de rendimiento similares en ambas tareas. Este comportamiento sugiere que, independientemente del framework elegido, los usuarios pueden esperar un desempeño consistente en términos de eficiencia en la extracción y visualización de contenido de video. Esto es particularmente relevante para aplicaciones que requieren una respuesta rápida y confiable, ya que permite a los desarrolladores optar por el framework que mejor se adapte a otras necesidades específicas de su proyecto sin comprometer el tiempo de procesamiento del video.

Integración con sensor acelerómetro y de proximidad:

Uno de los mayores desafíos en el desarrollo de aplicaciones híbridas radica en la limitación de acceder a características específicas de cada sistema operativo o dispositivo móvil. Al crear una única aplicación que se puede ejecutar en múltiples sistemas operativos móviles, se sacrifica la capacidad de aprovechar funcionalidades exclusivas y optimizadas para cada plataforma. Por lo tanto, una prueba fundamental para este benchmark será evaluar cómo y cuán fácil es integrar y utilizar sensores específicos del dispositivo en una aplicación híbrida, siendo el acelerómetro y el sensor de proximidad de interés en este caso. Este análisis permitirá determinar la efectividad y eficiencia con la que las aplicaciones híbridas pueden interactuar con el hardware del dispositivo, lo cual es crucial para aplicaciones que dependen de datos precisos y en tiempo real proporcionados por dichos sensores.

Esta es la distribución que presentan las 3 aplicaciones del sensor de proximidad y el acelerómetro:

Kotlin Multiplatform: La integración con sensores tiene el mismo problema que con la integración web, aún no existe una librería o interfaz que el desarrollador pueda utilizar para acceder a los sensores de los dispositivos de manera híbrida, toca utilizar librerías separadas, una para el desarrollo Android y otro para el desarrollo en IOS. Dejando eso de lado, la integración es igualmente relativamente sencilla, por ejemplo desde el punto de vista de Android, Kotlin proporciona varias librerías enfocadas en el hardware del dispositivo, cuenta con librerías para acceder a varios sensores ya incluidos, como lo son el sensor de proximidad y el acelerómetro. Finalmente, también cuenta con varias ayudas al programador, como definir eventos y manejadores de eventos frente algún cambio en la detección del sensor.

React Native: Para el acceso y manipulación de diversos sensores en dispositivos móviles, existe una biblioteca nativa llamada “react-native-sensors”. Esta biblioteca permite trabajar con sensores como el acelerómetro de manera eficiente, aunque no incluye el sensor de proximidad entre sus funcionalidades. Afortunadamente, hay bibliotecas externas desarrolladas por la comunidad de usuarios que suplen esta carencia. Una de estas bibliotecas es “react-native-proximity-sensor”, creada por el usuario de GitHub Kibolho, que permite el acceso y la manipulación del sensor de proximidad.

La biblioteca “react-native-sensors” destaca por su completa documentación y ejemplos de uso, facilitando considerablemente el monitoreo y la utilización de los datos obtenidos de los sensores. Por otro lado, “react-native-proximity-sensor” también cuenta con una documentación adecuada y ejemplos prácticos que facilitan su implementación. La instalación de esta biblioteca es sencilla, y su uso es bastante similar al de los sensores nativos, lo que agiliza el proceso de integración.

En general, la implementación y el uso de ambas bibliotecas resultaron ser eficientes y sin complicaciones. No se requirió un esfuerzo significativo adicional para integrar estos sensores en la aplicación, y los resultados obtenidos fueron los esperados. Esto permitió a la aplicación hacer uso de ambos sensores, mejorando su capacidad de interactuar con el hardware del dispositivo y ofreciendo una experiencia de usuario más rica y funcional, es importante denotar que estas librerías ambas son enfocadas para el desarrollo híbrido, por ende con una sola implementación pueden ser utilizadas tanto en IOS como en Android.

Flutter: En el caso de Flutter, se presenta una situación similar a la de React Native, pero con algunas diferencias clave. En Flutter, el sensor de proximidad está incluido dentro de la biblioteca nativa o estándar de sensores ofrecida por el framework, llamada “All Sensors”. Por otro lado, el sensor de acelerómetro no está incluido en esta biblioteca estándar, sino que se encuentra en una biblioteca creada por la comunidad, llamada “sensors\_plus”.

Ambas bibliotecas, “All Sensors” y “sensors\_plus”, cuentan con una extensa documentación y numerosos ejemplos de uso, lo que facilita considerablemente su implementación. La instalación de la biblioteca externa “sensors\_plus” es breve y clara, permitiendo una integración rápida y sin complicaciones. Una vez completada la implementación de ambas bibliotecas, los resultados obtenidos fueron satisfactorios, demostrando el correcto funcionamiento de los sensores.

En conclusión, todos los sensores pudieron ser integrados correctamente, obteniendo resultados consistentes en cuanto a su funcionamiento. Sin embargo, la implementación presentó algunas diferencias significativas. Ninguna de las bibliotecas nativas contenía todos los sensores deseados, lo que obligó a recurrir a bibliotecas creadas por la comunidad para completar la integración.

En el caso de React Native, fue necesario utilizar una biblioteca externa para el sensor de proximidad, mientras que en Flutter, el sensor de acelerómetro se encontraba en una biblioteca comunitaria. A pesar de estas limitaciones, ambas plataformas proporcionaron una experiencia de implementación relativamente sencilla y eficiente, gracias a la buena documentación y los ejemplos prácticos disponibles para estas bibliotecas.

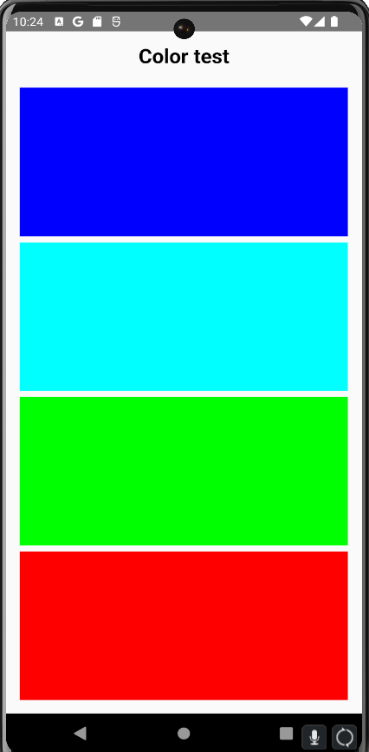
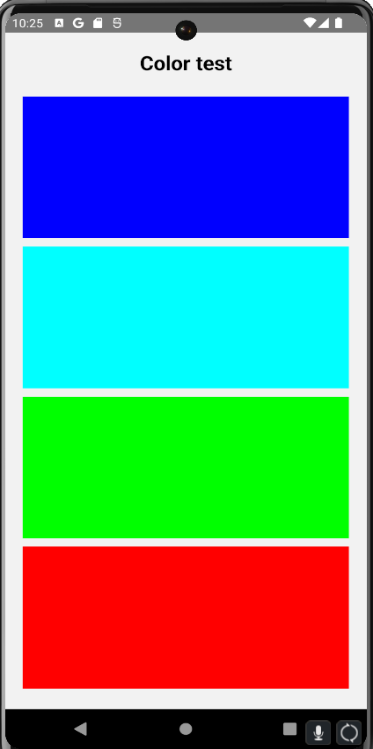
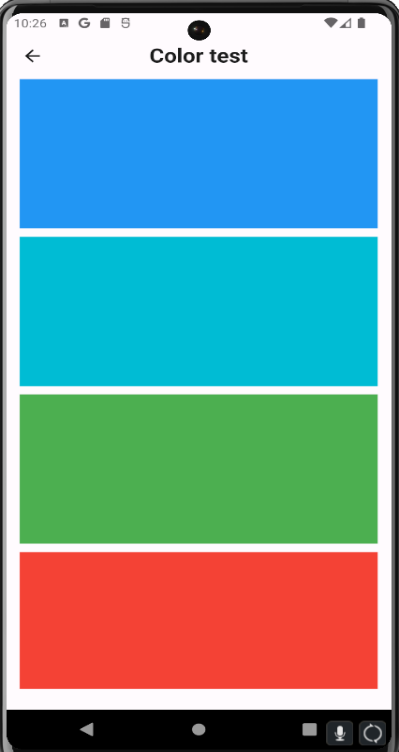
Por otro lado, el peor desempeño en términos de implementación fue presentado por Kotlin Multiplatform. Este enfoque no dispone de ninguna biblioteca híbrida de sensores, lo que obligó a desarrollar soluciones individuales para cada sistema operativo. Esta falta de un soporte unificado y la necesidad de desarrollar de manera específica para cada plataforma resultaron en un proceso más laborioso y menos eficiente.

En resumen, aunque todos los sensores fueron finalmente integrados y funcionaron correctamente, las diferencias en el soporte de bibliotecas y la facilidad de implementación variaron considerablemente entre las distintas plataformas, destacándose Kotlin Multiplatform como la opción con mayores desafíos en este aspecto.

Prueba de visualización de color:

En la fase conclusiva de nuestro estudio, implementamos un análisis comparativo enfocado en cuatro colores fundamentales: azul, cian, verde y rojo. Esta parte del estudio tuvo como finalidad explorar y documentar la consistencia en la visualización de estos colores en tres diferentes frameworks. Específicamente, buscamos identificar cualquier diferencia en la percepción de un color, buscando identificar si se mantiene uniforme o varía entre los frameworks evaluados. Esta comparación es esencial, ya que una representación precisa y uniforme de los colores es vital para el diseño efectivo de la interfaz de usuario en aplicaciones, afectando directamente la experiencia del usuario y la usabilidad del software.

Resultados



Kotlin Multiplatform React Native Flutter

Como se observa en la comparativa de colores, Kotlin Multiplatform y React Native muestran una visualización de los cuatro colores seleccionados (azul, cian, verde, rojo) que es considerablemente más brillante y con una tonalidad más intensa comparada con Flutter. Esta diferencia en la saturación de los colores puede tener implicaciones significativas en la experiencia del usuario. En muchos casos, los colores más vibrantes y cálidos, como los presentados en Kotlin y React, pueden ser percibidos como más atractivos y acogedores, potencialmente mejorando la experiencia del usuario. Por otro lado, la representación menos llamativa de los colores en Flutter podría resultar en una percepción menos favorable, donde los usuarios pueden encontrar la interfaz visualmente menos atractiva o energética. Estas variaciones en la percepción de los colores son críticas para el diseño de la interfaz, ya que pueden influir directamente en la aceptabilidad y preferencia de la aplicación por parte de los usuarios.

**Comparaciones finales:**

Teniendo en cuanta las distintas pruebas realizadas y los resultados obtenidos, se pueden evidenciar varios puntos a favor y en contra para cada uno de los frameworks:

Kotlin Multiplatform: Es el framework más nuevo en la comparativa, por ende, es el que menos cuenta con recursos como librerías nativas, documentación, librerías, herramientas o tutoriales externos creadas por la comunidad, por lo que, en este momento, si se desea realizar algún trabajo muy específico, por ejemplo utilizar un sensor no tan común como los evaluados, puede llegar a ser una tarea un poco compleja para el desarrollador, además toca añadir que muy seguramente toque realizar 2 implementaciones, una para cada sistema operativo.

Sin embargo Kotlin Multiplatform está siendo desarrollado por JetBrains, una reconocida compañía de software, sin duda, este framework con el paso del tiempo empezará a ganar popularidad, incrementando así sus recursos, tanto nativos como de la comunidad. Adicionalmente no hay que olvidar que Kotlin fue el lenguaje que mejor desempeño tuvo en términos de velocidad de procesamiento, esto es un factor sumamente importante ya que a pesar de que normalmente los procesamientos computacionalmente complejos son realizados en el Backend, es de suma utilidad contar con un lenguaje de programación rápido y eficiente, esto tiene un impacto directo por ejemplo en la renderización de los elementos gráficos de la aplicación.