Universitat Oberta de Catalunya

uoc.edu

Scan Food – Análisis de productos alimentarios

Sergio Durban Belmonte
Máster universitario de Desarrollo de aplicaciones para dispositivos móviles

Profesor: Carles Garrigues Olivella
Consultor: Eduard Martin Lineros

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Scan Food – Análisis de productos alimentarios*
Nombre del autor:	*Sergio Durban Belmonte*
Nombre del consultor/a:	*Nombre y dos apellidos*
Nombre del PRA:	*Eduard Martin Lineros*
Fecha de entrega (mm/aaaa):	12/2021
Titulación:	*Máster universitario de Desarrollo de aplicaciones para dispositivos móviles*
Área del Trabajo Final:	*Trabajo final de màster DADM*
Idioma del trabajo:	*Castellano*
Palabras clave	*Flutter, Android, iOS, escáner, multiplataforma*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados i conclusiones del trabajo.*

El trabajo consiste en realizar una aplicación con información sobre los alimentos que podemos encontrar en los supermercados. Esta aplicación ayudara a los consumidores a poder comprar productos con los que se sientan cómodos, a nivel nutricional como a nivel ecológico.

Esta información se extrae desde la plataforma OpenFoodFacts y los usuarios pueden analizar los productos escaneando el código de barras.

Abstract (in English, 250 words or less):

The project involves creating an application with information regarding the food we can find in supermarkets. This application will help consumers to be able to buy products that they feel comfortable with, on a nutritional level as well as on an ecological level.

This information is retrieved from the OpenFoodFacts platform and users can analyze the products via barcode scanning.

Índice

Lista de figuras

1. Introducción

1.1 Contexto y justificación del Trabajo

Cada vez hay más consumidores preocupados por los alimentos que consumen tanto a nivel ecológico como nutricional. Aunque muchos productos ya informan en sus paquetes de toda la información relativa, algunos no muestran de forma clara al consumidor si estos son alimentos procesados, medioambientalmente responsables.

Para saber todo esto hay varias calificaciones como los *grupos nova*[1], la calificación Nutri-Score[2] y el Eco-score[3], la ley actualmente fuerza a los fabricantes a publicar los valores Nutri-score en el envase del producto. Aun así, los fabricantes no están obligados a publicar la información de EcoScore o si su alimento es ultra procesado.

En las principales tiendas de aplicaciones ya encontramos algunos aplicativos que dan acceso a esta información, como puede ser *Yuka* (https://yuka.io/es/), *MyRealFood* (https://myrealfood.app/) y *El CoCo* (https://elcoco.es/). Todas estas aplicaciones tienen opciones de suscripción complicando el acceso a los datos.

Si nos revisamos las condiciones legales de las aplicaciones que existen en el mercado podemos ver que los datos salen de un proyecto que recopila la información (con licencia *OBbL* que permite usar, modificar y compartir de forma libre) Open Food Facts (https://es.openfoodfacts.org/).

El objetivo final de este trabajo es dar acceso a los usuarios a la información que se encuentra en esa base de datos de forma sencilla desde su teléfono móvil y sin restricciones de pago, de manera que los consumidores puedan ser informados y elegir su alimentación en función a sus preferencias.

Los usuarios podrán escanear mediante la cámara de su dispositivo móvil los códigos de barras de los productos y obtener los indicadores anteriormente mencionado y también los valores nutricionales de los productos para que el cómo consumidor pueda elegir los productos que le convencen y ejercer sus derechos como consumidor.

---

[1] Clasificación de alimentos basada en 4 grupos: No procesados (o mínimamente), Culinarios procesados, procesados, ultra procesados.
[2] Clasificación de productos con valores entre A y E, se valoran puntos negativos como el azúcar, grasas saturadas y puntos positivos como la proporción de verduras, nueces, vegetales…
[3] Clasificación de productos por su calificación ambiental (valores entre A y E), se evalúa el modo de producción, el origen de ingredientes, el empaquetado y el impacto a la biodiversidad.

1.2 Objetivos del Trabajo

El objetivo principal del trabajo es la publicación de una aplicación multiplataforma de una aplicación hibrida que muestre información nutricional de los productos escaneados mediante el código EAN.

Esto se realizará con el *framework Flutter* y desarrollando las partes nativas en *Kotlin* y *Swift* que sean necesarias para lograr el objetivo final.

1.2.1 Requisitos funcionales

- Escaneo de códigos *EAN*: Escaneo mediante la cámara del dispositivo móvil.
- Búsqueda de productos mediante *API*: Búsqueda de los productos mediante códigos *EAN* y títulos.
- Registro de productos en histórico: Para ser consultado con posterioridad en la pestaña de histórico.
- Filtrado por alergenos: Filtrado de productos por alergenos comprendidos en el producto con el fin de encontrar alternativas viables para el usuario.

1.2.2 Requisitos no funcionales

- La aplicación debe dar una experiencia multiplataforma uniforme entre las plataformas *Android* y *iOS*.
- Creación de pruebas automatizadas para asegurar el funcionamiento de la aplicación.
- Soporte de múltiples idiomas.
- Publicación en *Apple Store* y *Google Play*.

1.3 Enfoque y método seguido

Para tal de llevar a cabo el objetivo del trabajo se realizará un desarrollo de la aplicación con *Flutter* (permitiendo ejecutarse en *Android* y *iOS*). Para algunas acciones como podría ser escanear un documento se realizará un desarrollo nativo con un conector (*Platform Channels*[4]) para *Flutter*.

El hecho de elegir *Flutter* y no el desarrollo nativo está motivado por conseguir el aplicativo en los dos principales ecosistemas móviles con un coste menor. Gracias al uso de los *Platform Channels* podremos conectar de forma sencilla los lectores de códigos de barras nativos con nuestro código *Flutter*, haciendo que el único código que tengamos que desarrollar en nativo sea ese y reaprovechar el resto de código.

Este código nativo será escrito en *Kotlin* y *Swift* (para *Android* i *iOS*), en *Android* se utilizará la librería *KBarcode*[5], una librería moderna de escaneo de códigos a la que tendremos que realizar el trabajo de crearle los *Platform Channels* correspondientes. Por el lado de iOS trabajaremos con el SDK de *Vision*[6]que permite escanear códigos, a esto también tendremos que conectarlo a una interfaz común de *Platform Channels* para reutilizar este código entre las plataformas.

El histórico del usuario lo almacenaremos mediante el uso de *Hive*, una base de datos *NoSQL* para *Dart* que tiene su propio sistema de almacenamiento en fichero.

La base de datos de *Open Food Facts* es publica y se puede tanto copiar como utilizar la API publica que exponen, así que los datos necesarios para el aplicativo se pueden extraer de su API, incluso tienen una librería disponible para Dart (https://pub.dev/packages/openfoodfacts).

En cuanto a la *landing page*, será una página sencilla a partir de una plantilla realizada con *Gatsby* (https://www.gatsbyjs.com/), este *framework* de *Javascript* nos permite programar de forma rápida una página web estática. Esta será necesaria para poder realizar el despliegue de la aplicación en las correspondientes tiendas.

[4] https://flutter.dev/docs/development/platform-integration/platform-channels
[5] https://github.com/brightec/KBarcode
[6] https://developer.apple.com/documentation/vision/vndetectbarcodesrequest

1.4 Breve sumario de productos obtenidos

Al finalizar el proyecto se obtendrán los siguientes productos:

- Memoria del proyecto.
- Código fuente de la aplicación.
- Código fuente de la web.
- Presentación del proyecto.
- Website publicada con Landing y T&C.
- Aplicativo disponible en Play Store.
- Aplicativo disponible en Apple Store.

1.5 Planificación del Trabajo

Los recursos utilizados para el desarrollo de este trabajo son:

- **Portátil Gigabyte Aorus 15P XD** con procesador Intel i7-11800H (2,3 GHz), 16GB de memoria RAM y gráficos NVIDIA RTX 3070. El sistema operativo utilizado es Windows 11.
- **MacBook Pro 16'' (2020)** con procesador Intel i9 (2,3 GHz), 16GB de memoria RAM y gráficos AMD Radeon Pro 5500M. El sistema operativo utilizado es Mac OS X Big Sur. Este portátil solo se utilizará para compilar la aplicación en iOS.
- **iPhone 12** con la última versión de iOS disponible en cada momento, actualmente iOS 15.
- **Xiaomi MI 9T PRO** con la última versión de Android disponible para este dispositivo, actualmente Android 10.
- **Samsung Galaxy A20** con la última versión de Android disponible para este dispositivo, actualmente Android 11.
- **IntelliJ IDEA**: IDE de desarrollo para Flutter y Android.
- **XCode**: IDE de desarrollo para iOS.
- **Git**: para el control de versiones.
- **Draw.io**: Para la creación de diagramas.
- **Adobe XD**: Para la realización del prototipo final.
- **Microsoft Office**: Para la creación de documentación.

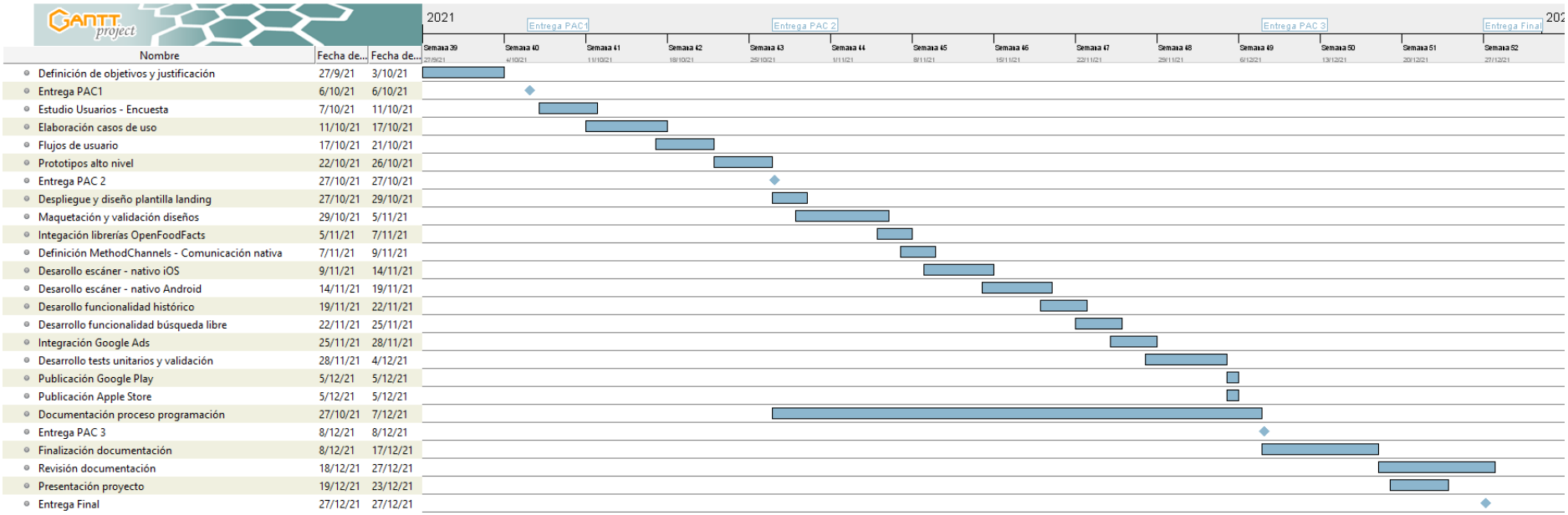
He decidido distribuir mis horas de la siguiente manera:

- De lunes a viernes: 2 horas diarias.
- Sábados, domingos: 5 horas diarias.

Esto suma en total 282 horas en total para todo el proyecto.

La división del trabajo se realizará en cuatro entregas parciales, y al ser desarrollado solo por mí, hará que el trabajo se realice de forma secuencial. Se divide el trabajo en 4 hitos con las siguientes fechas:

- PAC 1: del 27/09/2021 al 06/10/2021.
- PAC 2: del 07/10/2021 al 27/10/2021.
- PAC 3: del 27/10/2021 al 08/12/2021.
- Entrega final: del 08/12/2021 al 27/12/2021.



Nombre	Fecha de...	Fecha de...
Definición de objetivos y justificación	27/9/21	3/10/21
Entrega PAC1	6/10/21	6/10/21
Estudio Usuarios - Encuesta	7/10/21	11/10/21
Elaboración casos de uso	11/10/21	17/10/21
Flujos de usuario	17/10/21	21/10/21
Prototipos alto nivel	22/10/21	26/10/21
Entrega PAC 2	27/10/21	27/10/21
Despliegue y diseño plantilla landing	27/10/21	29/10/21
Maquetación y validación diseños	29/10/21	5/11/21
Integación librerías OpenFoodFacts	5/11/21	7/11/21
Definición MethodChannels - Comunicación nativa	7/11/21	9/11/21
Desarollo escáner - nativo iOS	9/11/21	14/11/21
Desarollo escáner - nativo Android	14/11/21	19/11/21
Desarollo funcionalidad histórico	19/11/21	22/11/21
Desarrollo funcionalidad búsqueda libre	22/11/21	25/11/21
Integración Google Ads	25/11/21	28/11/21
Desarrollo tests unitarios y validación	28/11/21	4/12/21
Publicación Google Play	5/12/21	5/12/21
Publicación Apple Store	5/12/21	5/12/21
Documentación proceso programación	27/10/21	7/12/21
Entrega PAC 3	8/12/21	8/12/21
Finalización documentación	8/12/21	17/12/21
Revisión documentación	18/12/21	27/12/21
Presentación proyecto	19/12/21	23/12/21
Entrega Final	27/12/21	27/12/21

Ilustración 1. Diagrama Gantt Planificación

2. Diseño

2.1 Diseño centrado en el usuario (DCU)

En esta primera fase del proyecto el objetivo es conocer las necesidades de nuestros usuarios, esto lo realizaremos mediante una **encuesta**, esta nos ayudará a poder satisfacer sus necesidades.

2.1.1 Encuesta

La encuesta ha sido distribuida entre compañeros de trabajo y familiares. Entre los 74 encuestados encontramos gente con diferentes alergias alimentarias e intolerancias, como podría ser la intolerancia al gluten, alergia a los frutos secos o intolerancia de la lactosa.

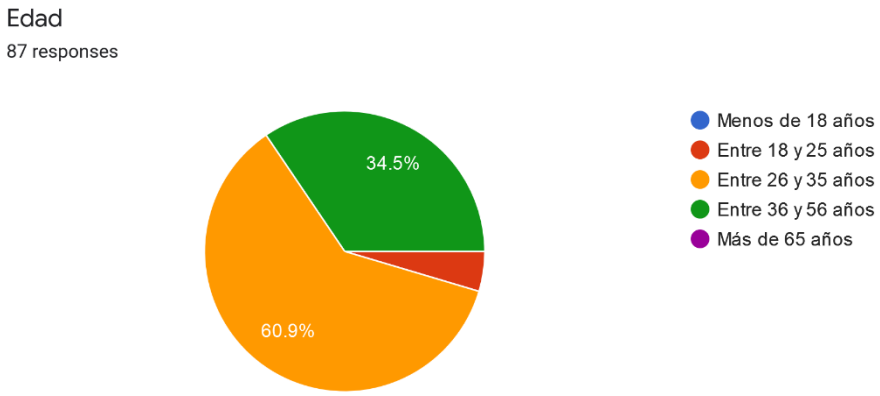


Ilustración 2. Encuesta: Distribución edad encuestados

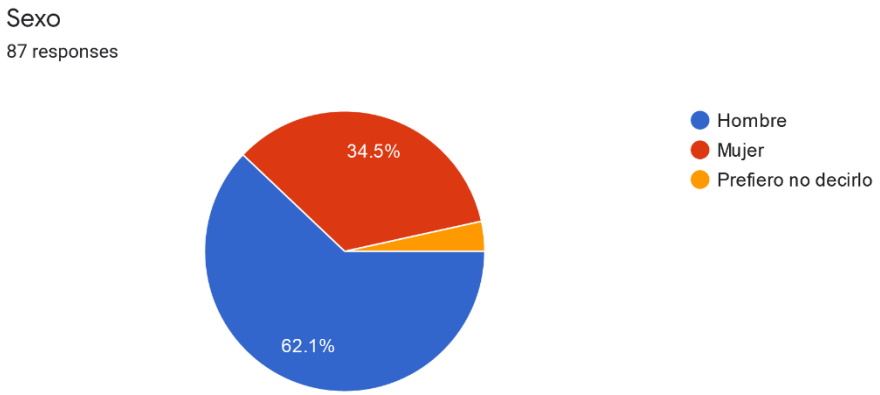


Ilustración 3. Encuesta: Distribución sexos encuestados

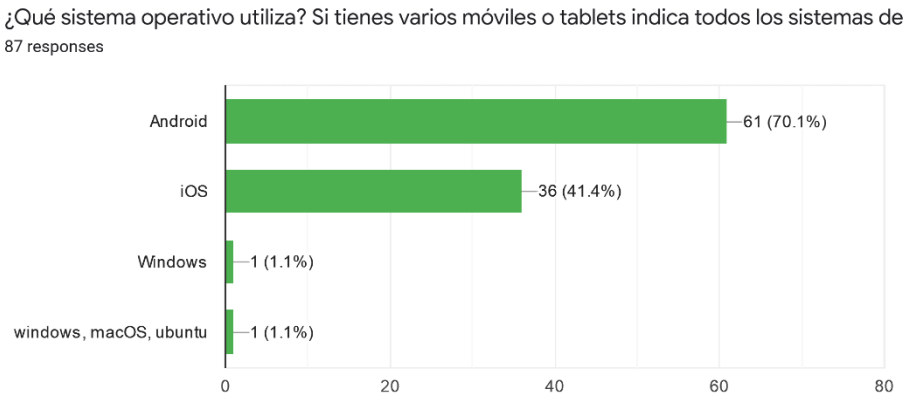


Ilustración 4. Encuesta: Distribución sistema operativo

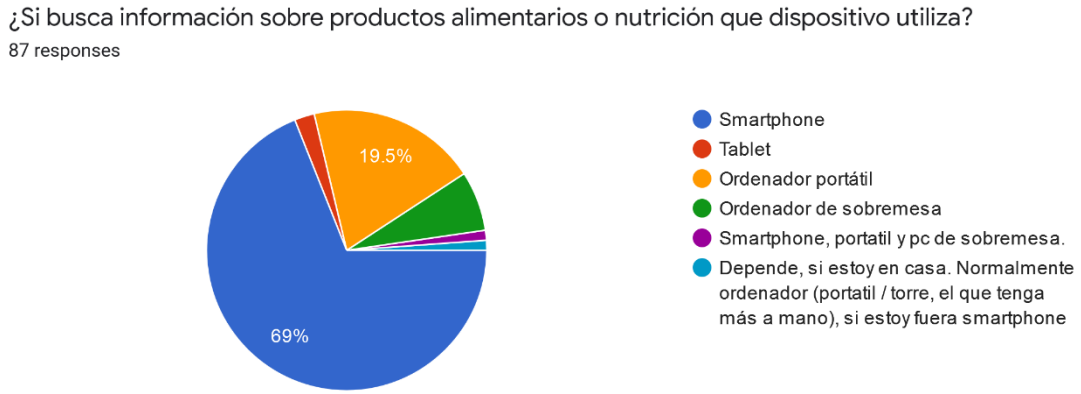


Ilustración 5. Encuesta: Búsqueda productos

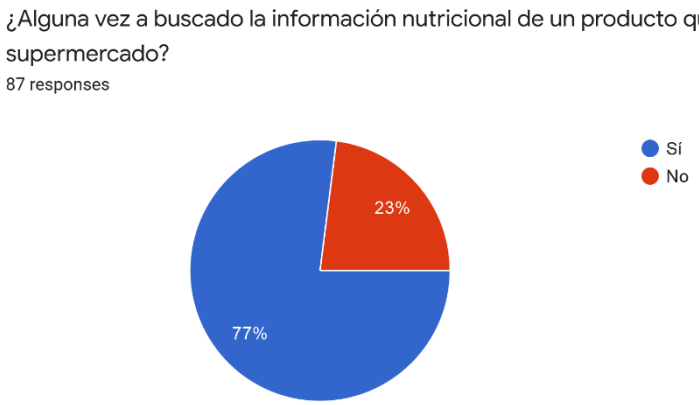


Ilustración 6. Encuesta: Búsqueda información nutricional

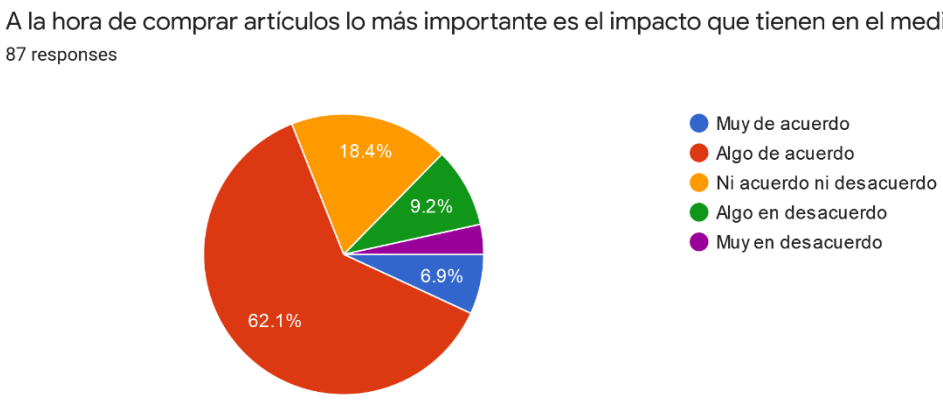


Ilustración 7. Encuesta: Importancia impacto medio ambiente.

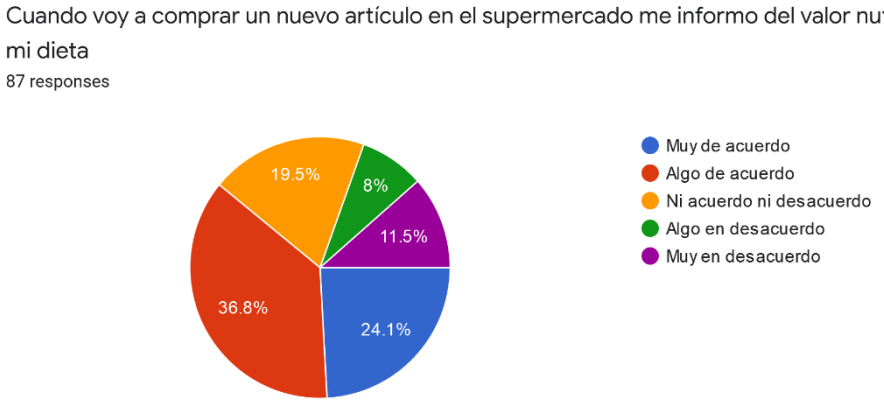


Ilustración 8. Encuesta: Importancia valor nutricional.

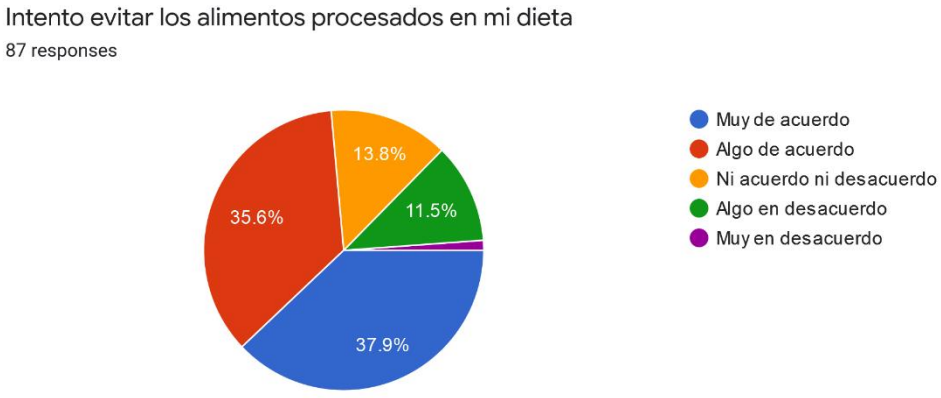


Ilustración 9. Encuesta: Evitar alimentos procesados en dieta.

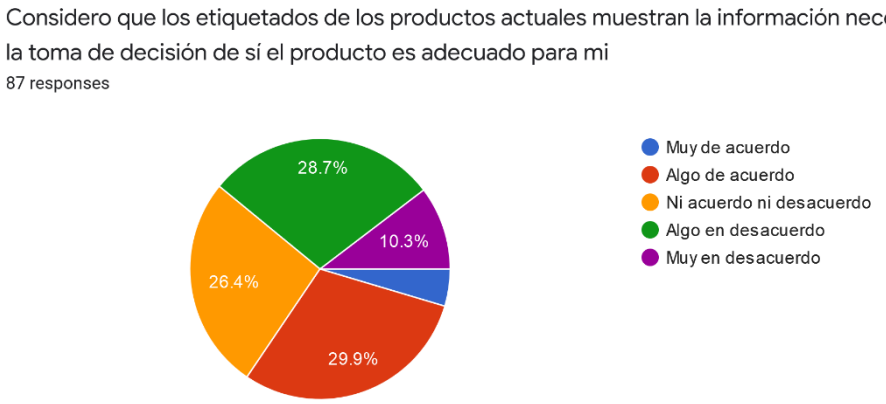


Ilustración 10. Encuesta: Buen etiquetado producto.

Las preguntas estaban orientadas a dar información relevante sobre qué información quieren los usuarios ver en la aplicación:

- Podemos ver la mayoría de los usuarios utiliza dispositivos móviles para informarse sobre productos alimentarios.
- La mayoría de los usuarios se han informado de los valores nutricionales de los productos que compran en los supermercados.
- Los usuarios ven relevante tener la información del impacto medioambiental de los productos que consumen.
- La mitad de los usuarios encuestados comprueban el valor nutricional de los productos que compran en el supermercado.
- La mayoría de los usuarios intentan evitar los alimentos procesados en su dieta.
- Al menos la mitad de los usuarios consideran que el etiquetado de los productos no les da suficiente información sobre el producto que van a comprar.

Con esta información podemos determinar que es importante enseñar a los usuarios los valores de calificación Nutri-Score, Eco-Score y grupos NEBA.

2.1.2 Entrevista

Durante el proceso de encuesta se encontró un usuario muy interesado en la App ya que tiene muchos problemas para encontrar alimentos nuevos ya que no puede consumir gluten.

P. Te cuesta mucho localizar los productos que puedes consumir en el supermercado?

R. Resulta más sencillo que antes, pero hay mucho que mejorar.

Hasta hace unos 20 años no se etiquetaba en absoluto si el producto contenía alguna traza, ya sea de gluten u otro tipo de alergeno. Mercadona a través de su etiquetado ayudó muchísimo a la comunidad y sirvió de ejemplo para muchas otras cadenas de supermercados, sin embargo, hoy en día muchas de las importantes cadenas se resisten a declarar los alérgenos en los productos de su propia marca. También hay que mencionar que Mercadona usa su propio etiquetado y protocolo cuando deberían regirse por estándares europeos y nacionales, lo cual tampoco lo hace 100% fiable.

P. Crees que el etiquetado de los productos te ayuda a localizar cuales tienen gluten o no?

R. La mayoría de los productos sigue su propio diseño, en muchos casos siendo bastante difícil encontrar la denominación. E incluso cuando un producto informa en la etiqueta que no lleva gluten, puede no ser completamente cierto, a no ser que use el estándar europeo o español.

El estándar para declarar un producto libre de gluten se establece por cada país de forma independiente, por lo que, en muchos casos, personas muy sensibles pueden comer productos declarados sin gluten en algunos países (como en Reino Unido) y caer enfermos.

Por otra parte, en España se encuentra la FACE (Federación de Asociaciones de Celíacos de España). Una organización que audita en las fábricas y restaurantes a través de un protocolo, para verificar que el producto se encuentre libre de gluten. Al hacerte miembro de la organización (tienes que pagar una subscripción) te mandan un listado con todos los productos por marca que disponen de la aprobación. Disponen también de un sistema de etiquetado que muy pocas marcas y restaurantes utilizan, ya que la auditoría es compleja o simplemente ignoran esta certificación.

P. A la hora de comprar nuevos productos que no suelen estar en tu dieta, como lo haces?

R. Me aseguro de que la marca dispone de etiquetado garantizando que no lleva gluten y me informo a través de las redes para una verificación.

Para evitar problemas intento llevar una dieta con productos naturales.

P. Has utilizado alguna vez alguna App para ayudarte con tu alergia alimentaria?

P. En alguna ocasión has utilizado alguna aplicación móvil para intentar localizar productos sin gluten? En caso afirmativo como fue la aplicación que utilizaste y como fue la experiencia.

R. No. Personalmente he tenido mala experiencia en España con algunas aplicaciones, que en algunos países como en Alemania funcionan mejor. Principalmente por disponer de una base de datos muy limitada. Aunque sí es cierto que llevo muchos años sin utilizar estas aplicaciones por desconfianza.

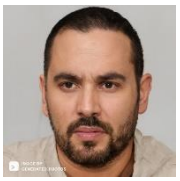
De la entrevista podemos extraer de que existe una problemática real en el etiquetado de productos, y sería bueno indicar las posibles alergias destacadas en el diseño final. La esto verifica la idea inicial de trabajar con *OpenFoodFacts***,** este proyecto comunitario tiene bastante recorrido y datos actualizados por la comunidad. También sería ideal en una futurible versión de la aplicación programar la subida de datos a este proyecto para seguir colaborando en la base de datos abierta.

2.1.3 Personas

De la información extraída podemos determinar que los usuarios que utilizaran nuestra aplicación serán adultos, mayores de edad, emancipados, que viven

Basándonos en los resultados y conclusiones de la encuesta y entrevista, elaboramos los perfiles de personas que podrían utilizar nuestra aplicación.

Descripción de Persona	
 **Ilustración 11. Persona 1** **Nombre: Víctor** **Edad: 36** **Estudios: Estudios obligatorios** **Profesión: Recepcionista en un hotel.** **Estado civil: Casado** **Residencia: Zaragoza**	Víctor vive con su pareja en Zaragoza y trabaja de recepcionista en un hotel de una gran cadena. Él no tiene mucho tiempo para ir a comprar o prepararse su comida así que consume muchos productos que le proporciona el hotel. Al comer mucho a deshoras por su trabajo se preocupa mucho de su nutrición ya que últimamente ha aumentado mucho de peso y quiere empezar a controlar que productos son los que consume, así que ha decidido evitar los productos con un alto contenido calórico.
Escenario	
Hoy Víctor vuelve a comer en la oficina, la empresa le proporciona productos de comida preparada típicos que podemos encontrar en el supermercado, del estilo ensaladas marca Florette o canelones marca Eroski. **Antes de elegir que comerá hoy le gustaría saber cuál de todos estos productos para ver cuál es su contenido calórico, azucares, grasas y fibras. Así que abre la aplicación para escanear con la cámara el código de barras ean de cada uno de los productos y proceder a comparar que producto le conviene más para su dieta.** **Necesidades cubiertas:** - **Proporciona información nutricional sobre varios productos de cara a compararlos y elegir cual le beneficiara más para su dieta**	

Descripción de Persona	
 **Ilustración 12. Persona 2** **Nombre: Alejandro** **Edad: 43** **Estudios: Ing. informática** **Profesión: Programador** **Estado civil: Soltero** **Residencia: Barcelona**	Alejandro vive en un pueblo cercano de Barcelona y trabaja de programador en una empresa de la capital. Él come todos los días en la oficina y se prepara la comida en casa ya que tiene varias intolerancias alimentarias (gluten, lactosa...). Debe tener mucho cuidado con los productos que consume, sobre todo con los procesados porque cualquier error al seleccionar un nuevo producto en el supermercado le puede hacer pasar un mal rato con dolores de estómago e incluso provocarle una visita al hospital.

Escenario

Alejandro ha salido a comprar al super de la esquina, una vez dentro del super ve que han retirado las galletas sin gluten que el consumía.

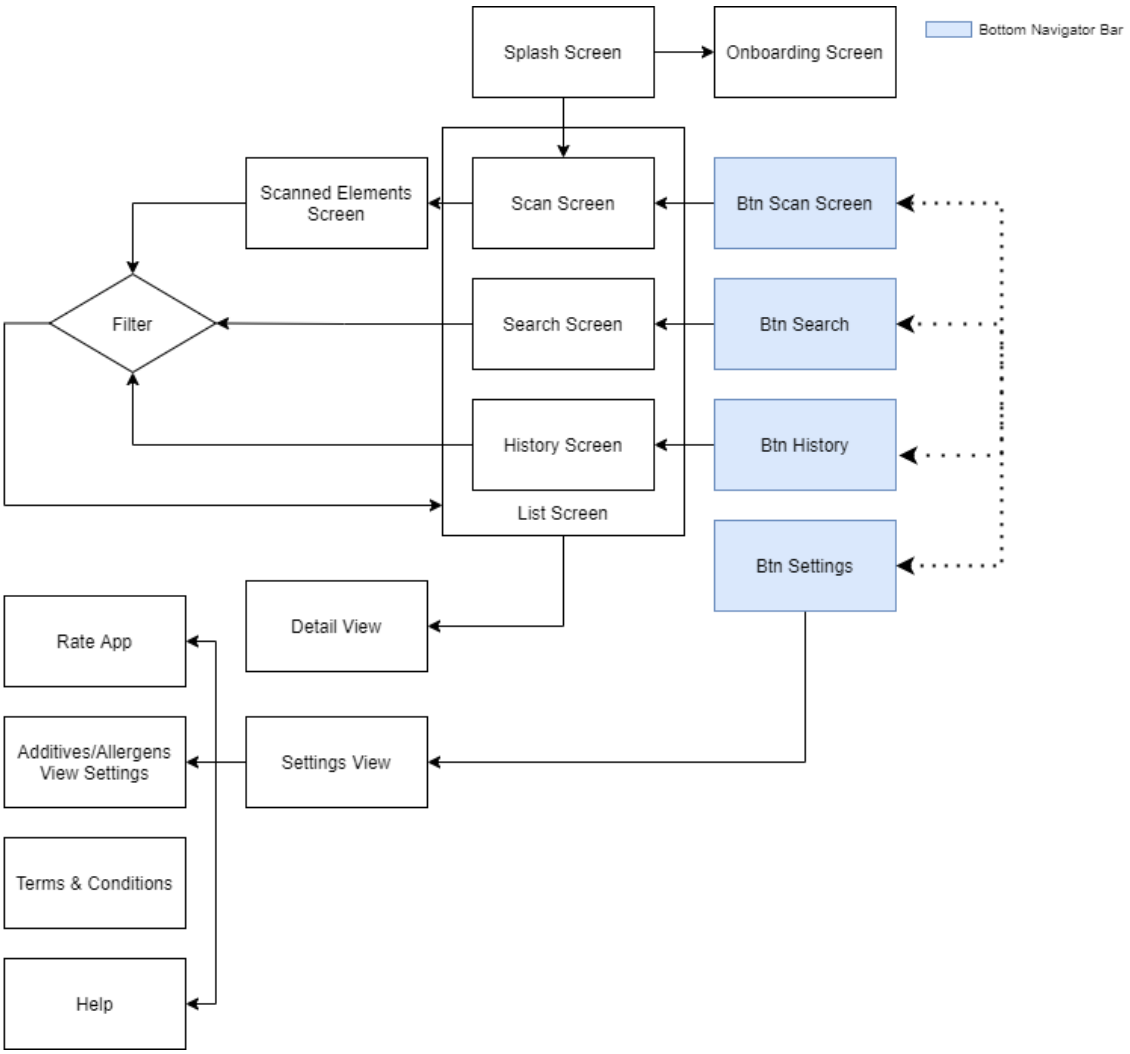
Así que ahora comienza la búsqueda titánica de buscar una nueva marca y producto que realice galletas sin gluten y sin lactosa. Así que enciende la aplicación y comienza a escanear los códigos de barras de los paquetes de galletas que encuentra en las estanterías, después con un filtro retira los que no puede consumir y selecciona una nueva marca.

Necesidades cubiertas:

- **Comprobar múltiples productos de forma rápida en búsqueda de alergenos para poder seleccionar entre los que puede consumir realmente.**

2.1.4 Árbol de navegación

En la siguiente figura se muestra el árbol de navegación de la aplicación. En este podemos ver los diferentes caminos que el usuario puede utilizar dentro de la aplicación.



Al entrar a la aplicación el usuario verá la *splashscreen*, aquí el usuario podrá ver el logo y el color primario de la aplicación al fondo. Si el usuario es nuevo de aquí navegara al *onboarding* donde se le realizara una breve introducción a la aplicación. Y de ahí saltará a la página principal en concreto a la *scanscreen* de ahí al usuario podrá saltar a otras pantallas como la *searchscreen*, *historyscreen*, o bien escanear unos códigos de barras (activando un botón con un numero de los códigos de barras detectados). Todas estas vistas tienen en común ser un listado de productos, este listado puede ser filtrado por alergenos, marcando en la lista los artículos que el usuario no puede consumir.

Desde esa misma pantalla el usuario puede ir a la vista de detalle, donde podrá ver la información del producto en detalle, remarcando la información nutricional, Nutri-Score, Nova y la puntuación Eco-Score.

Aparte de todo esto el usuario también puede ir a la pantalla de Configuración, allí puede encontrar un botón para calificar la aplicación, otro para ir a la página web y ver los términos y condiciones y un botón de ayuda que abrirá la aplicación de correo por defecto. Aparte de eso también hay una vista de *Aditives/Allergens* donde tendrá una lista completa de todos los aditivos y alergenos donde podrá seleccionar cuales quiere ver destacados en la vista de lista y vista individual.

2.2 Prototipado

2.1.5 Funcionalidades principales

- La aplicación debe permitir escanear códigos de barras para poder realizar búsquedas de los productos.

- La aplicación debe permitir escanear múltiples productos para poder compararlos en un flujo sencillo para el usuario.

- La aplicación debe permitir filtrar por alergenos para facilitar la selección de productos a los usuarios con alergias alimentarias.

- La aplicación debe informar de los valores nutricionales del producto de una forma sencilla a los usuarios (con uso de semáforos).

- La aplicación debe poder usarse sin registro ya que no necesitamos información de los usuarios ni sincronizar sus perfiles entre múltiples aplicaciones.

- La aplicación debe permitir al usuario personalizar que propiedades de los productos destacar en los listados (por defecto se destacaran EcoScore, NutriScore y EVA), permitiendo a los usuarios saber a simple vista si un producto contiene alergenos.

- La aplicación debe almacenar el contenido del usuario localmente, de esta manera protegeremos su privacidad.

2.2.1 Prototipo LoFi

Teniendo en cuenta la investigación realizada en el apartado anterior, se realiza la primera fase de diseño en papel.

Al arrancar la aplicación el usuario ve la *SplashScreen*, esta página sirve para que la aplicación prepare los accesos a la base de datos y otras operaciones que necesite para funcionar la aplicación con normalidad.

Ilustración 13. Diseño LoFi – SplashScreen

De esta página el usuario (si es un nuevo usuario) viajará a la página de onboarding, una página con 3 diapositivas explicando al nuevo usuario que se le ofrecerá en la aplicación.

Ilustración 14. Diseño LoFi – OnboardingScreen

Después de este visual el usuario ya se encontrará con la pantalla de la cámara (es decir, ya podrá comenzar a escanear) y navegar por el resto de la aplicación mediante la barra de navegación inferior.

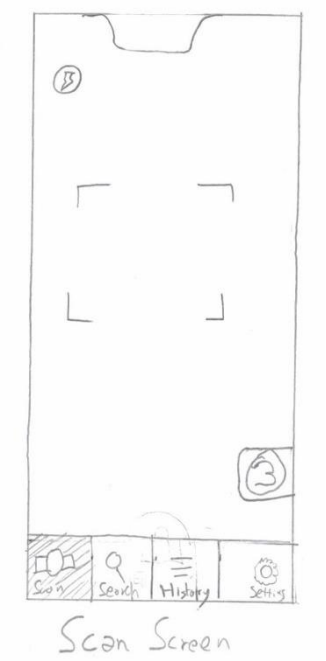


Ilustración 15. Diseño LoFi – ScanScreen

Cuando un usuario apunta a un código de barras aparece la barra lateral donde le indica la cantidad de códigos de barras que a escaneado y haciendo click allí puede ir a la vista de lista donde podrá ver todo lo que lleva escaneado hasta ese momento. Eso le permite al usuario escanear múltiples productos para luego hacer una elección entre varios en la pantalla de selección.

Los usuarios también pueden buscar por nombre o por el número del código de barras productos.

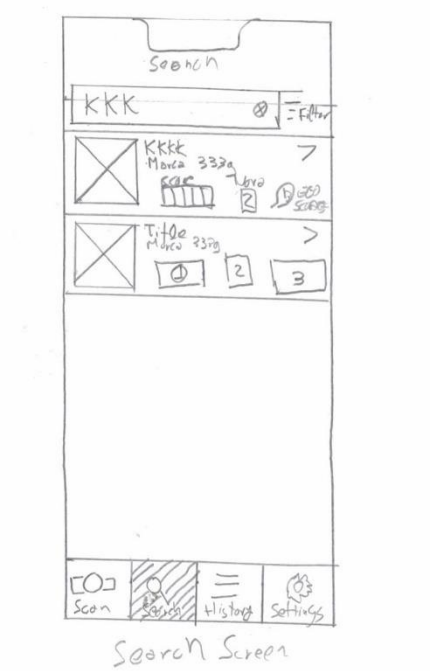


Ilustración 16. Diseño LoFi – SearchScreen

Esta misma pantalla se reutiliza sin el cuadrado de búsqueda tanto en la historia como en la pantalla de selección al escanear.

Ilustración 17. Diseño LoFi – HistoryScreen

Todas estas pantallas tienen en común que el usuario puede realizar un filtro a posteriori para hacer desaparecer de la pantalla los elementos que contienen ciertos alergenos.

Ilustración 18. Diseño LoFi – Modal Inferior – Filtro

Aparte las pantallas antes mencionadas también encontramos la pantalla de configuración, desde donde podremos Calificar la aplicación, ver los términos y condiciones, contactar al soporte y configurar los alergenos que queremos que salgan destacados en los listados (o bien que sigan saliendo el NutriScore, Nova y EcoScore).

Ilustración 19. Diseño LoFi – SettingsScreen

La pantalla de selección de los elementos a destacar es una pantalla con la posibilidad de seleccionar tres elementos del listado de entre todos los alergenos disponibles.

Ilustración 20. Diseño LoFi – Allergens/ScoreScreen

Y por último la pantalla de detalle de producto a la que podremos acceder desde cualquiera de las secciones de listado de la aplicación. En este encontraremos las puntuaciones (NutriScore, Nova, EcoScore), un resumen de los niveles nutricionales con semáforos (Rojo, Amarillo, Verde) para indicar si son adecuados y un listado de alergenos.

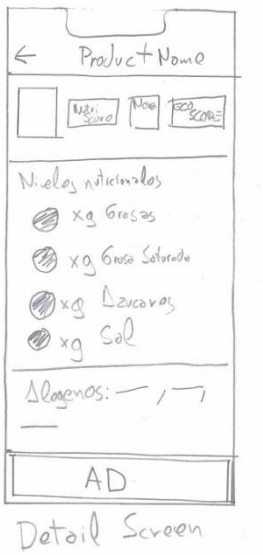


Ilustración 21. Diseño LoFi - DetailScreen

2.2.2 Prototipo HiFi

Primero de todo definiremos la fuente a utilizar y la composición de colores de la aplicación.

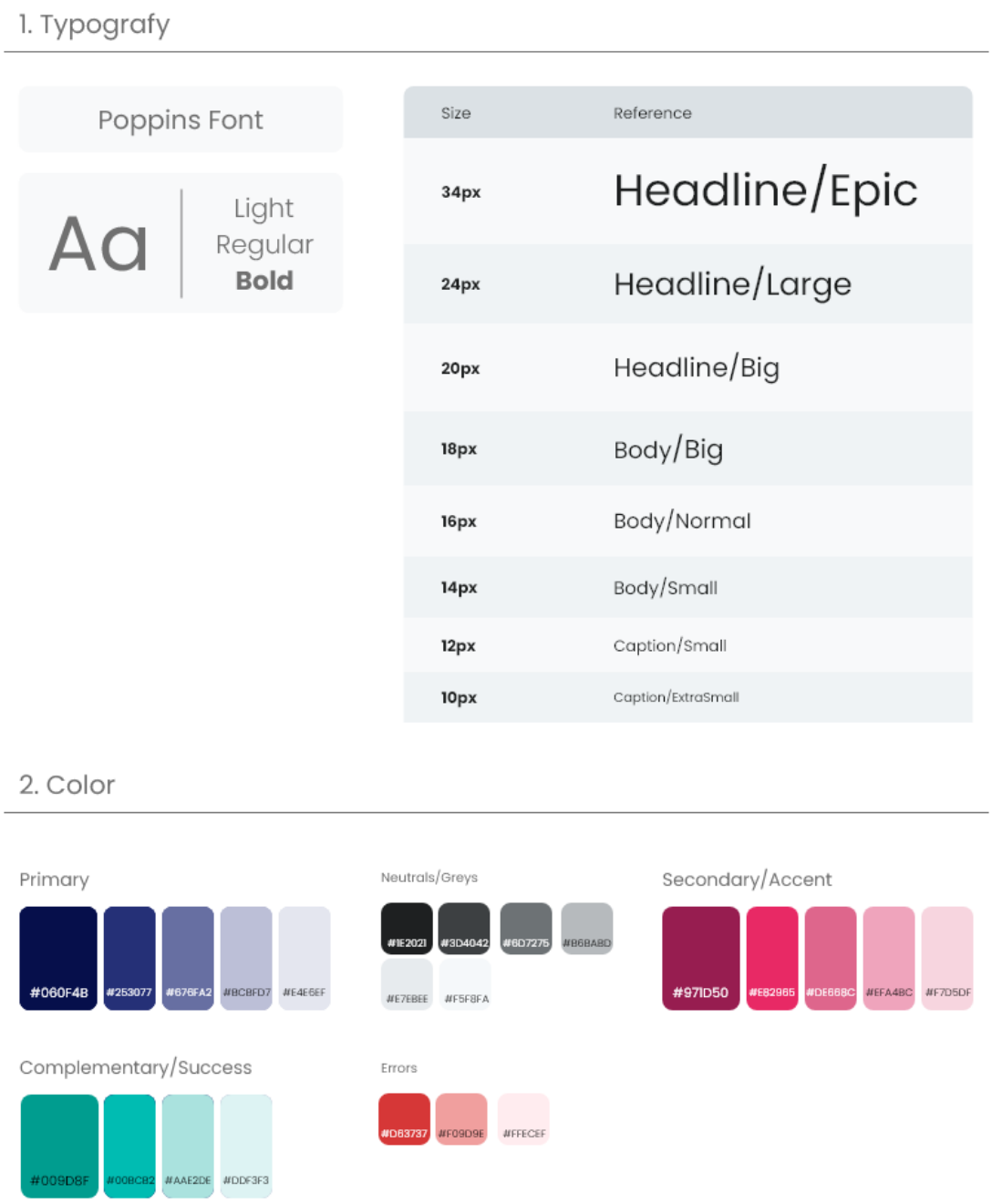


Ilustración 22. Guía de estilo.

Lo primero antes de realizar el diseño HiFi es la elección de la fuente, para esta aplicación se elige Poppins (desde *Google Fonts*[7]), es una fuente sin serifa que facilitara la lectura en la aplicación. Esta fuente tiene licencia *Open Font License*[8], que nos permitirá utilizar sin problemas la fuente en la aplicación.

[7] https://fonts.google.com/specimen/Poppins
[8] https://scripts.sil.org/cms/scripts/page.php?site_id=nrsi&id=OFL

Sobre la paleta de colores, tenemos por un lado el color principal con sus temperaturas, un secundario y un complementario (aplicando el sistema de color de *Material Design*[9]). Aparte de esos colores también se ha elegido el color de error, teniendo 3 tonalidades para así mostrar al usuario diferentes tipos de errores en función de la magnitud.

Lo siguiente que se diseña es el *Logo* de la aplicación que la acompañara en la SplashScreen y en el icono del drawer del teléfono.

Ilustración 23. Logo aplicación

El logo de la aplicación se ha realizado en InkScape en vectorial, este logo este realizado con partes de otras imágenes vectoriales con licencias abiertas que nos permite la redistribución y modificado de estas. Todas las imágenes proceden de la Wikipedia.

Imagen	Licencia
Food_Bank_icon.svg	CC0
Magnifying_glass_icon.svg	Dominio publico
Wikipedia_barcode_128.svg	Dominio publico

Una vez tenemos listo el logo de la aplicación ya podemos empezar con las pantallas de la aplicación. Estas pantallas se realizan con Adobe XD y se realiza una presentación interactiva para validar el diseño, esta es visible en la siguiente url:

https://xd.adobe.com/view/d1ea9b28-991c-4e97-85c2-d1e29bf263dc-84b9

En la URL se puede ver de forma interactiva el diseño HiFi que listo a continuación.

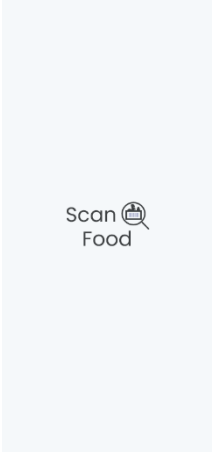


Ilustración 24. Diseño HiFi – SplashScreen

[9] https://material.io/design/color/the-color-system.html#color-usage-and-palettes

De la página SplashScreen pasamos a la página de OnboardingScreen.

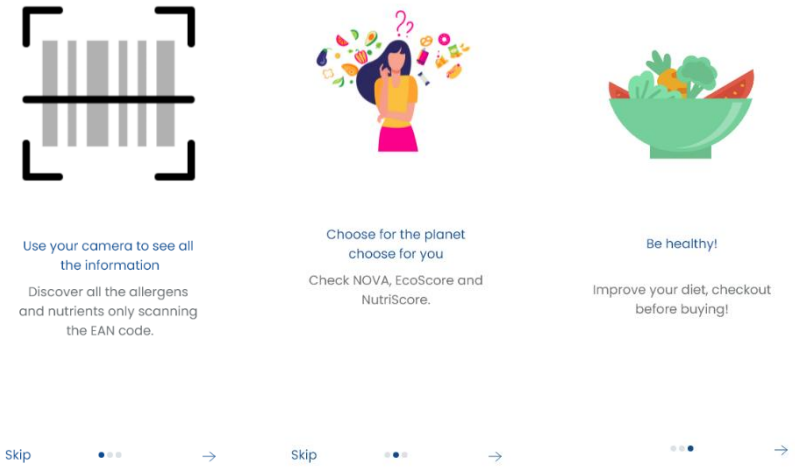


Ilustración 25. Diseño HiFi – Onboarding

En estas páginas el usuario recibe información sobre que podrá realizar en la aplicación y directamente salta a la ScanScreen.

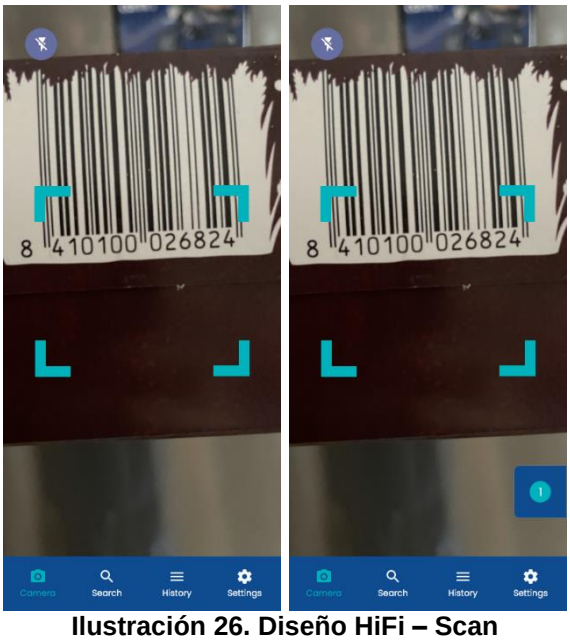


Ilustración 26. Diseño HiFi – Scan

De aquí los usuarios pueden saltar a la vista de detalle de los escaneos donde ven el listado de ítems escaneados, y pueden filtrar el resultado para quitar los elementos con alergenos.

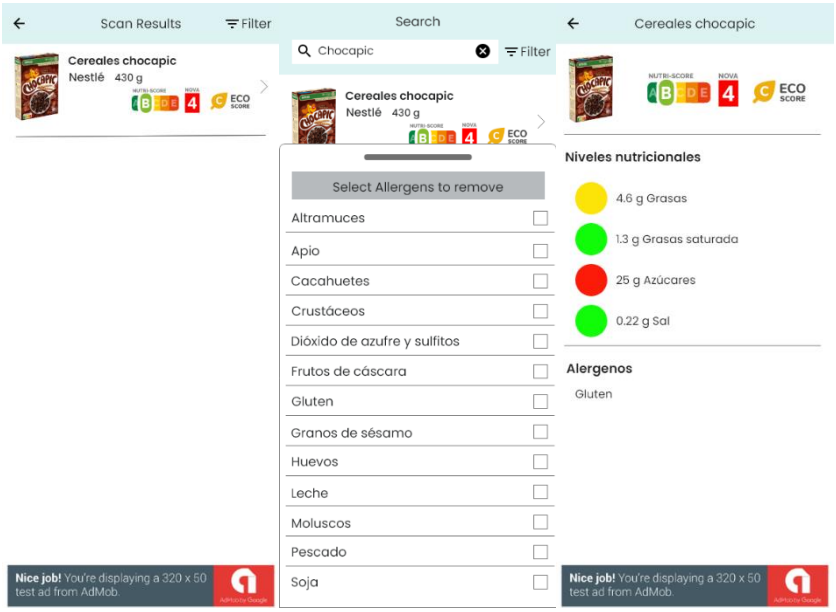


Ilustración 27. Diseño HiFi – ScanResults

La pantalla de History y la pantalla de Search funciona de forma muy similar con un listado.

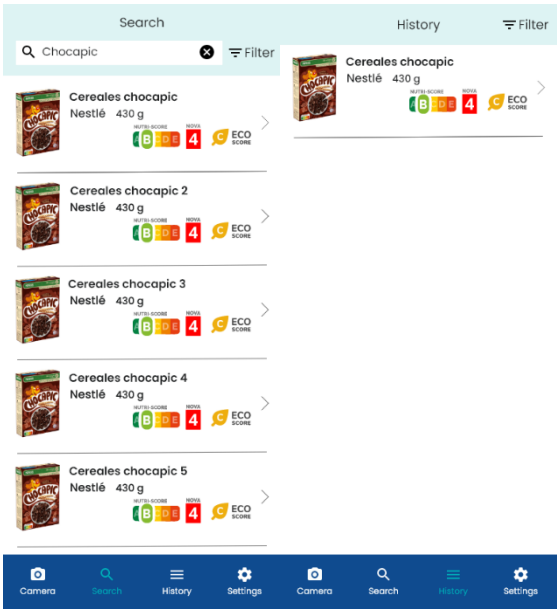


Ilustración 28. Diseño HiFi - Search/History

Y para acabar el usuario en la pantalla de Settings puede configurar que es lo que quiere ver como destacado en las listas de la aplicación.

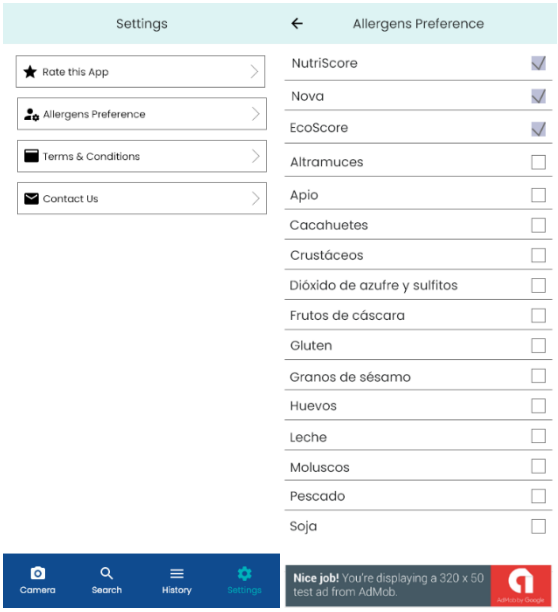


Ilustración 29. Diseño HiFi. Settings/Allergens

2.3 Diseño Técnico

2.3.1 Casos de uso

Los casos de uso identifican las funcionalidades del sistema desde el visto de usuario. A continuación, se muestran los casos de uso y sus especificaciones.

CU_001 – Mostrar onboarding		
Nombre	Mostrar onboarding	
Prioridad	Alta	
Actores	Todos los usuarios	
Descripción	El sistema informa al usuario mediante un Onboarding	
Precondición		
Iniciado por	Usuario	
Flujo	1	El usuario inicia la aplicación por primera vez
Postcondición	El sistema viaja a la pantalla de escaneo de productos.	

CU_002 – Escanear productos		
Nombre	Escanear productos	
Prioridad	Alta	
Actores	Todos los usuarios	
Descripción	El sistema debe permitir escanear *códigos ean* mediante la cámara.	
Precondición	El sistema debe tener la cámara preparada para capturar imagen	
Iniciado por	Usuario	
Flujo	1	El usuario inicia la aplicación por segunda vez o si es la primera pasa del Onboarding (CU_000).
	2	El usuario enfoca a un código EAN.
	3	El sistema traduce el código EAN a números.
	4	El sistema realiza la búsqueda mediante la API del producto.
Postcondición	El sistema registra los productos en el histórico y muestra un botón para poder ver el listado de productos.	

CU_003 – Listar productos escaneados		
Nombre	Listar productos escaneados.	
Prioridad	Alta	
Actores	Todos los usuarios	
Precondición	El usuario a escaneado uno o múltiples códigos ean mediante la cámara. (Caso de uso CU_002).	
Iniciado por	Usuario	
Flujo	1	El usuario hace clic en el icono que aparece al escanear uno o múltiples códigos.
Postcondición	El sistema muestra la lista de productos de usuario.	

CU_004 – Listar histórico		
Nombre	Listar histórico	
Prioridad	Alta	
Actores	Todos los usuarios	
Precondición	El sistema de encuentra en las pantallas principales, donde es visible la barra de navegación inferior.	
Iniciado por	Usuario	
Flujo	1	El usuario hace clic en la barra de navegación inferior y selecciona Histórico.
	2	El sistema busca y muestra los productos en el histórico.
Postcondición	El sistema muestra la lista de productos de usuario.	

CU_005 – Buscar productos		
Nombre	Listar histórico	
Prioridad	Alta	
Actores	Todos los usuarios	
Precondición	El sistema de encuentra en las pantallas principales, donde es visible la barra de navegación inferior.	
Iniciado por	Usuario	
Flujo	1	El usuario hace clic en la barra de navegación inferior y selecciona Buscar.
	2	El usuario escribe el producto o código de barras que quiere buscar.
	3	El sistema realiza la búsqueda mediante la API del producto.
Postcondición	El sistema muestra el resultado de la búsqueda al usuario.	

CU_006 – Filtrar por alergenos		
Nombre	Filtrar por alergenos	
Prioridad	Media	
Actores	Todos los usuarios	
Precondición	El sistema de encuentra en el caso de uso CU_003, CU_004, CU_005.	
Iniciado por	Usuario	
Flujo	1	El usuario hace clic en el icono en la barra superior derecha.
	2	El sistema muestra al usuario el listado de alergenos.
	3	El usuario selecciona los alergenos que quiere filtrar
Postcondición	El sistema retira del listado de productos los que contienen esos alergenos.	

CU_007 – Configurar alergenos		
Nombre	Configurar alergenos	
Prioridad	Media	
Actores	Todos los usuarios	
Precondición	El sistema se encuentra en la pantalla de Settings.	
Iniciado por	Usuario	
Flujo	1	El usuario hace clic en la barra de alergenos.
	2	El sistema muestra al usuario el listado de alergenos.
	3	El usuario selecciona tres alergenos y vuelve a la pantalla anterior.
Postcondición	El sistema cambia los tres elementos destacados de los productos que muestra en los listados.	

CU_007 – Calificar la aplicación		
Nombre	Calificar la aplicación	
Prioridad	Media	
Actores	Todos los usuarios	
Precondición	El sistema se encuentra en la pantalla de settings.	
Iniciado por	Usuario	
Flujo	1	El usuario hace clic en la barra de “Calificar la aplicación”
Postcondición	El sistema califica la aplicación en la AppStore mediante el uso del modal nativo.	

CU_008 – Contáctanos		
Nombre	Contáctanos	
Prioridad	Media	
Actores	Todos los usuarios	
Precondición	El sistema se encuentra en la pantalla de settings.	
Iniciado por	Usuario	
Flujo	1	El usuario hace clic en la barra de “Contáctanos”.
Postcondición	El sistema lanza la aplicación de e-mail para que el usuario pueda contactar al servicio técnico.	

CU_009 – Vista producto individual		
Nombre	Vista producto individual	
Prioridad	Alta	
Actores	Todos los usuarios	
Precondición	El sistema se encuentra en los casos de uso CU_003, CU_004 y CU_005.	
Iniciado por	Usuario	
Flujo	1	El usuario hace en un producto.
Postcondición	El sistema muestra el visual de vista individual y almacena los datos del usuario	

CU_010 – Mantenimiento histórico		
Nombre	Mantenimiento histórico	
Prioridad	Baja	
Actores	Todos los usuarios	
Precondición	El sistema tiene almacenado los datos en el histórico de búsquedas.	
Iniciado por	Sistema	
Flujo	1	El sistema comprueba la cantidad de productos almacenados.
	2	Si el sistema contiene más de 30 elementos debe borrar el elemento que se insertó hace más tiempo.
Postcondición	El sistema siempre tiene 30 elementos o menos almacenados como histórico.	

2.3.2 Arquitectura del sistema

Al principio del proyecto se tomó la elección de trabajar utilizando Flutter, debido a esta decisión utilizaremos los patrones más comunes y que mejor encajan con el proyecto.

Para empezar, utilizaremos el patrón *BLoC* para gestionar las interacciones de las vistas (substituyendo el patrón MVVM).

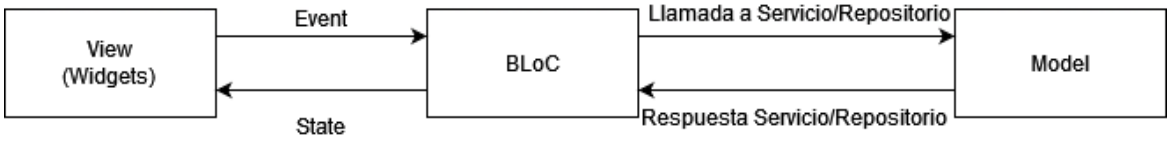


Ilustración 30. Patrón BLoC

Podemos ver cierta similitud al MVVM ya que lo que hemos hecho es substituir el *ViewModel* por *BLoC*. La idea del patrón es separar la capa de *presentación* de la *lógica de negocio*, la capa de *presentación* envía al *BLoC* mediante un *Stream* los eventos que se han producido en el visual (interacciones del usuario) y a su vez la vista de suscribirá a ese *Stream* para escuchar los cambios de estados que se produzcan (sean motivados por la vista o por algún factor externo que interactúe con el *BLoC*).

Además del uso de *BLoC*, también separaremos el código en cuatro capas:

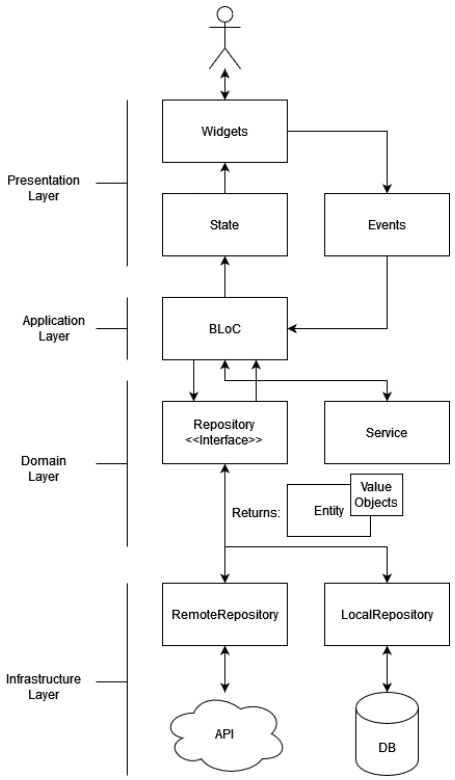


Ilustración 31. Separación por capas

- UI: La capa donde tendremos todas las vistas y los widgets.
- Application: Donde tendremos los casos de uso o *BLoC*.
- Domain: Donde residen nuestras entidades, los procesos y las reglas del negocio de la aplicación.
- Infrastructure: La última capa que gestiona las conexiones y los accesos a los datos (conexiones vía api, almacenamiento interno…).

Al separar por estas capas aumentaremos la mantenibilidad del código a la hora de hacer realizar futuras iteraciones sobre el producto.

Otro de los retos es el uso de la cámara y el escaneo de los *códigos ean*, *Flutter*, como *framework* multiplataforma, no es capaz de acceder al hardware nativo (cámara, micrófono, altavoces, osciloscopio…), para realizar esto, *Flutter* nos proporciona la posibilidad de utilizar vistas nativas de *iOS* o *Android*.

Estas vistas nativas son situadas por *Flutter* por composición en los visuales del *framework*, es decir, la vista puede estar situada y seguir utilizando los demás widgets de *Flutter*.

Aparte de estas vistas también tendremos que definir un *Method Channel* que nos servirá para comunicarnos de forma bidireccional a las acciones, es decir, cada vez que encontremos un *código ean* le enviaremos un mensaje a *Flutter* para que gestione la búsqueda del producto.

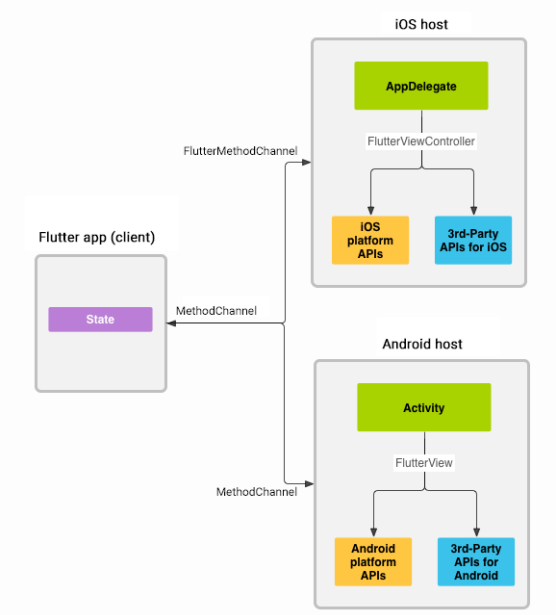


Ilustración 32. Flutter MethodChannel

2.3.3 Modelo de datos

La fuente de información principal de datos de la aplicación es la API de OpenFoodFacts, pero el aplicativo guarda un histórico de las ultimas 30 búsquedas así que lo almacenara en local.

Para almacenar estos datos en local se utilizará *Hive* (https://pub.dev/packages/hive) una librería de datos Llave-Valor que nos permitirá guardar los objetos de dominio directamente en disco para ser utilizados más tarde.

Lo que guardaremos será simplemente la representación del producto y el binario de la imagen que descargamos.

Nuestro código de declaración de tipo *Hive* quedara tal que así:

```
enum NutriScore {
    A,
    B,
    C,
    D,
    E,
}

enum Nova {
    1,
    2,
    3,
    4
}

enum EcoScore {
    A,
    B,
    C,
    D,
    E,
}

@HiveType(typeId: 0)
class Product {
    @HiveField(0)
    final String name;

    @HiveField(1)
    final UInt8List image;

    @HiveField(2)
    final Nutriscore nutriscore;
```

```
    @HiveField(3)
    final Nova nova;

    @HiveField(4)
    final EcoScore ecoScore;

    @HiveField(5)
    final double grasas;

    @HiveField(6)
    final double gSaturadas;

    @HiveField(7)
    final double azucar;

    @HiveField(8)
    final double sal;

    @HiveField(9)
    final List<String> alergenos;
}
```

Con esta clase y los decoradores agregamos a la Box[10], la definición de los datos y con esto podremos añadir elementos a la box de la siguiente manera:

```
final box = await Hive.openBox<Product>('products');

box.add(productItem);
```

Y luego podremos iterar sobre los productos guardados para mostrarlos:

```
final productBox = Hive.box('products');

for (var i = 0; i < productBox.lenght; i++) {
    final product = productBox.get(i) as Product
}
```

[10] Equivalente a tabla SQL.

3. Implementación

En esta sección se exponen los aspectos más relevantes de la implementación y los métodos utilizados.

Para iniciar el proyecto se ha utilizado *Intellij IDEA* con el plugin de *Flutter*, que permite arrancar un nuevo proyecto seleccionando en que lenguajes nativos vamos a trabajar.

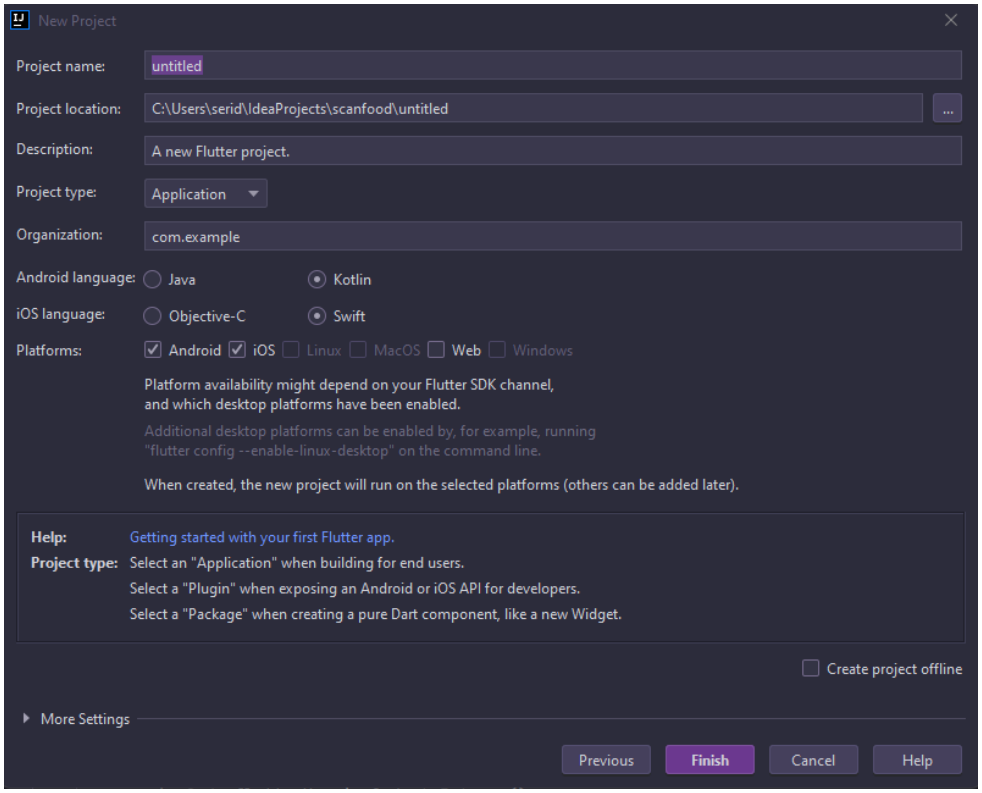


Ilustración 33. Arranque proyecto Intellij IDEA.

Para este mismo propósito se puede utilizar también *Android Studio* ya que también dispone del plugin de *Flutter*.

Una vez creado el proyecto, tendremos la estructura típica de *Flutter*, donde tenemos la carpeta lib (donde reside el código Dart), *Android* (donde reside la parte nativa de *Android*) y *iOS* (donde tenemos el proyecto que podremos abrir con *XCode*).

A partir de este punto, podremos compilar en Windows la aplicación en *Android* y en *MacOS* la aplicación para *Android* y *iOS*. Como los visuales y la lógica principal de la aplicación se sitúa en la parte de *Dart*, se realiza la gran parte del trabajo en el ordenador *Windows* (ya que el ordenador con *Mac* es una cesión).

3.1 Librerías y Patrones en Flutter

El proyecto utiliza múltiples librerías de *Flutter* para facilitar el desarrollo, estas las he dividido entre diferentes tipologías.

3.1.1 Librerías Flutter: Core

Las librerías que he denominado Core son las principales de la aplicación, estas son utilidades o extensiones que instalamos para facilitarnos el desarrollo, estas librerías son:

- ***Get It*** (https://pub.dev/packages/get_it)

  La librería Get It es un *Service Locator*, básicamente, es una clase que conoce como se obtener todos los servicios de nuestra aplicación, con esta librería podemos registrar todas las clases/servicios y recogerlos en otro lugar de la aplicación (sea construyendo una nueva instancia o recogiendo un *singleton* de memoria).

- ***Injectable*** (https://pub.dev/packages/injectable)

  Injectable es una librería que ayuda a *Get It* automatizando la construcción y el registro de las clases/servicios. Esta librería nos da diversos decoradores que podemos utilizar para registrar las clases o para inyectar los servicios por constructor.

- ***Uuid*** (https://pub.dev/packages/uuid)

  Genera *UUIDs* válidos, se utiliza para generar canales de comunicación a nativo.

- ***Flutter Bloc*** (https://pub.dev/packages/bloc)

  La librería *BLoC* viene a implementar el patrón de diseño con el mismo nombre (*Business Logic Component*).

  Este patrón de diseño ayuda a separar la lógica de negocio de la vista. De esta manera facilitando la reutilización y testeabilidad del producto.

Ilustración 34. Patrón BLoC

La vista está conectada a un *stream* de eventos de la clase que implementa la librería, este *stream* envía estados de actualización a los

que actualizan la vista y la vista a su vez envía los eventos de los inputs recibidos al *BLoC*.

- ***In App Review*** (https://pub.dev/packages/in_app_review)

  La librería nos permite sacar un modal nativo de puntuación para la aplicación (utilizando en *iOS SKStoreReviewController* y la API de In-App Review en Android).

- ***Permission Handler*** (https://pub.dev/packages/permission_handler)

  Esta librería como la anterior, nos crea una abstracción con la que pedir permisos nativos del usuario, en lugar de tener que implementar en ambas cámaras la solicitud de los permisos, desde *Flutter* pedimos con una línea de código los permisos de cámara con esta librería.

- ***Equatable*** (https://pub.dev/packages/equatable)

  Esta librería simplifica el proceso de comparar dos objetos complejos permitiendo definir una lista de elementos que serán comparados uno a uno para asegurar que los objetos son iguales.

  Se utiliza en los estados y eventos de BLoC ya que el stream de datos descarta los inputs que son exactamente iguales (dobles clicks en botones, por ejemplo) y tampoco envía actualizaciones de estado que sean iguales a la última enviada (evitando renderizar de nuevo el visual sin cambios).

- ***Url Launcher*** (https://pub.dev/packages/url_launcher)

  Con ella podemos abrir urls de forma nativa con una simple línea desde *Flutter* (ya sea para abrir un navegador o otra aplicación mediante los url scheme).

- ***Get Storage*** (https://pub.dev/packages/get_storage)

  Es una librería para almacenar datos estilo llave-valor similar a *shared preferences*, la diferencia es que está totalmente programada en *Flutter* haciendo que funcione en todos los lugares donde el motor de *Flutter* está disponible.

  La librería se utiliza para saber si el usuario ha pasado por la pantalla de Onboarding y para almacenar pequeños datos de información.

- ***Hive*** (https://pub.dev/packages/hive)

  Esta es la que se utiliza como base de datos para cachear los datos que recogemos desde la API, funciona también por clave-valor, pero los valores son más complejos. Con este sistema se crean clases de tipo *Hive*

que se pasan por un generador de código para saber cómo ser guardadas en un Box (que a efectos prácticos funciona como un array).

Sobre este box podemos realizar una selección de los ítems por su ID o recogerlos todos, para insertarlos simplemente les daremos una ID y lo insertaremos.

- ***Openfoodfacts*** (https://pub.dev/packages/openfoodfacts)

  Es la librería que nos proporciona la asociación propietaria de los datos. Con esta librería estaremos abstraídos de tener que programar a mano las llamadas a la API ya que la librería nos las encapsula.

3.1.2 Librerías Flutter: Visuales

Estas librerías son las que por su utilidad visual utilizamos en el proyecto:

- ***Introduction Screen*** (https://pub.dev/packages/introduction_screen)

  Contiene la lógica para mostrar la página de Onboarding, podemos crear pantallas de Onboarding dando una imagen o widget y un widget o un título para la parte inferior.

- ***Lottie*** (https://pub.dev/packages/lottie)

  Lottie es una librería creada por Airbnb, esta librería permite que los diseñadores puedan exportar animaciones realizadas con el programa *After Effects* en un formato que esta misma librería nos permite importar en nuestra aplicación y renderizar esta.

  Se utiliza para las animaciones del Onboarding. Estas han sido extraídas de lottiefiles.

Animación	Url
Animación 1 – Escaneo	https://lottiefiles.com/8998-scanning
Animación 2 – Food Bowl	https://lottiefiles.com/24703-food-animation
Animación 3 – Healthy or Junk	https://lottiefiles.com/36895-healthy-or-junk-food

- ***Flutter SVG*** (https://pub.dev/packages/flutter_svg)

  Los iconos fueron realizados en svg, así que *Flutter* necesita una librería para poderlos renderizar. Esta misma convierte los trazos del SVG a un formato que *Flutter* puede renderizar en la pantalla

- ***Top Snackbar Flutter*** (https://pub.dev/packages/top_snackbar_flutter)

Esta última librería es la encargada de sacar el *snackbar* superior en la pantalla de escaneo cuando un código de barras no es encontrado en OpenFoodFacts.

3.1.3 Librerías de Firebase

La aplicación también tiene 3 librerías de *Firebase* integradas: *Core*, *Analytics* y *Crashlytics*.

Estas tres integraciones permiten recoger feedback de la aplicación publicada, con *Analytics* podemos analizar que paginas son las que los usuarios utilizan y extraer datos útiles como el uso del buscador o el uso de la cámara, pudiendo hacer foco en mejorar esas partes en las siguientes iteraciones y con *Crashlytics* la aplicación nos informara de los errores que los usuarios se encuentren mientras ejecutan la aplicación.

3.2 Estructura del proyecto

Para la parte de *Flutter* se ha organizado siguiendo una arquitectura por capas.

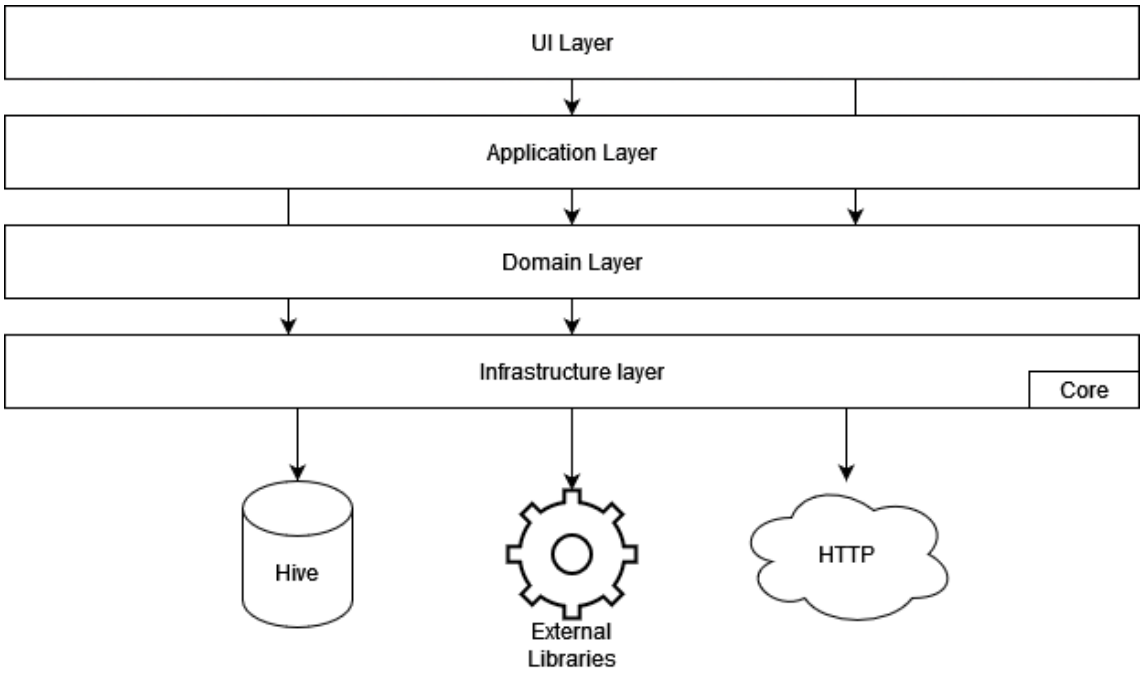


Ilustración 35. Organización por capas.

La arquitectura sigue los principios del *DDD* (Domain Driven Design), el diseño del producto se centra en la capa de Dominio, donde expresaremos todos los repositorios, objetos y procedimientos de la aplicación.

Las capas que podemos ver son:

- ***UI Layer***: La capa de presentación se limita simplemente a la parte visual del proyecto. Esta capa es la encargada de mostrar los datos y de recibir los inputs del usuario (y los inputs del *framework*, como pueden ser los cambios de estado del teléfono móvil).

- ***Application Layer***: En esta nos encontramos servicios (pequeños controles de lógica) y clases de tipo *BLoC* para gestionar los estados en la aplicación, habitualmente cada vista tendrá asociada una clase de tipo BLoC (si es que tiene estados).

- ***Domain Layer***: Esta capa contiene las entidades y repositorios con los que trabaja nuestra aplicación, en esta capa nos encontramos clases abstractas que serán reimplementadas en la capa de infraestructura que es la que puede comunicarse con librerías y *APIs* externas.

- ***Infrastructure Layer***: Es la última capa de la aplicación y la responsable de comunicarse con los servicios externos (sean librerías o *APIs*), y de trabajar con las bases de datos locales.

3.3 Comunicación Bridge Flutter-Nativo

Utilizando *Flutter* no tenemos la capacidad de acceder a elementos físicos del dispositivo, la cámara, micrófono, sensores, gps… El *framework* te ofrece la posibilidad de incluir vistas nativas dentro del sistema de *Flutter*, haciendo así que una vista de nativo queda integrada en el diseño de la aplicación.

Esto en si no es suficiente para la aplicación así que *Flutter* también tiene la posibilidad de enviar mensajes entre nativo y el motor de *Flutter*, esto se realiza con los *method channel*, estos son un canal de comunicación bidireccional y donde esta comunicación puede ser iniciada tanto desde el lado de nuestro código *Flutter* como desde el código nativo[11].

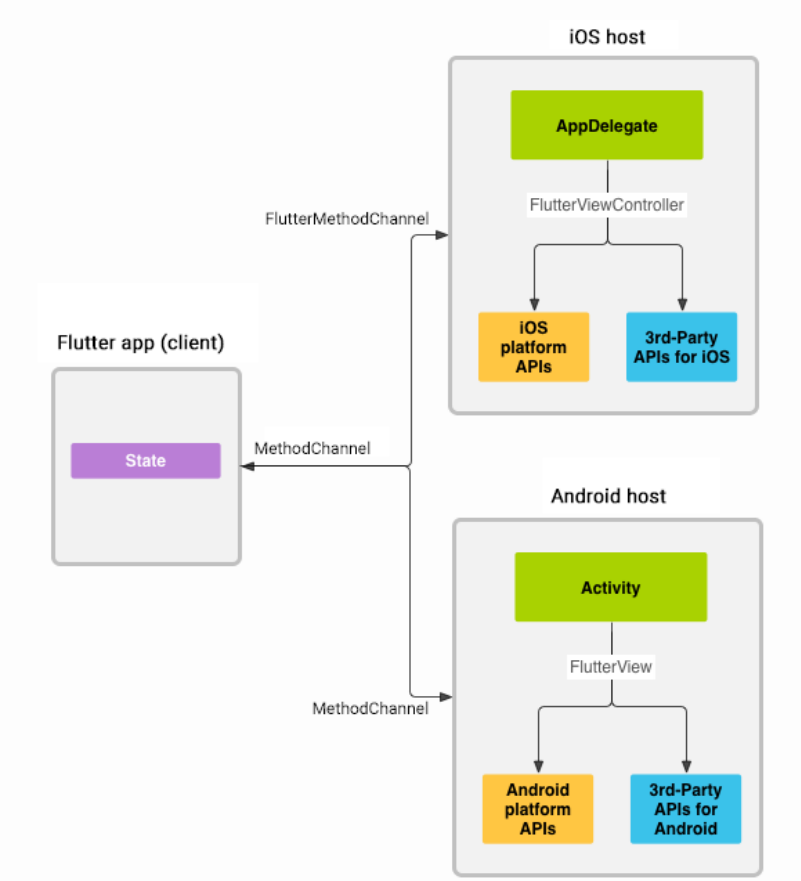


Ilustración 36. Comunicación Nativo-Flutter - Method Channel

Por lo tanto, si nosotros definimos una interfaz de comunicación común en Flutter, solo nos quedaría tener que programar las partes nativas que cumplan el contrato definido para poder tener el código en multiplataforma.

[11] Cuando mencionamos código nativo nos referimos a:
- Kotlin o Java en Android
- Swift o Objective-c en iOS
- C++ en Windows
- Objective-c en MacOS
- C en Linux

En este caso hemos estructurado nuestro código de *Flutter* dentro de la capa de *infrastructure* ya que entendemos nativo como algo externo a nuestro dominio de *Flutter*. Allí separamos en dos conceptos, vista y controlador. La vista es la encargada de renderizar y colocar la *FlutterView* dentro de nuestro ecosistema y el controlador, es el encargado de gestionar la mensajería nativa.

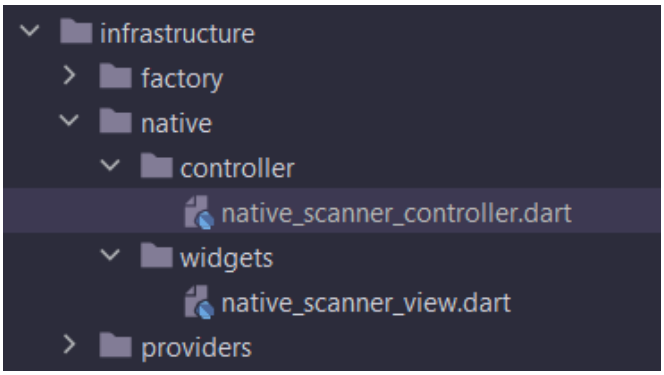


Ilustración 37. Referencia estructura flutter-nativo.

En este caso, nosotros querremos mostrar una cámara en tiempo real en nuestra native_scanner_view y recibir todos los códigos de barras encontrados en tiempo real mediante native_scanner_controller.

Nuestro controlador, define un canal de eventos (*EventChannel*, que nos enviara los datos como una secuencia de eventos asíncronos) y un canal síncrono (*MethodChannel*) que utilizaremos para avisar a *Flutter* que la cámara esta activa y que ya podemos iniciar la subscripción al canal de eventos.

Por este *EventChannel* enviaremos los *Strings* que detecta nuestro lector para que desde Flutter podamos buscarlos mediante la API de openfoodfacts (o cargarlos de cache). Estos datos que se envían por los canales de comunicación son convertidos de forma automática entre los diferentes formatos[12] utilizando un *message codec*.

Por el otro lado, la vista simplemente renderiza un objeto de tipo *AndroidView* o *UIKitView* en función de cuál es la plataforma sobre la que estamos ejecutando Flutter, esta vista se inicializa indicando cual es el tipo de vista que esperamos renderizar (en este caso la hemos titulado *scanfood/native-scanner/view*) canal de comunicación para el *MethodChannel* y *EventChannel* (estos canales los hemos titulado *scanfood/native-scanner/actions/random-uuid*).

[12] https://api.flutter.dev/flutter/services/StandardMessageCodec-class.html

3.3.1 Android

Para desarrollar la parte de Android, hemos decidido utilizar la librería *KBarcode*, esta librería nos abstrae de la necesidad de arrancar la cámara y programar la detección de los códigos de barras, esta librería está programada en *Kotlin* y con dependencias modernas (*camera2*, *mlkit*) asegurando el funcionamiento en dispositivos modernos.

Al utilizar esta librería también nos podemos centrar en la parte importante que es el canal de comunicación y que el funcionamiento de todo sea correcto.

La estructura del proyecto de Android tiene la siguiente estructura:

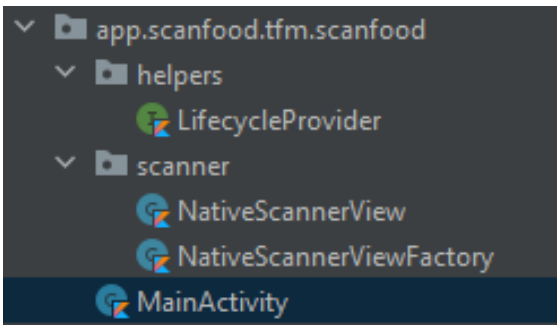


Ilustración 38. Estructura Android

Lo primero que tenemos que hay que realizar es declarar una vista nativa, para ello tenemos que crear una *ViewFactory*, que es la encarga de devolver nuestra *PlatformView* cada vez que *Flutter* solicite incrustarla.

```
class MainActivity: FlutterActivity() {
    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)

        flutterEngine
                .platformViewsController
                .registry
                .registerViewFactory( viewTypeId: "scanfood/native-scanner/view", NativeScannerViewFactory(object : LifecycleProvider {
                    override fun getLifecycle(): Lifecycle {
                        return (activity as LifecycleOwner).lifecycle
                    }
                }, flutterEngine.dartExecutor.binaryMessenger))
    }
}
```

Ilustración 39. Código Android - Añadir *ViewFactory* al *Engine*

Como podemos ver en la anterior figura, añadimos al motor de *Flutter* una factoría de vistas con el tipo de vista que definimos en *Flutter* y ahí añadimos nuestra factoría.

```
class NativeScannerViewFactory(
    private val lifecycleProvider: LifecycleProvider,
    private val binaryMessenger: BinaryMessenger): PlatformViewFactory(StandardMessageCodec.INSTANCE) {
    override fun create(context: Context, viewId: Int, args: Any?): PlatformView {
        val creationParams = args as Map<String?, Any?>?
        return NativeScannerView(context, viewId, creationParams, lifecycleProvider, binaryMessenger)
    }
}
```

Ilustración 40. Código Android – *ViewFactory*

Esta ViewFactory nos crea una Vista que cumple las siguientes interfaces:

- *DefaultLifecycleObserver*

  Esta interfaz la utilizamos para subscribirnos a los cambios de estado de la aplicación y así tener acceso a los métodos *onCreate*, *onPause*… Estos los utilizamos para encender, parar, resumir nuestro detector.

- *PlatformView*

  Esta interfaz nos obliga a devolver una *View* de Android, esta corresponde a la *BarcodeView* que nos proporcionara la librería de escaneo.

- *EventChannel.StreamHandler*

  Nos dará acceso a los métodos *onListen*, *onCancel* que utilizaremos para registrar nuestro *EventSink* por donde enviaremos los códigos de barras. Cuando *Flutter* cree el canal de eventos automáticamente se recibirá por el metodo *onListen* el *eventSink* y cuando *Flutter* elimine el stream se recibirá por *onCancel* la eliminación de este.

Cuando nuestra vista arranca simplemente preparemos la librería, nos registraremos en el *eventChannel*, añadiremos el observador del ciclo de vida de Android y avisaremos a *Flutter* que estamos listos para enviar eventos.

```
init {
    eventChannel.setStreamHandler(this)
    methodChannel.invokeMethod( method: "event_channel_ready",  arguments: null);
    lifecycleProvider.getLifecycle()?.addObserver( observer: this);

    barcodeView.setOptions(
            Options.Builder()
                    .cameraFacing(CameraCharacteristics.LENS_FACING_BACK)
                    .cameraFlashMode(CameraMetadata.FLASH_MODE_OFF)
                    .barcodeFormats(
                            intArrayOf(
                                    Barcode.FORMAT_EAN_13, Barcode.FORMAT_EAN_8
                            )
                    )
                    .barcodesSort( barcodesSort: null)
                    .previewScaleType(BarcodeView.CENTER_CROP)
                    .clearFocusDelay(BarcodeView.CLEAR_FOCUS_DELAY_DEFAULT)
                    .build()
    )
}
```

Ilustración 41. Código Android - Arranque Vista

Y por último solo nos queda registrarnos al *Listener* de la librería para enviar los eventos a *Flutter*.

```
barcodeView.onBarcodeListener = OnBarcodeListener { barcode ->
    if(barcode.rawValue != null) {
        eventSink?.success(barcode.rawValue)
    }
}
```

Ilustración 42. Código Android - Envió de eventos

3.3.2 iOS

La estructura de la parte de la aplicación de *iOS* ha quedado de la siguiente manera:

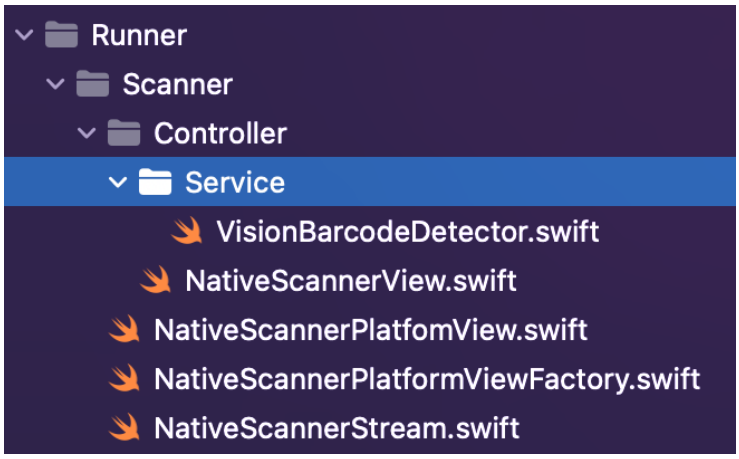


Ilustración 43. Estructura iOS

A diferencia de la parte de Android en este punto no se ha utilizado una librería externa para realizar la lectura de códigos de barras, sino que se ha utilizado la librería *Vision* de Apple que permite a partir de una imagen determinar si hay códigos en la pantalla.

```
typealias CallbackBarcode = ((String) -> Void)

class VisionBarcodeDetector {
    private let sequenceHandler = VNSequenceRequestHandler()

    private func completeImageRequest(
        for frame: CVImageBuffer,
        completion: CallbackBarcode
    ) {
        let barcodeDetectionRequest: VNDetectBarcodesRequest = {
            let barcodeDetectRequest = VNDetectBarcodesRequest()

            barcodeDetectRequest.symbologies = [.ean8, .ean13]

            return barcodeDetectRequest
        }()

        try? self.sequenceHandler.perform([barcodeDetectionRequest], on: frame)
        guard let firstBarcode = barcodeDetectionRequest.results?.first?.payloadStringValue else {
            return
        }

        completion(firstBarcode)
    }

    func barcode(forPixelBuffer pixelBuffer: CVImageBuffer, completion: CallbackBarcode) {
        self.completeImageRequest(for: pixelBuffer, completion: completion)
    }
}
```

Ilustración 44. Código iOS - Detector códigos de barras

Para implementar la parte de visualización en directo se ha utilizado la librería *AVFoundation* de Apple, con esta sacamos en tiempo real por pantalla (con *AVCaptureVideoPreviewLayer*) o que la cámara está capturando en ese momento, y además se recoge cada uno de los *frames* capturados utilizando el protocol *AVCaptureVideoDataOutputSampleBufferDelegate* y estos se envían al detector de *Vision*.

```
// MARK: - AVCaptureVideoDataOutputSampleBufferDelegate
func captureOutput(_ output: AVCaptureOutput, didOutput sampleBuffer: CMSampleBuffer, from connection: AVCaptureConnection) {
    guard let pixelBuffer = CMSampleBufferGetImageBuffer(sampleBuffer) else {
        return
    }

    self.visionBarcodeDetect.barcode(forPixelBuffer: pixelBuffer) { barcode in
        delegate?.barcodeOutput(string: barcode)
    }
}
```

Ilustración 45. Código iOS - *AVCaptureVideoDataOutputSampleBufferDelegate*

El *VideoPreviewLayer* se añade como *subLayer* en una *UIView* que implementa el protocolo antes mencionado, esta *subview* define un protocolo para que la *PlatformView* que la contiene lo implemente.

```
public protocol NativeScannerViewDelegate: NSObjectProtocol {
    func barcodeOutput(string: String?)
}
```

Ilustración 46. Código iOS - *NativeScannerViewDelegate* protocol

La *PlatformView* implementa el *delegate* y reenvía los códigos capturados hacía el *EventHandler* de datos.

```
func barcodeOutput(string: String?) {
    if let string = string {
        nativeScannerStream?.sendBarcode(string)
    }
}
```

Ilustración 47. Código iOS - Implementación *NativeScannerViewDelegate*

La parte de comunicaciones con *iOS* funciona de forma similar a la de *Android*, primero registramos la ViewFactory en el engine de *Flutter*.

```
@UIApplicationMain
@objc class AppDelegate: FlutterAppDelegate {
  override func application(
    _ application: UIApplication,
    didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?
  ) -> Bool {
    GeneratedPluginRegistrant.register(with: self)

    NativeScannerPlatformViewFactory.register(with: registrar(forPlugin: "NATIVE_SCANNER")!)

    return super.application(application, didFinishLaunchingWithOptions: launchOptions)
  }
}
```

Ilustración 48. Código iOS - Añadir *ViewFactory* al *Engine 1/2*

```
// Pragma: Register Factory
public static func register(with registrar: FlutterPluginRegistrar) {
    registrar.register(
        NativeScannerPlatformViewFactory(messenger: registrar.messenger()),
        withId: "scanfood/native-scanner/view"
    )
}
```

Ilustración 49. Código iOS - Añadir ViewFactory al Engine 2/2

También tenemos el fragmento de código que monta las vistas a partir de la petición de *Flutter*, si nos fijamos podemos ver como el *Map* que hemos enviado desde *Flutter* se convierte en una *NSDictionary* al llegar a Swift.

```
func create(withFrame frame: CGRect, viewIdentifier viewId: Int64, arguments args: Any?) -> FlutterPlatformView {
    let arguments = args as? NSDictionary
    let methodChannel = arguments?["method_channel_id"] as? String
    let eventChannel = arguments?["event_channel_id"] as? String

    return CameraPhotoPlatformView(
        frame: frame,
        viewIdentifier: viewId,
        binaryMessenger: messenger,
        withMethodChannelId: methodChannel,
        andEventChannelId: eventChannel
    )
}
```

Ilustración 50. Código iOS – *ViewFactory*

Y como en Android también arrancamos todos los servicios al iniciar la PlatformView.

```
        self.frame = frame
        self.viewId = viewId
        _view = NativeScannerView(frame: frame)
        guard let methodChannelId = methodChannelId,
              let eventChannelId = eventChannelId else {
            nativeScannerStream = nil

            super.init()

            return
        }

        let eventChannel = FlutterEventChannel(name: eventChannelId, binaryMessenger: messenger)
        nativeScannerStream = NativeScannerStream()
        eventChannel.setStreamHandler(nativeScannerStream)

        let channel = FlutterMethodChannel(name: methodChannelId, binaryMessenger: messenger)

        channel.invokeMethod("event_channel_ready", arguments: nil)

        _view.startReader()

        _view.backgroundColor = UIColor(displayP3Red: 0, green: 0, blue: 0, alpha: 1)

        super.init()

        _view.delegate = self
```

Ilustración 51. Código iOS - Arranque Vista

Como podemos ver también registramos el *EventHandler* aunque en este caso lo hacemos hacia el protocolo lo hemos implementado en otra clase llamada *NativeScannerStream*. Esta clase tiene los métodos *onListen* y *onCancel* como en *Android*.

```
class NativeScannerStream: NSObject, FlutterStreamHandler {
    private var eventSink: FlutterEventSink?

    func onListen(withArguments arguments: Any?, eventSink events: @escaping FlutterEventSink) -> FlutterError? {
        self.eventSink = events

        return nil
    }

    func onCancel(withArguments arguments: Any?) -> FlutterError? {
        self.eventSink = nil

        return nil
    }

    func sendBarcode(_ barcode: String) {
        eventSink?(barcode)
    }
}
```

Ilustración 52. Código iOS - *NativeScannerStream*

3.4 Landing Web – Política privacidad

URL Web: https://scanfood.app

Otro de los apartados entregables de la aplicación es tener una pagina web sencilla como Landing de la aplicación y donde colocar la política de privacidad (obligatoria para la publicación).

Esta página web se ha realizado con *Gatsby* (https://www.gatsbyjs.com/), esta tecnología construye páginas webs estáticas a partir de contenido dinámico con *ReactJS*.

Para *Gatsby* hay una multitud de plantillas disponibles para utilizar, en mi caso he utilizado la plantilla Automatic landing page (github: ImedAdel/automatic-gatsbyjs-app-landing-page). Esta plantilla nos proporciona una base con la que trabajar nuestra web, pudiendo colocar simplemente la imagen principal y cambiar los textos a través de un fichero de configuración.

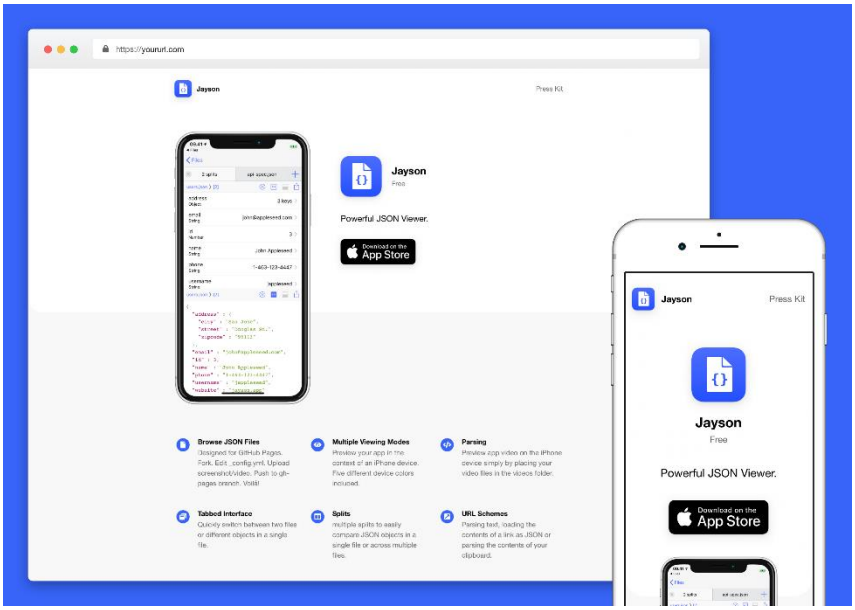


Ilustración 53. Plantilla Gatsby

Sobre la página de política de privacidad y condiciones, se ha revisado varias aplicaciones similares para escribirla y se ha colocado una nueva página en *Gatsby*.

La web ha sido publicada en *Firebase Hosting*, que nos proporciona un hosting gratuito (con tráfico limitado a 10 GB mensuales). La publicación se realiza mediante las herramientas CLI de Firebase.

3.5 Herramientas externas

Para facilitar esta fase de implementación se han contado con varias herramientas externas, estas, aunque son empresariales tienen un *free-tier* que permite ser utilizadas sin tener que realizar un pago.

3.5.1 Traducción *Localizely*

Localizely (https://localizely.com) es una herramienta de traducción que nos abstrae de tener que generar nosotros mismos los ficheros ARB y nos da una buena interfaz con la que trabajar desde un entorno web.

Los creadores de esta herramienta son los que también crearon la librería *Intl Utils* (https://pub.dev/packages/intl_utils), esta permite pasar de ficheros ARB a código autogenerado.

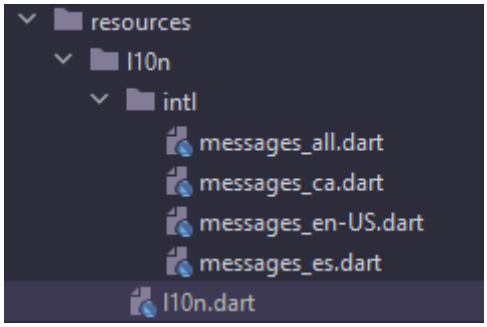


Ilustración 54. Ficheros autogenerados por intl_utils

Este código cumple con las interfaces *LocalizationsDelegate* cosa que nos permite añadir directamente las traducciones a *Flutter*.

```dart
import 'package:scanfood/resources/l10n/l10n.dart';

class ScanFood extends StatelessWidget {
  const ScanFood({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      initialRoute: SplashScreen.ROUTE,
      theme: AppTheme.lightTheme,
      routes: routes,
      localizationsDelegates: const [
        S.delegate,
        GlobalMaterialLocalizations.delegate,
        GlobalCupertinoLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate
      ],
      supportedLocales: S.delegate.supportedLocales,
```

Ilustración 55. Integración *intl_utils* a *Flutter*

3.5.2 CI/CD – *Codemagic*

Codemagic (https://codemagic.io) es una herramienta que permite configurar flujos de construcción de las aplicaciones móviles para poder realizar tareas automatizadas de publicación o compilación. En el paquete gratuito se disponen de 500 minutos de máquinas *MacOS* para realizar las compilaciones de aplicaciones de *Android* y *iOS*.

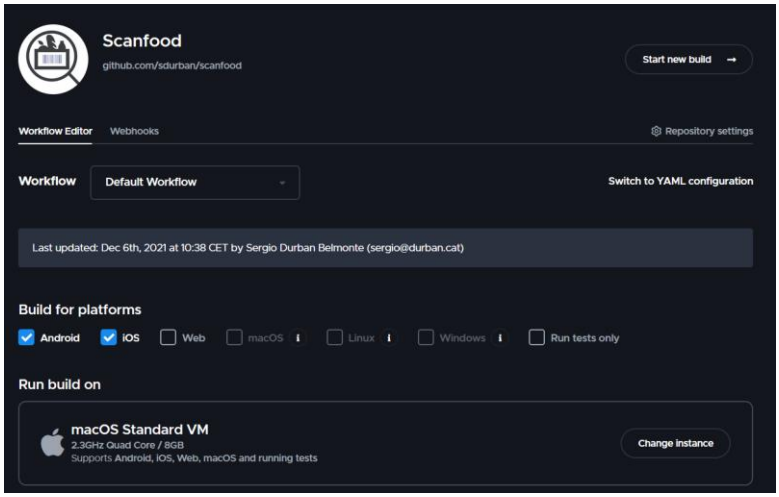


Ilustración 56. Codemagic - Configuración

Este aplicativo tiene un panel en el cual se puede configurar toda clase de opciones. Se pueden ejecutar las pruebas automáticas de la aplicación antes de compilar para asegurar que la aplicación es funcional (parando así la fase de compilación).

Al compilar la aplicación podemos elegir con que certificados firmaremos esta, e incluso publicar directamente la aplicación desde esta plataforma.

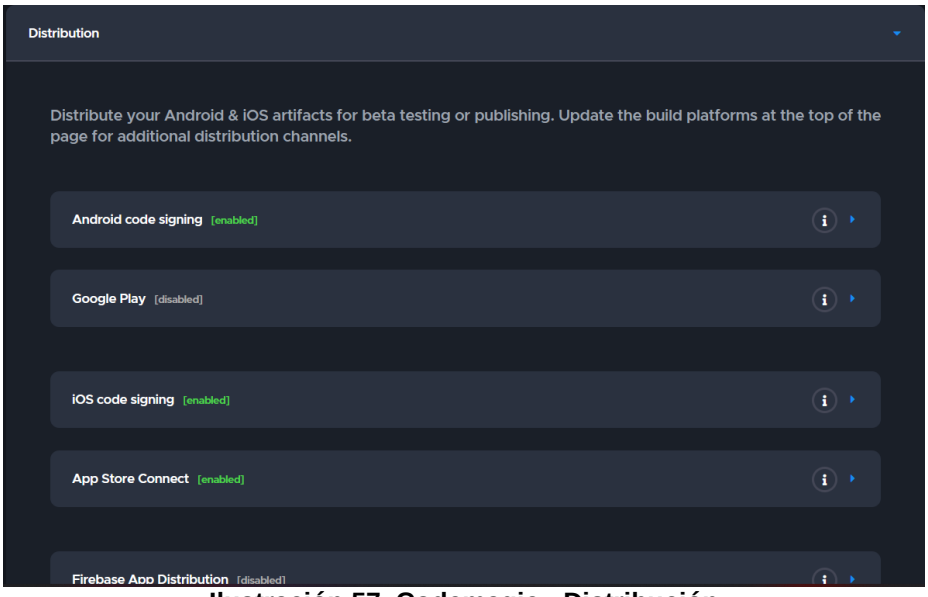


Ilustración 57. Codemagic - Distribución

Esto nos permite poder publicar actualizaciones en *iOS* sin tener que disponer de un *MacOS*, cosa que es viable si no tenemos que actualizar ninguna de las partes del conector de *Flutter*.

Como se ve en la figura anterior también se puede enviar directamente las compilaciones a *Firebase App Distribution*, cosa que nos ayudara a poder comprobar nuestra aplicación en dispositivos iOS antes de salir al mercado.

Como mencione en el apartado de Planificación, el Mac OS solo se utilizó para depurar la aplicación mientras se programaba la parte nativa y el resto de programación se ha realizado desde la maquina con Windows, esta herramienta ayuda a que esto sea posible.

Los flujos se pueden configurar de forma automática para que se activen cuando se hace un push a una rama en concreta, se actualiza una *pull request* o cualquier acción que pueda suceder en *Git*.

Una vez la compilación a arrancado podemos ver en tiempo real como va el proceso, o podemos revisar los logs más adelante para ver posibles errores, *warnings* o descargar los artefactos generados.

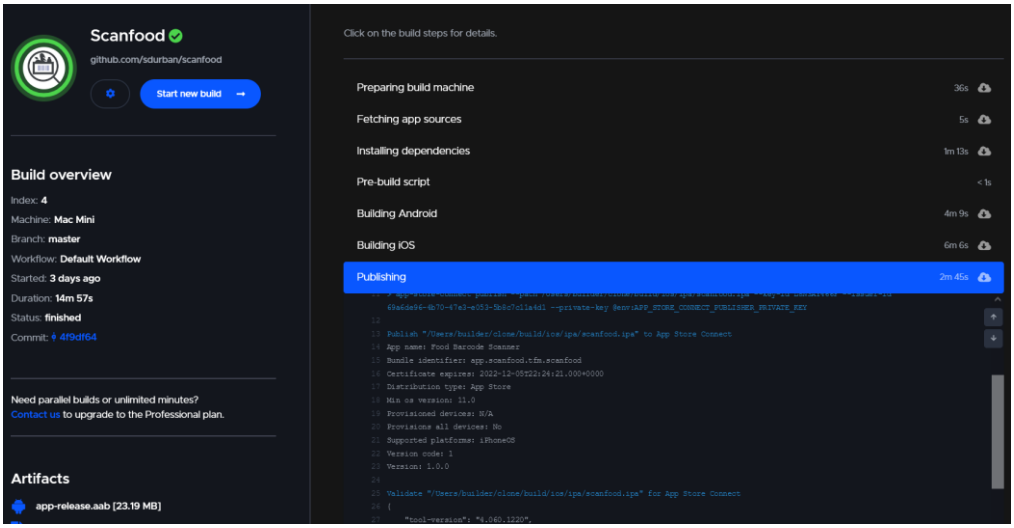


Ilustración 58. Codemagic - Resultado compilación

3.6 Pruebas Unitarias

En la aplicación nos hemos centrado única y exclusivamente en las pruebas unitarias de esta, y solo en las capas *Application* y *Domain.* Estas pruebas cubren todos los blocs (a excepción de cámara y búsqueda por tiempos).

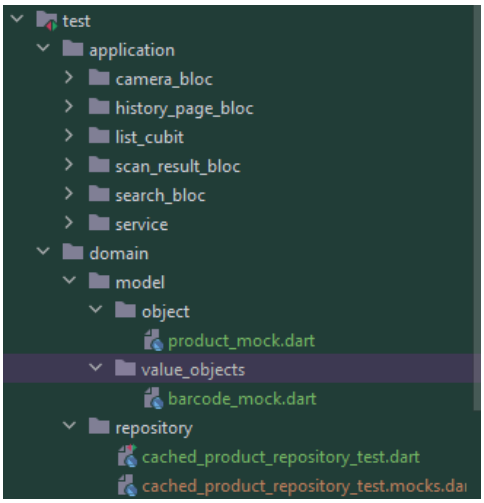


Ilustración 59. Estructura pruebas unitarias

Para realizar estas pruebas utilizamos siguientes las librerías:

- ***Mockito*** (https://pub.dev/packages/mockito)

  Se utiliza para *mocks*, y utilizarlos mediante *stubs* y verificar las llamadas a estos *stubs*.

- ***Bloc Test*** (https://pub.dev/packages/bloc_test)

  Esta librería del autor de BLoC nos ayuda a poder testear el patrón mandando eventos y simplificándonos el trabajo de escuchar al stream de respuesta

- ***Faker*** (https://pub.dev/packages/faker)

  La utilizamos para generar datos falsos, esta librería contiene diferentes métodos para generar datos falsos sean números, nombres de persona, calles, localizaciones…

Las pruebas las tenemos agrupadas por grupos y por objetivo a probar de manera que al ejecutarnos nos queda un resultado similar a la siguiente ilustración.

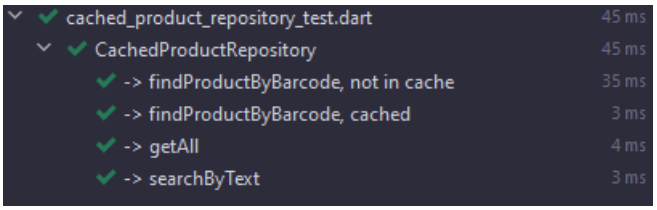


Ilustración 60. Ejecución de test unitario

3.7 Publicación en tiendas de aplicaciones

URL App Store:
https://apps.apple.com/us/app/food-barcode-scanner/id1597807943

URL Google Play:
https://play.google.com/store/apps/details?id=app.scanfood.tfm.scanfood

Para publicar en las respectivas tiendas (Play Store, Apple Store) es necesario tener cuenta de desarrollo en estas. La cuenta de Play Store la tenía desde hace muchísimos años ya que en 2012 publique mi primera aplicación, pero la cuenta de Apple Store la tenía congelada desde hace varios años, así que para realizar la reactivación es necesario pagar la tasa anual de 99€/año.

Al intentar crear la app en la Apple Store menciono que el nombre *ScanFood* estaba ya siendo utilizado así que cambie el nombre a *Food Barcode Scanner*, también cambie el nombre de la aplicación en Android.

Después se configuro *Codemagic* para que la subida a Apple fuera automática y manual en Google Play (la primera subida no funciona hasta que no se ha subido al menos una iteración). Una vez con las aplicaciones en las tiendas pase a la fase de rellenar todos los campos requeridos como la privacidad de estas, titulo, descripción…

Para generar los pantallazos que han subido utilice el mockup HiFi y el aplicativo AppMockUp Studio (https://studio.app-mockup.com), que permite generar pantallazos en diferentes resoluciones con plantillas.

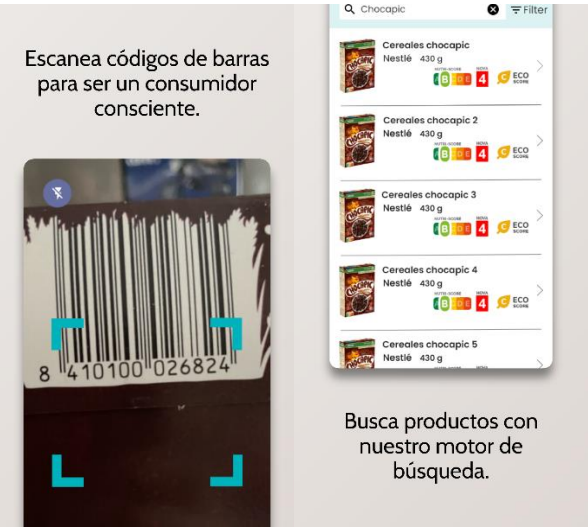


Ilustración 61. Screenshots publicados en las tiendas de aplicaciones

La aplicación se ha publicado en ambas tiendas sin problemas, con dos iteraciones para poder añadir los anuncios (validación de *admob*).

Solo se ha activado la aplicación en 13 países por limitaciones de la API: Canadá, Estados Unidos, Alemania, Bélgica, España, Francia, Irlanda, Italia, Países Bajos, Reino Unido, Suecia y Australia.

3.8 Revisión de la planificación

En cuanto a las fases de implementación se ha podido completar el desarrollo de la aplicación en ambas plataformas. Lo único pendiente es la activación de los anuncios de *admob* (pendiente de aprobación) pero estos están comentados en el código para cuando sea posible la activación.

Por otro lado, se ha tenido que prescindir en la fase de programación de la configuración de los alergenos, esa sección permitía desde configuración seleccionar unos alergenos para realizar un filtro automático sin tener que hacerlo manual en el listado. Se ha prescindido de esta por motivos de tiempo y que la definición visual no quedaba muy clara, ya que el usuario se podía confundir si no se recordaba de que ese filtro existía ya. Para esta funcionalidad se debería volver a trabajar en el apartado de diseño para intentar dejar más clara como es la interacción del usuario.

También se ha prescindido de la funcionalidad del mantenimiento del histórico, por motivos puramente de tiempo, a la hora de recortar he pensado que esta funcionalidad era prescindible ya que es invisible para el usuario y únicamente le sirve para no tener un histórico demasiado largo. Esto se puede implementar en la siguiente iteración del producto.

Al producto se le han destinado unas 135h en la fase de implementación cuando habían estipuladas unas 110h, demostrando que la planificación fue optimista en cuanto al cumplimiento de los objetivos. La mayor desviación ha sido producida por la conexión entre el código nativo y *Flutter*, que ha dedicado la gran parte del desarrollo del producto.

También se han trabajo en la implementación del *CI/CD* que no está planificada de forma inicial, pero el tiempo invertido en esto nos ha agilizado a la hora de probar la aplicación en dispositivos *iOS* al no tener que estar arrancando el *MacOS* en cada iteración del producto.

De cara a las fases finales del trabajo se dedicará más horas de las estimadas para intentar cerrar los flecos con una nueva publicación de la versión 1.1 con la publicidad activa (si es que llega la aprobación antes de la fecha) y se trabajara en la funcionalidad del mantenimiento del histórico, ya sea haciéndolo de forma automática como está planeado o añadiendo un botón de limpiar este.

4. Conclusiones

Al concluir el proyecto, podemos decir que se hizo una planificación bastante optimista del desarrollo, al planificar el tiempo fui optimista pero luego durante el desarrollo me encontré con problemas al desarrollar la parte nativa del aplicativo. Otro problema que hubo durante el desarrollo fue no definir del todo funcionalidades como la preferencia en alergenos, se hizo el diseño y el caso de uso en la sección de *Settings*, pero no se pensó en cómo se informaría al usuario en los listados ni cuando este entrara a un producto con el alergeno.

También en este último periodo detecte un bug en los dispositivos Xiaomi con la publicidad (https://github.com/googleads/googleads-mobile-flutter/issues/448), actualmente esta reportado como issue en github, y se ha vuelto a publicar la versión sin anuncios en la tienda de Google Play (a falta de aceptación).

Fue una buena decisión dar versiones a familiares y amigos durante el transcurso de la implementación cosa que ayudo a agilizar el descubrimiento de bugs y validar los diseños obtenidos.

En el apartado de diseño se han utilizado los conocimientos adquiridos en las asignaturas de *Diseño de productos interactivos multidispositivo* y *Diseño avanzado de productos interactivos multidispositivo*.

Sobre la idea de realizar el trabajo en *Flutter* se me ocurrió en el transcurso de la asignatura *Nuevas tendencias en el desarrollo de aplicaciones móviles,* en la que se trabajó con *Flutter* dando una introducción a la tecnología, aunque no se llegó a la fase de combinar con código nativo.

El proyecto ha sido una apuesta personal para demostrar que es posible un desarrollo hibrido de visuales combinado con partes nativas programando así los proyectos en *Swift*/*Kotlin* y *Dart*. Creo que esto ha quedado demostrado que es viable durante el transcurso del proyecto y que es una opción viable hoy en día.

Dicho la anterior, *Flutter* aún es un *framework* en desarrollo y como tal, hay falta de documentación y ejemplos en internet. Las librerías que podemos encontrar no son estables y todavía falta recorrido para que la tecnología se asiente.

Sobre esa apuesta personal, también he influenciado a la empresa donde trabajo, donde estamos apostando por realizar aplicaciones hibridas en *Flutter* con bastante código nativo (quiero agradecer a esta, la oportunidad de trabajar en el mundo móvil).

En cuanto a arquitectura, el uso de la inversión de dependencias (como se vio en *Desarrollo avanzado de aplicaciones para dispositivos Android*), la organización por capas y el uso del patrón *BLoC*, viene derivado de mi experiencia como programador previa (como developer *Backend-DevOps*) y es mi intento de acercar mi estilo de programación al mundo de *Flutter*.

La idea de incluir una herramienta *CI/CD* surge porque el desarrollo principal se produce en *Windows* y para poder continuar el desarrollo sin necesidad de la maquina *MacOS* se puede disponer de ello y así la plataforma permite generar bundles y subirlos a Apple Store sin tener que encender el otro ordenador. Como tengo experiencia en este campo (es el mismo proveedor que utilizo en la empresa), no ha quitado mucho tiempo de desarrollo.
En cuanto a las pruebas unitarias automáticas, no se han creado todos los necesarios por falta de tiempo, pero en siguientes versiones de la aplicación se volverá a iterar sobre esto para acabar de completar las pruebas automáticas.

Como resumen, estoy orgulloso del trabajo realizado y creo que ha quedado una buena aplicación, esta es usable por los usuarios y tiene buen rendimiento. Se ha conseguido cumplir con todos los requisitos funcionales y no funcionales, el diseño de la aplicación es funcional y claro y los usuarios con los que se ha probado la aplicación no han tenido problemas para utilizarla.

Por lo tanto, como líneas futuras de trabajo se priorizarían la eliminación del historial y establecer un máximo de ítems en este y la preferencia de alergenos prestablecidos (casos de uso restantes).

Una vez añadido esto se trabajaría en el apartado de ASO[13], mejorando el logo que ha quedado muy simple y localizando la ficha de la aplicación a más idiomas.

A partir de aquí se trabajaría en nuevas funcionalidades para la aplicación que quedaron fuera de la fase inicial:

- Añadir productos que la aplicación no localiza.
- Información sobre aditivos.
- Lista de ingredientes.
- Productos similares (sustitutos de la misma categoría).
- Acabar el testeo automático.

[13] ASO: App Store Optimization. Es el proceso de optimización de descripciones, *keywords*, logos para posicionar la aplicación en los primeros puestos del ranking de las tiendas de aplicaciones.

5. Glosario

- **Admob**: Empresa de publicidad móvil propiedad de Google, se utiliza para monetizar la aplicación mediante anuncios.

- **API**: *Application Programming Interface*. Interfaz de programación de aplicaciones, es el conjunto de definiciones y protocolos que se utiliza para integrar software en la aplicación. Permite la comunicación entre dos aplicaciones de software mediante el uso de reglas.

- **CI/CD**: *Continuous Integration/Continuous Delivery*. Método para distribuir aplicaciones de forma automatizada.

- **Código EAN**: *European Article Number.* Es uno de los sistemas de códigos de barras más adoptado mundialmente, se encuentra habitualmente en todos los productos que podemos encontrar en el supermercado.

- **Dart**: Lenguaje de programación creado por Google, es mayoritariamente utilizado junto al *framework* Flutter**.**

- **DCU**: *Diseño Centro en el Usuario*, filosofía de diseño en que asume que todo el proceso se centra en el usuario teniendo en cuenta todas sus necesidad y características.

- **Firebase**: Plataforma propiedad de Google que facilita servicios como estadísticas, recogida de errores, acceso a servidores gracias a varios *SDK* para móvil y web.

- **Flutter**: Framework que permite programar aplicaciones compiladas de forma nativa (binarios no interpretados) para múltiples plataformas**.**

- **Framework**: Paquete cerrado con un conjunto de herramientas, librerias y patrones que facilitan programar un proyecto.

- **Git**: Software de control de versiones, facilita el control de versiones y de trabajo realizado mediante un sistema de anotaciones y versionado del código**.**

- **IDE**: *Integrated Development Enviorment*, software que contiene una serie de utilidades para ayudar al desarrollador a programar su obra.

- **Intellij IDEA**: *IDE* creado por la empresa Intelij**.**

- **Keyword**: Palabras clave que definen nuestro producto y que se utilizaran para indexar el producto en los buscadores.

- **Kotlin**: Lenguaje de programación que corre sobre la máquina virtual de Java, este fue creado por la empresa Intellij.
- **OBbL**: *Licencia Abierta de Bases de Datos*, es un acuerdo de licencia que permite a los usuarios, compartir, modificar y usar libremente una base datos.

- **Open Font License**: Licencia de software libre para el uso en las tipografías**.**

- **ReactJS**: Librería de Javascript de código abierto diseñada para crear interfaces de usuario.

- **SDK**: *Software development kit*, paquete de herramientas y datos que facilita a los programadores a desarrollar programas con ese conjunto de utilidades.

- **Swift**: Lenguaje de programación creado por Apple, es utilizado para el desarrollo de iOS y MacOS**.**

- **UUID**: *Universally unique identifier*, código de identificación universal que se utiliza para denominar objetos de forma única e inequívoca.

- **XCode**: IDE creado por Apple para el desarrollo de software en iOS, MacOS, watchOS y tvOS.

6. Bibliografía

- Clarisó Viladrosa, Robert. Introducción al trabajo final. PID_00197259. FUOC. [fecha de consulta: septiembre de 2021]

- Saénz Higueras, Nita. Vidal Oltra, Rut. Redacción de textos científico-técnicos. P08/89018/00445. FUOC [fecha de consulta: septiembre de 2021].

- Guzzi, Vincenzo. Moore, Kevin D. Ngo, Vincent. Katz, Michael. Flutter Apprentice. Razerware LLC. 2021, disponible en : < https://www.raywenderlich.com/books/flutter-apprentice >

- Rešetár, Matt. Flutter Firebase & DDD Course 1. Domain-Driven Design Principles [en línea] [fecha de consulta: octubre de 2021] disponible en < https://resocoder.com/2020/03/09/flutter-firebase-ddd-course-1-domain-driven-design-principles/ >

- Sánchez Fernández, Jorge. El patrón BLoC junto a clean architecture en Flutter. [en línea] [fecha de consulta: octubre de 2021] disponible en < https://xurxodev.com/el-patron-bloc-junto-a-clean-architecture-en-flutter/ >

- Kayfitz, Bira. Getting Started with the BLoC Pattern. [en línea] [fecha de consulta: octubre de 2021] disponible en < https://www.raywenderlich.com/4074597-getting-started-with-the-bloc-pattern >

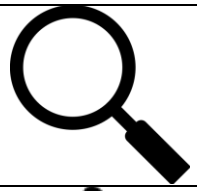
- Flutter team. Hosting native Android and iOS views in your Flutter app with Platform Views [en línea] [fecha de consulta: noviembre de 2021] disponible en < https://docs.flutter.dev/development/platform-integration/platform-views >

- Flutter team. Writing custom platform-specific code [en línea] [fecha de consulta: noviembre de 2021] disponible en < https://docs.flutter.dev/development/platform-integration/platform-channels >

- Ghorbaninia, Emad. Vision Framework Tutorial for iOS: Scanning Barcodes [en línea] [fecha de consulta: noviembre de 2021] disponible en < https://www.raywenderlich.com/12663654-vision-framework-tutorial-for-ios-scanning-barcodes >

7. Anexos

7.1 Licencia logotipos

Logotipo	URL	Licencia
	https://commons.wikimedia.org/wiki/File:Wikipedia_barcode_128.svg	Public Domain
	https://commons.wikimedia.org/wiki/File:Magnifying_glass_icon.svg	Public Domain
	https://commons.wikimedia.org/wiki/File:Food_Bank_icon.svg	CC0 License

El resto de los iconos pertenecen a la colección de material design y estas incluidos en el *Framework Flutter*, estos utilizan la licencia Apache.

7.2 Licencias ficheros *Lottie* (animaciones)

Logotipo	URL	Licencia
	https://lottiefiles.com/8998-scanning	Lottie Simple License
	https://lottiefiles.com/24703-food-animation	Lottie Simple License
	https://lottiefiles.com/36895-healthy-or-junk-food	Lottie Simple License

Todos los ficheros *Lottie* alojados en lottiefiles.com se distribuyen con la licencia Lottie Simple License, esta licencia permite la copia, reproducción, modificación con fines comerciales siempre que las modificaciones de estos se publiquen con la misma licencia.