Universidad de Alcalá
Escuela Politécnica Superior

MÁSTER EN DESARROLLO ÁGIL DE SOFTWARE PARA LA WEB

Trabajo Fin de Máster

DESARROLLO DE INTERFACES DE USUARIO CON FLUTTER

ESCUELA POLITECNICA SUPERIOR

Autor: Francisco Miguel Sáez Bravo

Tutor/es: Antonio Ortiz Baíllo

2022

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER EN DESARROLLO ÁGIL DE SOFTWARE PARA LA WEB

Trabajo Fin de Máster

DESARROLLO DE INTERFACES DE USUARIO CON FLUTTER

Autor: **Francisco Miguel Sáez Bravo**

Director: **Antonio Ortiz Baíllo**

Tribunal:

Presidente: _____

Vocal 1º: _____

Vocal 2º: _____

Calificación: _____

Alcalá de Henares a de de 2022

Índice Resumido

Índice Detallado

Índice de Figuras

INTRODUCCIÓN

A modo de introducción, y con el fin principal de contextualizar el marco del proyecto, es importante realizar un primer acercamiento a Flutter como herramienta de desarrollo, así como a los principios básicos que rigen su lógica funcional.

La evolución tecnológica experimentada en los últimos tiempos ha propiciado un cambio de paradigma dentro del ámbito del desarrollo de software, dirigiéndose hacia una progresiva componentización del mismo. La componentización, entendida como un enfoque de desarrollo que consiste en la división del software en piezas identificables que pueden ser construidas y desplegadas de forma independiente (Douglas Mcllroy, 1968), representa el tránsito entre modelos monolíticos tradicionales (alto grado de acoplamiento, funcionalidad única y centralizada) y las actuales arquitecturas de microservicios (elementos independientes que pueden ser construidos, desplegados, testeados y escalados de forma aislada, respondiendo a una coordinación común) (Villamizar et al., 2015).

De igual modo, el uso cada vez más extendido de dispositivos como smartphones o tablets ha generado nuevas necesidades y oportunidades dentro del contexto del desarrollo de software. Bajo esta premisa, a la hora de afrontar un proyecto de desarrollo, es preciso tomar en consideración los diferentes ecosistemas tecnológicos (IOS, Android y web), ya que un planteamiento más tradicional, centrado únicamente en entornos web, no permitiría acceder a todo el público objetivo.

Este último factor ha motivado la aparición de herramientas destinadas al desarrollo de aplicaciones multiplataforma. Estas herramientas, siguiendo en su mayoría el mismo enfoque de componentización software citado previamente, permiten, a través de una misma estructura interna, la exportación y visualización de dichas aplicaciones en cualquier tipo de dispositivo, independientemente del sistema operativo subyacente en cada uno de ellos.

Por su parte, Flutter aúna los conceptos descritos hasta este punto, abordando el desarrollo de aplicaciones desde una perspectiva diferencial. Flutter, en esencia, es un kit de herramientas de interfaz de usuario portátil open-source desarrollado por Google, orientado a la creación de aplicaciones nativas en entornos móviles, web y, en sus últimas versiones, en entornos de escritorio.

Para poder valorar el componente diferencial de este framework de desarrollo, es necesario comprender la estructura interna de una aplicación. En términos generales, toda aplicación se divide en dos partes claramente diferenciadas, distinguiendo entre el comportamiento (que engloba todos los procesos internos que determinan la lógica de negocio y funcionalidad de la

aplicación) y la presentación (que representa el apartado visual, núcleo de la interacción del usuario con la propia aplicación).

A diferencia de las soluciones presentes actualmente en mercado, las cuales emplean dos lenguajes diferentes a la hora de gestionar los dos ámbitos funcionales descritos previamente, Flutter permite combinar y gestionar de forma integral ambos entornos a través de un mismo lenguaje: Dart.

Dart es un lenguaje de programación orientado a objetos, desarrollado igualmente por Google y empleado para el desarrollo de aplicaciones web y móviles. Tal y como se ha mencionado, este lenguaje permite expresar al mismo tiempo tanto el comportamiento como la interfaz gráfica, como puede verse en la tabla siguiente, eliminando así la necesidad de recurrir a lenguajes de marcado adicionales, sentando las bases de una perspectiva *UI as code* (Frank Moreno , 2019).

Framework	Lenguaje de comportamiento	Lenguaje de presentación
Xamarin	**C#**	XAML
React Native	JavaScript	JSX
NativeScript	JavaScript	XML
Flutter	**Dart**	**Dart**

Figura 1. Tabla que muestra la utilización de lenguajes por ámbitos funcionales

Siguiendo con este análisis de los rasgos característicos de Flutter, cabe destacar que el modelo de componentización del software influye notablemente en el modelo arquitectónico del mencionado framework, pues toda su lógica funcional se basa en la construcción y composición de los denominados *widgets*. Estos elementos son bloques de construcción reutilizables empleados para la elaboración de interfaces de usuario, representando por consiguiente la unidad mínima de funcionalidad dentro del ecosistema de Flutter.

En complemento a este último aspecto y a diferencia de alternativas como React Native, Flutter no se apoya en componentes primitivos de las diferentes plataformas (bloques preconstruidos que conforman las interfaces de los sistemas Android o iOS), sino que, en contraposición, permite controlar cada píxel de la propia pantalla, otorgando así un mayor grado de control y personalización sobre la interfaz de usuario.

Finalmente, todos estos aspectos dan forma al carácter declarativo de Flutter. La mayoría de los frameworks de desarrollo suelen emplear un estilo de programación imperativo a la hora de construir la interfaz de usuario, teniendo que definir manualmente tanto el propio componente de la interfaz como sus consecuentes mutaciones y cambios de estado.

Flutter reduce la carga de responsabilidad a este respecto, encargándose directamente de los procesos de transición entre estados. El framework construye la interfaz de usuario para reflejar el estado actual de la aplicación, y si este cambia, los componentes afectados se redibujan, generando una nueva instancia el widget en cuestión.

Como conclusión, la capacidad de poder gestionar todo el ecosistema funcional de una aplicación (y su posterior despliegue en los diferentes entornos existentes) a través de una única base de código (Dart), sumado al dinamismo que aportan la componentización basada en widgets y el posicionamiento *UI as code* declarativo, convierten a Flutter en una alternativa muy interesante a la hora de abordar el desarrollo de aplicaciones multiplataforma.

OBJETIVOS DEL PROYECTO

El objetivo principal de este Proyecto de Fin de Máster (TFM) es la realización de un estudio sobre Flutter como herramienta de desarrollo de interfaces de usuario y aplicaciones multiplataforma, desde un enfoque fundamentalmente teórico, así como a través de una aproximación a nivel práctico.

Para alcanzar la consecución del objetivo expuesto, se contemplan diversos objetivos específicos o subobjetivos ligados al mismo, siendo estos:

- Contextualización del proyecto.
  - ✓ Breve descripción de los antecedentes: evolución y condicionantes previos a la situación actual.
  - ✓ Descripción del estado del arte en lo que respecta al ámbito del desarrollo aplicaciones, destacando el papel de Flutter dentro del mismo.
- Elaboración de un informe teórico sobre la herramienta.
  - ✓ Descripción de los factores determinantes para el uso de Flutter.
  - ✓ Introducción y descomposición detallada de los conceptos y pilares principales sobre los que esta se sustenta.
  - ✓ Estudio de su arquitectura y componentes fundamentales.
- Desarrollo práctico de un sencillo prototipo basado en la herramienta Flutter, buscando así ejemplificar la funcionalidad y estructura real de las soluciones tecnológicas de este tipo.
  - ✓ Tomando como referencia el trabajo evaluable correspondiente a la asignatura de *Aplicaciones Nativas e Híbridas* (aplicación de gestión de películas desarrollada en Android Studio), construcción de un prototipo que permita replicar esta misma funcionalidad dentro del ecosistema Flutter-Dart.
  - ✓ Desarrollo del apartado de backend de la aplicación, correspondiente a la gestión de BBDD, peticiones HTTP, estructuras de datos, etc.

- ✓ Desarrollo del apartado de frontend de la aplicación, correspondiente a la construcción de las interfaces, menús y opciones de navegación.

- Elaboración de un informe comparativo de carácter general con herramientas similares dentro de este ámbito.

  - ✓ Descripción teórica React Native e Ionic.

  - ✓ Informe comparativo entre Flutter, React Native e Ionic en base a sus características más destacables.

ESTADO DEL ARTE

En la actualidad, el ecosistema de las apps representa una parte esencial de las rutinas y actividades diarias de la sociedad en su conjunto, ofreciendo servicios y respuestas ante prácticamente cualquier necesidad existente en el consumidor final.

A fin de ejemplificar esta circunstancia, y tomando como referencia valores del año 2022, (Ian Blair, 2022) en la actualidad el total de apps disponibles en la plataforma App Store alcanza los 1,96 millones, cantidad que, en el caso de la plataforma de Google (Google Play), asciende hasta los 2,87 millones.

En relación con este factor, se estima que un 49% de la población mundial abre una misma aplicación más de 11 veces al día. En el contexto de la población *Millenial* (William Strauss & Neil Howe, 1991), nacidos entre el año 1981 y 1996, estos datos son aún más significativos, pues un 21% de los integrantes de este segmento social llegan a abrir una misma aplicación más de 50 veces en el mismo día. En términos económicos, en conjunto, los mencionados aspectos se traducen en unas ganancias estimadas para el sector de 935 mil millones de dólares en el año 2023.

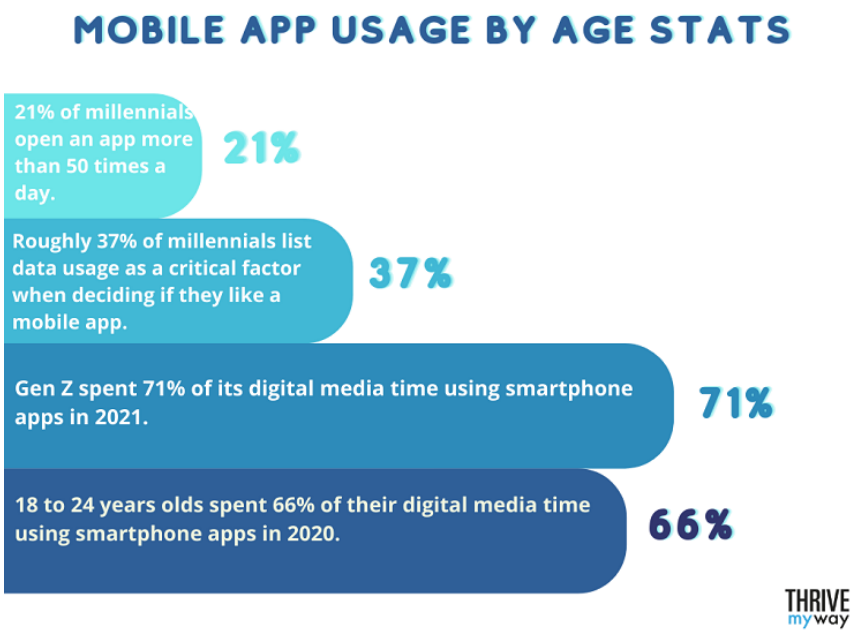

Figura 2. Estadísticas de uso de apps por edad (Fuente: *Thrivemyway*)

Estas referencias estadísticas permiten evidenciar el impacto socioeconómico que representan las aplicaciones en la actualidad, pero este impacto no es sino consecuencia de un interesante proceso evolutivo experimentado durante los últimos años.

Originalmente, una app se definía como "*un componente informático o software diseñado para cumplir con una funcionalidad concreta*". Posteriormente, y condicionada tanto por la explosión de los smartphones y dispositivos móviles en general, como por la aparición de nuevas tecnologías vinculadas a estos nuevos contextos de desarrollo, esta acepción se ha visto ampliada, incluyendo: "*[…] una funcionalidad concreta* ***y que puede ser descargada en un teléfono o cualquier otro dispositivo móvil***".

Esta evolución en la propia definición es, en esencia, un reflejo del cambio de paradigma que ha atravesado el desarrollo de aplicaciones en los últimos tiempos. Este proceso, ante la necesidad de adaptarse a un ciclo de vida de desarrollo considerablemente más corto que en el caso del software convencional, y al depender directamente de la madurez tecnológica de los dispositivos móviles, ha propiciado el desarrollo de diferentes tipos de aplicaciones, caracterizadas por los recursos disponibles y necesidades existentes en cada momento (Wafaa S. El-Kassas, 2017).

Durante las primeras etapas del desarrollo de aplicaciones, influenciadas notablemente por los principios de la web 2.0 y las arquitecturas cliente-servidor, proliferó el desarrollo de las denominadas **aplicaciones web**. Estas soluciones, como su propio nombre indica, se caracterizan por el empleo de tecnologías y lenguajes propios del entorno web, como son HTML, CSS y JavaScript. Este tipo de aplicaciones no precisan de su instalación en ningún dispositivo, ya que el acceso a las mismas se realiza a través de navegadores web mediante una URL.

Como puntos a favor, las aplicaciones web, de facto, están construidas sobre estándares tecnológicos trasversales a todo el sector (tecnologías web), lo que permite que, a través de un único desarrollo, una misma aplicación pueda ser desplegada en diferentes plataformas, optimizando por consiguiente los procesos de mantenimiento asociados a la misma.

En contraposición, este tipo de soluciones cuentan con un rendimiento considerablemente pobre, pues el empleo de los navegadores web como intérpretes de código reduce notablemente la velocidad de respuesta durante los procesos de interacción con el usuario. Igualmente, las aplicaciones web no cuentan con la posibilidad de acceder a funciones nativas de los dispositivos móviles, como pueden ser la cámara o el sistema de localización, entre otros muchos (Devsaran, 2016).

Posteriormente, e impulsado por el auge de los dispositivos móviles inteligentes, este proceso de evolución conduce al desarrollo de **aplicaciones nativas**, las cuales son desarrolladas empleando las herramientas y lenguajes propios de cada una de las principales plataformas móviles (iOS/Android). Esto hace que los productos resultantes adquieran un nivel de rendimiento mucho mayor que en el caso de las aplicaciones web, además de posibilitar un acceso directo a las funcionalidades internas de los dispositivos móviles.

De igual modo, este modelo de desarrollo también condiciona el despliegue de las propias aplicaciones, pues una aplicación desarrollada, por ejemplo, en Android Studio, solo podrá ser desplegada en dispositivos cuyo sistema operativo sea Android, ocurriendo exactamente lo mismo en el caso de las arquitecturas iOS.

Este último factor supone un impacto importante a la hora de abordar el proceso de desarrollo y mantenimiento de una aplicación. Bajo esta perspectiva, si se desea elaborar una solución integral buscando abarcar todo el mercado objetivo, será preciso realizar dos desarrollos en

paralelo, que permitan acceder tanto al ecosistema Android como al ecosistema iOS, duplicando por consiguiente los costes y recursos necesarios (Piotr Nawrocki, 2021).

Buscando dar respuesta a esta nueva problemática, y como último escalón del mencionado proceso evolutivo, surgen las denominadas **aplicaciones híbridas y multiplataforma**, cuyo desarrollo se caracteriza por una simbiosis entre las tecnologías propias de la web y las tecnologías nativas. De forma resumida, estas aplicaciones se diseñan y codifican en base a los estándares web (HTML, CSS, JavaScript), a lo que se le añade una capa funcional que permite el acceso a las diferentes funcionalidades móviles nativas. Esto, en definitiva, se traduce en el aspecto más diferencial de este tipo de soluciones: la existencia de una misma y única base de código (Minh Huynh et al., 2017).

De nuevo, bajo esta perspectiva se consigue una estandarización de la arquitectura subyacente, lo que favorece un despliegue prácticamente uniforme en los diferentes ecosistemas tecnológicos, minimizando así los tiempos y costes de desarrollo al no precisar de desarrollos paralelos.

A cambio, esta lógica funcional produce un ligero empeoramiento del rendimiento general con respecto a las soluciones nativas, presentando además ciertas carencias en lo referente a experiencia de usuario. Como consecuencia de ello, se precisa de esfuerzos adicionales en cuanto a maquetación si se desea adaptar el *look and feel* de la interfaz de la aplicación a cada uno de los formatos nativos.

Figura 3. División funcional tipos de aplicaciones (Fuente: *Witsdigital)*

Con el fin de completar este análisis del estado del arte referente al desarrollo de aplicaciones, resulta conveniente destacar que todos los modelos de desarrollo citados hasta este punto (y, por ende, los diferentes tipos de aplicaciones derivados de estos) son totalmente viables en la actualidad, pues en base a las necesidades y contextos sobre los que se desee incidir, unos tipos de aplicación se adaptarán mejor que otros.

Por ejemplo, en aplicaciones de pequeña y mediana escala donde los márgenes económicos son más ajustados, optar por un modelo basado en aplicaciones multiplataforma permitiría acceder a un nicho de mercado mucho mayor, reduciendo en paralelo los costes de desarrollo y mantenimiento. Por otro lado, para aplicaciones de mayor calibre en las que el rendimiento representa la piedra angular de su éxito, una aplicación nativa sería la opción más adecuada, pues el beneficio a obtener compensaría la necesidad de llevar a cabo los desarrollos correspondientes a los ecosistemas de iOS y Android.

A modo de conclusión, y pese a la evidente mejora que proporcionan los frameworks de aplicaciones híbridas y multiplataforma con la posibilidad de estandarizar el despliegue a través de un único repositorio de código, el desarrollo de aplicaciones es un ecosistema en constante cambio y crecimiento, y sobre el que aún queda mucho camino por recorrer.

Retomando una de las ideas citadas en el apartado de introducción del presente proyecto, pese a contar con una única base de código, la división entre la parte de lógica funcional (*Backend*) e interfaz de usuario (*Frontend*) actualmente implica la utilización de dos lenguajes diferentes y específicos para gestionar cada uno de estos ámbitos funcionales.

Las nuevas vías de investigación e innovación surgen del intento de unificar ambas perspectivas en un mismo lenguaje, el cual, realizando la transición desde los modelos imperativos tradicionales hacia un paradigma declarativo, permita definir de forma simultánea tanto el diseño como el comportamiento de una determinada app.

Este concepto recibe el nombre de **UI as Code**, y es precisamente en este punto donde la figura de Flutter adquiere especial relevancia.

FLUTTER COMO FRAMEWORK DE APLICACIONES MULTIPLATAFORMA

Contextualización

Como parte de la evolución tecnológica expuesta en el apartado anterior, y en respuesta a las problemáticas intrínsecas que lleva asociadas este proceso en sus diferentes fases, Flutter se alza como un elemento trasgresor a los estándares actuales.

En estos primeros compases de su "vida", Flutter ha puesto sobre la mesa todo un nuevo abanico de posibilidades en lo referente al desarrollo de aplicaciones (Lukas Dagne, 2019). Desde una perspectiva más a largo plazo, posiblemente esté asentando, en paralelo, las bases de nuevos modelos de desarrollo y de economización del software, sin desprestigiar en ningún momento la calidad del mismo, sino más bien incluso potenciándola.

Flutter es, en esencia, un framework gratuito y open-source orientado a la creación de interfaces de usuario, creado y mantenido por Google y lanzado a mercado en diciembre de 2018. Desde este momento, su popularidad entre los usuarios ha crecido progresivamente, alcanzando porcentajes de uso similares a otras soluciones con mayor bagaje y experiencia dentro del sector, como puede ser React Native, llegando incluso a superar a esta última durante el pasado año, tal y como puede apreciarse en la Figura 4.

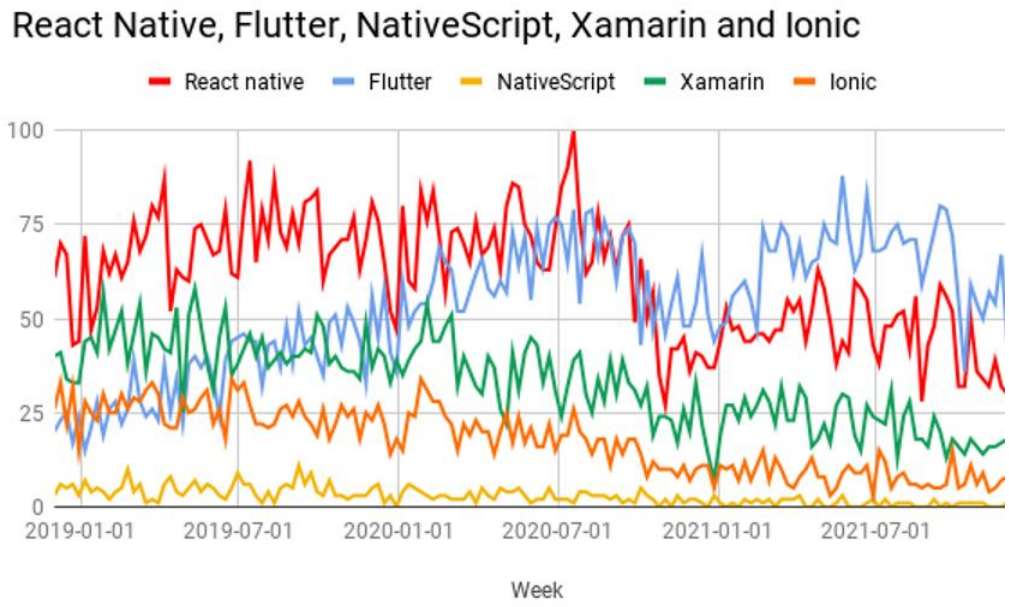

Figura 4. Gráfico porcentajes de uso de frameworks multiplataforma (Fuente: *Mentormate)*

Esta tendencia ascendente en cuanto al empleo de Flutter en el contexto del desarrollo de aplicaciones no es sino consecuencia directa de una interesante propuesta de valor, tanto a nivel lógico como a nivel funcional. La herramienta en cuestión nace bajo una visión de empoderamiento del usuario, buscando abordar el desarrollo de interfaces y aplicaciones desde una perspectiva simplificada, apoyándose en una máxima: "*Write once, deploy everywhere"* (CK Hicks, 2019).

Descripción General

Siguiendo este mantra, a nivel estructural Flutter representa la simbiosis entre dos elementos principales: por un lado, Flutter es un SDK (Software Development Kit), es decir, una colección de herramientas que sirven como apoyo en el proceso de desarrollo de aplicaciones, incluyendo soluciones que permiten la compilación de todo el código generado en los respectivos entornos nativos (IOS y Android). Por otro lado, cumple con las funciones de framework, poniendo a disposición del usuario una colección de elementos reutilizables de interfaz de usuario (menús, botones, formularios, etc.) que pueden ser personalizados a fin de ofrecer una funcionalidad adaptada a unas necesidades y requisitos más específicos.

En base a esta estructura lógica, y de nuevo tomando como referencia los aspectos citados en el apartado *Estado del Arte* del presente documento, el objetivo principal de Flutter ha ido orientado a resolver la complejidad que, en algunos puntos, presenta el desarrollo de aplicaciones en la actualidad, pudiendo destacar los siguientes:

- **Ciclos de desarrollo más largos y costosos**: de optar por desarrollos nativos, se plantea una dicotomía entre elaborar un único proyecto y, con ello, apostar por un único ecosistema (IOS o Android), o establecer varias estrategias y equipos de desarrollo que, en paralelo, construyan una solución específica para cada entorno. Esto, en definitiva, tiene un impacto evidente sobre el proceso de desarrollo en su conjunto, en términos de coste, organización y funcionalidad final del producto.

- **Empleo de múltiples lenguajes**: en relación con el aspecto anterior, desde el punto de vista del desarrollador, estos se ven obligados a trabajar en dos ámbitos completamente distintos, pues de cara a abordar íntegramente las necesidades del mercado, las aplicaciones se construyen sobre dos entornos funcionales independientes e incompatibles entre sí. Estos, por consiguiente, aplican sus propios estándares de lenguaje, por lo que los elementos funcionales desarrollados en uno de ellos no pueden ser reutilizados en el otro, afectando considerablemente a la productividad general del desarrollador.

  Igualmente, con la intención de hacer frente a esta problemática, en los últimos tiempos se ha tendido a extender, quizá en demasía, la funcionalidad de ciertos lenguajes de

programación, buscando abordar una mayor cantidad de ámbitos funcionales con una misma herramienta. Esto ha ocurrido, por ejemplo, con JavaScript, pero este, al igual que prácticamente cualquier lenguaje, está ideado y optimizado para responder a un contexto y necesidades tecnológicas concretas, por lo que trasladar su lógica funcional fuera de este entorno suele conllevar una devaluación del rendimiento.

- **Alta latencia de construcción y compilación:** especialmente en el caso de Android y Android Studio, los tiempos de compilación y reconstrucción de las aplicaciones, suelen prolongarse significativamente en el tiempo, más aún si se trata del primer despliegue. Pese a que esta mecánica ha evolucionado hacia una situación algo más estable, sigue siendo un factor que, de nuevo, afecta negativamente a la productividad en el desarrollo de aplicaciones.

- **Cierta inestabilidad en las soluciones multiplataforma existentes:** al igual que se citó en el apartado anterior, las herramientas orientadas al desarrollo multiplataforma solventan algunos de los aspectos indicados, pero vienen acompañadas de ciertas debilidades en cuanto a rendimiento y experiencia de usuario, puesto que, a fin de cuentas, no responden a una funcionalidad realmente nativa, y han de suplir esta circunstancia a través de diversos mecanismos.

Para hacer frente a estas limitaciones funcionales, Flutter abandera un posicionamiento sustentado en dos estandartes principales: un alto rendimiento y un **control total** de la interfaz de usuario.

En la actualidad, gran parte de los frameworks para el desarrollo de aplicaciones multiplataforma recurren a JavaScript y HTML para llevar a cabo los procesos de renderización de las interfaces, llegando incluso a proporcionar elementos nativos propios de cada uno de los entornos. Esta mecánica es la principal causante de las carencias en cuanto a rendimiento descritas con anterioridad, pues para que esta lógica de funcionamiento sea viable, se precisa de lo que se conoce como *JavaScript Bridge.*

JavaScript no se compila en código nativo, es interpretado y, esta interpretación se hace sobre la marcha durante la ejecución de la aplicación. Por ello, a grandes rasgos el denominado *JavaScript Bridge* es una capa intermedia entre el código generado por la plataforma (JavaScript/HTML) y el código y funcionalidades nativas, la cual permite realizar peticiones a la API de cada sistema operativo a fin de renderizar los componentes específicos de cada uno de ellos. En consecuencia, esto es lo que genera la pérdida de rendimiento en los procesos internos de la aplicación, ya que, bajo este modelo, esta última necesita de una capa extra para renderizar definitivamente la interfaz de usuario.

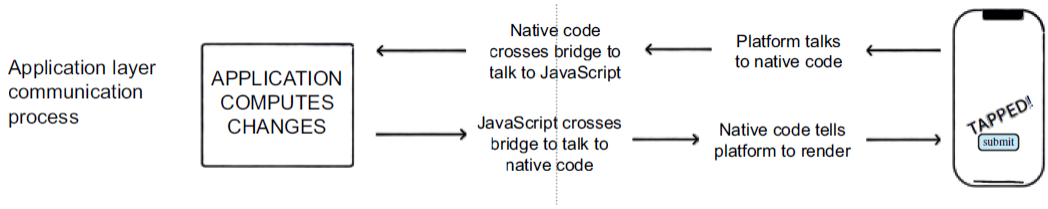

Figura 5. Representación de JavaScript Bridge (Fuente: *css-tricks*)

En contraposición, la propuesta de Flutter en cuanto a rendimiento pasa por eliminar directamente este sistema, pues el propio framework se convierte en el encargado de la renderización de la interfaz **píxel por píxel**, por lo que no precisa realizar llamadas a librerías o componentes externos para acceder a los diferentes componentes nativos.

Esto nos conduce directamente a la capacidad de establecer un control total de la interfaz de usuario. Mientras que, por lo general, el resto de frameworks replica las funcionalidades nativas, ya sea a través de elementos definidos en HTML y estilizados mediante CSS; o a través de las mencionadas peticiones directas a la API de cada uno de los sistemas operativos, Flutter, por su parte asume la responsabilidad total en el desarrollo de la interfaz de usuario. Con ello, se convierte en un elemento agnóstico de la plataforma en la que se despliegue, eliminando así las limitaciones asociadas a cada una de ellas.

La consecuencia directa de este hecho es la consecución de uno de los principios básicos de Flutter: el empoderamiento del usuario. Si bien otros frameworks ofrecerán tanta funcionalidad o llegarán tan lejos como permita cada uno de los componentes funcionales nativos que replica, Flutter, al no depender de una infraestructura o estándares concretos, otorga a los usuarios total libertad a la hora de elaborar la interfaz de usuario y sus respectivos componentes, empleando para ello herramientas que garantizan esta posibilidad sin suponer una penalización en cuanto a rendimiento o limitaciones de diseño.

De igual modo, pese a que en un primer momento esta perspectiva lleve a pensar que el *look and feel* de los entornos nativos puede verse afectado o distorsionado, Flutter cuenta con librerías integradas que proporcionan patrones de diseño adaptados a cada uno de los entornos funcionales (concretamente, *Material* en el caso de Android y *Cupertino* para iOS), tal y como se muestra en la Figura 6. Además, Flutter incluye reglas semánticas a nivel de interfaz que habilitan y simplifican la construcción de diseños más complejos, posicionándose como una alternativa interesante a las reglas de diseño de CSS, que en ocasiones pueden llegar a ser ligeramente densos.

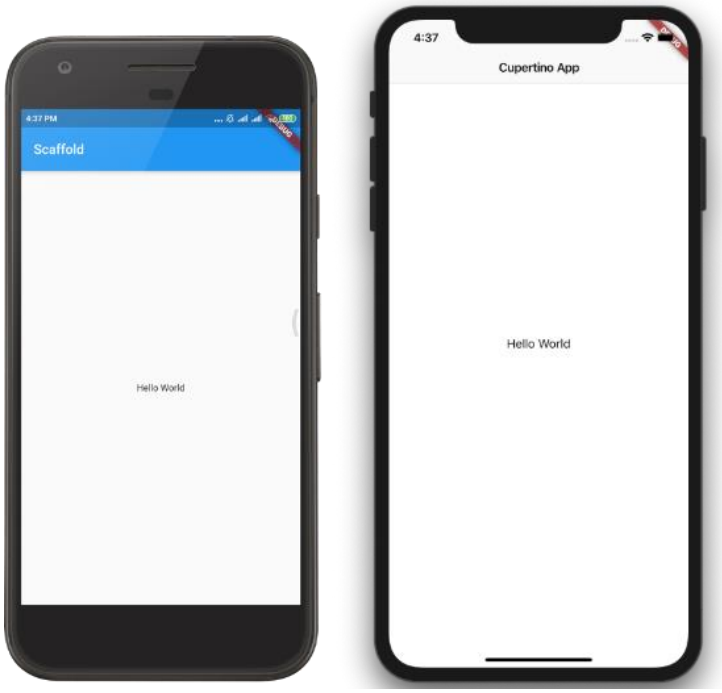

Figura 6. Ejemplo básico interfaces nativas con Flutter (Fuente: *Deimantas, A., Medium*)

Lenguaje de Programación Dart

Todos los aspectos que acompañan esta propuesta de valor son posibles gracias al empleo de **Dart** como lenguaje de programación, el cual sustenta toda la lógica funcional del framework. Dart es un lenguaje de programación orientado a objetos, fuertemente tipado y desarrollado por Google, al igual que Flutter. Cuenta con una sintaxis similar a la del entorno Java y, aunque técnicamente no tiene un uso restringido, se utiliza principalmente en la creación de componentes e interfaces frontend para web (AngularDart y, muy recientemente y en fases experimentales, Flutter web) y aplicaciones móviles (Flutter).

Una de las características principales de Dart es su flexibilidad en lo referente a la ejecución del código, lo cual influye directamente en las posibilidades de desarrollo que este proporciona. Sobre esta cuestión, Dart presenta dos formatos de compilación:

- **Compilación AOT (Ahead Of Time)**: empleada a la hora de ejecutar una determinada aplicación en su versión *release* y centrada especialmente en la optimización del rendimiento de la misma. En esta mecánica, el código es compilado antes de su ejecución, convirtiendo Dart en código nativo desde el inicio.

- **Compilación JIT (*Just-In-Time*)**: empleado durante las fases de desarrollo, este método de compilación busca agilizar los workflows y las modificaciones de código. Para lograr

este objetivo, el código se compila sobre la marcha, habilitando así una mecánica de *hot reload* que permite implementar los cambios y reconstruir las interfaces de manera dinámica en mitad de una misma ejecución.

En concordancia con esta característica, Dart otorga a Flutter la mejora de rendimiento indicada previamente, pues gracias a la compilación AOT se elimina la necesidad de establecer un puente entre el código generado en el framework y los componentes y entornos nativos, haciendo que tanto el despliegue como la ejecución de las aplicaciones sea notablemente más ágil.

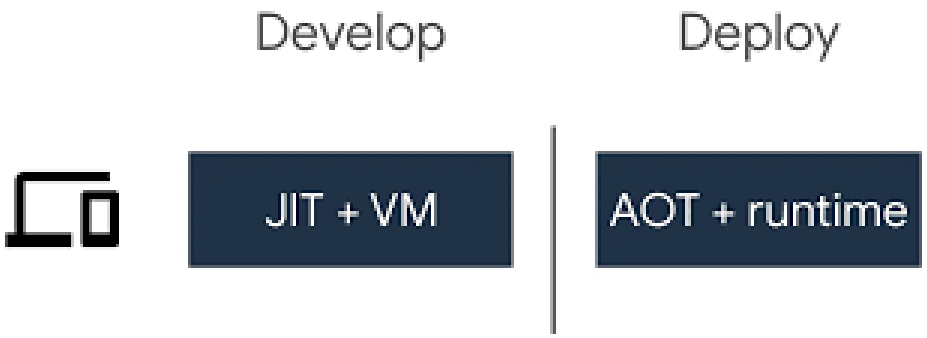

Figura 7. Diagrama compilación JIT/AOT (Fuente: *Copyfuture*)

Por último, de cara a establecer una visión global y completa de Dart como lenguaje de programación, es necesario exponer la que, quizá, es la piedra angular de su lógica funcional y, por ende, de Flutter. El modelo agnóstico que se obtiene a través de la desvinculación, a nivel de código, de los entornos nativos, da lugar a un aspecto diferencial con respecto a soluciones similares: la capacidad de unificar en un mismo lenguaje o base de código tanto el diseño de las interfaces como su comportamiento funcional. Este factor establece una sinergia directa con la naturaleza declarativa de este lenguaje en cuestión, lo que, en su conjunto, le hace orientarse hacia la denominada *declarative UI.*

Declarative UI y UI as Code

El término *declarative UI* representa el último paso en la transición entre los modelos imperativos tradicionales y el paradigma declarativo que, durante los últimos tiempos, ha supuesto un notable impacto sobre los estándares del desarrollo software en general.

A grandes rasgos, el paradigma imperativo define el conjunto de órdenes para resolver un determinado conjunto de problemas, describiendo los pasos necesarios para encontrar la solución y las mutaciones de estado correspondientes. Por contraposición, el modelo

declarativo expresa una lógica computacional sin necesidad de describir explícitamente su flujo de control, centrándose más en qué va a hacer la aplicación que en los propios "cómos".

El auge reciente de este paradigma se debe a que, en esencia, un modelo declarativo proporciona una capa de abstracción superior (ya que no ha de definir los flujos de control), lo que ha apoyado y motivado el proceso de "*softwarización"* a diferentes niveles en el que actualmente nos vemos sumergidos. El ejemplo más claro a este respecto podría ser Kubernetes, el cual, basado en un esquema declarativo, ha permitido automatizar los recursos y despliegues en entornos *Cloud*, extendiendo las vías de "*servitizacion*" que ya presentaban los modelos comerciales IaaS, PaaS o SaaS que, hoy en día, concentran gran parte de la oferta tecnológica.

Trasladando este planteamiento al contexto de la interfaz de usuario, los modelos imperativos se basan en la construcción manual de los diferentes componentes funcionales de la interfaz, definiendo en paralelo métodos complementarios que, posteriormente, permitirán alterar el estado interno de los mismos a medida que la aplicación avance en su ejecución.

Por su parte, bajo la perspectiva de una *declarative UI* no se gestionan los cambios de estado como tal, sino que desde el inicio se establece una definición de las diferentes "variantes" de una misma interfaz de usuario en función de los posibles escenarios que puedan tener lugar en tiempo de ejecución. Por lo tanto, un cambio de estado no supone la modificación del elemento afectado, sino que directamente desencadena una reconstrucción total o parcial de la interfaz de usuario, eliminando posibles discordancias en lo referente a la gestión interna de los estados y permitiendo una actualización más fiel de los contenidos tras una modificación. Este paradigma, al menos a nivel de interfaz de usuario, no ha conseguido postularse como una opción realmente viable hasta hace relativamente poco, pues ha precisado de una evolución considerable en los procesadores y CPUs que permitiera llevar a cabo estos procesos de reconstrucción en espacios de tiempo aceptables.

De la mencionada sinergia a través de un único lenguaje entre la capacidad de gestionar los elementos de diseño y los aspectos funcionales de forma simultánea, y el paradigma declarativo que rige la lógica funcional, surge lo que en apartados previos se ha definido como ***UI as Code.***

A efectos prácticos, esta estandarización de la infraestructura subyacente hace que herramientas propias del contexto funcional o conductual (reutilización de código, abstracción, mantenimiento, etc.) puedan aplicarse igualmente al contexto de diseño, por lo que el desarrollo de ambos ámbitos pasa a entenderse como una unidad, dando lugar a una definición lógica de la interfaz a través del propio código (*UI as Code)*. Esta mecánica genera una integración natural entre ambos contextos, favoreciendo una mayor flexibilidad en las interfaces y aplicaciones, y facilitando los procesos de desarrollo.

En definitiva, los componentes funcionales básicos de Flutter son igualmente los componentes básicos de la interfaz. Estos elementos se conocen como ***Widgets***.

Widgets

La capacidad de control total sobre la interfaz de usuario que proporciona el agnosticismo del lenguaje Dart, así como el nivel de abstracción funcional que habilita el paradigma declarativo, todo ello en concordancia con un planteamiento orientado a objetos y de carácter funcional que promueve la componentización y reutilización de los elementos, permite alcanzar una altísima atomicidad en lo referente a la construcción de interfaces.

Dentro del ecosistema de Flutter, esta atomicidad se centraliza en los denominados *widgets*, citados al cierre de la sección anterior. Un widget es una descripción inmutable de una sección de la interfaz de usuario[1], es decir, son los bloques de construcción a través de los cuales se define la interfaz de una aplicación.

Estableciendo una clasificación general, inicialmente los widgets podrían dividirse en 3 grupos principales, atendiendo a su funcionalidad y tipología:

- ***Widgets de valor***: son los elementos encargados de almacenar un determinado valor, ya sea con el fin de mostrar un valor preestablecido a los usuarios o con el fin de gestionar la introducción de un nuevo valor por parte de los mismos. Algunos ejemplos de este tipo de widgets podrían ser `Text, Icon o Image`, entre otros.

- ***Widgets de diseño***: los widgets de diseño son los encargados de gestionar la disposición de los componentes de interfaz de usuario generados, englobando funcionalidades como el posicionamiento relativo de cada sección de la interfaz, su tamaño o su forma. Algunos ejemplos son `Card, Row, Scrollable, o Padding`.

- ***Widgets de navegación***: finalmente, este tipo de widgets permiten gestionar la presencia y transición de los usuarios entre las diferentes pantallas de la aplicación. `Drawer, Navigation o TabBar` podrían servir como ejemplos a este respecto.

De igual forma, cabe destacar que existen ciertos widgets que no se engloban dentro de los mencionados grupos, puesto que su funcionalidad es algo más abstracta y no se adapta a un ámbito funcional concreto. En este apartado encontramos, por ejemplo, el widget `GestureDetector`, encargado de identificar las acciones de pulsación del usuario (tap, doble tap, arrastre, etc.), o el widget `Cupertino`, encargado de aplicar la estética y *look and feel* correspondiente a un elemento nativo del ecosistema IOs.

Uno de los aspectos más importantes de los widgets como instrumento funcional reside en que un widget, por sí mismo, no cuenta con un estado mutable, por lo que en consecuencia todos los atributos que lo componen son inmutables. Si se precisa de una gestión de estados, esta circunstancia ha de ser implementada conscientemente, lo cual nos conduce a una segunda forma de clasificación, en este caso atendiendo a la naturaleza de cada widget:

[1] https://api.flutter.dev/flutter/widgets/Widget-class.html

- ***Stateless Widgets***: aquellos widgets que no precisen de una gestión de estados extenderán de `StatelessWidget`, estando destinados únicamente a la emisión o representación de datos fijos. Bajo esta circunstancia, no es posible cambiar el estado interno de estos datos, por lo que tampoco se volverá a renderizar este componente de la interfaz tras producirse un cambio en tiempo de ejecución.

- ***Statefull Widgets***: en este caso este tipo de widgets extenderán de `StatefulWidget` y, aunque en esencia representan igualmente una sección de la interfaz de usuario, en este caso sí se pueden modificar los datos o atributos internos de los mismos, de forma que la interfaz de usuario se reconstruye cada vez que se produce un cambio, pudiendo reflejar dichas modificaciones dinámicamente.

Por último, es igualmente importante hacer énfasis en que los tipos de widgets no han de limitarse a los tipos o subdivisiones expuestas previamente, pues otro de los rasgos diferenciadores de Flutter reside en la capacidad de composición, casi infinita, que proporcionan estas unidades funcionales.

A este respecto, en base a lo comentado al principio de esta sección, un widget representa la unidad mínima de construcción de interfaces dentro del entorno de Flutter, pudiendo además especializarse en funcionalidades o ámbitos concetos (valor, navegación, etc.). Este hecho, de nuevo, se complementa con la capacidad agnóstica y declarativa del ecosistema funcional sobre el que se apoyan, lo que, en su conjunto, permite una gestión integral de cada píxel de la interfaz gráfica.

Gracias a ello, la riqueza funcional de Flutter como framework de desarrollo surge de la ***composición progresiva de los mencionados widgets***, combinando entre sí widgets "primitivos" con funcionalidades muy concretas, a fin de generar estructuras más complejas que respondan a requisitos funcionales o necesidades más avanzadas o específicas. Evidentemente, este proceso de composición puede ampliarse tanto como se desee, aumentando proporcionalmente las alternativas y recursos de desarrollo.

Esta composición de elementos deriva en la construcción de un árbol, de forma similar a la estructura del DOM en desarrollos más orientados al entorno web, lo cual está íntimamente relacionado con la lógica de renderizado de Flutter.

Recuperando le definición de Widget expuesta previamente, estos representan descripciones inmutables de una sección de la interfaz de usuario. En conjunto, conforman el árbol de widgets de una determinada aplicación, abarcando tanto los aspectos funcionales como los de configuración.

En el momento en el que se decide ejecutar la aplicación en cuestión, surge un nuevo árbol, denominado árbol de elementos, que actúa como nexo entre los componentes de configuración (árbol de widgets) y su representación real en la interfaz de usuario. Haciendo un símil con funcionalidades propias de lenguajes como *C*, el árbol de elementos actuaría de forma parecida a los punteros, sirviendo como referencias lógicas de los verdaderos componentes funcionales.

En base a esta última descripción, los componentes gráficos de cada uno de los widgets (es decir, lo que en definitiva se representa en la interfaz y pantallas de los usuarios) dan forma al denominado árbol de renderización, el cual se encarga de gestionar la disposición y características de diseño de los mismos, además de gestionar los inputs provenientes de los usuarios.

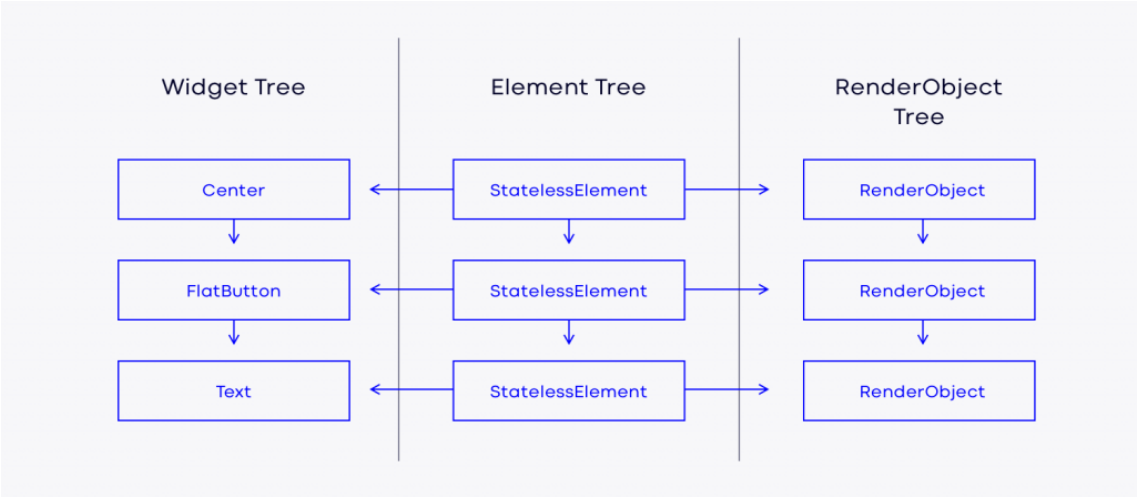

Figura 8. Estructura Widget, Element y Render tree (Fuente*: Londhe, O., Medium)*

De forma simplificada, bajo la arquitectura funcional de Flutter, los usuarios interactúan con el árbol de renderización, el cual, a través del árbol de elementos, traslada esta información al árbol de widgets. Este concentra los componentes que sustentan la lógica funcional de la aplicación, lugar donde finalmente se producen los procesos de reconstrucción.

Este flujo de acciones da pie de forma directa a establecer una definición general del ciclo de vida de un widget. Cabe destacar que el ciclo de vida que se expone a continuación corresponde a un *StatefulWidget*, pues, en base a lo comentado anteriormente, son aquellos que presentan una naturaleza mutable y, por consiguiente, cuentan con un ciclo de vida más completo.

A la hora de explicar las diferentes fases y componentes que conforman el mencionado ciclo de vida, nos apoyaremos en el diagrama mostrado en la Figura 9[2]:

[2] ALBERTENGO, G., & WANG, Y. (2022). Review and testing of plugins in Flutter for Android and IOS.

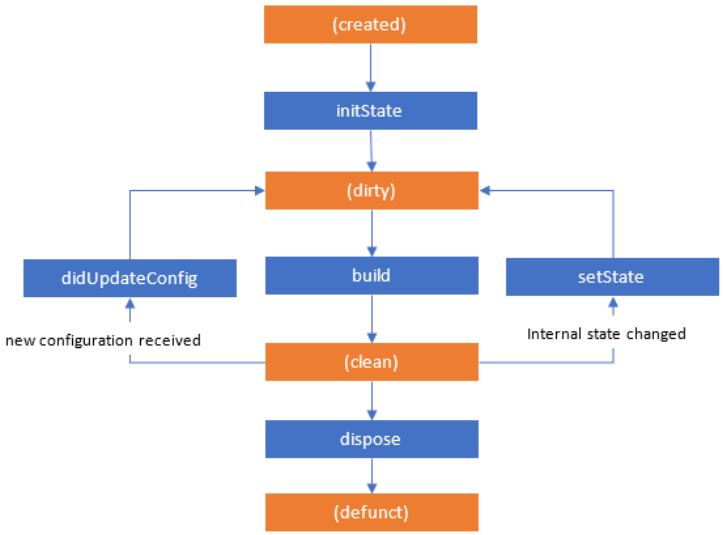

Figura 9. Diagrama del ciclo de vida de un `StatefulWidget` (Fuente*: Londhe, O., Medium)*

Como punto de partida, la creación de un `StatefulWidget` en Flutter conlleva la creación de un objeto `State`, en el cual se definen los diferentes estados mutables del widget en cuestión. Esta acción se lleva a cabo a través del método `createState()`, y su ejecución finaliza con la asignación de un `BuildContext` al mencionado `State`, que, proporciona la información correspondiente a la posición de este nuevo widget dentro del árbol de widgets.

Una vez creado el widget (junto con su correspondiente `State`*)* a través de su respectivo constructor, se realiza una **única** llamada al método `initState()`. Este método se emplea para definir los datos iniciales con los que el widget ha de contar para su correcto funcionamiento, entre los que se incluyen los correspondientes a su contexto (`BuildContext`), aquellos que dependen de sus componentes "padre" en el árbol de widgets y, finalmente, los provenientes de vinculaciones lógicas con otros widgets (como por ejemplo `ChangeNotifiers`, que se explicará en mayor detalle en el apartado de descripción de la aplicación prototipo).

Justo a continuación de la llamada a `initState()` se produce la ejecución del método `didChangeDependencies()`, el cual se ejecutará de forma automática cada vez que se produzca una modificación en alguno de los elementos de los que depende el widget en cuestión. En caso de que se dé esta circunstancia, se disparará el método `build()`, procediendo así a la reconstrucción y actualización del componente afectado.

Dentro de esta gestión de dependencias se incluye igualmente la posibilidad de un cambio en la configuración del widget, lo cual normalmente se debe a que el componente padre pasa una nueva variable a este a través del constructor. En estos casos se ejecuta el método `didUpdateWidget()`, actuando de forma similar al previamente citado `didChangeDependencies()`.

De igual modo, lo lógico es que los cambios dentro del widget provengan, además de como respuesta a dependencias con terceros, de la propia ejecución de la aplicación e interacción de los usuarios. Esta circunstancia se maneja a través del método `setState()`, empleado indistintamente tanto por el propio framework a nivel interno como por los desarrolladores, a la hora de habilitar la lógica funcional de la aplicación. Básicamente, `setState()` se emplea para notificar una modificación en los datos que desencadena una reconstrucción del widget situado en el `BuildContext` afectado, permitiendo adaptar la interfaz al nuevo escenario generado.

Finalmente, en la fase final de este ciclo de vida se produce la eliminación del widget, circunstancia que ocurre en el momento en el que se elimina permanentemente el elemento `State` del mismo. En este caso se recurre al método `dispose()`, que suele emplearse para eliminar vinculaciones (como por ejemplo la suscripción a las modificaciones) con otros widgets del árbol.

A modo de indicación, este último escenario se produce cuando se elimina permanentemente el elemento `State`, tal y como se ha expuesto previamente, pero existe la posibilidad de que este elemento solo se elimine temporalmente. Esta situación tiene lugar cuando `State` es eliminado del árbol, pero se prevé su reinserción inminente de nuevo en este último. Esta mecánica, aunque de uso marginal, se gestiona mediante el método `deactivate()` y es necesaria puesto que es posible mover el objeto `State` de un punto a otro del árbol en un momento dado.

DESARROLLO DE LA APLICACIÓN PROTOTIPO

El objetivo principal marcado para el presente proyecto consiste en establecer una perspectiva completa de Flutter como herramienta de desarrollo de interfaces y aplicaciones multiplataforma. A fin de dar respuesta a esta premisa, y buscando con ello evidenciar las características principales de un desarrollo tecnológico basado en Flutter y Dart, se plantea la construcción de una aplicación a modo de prototipo funcional.

Descripción de la herramienta de partida

Para llevar a cabo este proceso, se toma como referencia la aplicación Android desarrollada como trabajo evaluable en la asignatura de *Aplicaciones Nativas e Híbridas*, perteneciente al currículo del *Máster en Desarrollo Ágil de Software para la Web* y propuesto durante el curso 2020/2021. A grandes rasgos, el propósito principal de esta práctica consistía en la elaboración de una aplicación de gestión de películas, a través de la cual los usuarios pudieran establecer un registro de las películas que han visto recientemente, junto con la fecha y lugar de visionado.

A fin de concretar en mayor medida estos aspectos, los requisitos funcionales y no funcionales de la aplicación, así como sus posibles restricciones, quedan reflejados en la siguiente especificación:

> *"Se propone como trabajo, la creación de una aplicación para gestionar y llevar el control de aquellas películas que un usuario ha ido a ver al cine. Deberán gestionarse al menos los siguientes datos sobre cada película: Título, Actor principal, Fecha de visionado y ciudad donde se fue a ver. Adicionalmente, se podrán añadir otros datos que el alumno considere.*
>
> *La aplicación tendrá, al menos, tres funcionalidades: añadir una nueva película, borrar una película y consultar o listar películas previamente creadas. Igualmente se podrán añadir otras operaciones adicionales.*
>
> *Para añadir una nueva película deberá hacerse mediante una consulta a un servicio externo (como, por ejemplo, OMDBAPI http://www.omdbapi.com/), el cual permitirá obtener un listado de posibles películas, así como sus datos (utilizando JSON).*

> *La aplicación deberá ser correcta desde el punto de vista de la usabilidad: buena elección de colores, textos, etc. mantener informado al usuario en todo momento, evitar errores inesperados, etc. El enunciado de este ejercicio es suficientemente abierto por lo que el alumno puede elegir añadir diferente funcionalidad y/o datos según lo considere."*

Cabe destacar que, a este respecto, no resulta especialmente interesante realizar una descripción de dicha solución a nivel funcional, pues dada la naturaleza de ambos entornos de desarrollo (Android Studio y Flutter) y sus correspondientes lenguajes de programación (Java y Dart), sería complicado establecer paralelismos entre ambas soluciones, más allá de los objetivos funcionales que persiguen, que son los que, en definitiva, se han replicado.

Descripción detallada del prototipo Flutter

Hasta este punto, a lo largo del presente documento se ha llevado a cabo una exposición detallada de las características básicas de Flutter y Dart desde una perspectiva teórica. En este apartado, de forma complementaria a ello, se pretende completar esta visión funcional de Flutter desde el punto de vista del desarrollo tecnológico.

Antes de comenzar con la propia descripción de la solución planteada, resulta conveniente realizar algunas indicaciones que permitan contextualizar el marco de desarrollo sobre el que se sitúa el proyecto. En relación con esto último, los recursos tecnológicos empleados en la construcción de la aplicación son, por un lado, *Visual Studio Code* (*VSCode*) como entorno de desarrollo y, por otro lado, *Firebase* como arquitectura de backend para proporcionar el soporte necesario a los flujos de actividad de la aplicación.

El empleo de *VSCode* como IDE (*Integrated Development Environment)* se justifica debido a que, más allá de cierta preferencia personal, es el IDE más popular entre los desarrolladores de Flutter. Aunque inicialmente esto no parezca un factor diferencial, la presencia e influencia de una comunidad numerosa y prolífica puede afectar muy positivamente al proceso de desarrollo, retroalimentándose a través del descubrimiento y resolución de problemas y generando nuevo contenido permanentemente. Esto enriquece de forma directa el ecosistema general, lo cual se refleja, por ejemplo, en la existencia de más de 200 plugins de Flutter dentro de la oferta funcional de *VSCode*.

Desde el apartado técnico, *VSCode* también aporta ciertos beneficios al desarrollo de aplicaciones e interfaces con Flutter, pues en definitiva es un IDE muy ligero, permitiendo su despliegue sobre casi cualquier sistema y garantizando un rendimiento notable, siempre y cuando el desarrollador no se exceda en la instalación de extensiones.

Firebase, por su parte, es una arquitectura backend apoyada sobre el núcleo tecnológico de Google. Estructurada en un formato de *BaaS* (*Backend as a Service*), *Firebase* pone a disposición

de sus usuarios una panoplia de herramientas alojadas en la nube y destinadas a la construcción, optimización, monitorización y escalado de aplicaciones.

El modelo de desarrollo convencional precisa de la construcción, prácticamente en paralelo, de la arquitectura frontend y backend, a fin de ofrecer una solución completa y funcionalmente viable. La integración de los servicios de *Firebase* en el desarrollo de aplicaciones rompe con dicho modelo, automatizando gran parte del mencionado apartado de backend y permitiendo que los desarrolladores focalicen sus esfuerzos en el desarrollo funcional de la aplicación y la consecuente experiencia de usuario.

Esto conlleva también un cambio en el propio funcionamiento de los sistemas, pues bajo el modelo convencional, el frontend únicamente lanzaba llamadas a las APIs expuestas por el backend, y era en esta parte servidora donde se llevaba a cabo el procesamiento de las peticiones en cuestión. Haciendo uso de *Firebase*, esta capa de backend se difumina, manteniendo en el cliente todo el peso de la ejecución.

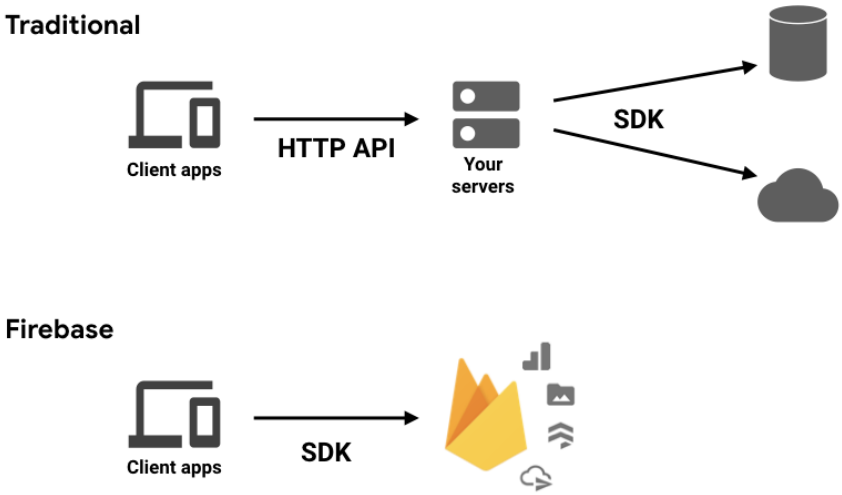

Figura 10. Introducción *Firebase* en estructura backend (Fuente: *Stevenson, D., Medium*)

La oferta de servicios que presenta este stack tecnológico es considerablemente amplia, abarcando funcionalidades de autenticación, base de datos, métricas, testing o mensajería, tal y como se refleja en la Figura 11. De cara al desarrollo del presente prototipo, dada la naturaleza y el alcance del mismo, se hará uso del servicio de base de datos en tiempo real: *Firebase Realtime Database*.

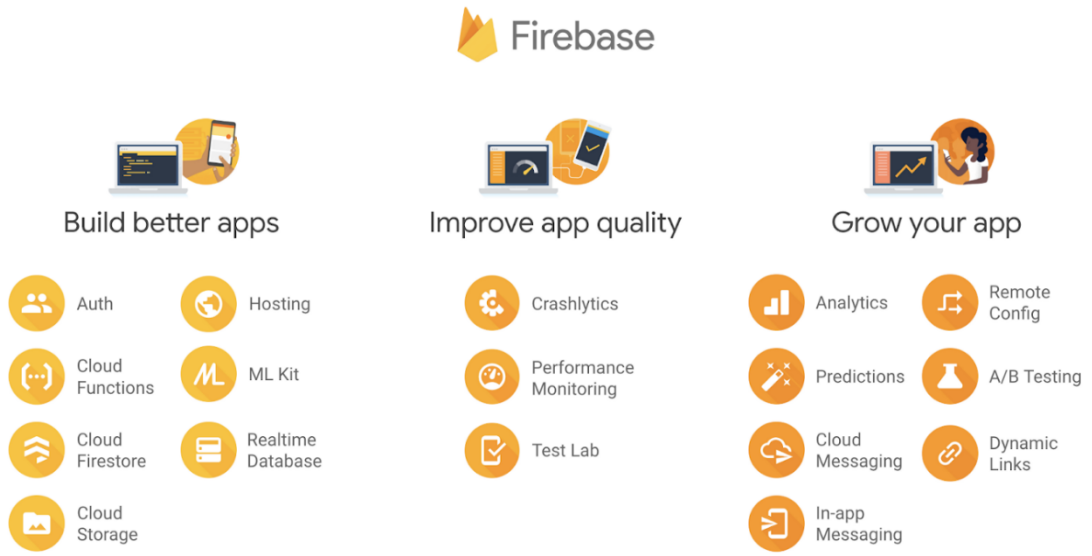

Figura 11. Oferta de servicios Firebase (Fuente: *Stevenson, D., Medium*)

Firebase Realtime Database[3] es una base de datos NoSQL alojada en la nube, donde los datos se almacenan en formato JSON y se sincronizan en tiempo real con cada uno de los clientes conectados a la misma. Como valor añadido, este servicio se adapta perfectamente a un desarrollo multiplataforma, puesto que en estos casos se proporciona una misma instancia de la base de datos para todos los consumidores de esta, recibiendo automáticamente actualizaciones con los datos más recientes independientemente del ecosistema subyacente en cada uno de ellos.

Finalmente, tal y como se comprobará a continuación, para la descripción de las interfaces de la aplicación prototipo se recurre a una representación jerárquica del árbol de widgets correspondiente a cada una de ellas, buscando con esto clarificar su estructura interna y contexto funcional.

Como se comentó anteriormente, la generación de este árbol de widgets es un mecanismo intrínseco de Flutter, a través del cual este gestiona su lógica funcional y los respectivos ciclos de vida de cada uno de los elementos que lo conforman. Igualmente, este "diagrama" interno de dependencias se hace accesible para el desarrollador, gracias a las denominadas *Dart DevTools.*

Dart DevTools[4] son un conjunto de herramientas orientadas a la depuración y monitorización del rendimiento para el ecosistema de Dart y Flutter que permiten, por ejemplo, la inspección del diseño o la gestión de la memoria. Concretando en el marco funcional de nuestro proyecto, este paquete de herramientas cuenta con una funcionalidad denominada *Widget Inspector*, que, durante la fase de depuración, permite inspeccionar a través de una interfaz gráfica el diseño

[3] https://firebase.google.com/docs/database
[4] https://dart.dev/tools/dart-devtools#:~:text=Dart%20DevTools%20is%20a%20suite,Flutter%20web

lógico de un determinado widget, inspección que se realiza a través de su correspondiente árbol de widgets.

Una vez expuestos estos conceptos generales, es posible profundizar en la descripción técnica de la solución elaborada, comenzando inicialmente desde una perspectiva a más alto nivel para, a continuación, centrarse directamente en los aspectos y estructuras funcionales de esta.

Siguiendo este planteamiento, se considera interesante, en primer lugar, abordar la estructura general de proyecto, tanto la que presenta Flutter por naturaleza, como su evolución, a la que se ha llegado como consecuencia del desarrollo del proyecto en cuestión. A este respecto, en la Figura 12 se recoge la distribución de directorios del proyecto en el momento de su creación, correspondiente por tanto a la estructura genérica de todo proyecto Flutter.

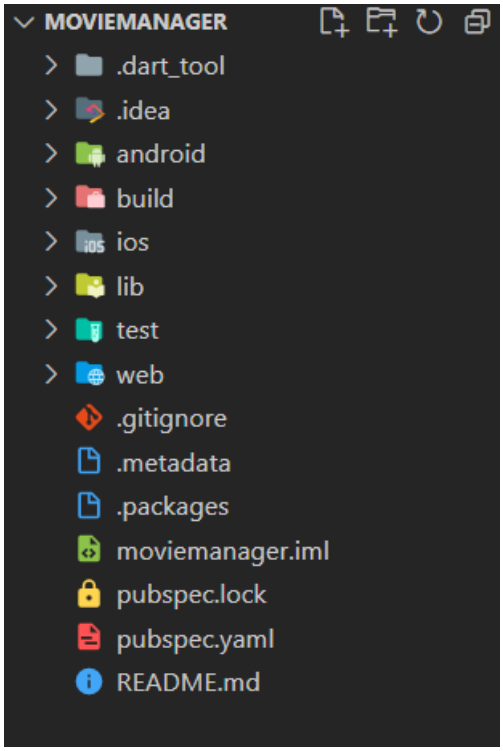

Figura 12. Estructura de directorios inicial

Analizando más en detalle la imagen superior, conviene destacar especialmente 4 de los directorios y archivos que, de forma automática, se han generado como consecuencia de la creación del proyecto***: android, ios, lib*** y ***pubspec.yaml***.

En primer lugar, y representando un aspecto importante de la lógica de funcionamiento de Flutter, encontramos los directorios *android* e *ios*. La función de estos consiste en habilitar la ejecución de una misma aplicación Flutter (y, por consiguiente, un mismo código Dart, recordando lo mencionado en apartados anteriores) en los diferentes ecosistemas, otorgando con ello la naturaleza multiplataforma a la herramienta.

Más concretamente, el directorio *android* contiene un proyecto Android compuesto por una única actividad, encargada de ejecutar la aplicación Flutter dentro de esta plataforma. De forma idéntica, el directorio *ios* presenta esta misma mecánica, en este caso con un proyecto iOS.

Por otro lado, el directorio *lib* contiene el código fuente que se elaborará progresivamente durante el proceso de desarrollo de la aplicación y que se empleará para su construcción, estructurándose en diversos en archivos de tipo *.dart*. En un primer momento, esta carpeta se encuentra totalmente vacía, a excepción del fichero *main.dart*, el cual representa el punto de entrada y el núcleo funcional de la ejecución de la aplicación.

Por último, el fichero *pubspec.yaml* es el archivo de configuración del proyecto Flutter generado. Este archivo se emplea principalmente para gestionar las dependencias del proyecto con librerías o extensiones de terceros.

Prosiguiendo con la estructura de proyecto, el proceso de desarrollo conlleva, evidentemente, el crecimiento y evolución del directorio *lib*, pues tal y como se ha mencionado antes, es el que concentra todo el código fuente de la aplicación. Como consecuencia de este factor, en la Figura 13 se representa la estructura final del directorio *lib* en el momento de la finalización del proyecto.

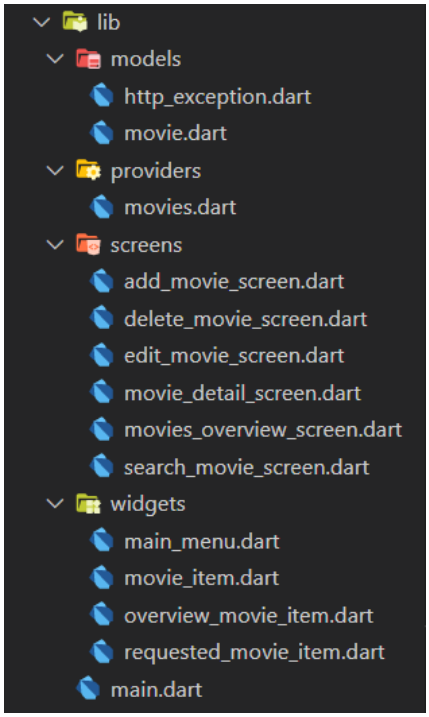

Figura 13. Estructura de directorios final

Como podemos apreciar, el directorio *lib* queda ahora subdividido en 4 directorios diferentes, que tienen como fin estructurar y categorizar (a nivel lógico) los contenidos generados en base

a su naturaleza funcional dentro del marco del proyecto. Esta división sigue un criterio similar al establecido por el patrón MVC (*Modelo-Vista-Controlador)*.

A grandes rasgos, el patrón MVC permite establecer un diseño lógico para la aplicación, distribuyendo sus diferentes componentes a través de un sistema multicapa que se divide en una primera capa de presentación o **vista** (que contiene la propia interfaz con la que interactúa el usuario), una capa de lógica de negocio o **controlador** (que se encarga de gestionar los principales procesos funcionales), y una última capa que recoge el **modelo** de datos (entidades) necesario para la ejecución de los flujos de actividad. Esta separación entre los diferentes componentes de la aplicación facilita especialmente su mantenimiento y escalabilidad general, pues al tener áreas definidas, es más sencillo lidiar con las problemáticas asociadas a cada contexto específico.

Realizando, por tanto, un paralelismo entre dicho patrón y la estructura final de nuestro proyecto, los directorios *widgets* y *screens* corresponderían a la capa de vista, el directorio *models* a la capa de modelo y el directorio *providers* a la capa de controlador. En base a esta clasificación funcional, estamos en disposición de realizar una descripción técnica detallada de la aplicación, a través de las mencionadas capas y sus respectivos componentes:

- ***Models**:* como se ha indicado en el párrafo anterior, en este directorio se localizan las entidades que componen el modelo de datos de la aplicación, siendo estas en concreto las clases *http_exception.dart* y *movie.dart.*

  - ***Http_exception.dart***: la existencia de esta clase se debe a la necesidad de proporcionar un mecanismo de notificación de excepciones personalizado, que pueda servir al desarrollador para lanzar excepciones con descripciones más específicas y adaptadas a las mecánicas concretas de esta aplicación.

    A nivel funcional, se trata de una entidad compuesta por un único atributo (la descripción personalizada de la excepción a disparar) y su correspondiente método `toString()` como se puede ver en la figura siguiente.

```dart
class HttpException implements Exception {
  final String message;

  HttpException(this.message);

  @override
  String toString() {
    return message;
  }
}
```

Figura 14. Estructura interna *http_exception.dart*

  - ***Movie.dart:*** a efectos prácticos, se trata de una clase "plana", compuesta únicamente por los diferentes atributos que conforman la información de una

película (título de la película, actores principales, valoración, fecha y lugar de visionado y póster promocional en formato de url) y su correspondiente constructor.

Igualmente, en la Figura 15 se puede apreciar que dentro de los mencionados atributos se incluye el atributo `id`. Esto se debe a que cuando se registra un dato en la base de datos de *Firebase*, esta automáticamente le asigna un valor de `id`, aunque dicho parámetro no esté definido inicialmente en la estructura de datos a registrar.

Por tanto, de cara a la lectura desde BBDD, el modelo de datos ha de estar preparado para recibir este parámetro.

```
3   class Movie with ChangeNotifier {
4     final String id;
5     final String title;
6     final String mainActor;
7     final String city;
8     final String imageUrl;
9     final String viewed;
10    final double rating;
11
12    Movie(this.id, this.title, this.mainActor, this.city, this.imageUrl,
13        this.viewed, this.rating);
14  }
```

Figura 15. Estructura interna *movie.dart*

- ***Providers:*** el nombre escogido para el presente directorio se debe a dos razones complementarias entre sí. Por un lado, al tratarse de la capa de controlador, los elementos bajo este directorio serán los encargados de proveer al resto de la aplicación de la lógica de negocio correspondiente a cada dominio funcional. Por otro lado, la gestión de la lógica de negocio para este proyecto en concreto está íntimamente vinculada a la utilización del paquete *Provider*, que proporciona un mecanismo de gestión de estado, o más concretamente, de aquellos datos que propician un cambio de estado dentro del sistema.

  Este directorio se compone únicamente del fichero *Movies.dart*, puesto que, en base a los requisitos del proyecto, solo es necesario gestionar un dominio funcional: el correspondiente a la gestión de películas.

  - ***Movies.dart***: como su propio nombre indica, esta clase centraliza toda la lógica funcional destinada a la gestión integral de las películas, abarcando las funciones básicas de consulta, introducción, edición y eliminación de estas contra la base de datos, así como diversas funcionalidades de búsqueda.

    Profundizando a este respecto, todas estas mecánicas internas se gestionan a través de un elemento principal, una lista de objetos tipo `movie` que permite mantener un registro en tiempo real de las películas almacenadas en el sistema.

Como consecuencia de este planteamiento, esta lista de películas, junto con sus correspondientes métodos, ha de ser accesible por las diferentes interfaces de la aplicación, pues su estado interno (si está llena o vacía, la cantidad de películas que haya en cada momento, etc.) influye en mayor o menor medida en el estado de estas interfaces.

Esta transmisión de información entre componentes podría realizarse mediante el paso directo de variables a través de los constructores de los widgets implicados, pero esto, por el contrario, supondría establecer un vínculo lógico y funcional entre los mismos, lo cual generaría un acoplamiento innecesario, alterando la naturaleza funcional de dichos widgets.

A fin de evitar esta última circunstancia, se decide emplear el mencionado paquete *Provider*. Dicho paquete habilita un modelo similar a una publicación/subscripción, sin incurrir en las mencionadas vinculaciones funcionales entre componentes.

De un modo simplificado, dentro de esta lógica de funcionamiento podemos encontrar dos roles diferentes. Por un lado, el propio *Provider*, correspondiente a la entidad que centraliza la gestión del estado o dato en cuestión y que se encarga de ejecutar y publicar las modificaciones realizadas sobre dicha información. Por otro lado, podemos encontrar **n** *listeners*, es decir, widgets que se han subscrito a esta información y que, o bien solicitan la modificación de esta en respuesta a las interacciones de los usuarios, o bien precisan de su actualización en tiempo real para llevar a cabo sus procesos internos.

Desde la perspectiva de código, para que la clase *Movies.dart* actúe como *Provider* ha de heredar de `ChangeNotifier`, gracias a lo cual podrá realizar llamadas al método `notifyListeners()`, permitiendo así publicar los cambios generados y notificar a los widgets (en este caso, *listeners*) afectados. Por su parte, los diferentes *listeners* (Figura 16) pueden lanzar llamadas a las funciones relacionadas con la lista de películas (Figura 17) o definir una variable que almacene en tiempo real el estado de la información tras cada actualización (Figura 18).

```
final newMovie = Movie(
    json.decode(response.body)['name'],
    movie.title,
    movie.mainActor,
    movie.city,
    movie.imageUrl,
    movie.viewed,
    movie.rating);
movies.add(newMovie);
notifyListeners();
```

Figura 16. Notificación de cambios en la información por parte del Provider

```
final moviesData = Provider.of<Movies>(context);
```

Figura 17. Ejemplo de suscripción a la lista de películas

```
Provider.of<Movies>(context, listen: false).addMovie(_newMovie);
```

Figura 18. Ejemplo de acceso a funcionalidades de la lista (añadir nueva película)

Como último aspecto a destacar, en la Figura 18 podemos apreciar que en uno de los parámetros se indica `listen: false`. Esto se debe a que es posible que una determinada interfaz precise del acceso a las funcionalidades de la lista de películas (en este caso concreto, la función de añadir una nueva película) pero que, a su vez, no necesite conocer el estado de la lista en cuestión, por lo que decide conscientemente no quedarse "escuchando" los cambios sobre la misma.

- ***Widgets:*** en este punto damos el salto a la descripción de la capa de vista de la aplicación, subdividida, como se indicó previamente, en los directorios *Widgets y Screens*. Esta estructura se apoya, y a la vez permite ejemplificar, la lógica de composición que caracteriza al ecosistema Flutter.

  Tomando como referencia la descripción teórica de Flutter expuesta en apartados anteriores, la verdadera potencia funcional del framework surge de la progresiva combinación de widgets "primitivos" a fin de construir una nueva estructura personalizada que permita dar respuesta a una necesidad más concreta, adaptada a las especificaciones de cada proyecto.

  Apoyándonos en este planteamiento, durante el proceso de desarrollo suelen surgir combinaciones de widgets que se repiten en diversos puntos de la aplicación. De ser identificadas, estas pueden ser exportadas como un mismo elemento independiente y con entidad propia, lo que permite su replicación y reutilización a lo largo de las diferentes interfaces de la aplicación de una forma más orgánica. Esto, en definitiva, supone la creación de los nuevos widgets descritos en el párrafo anterior, quedando almacenados en el directorio *Widgets*.

  A modo de puntualización, en esta sección se describirá únicamente el apartado de la capa de vista correspondiente a este directorio de *Widgets*, mientras que los elementos correspondientes al directorio *Screens*, es decir, las interfaces finales del sistema, se describirán a continuación como una sección aparte. Esto se debe a que los elementos dentro de *Widgets* tienen una función más estructural a nivel de UI, mientras que las interfaces finales representan la simbiosis entre diseño y funcionalidad, por lo que se considera más interesante tratarlo de forma independiente.

  Retomando el hilo inicial, el directorio *Widgets* se compone de las siguientes estructuras de UI:

- ***Main_menu.dart***: se trata del widget empleado para la definición del menú hamburguesa que se utilizará en las diferentes interfaces de la aplicación, y presenta el diseño capturado en la Figura 19.

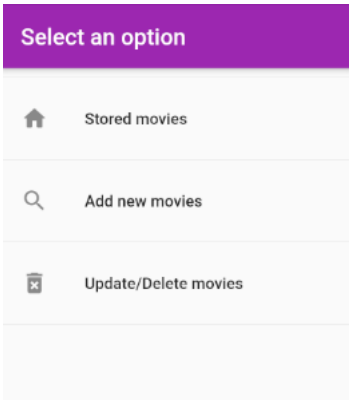

Figura 19. Diseño *main_menu.dart*

Profundizando a nivel de código, la pieza fundamental del presente widget es el elemento `Drawer`. Se trata de un widget perteneciente a la biblioteca estándar de Flutter, y proporciona la estructura básica de un menú hamburguesa.

```
Widget build(BuildContext context) {
  return Drawer(
    child: Column(
      children: [
        AppBar(
          title: Text('Select an option'),
          automaticallyImplyLeading: false,
        ), // AppBar
        Divider(),
        ListTile(
          leading: Icon(Icons.home_rounded),
          title: Text('Stored movies'),
          onTap: () {
            Navigator.of(context)
                .pushReplacementNamed(MoviesOverviewScreen.routeName);
          },
        ), // ListTile
        Divider(),
```

Figura 20. Estructura interna *main_menu.dart*

Sobre esta estructura básica se define una disposición de contenidos en formato de columna donde, en primer lugar, se incluye un elemento `AppBar` que permite incluir un título para el menú. A continuación de ello, las diferentes opciones de navegación dentro de este se definen a través de un elemento `ListTile`.

Este último widget permite la introducción de entre una y tres líneas de código dentro de un mismo "contenedor", pudiendo rodearlas de iconos u otros

widgets. En nuestro caso concreto, se incluye un pequeño icono que sirve como representación de la opción a elegir, junto con el título de la opción en cuestión.

Finalmente, se añade un último parámetro (`onTap`) a esta `ListTile` para gestionar la pulsación del usuario sobre la opción escogida, redirigiendo a este hacia la interfaz correspondiente. Esta misma estructura de `ListTile` se replica 3 veces, pues el menú consta de 3 opciones diferentes, y se emplea un elemento `Divider` en cada una de ellas para dibujar una pequeña línea que marque la división entre las opciones planteadas.

- ***Movie_item.dart***: a este respecto resulta conveniente realizar una pequeña indicación. Tanto este widget como los dos siguientes tienen como objetivo la representación de una película, pero su contexto de aplicación y la información que es necesario mostrar en cada uno de ellos difiere. Por lo tanto, se considera oportuno elaborar diferentes estructuras para adaptarse en mayor medida a los mencionados contextos.

  Este primer widget, siendo el más complejo de los tres, se emplea para la representación gráfica de la información de la película dentro de la interfaz *Delete Movie Screen* que se describirá a continuación. Este diseño se captura en la Figura 21.

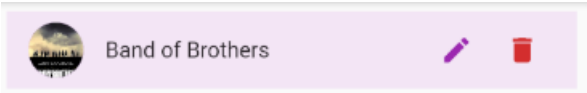

Figura 21. Diseño *movie_item.dart*

A nivel de código, en este caso se parte de un elemento `SizedBox` que permite la representación de contenidos dentro de un contenedor con las dimensiones de altura y anchura delimitadas. Dentro de dicho `SizedBox`, se representan de forma individual cada una de las películas almacenadas en la base de datos.

```
return SizedBox(
  child: ListTile(
    tileColor: Colors.purple[50],
    title: Text(title),
    leading: CircleAvatar(
      backgroundImage: NetworkImage(imageUrl),
    ), // CircleAvatar
    trailing: Row(
      mainAxisSize: MainAxisSize.min,
      children: [
        IconButton(
          onPressed: () {
            Navigator.of(context)
                .pushNamed(EditMovieScreen.routeName, arguments: id);
          },
          icon: Icon(Icons.edit),
          color: Theme.of(context).primaryColor,
        ), // IconButton
        IconButton(
          onPressed: () async {
            try {
              showDialog(
```

Figura 22. Estructura interna *movie_item.dart*

Para llevar a cabo esta renderización de contenidos se recurre, de nuevo, al widget `ListTile` descrito anteriormente. Buscando un diseño lo más amigable y funcional posible para los usuarios, se decide mostrar únicamente el póster de la película en cuestión junto con su título a través de este `ListTile`.

A diferencia del menú hamburguesa, para este widget en concreto se incluye, como elemento final de `ListTile`, un elemento de tipo `Row`, que permite representar sus widgets "hijo" en formato horizontal. Estos widgets hijo son, por consiguiente, dos elementos `IconButton` que representan las dos posibles acciones a realizar por parte del usuario: el borrado de la película (representado con un icono de papelera) y la edición de la misma (representado con un lápiz).

La diferencia entre `Icon` e `IconButton` reside en que estos últimos cuentan con el parámetro `onPressed`, a través del cual se permite gestionar la respuesta ante una pulsación del botón por parte de un usuario.

- ***Overview_movie_item.dart:*** este widget, por su parte, se aplica en la representación, dentro de la interfaz *Movie Overview Screen*, de cada una de las películas almacenadas en la base de datos de la aplicación. Su diseño se observa en la Figura 23.

Figura 23. Diseño *overview_movie_item.dart*

A nivel de código, la estructura básica del widget es proporcionada por el elemento `ClipRRect`, que envuelve los elementos subyacentes a este dentro de un rectángulo con las esquinas redondeadas. Ya dentro de este rectángulo, se recurre a un elemento `GridTile` para la representación de los contenidos, el cual, de forma similar a `ListTile`, permite una combinación de texto, iconos e imágenes, en este caso en formato vertical.

```
return ClipRRect(
  borderRadius: BorderRadius.circular(10),
  child: GridTile(
    child: GestureDetector(
      onTap: () {
        Navigator.of(context)
            .pushNamed(MovieDetailScreen.routeName, arguments: title);
      },
      child: Image.network(
        imageUrl,
        fit: BoxFit.cover,
      ), // Image.network
    ), // GestureDetector
    footer: GridTileBar(
      backgroundColor: Colors.black87,
      title: Text(
        title,
        textAlign: TextAlign.center,
      ), // Text
    ), // GridTileBar
  ), // GridTile
); // ClipRRect
```

Figura 24. Estructura interna *overview_movie_item.dart*

Al mencionado `GridTile` se le añade un elemento `GestureDetector`, que básicamente otorga al widget sobre el que se aplica la capacidad de gestionar diferentes tipos de pulsación por parte del usuario. Concretamente, el widget *overview_movie_item* detectará una pulsación simple del usuario sobre una de las películas, redirigiendo al mismo a la interfaz correspondiente del detalle de esa película en particular.

Además del citado `GestureDetector`, que es incluido más bien como añadido funcional, `GridTile` se compone de una imagen, que representa el póster de la película, y una `GridTileBar`, que permite incluir un `footer` en dicho `GridTile` para mostrar el nombre de la película.

- ***Requested_movie_item.dart:*** finalmente, este último widget permite representar la información de las películas que se obtienen como resultado de la petición a la API de omdb dentro de la interfaz *Search Movie Screen.* Su diseño se recoge en la Figura 25.

Figura 25. Diseño *requested_movie_item.dart*

A nivel de código, su estructura se define a través de un `ListTile` en el que se incluye tanto el título como el año de estreno de la película, junto con la imagen del póster de esta, a fin de facilitar la identificación de la película deseada por parte del usuario.

```
return Container(
  decoration: BoxDecoration(border: Border.all(color: Colors.white)),
  child: ListTile(
    tileColor: Colors.purple[100],
    title: Text(
      title,
      textAlign: TextAlign.center,
      style: TextStyle(fontWeight: FontWeight.bold),
    ), // Text
    leading: CircleAvatar(
      backgroundImage: NetworkImage(imageURL),
    ), // CircleAvatar
    trailing: Text(
      year,
      style: TextStyle(fontWeight: FontWeight.bold),
    ), // Text
  ), // ListTile
); // Container
```

Figura 26. Estructura interna *requested_movie_item.dart*

Cada uno de estos elementos `ListTile` se envuelven dentro de un elemento `Container`. Esta decisión se debe a que este último elemento permite incluir aspectos de diseño estético en el contenedor donde se encuentran los contenidos, valga la redundancia. En este caso concreto, se emplea para dibujar unas líneas blancas alrededor de cada uno de los `ListTile` que permiten diferenciar los diferentes resultados obtenidos.

El aspecto más destacable a este respecto es que, mientras que los demás widgets tratados hasta este punto se centraban en la renderización de elementos situados dentro del domino de la aplicación, en este caso se interpreta y renderiza de forma dinámica el contenido de la respuesta proporcionada por la mencionada API de películas.

Finalmente, en base a lo expuesto en la descripción del directorio *Widgets,* a continuación, se procede a la descripción de las diferentes interfaces que componen la aplicación. A modo de introducción de esta nueva sección, las mencionadas interfaces son las siguientes:

- ***Movies Overview Screen***
- ***Search Movie Screen***
- ***Add Movie Screen***
- ***Movie Detail Screen***
- ***Delete Movie Screen***
- ***Edit Movie Screen***

Movies Overview Screen

La funcionalidad principal de la interfaz *Movies Overview Screen* consiste en proporcionar al usuario una perspectiva general y actualizada de todas las películas registradas en el sistema. Para alcanzar la funcionalidad esperada, y a fin de clarificar la estructura interna de esta interfaz en cuestión, se obtiene el árbol de widgets expuesto en la Figura 27.

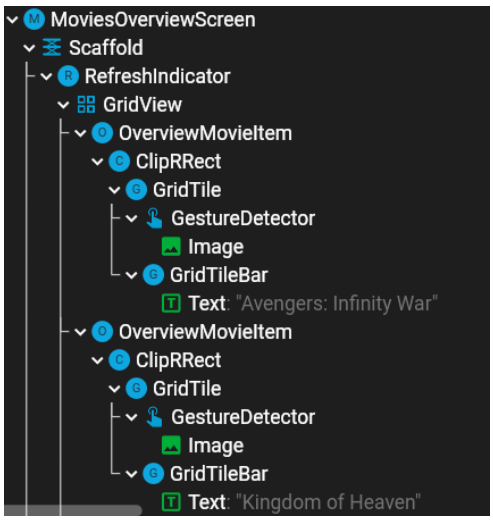

Figura 27. Árbol de widgets *MoviesOverviewScreen*

Analizando dicho árbol de widgets, es posible elaborar una descripción general de la interfaz desde la perspectiva de diseño. Atendiendo a la estructura jerárquica representada en la Figura 27, el primer elemento que encontramos y que, tal y como se comprobará a continuación, es recurrente en todas las interfaces de la aplicación, es el elemento `Scaffold`.

`Scaffold` representa el widget básico de toda aplicación Flutter. Este ocupa todo el espacio de la interfaz de usuario y proporciona al desarrollador todos los elementos funcionales necesarios para la construcción de una aplicación responsiva de propósito general. En resumen, podría decirse que establece el lienzo sobre el que se construye una aplicación Flutter.

Entrando ya en la estructura propia de esta interfaz más allá del `Scaffold`, esta queda envuelta por un elemento `RefreshIndicator` que permite al usuario, a través de un scroll vertical, actualizar manualmente la pantalla, haciendo así efectivos los posibles cambios que se hayan podido producir sobre la lista de películas.

Las diferentes películas almacenadas en la base de datos (representadas a través de sendos widgets de tipo *overview_movie_*item, descritos anteriormente) se estructuran dentro de esta interfaz a través del elemento `GridView`. Este widget representa un array scrolleable de widgets, los cuales, en nuestro caso concreto, se han estructurado en dos columnas paralelas mediante la modificación del parámetro `crossAxisCount`, ya que el valor asignado a este parámetro determinará el número de columnas en las que se distribuirá el contenido.

A nivel funcional, la interfaz *Movie Overview Screen* se define como un *Stateful Widget*, puesto que su estado general depende en todo momento de las películas almacenadas en la base de datos. A este respecto, como parte de su método `initState()`, se realiza una llamada al *Provider* de películas (*movies.dart)* a fin de obtener, desde antes incluso de renderizar los contenidos, la lista de todas las películas registradas en la base de datos. El scroll de actualización manual del usuario dispara este mismo procedimiento.

Search Movie Screen

La funcionalidad principal de la interfaz *Search Movie Screen* es la de proporcionar al usuario un punto de interacción a través del cual pueda realizar una búsqueda de las películas que desee registrar, recurriendo para ello a la API expuesta por *omdbapi*. Para alcanzar la funcionalidad esperada, y a fin de clarificar la estructura interna de esta interfaz en cuestión, se obtiene el árbol de widgets expuesto en la Figura 28.

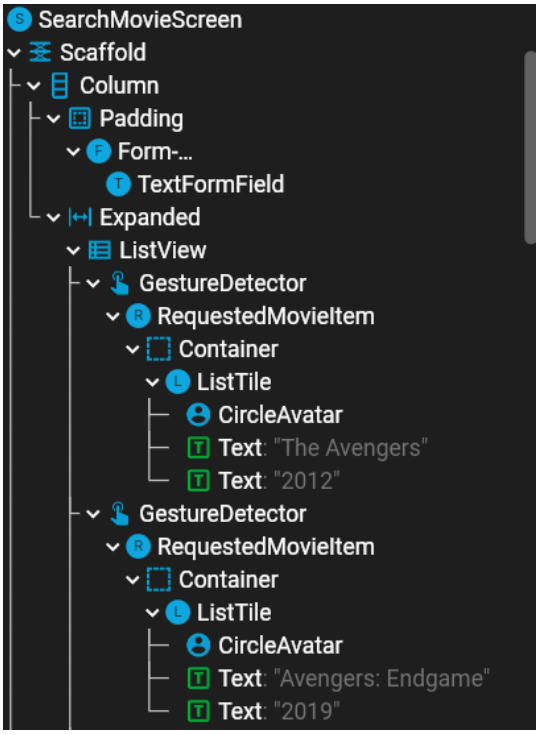

Figura 28. Árbol de widgets *SearchMovieScreen*

Desde la perspectiva de diseño, la disposición de los contenidos se estructura en formato vertical mediante la utilización del elemento `Column`. Profundizando en esta disposición en columna, la interfaz posee dos secciones claramente diferenciadas.

- Por un lado encontramos, envuelto por un elemento `Padding` que simplemente permite ajustar sus dimensiones, un cuadro de texto que cumple con la función de formulario de datos. A través de este formulario se gestiona la introducción del título de la película por parte del usuario, lanzando la llamada a la API de *omdb* en caso de una correcta introducción de la información o, en caso contrario, informando al usuario de cualquier error que haya podido surgir durante el proceso.
- Por otro lado, justo a continuación de este cuadro de texto se encuentra la segunda sección de la interfaz. En ella se realiza la representación gráfica de cada uno de los resultados obtenidos tras la petición (widget *requested_movie_item)*. Estos resultados se estructuran dentro de un elemento `ListView`, lo que se traduce en una lista scrolleable de widgets ordenados linealmente.

Cada uno de los elementos que componen la `ListView` se complementan con un `GestureDetector`, convirtiendo dichos elementos en secciones clickables, lo que permite al usuario seleccionar una película en concreto.

A nivel funcional, esta interfaz vuelve a definirse como un *Stateful Widget*, pues el contenido de la misma depende del resultado de la petición a la API de *omdb*. En un primer momento la lista de resultados estará vacía, por lo que no se renderizará contenido alguno. Pero, tras la ejecución

de la petición, se produce la decodificación de la respuesta recibida con el fin de obtener los campos de información necesarios para renderizar los respectivos componentes de la interfaz, lo que evidentemente ha de propiciar una actualización del estado a través del método `setState()`.

Add Movie Screen

Una vez seleccionada la película que se desea registrar, la interfaz *Add Movie Screen* presenta ante el usuario una pantalla donde este puede introducir la fecha y lugar de visionado para la película en cuestión. Para alcanzar la funcionalidad esperada, y a fin de clarificar la estructura interna de esta interfaz en cuestión, se obtiene el árbol de widgets expuesto en la Figura X.

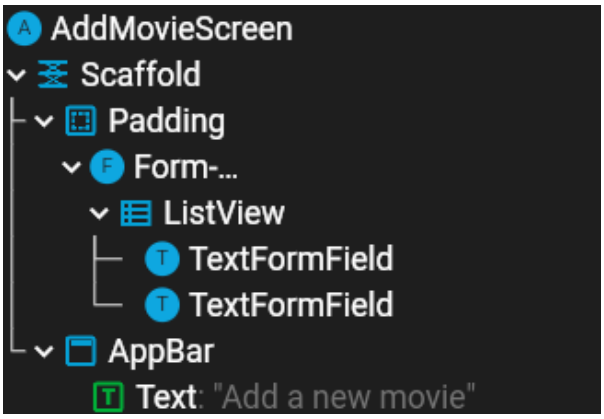

Figura 29. Árbol de widgets *AddMovieScreen*

A nivel de diseño, la interfaz presenta una estructura bastante sencilla, pues únicamente cuenta con dos cuadros de texto en formato de formulario a través de los cuales se produce la introducción de información por parte del usuario. Estos cuadros de texto se estructuran linealmente en formato vertical a través del empleo de una `ListView`.

Pasando a la perspectiva funcional, esta interfaz cuenta con algo más de profundidad. En primer lugar, al interactuar con el cuadro de texto correspondiente a la fecha de visionado se produce el despliegue de un calendario, sobre el que el usuario podrá seleccionar la fecha exacta en la que vio la película. Este mecanismo, aunque aparentemente trivial, supone el parseo de formatos entre el tipo `date` y el tipo `String`, lo cual en muchas ocasiones llega a complicarse en demasía.

En segundo lugar, retomando la estructura de discurso empleada en las anteriores interfaces, *Add Movie Screen* se define como un *Stateful Widget*. Esto es debido a que, además de la gestión del formulario como tal, de forma transparente para el usuario se produce una nueva consulta

a la API de *omdb*, la cual termina de proporcionar la información necesaria para la creación del objeto `movie`.

A través de la `ID` de *imdb* de la película seleccionada en la interfaz anterior, y que por consiguiente es trasferida como argumento a esta nueva interfaz, se genera una petición a *omdb* en busca de los actores principales y la valoración de la película, pues para acceder a dicha información han de emplearse parámetros diferentes a los de la petición anterior (la primera petición se hace a través del nombre de la película, y esta última a través de la `ID`, ya que en la API de *omdb* no existe una petición que devuelva toda esta información a través de una única llamada).

En consecuencia, en el momento en el que se reciben los últimos datos para la construcción de la película, estos han de almacenarse, y para que dicho almacenaje se haga efectivo es necesario guardar este nuevo estado de la interfaz y su correspondiente información a través, de nuevo, de `setState()`.

Movie Detail Screen

Tras haber llevado a cabo el registro de, al menos, una película dentro del sistema, se abre la posibilidad de acceder a la interfaz *Movie Detail Screen*, que como su propio nombre indica, permite al usuario consultar toda la información correspondiente a una determinada película. Para alcanzar la funcionalidad esperada, y a fin de clarificar la estructura interna de esta interfaz en cuestión, se obtiene el árbol de widgets expuesto en la Figura 30.

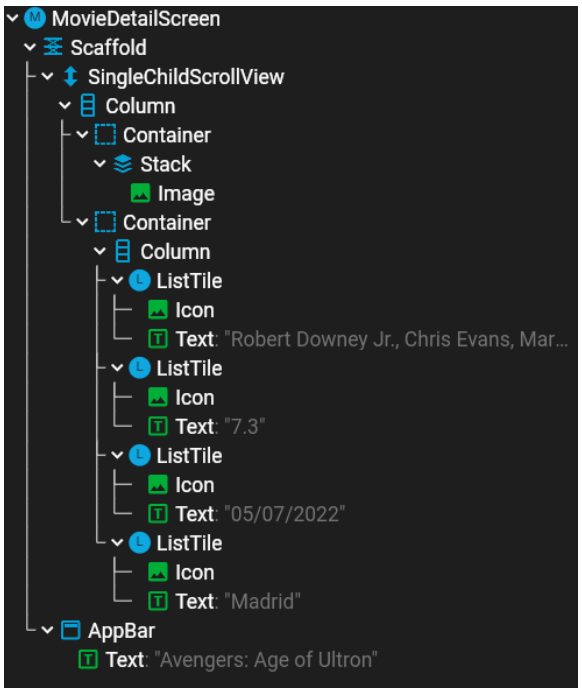

Figura 30. Árbol de widgets *MovieDetailScreen*

De nuevo, comenzando el análisis de la interfaz desde una perspectiva de diseño, la estructura del contenido se define a través de un elemento `SingleChildScrollView`, que actúa de forma similar a `ListView` pero presenta un funcionamiento más optimizado a la hora de gestionar widgets y tamaños diferentes, como es el caso.

A grandes rasgos, y tal y como se refleja en la imagen anterior, dentro de este `SingleChildScrollView` podemos encontrar dos tipos diferentes de widget, cada uno de ellos localizado dentro de un *Container* diferente.

El primero de estos *Container* se emplea para la representación del póster de la película, intentando ajustar al máximo tanto el tamaño de dicho contenedor como la resolución de la imagen, para mantener así la mayor calidad posible. El segundo *Container* se utiliza para la representación del resto de información de la película, es decir, sus actores principales, su valoración y su fecha y lugar de visionado, a través de sendos elementos `ListTile`. A su vez, estos elementos `ListTile` se componen de un icono representativo del atributo en cuestión, junto con la información correspondiente a dicho atributo.

Por último, a fin de completar esta descripción del diseño, el título de la película seleccionada se muestra como título de la propia pantalla a través del parámetro `title`, contenido en el elemento `AppBar`.

Desde la perspectiva funcional, la interfaz recibe el título de la película seleccionada por el usuario en *Movie Overview Screen*, el cual es empleado para realizar una búsqueda de la película dentro de la base de datos, previa llamada al *Provider*. Con ello, se pretende obtener toda la información correspondiente a la película en cuestión, para así proceder a su representación gráfica en los mencionados widgets.

A diferencia del resto de interfaces tratadas hasta este punto, *Movie Overview Screen* se define como un *Stateless Widget*, pues una vez obtenida la información de la película seleccionada, el contenido a mostrar será siempre el mismo, no siendo necesaria, por consiguiente, una gestión de estados.

Delete Movie Screen

La interfaz *Delete Movie Screen* pone a disposición del usuario una pantalla a través de la cual este pueda editar o borrar alguna de las películas almacenadas en el sistema. Para alcanzar la funcionalidad esperada, y a fin de clarificar la estructura interna de esta interfaz en cuestión, se obtiene el árbol de widgets expuesto en la Figura 31.

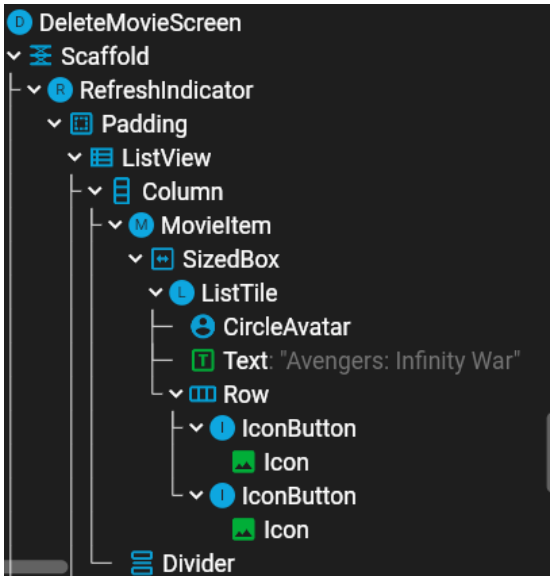

Figura 31. Árbol de widgets *DeleteMovieScreen*

El diseño de la presente interfaz está prácticamente monopolizado por el widget *Movie_item* descrito con anterioridad, pues fuera de ello la interfaz en sí únicamente se encarga de delimitar las dimensiones de cada uno de estos elementos a través de `Padding` y de establecer una disposición vertical de los mismos a través de los elementos `ListView` y `Column`.

Como funcionalidad añadida, toda esta estructura se envuelve dentro de un elemento `RefreshIndicator` que, al igual que en pasadas interfaces, permite al usuario realizar una actualización manual de la pantalla, disparando de nuevo la llamada al *Provider* para la obtención de la lista de películas en su última versión.

A nivel funcional, de nuevo nos encontramos ante un *Stateless Widget*, pues al igual que en el caso anterior, la estructura e información de esta interfaz no se ven afectadas por posibles modificaciones.

Edit Movie Screen

Finalmente, la última de las interfaces de la aplicación tiene como principal función otorgar al usuario la posibilidad de editar la información correspondiente a una determinada película. Para alcanzar la funcionalidad esperada, y a fin de clarificar la estructura interna de esta interfaz en cuestión, se obtiene el árbol de widgets expuesto en la Figura 32.

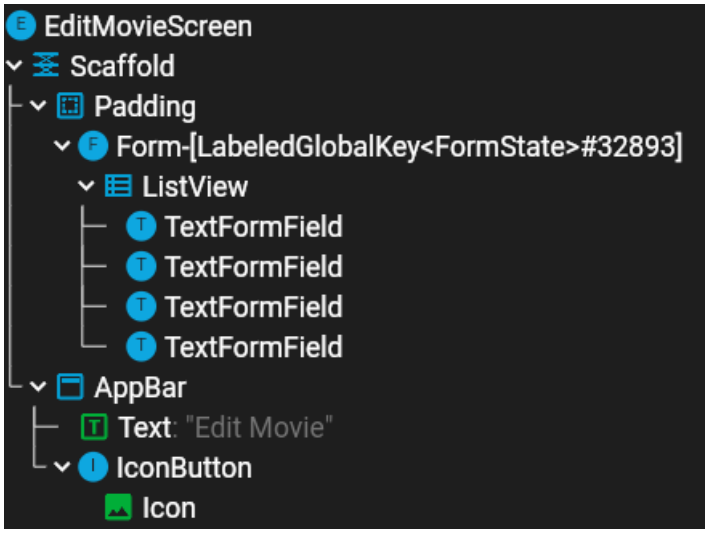

Figura 32. Árbol de widgets *EditMovieScreen*

Desde la perspectiva de diseño, más allá del elemento `Padding` que se emplea para una colocación correcta de los contenidos, el resto de la interfaz se compone de 4 campos de texto, (organizados linealmente a través de una `ListView`) usados para la representación los diferentes atributos de la película seleccionada, y que, en conjunto, conforman el formulario que se validará para llevar a cabo la modificación de la película en cuestión.

De los mencionados campos de texto, solo serán editables aquellos que corresponden a la información introducida por el usuario (fecha y lugar de visionada), pues en caso de modificar datos propios de la película, como podría ser el título, se alteraría la integridad del modelo de datos de la aplicación.

Como último aspecto a destacar desde el diseño, se incluye como parte de la `AppBar` un `IconButton` que, una vez realizadas las modificaciones deseadas en los correspondientes campos de texto, puede ser pulsado por el usuario para guardar dichos cambios, disparando así la validación del formulario.

Y para concluir, desde la perspectiva funcional la interfaz se define como un *Stateful Widget*, puesto que esta ha de manejar la nueva información introducida en el formulario y, con ella, construir de nuevo la película afectada, lo que altera su propia estructura de datos.

Main.dart

Pese a que no corresponde como tal a ninguna interfaz del sistema, no podemos concluir una descripción técnica de la aplicación sin abordar el rol y la lógica funcional que cumple la clase *main* dentro de la arquitectura general del proyecto.

A grandes rasgos, tal y como se indicó en apartados previos, esta clase proporciona el punto de entrada para la ejecución de la aplicación, centralizando e inicializando en ella los diferentes

aspectos funcionales y de diseño trasversales a toda la solución y que intervienen de forma constante en el workflow general de esta.

Concretando a este respecto, dentro de la clase *main* se inicializa el componente `MaterialApp`, a través del cual se definen las características principales del tema de la aplicación (gama de colores, fuente de letra, etc.), así como las rutas que permiten gestionar la navegación entre interfaces, indicando con ello la interfaz que actúa como *home*.

Por último, dentro de esta clase también se inicializan los diferentes *Providers* que intervienen en los flujos de actividad de la aplicación, en nuestro caso concreto únicamente el componente *Movies.*

```dart
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MultiProvider(
      providers: [
        ChangeNotifierProvider(
          create: (_) => Movies(),
        ) // ChangeNotifierProvider
      ],
      child: MaterialApp(
        title: 'MOVIES',
        theme: ThemeData(
          primarySwatch: Colors.purple,
          accentColor: Colors.deepOrange,
        ), // ThemeData
        home: MoviesOverviewScreen(),
        routes: {
          MovieDetailScreen.routeName: (ctx) => MovieDetailScreen(),
          MoviesOverviewScreen.routeName: (ctx) => MoviesOverviewScreen(),
          SearchMovieScreen.routeName: (ctx) => SearchMovieScreen(),
          AddMovieScreen.routeName: (ctx) => AddMovieScreen(),
          DeleteMovieScreen.routeName: (ctx) => DeleteMovieScreen(),
          EditMovieScreen.routeName: (ctx) => EditMovieScreen(),
        },
      ), // MaterialApp
    ); // MultiProvider
```

Figura 33. Estructura de código *main.dart*

Demostración workflow

A fin de completar esta descripción técnica de la aplicación Flutter desarrollada, resulta interesante realizar una breve demostración de lo que sería un caso de uso real por parte de un usuario, así como de los posibles escenarios que pueden llegar a tener lugar.

En primer lugar, en el momento en el que el usuario abra la aplicación aparecerá ante él la interfaz *Movie Overview Screen*, que inicialmente en esta primera interacción se encontrará totalmente vacía, tal y como se muestra en la Figura 34.

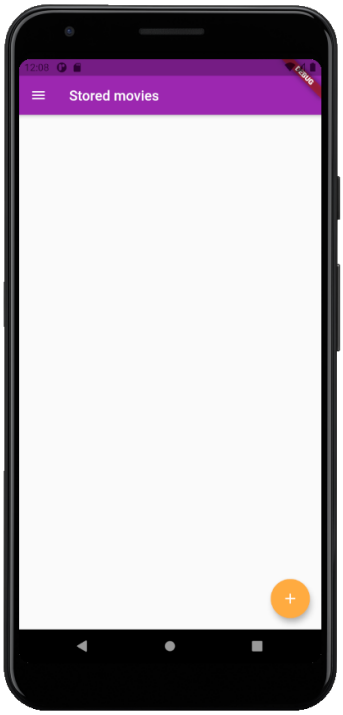

Figura 34. Estado inicial *MovieOverviewScreen*

A continuación, el usuario tiene dos opciones. La primera de ellas pasaría por pulsar el botón flotante situado en la zona inferior derecha de la pantalla, el cual le trasladaría directamente a la interfaz donde se realiza la búsqueda de películas (*Search Movie Screen)*. La segunda opción pasaría por acceder al menú hamburguesa de la zona superior izquierda y, desde ahí, seleccionar qué acción desea realizar, a elegir entre consultar las películas almacenadas, añadir una nueva película o eliminar/editar películas ya almacenadas.

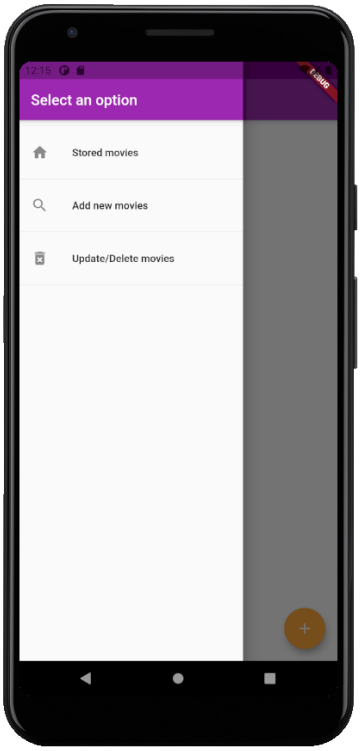

Figura 35. Despliegue menú hamburguesa

En este punto, asumimos que el paso más lógico sería optar por la opción de añadir una nueva película, ya que de momento el resto de las interfaces no ofrecerían funcionalidad alguna al no haber aún películas registradas en el sistema. Por lo tanto, y partiendo de esta premisa, el usuario llegaría a la pantalla de búsqueda de películas, donde podrían darse dos circunstancias.

En primer lugar, es posible que el usuario confirme el envío del campo de texto sin haber introducido ningún caracter. En este caso, el sistema le respondería automáticamente, informando de la necesidad de incluir el título de una película para proceder con la búsqueda.

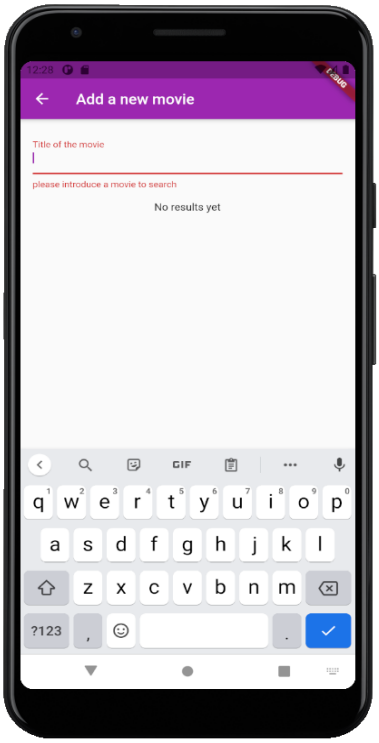

Figura 36. Notificación campo vacío al buscar película

En segundo lugar, o como respuesta a la anterior situación, el usuario puede introducir correctamente el título de una película, tras lo cual aparecerá una lista con aquellas películas/series que coincidan con el título proporcionado.

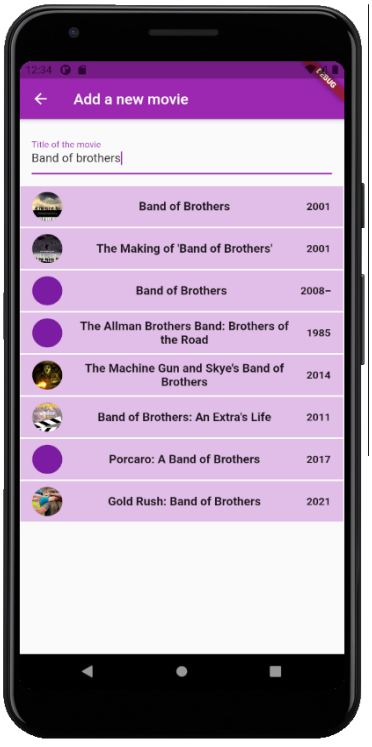

Figura 37. Resultado búsqueda de película

Una vez se muestre esta lista de opciones, el usuario procederá a seleccionar aquella película que responda a su criterio de búsqueda, tras lo cual será trasladado a la interfaz *Add Movie Screen*, donde este habrá de proporcionar una fecha y lugar de visionado para la película en cuestión.

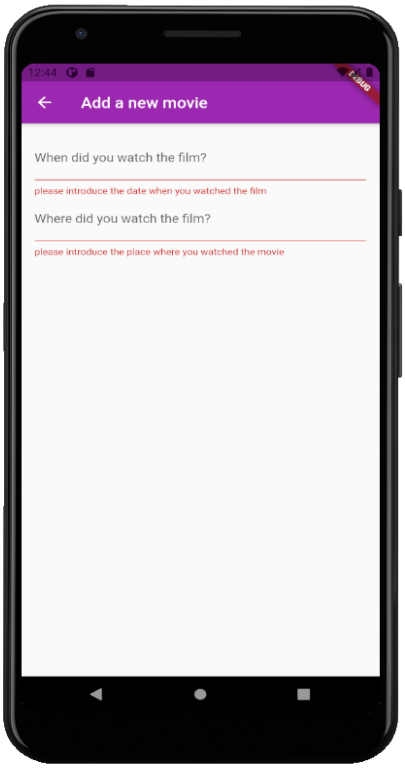

Figura 38. Notificación campos vacíos fecha y lugar visionado

De nuevo, se cuenta con medidas de contingencia a la hora de supervisar la introducción de datos por parte del usuario, notificando a este de manera similar al caso citado previamente si se produce una introducción errónea. Simplemente cabe destacar que, en esta interfaz concreta, al contar con 2 cuadros de texto, el contenido de ambos se evalúa de forma independiente, por lo que el "error" se disparará tanto si ambos campos están vacíos como si solo uno de ellos lo está.

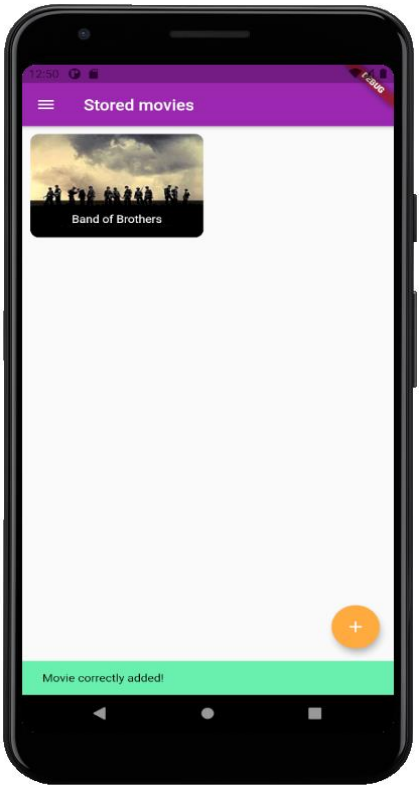

Figura 39. Interfaz principal tras inserción película

En caso de seleccionar correctamente una fecha e introducir correctamente el lugar de visionado, se aúna toda la información recabada durante el proceso y construye el objeto `movie` que se va a almacenar. Una vez almacenado, el usuario es redirigido de forma automática a la pantalla inicial (*Movie Overview Screen)*, incluyendo un mensaje de confirmación en la parte inferior de la pantalla. Como se aprecia en la Figura 39, esta interfaz ya no se encuentra vacía, sino que contiene un pequeño recuadro con el póster y título de la nueva película almacenada.

Como aspecto a destacar durante este proceso, existe la posibilidad de que un usuario proceda al registro de una película que ya se encuentre previamente registrada en la base de datos de la aplicación. Ante esta situación, en el momento del segundo registro de la mencionada película, se disparará un mensaje de error como el que se muestra en la Figura 40 que notificará al usuario esta problemática, procediendo a la cancelación del proceso de almacenaje en la base de datos.

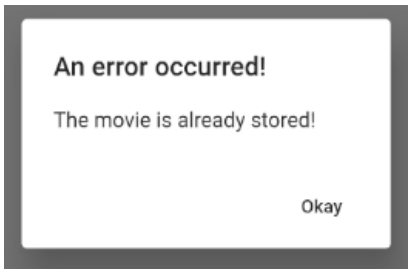

Figura 40. Mensaje de error película repetida

Prosiguiendo con este flujo de acciones, ahora que el sistema ya ha registrado una película, el usuario podría, desde esta interfaz inicial, acceder a la película en cuestión, lo que abrirá una pantalla con el detalle de esta (*Movie Detail Screen)*. Tal y como se ha expuesto en apartados previos, esta interfaz presenta de forma gráfica la información correspondiente a la película seleccionada, incluyendo el título de la película, el póster, los actores principales, la valoración y la fecha y lugar de visionado.

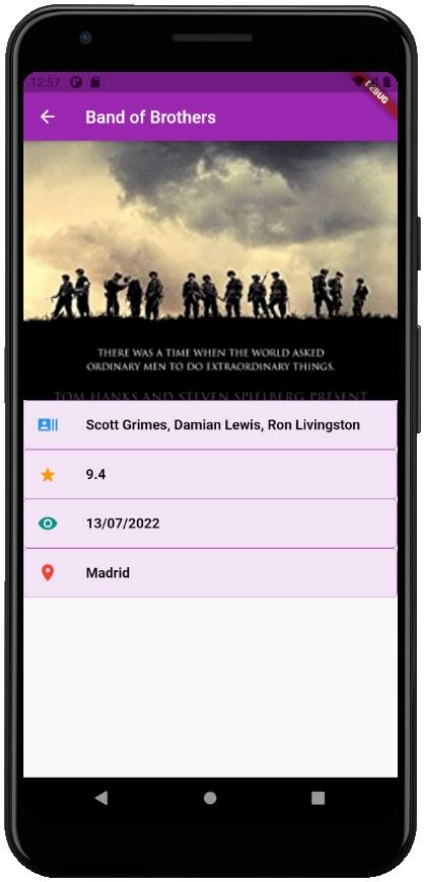

Figura 41. Interfaz *MovieDetailScreen*

Acercándonos ya al final de este primer ciclo de ejecución, el usuario podría en este momento tomar dos elecciones: **eliminar** la película o **editar** su correspondiente información. Para acceder a cualquiera de las dos opciones será necesario interactuar con el ya mencionado menú hamburguesa de la Figura 35 (*Despliegue menú hamburguesa*).

La selección de este apartado desde el menú hamburguesa conduce al usuario hasta la interfaz *Delete Movie Screen*, en la que, en formato más reducido, se muestran las diferentes películas almacenadas en el sistema, ***junto a las cuales aparecen las opciones de edición y eliminación***.

Figura 42. Interfaz *DeleteMovieScreen*

A fin de establecer una descripción de un ciclo completo, se asume que el usuario seleccionará la opción de editar la película, tras lo cual se abre la última de las interfaces, *Edit Movie Screen*. En esta pantalla se muestra la información correspondiente al título, actores principales y fecha y lugar de visionado, pero solo se permitirá la modificación de aquellos datos que fueron introducidos por el usuario, es decir, estos dos últimos (fecha y lugar de visionado).

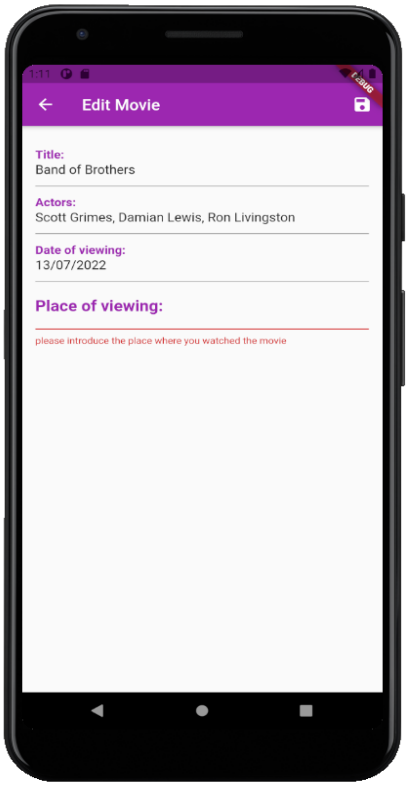

Figura 43. Notificación campo vacío en edición película

De forma idéntica a las otras situaciones donde se requiere la introducción de datos por parte del usuario, se supervisa dicho proceso a fin de evitar datos erróneos, notificando al usuario en caso de que así sea, como se puede ver en la Figura 43.

Figura 44. Actualización exitosa de la película

Tras realizar las modificaciones pertinentes y pulsar el botón de guardado, el usuario es de nuevo redirigido a la página principal, previa notificación del éxito en el proceso de modificación (Figura 44)

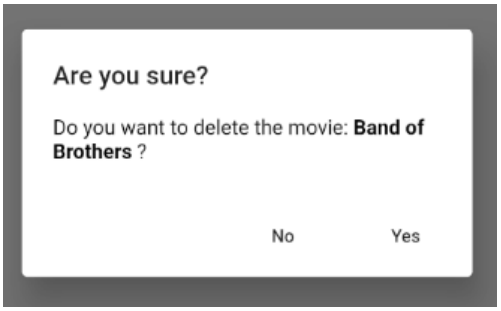

Figura 45. Solicitud confirmación borrado película

Como último paso, el usuario podrá acceder nuevamente a la sección de edición/eliminación de películas, procediendo finalmente al borrado de la película seleccionada. Antes que la película sea eliminada permanentemente del sistema, se solicitará al usuario la confirmación de dicha acción a través de un cuadro de diálogo, pues será un proceso irreversible (Figura 45)

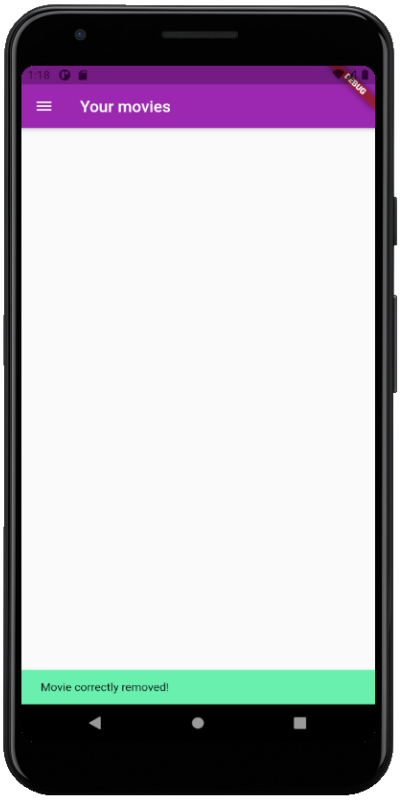

Figura 46. Eliminación exitosa de la película

En caso de que la respuesta a dicho cuadro de diálogo sea NO, la aplicación se mantendrá en su estado actual, mientras que si por el contrario el usuario confirma la eliminación definitiva de la película, se procederá a su borrado, notificando al usuario en el momento de la finalización del proceso. (Figura 46)

COMPARATIVA FLUTTER vs REACT NATIVE vs IONIC

Con la intención de contextualizar de forma completa la figura de Flutter dentro del ecosistema del desarrollo de aplicaciones multiplataforma, así como de resaltar sus aspectos más diferenciales con respecto a soluciones similares, se plantea una comparación general entre Flutter y dos de los frameworks más conocidos dentro de este ámbito: *React Native* e *Ionic*.

La elección de *React Native* para la presente comparación se debe a que es la herramienta de referencia en el mercado, la cual, como se comprobó previamente, durante los últimos años ha contado con el porcentaje de uso más alto entre los desarrolladores, hasta que fue superada este último año precisamente por Flutter. Además, desde el apartado funcional *React Native* presenta ciertas similitudes con el modelo de desarrollo propuesto por el ecosistema *Flutter*.

Por su parte, la elección de *Ionic* se apoya, por encima de todo, en la experiencia previa con dicho framework, pues fue utilizado como parte del currículo del *Máster en Desarrollo Ágil de Software para la Web*.

En consecuencia, a continuación se realizará una introducción teórica de ambos frameworks (*React Native* e *Ionic*) para, finalmente, concluir con la comparación final entre las diferentes herramientas en base a sus aspectos más destacables.

React Native

React Native (O'Reilly, 2017) es un framework JavaScript orientado al desarrollo de aplicaciones multiplataforma de renderización nativa en iOS y Android. Su estructura funcional parte de *React*, la librería JavaScritpt para la construcción de interfaces de usuario desarrollada por Facebook. A diferencia de esta última, centrada en los navegadores web, *React Native* se orienta hacia las plataformas móviles.

Dentro del ecosistema de *React*, tanto en la librería original como en *React Native*, las aplicaciones se codifican empleando JavaScript y XML, dando lugar a la sintaxis conocida como JSX. Al estar basado en un estándar web, *React Native* permite el desarrollo multiplataforma y, en el momento del despliegue final de la aplicación, es cuando se recurre al *JavaScript Bridge* descrito en apartados anteriores, renderizando los contenidos en formato nativo en los diferentes entornos.

Ionic

Ionic[5] es un kit de herramientas de interfaz de usuario de código abierto creado por Drifty Co. y lanzado en el año 2013. Este framework se centra en la creación de aplicaciones móviles híbridas, empleando para ello tecnologías web (HTML, CSS y JavaScript).

Una aplicación *Ionic* es, a grandes rasgos, solo una página web que se ejecuta dentro de una shell de aplicación nativa, empleando características de aceleración del hardware nativo en el navegador y optimizando el renderizado gracias a que evade la manipulación del DOM. Como aspecto a destacar, *Ionic* presenta métodos de integración con los principales framewoks de desarrollo, como *Angular*, *React* y *Vue*.

Comparación entre frameworks

Tras realizar un análisis de los mencionados frameworks (Wenhao Wu, 2018) (Michael Gonsalves, 2019), buscando identificar sus aspectos más característicos y aquellas mecánicas que rigen sus respectivas lógicas funcionales; y apoyándonos en los fundamentos de Flutter y el correspondiente desarrollo tecnológico descritos en este documento, es posible establecer una comparación entre las diferentes herramientas en base a 4 criterios diferentes.

El primero de ellos viene determinado por **el lenguaje de programación** subyacente a cada una de las citadas soluciones. Por un lado, *Ionic* se apoya en el abanico completo de tecnologías web para llevar a cabo los desarrollos, haciendo uso del denominado *Cordova Wrapper* para gestionar el acceso a controladores nativos. Bajo esta perspectiva, el carácter multiplataforma se obtiene recurriendo al estándar web para realizar un despliegue uniforme en los diferentes ámbitos, precisando de integraciones con terceros para proporcionar funcionalidades realmente nativas.

Por otro lado, *React Native* centraliza su codificación en el mix entre JavaScript y XML (el mencionado JSX), reduciendo significativamente la frontera entre diseño y funcionalidad, lo que simplifica los procesos de desarrollo. Igualmente, *React Native* también permite la codificación de componentes en Swift, Objective-C o Java en caso en que sea necesario en algún punto del desarrollo.

En este punto, el valor de Dart como lenguaje empleado para el desarrollo multiplataforma es diferencial. En base a la propuesta funcional de *Ionic* y *React Native*, *Flutter* es el único framework que permite unificar, bajo un mismo lenguaje de programación, la gestión simultánea del diseño y la lógica de negocio, proporcionando además un mecanismo de despliegue uniforme para este en cualquier plataforma.

[5] https://ionicframework.com/docs

Este factor nos conduce directamente al siguiente criterio de comparación: **el rendimiento**. Pese a que podríamos afirmar que ninguna de las herramientas descritas presenta un rendimiento pobre, sí es verdad que cuentan con algunas debilidades frente al modelo de *Flutter*.

Tal y como se ha descrito anteriormente, el empleo de tecnologías web por parte de *Ionic* hace que las funcionalidades nativas sean trasladadas a un segundo plano, y que para su acceso se precise de componentes externos. Esto, por ende, condiciona considerablemente el rendimiento general de la aplicación.

Igualmente, el hecho de que *React Native* recurra al *JavaScript Bridge* para efectuar la transición entre el código JSX generado y los correspondientes entornos nativos, supone un deterioro en el rendimiento de la solución. Esta mecánica y sus correspondientes implicaciones quedan expuestas en mayor profundidad en el apartado de descripción teórica de *Flutter*.

A este respecto, las dos mecánicas de compilación presentes en Flutter permiten proporcionar un nivel de rendimiento superior, tanto durante las fases de desarrollo (a través del compilador JIT) como durante las fases de despliegue y ejecución de la herramienta en los diferentes entornos nativos (compilador AOT).

Continuando con este ejercicio de comparación, es igualmente importante destacar el impacto y lógica de funcionamiento de cada una de estas herramientas a la hora de abordar la construcción de **interfaces de usuario**.

En lo referente a este ámbito, *Ionic* presenta un equilibrio interesante. Por un lado, y recordando los aspectos tratados en la presente comparación, el hecho de depender íntegramente del ecosistema tecnológico de la web hace que los aspectos propiamente nativos no cuenten con una calidad excelsa. A cambio, el hecho de emplear un estándar tecnológico trasversal a toda la industria como son las mencionadas tecnologías web hace que todos sus componentes sean totalmente reutilizables en prácticamente cualquier entorno o ámbito de aplicación.

React Native, en cambio, sí que proporciona un *look and feel* bastante similar a soluciones nativas por naturaleza, gracias a la transición realizada desde JSX hacia los componentes de cada ecosistema tecnológico (iOS/Android). Por contraposición, el set de componentes nativos proporcionados por este framework no es demasiado amplio, por lo que se requiere de labores de personalización y adaptación de estos en función de la mecánica a implementar, lo que reduce su capacidad de reutilización.

Flutter, como ya se ha comentado anteriormente, presenta una visión disruptiva a este respecto, pues la naturaleza agnóstica ante cada una de las plataformas nativas obtenida gracias a Dart, otorga a Flutter un control total sobre cada píxel de la interfaz. Esto, además, elimina cualquier limitación intrínseca vinculada a estos entornos y componentes nativos, aumentando exponencialmente las posibilidades de diseño y creación de interfaces. De igual modo, resulta conveniente destacar que existe un alto grado de reutilización de código dentro de la lógica de *Flutter*, gracias a la atomicidad que proporcionan los widgets y a la capacidad de adaptación que se obtiene mediante su composición.

Como último aspecto a destacar, saliendo ligeramente del ámbito puramente técnico, **la masa de usuarios y comunidades** detrás de cada una de las herramientas descritas evidencia ciertas diferencias entre estas.

A este nivel *React Native* ha sido, hasta hace bien poco, el líder indiscutible, debido, por supuesto, a sus características funcionales, pero influido igualmente por la popularidad de JavaScript dentro del marco tecnológico actual. Esto hace que el salto desde el desarrollo web al ámbito del desarrollo de aplicaciones sea prácticamente inapreciable, atrayendo con ello a una gran masa de usuarios.

Ionic, por su parte, no cuenta con la repercusión de *React Native* dentro del sector, pero dada su longevidad dentro del mercado, ha sido capaz de generar progresivamente una comunidad de usuarios no tan numerosa pero igualmente activa.

Finalmente, *Flutter*, pese a ser el framework más novedoso de los 3 citados en esta comparación, ha conseguido hacerse un hueco dentro del mercado gracias a su versatilidad y al innegable impacto que supone contar con el respaldo de un gigante como Google. En cualquier caso, *Flutter* ha experimentado un crecimiento exponencial, llegando a generar una comunidad que, actualmente supera en número a la de *React Native*.

Las estadísticas de GitHub en cuanto a *Stars* (valoración total del repositorio del lenguaje) y *Forks* (copias personales del repositorio original) corroboran las afirmaciones realizadas, lanzando los siguientes datos en el momento de la elaboración de este documento:

- **Flutter**[6]: 134K Gihub Stars / 22,7K GitHub Forks
- **React Native**[7]: 104K GitHub Starts / 22,3K GitHub Forks
- **Ionic**[8]: 47,7K GitHub Stars / 13,7 GitHub Forks

Es igualmente importante destacar que la "juventud" de *Flutter* implica que la cantidad de contenidos generados en torno a esta herramienta y mediante la misma, hoy por hoy, no son muy numerosos. En el escenario opuesto, el bagaje de los años en mercado con los que cuentan *React Native* e *Ionic* ha propiciado la generación de una gran cantidad de contenido sobre estas.

[6] https://github.com/flutter/flutter
[7] https://github.com/facebook/react-native
[8] https://github.com/ionic-team/ionic-framework

CONCLUSIONES DEL PROYECTO

Una vez llegados a este punto, y realizando un ejercicio de análisis y reflexión sobre los resultados de la comparativa expuesta en el punto anterior, podemos afirmar que la llegada de Flutter ha supuesto (y sigue suponiendo) un impacto considerable dentro del ámbito del desarrollo de aplicaciones, e incluso, más allá de este.

Por encima de todo, haber alcanzado la verdadera unificación del comportamiento y el diseño a través de un mismo lenguaje de programación, más concretamente Dart, ya de por sí supone un cambio estructural en el paradigma de desarrollo actual. A través de este recurso, Flutter aglutina lo mejor de los dos mundos, proporcionando la versatilidad de un desarrollo multiplataforma, evitando con ello la necesidad de paralelizar proyectos, equipos de trabajo y perfiles específicos; garantizando a la par el nivel de rendimiento, apariencia y capacidad de respuesta de un entorno nativo.

De igual modo, el consecuente agnosticismo inherente a todo este ecosistema con respecto a cualquiera de los entornos funcionales (ya sea iOS, Andorid o la propia web) elimina las dependencias y limitaciones funcionales vinculadas a cada uno de los espacios nativos. Esto supone un empoderamiento total de la figura del desarrollador, dotando al mismo de un grado de libertad (casi) completo, otorgándole la posibilidad y la responsabilidad de gestionar cada píxel de la interfaz de usuario.

Este control total sobre la interfaz permite la reducción de esta su mínima expresión, dando lugar a una atomicidad que se encarna en los ya citados widgets. Como elementos individuales, su ligereza y capacidad de adaptación permiten dar respuesta a funcionalidades más estandarizadas de una forma orgánica y prácticamente automática. En asociación, las posibilidades de composición de dichos widgets multiplican exponencialmente las opciones funcionales y de diseño, permitiendo construir soluciones personalizadas, pudiendo adaptarse a cualquier contexto y ámbito.

Abstrayéndonos del desarrollo de aplicaciones como tal, estos mismos conceptos, apoyados además en el grado de abstracción funcional que aporta el paradigma, pueden estar sentando las bases de los nuevos modelos de desarrollo emergentes.

En la actualidad, el ecosistema tecnológico en su conjunto ha adquirido un grado de complejidad importante en términos generales. La misma evolución tecnológica que ha puesto sobre la mesa nuevas necesidades y oportunidades de desarrollo, ha traído consigo una sobreexplotación y sobre adaptación de estas (véase el ejemplo de JavaScript), dando lugar a un ecosistema muy poblado y con demasiada redundancia.

El hecho de contar con una infraestructura agnóstica respecto a cualquier contexto funcional proporciona, en este caso a Flutter, una gran flexibilidad y adaptabilidad, potenciando así su

capacidad de integración de forma natural en diversos entornos, problemáticas y proyectos, no entrando en conflicto con herramientas o soluciones preexistentes.

De igual manera, no podemos menospreciar la juventud de Flutter como herramienta de desarrollo de aplicaciones. Pese a su más que justificada irrupción en el mercado, lo que ha conllevado su auge entre los desarrolladores, se trata de un ecosistema en pleno proceso de maduración, durante el cual (aunque cada vez en menor medida) los cambios en el lenguaje y en el framework se han sucedido de forma dinámica. Por consiguiente, el futuro más cercano de Flutter dependerá de este proceso de maduración, poniendo a prueba su capacidad de estabilización llegado el momento.

En cualquier caso, los principios que sustentan su lógica funcional, su posicionamiento y propuesta de valor de cara al mercado y el apoyo trasversal que proporciona Google de cara a su evolución, nos lleva a pensar que Flutter y Dart tienen el potencial para consolidarse como referentes en lo que respecta al desarrollo de aplicaciones nativas multiplataforma, o quizá, algo más.

BIBLIOGRAFÍA

McIlroy, D. (1968). *Mass-Produced Software Components, Software Engineering Concepts and Techniques (1968 NATO Conference on Software Engineering).* Recuperado el 28 de mayo de 2022, de https://www.cs.dartmouth.edu/~doug/components.txt

Blair, I (2022). *Mobile App Download Statistics & Usage Statistics (2022).* Buildfire. Recuperado el 14 de junio de 2022, de https://buildfire.com/app-statistics/#:~:text=21%25%20of%20Millennials%20open%20an,and%2030%20apps%20each%20month

Thrivemyway (2022). *87 Essential Mobile App Stats* [Fotografía]. Recuperado de https://thrivemyway.com/mobile-app-stats/

Strauss, W., & Howe, N. (1991). *Generations: The history of America's future, 1584 to 2069* (Vol. 538). New York: Quill.

El-Kassas, W. S., Abdullah, B. A., Yousef, A. H., & Wahba, A. M. (2017). *Taxonomy of cross-platform mobile applications development approaches*. Ain Shams Engineering Journal, *8*(2), 163-190.

Devsaran (2016). *From History of Web Application Development*. Recuperado el 14 de junio de 2022, de https://www.devsaran.com/blog/history-web-application-development

Nawrocki, P., Wrona, K., Marczak, M., & Sniezynski, B. (2021). *A comparison of native and cross-platform frameworks for mobile applications*. Computer, *54*(3), 18-27.

Huynh, M. Q., Ghimire, P., & Truong, D. (2017). *Hybrid app approach: could it mark the end of native app domination?*. Issues in Informing Science and Information Technology, *14*, 049-065.

Witsdigital. (2020). *What is a Mobile App and why is it important?* [Fotografía]. Recuperado el 17 de junio de 2022, de https://www.witsdigital.com/blog/types-of-mobile-apps-1023

Sharma, S. (2020). *Native vs Hybrid vs Cross-Platform – What to choose?* Medium. Recuperado el 15 de junio de 2022 de https://medium.flutterdevs.com/native-vs-hybrid-vs-cross-platform-what-to-choose-3221130f7cc5

Expertappdevs (2022). *Native vs. Hybrid vs. Cross-Platform: Which Is Best For Mobile App Development*. Medium. Recuperado el 17 de junio de 2022, de https://medium.com/@expertappdevs/native-vs-hybrid-vs-cross-platform-which-is-best-for-mobile-app-development-82ccb07bf905

Kidecha, S. (2020). *Native vs. Hybrid vs. Cross-Platform: How and What to Choose?* Dzone. Recuperado el 20 de junio de 2022, de https://dzone.com/articles/native-vs-hybrid-vs-cross-platform-how-and-what-to

Dagne, L. (2019). *Flutter for cross-platform App and SDK development.*

Mentormate. (2021). *5 Reasons Why Flutter Matters More Than Ever* [Fotografía]. Recuperado el 1 de julio de 2022, de https://mentormate.com/blog/5-reasons-why-flutter-matters-more-than-ever/

CK Hicks. (2019). *Cross-Platform Native App Development — Life Before and After Flutter*. Crema. Recuperado el 3 de julio de 2022, de https://www.crema.us/blog/cross-platform-native-app-development-life-before-and-after-flutter

Css-tricks. (2018). *Flutter: Google's take on cross platform* [Fotografía]. Recuperado el 6 de julio de 2022, de https://css-tricks.com/flutter-googles-take-on-cross-platform/

Deimantas, A. (2020). *P1. Flutter — Making Platform Specific UI Mobile Application (Android with Material, iOS with Cupertino)* [Fotografía]. Recuperado el 6 de julio de 2022, de https://aurimas-deimantas.medium.com/p1-flutter-making-platform-specific-ui-mobile-application-android-with-material-ios-with-b50bc958a31

Copyfuture. (2021). *Dart's JIT and AOT patterns* [Fotografía]. Recuperado el 6 de julio de 2022, de https://copyfuture.com/blogs-details/20210807113106624d

Londhe, O. (2021). *Flutter's 3 tree architecture* [Fotografía]. Recuperado el 8 de julio de 2022, de https://medium.com/@omlondhe/flutters-3-tree-architecture-9263b2bd50d1

ALBERTENGO, G., & WANG, Y. (2022). Review and testing of plugins in Flutter for Android and IOS [Fotografía]. Recuperado el 8 de julio de 2022.

Stevenson. D. (2018). *What is Firebase? The complete story, abridged.*[Fotografía]. Recuperado el 12 de julio de 2022, de [1] https://medium.com/firebase-developers/what-is-firebase-the-complete-story-abridged-bcc730c5f2c0

O'Reilly. (2017). *What's React Native?* O'Reilly. Recuperado el 16 de julio de 2022, de https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html#:~:text=React%20Native%20is%20a%20JavaScript,browser%2C%20it%20targets%20mobile%20platforms.

Gonsalves, M. (2019). *Evaluating the mobile development frameworks Apache Cordova and Flutter and their impact on the development process and application characteristics* (Doctoral dissertation).

Wu, W. (2018). React Native vs Flutter, Cross-platforms mobile application frameworks.

Latture, A. D. (2020). Backdrop: An Exploration of Flutter.

GLOSARIO

A

Acoplamiento. Forma y nivel de interdependencia entre módulos o componentes software.

Aplicación. Tipo de solución software orientada para su despliegue en dispositivos portátiles.

AOT. Siglas de *Ahead Of Time.* En el contexto de Flutter, modo de compilación de Dart orientado a las fases de despliegue y ejecución.

B

Backend. La parte de un sistema informático a la que no accede directamente el usuario, normalmente responsable de almacenar y manipular datos.

C

Composición. Aplicado al Desarrollo de software, combinación de componentes software más sencillos a fin de dar lugar a un elemento y funcionalidad más complejos.

D

Dart. Lenguaje de programación declarativo y orientado a objetos desarrollado por Google.

Devtools. Kit de herramientas de desarrollador proporcionado por Dart para las fases de debug.

Debug. Fase de identificación y eliminación de errores de un dispositivo hardware o software.

Declarativo. paradigma de programación en el que el programador define lo que debe realizar el programa sin definir cómo debe implementarse.

F

Flutter. Framework de interfaz de usuario gratuito y de código abierto de Google para la creación de aplicaciones móviles nativas.

Frontend. La parte de un sistema informático con el que el usuario interactúa directamente.

Firebase. Plataforma de servicios backend desarrollada por Google y accedida por los usuarios en formato Baas (Backend as a Service)

H

HTML. Siglas de Hypertext Markup Language. Sistema estandarizado para el etiquetado de archivos de texto a fin de conseguir efectos de fuente, color o gráficos, entre otros.

I

Ionic. Framework para el desarrollo de aplicaciones multiplataforma desarrollado por Drifty Co.

Imperativo. Paradigma de desarrollo de software en el que las funciones se codifican implícitamente en cada paso necesario para resolver un problema

Interfaz de Usuario. Medios a través de los que el usuario y un sistema informático interactúan.

J

JavaScript. Lenguaje de programación orientado a objetos y comúnmente empleado en entornos web.

JIT. Siglas de *Just In Time.* En el contexto de Flutter, modo de compilación de Dart orientado a optimizar los procesos vinculados a la fase de desarrollo.

M

MVC. Patrón arquitectónico que separa una determinada aplicación en tres componentes lógicos principales: el modelo, la vista y el controlador.

O

Objeto. Paquetes software de datos y sus correspondientes métodos.

R

React Native. Framework de código abierto desarrollado por Facebook y orientado al desarrollo de interfaces de usuario y aplicaciones multiplataforma.

W

Widget. bloques de construcción a través de los cuales se define la interfaz de una aplicación Flutter.

X

XML. Metalenguaje que permite a los usuarios definir sus propios lenguajes de marcado personalizados, especialmente para mostrar documentos en Internet.

Universidad de Alcalá
Escuela Politécnica Superior

ESCUELA POLITECNICA
SUPERIOR

Universidad
de Alcalá