**Universidad ORT Uruguay**
**Facultad de Ingeniería**

# Tecnologías de software para interfaz de usuario adaptativa y multiplataforma

Entregado como requisito para la obtención del título de

Licenciado en Ingeniería de Software

**Martina Severo – 192680**

**Santiago Tonarelli – 229484**

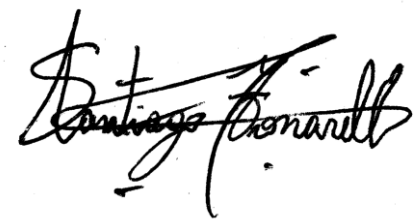
**Tutor: Martín Solari**

**2021**

## Declaración de autoría

Nosotros, Martina Severo y Santiago Tonarelli, declaramos que el trabajo que se presenta en esta obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos el trabajo integrador de la Licenciatura en Ingeniería de Software;
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- En la obra, hemos acusado recibo de las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.



Martina Severo

27 de julio de 2021

Santiago Tonarelli

27 de julio de 2021

## Abstract

Las interfaces de usuario sirven como mecanismo de comunicación entre el usuario y las aplicaciones. Debido a esto, se convierte en uno de los aspectos fundamentales a la hora de definir la calidad de una aplicación. Con el surgimiento de los dispositivos móviles, la adaptabilidad y portabilidad se han convertido en un desafío a la hora de desarrollar interfaces de usuario. En este proyecto, se investigarán las bases conceptuales y tecnológicas para desarrollar interfaces de usuario de software adaptativas y multiplataforma. Actualmente, existen diversas tecnologías que implementan estrategias arquitectónicas para afrontar este problema. En este proyecto, se estudian tres tecnologías que soportan el desarrollo de interfaces de usuario: React.js, React Native y Flutter. A partir del desarrollo de una aplicación para cada una de ellas, obtuvimos un resultado que es adaptable a distintas pantallas y distintos entornos de ejecución (web, Android, IOS). Observamos aspectos comunes entre las tecnologías estudiadas y también, ciertas diferencias entre resultados, como la personalización de componentes, diferencias entre plataformas e implementación del sistema de diseño.

## Palabras clave

Adaptabilidad, Flutter, Fragmentación, Interfaz de usuario, Material Design, Portabilidad, React.js, React Native, Sistema de diseño.

# Índice

# 1. Introducción

## 1.1. Contexto

En la medida en que el software es usado cada vez por más personas, la interfaz de usuario, es decir, el medio por el cual los usuarios se comunican con las aplicaciones se convierte en uno de los aspectos fundamentales de su calidad.

La ingeniería de la interfaz de usuario es una tarea compleja que debe abordar en forma interdisciplinaria aspectos de diseño, cognición humana y tecnología de software. Haciendo enfoque en el punto de vista técnico, uno de los mayores desafíos para la ingeniería de interfaz de usuario es que sea adaptativa y multiplataforma. Cuando hablamos de adaptabilidad, nos referimos a la facilidad para poder adaptarse a diferentes entornos o condiciones. Esto se relaciona al concepto de ser multiplataforma, es decir, capaz de funcionar en distintas plataformas. Ejemplos de éstas últimas pueden ser Android, iOS o la web, teniendo en cuenta que cada una tiene sus propias características, ya sea en cuanto a estilos o funcionamiento.

Uno de los desafíos que se nos presenta hoy en día respecto a la interfaz de usuario es el contexto de uso en dispositivos móviles, particularmente en los celulares. Este desafío, no solamente implica un dispositivo o una pantalla, sino mucho más, ya que se trata de un dispositivo que está mucho más cerca del usuario. Se usa de una forma distinta, de manera más continua, pero a la vez más interrumpida. Se usa para nuevas tareas, que antes no eran posibles con la computación de escritorio.

Actualmente, la mayoría de las aplicaciones de software tienen como requisito que su interfaz sea adaptable en distintos tipos de pantallas, dispositivos y contextos de ejecución. Cada una de estas dimensiones se ha multiplicado en alternativas: resolución de pantalla y densidad de píxeles, navegadores web y dispositivos móviles, preferencias del usuario en el ingreso y la salida de datos. Adicionalmente, la interfaz de usuario debe ser accesible para usuarios con distintas necesidades y cumplir con normas técnicas. Es importante destacar que

los clientes y usuarios demandan una mayor calidad visual en la interfaz de usuario, así como poder estilizarla de acuerdo con una determinada identidad. Todo esto sin dejar de lado aspectos como la usabilidad, performance, mantenibilidad y seguridad de la aplicación.

Otro desafío que se nos presenta es el del crecimiento exponencial y la diversidad de los usuarios actuales que se ha presentado en el correr de los años. Anteriormente, teníamos una era de la computación dirigida a los especialistas. Hoy en día, estamos en una era donde la computación es para todos, sin hacer distinciones.

Todo lo mencionado es importante a la hora de considerar la adaptabilidad y la multiplataforma relacionado a la interfaz de usuario, ya que se debe tener en cuenta al público objetivo y a la situación actual.

## 1.2. Objetivos

Los objetivos establecidos son una continuación de la tesis “Impacto de los sistemas de diseño en la ingeniería de la interfaz de usuario del software”, realizada por Andrés Mauro, en la cual se estudiaba sobre sistemas de diseño, teniendo éstos como meta hacer las interfaces de usuario adaptables, usables y multiplataformas.

En este proyecto, se investigarán las bases conceptuales y tecnológicas para desarrollar interfaces de usuario de software adaptativas y multiplataforma.

A su vez, tendremos en cuenta estrategias arquitectónicas para resolver los problemas de calidad. Esto es, decisiones a alto nivel y más fundamentales que se deberían tomar y que definen como se van a abordar ciertos problemas de calidad. El hacer que una aplicación sea multiplataforma o adaptable dependerá de dichas estrategias, por ello son relevantes.

Utilizaremos, como sistema de diseño, Material Design, el cual, según su propia definición, “…se basa en las mejores prácticas en los campos del diseño web y tradicional, y se basa en la investigación de la experiencia del usuario y la ciencia cognitiva. Las pautas de diseño que se desarrollaron a partir de los hallazgos de

la investigación están destinadas a aplicarse universalmente en todas las plataformas y dispositivos.” [1]

## 1.3. Metodología

Primero estudiaremos el problema, su evolución histórica y las estrategias arquitectónicas. Posteriormente, analizaremos diversas tecnologías que permiten desarrollar aplicaciones multiplataforma adaptables utilizando el sistema de diseño Material Design, tales como React.js, React Native y Flutter. Luego apuntaremos a la validación de ciertos puntos basándonos en lo anterior:

- cómo está compuesta su arquitectura,
- que componentes posee,
- que tan integrado está con el sistema de diseño que propone Material Design,
- grado de dificultad de aprendizaje,
- complejidad del código para poder crear componentes de dicho sistema de diseño,
- herramientas necesarias y resultados en distintas plataformas.

Por un lado, tendremos el estudio teórico, el cual incluye varios pasos como definir los conceptos base, estudiar las soluciones propuestas a nivel de plataformas de desarrollo y frameworks y desarrollar brevemente el modelo de calidad que está por detrás de la adaptabilidad.

Por otro lado, también vamos a realizar un trabajo práctico, es decir, definir el estudio de un caso o prototipo, utilizando distintas tecnologías y para más de una plataforma. Además, como parte de dicho trabajo práctico, definiremos las plataformas, técnicas a abordar y dimensiones o variables bajo las cuales evaluaremos los resultados.

# 2. Marco conceptual

## 2.1. Interfaz de usuario

La interfaz de usuario (UI por sus siglas en inglés) es un medio a través del cual el usuario interactúa con un sistema o aplicación en un dispositivo (computadora, móvil, etc.). Es el conjunto de todos los componentes de un sistema interactivo que proporcionan información y controles para que el usuario realice tareas específicas con el sistema interactivo. Una buena UI debe ser capaz de minimizar el esfuerzo requerido por parte del usuario, a la vez que debería cumplir con ciertas características que faciliten su usabilidad y modificabilidad. [2] [3] [4] [5]

Los aspectos más relevantes a la hora de diseñar la interfaz de usuario incluyen la adaptabilidad, la mantenibilidad y la usabilidad, en los cuales indagaremos con más detalle posteriormente.

## 2.2. Evolución histórica de la UI

### 2.2.1. La Web

La Web es el universo de información accesible por red y la encarnación del conocimiento humano. Tiene un cuerpo de software y un conjunto de protocolos y convenciones. A través del uso del hipertexto (texto con enlaces) y técnicas multimedia, la web es fácil, para cualquier persona, de navegar, buscar y contribuir. [6]

La web es concebida como un mundo sin fisuras, en el cual toda información, sin importar la fuente, puede ser accedida de forma consistente y simple. Su principio de lectores universales es que, una vez que los datos están disponibles, deberían ser accesibles desde cualquier tipo de computadora, en cualquier país, y una persona (autorizada) debería tener que usar solo un simple programa para acceder a ellos. [7]

Con el surgimiento de dispositivos móviles poderosos en los últimos años, el rol de la web como una plataforma para contenido, aplicaciones y servicios en dichos dispositivos se ha tornado sumamente importante.

Por ello, hoy en día, podemos decir que el unificador universal es la web mobile o móvil. Una persona puede tener Android, Windows, pero siempre va a tener a su disposición la web, sin importar la plataforma en donde esté. Estamos en una situación donde la promesa inicial de la web de dar acceso a todos los usuarios, independientemente del dispositivo donde están, se está realizando.

El desarrollo extenso de dispositivos móviles habilitados para la Web los hace un foco de elección para los creadores de contenido. Entender sus fuerzas y limitaciones y usar tecnologías que se amolden a estas condiciones es clave para crear contenido web compatible con dichos dispositivos. [8]

La web móvil también ha logrado una estabilidad con respecto a los estándares que permiten su funcionamiento. Por ejemplo, JavaScript es un lenguaje que corre en todos los navegadores, tanto mobile como desktop. Ejemplos de estos serían Microsoft Edge, Google Chrome, etc. Debido a esto, la programación de frontend, en gran medida, se hace basándose en los estándares HTML y JavaScript porque están disponibles en todos los dispositivos que tienen interfaz de usuario.

Los dispositivos móviles de hoy en día cuentan con un navegador web, sin hacer distinciones entre plataformas. Por esto, podemos destacar la relevancia de la web móvil cuando hablamos de interfaces de usuario.

#### 2.2.1.1. HTML

HTML es el lenguaje para describir la estructura de las páginas web. Les da a los autores medios para:

- Publicar documentos online con encabezados, texto, tablas, listas, imágenes, etc.

- Conseguir información online por medio de enlaces de hipertexto, con tan solo hacer un clic.
- Diseñar formas para conducir transacciones con servicios remotos, para usar en búsquedas de información, hacer reservaciones, ordenar productos, etc.
- Incluir hojas de cálculo, clips de video, sonido y otras aplicaciones directamente en los documentos.

Con HTML, los autores describen la estructura de las páginas utilizando markup. Los elementos del lenguaje etiquetan piezas de contenido tales como “paragraph”, “list”, “table”, entre otros. [9]

#### 2.2.1.2. JavaScript

JavaScript es un lenguaje de secuencia de comandos (scripting) que se utiliza para agregar comportamiento a las páginas web. Puede ser usado para validar los datos ingresados en un formulario (por ejemplo, ver si está en el formato correcto o no) y proveer la funcionalidad de arrastrar y soltar. También para cambiar estilos sobre la marcha, animar elementos de la página tales como menús y manejar funcionalidades de botones, entre otros. [10]

### 2.2.2. Layout líquido/fluido

En primer lugar, el concepto de layout hace referencia a la manera en que están distribuidos los elementos y formas dentro de un diseño. El diseño de un layout trata sobre el arreglo de estos objetos visuales dentro de una cuadricula (grid) con el fin de transmitir un cierto mensaje. Si el layout no es leído y percibido correctamente por el usuario, el diseño es ineficaz, sin importar que tan moderno se vea. [11]

Un layout fluido o líquido es un tipo de diseño de página web en el cual el layout de la página se reajusta a medida que el tamaño de la ventana cambia. Esto se logra definiendo área de la página usando porcentajes en vez de anchos fijos de pixeles. Estos layouts se diseñan con la idea de que se expandan hasta ocupar el total disponible de la pantalla del dispositivo que

renderiza la página web, redimensionando sus elementos si se cambia la resolución o si se visualiza la página desde diferentes dispositivos. [12] [13] [14]

En la imagen 1 se puede observar un ejemplo de esto, donde los elementos mostrados dependen del tamaño de la pantalla. En la misma se puede ver que cuando la ventana del navegador se encuentra más expandida, los datos mostrados ocupan más espacio (izquierda). Cuando el tamaño de la ventana se reduce, la información se reajusta, cambiando su disposición (derecha).

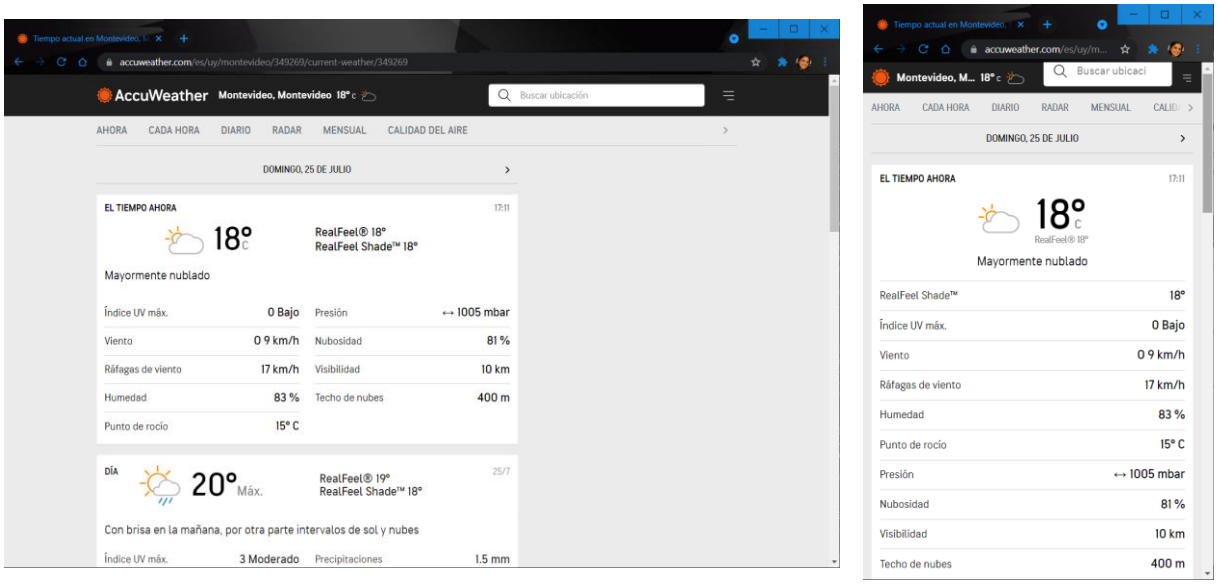



Imagen 1 - Efecto del layout líquido según el tamaño de la pantalla

### 2.2.3. Diseño responsive

El diseño responsive responde a cambios en el ancho de la ventana al ajustar el posicionamiento de los elementos de diseño para que quepan en el espacio disponible. Es fluido y se adapta al tamaño de la pantalla sin importar cual sea el dispositivo. [15] [16]

Por ejemplo, cuando nos referimos a un sitio web responsive, hablamos de una web capaz de adaptarse al formato del dispositivo sin preguntar, mostrando lo mismo a todos los usuarios. Por esto, los datos son más fidedignos, contribuyendo a mejorar el posicionamiento y pudiendo ofrecer una funcionalidad similar a una aplicación. Los mayores inconvenientes del

uso de sitios web responsives son que la aplicación no se puede distribuir a través de una tienda de aplicaciones y que el usuario necesitará conectividad constante para hacer uso del sitio web. [17]

### 2.2.4. Diseño adaptativo

Por otro lado, el diseño adaptativo tiene múltiples tamaños de diseño fijos. Cuando el sitio detecta el espacio disponible, selecciona el diseño más apropiado para la pantalla. Por ejemplo, al abrir un navegador, el sitio escoge el mejor diseño para esa pantalla, por lo que reajustar el navegador no tiene impacto en el diseño. [15]

Este diseño genera diversos templates (plantillas), los cuales son optimizados y únicos para cada clase de dispositivo. A modo de ejemplo, se tiene un diseño para las computadoras, otro para las tablets y otro para los smartphones.

La idea detrás del diseño adaptativo web es crear la mejor experiencia para los usuarios, sin importar el navegador o el dispositivo que se esté usando. En vez de utilizar un solo diseño para todo, este diseño tiene un enfoque particionado. Por ejemplo, a los usuarios en un computador se les muestra un sitio web enteramente, mientras que los que utilizan un dispositivo móvil ven la versión móvil de dicho sitio. Esto permite separar el diseño web del móvil, pudiendo agregar características diferentes según la preferencia del usuario al utilizar los distintos tipos de dispositivos. [18]

## 2.3. Atributos de calidad de la UI

### 2.3.1. Adaptabilidad

La adaptabilidad es la capacidad del sistema de poder adaptarse de manera eficiente al cambio. [19]

Como menciona Andrés Mauro en su tesis, “en el caso de la UI, la adaptabilidad está ligada al hecho de poder adaptarse a, por ejemplo:

- diferentes tipos de pantalla: smartphones, tablets, laptops, monitores, smartwatches, smart TV's. Cada uno de ellos presenta a su vez diferentes resoluciones de pantalla como UHD, FHD, HD, etc. También están presentes las relaciones de aspecto como 4:3 o 16:9 que indican la proporción de píxeles que hay entre ambos ejes y que luego repercuten en la densidad de píxeles.

- plataformas: Android, iOS, Linux, macOS, Windows, entre otras.

- contexto: puede afectar a la UI en aspectos visuales, auditivos y táctiles. Un ejemplo de afectación en aspectos visuales es el caso de los temas. Los temas claros se visualizan mejor en contextos de altas condiciones lumínicas como son la calle u otros espacios abiertos a plena luz del sol. Los temas oscuros se visualizan mejor en contextos de bajas condiciones lumínicas como son la casa, la oficina u otros espacios cerrados. Además, los temas oscuros reducen la fatiga visual en condiciones de oscuridad total, como el caso del uso de smartphones en la cama.

El contexto también afecta la UI en aspectos auditivos cuando nos encontramos realizando alguna otra acción que requiera de nuestra atención visual. Un ejemplo de esto es cuando manejamos. Al manejar se vuelve peligroso prestar atención a una pantalla, sin embargo, interactuar con el sistema mediante comandos por voz se vuelve muy práctico". [20]

### 2.3.2.Portabilidad

La portabilidad se refiere a la facilidad con la cual un software que fue construido para correr en una plataforma pueda ser cambiado para correr en otra diferente. [21]

A su vez, este atributo se subdivide en otras características:

- Adaptabilidad, referido a la capacidad del producto que le permite ser adaptado de forma efectiva y eficiente a diferentes entornos determinados de hardware, software, operacionales o de uso.

- Capacidad para ser instalado, es decir, la facilidad con la que el producto se puede instalar y/o desinstalar de forma exitosa en un determinado entorno.

- Capacidad para ser reemplazado, esto es, la capacidad del producto para ser utilizado en lugar de otro producto software determinado con el mismo propósito y en el mismo entorno. [22]

Si llevamos la portabilidad a las distintas plataformas, podemos notar ciertos desafíos que se presentan con respecto a este atributo de calidad. Por un lado, la parte web tiene un problema de portabilidad que es el navegador, incluyendo el engine de JavaScript y la renderización, entre otros. Por este motivo, hay que contemplar distintos navegadores y versiones.

Por otro lado, la parte móvil tiene distintos sistemas operativos, entre los que se destacan Android y iOS, pero estas familias a su vez tienen fragmentación interna de versiones. Esto es, para cada versión, hay que tener consideraciones según lo que pueden y no pueden soportar. Cuanto más nueva sea la versión del sistema operativo, menor dispositivos móviles podrán soportarlo, lo cual conlleva un gran desafío.

Debido a lo anterior, la interfaz de usuario se ve afectada por la portabilidad, la cual implica que el software pueda ser ejecutado en diversas plataformas. Esto significa considerar los diferentes casos para poder brindar una buena interfaz, que a la vez cumpla con las características de la plataforma en la que se muestra y con estándares de calidad.

#### 2.3.2.1. Desarrollo nativo

Una aplicación nativa está diseñada para ejecutarse en un sistema operativo móvil específico. Las mayores ventajas para este tipo de aplicaciones son que normalmente pueden acceder fácilmente a todas las funciones del dispositivo elegido y ejecutarse sin errores, si son desarrolladas correctamente. Sin embargo, esto viene con un trade-off. Una aplicación nativa no puede ejecutarse en un dispositivo que no utilice su mismo sistema operativo. Por ello, si se deseara ejecutar dicha

aplicación en varios sistemas operativos, se la tendría que desarrollar para cada uno de ellos, resultando en un proceso de desarrollo más lento y costoso. [17]

Si tomamos el ejemplo con Android e iOS, notamos que esto presenta un desafío; mantener una aplicación para ambas plataformas en paralelo. Esto es debido a que las mismas presentan diferencias en su código fuente, librerías e IDEs utilizados para su desarrollo. Incluso la tipografía, métodos de navegación y la ubicación de ciertos elementos de la interfaz de usuario es distinta. Sin embargo, en muchos casos, este problema es resuelto al asignar dos equipos de desarrollo que trabajen en paralelo, cada uno en una plataforma.

### 2.3.3. Mantenibilidad

La mantenibilidad se refiere al grado de efectividad y eficiencia con el que un sistema puede ser mantenido. [21]

Este concepto de alto nivel también aplica a la UI y está vinculado al usuario. El cambio es inevitable cuando se trata de UI; los usuarios pueden probar una aplicación y descubrir defectos o el cliente puede cambiar de opinión respecto a ciertas características que se muestran. En la mayoría de los casos, cuando se agregan o quitan funcionalidades, la UI se ve afectada. Por esto, a pesar de que se busca que sean atractivas visualmente, también es importante tener en cuenta la mantenibilidad al momento de realizar el código de una interfaz de usuario, pudiendo realizar cambios rápidamente sin romper la aplicación. No seguir buenas prácticas de codificación puede decrementar drásticamente la mantenibilidad de la implementación de la UI.

### 2.3.4. Usabilidad

La usabilidad se refiere a cuan fácil es para el usuario realizar una tarea deseada, junto con el tipo de soporte que le provee el sistema. [21]

Es un atributo de calidad que evalúa la facilidad de uso de las interfaces de usuario. La misma se define por 5 componentes de calidad:

- Capacidad de aprendizaje, es decir, ¿qué tan fácil es para los usuarios realizar tareas básicas la primera vez que encuentran el diseño?

- Eficiencia, refiriéndose a que, una vez que los usuarios han aprendido el diseño, ¿con qué rapidez pueden realizar las tareas?

- Memorabilidad, o sea, cuando los usuarios regresan al diseño después de un período sin usarlo, ¿con qué facilidad pueden restablecer la habilidad de uso?

- Errores, es decir, ¿cuántos errores cometen los usuarios? ¿qué tan graves son? ¿con qué facilidad pueden recuperarse de ellos?

- Satisfacción, refiriéndose a qué tan agradable es para el usuario usar el diseño.

La usabilidad es uno de los atributos más relevantes cuanto se trata de interfaces de usuario, ya que, si una aplicación o sitio web es difícil de usar o no es intuitivo y genera confusión, los usuarios no se sentirán atraídos a ella y no la utilizarán. [23]

## 2.4. Sistemas de diseño

Los sistemas de diseño podrían verse como un conjunto de directrices basadas en patrones y buenas prácticas del desarrollo y diseño, cuidadosamente organizadas y documentadas, que buscan gobernar el desarrollo de la UI de un producto digital. [20]

Son una estrategia de alto nivel, que también afectan la adaptabilidad y portabilidad. Un sistema de diseño unificado es esencial para la construcción de mejores productos rápidamente y para la experiencia de usuario. Ayudan a mantener la consistencia del producto, ya que todos los elementos mantienen coherencia entre ellos. Asimismo, sirven para mantener una guía de elementos reusables y unificar estilos, reduciendo la probabilidad de errores al realizar cambios en el diseño. Ejemplos de sistemas de diseño son Material Design (Google), Human Interface Guidelines (Apple), Fluent Design (Microsoft), Carbon Design (IBM), entre otros. [24]

Si consideramos el caso de Apple y Google, podemos notar que hay una diferencia en sus objetivos. Mientras que el sistema de diseño del primero se centra en aplicaciones que se integran con plataformas de Apple, Material Design es multiplataforma, pudiendo utilizarlo para aplicaciones de Android, iOS y la web. Por ser multiplataforma, permite que se desarrolle sobre las mismas bases para distintas plataformas y tamaños de pantalla.

# 3. Planteamiento del problema

Cuando hablamos de construcción de software, uno de los atributos que presenta un desafío arquitectónico y de alto nivel es la adaptabilidad.

Así como los atributos de calidad de la UI a veces son primordiales, estamos desarrollando en un mundo multiplataforma y multi-pantalla, donde se priorizan las preferencias de los usuarios. Donde el software y la UI tienen que ser adaptables a muchos contextos, a muchos tipos de preferencia, en un mundo que está cambiando las estrategias para desarrollar UI y las tecnologías. Donde los usuarios son diversas personas, o donde poseemos sistemas que tienen alcance para toda la población en todo momento y lugar.

## 3.1. Trade-offs de adaptabilidad

El problema de la adaptabilidad y de la multiplataforma es abierto, complejo y cambiante; las dimensiones de la adaptabilidad se siguen multiplicando, lo que implica ciertos trade-offs, como todas las decisiones arquitectónicas (seguridad, usabilidad, accesibilidad, funcionalidad, etc.) y en particular la portabilidad y la adaptabilidad obligan a tomar decisiones que sacrifican algunos aspectos, incluyendo la propia UI.

No se puede hacer todo para todas las plataformas de manera consistente y pareja, utilizando las características nativas de cada una, sino que a veces se tiene que renunciar a alguna función o prestación para poder lograr que una aplicación sea más adaptable. Por ejemplo, no se puede poner un elemento en un cierto lugar fijo de la pantalla ya que probablemente no se va a visualizar de forma correcta en todos los dispositivos.

## 3.2. Fragmentación de plataformas

Tomando el caso de Android e iOS, si hablamos de la distribución de las versiones de API, podemos decir que hay una diferencia entre la fragmentación interna de ambas plataformas bastante notable, incluso al día de hoy. Empezando por Android, basándonos en los datos proporcionados por Android Studio, existe una gran diferencia en cuanto al porcentaje de aplicaciones que

podrían ser soportadas según la versión de la plataforma. Por ejemplo, para la versión 4.4 (KitKat) lanzada en 2013, se tiene un 98.1%, mientras que para la versión 10.0, lanzada en 2019, el porcentaje es del 8.2%, notándose un decremento importante. Esto refleja la cantidad masiva de versiones de Android disponibles hoy en día y presenta un gran problema debido a la alta fragmentación. La distribución de uso de las versiones existentes es muy variada y la mayoría de los usuarios no tienen instalada la última versión de la plataforma en sus dispositivos. Una de las consecuencias que esto provoca es la vulnerabilidad a ataques maliciosos. A medida que se lanzan nuevas versiones, los dispositivos con versiones más viejas se vuelven vulnerables, ya que van dejando de ser compatibles con actualizaciones de seguridad y protección incorporada. [25]

Por otro lado, en el caso de iOS, a diferencia de Android, la fragmentación no es un problema tan grande, ya que Apple trata de mantener sus niveles de fragmentación al mínimo. Esto es, asegurarse de que los usuarios actualicen a la última versión lo antes posible, brindándoles actualizaciones fáciles y oportunas. Sin embargo, no todos lo hacen a la misma vez, e incluso hay versiones que están disponibles solo para ciertos dispositivos premium de la plataforma. Además, actualizar a la última versión no siempre ofrece la mejor experiencia de usuario. En cuanto a porcentajes, por ejemplo, para la versión de iOS 12.4, lanzada en 2019, el uso es del 7.33% mientras que para la versión 14.2, lanzada en 2020, es del 32.07%. [26]

## 3.3. Costo de desarrollo y mantenimiento

Otro punto a considerar es el costo de desarrollo sobre una base de código para que la UI sea multiplataforma. Si se tiene que mantener una base de código distinta para iOS, Android y la web y a su vez, para las transformaciones que éstas puedan tener (ya que las plataformas evolucionan con el paso del tiempo) se tiene un costo muy grande, aunque se cuente con expertos en todas ellas.

En resumen, los aspectos que comprenden el problema de la adaptabilidad y la multiplataforma son varios; el cambio tecnológico, la fragmentación de las plataformas, los desafíos del diseño responsive, lo cual se deriva en su mayoría

del usuario y la contextualización. Hoy en día, existen varias tecnologías que se enfrentan a dicho problema, permitiendo que con una sola base de código se puedan crear aplicaciones multiplataforma compatibles con distintos sistemas de diseño que buscan garantizar la adaptabilidad en distintas plataformas si se respetan.

# 4. Respuesta tecnológica

Para resolver el problema de la adaptabilidad de la interfaz de usuario en distintas plataformas, podemos considerar dos estrategias arquitectónicas. Éstas son el desarrollo nativo y los sitios web adaptables, permitiendo este último ser multiplataforma.

Hoy en día, existen varias tecnologías que permiten generar, en base a un único código fuente, aplicaciones nativas para las dos mayores plataformas móviles existentes en el mercado (Android e iOS), aplicaciones web adaptables o ambos.

Para nuestro caso de estudio, investigaremos tres tecnologías. Éstas, mediante las estrategias arquitectónicas mencionadas y la utilización de un sistema de diseño, facilitarán el desarrollo de aplicaciones multiplataforma adaptables.

## 4.1. React.js

### 4.1.1. Definición de React.js

React es una biblioteca de JavaScript para construir interfaces de usuario. Sus características incluyen el ser declarativo, es decir, ayuda a crear interfaces de usuario interactivas de forma sencilla. Esto se debe a que se encarga de actualizar y renderizar de manera eficiente los componentes correctos cuando los datos de una aplicación cambian. Las vistas declarativas hacen que el código sea más predecible, por lo tanto, fácil de depurar. También está basado en componentes; se pueden crear componentes encapsulados que manejen su propio estado y convertirlos en interfaces de usuario complejas. Debido a que la lógica de los componentes está escrita en JavaScript y no en plantillas, se pueden pasar datos de forma sencilla a través de una aplicación y mantener el estado fuera del DOM.

Además, React permite desarrollar nuevas características sin necesidad de volver a escribir el código existente, y también puede renderizar desde el servidor usando Node, así como potencializar aplicaciones móviles usando React Native. [27]

### 4.1.2. Definición de Material-ui

Específicamente para React.js, nos encontramos con Material-ui, un framework de terceros que permite la fácil implementación de los principales componentes de material.io en React.js. También permite la de algunos otros provistos por ellos, muy utilizados por personas o empresas que desarrollan en React.js. [28]

Un dato interesante para destacar es la cantidad de empresas grandes que utilizan material-ui (Netflix, Spotify, NASA, entre otras).

Material-ui promete tener escasos o ningún archivo .css en nuestros proyectos, embebiendo el estilo en código JavaScript, accesible por los componentes. También nos permite aplicar a todos nuestros componentes el tema de la aplicación de manera fácil. Mediante el uso de la paleta de colores que ofrece material.io, se insertan los colores primarios, light y dark, los cuales podrán ser utilizados desde cualquier componente. [29]

### 4.1.3. Arquitectura de una aplicación React.js

Existen buenas prácticas a seguir a la hora de desarrollar una aplicación utilizando React.js, pero no existe una arquitectura como tal a seguir estrictamente.

Para poder entender su arquitectura básica es necesario tener un conocimiento básico sobre JavaScript, HTML y CSS, previo a desarrollar una aplicación web utilizando la librería de React.js. Esto se debe a que durante todo el desarrollo, se deben poseer ciertos conceptos para poder entender el funcionamiento a bajo nivel y luego, sobre esa base, comenzar a desarrollar.

React.js es una librería y un framework de NodeJS. Las aplicaciones web en React.js se basan en el concepto de SPA (Single Page Application). Por esto, al generar un proyecto se crea la carpeta “src”. En ella, se encuentra todo el código JavaScript y estilos, mientras que la carpeta “public” contiene el único archivo con extensión .html de todo el proyecto. En dicha carpeta “src” se hará la división de paquetes (carpetas) dentro del proyecto.

A la hora de seguir una arquitectura para crear una aplicación en React.js, existen varios boilerplate recomendados. Éstos son repositorios vacíos para implementar, que contienen la división en carpetas, librerías y componentes que se utilizan frecuentemente a la hora de crear una aplicación en un determinado framework. En nuestro caso, basándonos en varios ejemplos publicados en la red, decidimos seguir un estilo arquitectónico que definiremos a continuación, donde cada carpeta puede ser tomada como un paquete que contiene funcionalidades dentro.

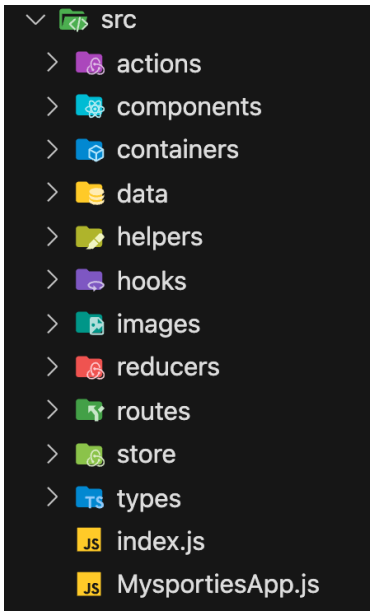



Imagen 2 - División de carpetas del proyecto en React.js

En la raíz de la carpeta, se encuentran los archivos base de la aplicación, los cuales serán ejecutados cuando se inicialice el proyecto. Luego, es importante destacar la carpeta “route”, que será donde registraremos todas las navegaciones de la aplicación. Hablaremos de ella en detalle más adelante.

La carpeta “store” es donde se inicializará el provider de la aplicación. Éste, utilizando Redux, permitirá a los demás componentes de la aplicación enviar eventos y leer datos, que serán almacenados en el contexto de la misma. Dicho contexto utilizará los reducers definidos en el paquete “reducers”. Estos son funciones reductoras que le establecen al provider cuando y como actualizar los estados almacenados en el contexto de la aplicación. Por

último, tenemos la carpeta “actions”, que refleja las acciones que deben tomar los reducers. Dichas acciones serán llamadas a la hora de querer modificar el estado del contexto de la aplicación.

La carpeta “routes” contiene las funciones que definirán como se manejarán las rutas dentro de la aplicación. Esto incluye cuando se mostrará el Drawer o el BottomNavegation, además de que componentes se visualizarán al mostrarlos.

La carpeta “components” busca contener las funciones que denotan diseño. El funcionamiento se los dará mediante la utilización de las acciones del contexto y las funciones que se encuentran dentro de la carpeta “containers”. Esta última incluye contenedores que además de tener diseño, contienen comportamiento y utilizan varios de los contenedores comentados anteriormente. Esto con el fin de encapsular cierto diseño, reutilizando componentes y evitando el código duplicado.

El resto de las carpetas fueron creadas debido al caso de estudio y no siguen ninguna arquitectura en particular.

### 4.1.4. Estructura y componentes

React.js nos permite crear interfaces de usuario interactivas de forma sencilla, encargándose de actualizar y renderizar de manera eficiente los componentes correctos cuando éstos cambian. [27]

Los componentes son lógica escrita en JavaScript, los cuales manejan su propio estado, convirtiéndose en interfaces complejas. Además, permiten pasar datos de forma sencilla entre ellos, manteniendo el estado fuera del DOM.

Un componente está compuesto por código HTML y JavaScript, en su mayoría. Luego, en él se puede hacer referencia a una hoja de estilos CSS, pero también se le puede dar estilos dinámicamente a través de JavaScript. Además, React.js permite devolver componentes que, a su vez, pueden

contener otros componentes dentro, los cuales se muestran según nuestras necesidades y el comportamiento que establezcamos.

A partir de esto, surgen los siguientes problemas:

- ¿Cómo elegimos que componentes mostrar?
- ¿Cómo compartimos información entre componentes?
- ¿Cómo organizamos el estilo que le damos a dichos componentes?

Para contestar las dos primeras preguntas no nos explayaremos mucho, ya que no son parte importante de nuestro caso de estudio.

Mediante el uso de Redux, creamos una store que se aplica a todos los componentes (ya que envuelve el componente principal) brindándoles información. Desde cada uno de ellos, podemos acceder a ciertos datos que mantenemos en el estado de la aplicación. Esto sin importar en que componente nos encontremos, usando los Hooks recomendados.

Luego, para elegir que componente mostrar, creamos las rutas utilizando la librería react-router-dom. Esta nos permite declarar que componente se muestra dependiendo del path en que nos encontremos.

## 4.2. React Native

### 4.2.1. Definición de React Native

React Native combina las mejores partes del desarrollo nativo con React, pudiendo usarlo en proyectos existentes de Android y iOS o creando una nueva aplicación desde cero. Está escrito en JavaScript y renderizado con código nativo. Es decir, permite crear versiones de componentes específicas de la plataforma para que una única base de código pueda compartir código entre plataformas. Además, permite crear aplicaciones verdaderamente nativas, sin comprometer las experiencias de los usuarios. Los componentes de React envuelven el código nativo existente e interactúan con las API nativas a través del paradigma de UI declarativo de React y JavaScript, permitiendo que se trabaje más rápido. También cuenta con actualizaciones rápidas, pudiendo ver los cambios tan pronto como se guardan. [30]

### 4.2.2. Definición de React Native CLI y Expo

Actualmente, existen dos maneras para construir aplicaciones en React Native; React Native CLI o React Native Expo CLI.

La primera opción provee el comando “react-native init”, lo cual genera un proyecto con lo mínimo necesario para correr, con proyectos nativos separados para las plataformas de Android e iOS. Se pueden modificar los proyectos nativos como se desee, escribiendo código nativo para cada plataforma. Sin embargo, una desventaja de utilizar esta opción es que se recomienda para aquellos que estén familiarizados con el desarrollo móvil. Esto es debido a que se requiere de entornos de desarrollo como Xcode (iOS) o Android Studio (Android) para que funcione. [31] [32]

Por otro lado, Expo CLI es un proyecto gratis y open source que provee un conjunto de herramientas de programación construido a partir de React Native. Provee muchas funcionalidades nativas incluidas, lo que ayuda a construir aplicaciones nativas en iOS y Android de manera más rápida usando únicamente JavaScript y React. Como ventaja, esta opción es adecuada para aquellos que son nuevos en cuanto al desarrollo móvil. Esto se debe a que Expo provee una cadena de compilación incorporada, encargándose de controlar los builds para iOS y Android. Dicha cadena funciona bien para compilar y testear en dispositivos de ambas plataformas. Además, permite correr y probar la aplicación en iOS, aunque no se disponga de una Mac. [32]

Sin embargo, en el caso de querer integrar una librería nativa de terceros o extender funcionalidades de la aplicación con código nativo, esto no es posible con Expo CLI. Los proyectos de Expo no revelan los proyectos nativos de iOS y Android, a diferencia de React Native CLI, que sí lo hace. [32]

Debido a que es muy importante hoy en día poder tener al alcance los proyectos nativos generados por estas tecnologías, basaremos nuestra investigación en React Native CLI.

### 4.2.3. Definición de React Native Paper

React Native Paper es una librería que cuenta con una colección de componentes customizables y listos para producción para React Native, basada en las pautas de Material Design. También puede decirse que es una cross-platform de Material Design para React Native, es decir, puede funcionar en múltiples tipos de plataformas o sistemas operativos. [26]

Algunas de sus características son que es open source y su complejidad, en cuanto a la creación de componentes, es menor comparada con el código estándar de React Native. Además, ofrece un soporte completo en cuanto a temas y estilos, como cambiar entre modos oscuro y claro, customización de colores por defecto, entre otros. Sin embargo, cabe destacar que no es reconocido por material.io, por lo que no es un recurso oficial, aunque es el más grande y conocido para React Native. [33]

### 4.2.4. Arquitectura de una aplicación React Native

Debido a que React Native es una extensión de React.js y que ambos utilizan JavaScript como lenguaje base, decidimos basar la investigación en esta tecnología en TypeScript. Esto con el fin de poder analizar distintas estructuras de estas tecnologías. TypeScript es un superconjunto de JavaScript, centrado en, a diferencia de este último, el tipado de las variables y objetos.

Al generar una aplicación en React Native CLI, compatible con TypeScript, se generan varias carpetas y archivos. Las primeras de las que hablaremos serán las de código nativo, “ios” y “android”. Estas contienen proyectos en lenguaje nativo de cada plataforma que equivalen a lo desarrollado en React Native. Esto nos permitirá instalar dependencias o configurar un comportamiento en específico para cierta plataforma, en el caso de no poder realizarse directamente con React Native.

Luego se generan otros archivos que configuran otros aspectos de la aplicación. Los más importantes son el “index.js”, que, al igual que en

React.js, será el inicio de la aplicación y el “App.tsx”, que, a diferencia de React.js, utiliza TypeScript y será la base de nuestra aplicación.

React Native no tiene una arquitectura predeterminada con la cual cumplir. Sin embargo, existen varios boilerplate recomendados a la hora de seguir una estructura para crear una aplicación en React Native. Lo mismo ocurre para React.js. En nuestro caso, basándonos en varios ejemplos publicados en la red, decidimos seguir un estilo arquitectónico que definiremos a continuación. Cada carpeta puede ser tomada como un paquete que contiene funcionalidades dentro.

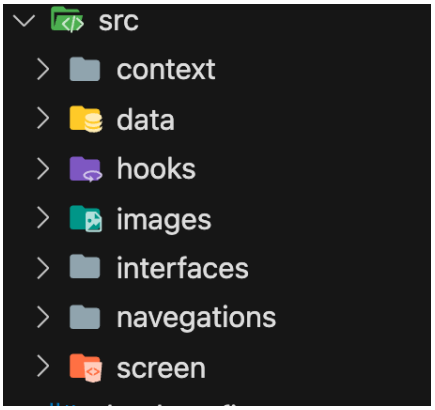



Imagen 3 - División de carpetas del proyecto en React Native

Siguiendo los ejemplos, pondremos todas las carpetas dentro de la carpeta “src”. Dichas carpetas representarán los paquetes que tendrá dicha aplicación.

La carpeta “screen” contiene las pantallas de la aplicación. En la carpeta “navegations” podremos encontrar los distintos tipos de navegación que tendrá la aplicación y que pantalla mostrará en cada caso.
La carpeta “context” contiene el contexto de la aplicación, en el cual se manejarán los datos y las acciones que se podrán realizar a los mismos.

El resto de las carpetas fueron creadas debido al caso de estudio y no siguen ninguna arquitectura en particular.

### 4.2.5. Estructura y componentes

La estructura de React Native, al igual que la de React.js, es en capas. Estas capas están compuestas por Views que contienen componentes específicos.

React Native proporciona componentes nativos como pueden ser las View, Text, Image, que corresponden a bloques de código de interfaz de usuario específica para una plataforma nativa. Esto permite que los componentes de React Native sean renderizados a cierta plataforma específica.

Al igual que en React.js, podemos crear componentes personalizados que podrán tener un comportamiento y un diseño según los deseos del programador. También existen ciertos componentes que solo están disponibles para una determinada plataforma en particular. De esta manera resultan no compatibles o compatibles parcialmente con otras plataformas.

## 4.3. Flutter

### 4.3.1. Definición de Flutter

Flutter es un kit de herramientas de interfaz de usuario (UI) multiplataforma de Google para la construcción de aplicaciones, compiladas nativamente para mobile, web y escritorio, a partir de una sola base de código. Su objetivo es permitir que los desarrolladores entreguen aplicaciones de alto rendimiento que se sientan naturales en diferentes plataformas. Esto adoptando las diferencias en donde existan, mientras comparten la mayor cantidad de código posible. [34]

Flutter utiliza el lenguaje Dart, el cual también soporta tareas centrales de desarrollador, como formatear, analizar y testear código. Es un lenguaje optimizado al cliente para desarrollar aplicaciones rápidas en cualquier plataforma. [35]

### 4.3.2. Material dentro de Flutter

Flutter cuenta con widgets visuales, de comportamiento y ricos en movimiento, implementando las pautas de Material Design. Algunos ejemplos de los que brinda están relacionados a la estructura de la aplicación y navegación, botones, inputs y selecciones, diálogos, alertas y paneles, layouts, entre otros. [34]

### 4.3.3. Arquitectura de una aplicación Flutter

Al crear un proyecto en Flutter, por defecto ya vienen ciertas carpetas, mientras que el código fuente debería ir en la carpeta “lib”. En este caso, existen boilerplate que ayudan a la distribución de componentes internamente. Sin embargo, estas divisiones son útiles al tener proyectos más complejos, donde se tenga comunicación con servicios o componentes más sofisticados.

Por todo lo detallado anteriormente, solo explicaremos la estructura básica de un proyecto en Flutter. El mismo cuenta con dos carpetas de código nativo, “ios” y “android”. Estas contienen proyectos en lenguaje nativo de cada plataforma que equivalen a lo desarrollado en Flutter. Luego, como comentamos, se encuentra la carpeta “lib”. Ella contiene el “main.dart” que es el punto de partida de la aplicación hecha en Flutter, además del resto de archivos con las distintas funcionalidades.

### 4.3.4. Estructura y componentes

Flutter es un framework desarrollado en el lenguaje de programación Dart, que es un lenguaje que permite crear aplicaciones multiplataforma. Flutter, al igual que React.js y React Native, está organizado en una serie de capas.

Estas capas, son los llamados widgets, los cuales se inspiran en los componentes de React.js. Estos describen como debería comportarse y verse la interfaz de usuario. Al igual que en React.js, Flutter provee al desarrollador con varios de estos widgets como el Text, Icon, entre otros, además de los

creados por terceros. A su vez, el desarrollador puede crear sus widgets personalizados como crea sus componentes en React.js.

Estos se dividen en Stateless y Statefull. Los Stateless son los widgets que no tienen un estado mutable, es decir, son inmutables. No pueden cambiar su estado en tiempo de ejecución de la aplicación, por lo que, no se pueden volver a dibujar cuando ocurre un cambio. Un ejemplo de un widget stateless provisto por Flutter es el Text.

Los Statefull en cambio, tienen un estado mutable, por lo que, se pueden redibujar varias veces durante su ciclo de vida. Un ejemplo de un widget statefull provisto por Flutter es el TextField. [34]

# 5. Desarrollo del prototipo

## 5.1. Propósito del sistema

Con el fin de poner a prueba las tecnologías y poder sacar conclusiones a nuestro problema, creamos una aplicación simple. La función de ésta es poder agendarse a ciertas actividades de un club deportivo, dejando registro de esta agenda para que el usuario pueda consultarla.

Dicha aplicación fue realizada en React.js, React Native y Flutter y utilizando el sistema de diseño Material Design.

### 5.1.1. Objetivos

Teniendo como objetivo la exploración de la interfaz de usuario y sus elementos, según cada una de las tecnologías aplicadas, pretendemos enfocarnos en los diversos componentes. Esto es, campos de texto, selectores de fecha y hora, grid, barras de progreso y validaciones. A partir de esto, analizaremos como funciona cada uno de ellos, para luego realizar una comparación y efectuar conclusiones basándonos en los resultados obtenidos.

### 5.1.2. Requerimientos funcionales

En cuanto a los requerimientos funcionales, la aplicación deberá contar con dos pantallas.

Una contendrá una lista de las actividades ofrecidas por el club deportivo, de las cuales se podrá escoger para agendarse. Luego de seleccionar una actividad, se pedirá nombre y cédula del usuario, junto con la hora y fecha en que desea registrarse.

La otra pantalla mostrará la agenda del usuario, es decir, la lista de actividades a las cuales se registró, junto con el día y hora correspondiente.

Se contará con una barra de menú, ya sea lateral o inferior, pudiendo seleccionar de las dos opciones disponibles (agendarse o ver agenda).

Además, la aplicación deberá ser adaptativa, adaptándose así al tamaño de la ventana o pantalla en que es ejecutada.

### 5.1.3. Bocetos del prototipo

Como primer paso, realizamos un esquema para tener como guía a la hora de comenzar con el desarrollo y así, poder tener uniformidad y consistencia en las distintas plataformas.

Cuando la orientación del dispositivo es vertical (portrait) la aplicación se verá como se muestra para móviles (imagen 4). Cuando es horizontal (landscape) tomará la forma como la que se visualiza para la web (imagen 5, 6 y 7). A continuación, brindamos los bocetos realizados para ambos casos.

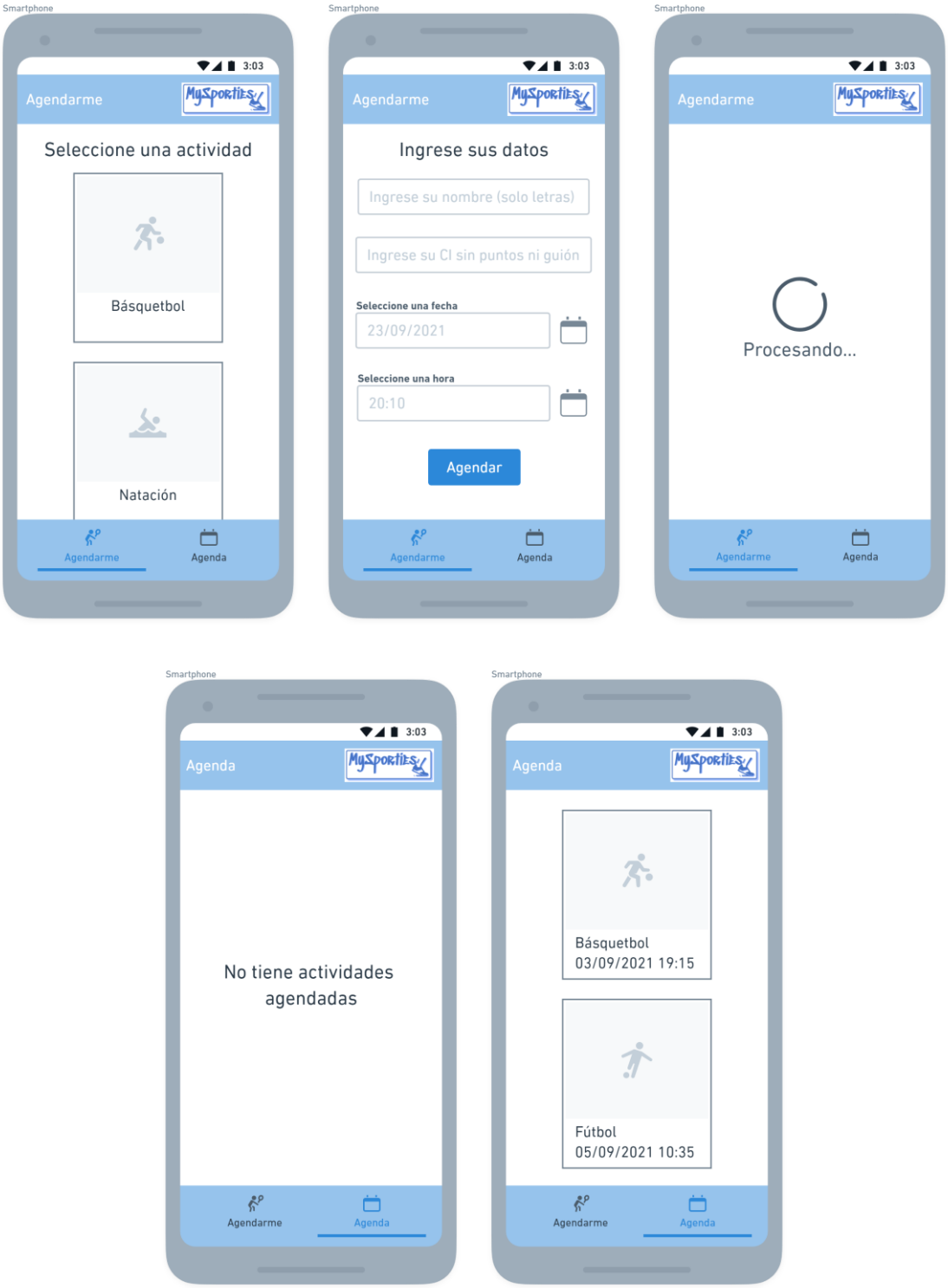



Imagen 4 - Orientación portrait

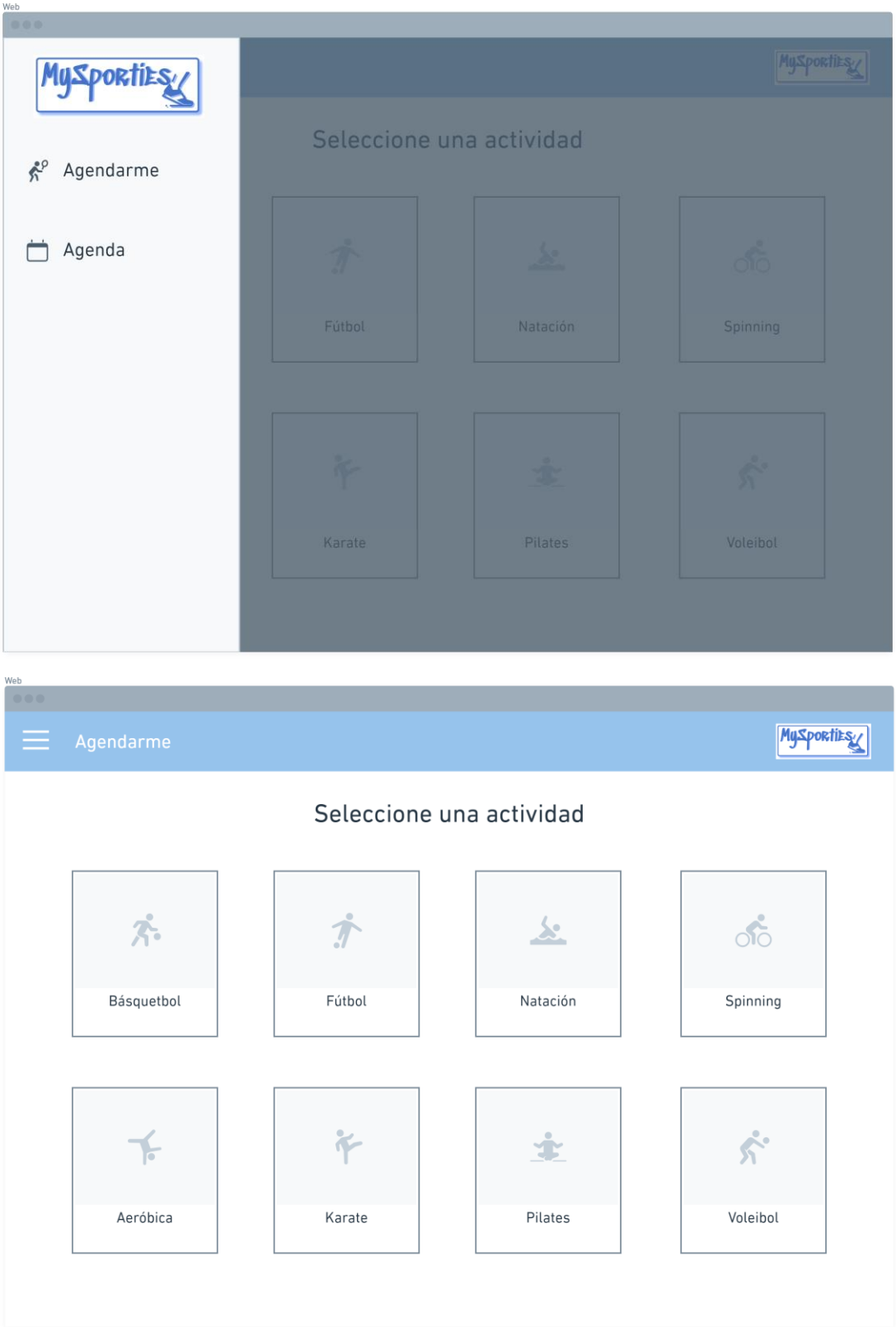



Imagen 5 - Orientación landscape

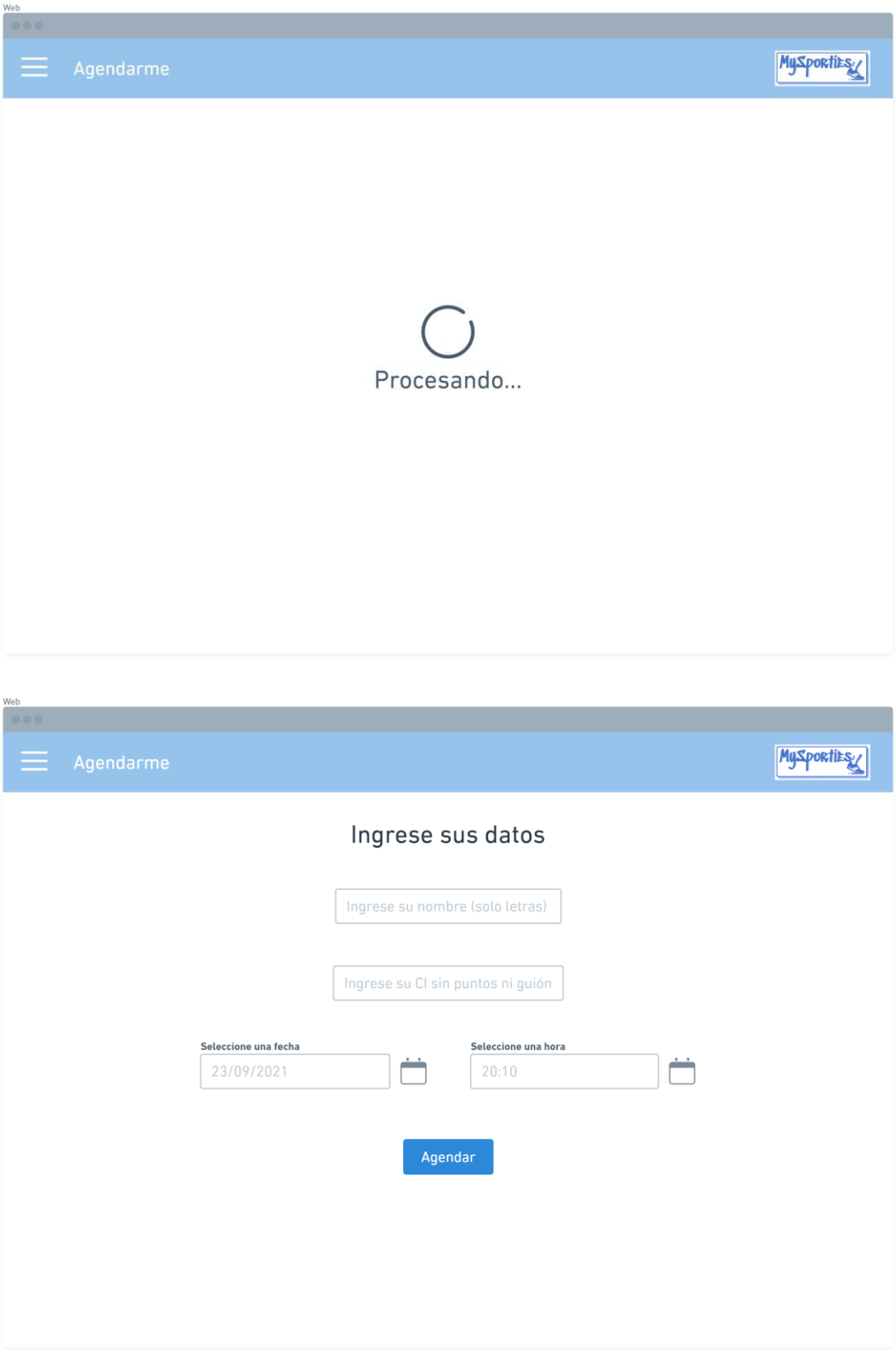



Imagen 6 - Orientación landscape

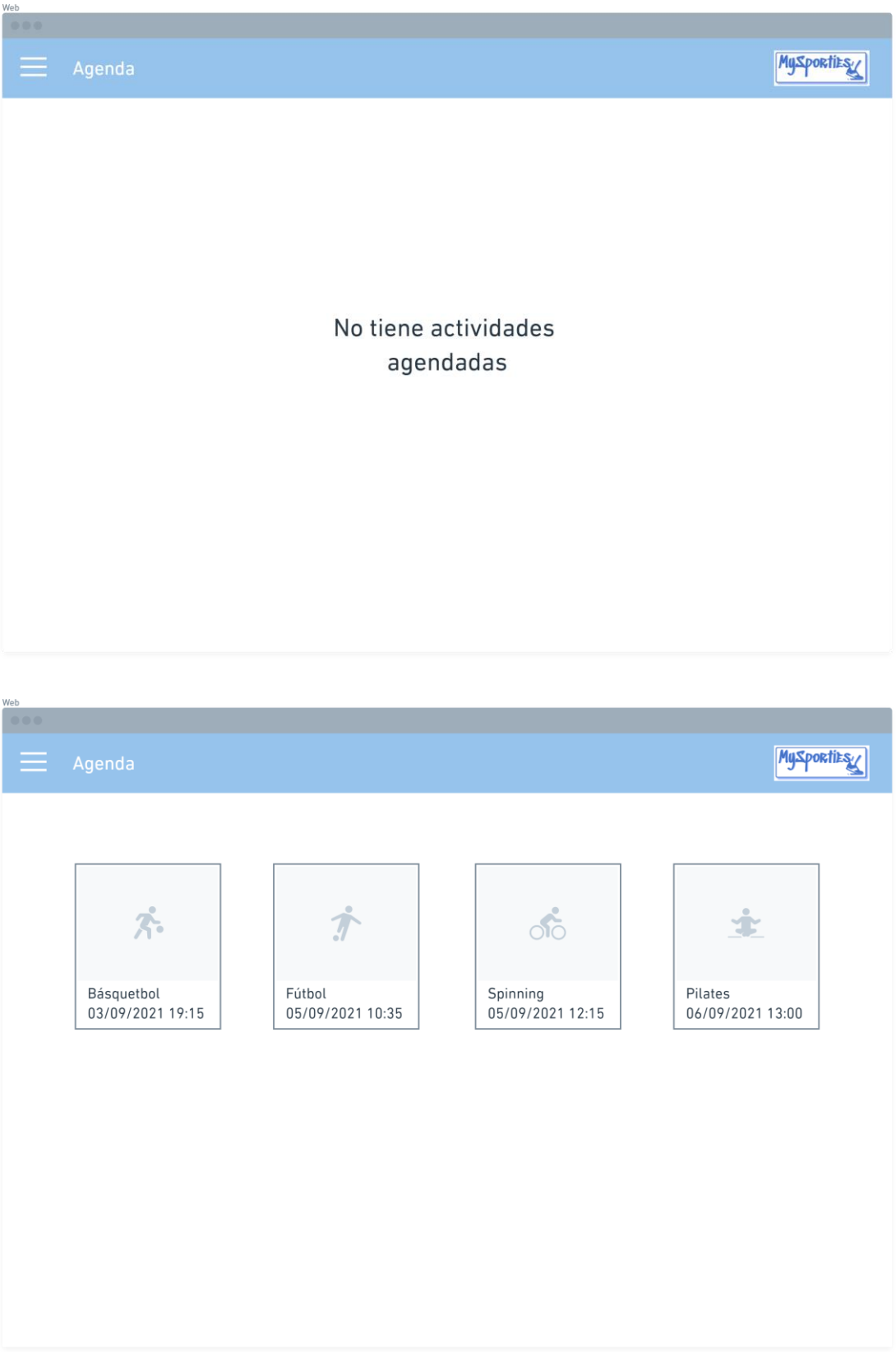



Imagen 7 - Orientación landscape

### 5.1.4. Paleta de colores y tipografía

Utilizando la paleta de colores que ofrece material.io, se insertan los colores primarios, light y dark, los cuales, podrán ser utilizados desde cualquier componente.

En nuestro caso, decidimos escoger una paleta de colores basados en azules, como se puede apreciar en la imagen 8, ya que este color es usualmente asociado con confianza, tranquilidad y orden. [29]


Imagen 8

Material Design recomienda, en el caso de ser necesario, la selección de una paleta de colores secundarios. Para este estudio de caso, no será utilizada, dada la baja complejidad de la aplicación.

En cuanto a la tipografía, decidimos utilizar la fuente Roboto, ya que está avalada por Material Design y es considerada el tipo de letra estándar. Además, dicha fuente ha sido refinada extensivamente para poder funcionar en un conjunto más amplio de plataformas soportadas. Es un poco más ancha y redonda, lo que hace que sea más clara y optimista.

## 5.2. Resultados

A lo largo de esta sección se detallan los resultados obtenidos en las distintas tecnologías. Asimismo, se extraerán fragmentos de código relevantes y capturas del prototipo, que permitirán comprender los resultados obtenidos. Es importante aclarar que los fragmentos de código fueron extraídos de las pruebas de concepto realizadas para cada una de estas tecnologías. Estos únicamente representan una forma de lograr los objetivos de diseño establecidos.

Para ver el código completo, el enlace al repositorio de React.js es https://github.com/SantiagoTonarelli/LIS-reactjs, el enlace al repositorio de React Native es https://github.com/SantiagoTonarelli/MySportiesRN y el enlace al repositorio de Flutter es https://github.com/SantiagoTonarelli/MySportiesFlutter.

### 5.2.1. Estilos

#### 5.2.1.1. Estilos en React.js

Debido a que en esta prueba de concepto se decidió utilizar material.io como sistema de diseño, nos propusimos entender cómo adaptarlo a React.js. En la página oficial de material.io, notamos varias recomendaciones para diversos lenguajes, para los cuales no da soporte directamente. Sin embargo, reconoce que existen organizaciones de terceros que brindan los componentes básicos para cumplir con el sistema de diseño propuesto por Material Design. Aunque no garantiza que lo cumplan estrictamente.

Material-ui permite aplicar a todos nuestros componentes el tema de la aplicación de manera fácil.

Para fijar la paleta de colores en nuestra aplicación, creamos el tema dentro de la clase MysportsApp.js. Además, creamos una función que retorna dicho tema en el AppRouter, haciendo posible que cualquier componente pueda usarlo, como se puede observar en el siguiente fragmento de código.

. . .

```
const theme = createMuiTheme({
    palette: {
        primary: {
            main: "#2195f2",
            light: "#6ec5ff",
            dark: "#0068bf",
        },
    },
});
export default function MysportiesApp() {
    return (
        <ThemeProvider theme={theme}>
            <Provider store={store}>
                <AppRouter />
            </Provider>
        </ThemeProvider>
        );
}
```

También, en cada componente, Material-ui recomienda crear sus propios estilos accediendo al tema de la aplicación para asignar. En nuestro caso, los colores, como se puede ver en el siguiente fragmento de código para el componente Home.

```
...
const useStyles = makeStyles((theme) => {
    return {
        root: {
            width: "35ch",
            margin: theme.spacing(2),
        },
        title: {
            margin: theme.spacing(2),
            color: theme.palette.primary.dark,
        },
        error: {
            margin: theme.spacing(6),
            color: theme.palette.primary.dark,
            position: "absolute",
            top: "40%",
        },
        media: {
            height: "20ch",
        },
    };
});
...
```

Para modificar propiedades específicas del estilo predefinido de Material-ui, utilizamos el generador de estilos makeStyles. Dentro de éste, creamos clases para los elementos deseados, pudiendo así cambiar sus propiedades como quisiéramos.

Para que se aplicaran esos estilos, utilizamos el atributo “className” en los componentes, junto con la llamada a la clase correspondiente. Esto es debido a que la palabra class es reservada en React.js, pero a la hora de renderizar el HTML en el navegador, es React el que se encarga de transformarlo en la propiedad CSS “class”.

En el siguiente fragmento de código, se puede ver como se le aplica el estilo personalizado a un tag devuelto por el componente Home.

```
...
return (
        <>
            <Grid
                container
                direction="row"
                justify="center"
                alignItems="center"
            >
                ...
                        <Card className={classes.root}>
                            <CardMedia
                                classNameclasses.media}
                                ...
                            />
                            ...
                        </Card>
                    ...
            </Grid>
        </>
);
```

Gracias a esto, no necesitamos crear ninguna hoja de estilos en nuestro proyecto. Esto se debe a que, al principio, aplicábamos los estilos utilizando SCSS, lo cual permitía mejorar nuestro uso de hojas de estilo. Luego, cuando comenzamos a profundizar respecto al uso de Material-ui, nos dimos cuenta de que no era necesario.

#### 5.2.1.2. Estilos en React Native

##### 5.2.1.2.1. Estilos personalizados

Los estilos personalizados en React Native son aplicables a sus componentes. Estos son creados mediante el uso de StyleSheet, el cual es una abstracción equivalente a las hojas de estilo CSS, pero con objetos JavaScript referenciados por un identificador.

En el siguiente fragmento de código, podemos observar cómo se crea un StyleSheet y como es referenciado en un componente, en este caso, de tipo View.

```
...
return (
    <ScrollView
...
>
      ...
        <View style={styles.loading}>
          …
        </View>
      ...
    </ScrollView>
  );
};
const styles = StyleSheet.create({
  ...
  loading: {
    flex: 1,
    alignItems: 'center',
    justifyContent: 'center',
  },
  ...
});
```

##### 5.2.1.2.2. Estilos con React Native Paper

React Native Paper permite definir, para todos sus componentes, el tema de la aplicación, el cual, en nuestro caso, únicamente definirá los colores a utilizar en la misma.

Como se puede observar en el siguiente fragmento de código, se crea un tema como un objeto JavaScript y se engloban todas las etiquetas de la aplicación en dicho tema, en el archivo raíz.

```
...
const theme = {
  ...
DefaultTheme,
  colors: {
    ...DefaultTheme.colors,
    primary: '#2195f2',
    light: '#6ec5ff',
    dark: '#0068bf',
  },
};
const App = () => {
  return (
    <PaperProvider theme={theme}>
...
    </PaperProvider>
  );
};
```

Luego, este puede ser invocado desde cualquier componente de la aplicación, como se puede ver en el siguiente fragmento de código.

```
...
const {colors} = useTheme();
  ...
  return (
    <View style={styles.center}>
      <FlatList
        ListHeaderComponent={() => (
          <Title
            style={{
              color: colors.primary,
              ...
            }}>
            Seleccione una actividad
          </Title>
        )}
        ...
      />
    </View>
  );
};
...
```

### 5.2.1.3. Estilos en Flutter

En el caso de Flutter no empleamos una hoja de estilos en particular, sino que fuimos aplicando estilos directamente a los componentes según nuestras necesidades. Esto se puede apreciar en el siguiente fragmento de código, donde aplicamos un estilo a un componente de tipo Text.

```
...
Text(
    'AGENDAR',
    style: new TextStyle(
      fontSize: 15.0,
    ),
...
```

Debido a la gran integración que existe entre el sistema de diseño Material Design y Flutter, es posible aplicar el tema de la aplicación sin mayor complejidad en el archivo raíz de la aplicación (main.dart). Dicho tema será aplicado automáticamente en la mayoría de los componentes.

En el siguiente fragmento de código, podemos ver como seleccionamos los colores equivalentes a los especificados por la paleta de material-io. Además, se escoge el tipo de letra a utilizar en toda la aplicación, aplicándose esta configuración automáticamente.

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
        ...
        child: MaterialApp(
          title: 'MySporties',
          theme: ThemeData(
              fontFamily: 'Roboto',
              primarySwatch: Colors.blue,
              primaryColor: Colors.blue[500],
              accentColor: Colors.blue[300],
              ...
           ),
          home: Home(),
        ));
  }
}
```

## 5.2.2.Navegación

### 5.2.2.1. Navegación en React.js

#### 5.2.2.1.1. Uso de React Router DOM

React Router Dom es una librería que le brinda al desarrollador la posibilidad de controlar las rutas de una aplicación web. Al tratarse de una Single Page Application (SPA), no es posible manejar el enrutamiento mediante una configuración externa a la aplicación. Por esto, React Router Dom utiliza un enrutamiento dinámico, que sucede mientras que la aplicación es procesada por el sistema.

En el siguiente fragmento de código, podemos visualizar como, utilizando dicha librería, es posible definir que componentes mostrar dependiendo de la ruta en la que nos encontremos. Además de permitirnos configurar un comportamiento determinado, en caso de que la ruta ingresada no esté definida, utilizando el componente Redirect.

```
import React from "react";
import {
    BrowserRouter as Router,
    Switch,
    Redirect,
    Route,
} from "react-router-dom";
import {Activities} from "../components/Activities";
import {Form} from "../components/Form";
import {Home} from "../components/Home";
...
export const AppRouter = () => {
    ...
    return (
        <Router>
            ...
            <Switch>
                <Route exact path="/form"
component={Form} />
                <Route exact path="/activities"
component={Activities} />
                <Route exact path="/" component={Home} />
                <Redirect to="/" />
            </Switch>
```

```
            ...
        </Router>
    );
};
```

#### 5.2.2.1.2. Barra de menús

Decidimos utilizar una librería de React.js llamada react-responsive. Esta permite saber cuándo se está reduciendo la pantalla de un navegador web o, en el caso del móvil, cuando está o no en modo landscape, desplegando un Drawer o un BottomNavegation, según el caso (imagen 9 y 10 respectivamente).

Ambos son provistos por Material-ui, incluido el código necesario para que funcionen y poder aplicarles estilos.

Es importante destacar que estos componentes los provee material-ui pero no coinciden completamente con los diseños presentados por material.io. En especial el Drawer. De todas formas, se asemeja al diseño esperado.

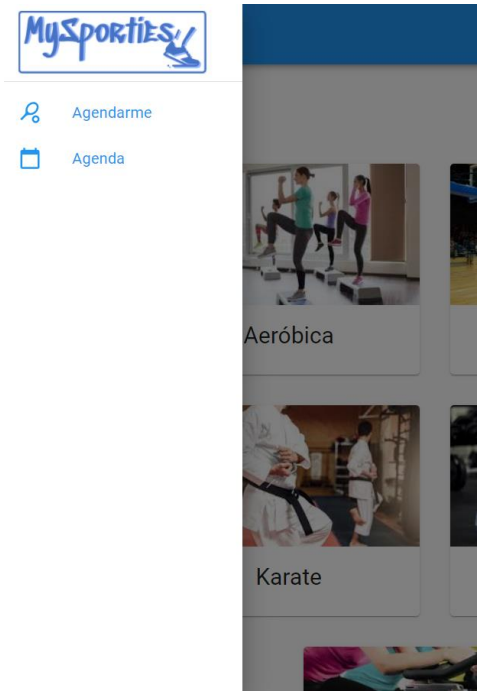



Imagen 9 - Drawer en React.js

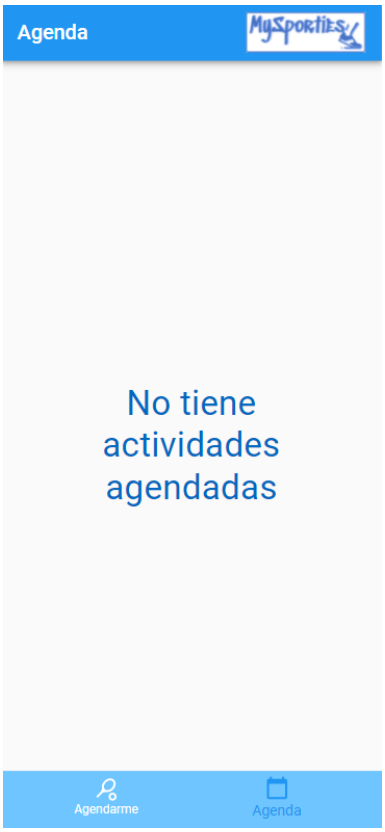



Imagen 10 - Bottom Navigation Bar en React.js

Los iconos utilizados en el BottomNavegation y en el Drawer fueron provistos por material-ui.

Tomamos esta decisión debido a que la interfaz tiene que poder adaptarse y ser fácil de usar en cualquier contexto. Por esto, si se está utilizando una pantalla estrecha, podemos suponer que se está accediendo desde un móvil, donde el manejo del dedo es distinto a que si está en modo landscape.

Por este motivo, realizamos una adaptación al código brindado por material-ui para poder desplegar un Drawer. Este es mostrado si el usuario desea, al seleccionar la opción dentro de la barra superior. Es importante destacar que, en el caso de encontrarse en posición landscape, no se mostrará el icono, bloqueando la posibilidad de desplegar el Drawer.

Como podremos ver en el siguiente fragmento de código, en el caso del BottonNavegation, no es necesario implementar el comportamiento

esperado, ya que este se mostrará al usuario. Esto siempre y cuando nos encontremos en posición portrait. Únicamente se centra en las acciones a definir en las rutas y el diseño es en su mayoría ofrecido por material-ui.

```
<BottomNavigation
        value={value}
        onChange={(event, newValue) => {
            setValue(newValue);
        }}
        showLabels
        className={classes.root}
>
        <CssBaseline />
        <BottomNavigationAction
            key="Agendarme"
            exact="true"
            className={classes.link}
            label="Agendarme"
            icon={<SportsTennisIcon />}
            component={Link}
            to="/activities"
        />
        <BottomNavigationAction
            key="Agenda"
            exact="true"
            className={classes.link}
            component={Link}
            to="/"
            label="Agenda"
            icon={<CalendarTodayIcon />}
        />
</BottomNavigation>
```

#### 5.2.2.2. Navegación en React Native

React Native provee una librería de navegación similar a la de React.js llamada react-navigation para poder navegar entre pantallas. Esta está centrada en la navegación entre pantallas existente en dispositivos móviles. Para este caso de estudio, hablaremos únicamente de tres tipos de navegación existentes en React Native. Esto es debido a que el resto no aplicaba a nuestro caso.

Cada uno de los patrones utilizados tuvieron que ser embebidos unos en los otros, logrando así el comportamiento esperado.

#### 5.2.2.2.1. Drawer Navigator

El Drawer es un patrón de navegación muy común entre distintas pantallas. La librería react-navegation incluye este componente. Es importante destacar que React Native Paper recomienda este Drawer, asegurando por su parte que este sigue los lineamientos establecidos por el sistema de diseño Material Design.

El resultado fue el que se puede observar en la imagen 11. Es importante aclarar que, siguiendo con el prototipo establecido, únicamente se mostrará dicho Drawer en el caso de encontrarse el dispositivo en posición landscape.

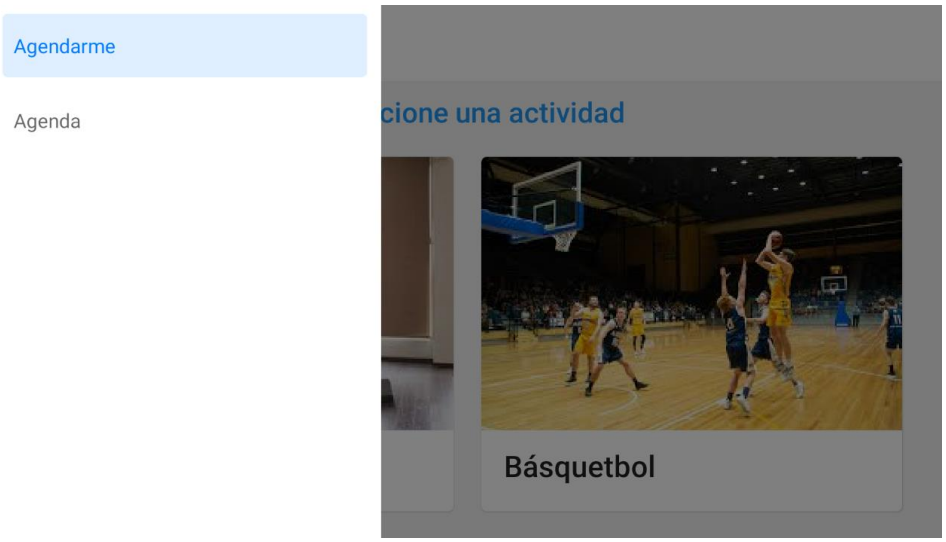



Imagen 11 - Drawer Navigator en React Native

En el siguiente fragmento de código, se muestra cómo se define el Drawer como componente y la definición de las pantallas a mostrar con sus respectivos nombres.

```
...
import {createDrawerNavigator} from '@react-
navigation/drawer';
import {StackCalendarNavigator, StackSelectNavigator}
from './StackNavigator';
const Drawer = createDrawerNavigator();
```

```
const DrawerNavigator = () => {
  return (
    <Drawer.Navigator>
      <Drawer.Screen name="Agendarme"
component={StackSelectNavigator} />
      <Drawer.Screen name="Agenda"
component={StackCalendarNavigator} />
    </Drawer.Navigator>
  );
};
...
```

Es importante destacar que React Native Paper no implementa ni recomienda el uso de una barra superior que siga con los principios de Material Design. Debido a esto, decidimos utilizar la barra superior que, por defecto, aparece en las distintas plataformas.

#### 5.2.2.2.2. Stack Navigator

El Stack Navigator es otro patrón ofrecido por la librería de navegación de React Native, que, a diferencia del resto, no establece un componente visible que especifique sus destinos. Este les brinda a sus componentes internos la posibilidad de customizar cuando crean necesario, la navegación entre ellos. De esta forma, le delega la responsabilidad de navegación a sus rutas, teniendo este, únicamente las posibles rutas existentes.

Este tipo de navegación fue utilizado para hacer la navegación desde la pantalla de selección de una actividad a la pantalla del formulario. Por defecto, realiza una transición animada, tanto en Android como en IOS.

Como se puede observar en el siguiente fragmento de código, es posible definir ciertos atributos a la hora de pasar de un componente a otro. En este caso, al navegar al formulario, es necesario enviarle la actividad seleccionada, con el fin de que el componente del formulario pueda registrar la actividad una vez completado el mismo.

```
import React from 'react';
import {createStackNavigator} from '@react-
navigation/stack';
```

```
import Activities from '../screen/Activities';
import Form from '../screen/Form';
import { Activity } from '../interfaces/interfaces';
export type StackParams = {
  Actividades: undefined;
  Formulario: {activity: Activity};
};
const StackAdd = createStackNavigator<StackParams>();
const StackSelectNavigator = () => {
  return (
    <StackAdd.Navigator>
      <StackAdd.Screen name="Actividades"
component={Activities} />
      <StackAdd.Screen name="Formulario" component={Form}
/>
    </StackAdd.Navigator>
  );
};
```

Es importante destacar el fuerte tipado requerido por Typescript, que especifica qué tipo de objeto es requerido para dicha transición.

A continuación, en el siguiente fragmento de código, podemos observar cómo al seleccionar una actividad en particular, es invocada la navegación hacia el formulario. Esto es posible gracias a que el Stack Navigator brinda a sus rutas una propiedad que delega dicha responsabilidad a estos componentes.

```
...
import {StackParams} from '../navegations/StackNavigator';
...
interface Props extends StackScreenProps<StackParams,
'Actividades'> {}
const Activities = ({navigation}: Props) => {
  ...
  return (
    <View style={styles.center}>
      <FlatList
        ...
        renderItem={({item}) => (
          <TouchableOpacity
            ...
            onPress={() => {
              startFormAddActivity();
              navigation.navigate('Formulario', {
                activity: item,
              });
            }}>
            <Card …>
              …
            </Card>
```

```
          </TouchableOpacity>
        )}
      />
    </View>
  );
};
```

#### 5.2.2.2.3. Material Bottom Tab Navigator

Por último, utilizamos el patrón Bottom Tab Navigator de navegación, ofrecido por la librería de React Native. Este tipo de navegación le permite al usuario navegar entre pantallas, seleccionando las posibles opciones de una barra inferior. Esta, además le indica al usuario en que pantalla se encuentra.

React Native Paper recomienda el uso de una librería externa que modifica dicho componente de navegación y lo adapta al diseño establecido por Material Design.

Sin embargo, por más que se obtiene un resultado similar en ambas plataformas (Android y IOS), el comportamiento en iOS es distinto, debido a que este no realiza el efecto correspondiente a la hora de seleccionar una pantalla distinta.

Es importante destacar que, para poder utilizar los iconos, fue necesario utilizar una librería externa a React Native Paper, recomendada por ellos, llamada vector icons. La misma tiene una instalación compleja, por lo que cuenta con una guía de instalación genérica. En el caso de preferir usar las versiones de React Native más recientes, es probable que se tengan que realizar modificaciones extras a las recomendadas en dicha documentación. En nuestro caso, tuvimos muchos problemas que lograron solucionarse gracias a una investigación en foros como Stack Overflow. Esto es debido a que, en el caso de utilizar React Native CLI, al instalar dicha librería, es necesario tener un conocimiento básico sobre el funcionamiento de los editores nativos, siendo estos Android Studio y XCode.

En el siguiente fragmento de código, se detalla la implementación del Bottom Tab Navegation, basado en Material Design, junto con la utilización de sus iconos.

```
...
import {createMaterialBottomTabNavigator} from '@react-
navigation/material-bottom-tabs';
...
import Icon from 'react-native-vector-
icons/MaterialIcons';
const Tab = createMaterialBottomTabNavigator();
const TabCalendarNavigator = () => {
...
  return (
    <Tab.Navigator
      screenOptions={({route}) => ({
        tabBarIcon: ({color, focused}) => {
          let iconName: string = '';
          switch (route.name) {
            case 'Agendarme':
              iconName = 'sports-tennis';
              break;
            case 'Agenda':
              iconName = 'calendar-today';
              break;
          }
          return <Icon name={iconName} size={20}
color={color} />;
        },
      })}>
      <Tab.Screen
        name="Agendarme"
        options={{title: 'Agendarme'}}
        component={StackSelectNavigator}
      />
      <Tab.Screen
        name="Agenda"
        options={{title: 'Agenda'}}
        component={StackCalendarNavigator}
      />
    </Tab.Navigator>
  );
};
...
```

#### 5.2.2.3. Navegación en Flutter

##### 5.2.2.3.1. Drawer

En el caso de Flutter, disponemos de un Widget que nos brinda la posibilidad de desplegar un Drawer por encima de la pantalla de la aplicación. Además, este cumple con los lineamientos establecidos por el sistema de diseño Material Design. Sin embargo, tuvimos varios inconvenientes a la hora de implementarlo, impidiéndonos lanzar la aplicación web. Por este motivo, optamos por utilizar un menú colapsable (imagen 12) similar al utilizado en diversas aplicaciones de Google a la fecha, por ejemplo, Gmail.

Este último, será visualizado únicamente en el caso de encontrarse el dispositivo en posición landscape, la cual, definimos que se aplicará cuando el ancho de la pantalla supere los 600 pixeles.

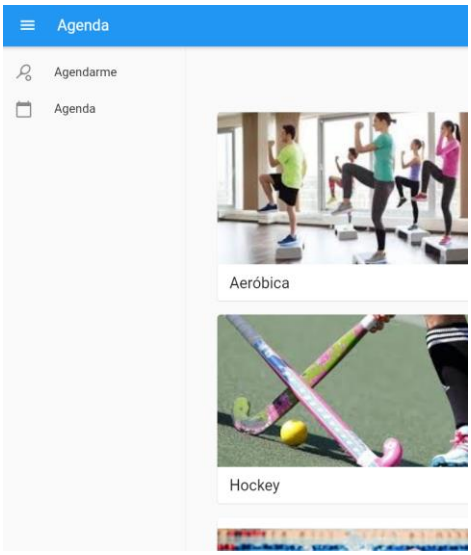



Imagen 12 - Menú desplegable (modo landscape)

##### 5.2.2.3.2. Bottom Navigation Bar

En el caso de que el dispositivo se encuentre en posición portrait, que, en nuestro caso, definimos en base a que el ancho de la pantalla sea superior a los 600 pixeles, mostraremos un Botton Navigation Bar. Este, sigue con el patrón de navegación, el cual

establece en la parte inferior de la pantalla, las posibles rutas a visualizar.

El Bottom Navegation Bar es un Widget ofrecido por Flutter que sigue los lineamientos establecidos por Material Design.

Como se puede visualizar en el siguiente fragmento de código, se le setean los colores establecidos en el tema de la aplicación, para las propiedades selectedItemColor, unselectedItemColor y backgroundColor. Contiene las opciones de navegación representadas mediante una lista de Widgets del tipo BottomNavigationBarItem. Estos últimos, permiten asociarlos con un icono y una etiqueta de texto.

Es necesario programar la transición dependiendo del ítem, dentro de su propiedad onTap. La variable _pageViewController contiene las pantallas a mostrar dependiendo de cada selección.

```
Widget _buildPortrait(BuildContext context) {
    return …
            bottomNavigationBar: BottomNavigationBar(
              backgroundColor: Colors.blue[200],
              selectedItemColor: Colors.blue[700],
              unselectedItemColor: Colors.white,
              onTap: (index) {
                _pageViewController.animateToPage(index,
                    duration: Duration(milliseconds: 200),
                    curve: Curves.bounceOut);
              },
              currentIndex: _currentIndex,
              items: [
                BottomNavigationBarItem(
                  icon: Icon(Icons.sports_tennis),
                  label: _titles[0],
                ),
                BottomNavigationBarItem(
                  icon: Icon(Icons.calendar_today),
                  label: _titles[1],
                ),
              ],
            ),
          ),
        )));
}
```

El resultado es similar al obtenido en React.js y al establecido por el sistema de diseño Material Design (imagen 13).

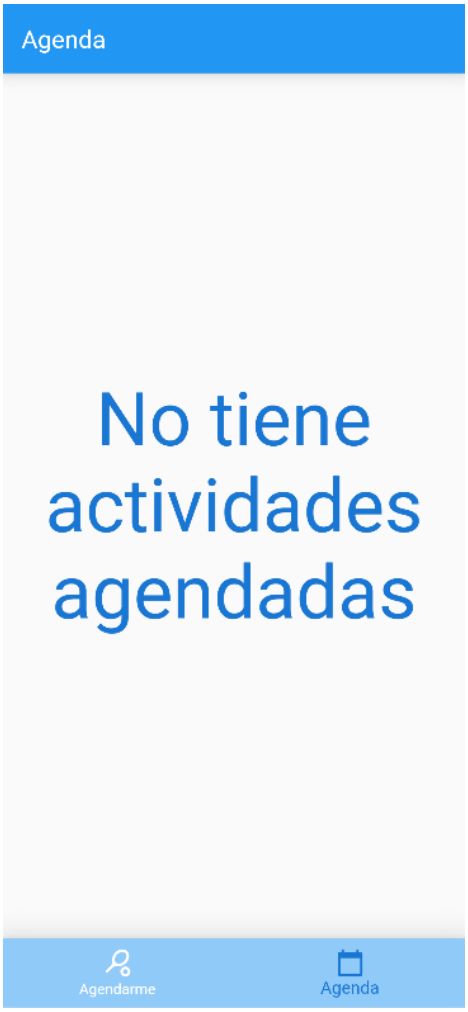



Imagen 13 - Bottom Navigation Bar (modo portrait)

### 5.2.3.Text Fields

#### 5.2.3.1. React.js

Existen varias formas de trabajar con los inputs en React.js. Sin embargo, material-ui ofrece un componente llamado TextField, que brinda el mismo funcionamiento que un input genérico de React.js, cumpliendo con el sistema de diseño Material Design. Sin necesidad de aplicar un estilo a dicho componente, este sigue los lineamientos y comportamientos de Material Design, como podemos ver en la imagen 14.

En nuestro caso, configuramos el tipo de entrada que recibirá el input, el error a mostrar, en caso de existir, y el texto guía, más conocido como hint. Además, utilizamos el useForm, el cual es un hook personalizado proveniente de la librería reat-hook-form, que nos permitirá obtener los datos del input.

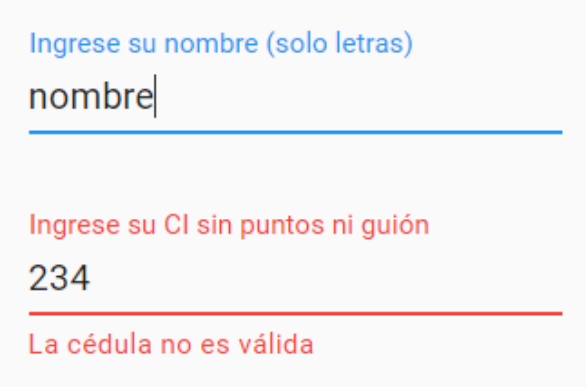



Imagen 14 - TextField en React.js

En el siguiente fragmento de código, podremos visualizar un ejemplo de uso de dicho componente, con una configuración en particular, enfocada en el ingreso del nombre de la persona.

```
<TextField
    label="Ingrese su nombre (solo letras)"
    type="text"
    error={errorName}
    id="standard-error-helper-text"
    {...register("name")}
    helperText={helperNameText}
    onChange={onNameChange}
/>
```

### 5.2.3.2. React Native

Al igual que React.js, React Native ofrece un componente especializado en el ingreso de datos. No obstante, hemos justificado anteriormente el uso React Native Paper como la librería más utilizada, a la hora de querer seguir con los lineamientos de Material Design en React Native. Por este motivo, utilizaremos el componente TextInput, brindado por la librería mencionada previamente, para las entradas de texto

necesarias en la prueba de contexto. Este componente se asemeja al resultado obtenido en React.js.

Sin embargo, a pesar de seguir los mismos lineamientos que React,js respecto a la obtención de datos, cuenta con un problema de diseño a la hora de rellenar estos inputs. Cuando se trata de manejar los errores, Material Design no recomienda la eliminación del texto de ayuda o hint, sino que coloca, por debajo de los datos ingresados, el mensaje de error. Esto cambiando el color de la línea base del TextInput (en nuestro caso a rojo). En cambio, el TextInput ofrecido por React Native Paper únicamente permite el cambio de color de la línea base y del hint. Sin poder agregar una descripción extra del error, incumpliendo de esta manera con los lineamientos de Material Design. Podemos visualizar este defecto en la imagen 15.

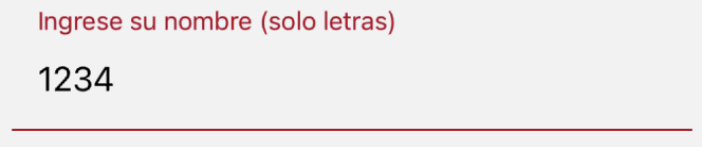



Imagen 15 - TextInput con error en React Native

En el siguiente fragmento de código podremos visualizar la creación de un TextInput, con la configuración donde especifica su hint, cómo actuar ante un cambio, su valor, estilo aplicado y booleano, indicando la existencia o no de un error.

```
<TextInput
    label="Ingrese su nombre (solo letras)"
    onChangeText={value => onChange(value, 'name')}
    value={name}
    style={styles.textInput}
    error={errorName}
/>
```

### 5.2.3.3. Flutter

Flutter sigue con los mismos lineamientos establecidos por el sistema de diseño Material Design y nos brinda un Widget especializado en el ingreso de datos, que se adapta a este.

El mismo permite configurar en sus propiedades que tipo de datos van a ser ingresados, un estilo personalizado, un hint a mostrar como ayuda al usuario y una acción a realizar en caso de que se quite el foco sobre este. En cuanto al ingreso de datos, permite aplicar validaciones, donde, en el caso de no cumplirlas, muestra el error, respetando lo establecido en el sistema de diseño. Además, permite realizar acciones en caso de que su valor cambie.

Con respecto a las otras tecnologías, el resultado es similar al obtenido en React.js. A continuación, podemos ver un fragmento de código donde se muestra la estructura de dicho Widget acompañado de la configuración mencionada anteriormente de sus propiedades.

```
<TextInput
    label="Ingrese su nombre (solo letras)"
    onChangeText={value => onChange(value, 'name')}
    value={name}
    style={styles.textInput}
    error={errorName}
/>
```

### 5.2.4. DateTime Picker

#### 5.2.4.1. React.js

Para la selección de fecha, usamos el componente KeyboardDatePicker y, para la selección de la hora, utilizamos el componente KeyboardTimePicker, ambos provistos por material-ui.

Como puede verse en la imagen 16, ambos son visualizados como diálogos donde el usuario ingresa una determinada fecha u hora dependiendo del componente.

Ambos siguen vagamente el lineamiento recomendado por Material Design, agregándole detalles distintos. En el caso de la selección de fecha, agrega un botón con el texto “Today”, que permite escoger el día actual. En el caso de la hora, permite únicamente la selección de la misma, utilizando un reloj. Material-ui justifica estos cambios basándose en la mejora de la experiencia de usuario que provocan estos elementos.

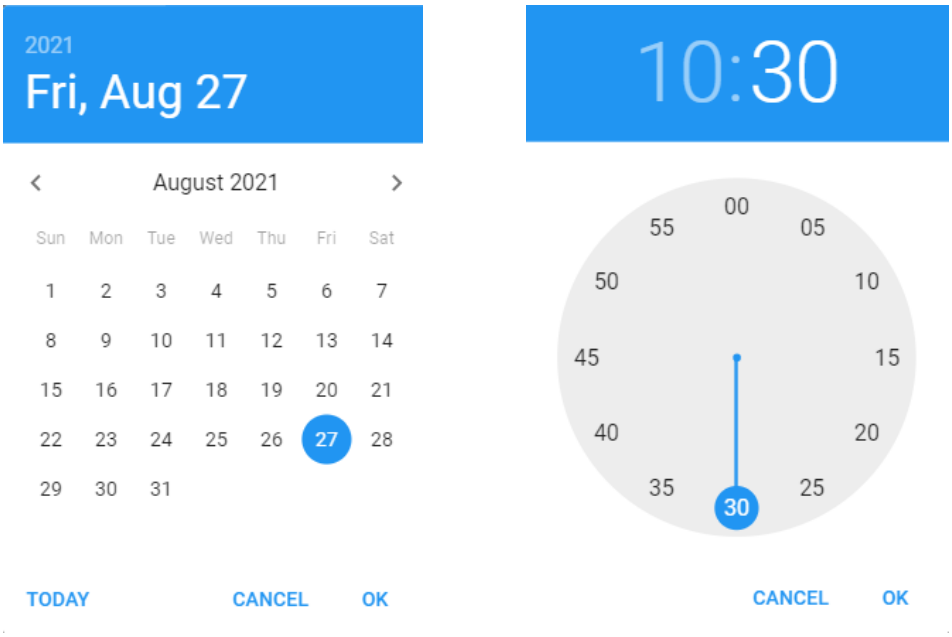



Imagen 16 – DateTime Picker en React.js

Como podemos ver en el siguiente fragmento de código, ambos tienen similares propiedades y se basan en la librería moment para la recolección de datos.

```
<MuiPickersUtilsProvider libInstance={moment}
utils={MomentUtils}>
    ...
        <KeyboardDatePicker
            margin="normal"
            id="date-picker-dialog"
            autoOk={true}
            showTodayButton={true}
            label="Selecione una fecha"
            format="DD/MM/yyyy"
            name="date"
            {...register("date")}
            value={selectedDate}
            inputValue={inputDateValue}
            onChange={onDateChange}
            rifmFormatter={dateFormatter}
            KeyboardButtonProps={{
                "aria-label": "change date",
            }}
            error={errorDate}
            helperText={helperDateText}
            className={classes.date}
        />
        <KeyboardTimePicker
            margin="normal"
```

```
                id="time-picker"
                ampm={false}
                label="Seleccione una hora"
                name="moment"
                {...register("hour")}
                value={selectedMoment}
                inputValue={inputMomentValue}
                onChange={onMomentChange}
                rifmFormatter={dateFormatter}
                KeyboardButtonProps={{
                    "aria-label": "change time",
                }}
                error={errorMoment}
                helperText={helperMomentText}
                className={classes.date}
            />
        ...
</MuiPickersUtilsProvider>
```

La librería moment permite el fácil manejo de las estampas de tiempo, centrándose en fechas, horas, minutos o días. Además, simplifica la validación de estos datos.

Es importante destacar el grado de madurez de estos componentes, al brindar un manejo con una de las librerías más utilizadas en cuanto a la gestión de fechas.

#### 5.2.4.2. React Native

En el caso de React Native, React Native Paper recomienda el uso de la librería react-native-paper-dates. Esta proporciona un diseño que se adapta mejor a lo establecido por Material Design.

No obstante, a diferencia de material-ui, estos componentes se encuentran ocultos y dejan a criterio del desarrollador como prefiere mostrarlos.

En nuestro caso, optamos por seguir los ejemplos mostrados en la documentación oficial, exponiéndolos mediante el uso de un componente del tipo Button, también proporcionado por React Native Paper. A través del uso de variables booleanas, definidas dentro de hooks, es que se hacen visibles, como se puede ver en el siguiente fragmento de código.

```
<Button
    style={styles.button}
    onPress={() => setOpen(true)}
    uppercase={false}
    mode="outlined">
    {date ? date?.format('DD/MM/yyyy') : 'Fecha'}
</Button>
…
<DatePickerModal
    mode="single"
    visible={open}
    onDismiss={onDismissSingle}
    date={datepicker}
    onConfirm={onConfirmSingle}
/>
<Button
    style={styles.button}
    onPress={() => setVisible(true)}
    uppercase={false}
    mode="outlined">
    {time ? time : 'Hora'}
</Button>
…
<TimePickerModal
    visible={visible}
    onDismiss={onDismiss}
    onConfirm={onConfirm}
    hours={12}
    minutes={14}
    label="Select time"
    cancelLabel="Cancel"
    confirmLabel="Ok"
    animationType="fade"
    locale={'en'}
/>
```

Sin embargo, al poner el dispositivo en modo landscape, el DatePickerModal se ve incompleto, impidiendo deslizar la pantalla para visualizarlo enteramente (imagen 17). Esto sucede en ambas plataformas, Android e iOS.

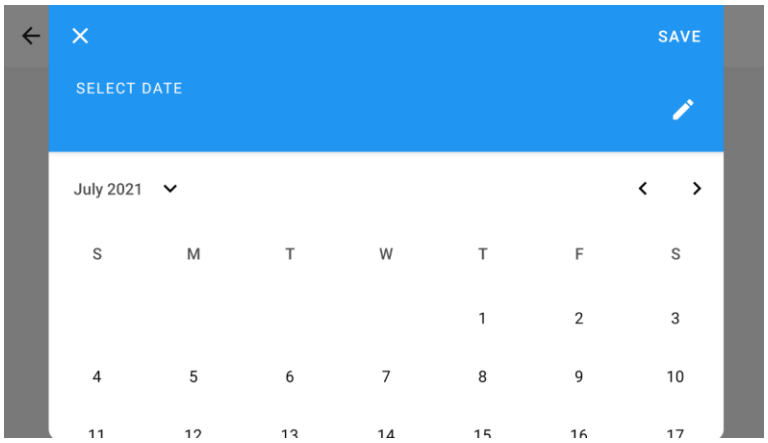


Imagen 17 - DatePickerModal en React Native (modo landscape)

### 5.2.4.3. Flutter

Flutter provee dos funciones que permiten mostrar diálogos para obtener una fecha u hora ingresada por el usuario. Estos cumplen con los estándares propuestos por el sistema de diseño Material Design. Además, no presentan problemas en cuanto a la posición en que se encuentre el dispositivo.

A continuación, veremos un fragmento de código que muestra la definición de dos funciones que serán llamadas al seleccionar el botón correspondiente a la fecha u hora. Estas modificarán un valor, que será evaluado posteriormente para validar estos datos.

```
Future<void> _selectDate(BuildContext context) async {
    final DateTime? picked = await showDatePicker(
        context: context,
        initialDate: selectedDate,
        firstDate: DateTime(2015, 8),
        lastDate: DateTime(2101));
    if (picked != null && picked != selectedDate)
      setState(() {
        selectedDate = picked;
      });
}
Future<void> _selectedTime24Hour(BuildContext context) async
{
final TimeOfDay? picked = await showTimePicker(
  context: context,
  initialTime: const TimeOfDay(hour: 10, minute: 47),
  builder: (BuildContext context, Widget? child) {
    return MediaQuery(
```

```
      data:
MediaQuery.of(context).copyWith(alwaysUse24HourFormat:
true),
      child: child!,
    );
  },
);
if (picked != null && picked != selectedTime)
  setState(() {
    selectedTime = picked;
  });
}
```

### 5.2.5. Grid

Según Material Design, un grid es una cuadrícula de diseño receptivo que se adapta al tamaño y la orientación de la pantalla, lo que garantiza la coherencia entre los diseños. [36]

En esta sección, veremos las distintas representaciones de un Grid de cada tecnología, detallando como éstas ayudan en las distintas plataformas.

#### 5.2.5.1. React.js

Para que los componentes internos se adapten, Material-ui provee un componente personalizado llamado Grid. Básicamente, es un div que renderiza los componentes dentro del mismo, en el espacio que dispone.

Para mostrar las imágenes de las actividades, seleccionamos las Cards que provee Material-ui, ya que coinciden con las Cards que provee material.io. Estas tienen un tamaño fijo, pero se acomodan fácilmente según el tamaño de pantalla que dispongan. En este caso, les asignamos un tamaño fijo, ya que la idea no es que sean responsive, sino que se adapten según la pantalla. A continuación, en las imágenes 18 y 19, podemos ver el resultado obtenido en un navegador y en un móvil, en posición landscape y portrait respectivamente.

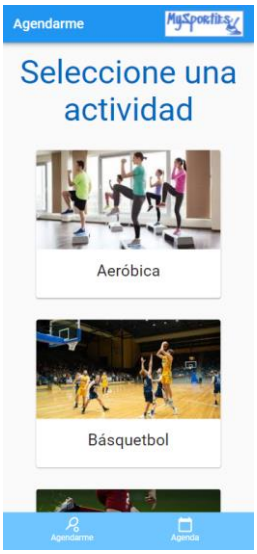



Imagen 19 - Grid en React.js (modo portrait)


Imagen 18 - Grid en React.js (modo landscape)

### 5.2.5.2. React Native

En el caso de React Native, React Native Paper no cuenta con un componente de tipo Grid como el brindado por material-ui en React.js. Existen librerías que permiten simular dicho comportamiento, pero al estar obsoletas por los titulares de las mismas y presentar fallas en su instalación, optamos por recrear dicho funcionamiento.

Utilizando el hook useWindowDimensions y teniendo en cuenta el tamaño asignado a las Cards proporcionadas por React Native Paper, logramos simular dicho comportamiento.

Como se puede ver en las imágenes 20 y 21, se aprovecha mejor el espacio disponible a la hora de colocar el móvil en posición landscape.

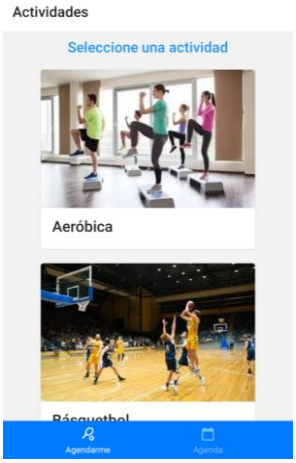



Imagen 21 - Grid en React Native (modo portrait)

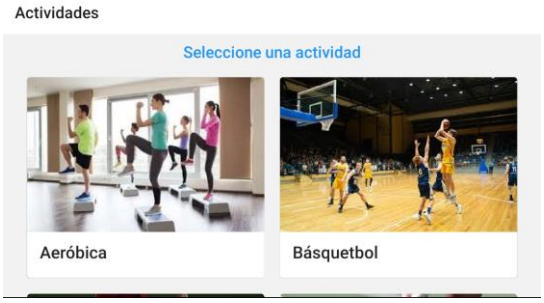



Imagen 20 - Grid en React Native (modo landscape)

No obstante, es importante aclarar que la animación, resultado de girar el dispositivo, no genera un efecto agradable, como lo es el caso del Grid utilizado en React.js.

### 5.2.5.3. Flutter

Flutter proporciona el Widget llamado Wrap, el cual tiene el mismo comportamiento que el Grid proporcionado por material-ui en React.js. Este widget presenta sus elementos como una fila y, cuando se queda sin espacio en la pantalla para visualizarlos, los ajusta a la siguiente línea.

Por defecto, la dirección en que intenta mostrar los componentes es horizontal. En nuestro caso, los componentes a mostrar son las Cards, ofrecidas por Flutter. Estas contienen las actividades que puede agendar el usuario y siguen los lineamientos establecidos por Material Design.

Como se puede ver en las imágenes 22 y 23, se aprovecha correctamente el espacio disponible horizontalmente para mostrar las actividades en los distintos tamaños de pantalla.

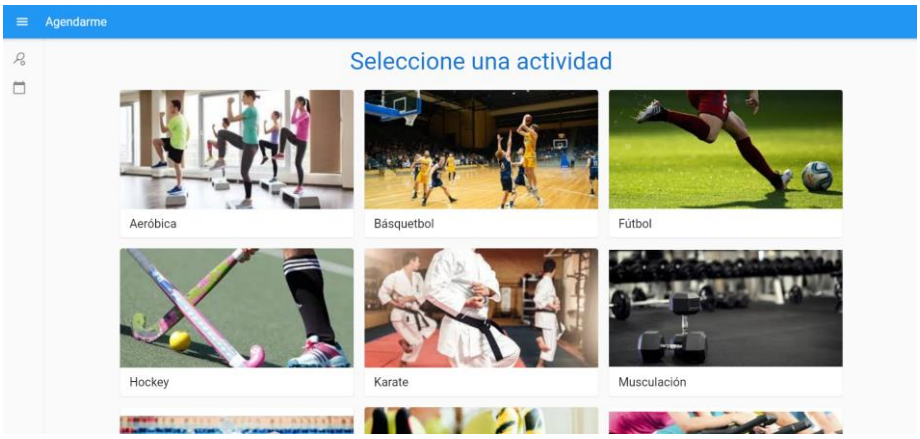



Imagen 22 - Grid en Flutter (modo landscape)


Imagen 23 - Grid en Flutter (modo portrait)

# 6. Conclusiones

A lo largo de este proyecto, hemos descubierto la importancia de tener en cuenta la experiencia de usuario a la hora de desarrollar una interfaz de usuario. Así como hacer énfasis en los atributos de calidad que más se relacionan con ella, como portabilidad, adaptabilidad, mantenibilidad y usabilidad. Esto nos permite desarrollar interfaces agradables para el usuario final. Utilizando las tecnologías que hemos visto, como Flutter, React.js y React Native y basándonos en un sistema de diseño como Material Design, es posible cumplir con los atributos de calidad mencionados anteriormente. Esto refiriéndonos al desarrollo de una interfaz de usuario, ahorrando o disminuyendo el tiempo y las personas necesarias para producir esto.

Existen otras tecnologías, además de las estudiadas en este proyecto, que cumplirían con la función del desarrollo multiplataforma adaptable. Nuestra selección de tecnologías a la hora de desarrollar un prototipo estuvo basada en una amplia investigación. Incluyendo una indagación previa de las características arquitectónicas y librerías de componentes de cada tecnología. Luego de dicha investigación, concluimos que eran las herramientas más utilizadas a la hora de desarrollar aplicaciones adaptables y multiplataforma.

Nuestro enfoque fue centrado en el desarrollo de aplicaciones multiplataforma y adaptables, ya sea para ser utilizadas como aplicaciones nativas desde un dispositivo móvil o únicamente desde un navegador web. A su vez, usando el sistema de diseño Material Design.

A la hora de elegir una tecnología que permita el desarrollo multiplataforma y adaptable, es importante contar con un buen sistema de diseño que sea soportado por la misma. Esto permite a los desarrolladores diseñar una interfaz de usuario uniforme, sin importar el nivel de complejidad de la misma. Logrando así, cumplir con los atributos de calidad necesarios a la hora de crear una interfaz de usuario.

## 6.1. Adopción y esfuerzo de desarrollo

Nuestros prototipos estuvieron centrados en evaluar los principales componentes de una interfaz de usuario, en distintas tecnologías. Es importante aclarar que, a la hora de comenzar a investigar, contábamos con conocimiento

intermedio en HTML, CSS y JavaScript. Sin embargo, desconocíamos el funcionamiento de las tecnologías a indagar.

### 6.1.1. React.js

Al empezar la investigación sobre cómo funcionaba React.js, nos resultó de mucha utilidad la documentación proporcionada, ya que fue clara y concreta acerca de cómo utilizar los distintos componentes. Esto ocasionó que encontráramos pocos inconvenientes, los cuales requirieron ser explorados en páginas recomendadas por los desarrolladores, por ejemplo, Stack Overflow.

Existe una gran dependencia y relación entre React.js y las etiquetas y estilos HTML y CSS, las cuales pueden ser usadas directamente en sus componentes programados con JavaScript. Esto facilitó nuestra comprensión sobre el framework de React.js.

Con respecto a la integración de Material Design, React.js cuenta con material-ui, una librería que se basa en este sistema de diseño, recomendada por el sitio oficial de Material Design. Utilizar esta librería no presentó dificultades, ya que su documentación es amplia y su grado de madurez es avanzado. Esto permite a los desarrolladores cumplir con Material Design, logrando así una interfaz de usuario multiplataforma adaptable para la web.

Lograr comprender el funcionamiento del framework de React.js y sus librerías aledañas, necesarias para el desarrollo del prototipo, sumando el desarrollo del mismo, nos llevó aproximadamente 120 horas.

### 6.1.2. React Native

Posteriormente, continuamos con el desarrollo del prototipo utilizando React Native. Esta librería basada en React.js nos permitió generar código nativo para las dos plataformas móviles dominantes en el mercado, Android e iOS. Para poder visualizar el código nativo, usamos React Native CLI. Además, preferimos utilizar como lenguaje de programación TypeScript sobre

JavaScript, con el fin de investigar las ventajas de usar un lenguaje fuertemente tipado a la hora de desarrollar una interfaz de usuario.

El tener conocimiento sobre React.js, su ciclo de vida y componentes, los cuales tienen un comportamiento similar al de React Native, nos permitió comprender fácilmente la tecnología. La documentación existente, al igual que en el caso de React.js, es extensa y completa, brindando también casos de uso concretos que ayudaron a asimilar los conceptos.

En un principio, el uso de un lenguaje fuertemente tipado, a diferencia de React.js, presentó una dificultad en el desarrollo y comprensión de los tipos necesarios para definir un componente, debido a las restricciones que impone. Por ejemplo, al crear una variable, siempre se debe indicar el tipo. No obstante, en las etapas más avanzadas del desarrollo, facilitó el uso de componentes y reutilización de código existentes para crear nuevas funcionalidades. Por esto, encontramos que, en el desarrollo de interfaces de usuario complejas, en el cual intervienen muchas personas, la utilización de un lenguaje fuertemente tipado podría facilitar el entendimiento del código fuente. Además, se logra que el mismo sea más ordenado y controlado por los desarrolladores.

Sin embargo, como comentamos en el capítulo anterior, Material Design no reconoce ninguna implementación realizada para React Native. Para el desarrollo del prototipo, nos enfocamos en React Native Paper y sus librerías recomendadas, con el fin de lograr los mismos resultados en cuanto a interfaz y experiencia de usuario que obtuvimos en el resto de las aplicaciones. Esto fue debido a que, gracias a nuestra investigación, observamos que era la solución que más se adaptaba a Material Design.

Debido a lo anteriormente comentado y al plazo del que dispusimos para realizar el prototipo, no logramos obtener la misma experiencia e interfaz de usuario que conseguimos utilizando Flutter o React.js. Muchos componentes genéricos de Material Design no se encuentran implementados correctamente o no siguen los lineamientos de este sistema de diseño. Un ejemplo de esto

es el Bottom Navigation Bar, el cual no presenta el mismo comportamiento y formato en Android e iOS.

De todas formas, los componentes básicos y la versatilidad del framework de React Native permiten crear interfaces de usuario adaptables y multiplataforma. Esto si el desarrollador conoce su funcionamiento y dispone del tiempo suficiente para lograrlo. En nuestro caso, esto se ve reflejado a la hora de mostrar más tarjetas (cards) con las actividades deportivas, cuando el dispositivo cuenta con el espacio de pantalla adecuado para hacerlo.

Conseguir el conocimiento necesario y desarrollar el prototipo presentado en React Native nos tomó aproximadamente 148 horas.

### 6.1.3. Flutter

Finalmente, el último prototipo que realizamos fue una aplicación multiplataforma y adaptable, tanto web como nativa, mediante el uso de Flutter. El desarrollo de este prototipo fue de vital importancia, ya que pudimos comparar el resultado a nivel multiplataforma web con React.js. También comparamos el de las aplicaciones multiplataforma nativas con React Native.

Como especifica la documentación de Flutter, éste se basa en el concepto de componentes, al igual que React.js, con una leve diferencia en los componentes mutables y no mutables. Esto, sumado a la claridad de la documentación y sus vastos ejemplos, facilitó su comprensión e hizo que el desarrollo del prototipo fuera más sencillo que para las tecnologías mencionadas anteriormente.

Al tener un conocimiento intermedio sobre el funcionamiento de los lenguajes fuertemente tipados, nos permitió comprender las estructuras básicas de Dart, el lenguaje utilizado por Flutter. No obstante, su curva de aprendizaje es elevada, aunque una vez asimilados los conceptos claves, el desarrollo se vuelve más fluido, con comparación con las otras tecnologías.

Flutter recomienda encarecidamente en su documentación, el uso de Material Design como sistema de diseño. Esto se debe a que la implementación de éste y el grado de conexión con la tecnología es muy alto, lo que permite al desarrollador lograr una interfaz de usuario unificada en Android, iOS y la web.

Obtener el nivel de conocimiento requerido para poder crear el prototipo, sumando su creación, nos llevó aproximadamente 86 horas.

### 6.1.4.Síntesis

A continuación, se brinda una tabla comparativa sobre lo mencionado anteriormente:

| Tecnología | Facilidad de adopción | Esfuerzo de adopción y desarrollo (horas) | Calidad de documentación y ejemplos | Líneas de código |
|---|---|---|---|---|
| React.js | Alta | 120 | Alta | 974 |
| React Native | Media | 148 | Alta | 792 |
| Flutter | Media | 86 | Alta | 749 |

Tabla 1 - Comparación de diversos aspectos del desarrollo para las tecnologías utilizadas

Podemos observar que la facilidad de adopción en React.js fue superior al resto. Esto se debe a, como comentamos anteriormente, su integración con Material Design y nuestro conocimiento previo sobre HTML, CSS y JavaScript. Sin embargo, notamos que, al ver las horas dedicadas, el factor más influyente fue la facilidad de desarrollo de la tecnología, una vez adoptada la misma.

No notamos grandes diferencias en la calidad de la documentación y ejemplos brindados por las distintas tecnologías. Respecto a las líneas de código, estas denotan la facilidad de desarrollo del prototipo en determinada tecnología. Esta afirmación se realiza teniendo en cuenta que en React Native no pudimos obtener el mismo resultado que en el resto de las tecnologías, ya

que su baja integración con Material Design repercutió en las horas de desarrollo.

En cuanto a la codificación, se buscó escribir un código limpio, evitando líneas innecesarias, apuntando a la mantenibilidad del mismo.

## 6.2. Identificación de dificultades durante el desarrollo

### 6.2.1. Adaptabilidad de la pantalla

Teniendo los tres prototipos finalizados, podemos afirmar que una gran dificultad fue lograr aprovechar el tamaño de la pantalla. Esto es, ubicar los componentes de manera adecuada, logrando así, la adaptabilidad objetivo.

En el caso de React.js, material-ui nos ofreció componentes que hacían este trabajo por nosotros. Por ejemplo, componentes que adaptaban la disposición de las tarjetas conteniendo las actividades según el tamaño de la pantalla. En cambio, en el caso de Flutter, tuvimos que personalizar dicho comportamiento, creando constantes y consultando continuamente el tamaño de pantalla disponible, con el fin de mostrar más tarjetas. En el caso de React Native, encontramos varios componentes brindados por terceros, que cumplían con este funcionamiento. Sin embargo, se encontraban obsoletos o desaconsejados por sus desarrolladores o presentaban fallas debido al incumplimiento de sus pruebas. Esto nos llevó, al igual que en el caso de Flutter, a programar dicho comportamiento.

### 6.2.2. Manejo de componentes

Al crear componentes en Flutter, se nos presentaron varios inconvenientes para comprender cuáles debían ser mutables y cuáles no, asumiendo en ciertas partes del proyecto que todos debían serlo. Al avanzar en su investigación, notamos que esto se debía al funcionamiento básico requerido en nuestro prototipo. Y en el caso de proyectos más complejos, podrían existir varios componentes no mutables, reutilizables a lo largo de toda la aplicación.

### 6.2.3. Sistema de diseño

En todas las plataformas, sin importar el grado de madurez de su adaptación con Material Design, existe una manera de aplicar un tema único a toda la aplicación. Logrando así, mantener un estilo en la misma. Aplicar este concepto nos resultó muy difícil en el desarrollo de todos los prototipos. Especialmente en el caso de React.js y React Native, donde, a diferencia de Flutter, los componentes no toman automáticamente los valores predefinidos en el tema, sino que hay que asignarlos. Esto es debido a que, al leer la documentación de estas plataformas, no se hace mucho énfasis, como es en el caso de Flutter. O se destinan a crear estilos personalizados, sin usar el tema, como el caso de React.js y React Native.

No fue posible lograr el mismo diseño en todas las tecnologías. Flutter sigue estrictamente los lineamientos del sistema de diseño Material Design y permite su seguimiento. En cambio, a pesar de que en React.js material-ui sigue la mayoría de estos lineamientos, también se desvía de muchos de ellos, bajo la justificación de que repercuten en la experiencia del usuario web. Finalmente, nos encontramos con el caso de React Native, en donde, la mayoría de los componentes existentes, son desarrollados por terceros, sin ser validados por Material Design. Esto dificulta enormemente lograr la misma interfaz y experiencia de usuario, optando en muchos casos por personalizar componentes existentes, logrando así cumplir con el sistema de diseño.

Es importante destacar que, el desarrollo o cumplimiento de Material Design, incluye también el uso de sus iconos y fuentes. Esto, no resultó difícil de aplicar a la hora de desarrollar en Flutter y React.js. Pero si provocó gran dificultad lograr utilizar dichas fuentes e iconos en React Native, ya que React Native Paper recomienda el uso de librerías de terceros para cumplir con los lineamientos. Dichas librerías cuentan con una documentación que no contempla posibles dificultades, tienen una compleja instalación y su funcionamiento, en muchos casos, no es el esperado. Debido a esto, tuvimos que modificar código nativo generado a partir del código fuente de nuestra aplicación en React Native para lograr este funcionamiento.

### 6.2.4. Diferenciación entre plataformas

Cuando desarrollamos en React.js y Flutter, no tuvimos que hacer diferencias en cuanto a la plataforma en la que nos encontráramos. Logrando así, obtener en todas estas el mismo resultado, salvo aspectos puntuales. Sin embargo, en el caso de React Native, en varias partes de la codificación, debimos tener en cuenta en que plataforma nos encontrábamos para brindar la misma experiencia de usuario. En muchos casos, dejando un incorrecto funcionamiento en determinada plataforma, debido al tiempo con el que se contaba.

## 6.3. Aspectos comunes y diferencias entre tecnologías

Todas las tecnologías que abarcamos en el desarrollo de prototipos permiten al desarrollador lograr una interfaz de usuario adaptable y multiplataforma. Siempre y cuando el desarrollador posea el conocimiento necesario para lograrlo.

### 6.3.1. Componentes

El principal aspecto que comparten estas tecnologías es el concepto de componentes. Con algunas diferencias mínimas, como en el caso de Flutter, que a pesar de reconocer su enfoque en React.js a la hora de crearlos, los define como widgets y los separa en mutables y no mutables. Esto ocasiona que su estructura en capas sea similar, permitiendo al desarrollador comprender una de ellas, conociendo previamente a la otra.

En React.js y React Native, a lo largo de los años, los desarrolladores han diferenciado dos tipos de componentes personalizados. Los destinados al diseño, normalmente considerados componentes. Y los que manejan acciones y estados, conocidos como contendedores. Esto nos hace pensar que el concepto de componentes utilizado por Flutter tuvo en cuenta esta tendencia.

A su vez, dicho concepto está muy ligado a las etiquetas HTML y al comportamiento y estilo que se le puede dar a una página web, utilizando CSS y JavaScript. Esto se debe a que, por detrás de estos componentes

personalizados que permiten desarrollar a un nivel de abstracción más alto, el enfoque en etiquetas definido por HTML sigue existiendo.

Lo anteriormente explicado da a conocer la evolución de los lineamientos propuestos por HTML, CSS y JavaScript. Denotando así, la importancia de conocer dicha estructura y funcionamiento. En algunos casos más, como React.js, en el cual podemos usar etiquetas HTML directamente, y en algunos casos menos, como React Native y Flutter. Por lo que, por más que se brinden estos frameworks como un nivel más alto de abstracción, siguen teniéndose en cuenta estos conceptos.

Además de lo comentado anteriormente, los estilos personalizados que se le pueden dar a estos componentes tienen un gran parecido con los que se le pueden definir a una etiqueta HTML. Compartiendo así, varios conceptos como el de margen y padding.

### 6.3.2. Layout

El concepto de layout tanto en React.js como en React Native, está directamente asociado con la creación de componentes complejos con paneles. En el caso de Flutter, casi todo es un widget, incluso los modelos de layout lo son. Como justificamos en la sección 6.3.1, estos están basados en el concepto de componente definido en React.js. Además, ya sea un componente o un widget, estos se pueden anidar, cambiar de tamaño, anclar, desanclar y cerrar. Permitiendo, de esta forma, crear en las distintas tecnologías, usando los estilos adecuados o un sistema de diseño, layouts que se adapten a diversas plataformas.

Por lo mencionado anteriormente, podemos decir que el manejo de layout es similar en todas las tecnologías estudiadas.

### 6.3.3. Personalización

Todas las tecnologías, además de sus componentes base, permiten crear componentes personalizados.

En el caso de Flutter, sus componentes base están directamente relacionados con Material Design, por lo cual no necesitamos una librería externa para utilizarlos. Por otro lado, React.js y React Native, además de sus componentes base, permiten crear componentes personalizados, los cuales pueden recibir propiedades customizadas. Estos pueden ser reutilizados, conteniendo, o dentro de, otros componentes. Además, las tres tecnologías cuentan con una gran personalización de estilos de dichos componentes, permitiendo alterar su aspecto.

Entonces, teniendo en cuenta la utilización de componentes recomendados por Material Design, podemos afirmar que todas estas plataformas cuentan con una amplia personalización de los mismos.

### 6.3.4. Mantenibilidad

La reutilización de componentes, con la que cuentan todas las tecnologías estudiadas, ayuda a que el código sea más mantenible. Esto se debe a que se puede englobar comportamientos y/o diseños similares en componentes, permitiendo ser usados nuevamente.

La reutilización de estilos en estos componentes dar lugar a futuros desarrolladores hacer referencia a estos, sin necesidad de volver a crearlos. Esto se ve reflejado en el uso de un tema genérico de la aplicación y de archivos destinados únicamente a almacenar dichos estilos.

La utilización de componentes de terceros, en nuestro caso recomendados por Material Design, favorece la mantenibilidad, ya que no es necesario estilizarlos ni definir su comportamiento básico. Esto aumenta el nivel de abstracción de nuestro código, dejando en manos de terceros, la codificación básica. Permitiendo así a los desarrolladores enfocarse aún más en cumplir las necesidades específicas de la aplicación. Centrándose exclusivamente en mantener actualizadas dichas librerías, ahorrando tiempo de desarrollo.

### 6.3.5. Contexto de datos

El manejo del contexto en todas estas tecnologías es muy similar. Por ello, podríamos decir que su utilización para almacenar ciertos datos que serán mostrados y modificados en diversos componentes, es un patrón que se repite en dichas tecnologías multiplataforma y que facilita el uso compartido de datos entre pantallas.

### 6.3.6. Grado de madurez

En el caso de React.js, el estado de madurez es avanzado, gracias a la gran cantidad de comunidades que existen, además de los desarrolladores que la utilizan. Esto se debe a que su apego con las etiquetas HTML y, el uso directo de CSS y JavaScript permite a desarrolladores con más experiencia comprender más fácilmente el funcionamiento de este framework. A su vez, es importante destacar el grado de madurez de material-ui, la cual permite a los desarrolladores utilizar el sistema de diseño propuesto por Material Design. Esto rompe con la dificultad existente a la hora de desarrollar una web adaptable y multiplataforma, reduciendo así el tiempo de desarrollo necesario para lograr este objetivo.

Si nos dirigimos a React Native, hablamos de una librería basada en los principios de React.js, que permite usar JavaScript y casi todas las librerías asociadas, a la hora de desarrollar aplicaciones móviles multiplataformas nativas. Su estado de madurez, pese a ser inferior al de React.js, es avanzado. Sin embargo, los sistemas de diseño asociados a React Native, no se encuentran bien respaldados. En especial el utilizado para nuestro prototipo, Material Design. Además, hay diferencias puntuales a la hora de crear componentes para determinada plataforma, teniendo que configurar ciertos estilos y comportamiento para lograr el mismo resultado en todas éstas. Por lo que podríamos decir que, esta tecnología es útil para desarrolladores que buscan seguir su propio sistema de diseño, ya que las herramientas que brinda esta tecnología permiten crear estilos y componentes genéricos personalizados. Dichos componentes, al ser configurados correctamente, usando las librerías adecuadas, pueden convertirse fácilmente

en componentes adaptables. Logrando así, aplicaciones multiplataforma y adaptables.

Por último, en el caso de Flutter, nos referimos a un framework por el cual Google apuesta a crear aplicaciones multiplataforma adaptables, tanto web como mobile nativo. Su grado de madurez es inferior al de las tecnologías mencionadas anteriormente. Sin embargo, posee una gran performance y adaptación con el sistema de diseño Material Design. Esto, sumado a la posibilidad de lograr desarrollar aplicaciones web y nativas, compartiendo el mismo diseño y con una fuente de código única, hacen que Flutter sea una gran apuesta para los desarrolladores del futuro. Es importante destacar que la posibilidad de generar una aplicación web, teniendo como base el código de Flutter, es posterior a la generación de aplicaciones móviles nativas que este framework posee. Por lo que, esta funcionalidad se encuentra más inestable a la fecha y aún no viene incorporada por defecto a la hora de crear una aplicación utilizando esta tecnología.

## 6.4. Líneas futuras de investigación

En base a este proyecto, surgen varias líneas de investigación que, debido al alcance del mismo no fueron posibles abordar. Entre ellas se destacan:

- La exploración de la adaptabilidad de la UI en dispositivos con paradigmas distintos, como podrían ser wearables, pantallas grandes, asistentes de voz o realidad virtual.
- Un estudio más minucioso acerca de la productividad en el mantenimiento a largo plazo de las aplicaciones de React.js, React Native y Flutter.
- La evolución, en más detalle, del sistema de diseño en la personalización, como es el caso de Material You, donde se observa una tendencia de mayor personalización centrada en el usuario y contexto.
- El estudio de interfaces de usuario multiplataforma y adaptables, tanto móviles como web, a partir de un único código fuente, en el contexto tecnológico. Teniendo en cuenta la nueva funcionalidad de React Native, que a la fecha se encuentra aún en desarrollo. Dicha funcionalidad

permite, al igual que Flutter, no solo lograr desarrollar aplicaciones móviles nativas para Android e iOS, sino también utilizar el mismo código fuente para crear una aplicación web.

# 7. Referencias bibliográficas


[1] Material Design, “Cross-platform adaptation”, [Online]. Disponible: https://material.io/design/platform-guidance/cross-platform-adaptation.html#when-to-adapt

[2] B. Shneiderman y C. Plaisant, Designing the User Interface: Strategies for Effective Human-Computer Interaction, USA, Pearson Education Inc., 2005.

[3] indeed, “What is a User Interface? Definition, Types and User Interface Example”, 2020 [Online]. Available: https://www.indeed.com/career-advice/career-development/user-interface

[4] GeeksforGeeks, “Software Engineering | User Interface Design”, 2018 [Online]. Available: https://www.geeksforgeeks.org/software-engineering-user-interface-design/

[5] Ergonomics of human-system interaction — Part 110: Interaction principles, ISO 9241-110, 2020.

[6] Página oficial de W3C (World Wide Web Consortium), “About The World Wide Web” [Online]. Available: https://www.w3.org/WWW/

[7] Página oficial de W3C (World Wide Web Consortium), “W3 Concepts”, [Online]. Available: https://www.w3.org/Talks/General/Concepts.html

[8] Página oficial de W3C (World Wide Web Consortium), “MOBILE WEB”, [Online]. Available: https://www.w3.org/standards/webdesign/mobilweb.html

[9] Página oficial de W3C (World Wide Web Consortium), “HTML & CSS”, [Online]. Available: https://www.w3.org/standards/webdesign/htmlcss.html

[10] Página oficial de W3C (World Wide Web Consortium), “The web standards model - HTML CSS and JavaScript”, 2014 [Online]. Available: https://www.w3.org/wiki/The_web_standards_model_-_HTML_CSS_and_JavaScript

[11] Flux Academy, “Why Is Layout Important in Graphic Design?”, [Online]. Available: https://www.flux-academy.com/blog/why-is-layout-important-in-graphic-design

[12] genbeta, “Diferencia entre layout fluid y responsive”, 2011 [Online]. Available: https://www.genbeta.com/desarrollo/diferencia-entre-layout-fluid-y-responsive

[13] TechTerms, “Fluid Layout”, 2015 [Online]. Available: https://techterms.com/definition/fluid_layout

[14] Smashing Magazine, “Fixed vs. Fluid vs. Elastic Layout: What’s The Right One For You?”, 2009 [Online]. Available: https://www.smashingmagazine.com/2009/06/fixed-vs-fluid-vs-elastic-layout-whats-the-right-one-for-you/

[15] Interaction Design Foundation, “Adaptive vs. Responsive Design”, 2020 [Online]. Available: https://www.interaction-design.org/literature/article/adaptive-vs-responsive-design

[16] UXPin, “Responsive Design vs. Adaptive Design: What’s the Best Choice for Designers?”, [Online]. Available: https://www.uxpin.com/studio/blog/responsive-vs-adaptive-design-whats-best-choice-designers/

[17] Clarcat, “Native vs Hybrid vs Responsive: ¿Qué tipo de desarrollo elegir?”, 2019 [Online]. Available: https://www.clarcat.com/native-vs-hybrid-vs-responsive-tipo-desarrollo-elegir/

[18] Torque Magazine, “Responsive vs. Adaptive Design”, 2013 [Online]. Available: https://torquemag.io/2013/09/responsive-design-vs-adaptive-design/

[19] C. Stary y A. Totter, “Measuring the adaptability of universal accessible Systems”, *Behav. IT*, vol. 22, pp. 101-116, mar. 2003.

[20] Andrés Mauro, “Impacto de los sistemas de diseño en la ingeniería de la interfaz de usuario del software”, Tesis de grado, Universidad ORT Uruguay, Montevideo, 2021.

[21] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, 3rd ed., USA, Pearson Education, Inc., 2015.

[22] Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, ISO/IEC 25010, 2011.

[23] Nielsen Norman Group, “Usability 101: Introduction to Usability”, 2012 [Online]. Available: https://www.nngroup.com/articles/usability-101-introduction-to-usability/

[24] Slashmobility, “¿Qué son los sistemas de diseño?”, 2020 [Online]. Available: https://slashmobility.com/blog/2020/07/que-son-sistemas-diseno/

[25] BrowserStack, “How to deal with Android Fragmentation?”, 2020 [Online]. Available: https://www.browserstack.com/guide/what-is-android-fragmentation

[26] BrowserStack, “Understanding the severity of iOS fragmentation”, 2020 [Online]. Available: https://www.browserstack.com/guide/what-is-ios-fragmentation

[27] Página oficial de React, [Online]. Available: https://es.reactjs.org/

[28] Página oficial de material-ui, [Online]. Available: https://material-ui.com/es/

[29] Material Design, “Color Tool”, [Online]. Available: https://material.io/resources/color/#!/?view.left=0&view.right=0

[30] Página oficial de React Native, [Online]. Available: https://reactnative.dev/

[31] Página official de React Native, ”Setting up the development environment”, [Online]. Available: https://reactnative.dev/docs/environment-setup

[32] xencov, “React Native - Expo vs React Native CLI?”, 2018 [Online]. Available: https://xencov.com/blog/react-native-expo-vs-react-native-cli

[33] Página official React Native Paper, [Online]. Available: https://reactnativepaper.com/

[34] Página oficial de Flutter, [Online]. Available: https://flutter.dev/

[35] Página oficial de Dart, [Online]. Available: https://dart.dev/

[36] Material Design, “Responsive layout grid”, [Online]. Available: https://material.io/design/layout/responsive-layout-grid.html#columns-gutters-and-margins