ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Acondicionamiento del IUMATI Framework a las nuevas versiones y lenguaje de Android con nuevos controladores de pantalla

Titulación: Grado en Ingeniería en Tecnologías de la Telecomunicación
Mención: Telemática
Autor: Yguanira del Pino Vega Vega
Tutor: Dr. Luis Hernández Acosta

Fecha: Noviembre 2020

ULPGC
Universidad de
Las Palmas de
Gran Canaria

ESCUELA DE INGENIERÍA DE TELECOMUNICACIÓN Y ELECTRÓNICA

TRABAJO FIN DE GRADO

Acondicionamiento del IUMATI Framework a las nuevas versiones y lenguaje de Android con nuevos controladores de pantalla

HOJA DE EVALUACIÓN

Calificación: _____

Presidente

Fdo.:

Vocal **Secretario**

Fdo.: Fdo.:

Fecha: Noviembre 202

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mi familia por siempre prestarme su apoyo durante todos estos años, por estar en los momentos buenos y los malos y porque sé que siempre estarán ahí para mi igual que yo para ellos. A mi madre por siempre confiar en mí y apoyarme hasta el infinito, a mis hermanos por acompañarme en el camino y mis amigos por todas esas risas y momentos divertidos, gracias a todos por estar para mí siempre.

A mi tutor, por toda la ayuda y dedicación del proyecto, que, aunque este año ha sido duro, con su ayuda he avanzado y por fin cumplido he cumplido esta gran meta.

También agradaceder a todas aquellas personas que durante años han aportado vivencias y experiencias nuevas a mi vida.

Y en especial, agradecer ha esas personas que ya no están entre nosotros, que estén donde estén espero que estén orgullosas de mí y de lo que he conseguido en la vida, en especial tú, papá, hace once años me hiciste cumplir una promesa que finaliza con este proyecto, para cumplirla he pasado por muchas cosas buenas y malas, pero siempre inolvidables. Gracias.

RESUMEN

Desde que se desarrollaron las aplicaciones móviles a finales de los noventa, esta industria ha estado en constante crecimiento llegando a una actualidad en la que existen más de ocho lenguajes de programación para aplicaciones móviles y más de quince entornos, programas y *frameworks* en los cuales implementarlas.

La industria de programación de aplicaciones móviles vigente no se ha estancado creando entornos, programas y *frameworks* para programadores. También se han desarrollado múltiples aplicaciones y programas para personas sin conocimientos de programación, permitiéndoles con una interfaz intuitiva y sencilla el desarrollo de aplicaciones móviles complejas. En este último ámbito entra IUMATI Framework, un *framework* cuyo objetivo es ayudar a la creación de aplicaciones móviles multiplataforma, tanto para programadores como personas sin conocimientos de programación. Genera las aplicaciones a través de un catálogo de categorías y productos del que obtiene los datos para configurarlas, siendo las categorías las representantes de la navegación y los productos las pantallas, ellos son los que tienes los datos para configurarlas. Por tanto, al partir de un catálogo los usuarios son capaces de crear múltiples aplicaciones.

Partiendo del IUMATI Framework original, en este Trabajo de Fin de Grado se adaptará a las nuevas versiones de Android con un nuevo lenguaje y con nuevos controladores de pantalla y más configurables.

ABSTRACT

Since the development of mobile applications in the late nineties, this industry has been in constant growth, reaching a point where there are more than eight programming languages for mobile applications and more than fifteen environments, programs and frameworks in which to implement them.

The current mobile application programming industry has not stagnated by creating environments, programs and frameworks for programmers. Many applications and programs have also been developed for people without programming knowledge, allowing them with an intuitive and simple interface to develop complex mobile applications. In this last area, IUMATI Framework is included, a framework whose objective is to help the creation of multiplatform mobile applications, both for programmers and people without programming knowledge. It generates the applications through a catalogue of categories and products from which it obtains the data to configure them, being the categories the representatives of the navigation and the products the screens, they are the ones that have the data to configure them. Therefore, when starting from a catalogue, users are able to create multiple applications.

Starting from the original IUMATI Framework, in this End of Degree Project it will be adapted to the new versions of Android with a new language and with new and more configurable screen controllers.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

CAPÍTULO I. INTRODUCCIÓN

En este capítulo se introduce el contexto de la realización de este TFG, cómo se estructura, el estado de arte y se hace un breve resumen de lo que acontecerá en los siguientes capítulos.

Acondicionamiento del IUMATI Framework a las nuevas versiones y lenguaje de Android con nuevos controladores de pantalla**.**

1. CONTEXTO

En este año 2020 se cumplen 47 años desde la creación del primer teléfono (Motorola Dyna TAC 8000X de Martin Cooper, 1983)[1] y 26 años desde la creación del primer smartphone (IBM Simon Personal Communicator, 1992)[2]. Hace casi medio siglo que estos dispositivos comparten el día a día de las personas, pero no siempre han tenido las funcionalidades que conocemos hoy en día.

Las primeras aplicaciones que el mundo conoció fueron las aplicaciones de contactos, agenda y editores de tonos, entre otras, que aparecieron a finales de la década de los 90. A partir de ahí comenzó una nueva rama de la programación. La irrupción de los grandes gigantes que conocemos hoy en día como Apple y Android permitieron a otros desarrolladores y compañías externas estar en este mercado para poder llegar a todos los usuarios que quisieran disfrutar de la experiencia de utilizar una aplicación.[3]

Actualmente el desarrollo de aplicaciones móviles es una industria completa que mueve millones a lo largo del mundo. Según el último informe de 2019 de Google Play mostrado por la empresa Google, hay 2.5 billones de dispositivos Android activos y más de 115 billones de instalaciones de aplicaciones del Google Play[4]. Ahora, la plataforma consta con 3.040.582 aplicaciones según muestra AppBrain[5] en su página web, número que seguirá incrementándose.

En términos generales, en las aplicaciones móviles se han establecido siete grandes categorías que se muestran en la Tabla 1 y cuatro tipos de aplicaciones, mostradas en la Figura 1.

Juegos
Comerciales
Educativas
De estilo
De Entretenimiento
Utilidad
Viaje

Tabla 1. Tipos de aplicaciones móviles

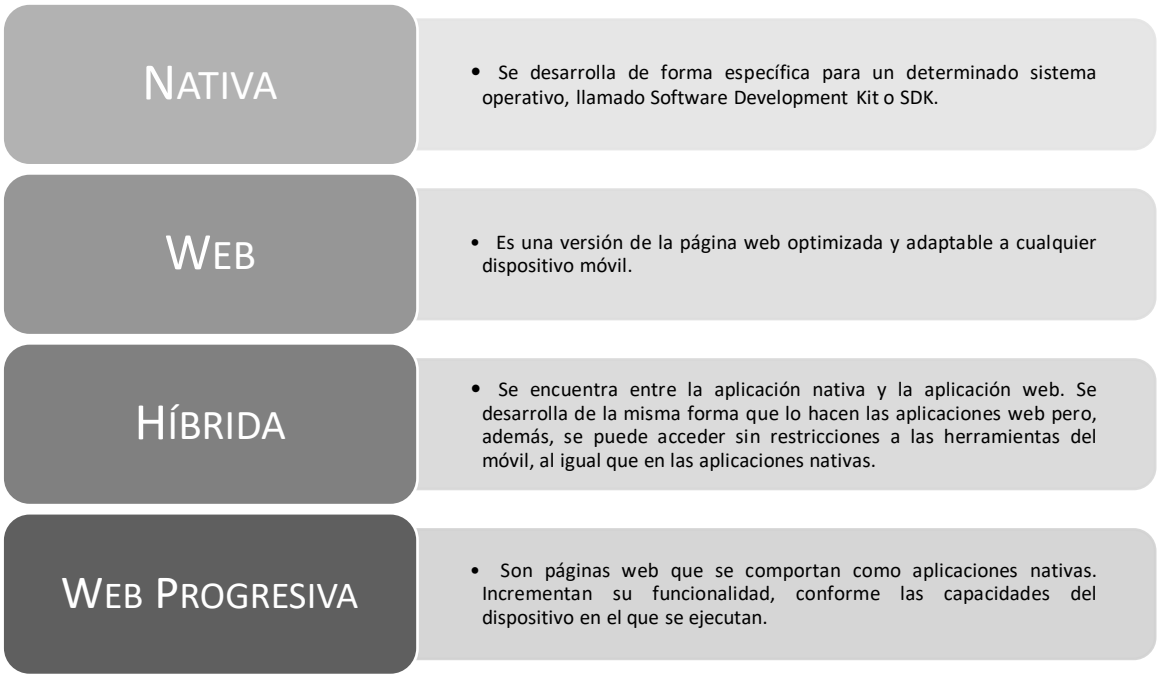

Figura 1. Tipos de aplicaciones móviles

Desde la etapa inicial de las aplicaciones móviles en las que sólo se programaban en dos lenguajes: Java usado en el sistema operativo Android y Objective-C empleado en iOs de Apple, los lenguajes han evolucionado, mezclado e incluso se han creado nuevos como se puede observar en la Tabla 2.

Lenguajes de programación móviles			
Android	iOS	Plataformas cruzadas	Otros
Java	Objective - C	JavaScript	C#, C y C++
Kotlin	Swift	TypeScript	Python
			Ruby

Tabla 2. Lenguajes de programación móvil

2. ANTECEDENTES

Como se evidenció en el apartado de introducción, después de la creación de las primeras aplicaciones móviles se generó una gran demanda de aplicaciones con nuevas funcionalidades, a la que los grandes gigantes respondieron liberando sus propios programas de desarrollo de aplicaciones para sus sistemas operativos al resto de programadores del mundo. Posteriormente,

con los múltiples lenguajes de programación se conformaron entornos de trabajo y *frameworks* para programadores.

A continuación, en la Figura 2*,* se muestran algunos de los diferentes programas, entornos de desarrollo y *frameworks* empleados en la actualidad para crear aplicaciones móviles.

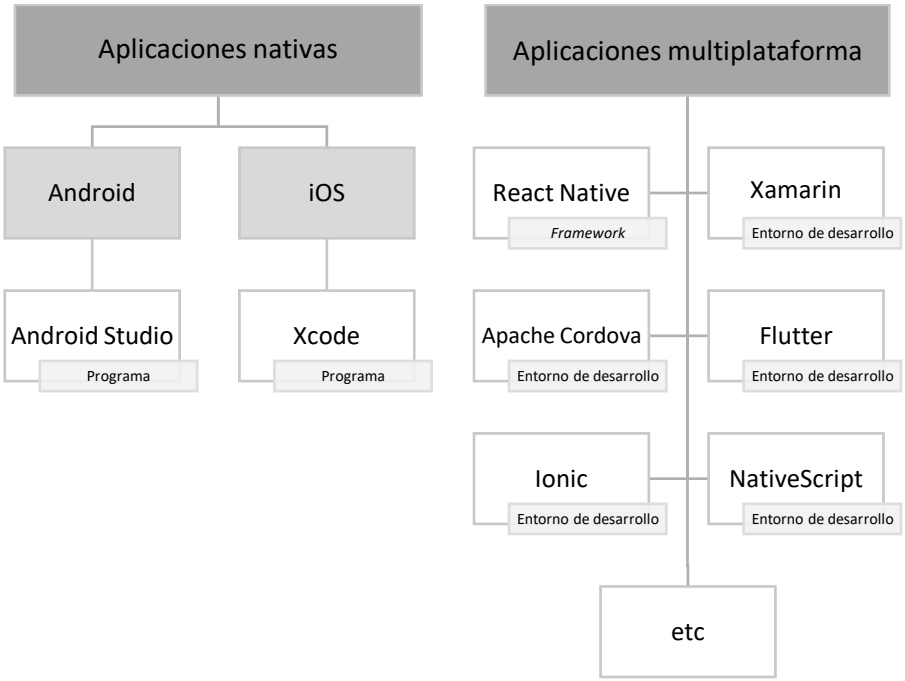

Figura 2. Programas, entornos y marcos de desarrollo de aplicaciones móviles

Esta industria no se ha estancado creando entornos, programas y *frameworks* para programadores. También se han desarrollado múltiples aplicaciones y programas para personas sin conocimientos de programación, permitiéndoles con una interfaz intuitiva y sencilla el desarrollo de aplicaciones móviles complejas. A este último ámbito es donde se orienta el proyecto redactado en este documento, el cual es un *framework* para crear aplicaciones orientado a personas que no saben programar.

3. OBJETIVOS

El objetivo principal de este Trabajo de Fin de Grado es la adaptación del IUMATI Framework, del cual se hablará posteriormente, a las versiones más actuales de Android, reescribiendo para ello desde cero el código de este usando el lenguaje de programación Kotlin en vez de Java además del reimplementado de las pantallas (aspecto y funcionalidad) ya existentes e incorporando nuevas.

El objetivo principal se desglosa en los siguientes objetivos específicos para una mejor comprensión de la estructura de este documento:

1. Análisis de la funcionalidad del IUMATI Framework implementado en Java, el cual configura una aplicación en función de un catálogo de categorías y productos.
2. Diseño de las pantallas de las que dispondrá el nuevo *framework* implementado en Kotlin, así como el patrón de diseño de clases que compondrá la implementación de cada pantalla.
3. Desarrollo del código necesario para implementación de las pantallas decididas en la etapa de diseño, lo cual indica desarrollar sus funcionalidades, gestión de datos y configuración de aspecto.
4. Análisis de un caso de uso del *framework* con un catálogo de categorías y productos real.

4. ESTADO DEL ARTE

Actualmente los *frameworks* más utilizados entre programadores son:

- Ionic: Es un kit de herramientas de interfaz de usuario móvil de código abierto para crear experiencias de aplicaciones web y nativas multiplataforma de alta calidad. Emplea una única base de código que se ejecuta en todas partes.[6]
- Angular: Es un marco de diseño de aplicaciones y una plataforma de desarrollo para crear aplicaciones de una sola página eficientes y sofisticadas.[7]
- PhoneGAP: Del equipo detrás de Apache Cordova, este marco es una distribución de código abierto de Cordova, que brinda la ventaja de la tecnología creada por un equipo diverso de profesionales junto con una comunidad de desarrolladores sólida, además de acceso al conjunto de herramientas PhoneGap, para que pueda acceder a móvil más rápido.[8]
- jQuery mobile: Es un sistema de interfaz de usuario basado en HTML5, diseñado para crear sitios web y aplicaciones receptivos que sean accesibles en todos los teléfonos inteligentes, tabletas y dispositivos de escritorio.[9]
- React Native: Este marco de trabajo combina las mejores partes del desarrollo nativo con React, la mejor biblioteca de JavaScript de su clase para crear interfaces de usuario. Prácticamente el 100% del código escrito funcionará como una app nativa en cualquier sistema seleccionado (Android – iOS), aunque el desarrollo se realizará solo una vez.[10]
- Framework 7: orientado a iOS, es un marco de código abierto y gratuito para desarrollar aplicaciones móviles, de escritorio o web con apariencia nativa. También es una

herramienta de creación de prototipos indispensable para mostrar el prototipo de la aplicación en funcionamiento lo antes posible en caso de que lo necesite.[11]

Sin embargo, como se ha comentado anteriormente no necesitas ser un programador para crear una aplicación. Hoy en día, existen múltiples aplicaciones que te permiten realizar este trabajo en unos sencillos pasos. Entre ellas encontramos:

- GoodBarber: Creador de aplicaciones nativas y web progresivas para comercio electrónico y gestión de contenido.[12]
- Swiftic: Ofrece siete plantillas diferentes que puedes combinar con siete estilos de navegación. También puedes combinar colores, imágenes de fondo e iconos con tus propias imágenes artísticas.[13]
- Mobincube: Todos sus planes te permiten crear aplicaciones nativas. Permiten crear tipos de página muy complejos: por ejemplo, se puede usar una base de datos basada en SQLite para sacar datos de un servidor externo o añadir módulos HTML a la aplicación.[14]
- Shoutem: Rendimiento nativo de iOS y Android, impulsado por React Native. Un número creciente de temas y diseños además de la posibilidad de creación fácil de nuevos temas y diseños con el kit de herramientas de interfaz de usuario de Shoutem.[15]
- Mobileroadie: Une el desarrollo de aplicaciones personalizadas y el marketing móvil. Tiene prestaciones muy interesantes para las comunidades online como chats y páginas de fans. También presenta algunas características muy avanzadas como la determinación geográfica de los objetivos para tus contenidos.[16]
- AppYourself: Creación de aplicaciones para iOS, Android, Windows Phone y también aplicaciones PWA. Incluye prestaciones para comercio electrónico, reservas en restaurantes vía Open Table y Resmio, galerías de fotos y por supuesto los canales habituales de noticias feed.[17]

5. ESTRUCTURA DEL DOCUMENTO

Los apartados anteriores de este capítulo introducen el contexto de la realización de este TFG, los objetivos a realizar y el estado de arte, los siguientes capítulos están orientados al desarrollo del proyecto.

En primer lugar, un capítulo de análisis detallará la investigación y descomposición del actual IUMATI Framework. A continuación, el capítulo de tecnologías empleadas describirá el Hardware y Software aplicados durante el desarrollo del TFG documentado en este documento. Se continuará

con un capítulo de diseño que abarcará los bocetos con los componentes iniciales de las pantallas a incorporar en el nuevo IUMATI Framework. Posteriormente, en el apartado de implementación se especificará minuciosamente el código desarrollado, es decir, se detallará cada una de las clases e interfaces implicadas en el nuevo IUMATI Framework.

Una vez completada la descripción del proyecto, su funcionamiento será detallado en el capítulo de guía de usuario para luego analizar su ejecución y resultado en el capítulo de caso de uso. Seguidamente, en el capítulo de conclusiones se expondrán, tras un breve resumen, los datos obtenidos y las resoluciones logradas a partir de estos, así como los objetivos alcanzados y las líneas futuras del proyecto. Finalmente, se finalizará el documento con un capítulo de presupuesto en el que se detallará la estimación monetaria para la realización de este TFG.

CAPÍTULO II. ANÁLISIS

En este capítulo se llevará a cabo una investigación y descomposición del actual IUMATI Framework.

1. IUMATI FRAMEWORK

Figura 3. Logo IUMATI Framework

En el Instituto Universitario de Microelectrónica Aplicada (IUMA) de la Universidad de Las Palmas de Gran Canaria (ULPGC), los doctores José María Quinteiro González y Luis Miguel Hernández Acosta desarrollaron en el año 2012 un *framework* denominado IUMATI Framework, cuyo logo se puede ver en la Figura 3, que permite la generación inmediata de aplicaciones nativas iOS y Android.

El *framework* fue implementado en Android Studio empleando el lenguaje de programación Java, y en Xcode empleando Objetive - C y Swift.

El objetivo del IUMATI Framework es ayudar a la creación de aplicaciones móviles multiplataforma, tanto para programadores como personas sin conocimientos de programación debido a que las aplicaciones se crean en función de los datos de contenido y configuración introducidos mediante interfaz web.

2. SPIDERCATALOG

SpiderCatalog es el plugin de WordPress[18] que emplea IUMATI Framework en la interfaz web y consiste en un catálogo personalizable diseñado para ayudar a visualizar productos en un formato elegante de catálogo. Es una herramienta útil para organizar los productos representados en su sitio web en forma de catálogo. Cada producto en el catálogo se asigna con una categoría y subcategoría correspondiente, lo que hace que sea más fácil buscar e identificar los productos necesarios dentro del catálogo.[19]

3. FUNCIONAMIENTO

El funcionamiento del IUMATI Framework se explica fácilmente mediante la Figura 4. En ella se aprecia que en la interfaz web el usuario configura de forma fácil, sencilla e intuitiva la aplicación y luego la interfaz se encarga de crear una estructura de catálogo de categorías y productos con la ayuda de una adaptación del *plugin* de WorldPress *SpiderCatalog*, para el cual el sistema está optimizado. Posteriormente, el *framework* solicita el catálogo para crear la aplicación y la interfaz

se lo envía en formato JSON. El *framework* lee el catálogo generado desde la nube y lo mapea a tablas de la base de datos para luego obtener dichos datos y configurar la aplicación resultante dependiendo de estos.

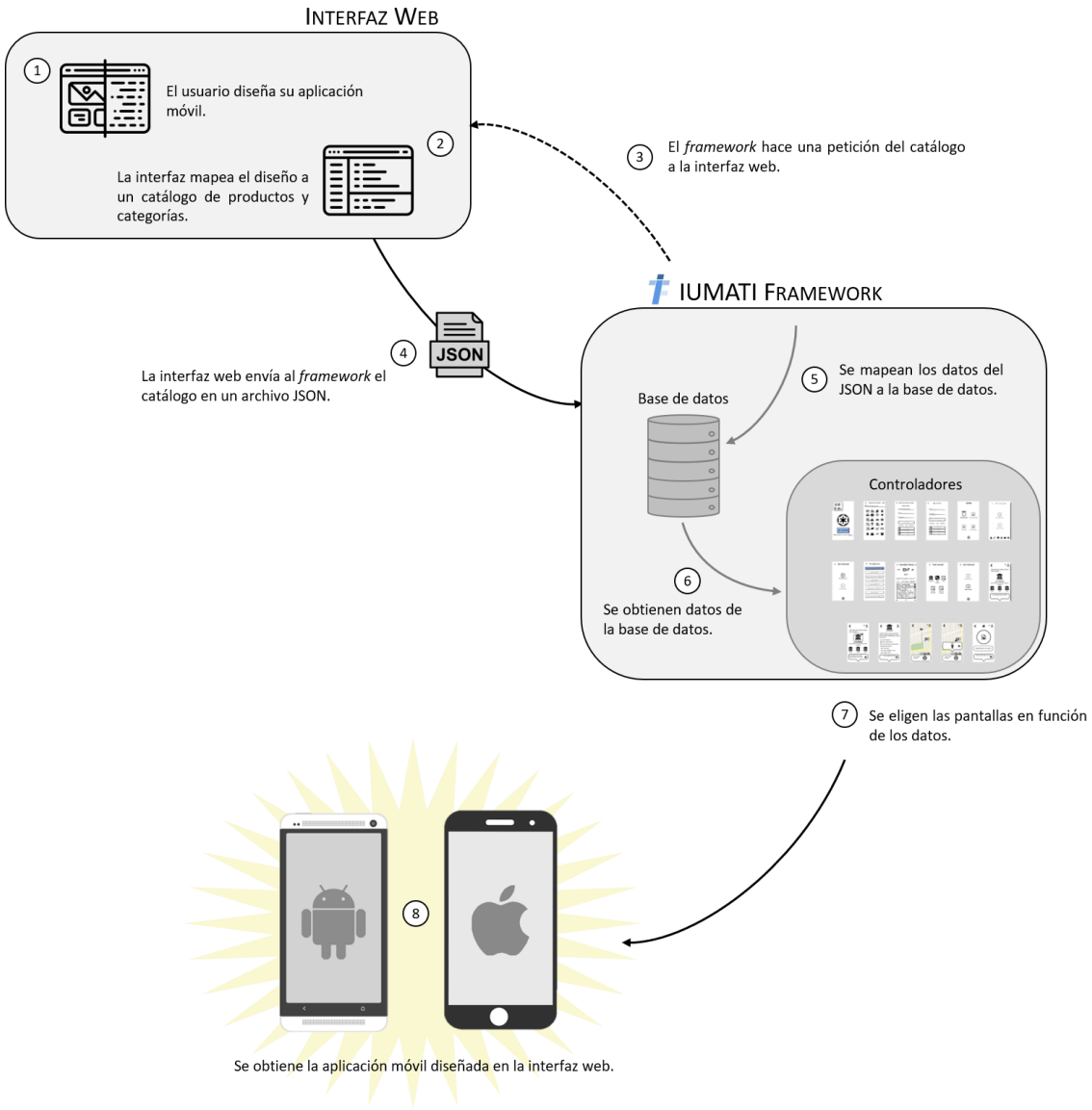

Figura 4. Esquemático funcionamiento IUMATI Framework

4. ESTRUCTURA DE DATOS

Como se ha comentado, la estructura de datos del *framework* se basa en un catálogo de categorías y productos. En él, las categorías que componen la aplicación están constituidas por diferentes niveles jerárquicos que ayudan a la organización. Asimismo, cada producto pertenece a una o varias categorías en las que se divide el catálogo.

A continuación, se muestra en la Figura 5 un diagrama explicativo de los parámetros que componen tanto las categorías como los productos y el significado de estos.

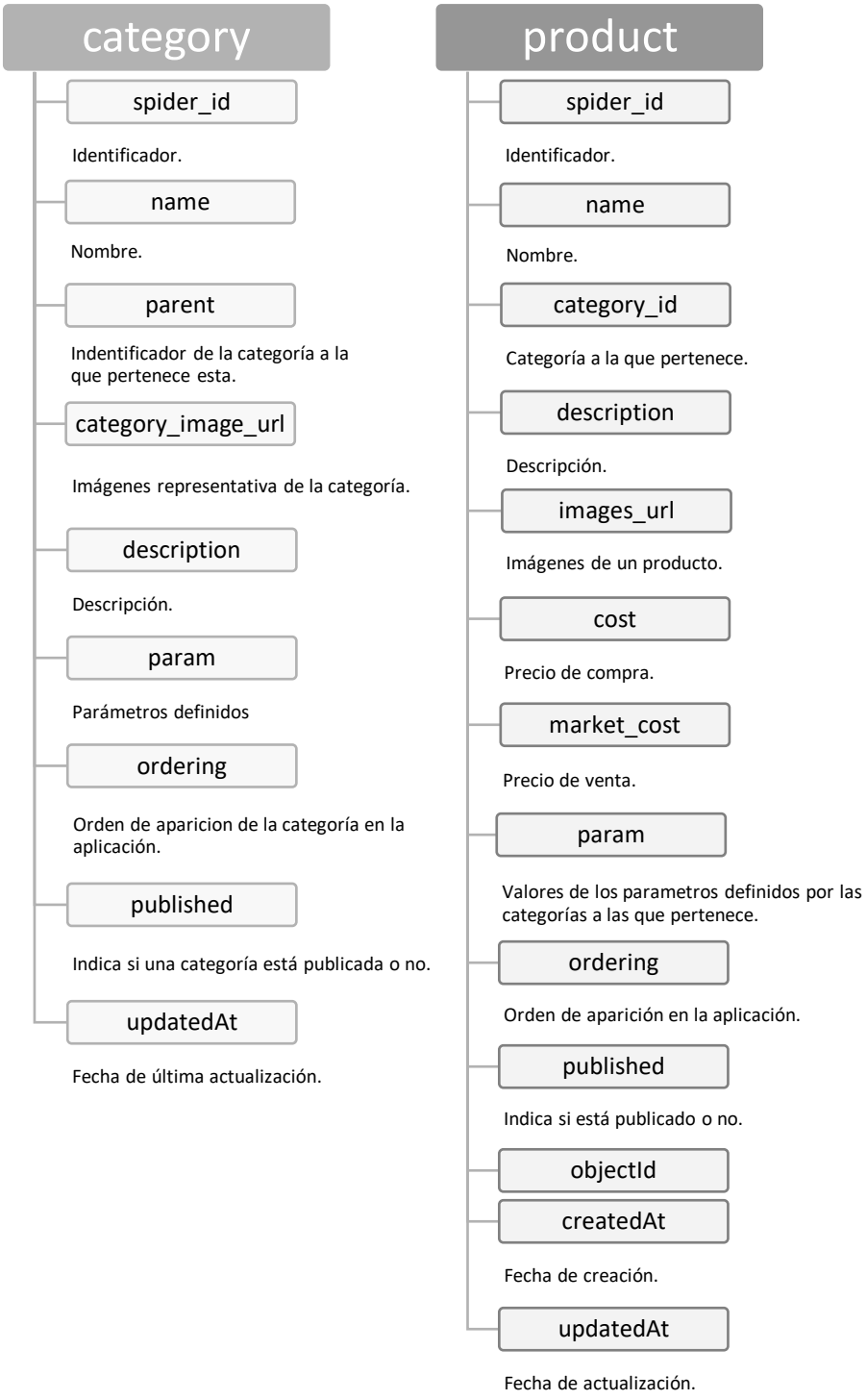

Figura 5. Estructura de categoría y producto detallada

Esta estructuración en categorías y productos permite crear una organización jerárquica con un alto porcentaje de crecimiento debido a que las categorías pueden tener tanto subcategorías como

productos asociados, un claro ejemplo de esta afirmación puede verse en la Figura 6, en la cual se muestra la jerarquía de categorías (recuadros de color) y productos (sin recuadro) de la aplicación de ejemplo *Katalog*.

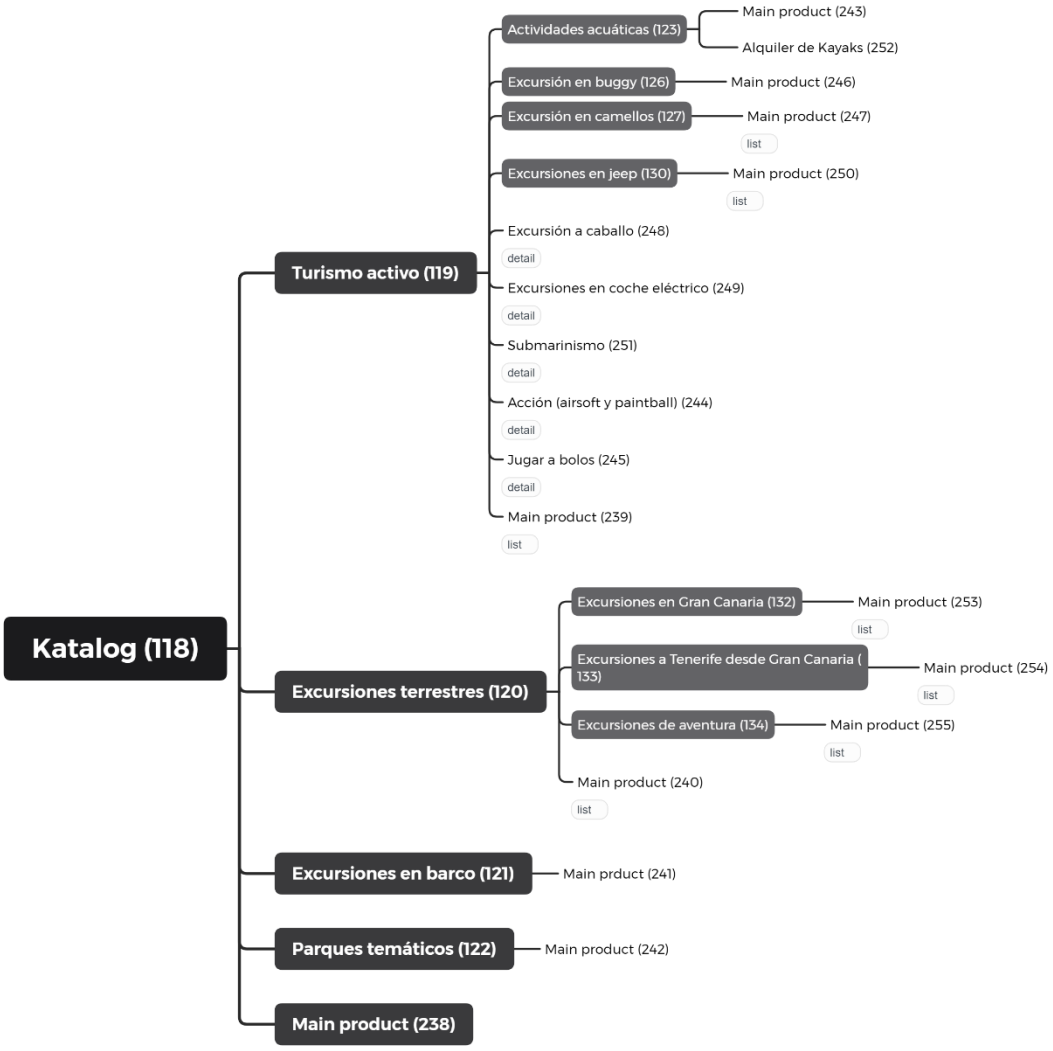

Figura 6. Esquemático con la jerarquía de la aplicación Katalog

En el ejemplo queda evidenciado la escalabilidad de la aplicación con la incorporación de nuevas categorías y/o productos sin afectar de forma significativa a la jerarquía, ya que la única conexión entre los elementos es el identificador asociado a la categoría superior, el cual puede ser modificado. Asimismo, tanto categorías como productos pueden ser eliminados sin mayor dificultad.

CAPÍTULO III. TECNOLOGÍAS EMPLEADAS

En este capítulo se describe el *Hardware* y *Software* aplicados durante el desarrollo del trabajo.

El IUMATI Framework está desarrollado, como se ha mencionado en capítulos anteriores para los sistemas operativos Android e iOs, sin embargo, hay que resaltar que este proyecto abarca la adaptación del IUMATI Framework para la creación de aplicaciones nativas Android. Por tanto, las tecnologías de implementación y testeo son referentes a dicho sistema operativo.

1. SOFTWARE

Respecto a las tecnologías software utilizadas tenemos una aplicación y un programa:

- Figma

  Es una herramienta de diseño de interfaz colaborativa. Figma, cuyo icono podemos ver en la Figura 7, es una aplicación basada en navegador para diseñar interfaces de usuario e interfaces de experiencia que proporciona todas las herramientas necesarias para la fase de diseño de un proyecto.

Figura 7. Logo de Figma

- Android Studio (v 4.0.1)

Figura 8. Logo de Android Studio

Android Studio es el entorno de desarrollo integrado (IDE) oficial para el desarrollo de aplicaciones para Android, basado en IntelliJ IDEA y cuyo icono se ve en la Figura 8. Además del potente editor de códigos y las herramientas para desarrolladores de IntelliJ, Android Studio ofrece incluso más funciones que aumentan tu productividad. [20]

2. HARDWARE

El hardware empleado ha sido teléfonos móviles cuya única función ha sido la de testeo durante la implementación y la fase del caso de uso. Las características relevantes para este TFG de los dispositivos son la comentadas a continuación:

- Realme X2

  Pantalla super AMOLED 6.4”, resolución de 2340 x 1080 y una densidad de 403 ppi. Sistema operativo Android 9 Pie (API 28).

- Samsung Galaxy J7 (2016)

  Pantalla super AMOLED HD 5.5”, resolución de 1920 x 1080 y una densidad de 294 ppi. Sistema operativo Android 8 Oreo (API 26).

CAPÍTULO IV. DISEÑO

En este capítulo se abarcan los bocetos con los componentes iniciales de las pantallas a incorporar en el nuevo IUMATI Framework.

1. LENGUAJE DE PROGRAMACIÓN

El lenguaje escogido para esta adaptación del IUMATI Framework como se ha mencionado en otros apartados es Kotlin.

Kotlin se originó en JetBrains, la compañía detrás de IntelliJ IDEA, en 2010, y ha sido de código abierto desde 2012. Es un lenguaje de programación "pragmático" de tipo estático, gratuito, de código abierto y de propósito general, inicialmente diseñado para JVM (Java Virtual Machine) y Android, que combina características de programación funcional y orientada a objetos. [21]

2. MOCKUPS DE LOS CONTROLADORES

El diseño de los controladores incorporados en el marco de trabajo se llevó a cabo empleando la herramienta software Figma, la cual ya ha sido mencionada en el documento y es una aplicación para diseñar interfaces de usuario. Asimismo, en función de los diseños se pudo estipular los componentes Android que incorporarían cada pantalla.

2.1. SPLASH

Este es el controlador inicial de la aplicación, la idea es tener una pantalla inicial en la que se configuren los datos de la aplicación y por tanto no es modificable. En consecuencia, su diseño es claro y sencillo, una pantalla como la Figura 9 en la que se muestra el nombre de la aplicación con su icono y una barra de progreso para indicar al usuario que se está configurando la aplicación. Es una estructura con la que la mayoría de los usuarios estarían de acuerdo, ya que su estilo no tiene un gran impacto con el diseño escogido para el resto de los controladores de la aplicación.

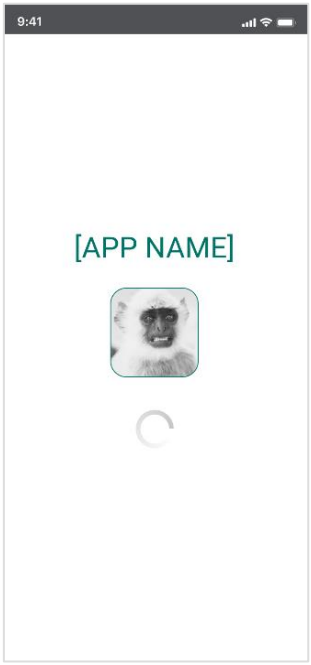

Figura 9. Diseño pantalla Splash

Teniendo en cuenta la estructura escogida, en términos de diseño de Android, la pantalla está compuesta por:

- *TextView*: contiene el nombre de la aplicación.
- *ImageView*: icono de la aplicación.
- *ProgressBar*: barra de proceso que indica que los datos de la aplicación se están cargando.

2.2. HOME

Este es el controlador que se encarga de la navegación principal de la aplicación, es decir, configura como se va a mover el usuario entre las diferentes pantallas. El diseño de este controlador consiste en dar la posibilidad al usuario de elegir entre múltiples tipos de navegación. Por tanto, para que esta funcionalidad sea posible la pantalla debe incorporar varios tipos de navegación y los escogidos son los mostrados en las Figuras 10 – 14.

Al diseñar este controlador con cinco tipos de navegación se le da la posibilidad al usuario de escoger entre cualquiera de ellos. Asimismo, también permite realizar combinaciones entre estos, lo que da una mayor versatilidad al usuario a la hora de modificar este controlador.

En términos de diseño de Android, esta pantalla está compuesta por:

- *Toolbar*: barra de aplicación superior.
- *TabLayout*: barra de pestañas.
- *NavigationView*: menú de navegación lateral desplegable.
- *BottomAppBar*: barra de aplicación inferior.
- BottomNavigation: barra de navegación inferior.
- ProgressBar: barra de proceso que indica que el contenido que no sea parte de la navegación principal se está configurando.

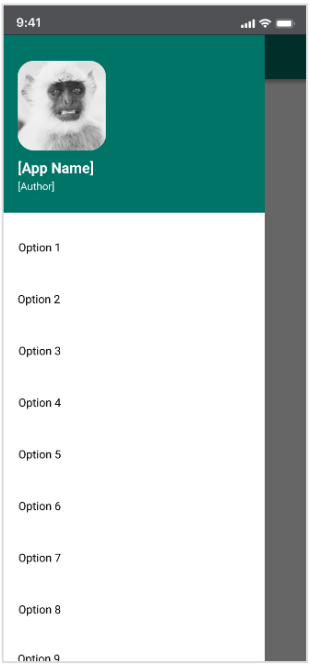

Figura 10. Navegación por menú lateral

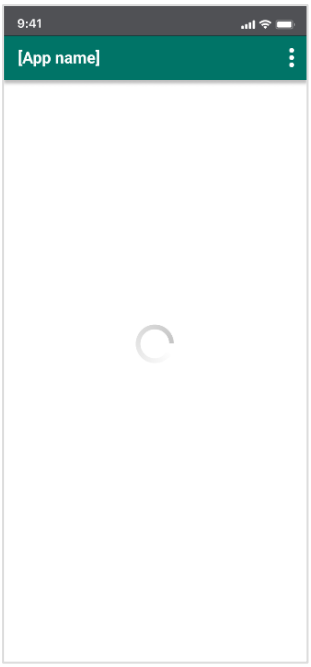

Figura 11. Navegación por barra de navegación superior

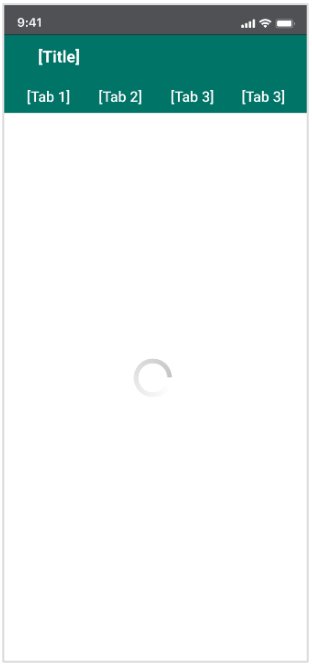

Figura 13. Navegación por pestañas

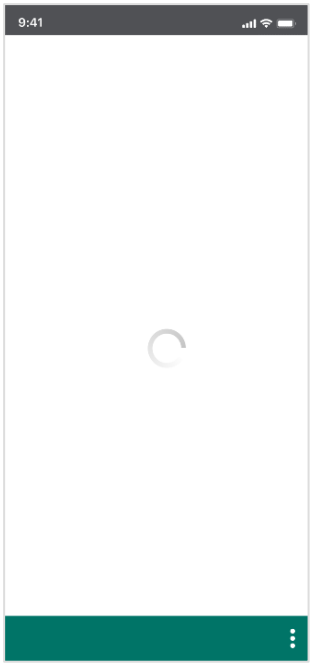

Figura 12. Navegación por barra de acciones inferior

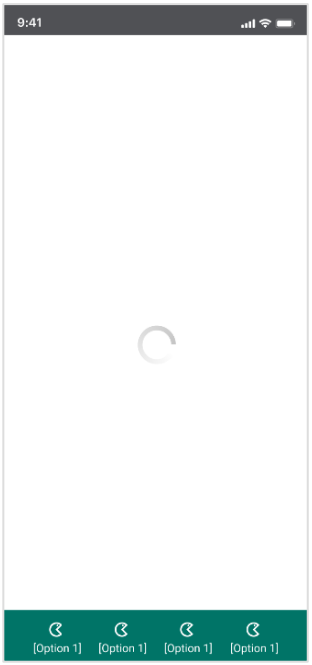

Figura 14. Navegación por menú inferior

2.3. GALLERY

El objetivo principal de este controlador es mostrar una galería de tarjetas modificables por el usuario. En las especificaciones se da la posibilidad al usuario de elegir cuantas tarjetas quiere mostrar en una fila, que el fondo de la tarjeta sea una imagen, que tenga título, que se pueda indicar si es destacada o no e incluso compartir la tarjeta. Todos estos elementos dan la posibilidad de crear combinaciones como las mostradas en la Figura 15.

Teniendo en cuenta las múltiples combinaciones de esta pantalla, en términos de diseño de Android, está compuesta por:

- *Toolbar*: barra de aplicación superior.
- *RecycleView*: contenedor con las vistas de cada uno de los elementos del controlador. Cada vista contiene:
    - Dos *ImageView*: una corresponde a la imagen del elemento y la otra al icono de destacados.
    - *TextView*: indica el nombre del elemento.
    - *Button*: botón para compartir.

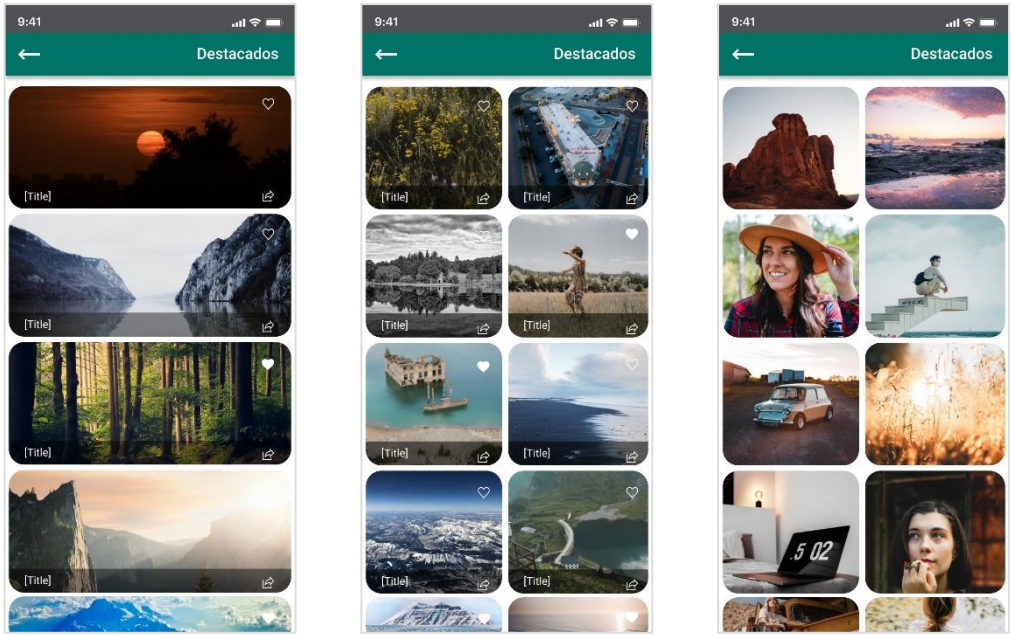

Figura 15. Diseños de pantalla Gallery

2.4. LIST

La idea de este controlador de pantalla es mostrar una lista de elementos con un diseño modificable por el usuario. Para ello, aparte de dar la posibilidad de que sea expandible, se permite mostrar un título, una imagen, un subtítulo, una descripción, indicar si es destacado o no y, además, posibilitar que se pueda compartir dicho elemento. Algunas de las combinaciones que el usuario puede hacer con los elementos anteriores son las enseñadas en la Figura 16.

Con el fin de diseñar esta pantalla en términos de Android, estaría compuesta por:

- *Toolbar*: barra de aplicación superior.
- *RecycleView*: contenedor con las vistas de cada uno de los elementos del controlador. Cuenta con dos tipos de vistas: una expansible y otra expandida.

  Vista expansible

  · Dos *ImageView*: la imagen del elemento y la otra al icono de destacados.

  · Tres *TextView*: el nombre del elemento, el subtítulo y una breve descripción.

  · *Button*: botón para compartir.

  Vista expandida

  · Dos *ImageView*: una corresponde a la imagen del elemento y la otra al icono de destacados.

- Dos TextView: el primero es un subtítulo y el tercero una breve descripción.
- *Button*: botón para compartir.

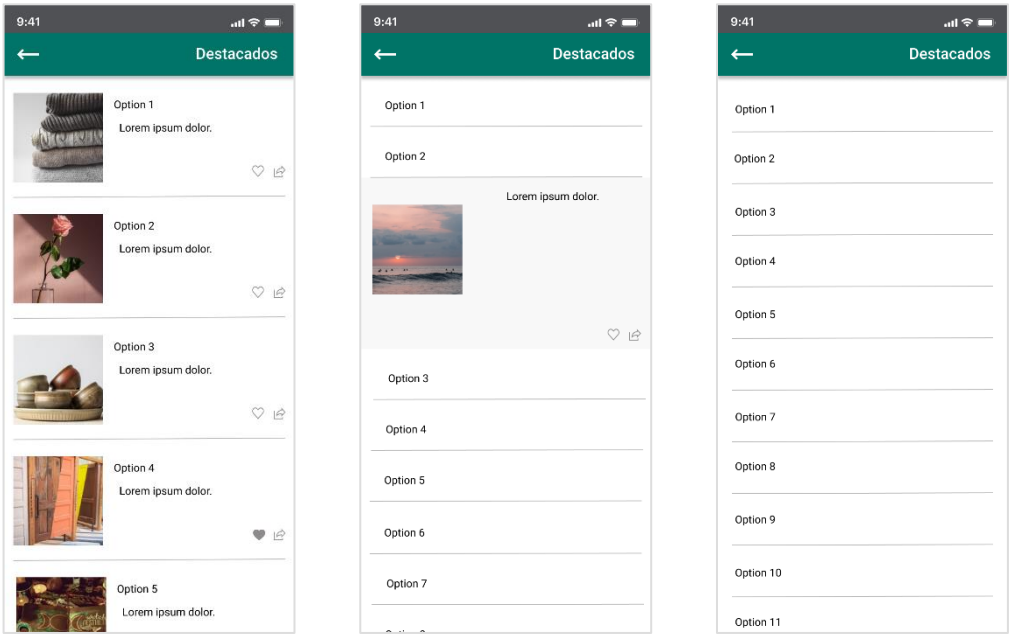

Figura 16. Diseños de la pantalla List

2.5. MAP

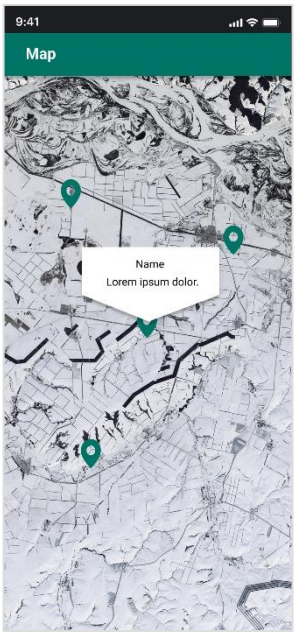

Figura 17. Diseño de la pantalla Map

El objetivo de este controlador es configurar un mapa que se modifica dependiendo de las especificaciones elegidas por el usuario. Es un mapa en el que se muestran una serie de localizaciones y el usuario pueda clicar sobre ellas para obtener más información o crear una ruta hasta ellas.

Teniendo en cuenta la funcionalidad de esta pantalla, en términos de diseño de Android, compuesta por:

- *Markers*: localizaciones en el mapa.
- Dos *TextView* por cada localización: uno contiene el título del punto en el mapa y el otro un pequeño subtítulo.

2.6. DETAIL

La idea de este controlador de pantalla es mostrar el detalle de un elemento que previamente se ha mostrado en otro controlador. Es una pantalla en la que aparece un carrusel de imágenes, un nombre, un subtítulo y una descripción completamente configurables en la que el usuario puede escoger los elementos a mostrar y su colocación dentro de la pantalla, un claro ejemplo de esta libertad de configuración se muestra en la Figura 18.

Con el fin de diseñar esta pantalla en términos de Android, estaría compuesta por:

- *Caroussel*: carrusel con imágenes.
- Tres *TextView*: indican el nombre del elemento, un subtítulo y la descripción del elemento.

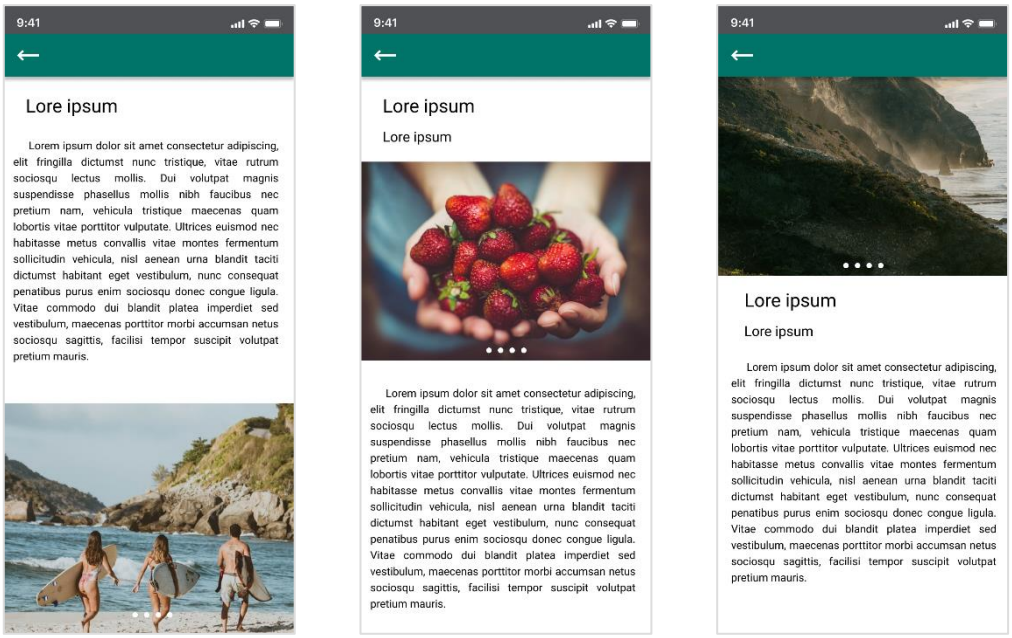

Figura 18. Diseños de la pantalla Detail

3. ESTRUCTURA DE LOS CONTROLADORES

La estructura escogida para la implementación de los controladores es una variante del patrón de diseño MVP (Modelo-Vista-Presentador) que incorpora las clases *Screen*, *Router* y *ViewModel*.

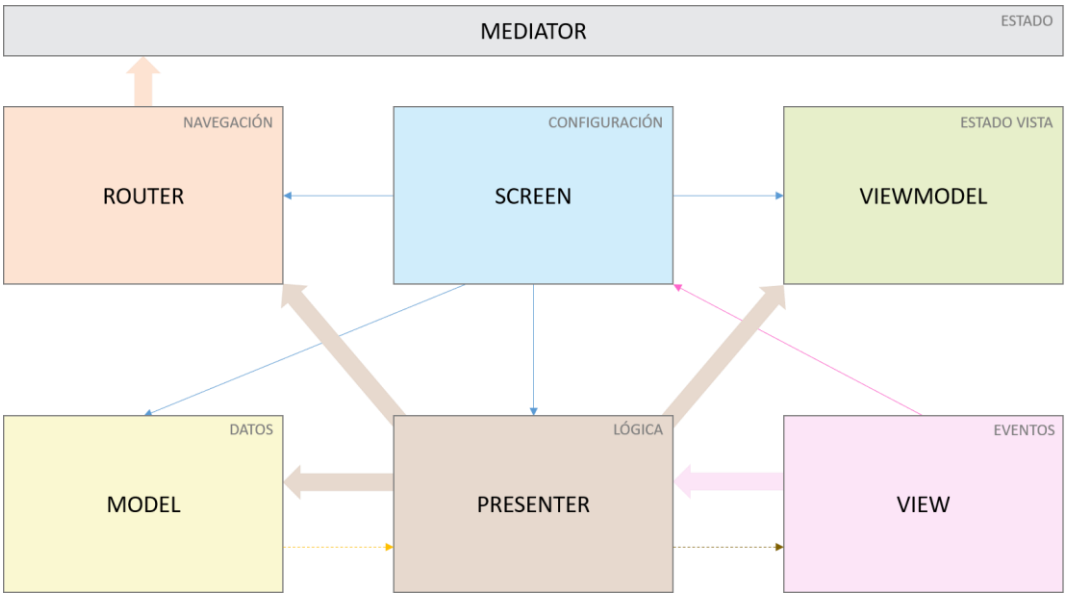

Figura 19. Patrón de diseño de los controladores de pantalla

En este patrón tenemos que cada controlador de pantalla mínimo está compuesto por seis clases y una interfaz *Contract* que define los métodos de la vista, el presentador, el modelo y el router, como se puede ver en la Figura 19.

- *View*: se encarga de la detección y notificación de eventos al presentador además de la actualización y gestión de la interfaz de usuario.
- *Presenter*: gestiona eventos procedentes de la vista, solicita navegación al router y datos al modelo. Cuando obtiene datos del modelo actualiza su estado. Es el puente entre la vista y el modelo.
- *Model*: se hace cargo del almacenamiento, mantenimiento de datos y gestión del repositorio (fuente de los datos).
- *ViewModel*: conserva el estado visual de la pantalla, es decir, almacena los datos de configuración de la pantalla mostrada.
- *Router*: clase encargada de la navegación entre pantallas. Obtiene datos almacenados en el mediador por otro controlador de pantalla e igualmente introduce datos en el mediador para otro controlador.
- *Screen*: configura e inyecta las dependencias entre el resto de las clases para una correcta gestión de datos y eventos.

Asimismo, hay una clase común a todos los controladores mencionada anteriormente y que se puede ver en la Figura 19 denominada *Mediator* que conserva el estado general de la aplicación, es decir, los datos de esta clase se mantienen hasta la finalización de la ejecución de la aplicación

independientemente del cambio entre pantallas. Por tanto, es la clase empleada para la comunicación entre los diferentes controladores de pantalla de la aplicación.

CAPÍTULO V. IMPLEMENTACIÓN

En este capítulo se detalla de forma exhaustiva cada una de las clases e interfaces creadas para la ejecución de IUMATI Framework.

1. INCORPORACIONES Y MODIFICACIONES

En la fase diseño de las diferentes pantallas se realizó el boceto de seis pantallas, durante el proceso de implementación se incorporaron dos pantallas más y se incorporó una nueva funcionalidad en la pantalla *Gallery*.

1.1. IMAGE VIEWER

La idea de este controlador de pantalla es mostrar una imagen. Para ello, aparte de dar la posibilidad al usuario de hacer *zoom*, también se posibilita que se pueda compartir. Esta pantalla no sería modificable por parte del usuario ya que tiene una estructura concreta que es la mostrada en la Figura 20.

Con el fin de diseñar esta pantalla en términos de Android, estaría compuesta por:

- *Toolbar*: barra de aplicación superior con la opción de compartir.
- *ImageView*: una corresponde a la imagen del elemento y la otra al icono de destacados.

Figura 20. Diseño de la pantalla ImageViewer

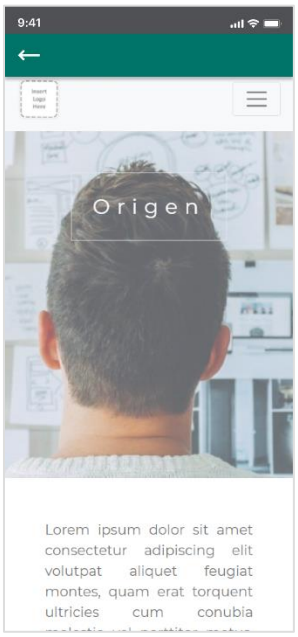

Figura 21. Diseño pantalla HTML

1.2. HTML

Este es el controlador con mayor configurabilidad de los diseñados debido a que la pantalla muestra una página HTML[22], como la de la Figura 21, completamente diseñada por el usuario, lo cual sólo pone como límite la imaginación de este.

En términos de diseño de Android, esta pantalla está compuesta por un WebView que permite mostrar contenido web como parte del diseño de la aplicación.

1.3. GALLERY (MODIFICACIÓN)

Se ha modificado esta pantalla con el objetivo de permitir tarjetas expandibles por lo que su estructura para su diseño en términos de Android se ha queda de la siguiente forma:

- *Toolbar*: barra de aplicación superior.
- *RecycleView*: contenedor con las vistas de cada uno de los elementos del controlador. Cuenta con dos tipos de vistas: una expansible y otra expandida.

  Vista expansible

  · Dos *ImageView*: una corresponde a la imagen del elemento y la otra al icono de destacados.

  · Tres *TextView*: el primero indica el nombre del elemento, el segundo es un subtítulo y el tercero una breve descripción.

  · Button: botón para compartir.

  Vista expandida

  · Dos *TextView*: el primero es un subtítulo y el segundo una breve descripción.

  · *Button*: botón para compartir.

Figura 22. Diseño pantalla Gallery (modificado)

2. MODELO

El apartado de modelo del *framework* es el que constituye todas las interfaces y clases relacionadas con la obtención de datos del archivo JSON, mapeado de datos a categorías y productos, así como el almacenado de ellos en la base de datos.

Este proyecto se ha implementado con una base de datos SQLite[23] y se ha empleado la biblioteca de persistencias Room[24] de Android, la cual brinda una capa de abstracción para SQLite.

Room crea una memoria caché de los datos en el dispositivo donde se ejecuta la aplicación y emplea tres componentes principales para el acceso a la base de datos localizada en la memoria caché:

- Base de datos: Contiene el titular de la base de datos y sirve como punto de acceso principal para la conexión subyacente a los datos persistentes y relacionales de la aplicación.
- Entidad: Representa una tabla dentro de la base de datos.
- DAO: Contiene los métodos utilizados para acceder a la base de datos.

En la Figura 23, se puede apreciar un esquemático con la comunicación entre la aplicación y la base de datos Room.

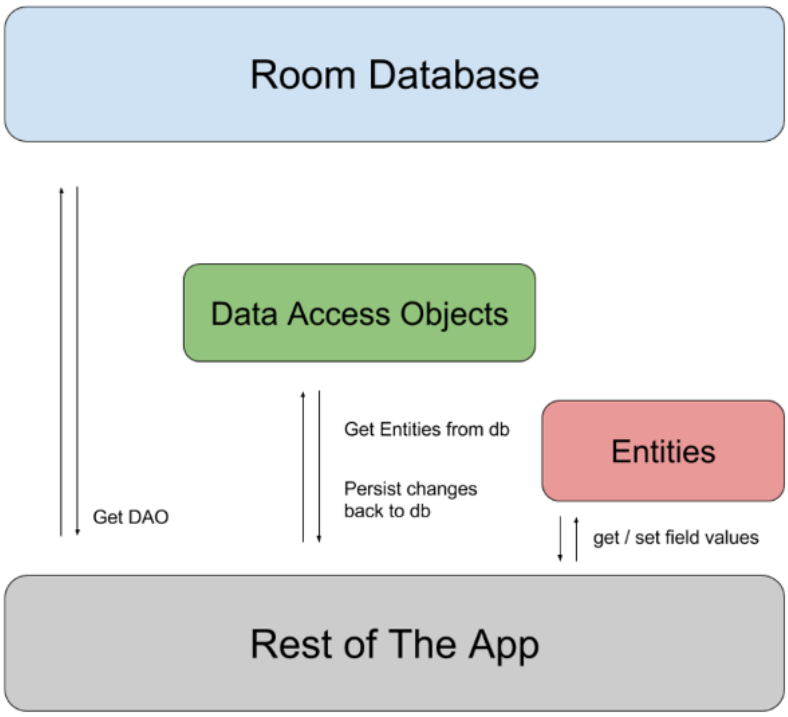

Figura 23. Diagrama de la arquitectura de Room

Como se puede ver Figura 23, la aplicación interactúa con la base de datos de Room con la finalidad de obtener los Objetos de Acceso a Datos (DAOs)[25] asociados con esa base de datos, que son los que definen las interacciones con las diferentes tablas de información presentes en la base de datos. Posteriormente, la aplicación usa cada DAO para obtener las Entidades (Entities), es decir, las clases que mapean las tablas de la base de datos. Por último, la aplicación usa una entidad para obtener y configurar valores que corresponden a columnas o filas de tabla dentro de la base de datos.

2.1. ENTIDADES

En esta sección se detallarán en profundidad que son las entidades nombradas anteriormente, es decir, aquellas clases que tienen una tabla de mapeo SQLite en la base de datos.

En la Figura 24 se puede ver un esquema UML (Unified Modeling Language)[22] de las entidades implicadas en este *framework* con sus variables y métodos que posteriormente se explicaran en detalle.

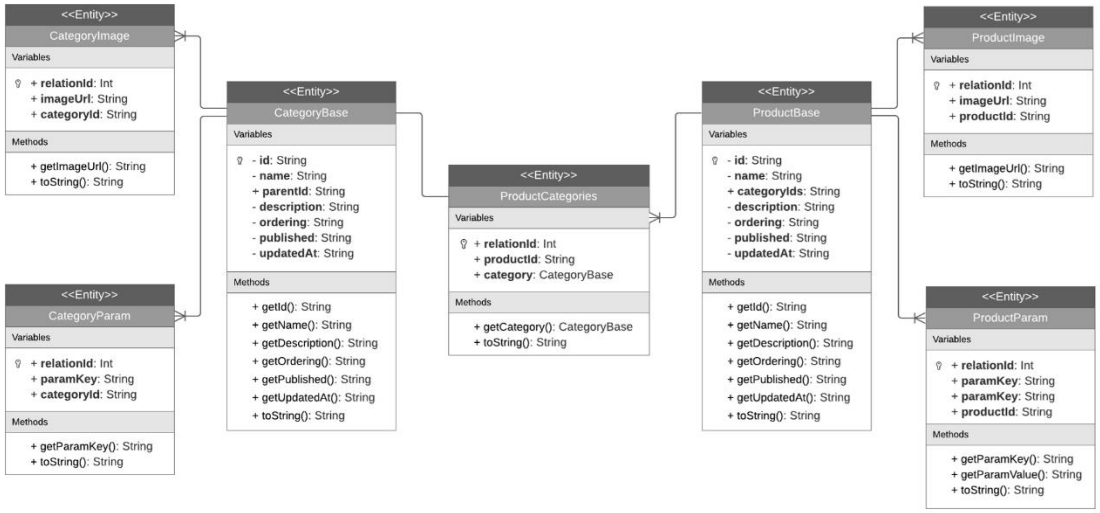

Figura 24. Esquema de las entidades que mapean las tablas de la base de datos

2.1.1. ENTIDADES RELACIONADAS CON LAS CATEGORÍAS

A continuación, se especificarán aquellas identidades que mapean tablas de la base de datos relacionadas con las categorías.

CATEGORYBASE

```
public class CategoryBase
```

Entidad correspondiente a la tabla *categories* de la base de datos, tabla que almacena los valores principales de una categoría.

Variables	
String[26]	id

	Identificador de una categoría y clave principal no nula de la tabla.
String	name Nombre de una categoría.
String	parentId Identificador de la categoría padre en la jerarquía de la aplicación.
String	description Descripción de una categoría.
String	ordering Número correspondiente al orden de aparición de la categoría en la aplicación.
String	published Indica si la categoría se muestra o no en la aplicación.
String	updatedAt Última fecha de modificación de la categoría.

Constructor
CategoryBase (String id, String name, String description, String parentId, String ordering, String published, String updatedAt) Crea un objeto categoría con los datos pasados por parámetros.

Métodos públicos	
String	getId () Obtiene el identificador de una categoría.
String	getName ()

	Obtiene el nombre de una categoría.
String	getDescription () Obtiene la descripción de una categoría.
String	getOrdering () Obtiene el orden de aparición en la aplicación de una categoría.
String	getPublished () Obtiene el sí una categoría se muestra o no.
String	getUpdatedAt () Obtiene la última fecha de modificación de una categoría.
String	toString () Obtiene el nombre de una categoría.

CATEGORYIMAGE

```
public class CategoryImage
```

Entidad correspondiente a la tabla *category_images* de la base de datos, tabla que almacena las URL de imágenes de una categoría.

Variables	
Int[26]	relationId Identificador autogenerado no nulo de la relación entre una categoría y una imagen.
String	imageUrl URL (Uniform Resource Locator)[27] de una imagen representativa de una categoría.

String	categoryId Identificador de la categoría asociada a la imagen.

Constructor
CategoryImage (String imageUrl, String categoryId) Crea un objeto imagen de categoría con los datos pasados por parámetros.

Métodos públicos	
String	getImageUrl () Obtiene la URL de una imagen de una categoría.
String	toString () Obtiene la URL de una imagen de una categoría.

CATEGORYPARAM

```
public class CategoryParam
```

Entidad correspondiente a la tabla *category_params* de la base de datos, tabla que almacena los parámetros de una categoría.

Variables	
Int	relationId Identificador autogenerado no nulo de la relación entre una categoría y un parámetro.
String	paramKey Nombre de un parámetro.
String	categoryId

	Identificador de la categoría asociada al parámetro.

Constructor
`CategoryParam (String paramKey, String categoryId)` Crea un objeto parámetro de categoría con los datos pasados por parámetros.

Métodos públicos	
String	`getParamKey ()` Obtiene el nombre de un parámetro de una categoría.
String	`toString ()` Obtiene el nombre de un parámetro de una categoría.

2.1.2. Entidades relacionadas con los productos

En esta sección, se especificarán aquellas identidades que mapean tablas de la base de datos relacionadas con los productos.

ProductBase

```
public class ProductBase
```

Entidad correspondiente a la tabla *products* de la base de datos, tabla que almacena los valores principales de un producto.

Variables	
String	`id` Identificador de un producto y clave principal no nula de la tabla.
String	`name`

	Nombre de un producto.
String	categoryIds Categorías a las que pertenece un producto.
String	description Descripción de un producto.
String	ordering Número correspondiente al orden de aparición del producto en la aplicación.
String	published Indica si el producto se muestra o no en la aplicación.
String	updatedAt Última fecha de modificación del producto.

Constructor
ProductBase (String id, String name, String description, String categoryIds, String ordering, String published, String updatedAt) Crea un objeto producto con los datos pasados por parámetros.

Métodos públicos	
String	getId () Obtiene el identificador de un producto.
String	getName () Obtiene el nombre de un producto.
String	getDescription ()

	Obtiene la descripción de un producto.
String	getOrdering () Obtiene el orden de aparición en la aplicación de un producto.
String	getPublished () Obtiene el sí un producto se muestra o no.
String	getUpdatedAt () Obtiene la última fecha de modificación de un producto.
String	toString () Obtiene el nombre de un producto.

PRODUCTIMAGE

```
public class ProductImage
```

Entidad correspondiente a la tabla *product_images* de la base de datos, tabla que almacena las URLs de las imágenes de un producto.

Variables	
Int	relationId Identificador autogenerado no nulo de la relación entre un producto y una imagen.
String	imageUrl URL (Uniform Resource Locator)[27] de una imagen.
String	productId Identificador del producto asociada a la imagen.

Constructor
`ProductImage (String imageUrl, String productId)` Crea un objeto imagen de producto con los datos pasados por parámetros.

Métodos públicos	
String	`getImageUrl ()` Obtiene la URL de una imagen de un producto.
String	`toString ()` Obtiene la URL de una imagen de un producto.

PRODUCTPARAM

```
public class ProductParam
```

Entidad correspondiente a la tabla *product_params* de la base de datos, tabla que almacena los parámetros de un producto.

Variables	
Int	`relationId` Identificador autogenerado no nulo de la relación entre un producto y un parámetro.
String	`paramKey` Nombre de un parámetro.
String	`paramValue` Valor de un parámetro.
String	`productId`

	Identificador del producto asociada al parámetro.

Constructor
`ProductParam (String paramKey, String paramValue, String productId)` Crea un objeto parámetro de producto con los datos pasados por parámetros.

Métodos públicos	
String	`getParamKey ()` Obtiene el nombre de un parámetro de un producto.
String	`getParamValue ()` Obtiene el valor de un parámetro de un producto.
String	`toString ()` Obtiene el parámetro y su valor de un producto

PRODUCTCATEGORIES

```
public class ProductCategories
```

Entidad correspondiente a la tabla *product_categories de* la base de datos, tabla que almacena la relación entre un producto y su categoría asociada.

Variables	
Int	`relationId` Identificador autogenerado no nulo de la relación entre un producto y una categoría.
String	`productId`

	Identificador de un producto.
Categorybase	category Campo de una categoría incrustado que permite que los campos anidados (es decir, campos de la clase *CategoryBase*) sean referenciados directamente en las consultas SQL.

Constructor
ProductCategories (String productId, Categorybase category) Crea un objeto que relaciona un producto y una categoría con los datos pasados por parámetro.

Métodos públicos	
Categorybase	getCategory () Obtiene la categoría asociada a un producto.
String	toString () Obtiene el nombre de la categoría asociada a un producto.

2.2. MODELOS JSON

En esta sección se describen aquellas clases que se encargan de mapear el archivo JSON con el catálogo recibido a las diferentes tablas de la base de datos. Mapeo JSON - Categorías

Este apartado incluye aquellas clases que se encargan de mapear las categorías provenientes del JSON a la base de datos.

CATALOGCATEGORIES

```
public class CatalogCategories
```

Esta clase gestiona el JSON de categorías.

Variables privadas		
status	String	status Estado del JSON que contiene las categorías.
results	List[26]<CatalogCategory>	results Lista con las categorías del JSON.

Métodos públicos	
String	getStatus () Obtiene el estado del JSON que contiene las categorías.
void	setStatus (String status) Modifica el estado del JSON que contiene las categorías.
List<CatalogCategory>	getResults () Obtiene la lista de categorías del JSON.
void	setResults (List<CatalogCategory> results) Modifica la lista de categorías del JSON.
String	toString () Obtiene la lista de categorías del JSON.

CATALOGCATEGORY

```
public class CatalogCategory
```

Clase que mapea una categoría a las correspondientes tablas de la base de datos.

Variables privadas		
spider_id	String	`id` Identificador de una categoría.
name	String	`name` Nombre de una categoría.
parent	String	`parentId` Identificador de la categoría superior en la jerarquía de la aplicación.
category_image_url	String	`imageUrls` URLs de las imágenes de una categoría.
description	String	`description` Descripción de una categoría.
param	String	`params` Parámetros de una categoría.
ordering	String	`ordering` Número que indica el orden de aparición de una categoría en la aplicación.
published	String	`published` Indica si una categoría se muestra en la aplicación o no.
updatedAt	String	`updatedAt` Fecha de la última modificación de una categoría.

Métodos públicos	
String	getId () Obtiene el identificador de la categoría.
void	setId (String id) Cambia el identificador de la categoría por el pasado por parámetro.
String	getName () Obtiene el nombre de la categoría.
void	setName (String name) Cambia el nombre de la categoría por el pasado por parámetro.
String	getParentId () Obtiene el identificador de la categoría superior de la categoría en la jerarquía de la aplicación.
void	setParentId (String parentId) Cambia el identificador de la categoría superior de la categoría en la jerarquía de la aplicación por el pasado por parámetro.
String	getImageUrls () Obtiene las URLs de las imágenes de la categoría.
void	setImageUrls (String imageUrls) Cambia las URLs de las imágenes de la categoría por las pasadas por parámetro.
String	getDescription () Obtiene la descripción de la categoría.
void	setDescription (String description)

	Cambia la descripción de la categoría por la pasada por parámetro.
String	getParams () Obtiene el nombre de los parámetros de la categoría.
void	setParams (String params) Cambia el nombre de los parámetros de la categoría por los pasados por parámetro.
String	getOrdering () Obtiene el número que indica el orden de aparición de la categoría en la aplicación.
void	setOrdering (String ordering) Cambia el número que indica el orden de aparición de la categoría en la aplicación por el pasado por parámetro.
String	getPublished () Obtiene si la categoría es publicada o no en la aplicación.
void	setPublished (String published) Cambia el valor de si la categoría es publicada o no con el valor pasado por parámetro.
String	getUpdatedAt () Obtiene la última fecha de modificación de la categoría.
void	setUpdatedAt (String updatedAt) Cambia la última fecha de modificación de la categoría por la pasada por parámetro.
String	toString () Obtiene el nombre de la categoría.

2.2.1. Mapeo JSON - Productos

Este apartado incluye aquellas clases que se encargan de mapear los productos provenientes del JSON a la base de datos.

CatalogProducts

```
public class CatalogProducts
```

Esta clase gestiona el JSON de productos.

Variables privadas		
status	String	status Estado del JSON que contiene los productos.
results	List<CatalogProduct>	results Lista con los productos del JSON.

Métodos públicos	
String	getStatus () Obtiene el estado del JSON que contiene los productos.
void	setStatus (String status) Modifica el estado del JSON que contiene los productos.
List<CatalogProduct>	getResults () Obtiene la lista de productos del JSON.
void	setResults (List<CatalogProduct> results) Modifica la lista de productos del JSON.
String	toString ()

	Obtiene la lista de productos del JSON.

CatalogProduct

```
public class CatalogProduct
```

Clase que mapea un producto a las correspondientes tablas de la base de datos.

Variables privadas		
spider_id	String	id Identificador de un producto.
name	String	name Nombre de un producto.
category_id	String	categoryIds Identificadores de las categorías superiores en la jerarquía de la aplicación.
image_url	String	imageUrls URLs de las imágenes de un producto.
description	String	description Descripción de un producto.
param	String	params Parámetros de un producto.
cost	String	cost Precio de compra de un producto.
market_cost	String	marketCost Precio de venta de un producto.

ordering	String	ordering Número que indica el orden de aparición de un producto en la aplicación.
published	String	published Indica si un producto se muestra en la aplicación o no.
createdAt	String	createdAt Fecha de creación de un producto.
updatedAt	String	updatedAt Fecha de la última modificación de un producto.

Métodos públicos	
String	getId () Obtiene el identificador del producto.
void	setId (String id) Cambia el identificador del producto por el pasado por parámetro.
String	getName () Obtiene el nombre del producto.
void	setName (String name) Cambia el nombre del producto por el pasado por parámetro.
String	getCategoryIds () Obtiene los identificadores de las categorías superiores del producto en la jerarquía de la aplicación.
void	setCategoryIds (String categoryIds)

	Cambia los identificadores de las categorías superiores del producto en la jerarquía de la aplicación por los pasados por parámetro.
String	getCost () Obtiene el precio de compra del producto.
void	setCost (String cost) Modifica el precio de compra del producto.
String	getMarketCost () Obtiene el precio de venta del producto.
void	setMarketCost (String marketCost) Modifica el precio de venta de un producto.
String	getImageUrls () Obtiene las URLs de las imágenes del producto.
void	setImageUrls (String imageUrls) Cambia las URLs de las imágenes del producto por las pasadas por parámetro.
String	getDescription () Obtiene la descripción del producto.
void	setDescription (String description) Cambia la descripción del producto por la pasada por parámetro.
String	getParams () Obtiene los parámetros del producto.
void	setParams (String params) Cambia los parámetros del producto por los pasados por parámetro.

String	getOrdering () Obtiene el número que indica el orden de aparición del producto en la aplicación.
void	setOrdering (String ordering) Cambia el número que indica el orden de aparición del producto en la aplicación por el pasado por parámetro.
String	getPublished () Obtiene si el producto es publicado o no en la aplicación.
void	setPublished (String published) Cambia el valor de si el producto es publicado o no con el valor pasado por parámetro.
String	getCreatedAt () Obtiene la fecha de creación del producto.
void	setCreatedAt (String createdAt) Cambia la fecha de creación del producto por la pasada por parámetro.
String	getUpdatedAt () Obtiene la última fecha de modificación del producto.
void	setUpdatedAt (String updatedAt) Cambia la última fecha de modificación del producto por la pasada por parámetro.
String	toString () Obtiene el nombre del producto.

2.3. SPIDER CATALOG API

Este apartado detalla la interfaz que necesita la librearía Retrofit[28] para acceder al servidor con el *SpiderCatalog* y poder descargar el catálogo de categorías y productos en formato JSON que configura la aplicación, de forma gráfica se puede ver este funcionamiento en la

Figura 25. Asimismo, Retrofit se encarga de generar la implementación necesaria para la ejecución de la interfaz.

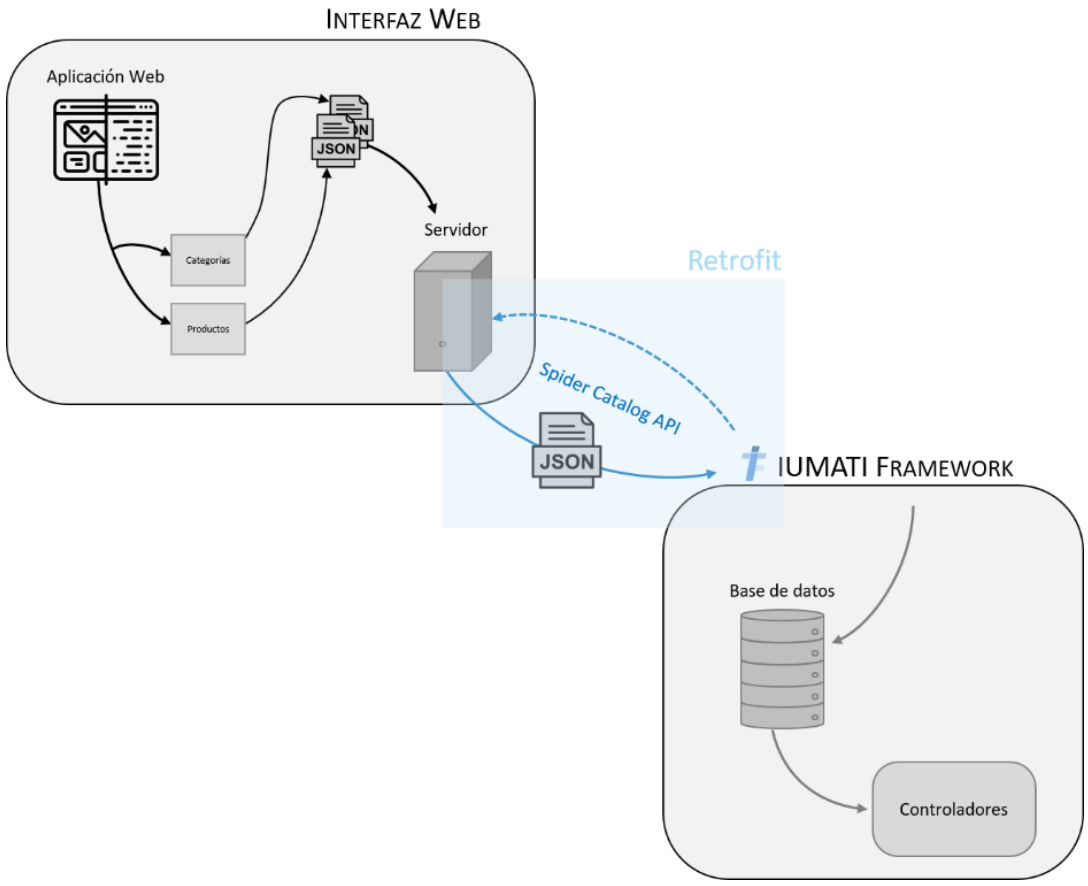

Figura 25. Esquemático del funcionamiento de Retrofit

CATALOGAPI

```
public interface CatalogApi
```

En esta interfaz se declaran métodos que realizan solicitudes al servidor web remoto que aloja el *SpiderCatalog* para obtener los datos que necesita el *framework* para crear la aplicación.

Métodos públicos	
Call<CatalogCategories>	getCatalogCategories(@Path("appId") String appId) Obtiene el catálogo de categorías haciendo una solicitud sincrónica o asincrónica al servidor web remoto.
Call<CatalogProducts>	getCatalogProducts (@Path("appId") String appId) Obtiene el catálogo de productos haciendo una solicitud sincrónica o asincrónica al servidor web remoto.

2.4. Interfaces

En esta sección se describen las interfaces que modelan los métodos necesarios tanto de las categorías como los productos que se emplean en el *framework*.

Category

```
public interface Category
```

Indica los métodos que debe implementar una categoría.

Métodos	
String	getId () Obtiene el identificador de la categoría.
String	getName () Obtiene el nombre de la categoría.
List<String>	getImageUrls () Obtiene las URLs de las imágenes la categoría.
String	getDescription ()

	Obtiene la descripción de la categoría.
List<String>	getParams () Obtiene los parámetros de la categoría.
String	getOrdering () Obtiene el número que indica el orden de aparición de la categoría en la aplicación.
String	getPublished () Obtiene si la categoría es publicada o no en la aplicación.
String	getUpdatedAt () Obtiene la última fecha de modificación de la categoría.
Category	getParent () Obtiene la categoría superior a la categoría en la jerarquía de la aplicación.
Product	getProduct () Obtiene el producto principal de la categoría.

PRODUCT

```
public interface Product
```

Indica los métodos que debe implementar un producto.

Métodos	
String	getId () Obtiene el identificador del producto.

String	getName () Obtiene el nombre del producto.
List<String>	getImageUrls () Obtiene las URLs de las imágenes del producto.
String	getDescription () Obtiene la descripción del producto.
Map<String, String>	getParams () Obtiene los parámetros del producto.
String	getOrdering () Obtiene el número que indica el orden de aparición del producto en la aplicación.
String	getPublished () Obtiene si el producto es publicado o no en la aplicación.
String	getUpdatedAt () Obtiene la última fecha de modificación del producto.
List<? extends String>	getCategories () Obtiene las categorías superiores del producto en la jerarquía de la aplicación.

2.5. Modelos de Alto Nivel

Estas clases son las que se encargan de acomodar los datos procedentes de la base de datos a las necesidades requeridas por los controladores, como se puede apreciar en la Figura 26.

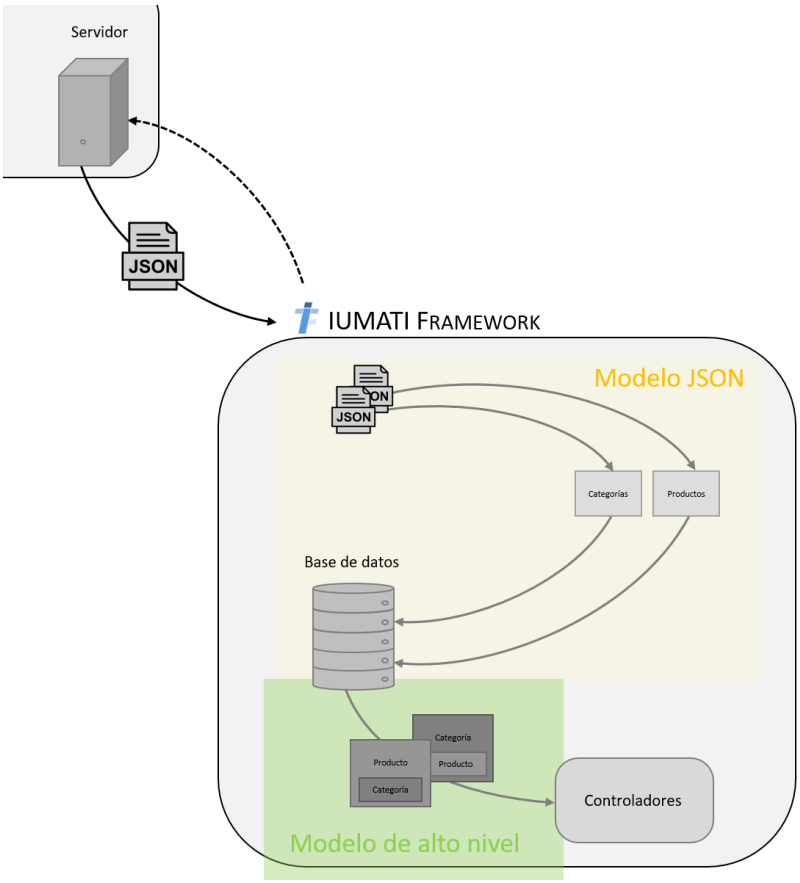

Figura 26. Esquemático de los modelos que interactuan con la base de datos

CATEGORYWITHRELATIONS

```
public class CategoryWithRelations implements Category
```

Esta clase se encarga de la creación de categorías completas, es decir, una categoría con sus imágenes y parámetros asociados en un solo objeto.

Métodos de inserción	
CategoryBase	category Datos básicos de una categoría.
List<CategoryParam>	params Lista de los parámetros de una categoría.

List<CategoryImage>	images Lista de imágenes de una categoría.
List<CategoryBase>	parents Lista de categorías superiores en la jerarquía.
CategoryWithRelations	parent Categoría superior a la categoría en la jerarquía de la aplicación.
ProductWithRelations	product Producto principal de la categoría.

Métodos públicos	
String	getId () Obtiene el identificador de la categoría.
String	getName () Obtiene el nombre de la categoría.
List<String>	getImageUrls () Obtiene las URLs de las imágenes de la categoría a partir de la lista de imágenes de categoría de la categoría.
String	getDescription () Obtiene la descripción de la categoría.
List<String>	getParams () Obtiene los parámetros de la categoría a partir de la lista de parámetros de categoría de la categoría.
String	getOrdering ()

	Obtiene el número que indica el orden de aparición de la categoría en la aplicación.
String	getPublished () Obtiene si la categoría es publicada o no en la aplicación.
String	getUpdatedAt () Obtiene la última fecha de modificación de la categoría.
CategoryWithRelations	getParent () Obtiene la categoría superior a la categoría en la jerarquía de la aplicación.
void	setParent (CategoryWithRelations parent) Modifica la categoría superior en la jerarquía de la categoría por la pasada por parámetro.
ProductWithRelations	getProduct () Obtiene el producto principal de la categoría.
void	setProduct (ProductWithRelations product) Modifica el producto principal de la categoría por el pasado por parámetro.
CategoryBase	getParentBase () Si la lista de categorías superiores en la jerarquía a la categoría es nula obtiene un valor nulo, en caso contrario obtiene los datos principales de la primera categoría superior en la jerarquía.
String	toString () Obtiene el nombre de una categoría.

PRODUCTWITHRELATIONS

```
public class ProductWithRelations implements Product
```

Esta clase se encarga de la creación de productos completos, es decir, un producto con sus imágenes y parámetros asociados en un solo objeto.

Métodos de inserción	
ProductBase	product Datos básicos de un producto.
List<ProductParam>	params Lista de los parámetros de un producto.
List<ProductImage>	images Lista de imágenes de un producto.
List<ProductCategories>	productCategories Lista de categorías a las que pertenece el producto.
List<CategoryWithRelations>	categories Lista de categorías completas a las que pertenece el producto.

Métodos públicos	
String	getId () Obtiene el identificador del producto.
String	getName () Obtiene el nombre del producto.
List<String>	getImageUrls ()

	Obtiene las URLs de las imágenes del producto a partir de la lista de imágenes de producto del producto.
String	getDescription () Obtiene la descripción del producto.
List<String>	getParams () Obtiene los parámetros del producto a partir de la lista de parámetros de producto del producto.
String	getOrdering () Obtiene el número que indica el orden de aparición de la categoría en la aplicación.
String	getPublished () Obtiene si la categoría es publicada o no en la aplicación.
String	getUpdatedAt () Obtiene la última fecha de modificación de la categoría.
List<CategoryWithRelations>	getCategories () Obtiene la lista de categorías completas a las que pertenece el producto.
void	setCategories(List<CategoryWithRelations> categories) Modifica la lista de categorías completas a las que pertenece el producto por la pasada por parámetro.
List<ProductCategories>	getProductCategories ()

	Obtiene la lista de categorías a las que pertenece el producto.
String	toString () Obtiene el nombre del producto.

2.6.DAOs

Para acceder a los datos de la aplicación, Room trabaja con Objetos de Acceso a Datos (DAO), en ellos se incluyen los métodos que ofrecen acceso abstracto a la base de datos, es decir, los objetos de acceso a datos son las clases principales donde se definen las interacciones con la base de datos.

2.6.1. CATEGORÍAS

A continuación, se especificarán DAOs relacionados con las categorías.

CATEGORIESDAO

```
public interface CategoriesDao
```

DAO que define las interacciones con las categorías de la base de datos.

Métodos de inserción	
void	insert (CategoryBase category) Inserta en la base de datos la categoría pasada por parámetro.
void	insert (List<CategoryBase> categoryList) Inserta en la base de datos las categorías pasadas por parámetro.
void	insert (CategoryBase... categories) Inserta en la base de datos desde una a varias categorías.

Métodos de modificación	
`void`	`update (CategoryBase category)` Modifica la categoría pasada por parámetro de la base de datos.
`void`	`update (List<CategoryBase> categoryList)` Modifica las categorías pasadas por parámetro de la base de datos.
`void`	`update (CategoryBase... categories)` Modifica desde una a varias categorías pasadas por parámetro de la base de datos.

Métodos de eliminación	
`void`	`delete (CategoryBase category)` Elimina la categoría pasada por parámetro de la base de datos.
`void`	`delete (CategoryBase... categories)` Elimina una o varias categorías de la base de datos.

Métodos de consulta	
`List<CategoryBase>`	`getCategories ()` Obtiene todas las categorías de la base de datos y las organiza de forma ascendente.
`CategoryBase`	`getCategory (final String id)` Obtiene la categoría en la base de datos correspondiente al identificador pasado por parámetro.
`void`	`delete ()` Elimina todas las categorías de la base de datos.

CATEGORYIMAGESDAO

```
public interface CategoryImagesDao
```

DAO que define las interacciones con las URLs de las imágenes de las categorías de la base de datos.

Métodos de inserción	
void	insert (CategoryImage image) Inserta en la base de datos la imagen de categoría pasada por parámetro.
void	insert (List<CategoryImage> imageList) Inserta en la base de datos las imágenes de categoría pasadas por parámetro.
void	insert (CategoryImage... images) Inserta en la base de datos una o varias imágenes de categoría.

Métodos de modificación	
void	update (CategoryImage... images) Modifica una o varias imágenes de categoría de la base de datos.

Métodos de eliminación	
void	delete (CategoryImage... images) Elimina una o varias imágenes de categoría de la base de datos.

Métodos de consulta	
List<CategoryImage>	getImages () Obtiene todas las imágenes de categorías de la base de datos.
List<CategoryImage>	getImages (final int categoryId)

	Obtiene todas las imágenes de categoría de la base de datos cuya categoría corresponda con el identificador pasado por parámetro.
void	`delete ()` Elimina todas las imágenes de categoría de la base de datos.

CATEGORYPARAMSDAO

```
public interface CategoryParamsDao
```

DAO que define las interacciones con los parámetros de las categorías de la base de datos.

Métodos de inserción	
void	`insert (CategoryParam param)` Inserta en la base de datos el parámetro de categoría pasado por parámetro.
void	`insert (List<CategoryParam> paramList)` Inserta en la base de datos los parámetros de categoría pasados por parámetro.
void	`insert (CategoryParam... params)` Inserta en la base de datos uno o varios parámetros de categoría.

Métodos de modificación	
void	`update (CategoryParam param)` Modifica el parámetro de categoría pasado por parámetro de la base de datos.
Void	`update (CategoryParam... params)`

	Modifica uno o varios parámetros de categoría de la base de datos.

Métodos de eliminación	
`void`	`delete (CategoryParam param)` Elimina el parámetro de categoría pasado por parámetro de la base de datos.
`void`	`delete (CategoryParam... params)` Elimina uno o varios parámetros de categoría de la base de datos.

Métodos de consulta	
`List<CategoryParam>`	`getParams (final int categoryId)` Obtiene todos los parámetros de categoría de la base de datos cuya categoría corresponda con el identificador pasado por parámetro.
`void`	`delete ()` Elimina todos los parámetros de categoría de la base de datos.

CategoriesWithRelationsDao

```
public interface CategoriesWithRelationsDao
```

DAO que define las interacciones de las categorías que emplean los modelos de alto nivel.

Métodos de consulta	
`List< CategoryWithRelations>`	`getCategoriesWithRelations ()` Obtiene todas las categorías de la base de datos y las coloca en orden ascendente.

List< CategoryWithRelations>	getCategoriesWithRelations (String parentId) Obtiene todas las categorías de la base de datos cuya categoría superior en la jerarquía corresponda con el identificador pasado por parámetro y las coloca en orden ascendente.
CategoryWithRelations	getCategoriesWithRelations (String id) Obtiene la categoría de la base de datos cuyo identificador corresponde con el pasado por parámetro.
CategoryWithRelations	getRootCategoryWithRelations () Obtiene la categoría de la base de datos cuya categoría superior en la jerarquía tenga el identificador cero y su orden de aparición en la aplicación sea uno.

2.6.2. PRODUCTOS

En esta sección se especificarán DAOs relacionados con los productos.

PRODUCTSDAO

```
public interface ProductsDao
```

DAO que define las interacciones con los productos de la base de datos.

Métodos de inserción	
void	insert (ProductBase product) Inserta en la base de datos el producto pasado por parámetro.
void	insert (ProductBase... products)

	Inserta en la base desde uno a varios productos pasados por parámetro.
void	insert (List<ProductBase> productList) Inserta en la base de datos los productos pasados por parámetro.

Métodos de modificación	
void	update (ProductBase product) Modifica el producto pasado por parámetro de la base de datos.
void	update (ProductBase... products) Modifica uno a varios productos pasados por parámetro de la base de datos.

Métodos de eliminación	
void	delete (ProductBase product) Elimina el producto pasado por parámetro de la base de datos.
void	delete (ProductBase... products) Elimina uno a varios productos pasados por parámetro de la base de datos.

Métodos de consulta	
List<ProductBase>	getProducts () Obtiene todos los productos de la base de datos y los organiza de forma ascendente.
ProductBase	getProduct (final String id) Obtiene la categoría en la base de datos correspondiente al identificador pasado por parámetro.

void	delete () Elimina todos los productos de la base de datos.

PRODUCTIMAGESDAO

```
public interface ProductImagesDao
```

DAO que define las interacciones con las URLs de las imágenes de los productos de la base de datos.

Métodos de inserción	
void	insert (ProductImage image) Inserta en la base de datos la imagen de producto pasada por parámetro.
void	insert (ProductImage... images) Inserta una o varias imágenes de producto pasadas por parámetro en la base de datos.
void	insert (List<ProductImage> imageList) Inserta en la base de datos las imágenes de producto pasadas por parámetro.

Métodos de modificación	
void	update (ProductImage image) Modifica la imagen de producto pasada por parámetro de la base de datos.
void	update (ProductImage... images) Modifica una o varias imágenes de producto pasadas por parámetro de la base de datos.

Métodos de eliminación	
void	delete (ProductImage image) Elimina la imagen de producto pasada por parámetro de la base de datos.
void	delete (ProductImage... images) Elimina una o varias imágenes de producto pasadas por parámetro de la base de datos.

Métodos de consulta	
List<ProductImage>	getImages () Obtiene todas las imágenes de productos de la base de datos.
List<ProductImage>	getImages (final int productId) Obtiene todas las imágenes de producto de la base de datos cuyo producto corresponda con el identificador pasado por parámetro.
void	delete () Elimina todas las imágenes de producto de la base de datos.

PRODUCTPARAMSDAO

```
public interface ProductParamsDao
```

DAO que define las interacciones con los parámetros de los productos de la base de datos.

Métodos de inserción	
void	insert (ProductParam param) Inserta en la base de datos el parámetro de producto pasado por parámetro.

void	insert (ProductParam... params) Inserta uno o varios parámetros de producto pasados por parámetro en la base de datos.
void	insert (List<ProductParam> paramList) Inserta en la base de datos los parámetros de producto pasados por parámetro.

Métodos de modificación	
void	update (ProductParam param) Modifica el parámetro de producto pasado por parámetro de la base de datos.
Void	update (ProductParam... params) Modifica uno o varios parámetros de producto pasados por parámetro de la base de datos.

Métodos de eliminación	
void	delete (ProductParam param) Elimina el parámetro de producto pasado por parámetro de la base de datos.
void	delete (ProductParam... params) Elimina uno o varios parámetros de producto pasados por parámetro de la base de datos.

Métodos de consulta	
List<ProductParam>	getParams (final int productId)

	Obtiene todos los parámetros de la base de datos cuyo producto corresponda con el identificador pasado por parámetro.
void	delete () Elimina todos los parámetros de producto de la base de datos.

PRODUCTCATEGORIESDAO

```
public interface ProductCategoriesDao
```

DAO que define las interacciones con las categorías de los productos de la base de datos.

Métodos de inserción	
void	insert (ProductCategories productCategories) Inserta en la base de datos la categoría de producto pasada por parámetro.
void	insert (ProductCategories... productCategories) Inserta una o varias categorías de producto pasadas por parámetro en la base de datos.
void	insert (List<ProductCategories> productCategories) Inserta en la base de datos las categorías de producto pasadas por parámetro.

Métodos de modificación	
void	update (ProductCategories productCategories) Modifica la categoría de producto pasada por parámetro de la base de datos.
Void	update (ProductCategories... productCategories)

	Inserta una o varias categorías de producto pasadas por parámetro de la base de datos.

Métodos de eliminación	
`void`	`delete (ProductCategories productCategories)` Elimina la categoría de producto pasada por parámetro de la base de datos.
`void`	`delete (ProductCategories... productCategories)` Elimina una o varias categorías de producto pasadas por parámetro de la base de datos.

Métodos de consulta	
`List<ProductCategories>`	`getProductCategories (final int productId)` Obtiene todas las categorías de producto de la base de datos cuyo producto corresponda con el identificador pasado por parámetro.
`void`	`delete ()` Elimina todas las categorías de producto de la base de datos.

PRODUCTSWITHRELATIONSDAO

```
public interface ProductsWithRelationsDao
```

DAO que define las interacciones con los productos que manejan los modelos de alto nivel.

Métodos de consulta	
`List< ProductWithRelations>`	`getProductsWithRelations ()`

	Obtiene todos los productos de la base de datos y las coloca en orden ascendente.
List< ProductWithRelations>	getProductsWithRelations (String categoryId) Obtiene todos los productos de la base de datos cuyo orden de aparición en la aplicación sea distinto de uno y en sus identificadores de categorías asociadas esté el identificador pasado por parámetro y los ordena por orden ascendente.
ProductWithRelations	getProductsWithRelations (String id) Obtiene el producto de la base de datos cuyo identificador corresponde con el pasado por parámetro.
ProductWithRelations	getProductsWithRelations (String categoryId) Obtiene el producto de la base de datos cuyo orden de aparición en la aplicación sea uno y en sus identificadores de categorías asociadas esté el identificador pasado por parámetro.

2.7. Base de datos y repositorio

En este apartado se explicarán las clases referentes a la base de datos y el repositorio implementados en este proyecto.

2.7.1. Base de datos

La base de datos es donde se almacenarán todos los datos referentes a productos y categorías.

CATALOGDATABASE

```
public abstract CatalogDatabase extends RoomDatabase
```

Esta clase se encarga de crear y obtener la base de datos a partir de las entidades creadas y declarar los métodos para obtener las DAOs asociadas.

Resumen

Variables	
CatalogDatabase	instance Instancia de la base de datos.

Métodos públicos	
CatalogDatabase	getInstance (Context[29] context, String appId) Obtiene la instancia de la base de datos.

Métodos privados	
CatalogDatabase	create (final Context context, String appId) Crea la base de datos.

Métodos abstractos	
CategoriesWithRelationsDao	getCategoryWithRelationsDao () Obtiene el DAO de las categorías con relaciones.
CategoriesDao	getCategoriesDao () Obtiene el DAO de las categorías.
CategoryParamsDao	getCategoryParamsDao () Obtiene el DAO de los parámetros de categoría.

CategoryImagesDao	getCategoryImagesDao () Obtiene el DAO de imágenes de categoría.
ProductsWithRelationsDao	getProductWithRelationsDao () Obtiene el DAO de los productos con relaciones
ProductsDao	getCategoriesDao () Obtiene el DAO de las categorías.
ProductParamsDao	getProductParamsDao () Obtiene el DAO de los parámetros de producto.
ProductImagesDao	getProductImagesDao () Obtiene el DAO de imágenes de producto.
ProductCategoriesDao	getProductWithCategoriesDao () Obtiene el DAO de categorías de producto.

Variables

instance

```
private static volatile CatalogDatabase instance
```

Variable que puede cambiar en cualquier momento y que almacena la instancia de la base de datos.

Métodos públicos

getInstance

```
public static synchronized CatalogDatabase getInstance (Context context,
String appId)
```

Si la variable de instancia de la base de datos es nula crea la base de datos.

Parámetros	
context	Context: contexto de la aplicación.
appId	String: identificador de la aplicación.

Retorno	
CatalogDatabase	Instancia de la base de datos.

Métodos privados

create

```
public static CatalogDatabase create (final Context context, String appId)
```

Con el contexto y el identificador de aplicación pasados por parámetros le pide al creador de bases de datos de Room que cree la aplicación.

Parámetros	
context	Context: contexto de la aplicación.
appId	String: identificador de la aplicación.

Retorno	
CatalogDatabase	Instancia de la base de datos.

2.7.2. Repositorio

Esta es la clase primordial del *framework*, se encarga de obtener el catálogo de categorías y productos, su posterior mapeo a la base de datos y gestiona todas las peticiones de datos de los diferentes modelos de los controladores de pantalla.

CatalogRepository

```
public class CatalogRepository
```

Esta clase se encarga de toda la configuración del catálogo de categorías y productos, así como del acceso a la base de datos y la gestión de peticiones de los diferentes modelos de los controladores de pantalla del *framework*.

Resumen

Variables	
Context	context Contexto de la aplicación.
String	appId Identificador de la aplicación.
CatalogApi	api Objeto de comunicación con el servidor web.
Boolean[30]	readyCategories Indica si las categorías están listas para su uso o no.
Boolean	readyProducts Indica si los productos están listos para su uso o no.
ProductsWithRelationsDao	productsWithRelationsDao Objeto que interacciona con los productos.
CategoriesWithRelationsDao	categoriesWithRelationsDao Objeto que interacciona con las categorías.
CatalogRepository	instance Instancia del repositorio.

Métodos privados	
CatalogRepository	create (Context context, String appId) Crea el repositorio.
void	persistCatalogCategories (Boolean fromJSON, CatalogPersistenceCallback readyCallback) Configura el catálogo de categorías.
void	loadCatalogCategories (CatalogPersistenceCallback callback) Carga las categorías de la aplicación.
void	downloadCatalogCategories (CatalogPersistenceCallback callback) Descarga las categorías de la aplicación.
void	asyncTaskCatalogCategories (List<CatalogCategory> results, CatalogPersistenceCallback callback) Gestiona y prepara las categorías del catálogo.
void	persistCatalogProducts (Boolean fromJSON, CatalogPersistenceCallback readyCallback) Configura el catálogo de productos.
void	loadCatalogProducts (CatalogPersistenceCallback callback) Carga los productos de la aplicación.
void	downloadCatalogProducts (CatalogPersistenceCallback callback) Descarga los productos de la aplicación.

void	asyncTaskCatalogProducts (List<CatalogProduct> results, CatalogPersistenceCallback callback) Gestiona y prepara los productos del catálogo.
List<CategoryWithRelations>	getCategories (ProductWithRelations product) Obtiene las categorías de un producto.
void	persistCategories (List<CatalogCategory> catalogCategoryList) Configura las categorías de la base de datos.
void	persistProducts (List<CatalogProduct> catalogProductList) Configura los productos de la base de datos.
List<String>	getProductCategories (String str) Obtiene una lista de categorías de producto de una cadena de caracteres.
Map[31]<String, String>	getParams (String str) Obtiene una lista asociativa de parámetros y valores de producto de una cadena de caracteres.
List<String>	getCategoryParams (String str) Obtiene una lista de parámetros de categoría de una cadena de caracteres.
List<String>	getImageUrls (String str) Obtiene una lista de URLs de imágenes de una cadena de caracteres.
List<CatalogCategory>	getCatalogCategories ()

	Obtienes las categorías del archivo JSON.
List<CatalogProduct>	getCatalogProducts () Obtienes los productos del archivo JSON.
String	getJSONFromAssets (String fileName) Obtiene el archivo JSON que almacena el catálogo de la carpeta de archivos.

Métodos públicos	
CatalogRepository	getInstance (Context context, String appId) Obtiene la instancia del repositorio.
void	persistCatalog (Boolean clearFirst, Boolean fromJSON, CatalogPersistenceCallback readyCallback) Gestiona la obtención y manejo del catálogo.
void	clearCatalog () Elimina todas las DAOs del *framework*.
Category	getRootCategory () Obtiene la categoría raíz de la aplicación.
Category	getCategory (String id) Obtiene una categoría a partir del identificador pasado por parámetro.
List<? extends Category>	getCategories (Category parent) Obtiene las categorías inferiores de la categoría pasada por parámetro en la jerarquía de la aplicación.
Product	getProduct (String id)

	Obtiene un producto a partir del identificador pasado por parámetro.
Product	getMainProduct (Category category) Obtiene el product principal de la categoría pasado por parámetro.
List<? extends Product>	getProducts (String categoryId) Obtiene todos los productos asociados al identificador de categoría pasado por parámetro.
List<? extends Product>	getProducts (Category category) Obtiene todos los productos de la categoría pasada por parámetro.

Variables

context

```
private final Context context
```

Variable que almacena el contexto actual de la aplicación.

appId

```
private final String appId
```

Identificador de la aplicación.

api

```
private final CatalogApi api
```

Objeto que implementa la interfaz de comunicación con el servidor con el plugin *SpiderCatalog* y que almacena el catálogo de categorías y productos.

readyCategories

```
private boolean readyCategories
```

Variable que indica que las categorías han sido descargadas y almacenadas correctamente en la base de datos.

readyProducts

```
private boolean readyProducts
```

Variable que indica que los productos han sido descargadas y almacenadas correctamente en la base de datos.

productsWithRelationsDao

```
private final ProductsWithRelationsDao productsWithRelationsDao
```

Variable que representa el objeto con las interacciones de los productos.

categoriesWithRelationsDao

```
private final CategoriesWithRelationsDao categoriesWithRelationsDao
```

Variable que representa el objeto con las interacciones de las categorías.

instance

```
private static volatile CatalogRepository instance
```

Variable que puede cambiar en cualquier momento y que almacena la instancia del repositorio.

Métodos privados

create

```
private static CatalogRepository create (final Context context, String appId)
```

Construye el objeto que implementa la interfaz de Retrofit y con ella, el contexto y el identificador de aplicación pasados por parámetros le crea el repositorio.

Parámetros	
context	Context: contexto de la aplicación.
appId	String: identificador de la aplicación.

Retorno	
CatalogRepository	Instancia del repositorio.

persistCatalogCategories

```
private void persistCatalogCategories (Boolean fromJSON,
CatalogPersistenceCallback readyCallback)
```

Si obtenemos a través del DAO de categorías completas que hay categorías y el *callback* no es nulo indicamos que las categorías están listas. En caso contrario, si hay que extraer los datos del archivo JSON se cargan y si no se descargan del servidor.

Parámetros	
`fromJSON`	`Boolean`: indica si el catálogo de categorías se carga desde un archivo JSON local o no.
`readyCallback`	`CatalogPersistenceCallback`.

loadCatalogCategories

```
private void loadCatalogCategories (CatalogPersistenceCallback callback)
```

Obtiene las categorías del JSON y ejecuta como tarea asíncrona la configuración y almacenaje de los datos obtenidos a la base de datos.

Parámetros	
`fromJSON`	`Boolean`: indica si el catálogo de categorías se carga desde un archivo JSON local o no.
`readyCallback`	`CatalogPersistenceCallback`.

downloadCatalogCategories

```
private void downloadCatalogCategories (CatalogPersistenceCallback callback)
```

Obtiene las categorías haciendo uso de la interfaz de conexión con el servidor web y ejecuta como tarea asíncrona la configuración y almacenaje de los datos obtenidos a la base de datos.

Parámetros	
`readyCallback`	`CatalogPersistenceCallback`.

asyncTaskCatalogCategories

```
private void asyncTaskCatalogCategories (List<CatalogCategory> results,
CatalogPersistenceCallback callback)
```

Gestiona las categorías y si están listas y además el *callback* no es nulo se indica que está todo preparado. Si las categorías no están listas son los productos.

Parámetros	
`results`	`List<CatalogCategory>`: indica si el catálogo de categorías se carga desde un archivo JSON local o no.
`readyCallback`	`CatalogPersistenceCallback`.

persistCatalogProducts

```
private void persistCatalogProducts (Boolean fromJSON,
CatalogPersistenceCallback readyCallback)
```

Si obtenemos a través del DAO de productos completos que hay productos y el *callback* no es nulo indicamos que los están listos. En caso contrario, si hay que extraer los datos del archivo JSON se cargan y si no se descargan del servidor.

Parámetros	
`fromJSON`	`Boolean`: indica si el catálogo de categorías se carga desde un archivo JSON local o no.
`readyCallback`	`CatalogPersistenceCallback`.

loadCatalogProdutcs

```
private void loadCatalogProducts (CatalogPersistenceCallback callback)
```

Obtiene los productos del JSON y ejecuta como tarea asíncrona la configuración y almacenaje de los datos obtenidos a la base de datos.

Parámetros	
`fromJSON`	`Boolean`: indica si el catálogo de categorías se carga desde un archivo JSON local o no.

readyCallback	CatalogPersistenceCallback.

downloadCatalogProducts

```
private void downloadCatalogProducts (CatalogPersistenceCallback callback)
```

Obtiene los productos haciendo uso de la interfaz de conexión con el servidor web y ejecuta como tarea asíncrona la configuración y almacenaje de los datos obtenidos a la base de datos.

Parámetros	
readyCallback	CatalogPersistenceCallback.

asyncTaskCatalogProducts

```
private void asyncTaskCatalogProducts (List<CatalogProduct> results,
CatalogPersistenceCallback callback)
```

Gestiona los productos y si están listos y además el *callback* no es nulo se indica que está todo preparado. Si las categorías no están listas son los productos.

Parámetros	
results	List<CatalogCategory>: indica si el catálogo de categorías se carga desde un archivo JSON local o no.
readyCallback	CatalogPersistenceCallback.

getCategories

```
private List<CategoryWithRelations> getCategories (ProductWithRelations
product)
```

Obtiene las categorías del producto pasado por parámetro, crea un lista de categorías y por cada producto, a través del DAO de categorías obtiene las categorías cuyo identificador corresponda con la categoría asociada al producto y la añade a la lista.

Parámetros	
product	ProductWithRelations: contexto de la aplicación.

Retorno	
List<CategoryWithRelations>	Lista de categorías asociadas al producto pasado por parámetro.

persistCategories

```
private void persistCategories (List<CatalogCategory> catalogCategoryList)
```

Obtiene los tres DAOs relacionados con las categorías y por cada una de las categorías de la lista pasada por parámetro obtiene sus imágenes y parámetros para posteriormente insertarlos en el respectivo DAO.

Parámetros	
catalogCategoryList	List<CatalogCategory>: lista de categorías.

persistProducts

```
private void persistProducts (List<CatalogProduct> catalogProductList)
```

Obtiene los cuatro DAOs relacionados con los productos y por cada uno de los productos de la lista pasada por parámetro obtiene sus imágenes y parámetros para posteriormente insertarlos en el respectivo DAO.

Parámetros	
catalogCategoryList	List<CatalogProduct>: lista de productos.

getProductCategories

```
private List<String> getProductCategories (String str)
```

Crea una lista donde almacenar los identificadores de categorías y luego recorre la cadena de texto que previamente se ha convertido a *array*, si no está vacío inserta cada elemento en la lista de identificadores de categoría.

Parámetros	
str	String: cadena de caracteres con los identificadores de categorías.

Retorno	
List<String>	Lista de identificadores de las categorías asociadas a un producto.

getParams

```
private Map<String, String> getParams (String str)
```

Crea un *array* asociativo en donde almacenar el nombre de parámetro de producto y su valor correspondiente y luego recorre la cadena de texto que previamente se ha convertido a *array* y va insertando cada elemento.

Parámetros	
str	String: cadena de caracteres con los parámetro y valores de un producto.

Retorno	
Map<String, String>	Lista asociativa de parámetros y valores de parámetro.

getCategoryParams

```
private List<String> getCategoryParams (String str)
```

Crea una lista donde almacenar los parámetros de categoría y luego recorre la cadena de texto que previamente se ha convertido a *array*, si no está vacío inserta cada elemento en la lista de parámetros de categoría.

Parámetros	
str	String: cadena de caracteres con los parámetros de una categoría.

Retorno	
List<String>	Lista de parámetros de una categoría.

getImageUrls

```
private List<String> getImageUrls (String str)
```

Crea una lista donde almacenar las imágenes de categoría y luego recorre la cadena de texto que previamente se ha convertido a *array*, si no está vacío inserta cada elemento en la lista de imágenes de categoría.

Parámetros	
`str`	`String`: cadena de caracteres con las URLs de imágenes de una categoría.

Retorno	
`List<String>`	Lista de imágenes de una categoría.

getCatalogCategories

```
private List<CatalogCategory> getCatalogCategories ()
```

Obtiene el archivo JSON con las categorías y lo mapea a categorías.

Retorno	
`List<CatalogCategory>`	Lista de categorías.

getCatalogProducts

```
private List<CatalogProduct> getCatalogProducts ()
```

Obtiene el archivo JSON con los productos y lo mapea a productos.

Retorno	
`List<CatalogProduct>`	Lista de productos.

getJSONFromAssets

```
private String getJSONFromAssets (String fileName)
```

Localiza el archivo y extrae el contenido en una cadena de caracteres con el formato de un JSON.

Parámetros	
`str fileName`	`String`: nombre del archivo.

Retorno	
String	Cadena de caracteres con el formato de un JSON

Métodos públicos

getInstance

```
public static synchronized CatalogRepository getInstance (Context context, String appId)
```

Si la variable de instancia de la base de datos es nula crea la base de datos.

Parámetros	
context	Context: contexto de la aplicación.
appId	String: identificador de la aplicación.

Retorno	
CatalogRepository	Instancia del repositorio.

persistCatalog

```
public void persistCatalogCategories (Boolean clearFirst, Boolean fromJSON, CatalogPersistenceCallback readyCallback)
```

En un hilo de ejecución secundario limpia el catálogo si la variable *clearFirst* en verdadera y gestiona el catálogo de categorías y productos.

clearCatalog

```
public void clearCatalog ()
```

Obtiene cado uno de los DAOs del *framework* para posteriormente eliminarlos.

getRootCategory

```
public Category getRootCategory ()
```

A través del DAO de categorías obtiene la categoría raíz y modifica la variable de categoría superior a el valor nulo. Además, modifica su producto asociado obteniéndolo previamente.

Retorno	
Category	Categoría raíz de la aplicación.

getCategory

```
public Category getCategory (String id)
```

A través del DAO de categorías obtiene la categoría cuyo identificador corresponde con el pasado por parámetro, obtiene la categoría superior de la aplicación y se la incorpora, al igual que modifica su producto asociado obteniéndolo previamente.

Parámetros	
id	String: identificador de una categoría.

Retorno	
Category	Categoría de la aplicación.

getCategories

```
public List<? extends Category> getCategory (String id)
```

A través del DAO de categorías obtiene la lista de categorías cuyo identificador de la categoría superior corresponde con el identificador la categoría pasada por parámetro. Posteriormente, por cada una de las categorías modifica su categoría superior por la pasada por parámetro y modifica su producto asociado obteniéndolo previamente.

Parámetros	
parent	Category: categoría de la aplicación.

Retorno	
List<? extends Category>	Lista de categorías inferiores de la categoría.

getProduct

```
public Product getProduct (String id)
```

A través del DAO de productos obtiene el producto cuyo identificador corresponde con el pasado por parámetro y modifica sus categorías asociadas obteniéndolas previamente.

Parámetros	
id	String: identificador de una categoría.

Retorno	
Product	Producto de la aplicación.

getMainProduct

```
public void getMainProduct (Category category)
```

A través del DAO de productos obtiene el producto asociado al identificador de la categoría pasada por parámetro. Asimismo, modifica las categorías a las que pertenece obteniéndolas previamente.

Parámetros	
id	String: identificador de una categoría.

Retorno	
Product	Producto de la aplicación.

getProducts

```
public List<? extends Product> getCategory (String categoryId)
```

A través del DAO de productos obtiene la lista de productos cuyo identificador de la categoría a la que pertenecen corresponde con el identificador de la categoría pasada por parámetro. Posteriormente, por cada uno de los productos modifica sus categorías obteniéndolas previamente.

Parámetros	
categoryId	String: identificador de una categoría.

Retorno	
List<? extends Product>	Lista de productos de la categoría.

getProducts

```
public List<? extends Product> getCategory (Category category)
```

Obtiene la lista de productos cuyo identificador de la categoría a la que pertenecen corresponde con el identificador de la categoría pasada por parámetro.

Parámetros	
category	Category: categoría de la aplicación.

Retorno	
List<? extends Product>	Lista de productos de la categoría.

3. LÓGICA DE LA APLICACIÓN

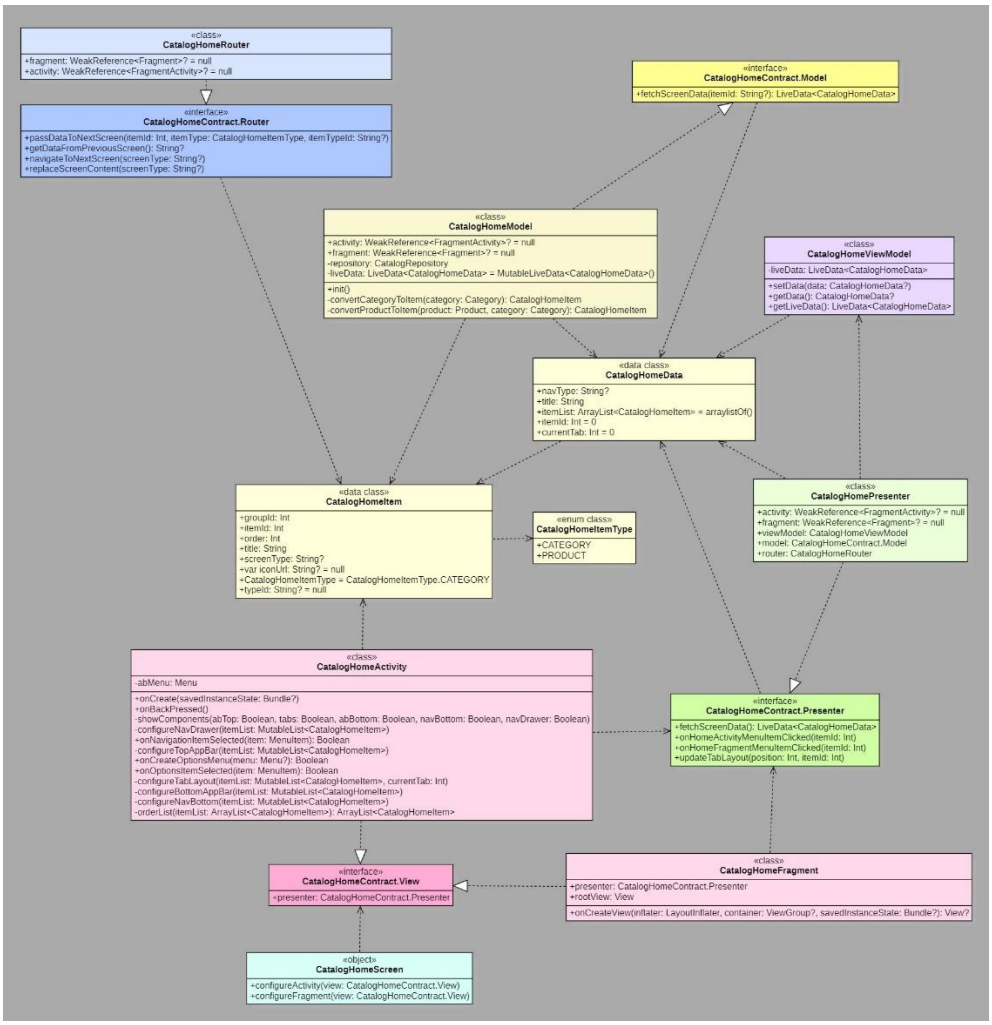

Figura 27. Diagrama UML de las clases que componen el controlador *Home*

Este apartado lo constituyen todos los controladores de pantalla, interfaces y clases disponibles para la aplicación. En la Figura 27 se muestra a modo de ejemplo cómo queda el diagrama UML de cada una de las clases que compone el controlador *Home*, para el resto de controladores la estructura el prácticamente la misma.

3.1. Clases generales

Con respecto a la logística del *framework*, encontramos clases generales que son empleadas por múltiples controladores.

3.1.1. CatalogMediator

```
class CatalogMediator: Application()
```

Esta clase se encarga del paso de estado entre los diferentes controladores.

Resumen

Variables	
String?	categoryId Identificador de una categoría.
String?	productId Identificador de un producto.

Variables

categoryId

```
var categoryId: String? = null
```

Representa el identificador de una categoría.

Valor inicial: nulo.

productId

```
var productId: String? = null
```

Representa el identificador de un producto.

Valor inicial: nulo.

3.1.2. ViewSelector

```
object ViewSelector
```

Esta clase se encarga de crear objetos de mensajería y fragmentos.

Resumen

Variables	
Intent[32]?	intent Objeto de mensajería entre componentes de una aplicación.
Fragment[33]?	fragment Fragmento.

Métodos públicos	
Intent	getNextActivity (context: Context, screenType: String?) Crea un objeto de mensajería entre componentes.
Fragment	getNextFragment (screenType: String?) Crea un fragmento.

Variables

intent

```
private var intent: Intent? = null
```

Representa un objeto de mensajería que describe una actividad a lanzar y contiene los datos necesarios para ello.

Valor inicial: nulo.

fragment

```
private var fragment: Fragment? = null
```

Representa un fragmento.

Valor inicial: nulo.

Métodos públicos

getNextActivity

```
fun getNextActivity (context: Context, screenType: String?)
```

Dependiendo del nombre de pantalla obtenido por parámetro crea el objeto de mensajería correspondiente con los datos necesarios para lanzar una actividad.

Parámetros	
context	Context: contexto actual de la aplicación.
screenType	String: nombre identificativo de una pantalla.

Retorno	
Intent	Objeto de mensajería con los datos necesarios para lanzar una actividad.

getNextFragment

```
fun getNextFragment (screenType: String?)
```

Dependiendo del nombre de pantalla obtenido por parámetro crea el fragmento correspondiente.

Parámetros	
screenType	String: nombre identificativo de una pantalla.

Retorno

Fragment	Fragmento correspondiente a un controlador.

3.1.3. ACTIVITYTITLESELECTOR

```
object ActivityTitleSelector
```

Esta clase se encarga de modificar el título mostrado en la barra de acciones de una actividad.

Resumen

Métodos públicos	
void	showTitle (activity: FragmentActivity, title: String?) Muestra el título de una actividad o no, o lo cambia.

Métodos públicos

showTitle

```
fun showTitle (activity: FragmentActivity, title: String?)
```

Dependiendo del nombre de la actividad obtenida por parámetro configura el título de una forma u otra. Para el caso del controlador *CatalogHome,* cuando no se emplea la navegación lateral se emplea el nombre de la categoría raíz, es decir, el nombre de la aplicación; en el caso de que si se emplee el título corresponde con el nombre de la categoría o producto mostrado en el fragmento. En el caso del controlador *CatalogDetail* no muestra título y para el resto de los controladores muestra el nombre correspondiente a la categoría o producto mostrados.

Parámetros	
activity	FragmentActivity: actividad actual.
screenType	String: nombre identificativo de una pantalla.

3.1.4. FilesCreator

```
object FilesCreator
```

Esta clase se encarga de almacenar en el dispositivo ficheros de imágenes a partir de URLs y devolver sus URIs a ese fichero recién creado.

Resumen

Métodos públicos	
Uri[34]	createImageFile (context: Context, name: String, imageurl: String) Crea un archivo imagen en el dispositivo y devuelve su URI (Uniform Resource Identifier)[35].
ArrayList[36]<Uri>	createImageFiles (context: Context, itemId: String, imageUrlList: ArrayList<String>) Crea varios archivos imagen en el dispositivo y devuelve una lista de URIs.

Métodos públicos

createImageFile

```
fun createImageFile (context: Context, name: String, imageUrl: String)
```

Crea un directorio donde guardar la imagen si no está ya creado. Posteriormente, crea el archivo de tipo PNG (Portable Network Graphics)[37] en el directorio especificado con el nombre pasado por parámetro. Si el archivo efectivamente se ha creado, obtiene la imagen a partir de la URL y la guarda dentro del archivo. Luego con el FileProvider obtiene la URI del archivo.

Parámetros	
context	Context: contexto actual de la actividad.
name	String: nombre identificativo del archivo.

imageurl	String: URL de la imagen.

Retorno	
Uri	Identificador del archivo.

createImageFiles

```
fun createImageFiles (context: Context, itemId: String, imageUrlList: ArrayList<String>)
```

Crea un array vacío que contendrá las URIs de las imágenes en el dispositivo, crea un directorio donde guardar las imágenes si no está ya creado. Posteriormente, crea un archivo de tipo PNG (Portable Network Graphics)[37] en el directorio especificado con el identificador de la categoría o producto al que pertenece y su posición en la lista por cada uno de los elementos en la lista. Si el archivo efectivamente se ha creado, obtiene la imagen a partir de la URL y la guarda dentro del archivo. Luego con el FileProvider obtiene la URI del archivo y la almacena en la lista de URIs.

Parámetros	
context	Context: contexto actual de la actividad.
itemId	String: nombre identificativo del archivo.
imageUrlList	ArrayList<String>: lista de URLs de imágenes.

Retorno	
ArrayList<Uri>	Lista con los identificadores de los diferentes archivos.

3.2. Controladores de pantalla

A continuación, se especificarán en detalle cada una de las clases implicadas en cada controlador de pantalla.

3.2.1. CatalogSplash

Este controlador de pantalla es el primero que se ejecuta y su función consiste en cargar los datos contenidos en un archivo JSON y obtener la categoría raíz de la aplicación navegando al controlador que le corresponde.

CatalogSplashViewModel

```
class CatalogSplashViewModel: ViewModel()
```

Esta clase se encarga conservar el estado de la pantalla CatalogSplash.

Resumen

Variables	
MutableLiveData<CatalogSplashData>	liveData Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
void	setData(data: CatalogSplashData?) Modifica el valor de la única variable de la clase.
CatalogSplashData?	getData() Obtiene el valor de la única variable de la clase.
LiveData[38]<CatalogSplashData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogSplashData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData(data: CatalogSplashData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogSplashData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogSplashData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogSplashData>	Única variable de la clase.

CATALOGSPLASHCONTRACT

```
interface CatalogSplashContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en el controlador *CatalogSplash*.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.
Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	presenter Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
void	fetchScreenData ()

	Método que busca los datos necesarios para la configuración de la pantalla.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogSplashData>	fetchScreenData(): LiveData<CatalogSplashData> Método que obtiene los datos necesarios para la configuración de la pantalla.

Router

```
interface Router
```

Interfaz que define las variables y métodos del router.

Métodos	
void	passDataToNextScreen (itemId: String) Método que se encarga de pasar los datos obtenidos por parámetro al siguiente controlador.
void	navigateToNextScreen (screenType: String?) Método que se encarga de cambiar al siguiente controlador.

CATALOGSPLASHSCREEN

```
object CatalogSplashScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla CatalogSplash.

Resumen

Métodos públicos	
`void`	`configureActivity (view: CatalogSplashContract.View)` Configura e inyecta las dependencias siendo la vista una FragmentActivity.
`void`	`configureFragment (view: CatalogSplashContract.View)` Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogSplashContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogSplashContract.View`: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogSplashContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogSplashContract.View`: vista que se muestra.

CATALOGSPLASHROUTER

```
class CatalogSplashRouter: CatalogSplashContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla CatalogSplash.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.

Métodos heredados	
De la interfaz CatalogSplashContract.Router	
void	passDataToNextScreen (itemId: String) Incorpora en el mediador información para el siguiente controlador.
void	navigateToNextScreen (screenType: String?) Inicia el siguiente controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

Métodos heredados

passDataToNextScreen

```
override fun passDataToNextScreen (itemId: String)
```

Si puede obtener el mediador a partir del fragmento, reescribe el identificador de categoría con el obtenido por parámetro.

Parámetros	
itemId	String: identificador de la categoría raíz.

navigateToNextScreen

```
override fun navigateToNextScreen (screenType: String?)
```

Si puede obtener el contexto de la actividad obtiene una instancia del controlador CatalogHome y la inicializa, terminando la actividad actual, sólo si el dato obtenido por parámetro corresponde con “home”.

Parámetros	
screentype	String: nombre de controlador

CATALOGSPLASHDATA

```
data class CatalogSplashData
```

Esta es una clase de datos contiene la información que configurará la vista.

Variables	
String	itemId Identificador de categoría u objeto.
String	screenType Nombre de controlador de pantalla.

CATALOGSPLASHMODEL

```
class CatalogSplashModel(): CatalogSplashContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador CatalogSplash.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogSplashData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogSplashModel() Construye un objeto de la clase CatalogSplashModel.

Métodos heredados	
De la interfaz CatalogSplashContract.Model	
LiveData<CatalogSplashData>	fetchScreenData () Carga los datos de la aplicación y obtiene los necesarios para el controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogSplashData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogSplashModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Carga el repositorio con los datos de la aplicación obtenidos de un JSON y si puede obtener la categoría raíz de la aplicación, obtiene de su producto principal su identificador y el tipo de controlador en el que va a ser mostrado.

Retorno	
LiveData<CatalogSplashData>	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

CATALOGSPLASHPRESENTER

```
class CatalogSplashPresenter (): CatalogSplashContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador CatalogSplash.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogSplashViewModel	viewModel Objeto que mantiene el estado de la pantalla.
CatalogSplashContract.Model	model Objeto del que se obtienen los datos para la configuración de la pantalla.

CatalogSplashContract.Router	router Objeto que re encarga de la navegación entre pantallas.

Métodos heredados	
De la interfaz CatalogSplashContract.Presenter	
LiveData<CatalogSplashData>	fetchScreenData () Obtiene los necesarios para el controlador del modelo asociado.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogSplashViewModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogSplashContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogSplashContract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener el contexto de la aplicación, le pide al modelo los datos necesarios para la navegación al primer controlador de la aplicación.

CATALOGSPLASHACTIVITY

```
class CatalogSplashActivity: AppCompatActivity(), CatalogSplashContract.View
```

Esta clase se dela actualización y gestión de interfaz de usuario del controlador *CatalogSplash*.

Resumen

Variables heredadas	
De la interfaz CatalogSplashContract.View	
CatalogSplashContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Activity[39] a través de androidx.appcompat.app.AppCompatActivity[40]	
void	onCreate (savedInstanceState: Bundle[41]?) Crea la vista.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogSplashContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el layout correspondiente a la actividad, carga el fragmento correspondiente al controlador y establece los efectos de transición entre pantallas.

Parámetros	
savedInstanceState	Bundle: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

CATALOGSPLASHFRAGMENT

```
class CatalogSplashFragment: Fragment(), CatalogSplashContract.View
```

Esta clase se encarga comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador *CatalogSplash*.

Resumen

Variables	
View	rootView Interfaz de usuario que implementa el fragmento.

Variables heredadas	
De la interfaz CatalogSplashContract.View	
CatalogSplashContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogSplashContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el layout correspondiente al fragmento, pedir al screen que configure el fragmento y pedirle al presentador que busque los datos necesarios para el controlador.

Parámetros	
`inflater`	`LayoutInflater`: objeto que infla cualquier vista en el fragmento.
`container`	`ViewGroup`: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
`savedInstanceState`	`Bundle`: estado anteriormente guardado del fragmento.

Retorno	
`View`?	Vista correspondiente a la interfaz de usuario del fragmento.

3.2.2. CATALOGHOME

Este controlador de pantalla es el encargado de los componentes de la navegación principal de la aplicación.

CATALOGHOMEVIEWMODEL

```
class CatalogHomeViewModel: ViewModel()
```

Esta clase se encarga conservar el estado de la pantalla *CatalogHome*.

Resumen

Variables	
`MutableLiveData<CatalogHomeData>`	`liveData` Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
void	setData (data: CatalogHomeData?) Modifica el valor de la única variable de la clase.
CatalogHomeData?	getData() Obtiene el valor de la única variable de la clase.
LiveData<CatalogHomeData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogHomeData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData (data: CatalogHomeData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogHomeData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData ()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogHomeData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData ()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogHomeData>	Única variable de la clase.

CATALOGHOMECONTRACT

```
interface CatalogHomeContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en controlador CatalogHome.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.
Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	presenter Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
void	fetchScreenData () Método que busca los datos necesarios para la configuración de la pantalla.
void	onHomeActivityMenuItemClicked (itemId: Int) Método que se encarga de navegar a otro controlador.
void	onHomeFragmentMenuItemClicked (itemId: Int) Método que se encarga de reemplazar un fragmento por otro de la interfaz de usuario.
void	updateTabLayout (position: Int, itemId: Int) Método que se encarga de actualizar el menú correspondiente a la navegación de tablas.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogHomeData>	fetchScreenData (itemId: String?)

	Método que obtiene los datos necesarios para la configuración de la pantalla.

Router

```
interface Router
```

Interfaz que define las variables y métodos del router.

Métodos	
`void`	`passDataToNextScreen (itemId: Int, itemType: CatalogHomeItemType, itemTypeId: String?)` Método que se encarga de pasar los datos obtenidos por parámetro al siguiente controlador.
`void`	`getDataFromPreviousScreen ()` Método que se encarga de obtener los datos pasados por el anterior controlador.
`void`	`navigateToNextScreen (screenType: String?)` Método que se encarga de cambiar al siguiente controlador.
`void`	`replaceScreenContent (screenType: String?)` Método que se encarga de cambiar el fragmento existente en el controlador por otro.

CATALOGHOMESCREEN

```
object CatalogHomeScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla CatalogHome.

Resumen

Métodos públicos	
`void`	`configureActivity(view: CatalogHomeContract.View)` Configura e inyecta las dependencias siendo la vista una FragmentActivity.
`void`	`configureFragment(view: CatalogHomeContract.View)` Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogHomeContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogHomeContract.View`: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogHomeContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogHomeContract.View`: vista que se muestra.

CATALOGHOMEROUTER

```
class CatalogHomeRouter: CatalogHomeContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla CatalogHome.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.

Métodos heredados	
De la interfaz CatalogHomeContract.Router	
void	getDataFromPreviousScreen () Obtiene el identificador de la categoría de la cual se van a extraer los datos.
void	passDataToNextScreen (itemId: Int, itemType: CatalogHomeItemType, itemTypeId: String?) Incorpora en el mediador información para el siguiente controlador.
void	navigateToNextScreen (screenType: String?) Inicia el siguiente controlador.
void	replaceScreenContent (screenType: String?) Reemplaza el fragmento actual del controlador por otro.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

Métodos heredados

getDataFromPreviousScreen

```
override fun getDataFromPreviousScreen ()
```

Si puede obtener mediador a partir de la actividad, obtiene el valor almacenado en la variable del identificador de categoría y reinicia todos los parámetros del mediador.

Retorno	
String?	Identificador de una categoría.

passDataToNextScreen

```
override fun passDataToNextScreen (itemId: Int, itemType: CatalogHomeItemType, itemTypeId: String?)
```

Si puede obtener el mediador a partir de la actividad, dependiendo de si recibe por parámetro una categoría o un producto almacena los identificadores pasados por parámetros en las correspondientes variables del mediador.

Parámetros	
itemId	Int: Identificador de una categoría o un producto.
itemType	CatalogHomeItemType: Identifica si es una categoría o un producto.
itemTypeId	String?: Identificador de una categoría.

navigateToNextScreen

```
override fun navigateToNextScreen (screenType: String?)
```

Si puede obtener el contexto de la actividad, obtiene una instancia de controlador correspondiente al nombre pasado por parámetro y la inicializa, terminando la actividad actual.

Parámetros	
screentype	String: nombre de controlador.

replaceScreenContent

```
override fun replaceScreenContent (screenType: String?)
```

Si puede obtener una instancia de fragmento de controlador correspondiente al nombre pasado por parámetro y puede obtener el gestor de fragmentos de la actividad reemplaza el fragmento existente por el nuevo.

Parámetros	
screentype	String: nombre de controlador.

CatalogHomeItemType

Esta clase es un tipo de datos especial que indica que la variable de tipo *CatalogHomeItemType* debe ser igual a uno de los valores predefinidos, *CATEGORY* o *PRODUCT*. Indica si los datos obtenidos pertenecen a una categoría o un producto.

CatalogHomeData

```
data class CatalogHomeData
```

Esta es una clase de datos que contiene la información que configura la vista.

Variables	
String?	navType Nombre de la estructura de navegación que tendrá el controlador. Valores actuales que puede tomar:

	– abTop – tabs – abBottom – navBottom – navDrawer – abTopNavBottom
String	`title` Nombre de la categoría de la cual obtenemos los datos.
ArrayList<CatalogHomeItem>	`itemList` Lista de datos de las categorías o productos hijas de una categoría.
Int	`itemId` Identificador de una categoría.
Int	`currentTab` Posición de la *tab* actual que se muestra en la vista.

CATALOGHOMEITEM

```
data class CatalogHomeItem
```

Esta es una clase de datos que contiene la información que configura la vista además de datos para navegación.

Variables	
Int	`grupoId` Grupo al que pertenece la categoría a producto.
Int	`itemId` Identificador de la categoría o producto.

Int	order Orden de aparición.
String	title Nombre de la categoría o producto.
String?	screenType Nombre del controlador de pantalla en el que se muestra. Valores actuales que puede tomar: – list – gallery – imageViewer – html – map – detail
String?	iconUrl URL de un icono representativo de la categoría o producto. Valor inicial: nulo.
CatalogHomeItemType	type Indica si es un producto o una categoría. Valor inicial: categoría.
String?	typeId Identificador de una categoría. Valor inicial: nulo

CATALOGHOMEMODEL

```
class CatalogHomeModel (): CatalogHomeContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador CatalogHome.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogHomeData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogHomeModel() Construye un objeto de la clase CatalogHomeModel.

Métodos heredados	
De la interfaz CatalogHomeContract.Model	
LiveData<CatalogHomeData>	fetchScreenData() Carga los datos de la aplicación y obtiene los necesarios para el controlador.

Métodos privados	
CatalogHomeItem	convertCategoryToItem (category: Category) Convierte un objeto de tipo categoría a un elemento del controlador.
CatalogHomeItem	convertProductToItem (product: Product, category: Category) Convierte un objeto de tipo producto a un elemento del controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogHomeData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogHomeModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si obtiene un identificador por parámetro obtiene la categoría completa y su producto principal. Asimismo, obtiene las categorías y productos asociados.

Retorno	
LiveData<CatalogHomeData>	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

Métodos privados

convertCategoryToItem

```
private fun convertCategoryToItem (category: Category)
```

Obtiene el producto principal de la categoría pasada por parámetro y con los datos de la categoría y su producto crea un elemento del controlador. El grupo asignado para los elementos obtenidos de categorías es uno.

Retorno	
CatalogHomeItem	Elemento del controlador obtenido de una categoría.

convertProductToItem

```
private fun convertProductToItem (product: Product, category: Category)
```

Del producto pasado por parámetro y la categoría crea un elemento del controlador. El grupo asignado para los elementos obtenidos de productos es dos.

Retorno	
CatalogHomeItem	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

CATALOGHOMEPRESENTER

```
class CatalogHomePresenter (): CatalogHomeContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador CatalogHome.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogHomeViewModel	viewModel Objeto que mantiene el estado de la pantalla.
CatalogHomeContract.Model	model Objeto del que se obtienen los datos para la configuración de la pantalla.
CatalogHomeContract.Router	router

	Objeto que se encarga de la navegación entre pantallas.

Métodos heredados	
De la interfaz CatalogHomeContract.Presenter	
LiveData<CatalogHomeData>	fetchScreenData() Obtiene los datos necesarios para la configuración de la vista.
void	onHomeActivityMenuItemClicked (itemId: Int) Navega a otro controlador.
void	onHomeFragmentMenuItemClicked (itemId: Int) Reemplaza el fragmento actual de la vista por otro.
void	updateTabLayout (position: Int, itemId: Int) Actualiza los valores de del menú de *tabs*.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogHomeViewModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogHomeContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogHomeontract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener los datos del *ViewModel* retorna esos datos, por el contrario, si puede obtener el contexto de la aplicación, le pide al modelo los datos necesarios para la configuración de la vista del controlador.

Retorno	
LiveData<CatalogHomeData>	Variable de la clase que contiene los datos necesarios para la configuración de la vista.

onHomeActivityMenuItemClicked

```
override fun onHomeActivityMenuItemClicked(itemId: Int)
```

Si puede obtener el elemento correspondiente al identificador pasado por parámetro de los datos almacenados en el *ViewModel*, le pide al router que actualice los datos del mediador y navegue a el siguiente controlador de pantalla.

onHomeFragmentMenuItemClicked

```
override fun onHomeFragmentMenuItemClicked(itemId: Int)
```

Si puede obtener el elemento correspondiente al identificador pasado por parámetro de los datos almacenados en el *ViewModel*, le pide al router que actualice los datos del mediador y navegue a el siguiente controlador de pantalla.

updateTabLayout

```
override fun updateTabLayout(position: Int, itemId: Int)
```

Actualiza la variable actual del menú de pestañas en el *ViewModel* y procede con la navegación.

CATALOGHOMEACTIVITY

```
class CatalogHomeActivity: AppCompatActivity(), CatalogHomeContract.View,
NavigationView.OnNavigationItemSelectedListener
```

Esta clase se encarga de la detección y notificación de eventos, así como la actualización y gestión de interfaz de usuario del controlador CatalogHome.

Resumen

Variables	
Menu	abMenu Menú correspondiente a las barras de la aplicación.

Variables heredadas	
De la interfaz CatalogHomeContract.View	
CatalogHomeContract.Presenter	presenter Presentador asociado a la vista.

Métodos privados	
void	showComponents (abTop: Boolean, tabs: Boolean, abBottom: Boolean, navBottom: Boolean, navDrawer: Boolean) Muestra los distintos tipos de componentes de navegación.
void	configureNavDrawer (itemList: MutableList[42]<CatalogHomeItem>)

	Configura el cajón de navegación.
void	configureTopAppBar (itemList: MutableList<CatalogHomeItem>) Configura la barra de aplicación superior.
void	configureTabLayout (itemList: MutableList<CatalogHomeItem>, currentTab: Int) Configura el menú de pestañas.
void	configureBottomAppBar (itemList: MutableList<CatalogHomeItem>) Configura la barra de aplicación inferior.
void	configureNavBottom (itemList: MutableList<CatalogHomeItem>) Configura el menú de navegación inferior.
void	orderList (itemList: ArrayList<CatalogHomeItem>) Reordena los elementos de la lista según su orden de aparición en la aplicación.

Métodos heredados	
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreate(savedInstanceState: Bundle?) Crea la vista.
void	onBackPressed () Actúa como la tecla para ir para atrás.
void	onCreateOptionsMenu (menu: Menu?) Inicializa el contenido del menú de opciones estándar de la Actividad.
Boolean	onOptionsItemSelected (item: MenuItem) Se llama cuando un elemento de la barra de acciones es pulsado para gestionar el evento.

De la interfaz NavigationView.OnNavigationItemSelectedListener	
Boolean	onNavigationItemSelected (item: MenuItem) Se llama cuando se selecciona un elemento en el menú de navegación.

Variables

abMenu

```
private lateinit var abMenu: Menu
```

Es el menú de las barras de navegación de la aplicación.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogHomeContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos privados

showComponents

```
private fun showComponents (
  abTop: Boolean,
  tabs: Boolean,
  abBottom: Boolean,
  navBottom: Boolean,
  navDrawer: Boolean
)
```

Dependiendo de si los valores pasados por parámetros son verdaderos o falsos se muestra el componente correspondiente en el *layout*.

configureNavDrawer

```
private fun configureNavDrawer (itemList: MutableList<CatalogHomeItem>)
```

Configura la navegación de cajón y le asigna a su menú los datos pasados por parámetros. Asimismo, establece el primer elemento del menú como el mostrado.

Parámetros	
itemList	MutableList<CatalogHomeItem>: lista de categorías y productos derivadas de la categoría actual.

configureTopAppBar

```
private fun configureTopAppBar (itemList: MutableList<CatalogHomeItem>)
```

Configura la barra de aplicación superior y le asigna a la variable abMenu los datos pasados por parámetros.

Parámetros	
itemList	MutableList<CatalogHomeItem>: lista de categorías y productos derivadas de la categoría actual.

configureTabLayout

```
private fun configureTabLayout (
   itemList: MutableList<CatalogHomeItem>,
   currentTab: Int
)
```

Configura la barra de pestañas y le asigna a su menú los datos pasados por parámetros. Asimismo, establece la pestaña actual con el pasado por parámetro.

Parámetros	
itemList	MutableList<CatalogHomeItem>: lista de categorías y productos derivadas de la categoría actual.
currentTab	Int: posición de la pestaña mostrada.

configureBottomAppBar

```
private fun configureBottomAppBar (itemList: MutableList<CatalogHomeItem>)
```

Configura la barra de aplicación inferior y le asigna a su menú los datos pasados por parámetros.

Parámetros

itemList	MutableList<CatalogHomeItem>: lista de categorías y productos derivadas de la categoría actual.

configureNavBottom

```
private fun configureNavBottom (itemList: MutableList<CatalogHomeItem>)
```

Configura el menú de navegación inferior y le asigna a su menú los datos pasados por parámetros.

Parámetros	
itemList	MutableList<CatalogHomeItem>: lista de categorías y productos derivadas de la categoría actual.

orderList

```
private fun configureBottomAppBar (itemList: ArrayList<CatalogHomeItem>)
```

Ordena los elementos pasados por parámetros por orden de aparición en la aplicación.

Parámetros	
itemList	ArrayList<CatalogHomeItem>: lista de categorías y productos derivadas de la categoría actual.

Retorno	
ArrayList<CatalogHomeItem>	Lista de categorías y productos derivadas de la categoría actual ordenadas.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el *layout* correspondiente a la actividad, carga el fragmento correspondiente al controlador, le pide al *screen* que configure la actividad, inicializa la variable abMenu y pide al presentador los datos de configuración de la vista. Una vez obtenidos

datos del presentador, dependiendo del tipo de navegación, muestra los componentes correspondientes, los configura y establece el elemento mostrado.

Parámetros	
savedInstanceState	Bundle: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

onBackPressed

```
override fun onBackPressed ()
```

Si el cajón de navegación está abierto lo cierra, si no navega al controlador de pantalla anterior.

onCreateOptionsMenu

```
override fun onCreateOptionsMenu (menu: Menu?)
```

Si el menú de la barra de aplicación superior no es nulo y la variable abMenu tiene elementos, los añade al menú de la barra de aplicación superior.

Parámetros	
menu	Menu: menú de opciones en el que coloca los elementos.

Retorno	
Boolean	Debe devolver verdadero para que se muestre el menú; si devuelve falso, no se mostrará.

onOptionsItemSelected

```
override fun onOptionsItemSelected (item: MenuItem)
```

Le pide al presentador que navegue a la categoría o producto correspondiente al elemento pulsado.

Parámetros

item	MenuItem: elemento del menú de la barra superior de navegación seleccionado.

Retorno	
Boolean	Devuelve falso para permitir que continúe el procesamiento normal del menú, verdadero para consumirlo aquí.

onNavigationItemSelected

```
override fun onNavigationItemSelected (item: MenuItem)
```

Si el menú de la barra de aplicación superior no es nulo y la variable abMenu tiene elementos los añade al menú de la barra de aplicación superior.

Parámetros	
item	MenuItem: elemento seleccionado.

Retorno	
Boolean	Verdadero para mostrar el elemento como el elemento seleccionado.

CatalogHomeFragment

```
class CatalogHomeFragment: Fragment(), CatalogHomeContract.View
```

Esta clase se encarga del comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador *CatalogHome*.

Resumen

Variables	
View	rootView

	Interfaz de usuario que implementa el fragmento.

Variables heredadas	
De la interfaz CatalogHomeContract.View	
CatalogHomeContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogHomeContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el layout correspondiente al fragmento.

Parámetros	
`inflater`	`LayoutInflater`: objeto que infla cualquier vista en el fragmento.
`container`	`ViewGroup`: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
`savedInstanceState`	`Bundle`: estado anteriormente guardado del fragmento.

Retorno	
`View`?	Vista correspondiente a la interfaz de usuario del fragmento.

3.2.3. CatalogList

Este controlador de pantalla se encarga de mostrar los datos de la base de datos en forma de lista, tanto básica como expandible.

CatalogListViewModel

```
class CatalogListViewModel: ViewModel()
```

Esta clase se encarga conservar el estado de la pantalla CatalogList.

Resumen

Variables	
`MutableLiveData<CatalogListData>`	`liveData` Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
void	setData (data: CatalogListData?) Modifica el valor de la única variable de la clase.
CatalogListData?	getData() Obtiene el valor de la única variable de la clase.
LiveData<CatalogListData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogListData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData (data: CatalogListData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogListData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData ()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogListData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData ()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogListData>	Única variable de la clase.

CATALOGLISTCONTRACT

```
interface CatalogListContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en controlador CatalogList.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.
Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	presenter Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
LiveData<CatalogListData>	fetchScreenData () Método que busca los datos necesarios para la configuración de la pantalla.
void	onListItemClicked (item: CatalogListItem) Método que se encarga de navegar a otro controlador al pulsar algún elemento de la lista.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogListData>	fetchScreenData (itemId: String?, structureType: String?) Método que obtiene los datos necesarios para la configuración de la pantalla.

Router

```
interface Router
```

Interfaz que define las variables y métodos del router.

Métodos	
void	passDataToNextScreen (itemId: Int, structureType: CatalogListItemType, itemTypeId: String?) Método que se encarga de pasar los datos obtenidos por parámetro al siguiente controlador.
void	getDataFromPreviousScreen () Método que se encarga de obtener los datos pasados por el anterior controlador.
void	navigateToNextScreen (screenType: String?) Método que se encarga de cambiar al siguiente controlador.

CatalogListScreen

```
object CatalogListScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla CatalogList.

Resumen

Métodos públicos	
void	configureActivity(view: CatalogListContract.View) Configura e inyecta las dependencias siendo la vista una FragmentActivity.
void	configureFragment(view: CatalogListContract.View)

	Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogListContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogListContract.View`: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogListContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogListContract.View`: vista que se muestra.

CATALOGLISTROUTER

```
class CatalogListRouter: CatalogListContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla CatalogList.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.

Métodos heredados	
De la interfaz CatalogListContract.Router	
void	getDataFromPreviousScreen () Obtiene el identificador de la categoría o producto a mostrar y qué es.
void	passDataToNextScreen (itemId: Int, itemType: CatalogListItemType, itemTypeId: String?) Incorpora en el mediador información para el siguiente controlador.
void	navigateToNextScreen (screenType: String?) Inicia el siguiente controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

Métodos heredados

getDataFromPreviousScreen

```
override fun getDataFromPreviousScreen ()
```

Si puede obtener el mediador a partir de la actividad, obtiene el valor almacenado en la variable del identificador. Dependiendo del identificador que obtenga le asigna un tipo *product* o *category* y los incorpora en un array asociativo. Posteriormente reinicia todos los parámetros del mediador.

Retorno	
HashMap<String, String?>	Array asociativo que contiene el identificador de una categoría o producto y su tipo.

passDataToNextScreen

```
override fun passDataToNextScreen (itemId: Int, itemType: CatalogListItemType, itemTypeId: String?)
```

Si puede obtener la actividad de fragmento y de esta el mediador, dependiendo de si recibe por parámetro una categoría o un producto almacena los identificadores pasados por parámetros en las correspondientes variables del mediador.

Parámetros	
itemId	Int: Identificador de una categoría o un producto.
itemType	CatalogListItemType: Identifica si es una categoría o un producto.
itemTypeId	String?: Identificador de una categoría.

navigateToNextScreen

```
override fun navigateToNextScreen (screenType: String?)
```

Si puede obtener la actividad del fragmento, obtiene una instancia de controlador correspondiente al nombre pasado por parámetro y la inicializa, terminando la actividad actual.

Parámetros	
screentype	String: nombre de controlador.

CATALOGLISTITEMTYPE

Esta clase es un tipo de datos especial que indica que la variable de tipo *CatalogListItemType* debe ser igual a uno de los valores predefinidos, *CATEGORY* o *PRODUCT*. Indica si los datos obtenidos pertenecen a una categoría o un producto.

CATALOGLISTDATA

```
data class CatalogListData
```

Esta es una clase de datos contiene la información que configura la vista.

Variables	
Array<String>?	componentsShown Array con los nombres de los componentes a mostrar en la vista. Valores actuales que puede tener: – title – subtitle – image – description – outstandingIcon – shareButton
Boolean	expandable Indica si la lista es expandible o no.
String	itemId Identificador de un producto o una categoría.
String	title Nombre del producto o la categoría de la cual obtenemos los datos.

ArrayList<CatalogListItem>	itemList Lista de datos de las categorías o productos hijos del cual estamos obteniendo los datos.

CATALOGLISTITEM

```
data class CatalogListItem
```

Esta es una clase de datos que contiene la información que configura la vista además de datos para navegación.

Variables	
String	itemId Identificador de un producto o una categoría.
String	title Nombre de categoría o producto.
String?	subtitle Subtítulo de categoría o producto. Valor inicial: nulo.
String?	description Descripción de categoría o producto. Valor inicial: nulo.
CatalogListItemType?	type Indica si es un producto, una categoría o un elemento simple. Valor inicial: nulo.
String?	typeId

	Identificador de una categoría. Valor inicial: nulo
ArrayList<String>	imageUrls Lista de URLs asociadas a la categoría o producto.
String?	itemType Nombre del controlador de pantalla en el que se muestra. Valores actuales que puede tomar: – list – gallery – imageViewer – html – map – detail
Boolean	outstanding Indica si el producto o la categoría pertenece a destacados o no.

CATALOGLISTMODEL

```
class CatalogListModel (): CatalogListContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador CatalogList.

Resumen

Variables	
WeakReference<Fragment>?	fragment

	Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogListData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogListModel() Construye un objeto de la clase *CatalogListModel*.

Métodos heredados	
De la interfaz CatalogListContract.Model	
LiveData<CatalogListData>	fetchScreenData(itemId: String?, structureType: String?) Obtiene los datos necesarios para el controlador.

Métodos privados	
CatalogListItem	convertCategoryToItem (category: Category) Convierte un objeto de tipo categoría a un elemento del controlador.
CatalogListItem	convertProductToItem (product: Product, category: Category) Convierte un objeto de tipo producto a un elemento del controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogListData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogListModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData (itemId: String?, structureType: String?)
```

Si obtiene un identificador por parámetro, actúa dependiendo de su tipo de una forma u otra.

- Para un producto obtiene sus datos, crea los datos de la clase básicos en los que sólo se muestra el título, no se da posibilidad de expandir ni añadir imágenes. La lista de elementos la obtiene del parámetro *itemList,* el cual es una cadena de texto con los elementos separados por coma y tiene la siguiente estructura en el JSON:

  'param_itemlist@@:@@[string 1],[string 2],[string 3]'

- Para una categoría, obtiene sus datos y su producto principal. De estos, crea los datos de la clase y para obtener la lista de elementos de la clase obtiene las categorías y productos asociados al actual.

Parámetros	
`itemId`	`String`: Identificador de una categoría o un producto.
`structureType`	`String`: Indica si el identificador pertenece a un producto o una categoría.

Retorno	
`LiveData<CatalogListData>`	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

Métodos privados

convertCategoryToItem

```
private fun convertCategoryToItem (category: Category)
```

Obtiene el producto principal de la categoría pasada por parámetro y con los datos de la categoría y su producto crea un elemento del controlador. Por defecto el subtítulo será la fecha de modificación.

Retorno	
`CatalogListItem`	Elemento del controlador obtenido de una categoría.

convertProductToItem

```
private fun convertProductToItem (product: Product, category: Category)
```

Del producto pasado por parámetro y la categoría crea un elemento del controlador. Por defecto el subtítulo será la fecha de modificación.

Retorno	
CatalogListItem	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

CATALOGLISTPRESENTER

```
class CatalogListPresenter (): CatalogListContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador CatalogList.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogListViewModel	viewModel Objeto que mantiene el estado de la pantalla.
CatalogListContract.Model	model Objeto del que se obtienen los datos para la configuración de la pantalla.
CatalogListContract.Router	router Objeto que se encarga de la navegación.

Métodos heredados	
De la interfaz CatalogListContract.Presenter	
LiveData<CatalogListData>	fetchScreenData() Obtiene los datos necesarios para la configuración de la vista.
void	onListItemClicked (item: CatalogListItem) Método que se encarga de navegar a otro controlador al pulsar algún elemento de la lista.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogListViewModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogListContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogListContract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener los datos del *ViewModel* retorna esos datos, por el contrario, si puede obtener el contexto de la aplicación, le pide al router los datos del mediador y luego le pide al modelo los datos necesarios para la configuración de la vista del controlador pasando por parámetro los datos obtenidos del router.

Retorno	
LiveData<CatalogListData>	Variable de la clase que contiene los datos necesarios para la configuración de la vista.

onListItemClicked

```
override fun onListItemClicked (item: CatalogListItem)
```

Si puede obtener el nombre del controlador correspondiente al elemento pasado por

parámetro, le pide al router que actualice los datos del mediador y navegue a el siguiente controlador de pantalla.

Parámetros	
item	CatalogListItem: Elemento pulsado de la lista.

CatalogListActivity

```
class CatalogListActivity: AppCompatActivity(), CatalogListContract.View
```

Esta clase se encarga de la detección y notificación de eventos, así como la actualización y gestión de interfaz de usuario del controlador CatalogList.

Resumen

Variables heredadas	
De la interfaz CatalogListContract.View	
CatalogListContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreate(savedInstanceState: Bundle?) Crea la vista.
De androidx.appcompat.app.AppCompatActivity	
void	onSupportNavigateUp () Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogListContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el layout correspondiente a la actividad, carga el fragmento correspondiente al controlador, establece los efectos de transición entre pantallas. Y si hay, configura la barra de acciones para que muestre la navegación hacia atrás.

Parámetros	
`savedInstanceState`	`Bundle`: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

onSupportNavigateUp

```
override fun onSupportNavigateUp ()
```

Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones haciendo uso del método *onBackPressed* heredado.

Retorno	
`Boolean`	Es verdadero si la navegación hacia arriba se completó correctamente y esta actividad se completó, falso en caso contrario.

CHILD

```
data class Child
```

Esta es una clase de datos contiene la información localizada en la vista expandida de la lista.

Variables	
`String?`	`subtitle` Subtítulo de la categoría o producto. Valor inicial: nulo.
`String?`	`description` Descripción de la categoría o producto.

	Valor inicial: nulo.
Boolean	outstanding Indica si el producto o la categoría perteneces a destacados.
String?	imageUrl Imagen representativa de la categoría o producto Valor inicial: nulo.

PARENT

```
data class Parent (val item: CatalogListItem):
ExpandableRecyclerViewAdapter.ExpandableGroup[43]<Child> ()
```

Esta es una clase de datos contiene la información localizada en la vista que se puede expandir de la lista.

Variables	
CatalogListItem	item Elemento del controlador.

Métodos heredados	
De ExpandableRecyclerViewAdapter.ExpandableGroup<Child>	
List<Child>	getExpandingItems () Crea una lista con los elementos que son parte la vista expandida de la lista.

Variables

item

```
val item: CatalogListItem
```

Representa el elemento del controlador.

Métodos heredados

getExpandingItems

```
override fun getExpandingItems ()
```

Crea una lista con los elementos que pertenecen a la vista expandida. Sólo hay un elemento que se expande dentro de la lista y contendría el subtítulo, descripción y la indicación de si es destacado o no un elemento.

Retorno	
List<Child>	Lista con los elementos de la vista expandida.

CatalogListRecycleAdapter

```
 class CatalogListRecycleAdapter (
  private val presenter: CatalogListContract.Presenter,
  private val componentsShown: Array<String>,
  private val expandable: Boolean,
  private var parents: ArrayList<Parent>
) : ExpandableRecyclerViewAdapter<Child, Parent,
CatalogListRecycleAdapter.PViewHolder, CatalogListRecycleAdapter.CViewHolder>(
  parents, ExpandingDirection.VERTICAL
)
```

Esta clase se encarga de adaptar los datos pasados por parámetros a unas vistas predeterminadas.

Resumen

Variables	
CatalogListContract.Presenter	presenter Presentador asociado a la vista.
Array<String>	componentsShown

	Array con los nombres de los componentes a mostrar en la vista.
Boolean	expandable Indica si la lista es expandible o no.
ArrayList<Parent>	parents Lista de elementos que compondrán la lista.
ArrayList<Parent>	parentsCopy Copia de la lista de elementos que compondrán la lista.

Constructores
CatalogListRecycleAdapter () Construye un objeto de la clase CatalogListRecycleAdapter.

Clases internas	
PViewHolder	Clase encargada de la vista expansible.
CViewHolder	Clase encargada de la vista expandida.

Métodos privados	
void	organizeParentComponents (parentViewHolder: PViewHolder, expandableType: Parent, componentsShown: Array<String>) Organiza los elementos de la vista expansible.
void	organizeChildComponents (childViewHolder: CViewHolder, expandedType: Child, componentsShown: Array<String>)

	Organiza los elementos de la vista expandida.
void	outstandingsFilter (outstandings: Boolean) Filtra los elementos de la lista en función de si son destacados o no.

Métodos heredados	
De ExpandableRecyclerViewAdapter	
PViewHolder	onCreateParentViewHolder (parent: ViewGroup, viewType: Int) Se encarga de inflar la vista con la vista expansible.
CViewHolder	onCreateChildViewHolder (child: ViewGroup, viewType: Int) Se encarga de inflar la vista con la vista expandida.
void	onBindParentViewHolder (parentViewHolder: PViewHolder, expandableType: Parent, position: Int) Se encarga de configurar la expansible.
void	onBindChildViewHolder (childViewHolder: CViewHolder, expandedType: Child, expandableType: Parent, position: Int) Se encarga de configurar la expandida.

Variables

presenter

```
private val presenter: CatalogListContract.Presenter
```

Representa el presentador asociado a la vista.

componentsShown

```
private val componentsShown: Array<String>
```

Array con los nombres de los componentes a mostrar en la vista expandida.

Valores actuales que puede tener:

- title
- subtitle

- image
- description
- outstandingIcon
- shareButton

expandable

```
private val expandable: Boolean
```

Indica si la lista es expandible o no.

parents

```
private val parents: ArrayList<Parent>
```

Lista de elementos que compondrán la lista de elementos general.

parentsCopy

```
private val parents: ArrayList<Parent>
```

Copia de la lista de elementos que compondrán la lista de elementos general.

Valor inicial: vacío.

Constructores

CatalogListRecycleAdapter

```
init
```

Añade todos los elementos de la variable *parents* en la de *parents copy.*

Clases internas

PViewHolder

```
class PViewHolder (itemView: View):
ExpandableRecyclerViewAdapter.ExpandableViewHolder (itemView)
```

Esta clase se encarga de la vista expansible. Obtiene los componentes de la vista y los inicializa como variables.

Variables	
TextView	title

	Componente en el que se muestra el título de una categoría o producto.
ImageView	image Componente en el que se muestra una imagen representativa de una categoría o producto.
TextView	subtitle Componente en el que se muestra el subtítulo de una categoría o producto.
LinearLayout	buttonsLayout Componente en el que se muestran los botones de la vista.
ImageView	outstandingIcon Componente en el que se muestra el icono de destacados.
Button	shareButton Componente correspondiente a la acción de compartir.

CViewHolder

```
class CViewHolder (itemView: View):
ExpandableRecyclerViewAdapter.ExpandableViewHolder (itemView)
```

Esta clase se encarga de la vista expandida. Obtiene los componentes de la vista y los inicializa como variables.

Variables	
ImageView	image

	Componente en el que se muestra una imagen representativa de una categoría o producto.
TextView	subtitle Componente en el que se muestra el subtítulo de una categoría o producto.
TextView	description Componente en el que se muestra una breve descripción de una categoría o producto.
LinearLayout	buttonsLayout Componente en el que se muestran los botones de la vista.
ImageView	outstandingIcon Componente en el que se muestra el icono de destacados.
Button	shareButton Componente correspondiente a la acción de compartir.

Métodos privados

organizeParentComponents

```
private fun organizeParentComponents (parentViewHolder: PViewHolder, expandableType: Parent, componentsShown: Array<String>)
```

Revisa uno por uno los componentes de la vista y si la variable *componentsShown* los contienen, si es verdadero le asigna su valor correspondiente, por el contrario, oculta el componente.

Parámetros	
parentViewHolder	PViewHolder: vista expansible.
expandableType	Parent: elemento que contiene los datos.
componentsShown	Array<String>: lista de los nombres de los componentes a mostrar en la vista.

organizeChildComponents

```
private fun organizeChildComponents (childViewHolder: CViewHolder,
expandedType: Child, componentsShown: Array<String>)
```

Revisa uno por uno los componentes de la vista y si la variable *componentsShown* los contienen, si es verdadero le asigna su valor correspondiente, por el contrario, oculta el componente.

Parámetros	
childViewHolder	CViewHolder: vista expandida.
expandedType	Child: elemento que contiene los datos.
componentsShown	Array<String>: lista de los nombres de los componentes a mostrar en la vista.

outstandingsFilter

```
fun outstandingsFilter (outstandings: Boolean)
```

Si *outstandings* es verdadero crea una lista en la que almacena los elementos de la lista parentsCopy que pertenecen a destacados, limpia la lista parents y añade le añade todos los resultados. En caso de ser falso, limpia la lista original de elementos y le añade todos los de la copia. Por último, notifica al adaptador que los datos han cambiado.

Parámetros	
outstandings	Boolean: verdadero para mostrar los elementos destacados, falso para mostrar todos.

Métodos heredados

onCreateParentViewHolder

```
override fun onCreateParentViewHolder(parent: ViewGroup, viewType: Int)
```

Se encarga de inflar la vista con el *layout* correspondiente a la vista expansible.

Parámetros	
parent	ViewGroup
viewType	Int

Retorno	
PViewHolder	Vista expansible del elemento del controlador.

onCreateChildViewHolder

```
override fun onCreateChildViewHolder(child: ViewGroup, viewType: Int)
```

Se encarga de inflar la vista con el *layout* correspondiente a la vista expansible.

Parámetros	
child	ViewGroup
viewType	Int

Retorno	
CViewHolder	Vista expandida del elemento del controlador.

onBindParentViewHolder

```
override fun onBindParentViewHolder (parentViewHolder: PViewHolder,
expandableType: Parent, position: Int)
```

Se encarga configurar la vista expansible. Organiza los componentes y en el caso de que la lista no sea expansible comprueba si se muestra el botón de compartir para definir su OnClickListener, el cual se encargar de crear una instancia y la lanza para compartir las imagen de la categoría o producto, su título y descripción. Además, define otro OnClickListener para el que llama al presentador para que gestione el evento.

Parámetros	
parentViewHolder	PViewHolder: vista expansible.
expandableType	Parent: elemento expandible.
position	Int: posición del elemento en la lista.

onBindChildViewHolder

```
override fun onBindChildViewHolder (childViewHolder: CViewHolder, expandedType:
Child, expandableType: Parent, position: Int)
```

Se encarga configurar la vista expandida. En el caso de que la lista sea expansible organiza los componentes, comprueba si se muestra el botón de compartir para definir su OnClickListener, el cual se encargar de crear una instancia y la lanza para compartir las imagen de la categoría o producto, su título y descripción. En el caso de que la lista no sea expandible la oculta.

Parámetros	
childViewHolder	CViewHolder: vista expansible.
expandedType	Child: element expandido.
expandableType	Parent: elemento expandible.
position	Int: posición del elemento en la lista.

CATALOGLISTFRAGMENT

```
class CatalogListFragment: Fragment(), CataloglistContract.View
```

Esta clase se encarga comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador CatalogList.

Resumen

Variables	
View	rootView Interfaz de usuario que implementa el fragmento.
Boolean	showOutstandings Indica si solo se muestran los objetos destacados o todos.

Variables heredadas	
De la interfaz CataloglistContract.View	
CatalogListContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreateOptionsMenu (menu: Menu?)

	Inicializar el contenido del menú de opciones estándar de la Actividad.
void	onPrepareOptionsMenu (menu: Menu) Modifica los elementos del menú de la barra de acciones antes de mostrarlos.
Boolean	onOptionsItemSelected (item: MenuItem) Se llama cuando un elemento de la barra de acciones es pulsado para gestionar el evento.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

showOutstandings

```
private var showOutstandings = false
```

Si es verdadera se muestran en la vista sólo los elementos de destacados, de lo contrario se muestran todos los elementos.

Valor inicial: falso.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogListContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?,
savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el *layout* correspondiente al fragmento, pedir al *screen* que lo configure, habilitar el cambio de opciones a la barra de acciones y pedir los datos de configuración al presentador.

Una vez obtenidos los datos, si hay actividad muestra el título correspondiente, muestra la vista a reciclar y oculta la barra de progreso. Se establece los componentes a mostrar por defecto, tanto para expansible como para no y si se pueden obtener de los datos se sustituyen.

Asimismo, se comprueba si entre los componentes a mostrar se encuentra la opción de destacados y se incorpora en la barra de acciones. Posteriormente, se configura la lista de elementos obtenidos como elementos de la clase `Parent`, se crea y pasa el adaptador a la vista reciclada para que adapte todos los elementos y finalmente, se incorpora un separador entre elementos.

Parámetros	
`inflater`	`LayoutInflater`: objeto que infla cualquier vista en el fragmento.
`container`	`ViewGroup`: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
`savedInstanceState`	`Bundle`: estado anteriormente guardado del fragmento.

Retorno	
`View`?	Vista correspondiente a la interfaz de usuario del fragmento.

onCreateOptionsMenu

```
override fun onCreateOptionsMenu (menu: Menu, inflater: MenuInflater)
```

Incorpora en el menú de la barra de aplicación superior un menú inflandolo.

Parámetros	
`menu`	`Menu`: menú de opciones en el que coloca los elementos.
`inflater`	`MenuInflater`

onPrepareOptionsMenu

```
override fun onPrepareOptionsMenu (menu: Menu)
```

Prepara el menú de opciones estándar de la pantalla para que se muestre. Dependiendo de si la variable showOutstandings es verdadera o no, cambia el título de la opción del menú denominada *outstandings*.

Parámetros	
menu	Menu: el menú de opciones como se mostró por última vez o inicializado por primera vez por *onCreateOptionsMenu*.

onOptionsItemSelected

```
override fun onOptionsItemSelected (item: MenuItem)
```

Le pide al presentador que navegue a la categoría o producto correspondiente al elemento pulsado.

Parámetros	
item	MenuItem: elemento del menú de la barra superior de navegación seleccionado.

Retorno	
Boolean	Devuelve falso para permitir que continúe el procesamiento normal del menú, verdadero para consumirlo aquí.

3.2.4. CatalogGallery

Este controlador de pantalla se encarga de mostrar vista de tarjetas, con la posibilidad de que dichas tarjetas sean expansibles.

CatalogGalleryViewModel

```
class CatalogGalleryViewModel: ViewModel()
```

Esta clase se encarga conservar el estado de la pantalla *CatalogGallery*.

Resumen

Variables	
MutableLiveData<CatalogGalleryData>	liveData Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
void	setData (data: CatalogGalleryData?) Modifica el valor de la única variable de la clase.
CatalogGalleryData?	getData() Obtiene el valor de la única variable de la clase.
LiveData<CatalogGalleryData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogGalleryData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData (data: CatalogGalleryData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogGalleryData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData ()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogGalleryData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData ()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogGalleryData>	Única variable de la clase.

CATALOGGALLERYCONTRACT

```
interface CatalogGalleryContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en controlador *CatalogGallery*.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.

Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	presenter Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
LiveData<CatalogGalleryData>	fetchScreenData () Método que busca los datos necesarios para la configuración de la pantalla.
void	onGalleryItemClicked (item: CatalogGalleryItem) Método que se encarga de navegar a otro controlador al pulsar algún elemento de la lista de tarjetas.
void	onImageGalleryItemClicked (activity: Activity, item: HashMap<String,String>, sharedImageView: ImageView)

	Método que se encarga de navegar al controlador ImageViewer al pulsar algún elemento de la lista de tarjetas.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogGalleryData>	fetchScreenData (itemId: String?, structureType: String?) Método que obtiene los datos necesarios para la configuración de la pantalla.

Router

```
interface Router
```

Interfaz que define las variables y métodos del router.

Métodos	
void	passDataToNextScreen (itemId: Int, structureType: CatalogGalleryItemType, itemTypeId: String?) Método que se encarga de pasar los datos obtenidos por parámetro al siguiente controlador.
HashMap<String,String>	getDataFromPreviousScreen () Método que se encarga de obtener los datos pasados por el anterior controlador.
void	navigateToNextScreen (screenType: String?) Método que se encarga de cambiar al siguiente controlador.

void	navigateToImageViewer (activity: Activity, item: HashMap<String,String>, sharedImageView: ImageView) Método que se encarga de cambiar al controlador ImageViewer.

CATALOGGALLERYSCREEN

```
object CatalogGalleryScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla CatalogGallery.

Resumen

Métodos públicos	
void	configureActivity(view: CatalogGalleryContract.View) Configura e inyecta las dependencias siendo la vista una FragmentActivity.
void	configureFragment(view: CatalogGalleryContract.View) Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogGalleryContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
view	CatalogGalleryContract.View: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogGalleryContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
view	CatalogGalleryContract.View: vista que se muestra.

CATALOGGALLERYROUTER

```
class CatalogGalleryRouter: CatalogGalleryContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla *CatalogGallery*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
String	EXTRA_ITEM Nombre de un campo de información que representa la URL de una imagen.

String	EXTRA_ITEM_IMAGE_TRANSITION_NAME Nombre de un campo de información que representa el nombre de una animación de transición.

Métodos heredados	
De la interfaz CatalogGalleryContract.Router	
HashMap<String,String>	getDataFromPreviousScreen () Obtiene el identificador de la categoría o producto a mostrar y qué es.
void	passDataToNextScreen (itemId: Int, itemType: CatalogGalleryItemType, itemTypeId: String?) Incorpora en el mediador información para el siguiente controlador.
void	navigateToNextScreen (screenType: String?) Inicia el siguiente controlador.
void	navigateToImageViewer (activity: Activity, item: HashMap<String,String>, sharedImageView: ImageView) Método que se encarga de cambiar al controlador ImageViewer.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

EXTRA_ITEM

```
private val EXTRA_ITEM
```

Representa el nombre estándar de información extra pasada por un Intent, corresponde con una url de imagen.

Valor inicial: `"item_image_url"`.

EXTRA_ITEM_IMAGE_TRANSITION_NAME

```
private val EXTRA_ITEM_IMAGE_TRANSITION_NAME
```

Representa el nombre estándar de información extra pasada por un Intent, corresponde con el nombre dado a una transición.

Valor inicial: `"item_image_transition_name".`

Métodos heredados

getDataFromPreviousScreen

```
override fun getDataFromPreviousScreen ()
```

Si puede obtener el mediador a partir de la actividad, obtiene el valor almacenado en la variable del identificador. Dependiendo del identificador que obtenga le asigna un tipo *product* o *category* y los incorpora en un array asociativo. Posteriormente reinicia todos los parámetros del mediador.

Retorno	
HashMap<String, String?>	Array asociativo que contiene el identificador de una categoría o producto y su tipo.

passDataToNextScreen

```
override fun passDataToNextScreen (itemId: Int, itemType: CatalogGalleryItemType, itemTypeId: String?)
```

Si puede obtener la actividad de fragmento y de esta el mediador, dependiendo de si recibe por parámetro una categoría o un producto almacena los identificadores pasados por parámetros en las correspondientes variables del mediador.

Parámetros	
itemId	Int: Identificador de una categoría o un producto.
itemType	CatalogGalleryItemType: Identifica si es una categoría o un producto.
itemTypeId	String?: Identificador de una categoría.

navigateToNextScreen

```
override fun navigateToNextScreen (screenType: String?)
```

Si puede obtener la actividad del fragmento, obtiene una instancia de controlador correspondiente al nombre pasado por parámetro y la inicializa, terminando la actividad actual.

Parámetros	
screentype	String: nombre de controlador.

navigateToImageViewer

```
override fun (activity: Activity, item: HashMap<String,String>,
sharedImageView: ImageView)
```

Si puede obtener la actividad del fragmento, crea una instancia de controlador ImageViewer, le pasa en las opciones los datos obtenidos por parámetro y la inicializa.

Parámetros	
activity	Activity: actividad actual.
item	HashMap<String,String>: array clave-valor con la URL de una imagen y el nombre de la transición.
sharedImageView	ImageView: contenedor actual en el que se muestra la imagen.

CatalogGalleryItemType

Esta clase es un tipo de datos especial que indica que la variable de tipo *CatalogGalleryItemType* debe ser igual a uno de los valores predefinidos, *CATEGORY* o *PRODUCT*. Indica si los datos obtenidos pertenecen a una categoría o un producto.

CATALOGGALLERYDATA

```
data class CatalogGalleryData
```

Esta es una clase de datos que contiene la información que configura la vista.

Variables	
String	spanCount Número de tarjetas en una fila.
Array<String>?	componentsShown Array con los nombres de los componentes a mostrar en la vista. Valores actuales que puede tener: – title – subtitle – image – description – outstandingIcon – shareButton
Boolean	expandable Indica si las tarjetas son expandibles o no.
String	itemId Identificador de un producto o una categoría.
String	title Nombre del producto o la categoría de la cual obtenemos los datos.
ArrayList<String>	itemImageUrls Lista de imágenes pertenecientes a un producto o una categoría.

ArrayList<CatalogGalleryItem>	itemList Lista de datos de las categorías o productos hijos del cual estamos obteniendo los datos.

CATALOGGALLERYITEM

```
data class CatalogGalleryItem
```

Esta es una clase de datos que contiene la información que configura la vista además de datos para navegación.

Variables	
String	itemId Identificador de un producto o una categoría.
String	title Nombre de categoría o producto.
String?	subtitle Subtítulo de categoría o producto. Valor inicial: nulo.
String?	description Descripción de categoría o producto. Valor inicial: nulo.
CatalogGalleryItemType?	type Indica si es un producto, una categoría o un elemento simple. Valor inicial: nulo.
String?	typeId

	Identificador de una categoría. Valor inicial: nulo
ArrayList<String>	imageUrls Lista de URLs asociadas a la categoría o producto.
String?	screenType Nombre del controlador de pantalla en el que se muestra. Valores actuales que puede tomar: – list – gallery – imageViewer – html – map – detail
Boolean	outstanding Indica si el producto o la categoría pertenece a destacados o no.

CATALOGGALLERYMODEL

```
class CatalogGalleryModel (): CatalogGalleryContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador CatalogGallery.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogGalleryData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogGalleryModel() Construye un objeto de la clase *CatalogGalleryModel*.

Métodos heredados	
De la interfaz CatalogGalleryContract.Model	
LiveData<CatalogGalleryData>	fetchScreenData(itemId: String?, structureType: String?) Obtiene los datos necesarios para el controlador.

Métodos privados	
CatalogGalleryItem	convertCategoryToItem (category: Category) Convierte un objeto de tipo categoría a un elemento del controlador.

CatalogGalleryItem	convertProductToItem (product: Product, category: Category) Convierte un objeto de tipo producto a un elemento del controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogGalleryData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogGalleryModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData (itemId: String?, structureType: String?)
```

Si obtiene un identificador por parámetro, actúa dependiendo de su tipo de una forma u otra.

- Para un producto obtiene sus datos y crea los datos de la clase, con la particularidad de que *itemList* permanece vacía mientras que en *itemImageUrls* se incorporan las imágenes del producto, exceptuando la primera, que corresponde a un icono representativo.

- Para una categoría, obtiene sus datos y su producto principal. De estos, crea los datos de la clase donde *itemImageUrls* permanece vacía mientras que en *itemList* almacena la lista de elementos de la clase, y obtiene las categorías y productos asociados al actual.

Parámetros	
`itemId`	`String`: Identificador de una categoría o un producto.
`structureType`	`String`: Indica si el identificador pertenece a un producto o una categoría.

Retorno	
`LiveData<CatalogGalleryData>`	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

Métodos privados

convertCategoryToItem

```
private fun convertCategoryToItem (category: Category)
```

Obtiene el producto principal de la categoría pasada por parámetro y con los datos de la categoría y su producto crea un elemento del controlador. Por defecto el subtítulo será la fecha de modificación.

Retorno	
`CatalogGalleryItem`	Elemento del controlador obtenido de una categoría.

convertProductToItem

```
private fun convertProductToItem (product: Product, category: Category)
```

Del producto pasado por parámetro y la categoría crea un elemento del controlador. Por defecto el subtítulo será la fecha de modificación.

Retorno	
CatalogGalleryItem	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

CATALOGGALLERYPRESENTER

```
class CatalogGalleryPresenter (): CatalogGalleryContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador *CatalogGallery*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogGalleryViewModel	viewModel Objeto que mantiene el estado de la pantalla.
CatalogGalleryContract.Model	model Objeto del que se obtienen los datos para la configuración de la pantalla.

CatalogGalleryContract.Router	router Objeto que se encarga de la navegación.

Métodos heredados	
De la interfaz CatalogGalleryContract.Presenter	
LiveData<CatalogGalleryData>	fetchScreenData() Obtiene los datos necesarios para la configuración de la vista.
void	onGalleryItemClicked (item: CatalogGalleryItem) Navegar a otro controlador al pulsar algún elemento de la lista de tarjetas.
void	onImageGalleryItemClicked (activity: Activity, item: HashMap<String,String>, sharedImageView: ImageView) Método que se encarga de navegar al controlador ImageViewer al pulsar algún elemento de la lista de tarjetas.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogGalleryViewModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogGalleryContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogGalleryContract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener los datos del *ViewModel* retorna esos datos, por el contrario, si puede obtener el contexto de la aplicación, le pide al router los datos del mediador y luego le pide al modelo los datos necesarios para la configuración de la vista del controlador pasando por parámetro los datos obtenidos del router.

Retorno	
LiveData<CatalogGalleryData>	Variable de la clase que contiene los datos necesarios para la configuración de la vista.

onGalleryItemClicked

```
override fun onGalleryItemClicked (item: CatalogGalleryItem)
```

Si puede obtener el nombre del controlador correspondiente al elemento pasado por parámetro, le pide al router que actualice los datos del mediador y navegue a el siguiente controlador de pantalla.

Parámetros	
item	CatalogGalleryItem: Tarjeta pulsada de la lista.

onImageGalleryItemClicked

```
override fun onImageGalleryItemClicked (activity: Activity, item: HashMap<String,String>, sharedImageView: ImageView)
```

Le pide al router que navegue al controlador ImageViewer pasandole los datos obtenidos por parámetros.

Parámetros	
activity	Activity: Actividad actual.
item	HashMap<String,String>: Tarjeta pulsada de la lista.
sharedImageView	ImageView: Contenedor que muestra la imagen de la tarjeta.

CATALOGGALLERYACTIVITY

```
class CatalogGalleryActivity: AppCompatActivity(), CatalogGalleryContract.View
```

Esta clase se encarga de la detección y notificación de eventos, así como la actualización y gestión de interfaz de usuario del controlador *CatalogGallery*.

Resumen

Variables heredadas	
De la interfaz CatalogGalleryContract.View	
CatalogGalleryContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreate(savedInstanceState: Bundle?) Crea la vista.
De androidx.appcompat.app.AppCompatActivity	
void	onSupportNavigateUp () Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogGalleryContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el layout correspondiente a la actividad, carga el fragmento correspondiente al controlador, establece los efectos de transición entre pantallas. Y si hay, configura la barra de acciones para que muestre la navegación hacia atrás.

Parámetros	
savedInstanceState	Bundle: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

onSupportNavigateUp

```
override fun onSupportNavigateUp ()
```

Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones haciendo uso del método *onBackPressed* heredado.

Retorno	
Boolean	Es verdadero si la navegación hacia arriba se completó correctamente y esta actividad se completó, falso en caso contrario.

CatalogGalleryRecycleAdapter

```
class CatalogGalleryRecyclerAdapter(
   private val context: Context,
   private val presenter: CatalogGalleryContract.Presenter,
   private val itemId: String?,
   private val itemImageUrls: ArrayList<String>?,
   private val itemList: ArrayList<CatalogGalleryItem>?,
   private val componentsShown: Array<String>
) : RecyclerView.Adapter<CatalogGalleryRecyclerAdapter.ViewHolder>()
```

Esta clase se encarga de adaptar los datos pasados por parámetros a una vista concreta.

Resumen

Variables	
Context	context Actividad.
CatalogGalleryContract.Presenter	presenter Presentador asociado a la vista.
Array<String>	componentsShown Array con los nombres de los componentes a mostrar en la vista.

String?	itemId Identificador de la categoría o producto que se muestra.
ArrayList<CatalogGalleryItem>?	itemList Lista de elementos que compondrán la lista de tarjetas.
ArrayList<CatalogGalleryItem>?	itemListCopy Copia de la lista de elementos que compondrán la lista de tarjetas.

Constructores
CatalogGalleryRecycleAdapter ( ) Construye un objeto de la clase *CatalogGalleryRecycleAdapter*.

Clases internas	
ViewHolder	Clase encargada de la vista.

Métodos privados	
void	organizeCategoryComponents (viewHolder: ViewHolder, componentsShown: Array<String>, item: CatalogGalleryItem) Organiza los elementos de la vista cuando es una categoría el elemento principal de la actividad.
void	organizeProductComponents (viewHolder: ViewHolder, item: String)

	Organiza los elementos de la vista cuando es un producto el elemento principal de la actividad.
void	outstandingsFilter (outstandings: Boolean) Filtra los elementos de la lista en función de si son destacados o no.

Métodos heredados	
De RecyclerView.Adapter	
ViewHolder	onCreateViewHolder (parent: ViewGroup, viewType: Int) Se encarga de inflar la vista.
void	onBindViewHolder (viewHolder: ViewHolder, index: Int) Se encarga de configurar la vista.
Int	getItemCount () Obtiene el número total de elementos que maneja el adaptador.

Variables

context

```
private val context: Context
```

Actividad actual.

presenter

```
private val presenter: CatalogGalleryContract.Presenter
```

Representa el presentador asociado a la vista.

componentsShown

```
private val componentsShown: Array<String>
```

Array con los nombres de los componentes a mostrar en la vista expandida.

Valores actuales que puede tener:

- title
- subtitle
- image
- description
- outstandingIcon
- shareButton

itemId

```
private val itemId: String
```

Identificador de la categoría o producto mostrado.

itemImageUrls

```
private val parents: ArrayList<Parent>
```

Lista de elementos que compondrán la lista de elementos general.

itemList

```
private val parents: ArrayList<Parent>
```

Lista de elementos que compondrán la lista de elementos general.

itemListCopy

```
private val parents: ArrayList<Parent>
```

Copia de la lista de elementos que compondrán la lista de elementos general.

Valor inicial: vacío.

Constructores

CatalogGalleryRecycleAdapter

```
init
```

Añade todos los elementos de la variable *parents* en la de *parents copy.*

Clases internas

ViewHolder

```
class ViewHolder (itemView: View): RecyclerView.ViewHolder
(itemView)
```

Esta clase se encarga de la vista. Obtiene los componentes de la vista y los inicializa como variables.

Variables	
TextView	title Componente en el que se muestra el título de una categoría o producto.
ImageView	image Componente en el que se muestra una imagen representativa de una categoría o producto.
ImageView	outstandingIcon Componente en el que se muestra el icono de destacados.
Button	shareButton Componente correspondiente a la acción de compartir.

Métodos privados

organizeCategoryComponents

```
private fun organizeCategoryComponents (viewHolder: ViewHolder,
componentsShown: Array<String>, item: CatalogGalleryItem)
```

Revisa uno por uno los componentes de la vista y si la variable *componentsShown* los contiene asigna su valor correspondiente, por el contrario, oculta el componente.

Parámetros	
viewHolder	ViewHolder: vista.
componentsShown	Array<String>: lista de los nombres de los componentes a mostrar en la vista.
item	CatalogGalleryItem: elemento del controlador.

organizeProductComponents

```
private fun organizeProductComponents (viewHolder: ViewHolder, item: String)
```

Carga la imagen pasada por parámetro y oculta el resto de los elementos de la tarjeta.

Parámetros	
viewHolder	ViewHolder: vista.
item	String: URL de una imagen del producto que muestra el controlador.

outstandingsFilter

```
fun outstandingsFilter (outstandings: Boolean)
```

Si *outstandings* es verdadero crea una lista en la que almacena los elementos de la lista parentsCopy que pertenecen a destacados, limpia la lista parents y añade le añade todos los resultados. En caso de ser falso, limpia la lista original de elementos y le añade todos los de la copia. Por último, notifica al adaptador que los datos han cambiado.

Parámetros	
outstandings	Boolean: verdadero para mostrar los elementos destacados, falso para mostrar todos.

Métodos heredados

onCreateViewHolder

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
```

Se encarga de inflar la vista con el *layout* correspondiente a la vista expansible.

Parámetros	
parent	ViewGroup
viewType	Int

Retorno	
ViewHolder	Vista del elemento del controlador.

onBindViewHolder

```
override fun onBindViewHolder (viewHolder: ViewHolder, index: Int)
```

Se encarga configurar la vista. Actúa de dos formas:

- Si hay elementos en la variable itemImageUrls organiza la vista en forma de galería de imágenes, crea una transición al contenedor de la imagen y define un OnClickListener, el cual se encarga de llamar al presentador indicando que una imagen de la galería ha sido pulsada.
- Si hay elementos en la variable itemList organiza la vista es forma de tarjetas, comprueba si se muestra el botón de compartir y en caso afirmativo crea una instancia y la lanza para compartir las imágenes de la categoría o producto. Además, define otro OnClickListener para el que llama al presentador para que gestione el evento.

Parámetros	
viewHolder	ViewHolder: vista.
index	Int: posición del elemento en la lista.

getItemCount

```
override fun getItemCount ()
```

Devuelve el número total de elementos del conjunto de datos que tiene el adaptador.

Retorno	
Int	Número total de elementos que gestiona el adaptador.
expandableType	Parent: elemento expandible.
position	Int: posición del elemento en la lista.

CHILD

```
data class Child
```

Esta es una clase de datos contiene la información localizada en la vista expandida de la tarjeta.

Variables	
String?	subtitle Subtítulo de la categoría o producto. Valor inicial: nulo.
String?	description Descripción de la categoría o producto. Valor inicial: nulo.

PARENT

```
data class Parent (val item: CatalogGalleryItem):
ExpandableRecyclerViewAdapter.ExpandableGroup<Child> ()
```

Esta es una clase de datos contiene la información localizada en la vista que se puede expandir de la tarjeta.

Variables	
CatalogGalleryItem	item

	Elemento del controlador.

Métodos heredados	
De ExpandableRecyclerViewAdapter.ExpandableGroup<Child>	
List<Child>	getExpandingItems ()
	Crea una lista con los elementos que son parte la vista expandida de la lista.

Variables

item

```
val item: CatalogGalleryItem
```

Elemento de la lista de tarjetas.

Métodos heredados

getExpandingItems

```
override fun getExpandingItems ()
```

Crea una lista con los elementos que pertenecen a la vista expandida. Sólo hay un elemento que se expande dentro de la lista y contendría el subtítulo, descripción.

Retorno	
List<Child>	Lista con los elementos de la vista expandida.

CATALOGGALLERYEXPANDABLERECYCLEADAPTER

```
class CatalogGalleryExpandableRecycleAdapter (
 private val presenter: CatalogGalleryContract.Presenter,
 private val componentsShown: Array<String>,
 private val expandable: Boolean,
 private var parents: ArrayList<Parent>
) : ExpandableRecyclerViewAdapter<Child, Parent,
```

```
CatalogGalleryExpandableRecycleAdapter.PViewHolder,
CatalogGalleryExpandableRecycleAdapter.CViewHolder>(parents,
ExpandingDirection.VERTICAL)
```

Esta clase se encarga de adaptar los datos pasados por parámetros a unas vistas predeterminadas.

Resumen

Variables	
CatalogGalleryContract.Presenter	presenter Presentador asociado a la vista.
Array<String>	componentsShown Array con los nombres de los componentes a mostrar en la vista.
Boolean	expandable Indica si la lista es expandible o no.
ArrayList<Parent>	parents Lista de elementos que compondrán la lista.
ArrayList<Parent>	parentsCopy Copia de la lista de elementos que compondrán la lista de tarjetas.

Constructores
CatalogGalleryExpandableRecycleAdapter ( ) Construye un objeto de la clase CatalogGalleryExpandableRecycleAdapter.

Clases internas	
PViewHolder	Clase encargada de la vista expansible.
CViewHolder	Clase encargada de la vista expandida.

Métodos privados	
void	organizeParentComponents (parentViewHolder: PViewHolder, expandableType: Parent, componentsShown: Array<String>) Organiza los elementos de la vista expansible.
void	organizeChildComponents (childViewHolder: CViewHolder, expandedType: Child, componentsShown: Array<String>) Organiza los elementos de la vista expandida.
void	outstandingsFilter (outstandings: Boolean) Filtra los elementos de la lista en función de si son destacados o no.

Métodos heredados	
De ExpandableRecyclerViewAdapter	
PViewHolder	onCreateParentViewHolder (parent: ViewGroup, viewType: Int) Se encarga de inflar la vista con la vista expansible.
CViewHolder	onCreateChildViewHolder (child: ViewGroup, viewType: Int) Se encarga de inflar la vista con la vista expandida.
void	onBindParentViewHolder (parentViewHolder: PViewHolder, expandableType: Parent, position: Int) Se encarga de configurar la expansible.
void	onBindChildViewHolder (childViewHolder: CViewHolder, expandedType: Child, expandableType: Parent, position: Int) Se encarga de configurar la expandida.

Variables

presenter

```
private val presenter: CatalogGalleryContract.Presenter
```

Representa el presentador asociado a la vista.

componentsShown

```
private val componentsShown: Array<String>
```

Array con los nombres de los componentes a mostrar en la vista expandida.

Valores actuales que puede tener:

- title
- subtitle
- image
- description
- outstandingIcon
- shareButton

expandable

```
private val expandable: Boolean
```

Indica si la lista es expandible o no.

parents

```
private val parents: ArrayList<Parent>
```

Lista de elementos que compondrán la lista de elementos general.

parentsCopy

```
private val parents: ArrayList<Parent>
```

Copia de la lista de elementos que compondrán la lista de elementos general.

Valor inicial: vacío.

Constructores

CatalogGalleryExpandableRecycleAdapter

```
init
```

Añade todos los elementos de la variable *parents* en la de *parents copy.*

Clases internas

PViewHolder

```
class PViewHolder (itemView: View):
ExpandableRecyclerViewAdapter.ExpandableViewHolder (itemView)
```

Esta clase se encarga de la vista expansible. Obtiene los componentes de la vista y los inicializa como variables.

Variables	
TextView	title Componente en el que se muestra el título de una categoría o producto.
ImageView	image Componente en el que se muestra una imagen representativa de una categoría o producto.
ImageView	outstandingIcon Componente en el que se muestra el icono de destacados.
Button	shareButton Componente correspondiente a la acción de compartir.

CViewHolder

```
class CViewHolder (itemView: View):
ExpandableRecyclerViewAdapter.ExpandableViewHolder (itemView)
```

Esta clase se encarga de la vista expandida. Obtiene los componentes de la vista y los inicializa como variables.

Variables	
TextView	subtitle Componente en el que se muestra el subtítulo de una categoría o producto.
TextView	description Componente en el que se muestra una breve descripción de una categoría o producto.
Button	shareButton Componente correspondiente a la acción de compartir.

Métodos privados

organizeParentComponents

```
private fun organizeParentComponents (parentViewHolder: PViewHolder, expandableType: Parent, componentsShown: Array<String>)
```

Revisa uno por uno los componentes de la vista y si la variable *componentsShown* los contienen, si es verdadero le asigna su valor correspondiente, por el contrario, oculta el componente.

Parámetros	
parentViewHolder	PViewHolder: vista expansible.
expandableType	Parent: elemento que contiene los datos.
componentsShown	Array<String>: lista de los nombres de los componentes a mostrar en la vista.

organizeChildComponents

```
private fun organizeChildComponents (childViewHolder: CViewHolder,
expandedType: Child, componentsShown: Array<String>)
```

Revisa uno por uno los componentes de la vista y si la variable *componentsShown* los contienen, si es verdadero le asigna su valor correspondiente, por el contrario, oculta el componente.

Parámetros	
childViewHolder	CViewHolder: vista expandida.
expandedType	Child: elemento que contiene los datos.
componentsShown	Array<String>: lista de los nombres de los componentes a mostrar en la vista.

outstandingsFilter

```
fun outstandingsFilter (outstandings: Boolean)
```

Si *outstandings* es verdadero crea una lista en la que almacena los elementos de la lista parentsCopy que pertenecen a destacados, limpia la lista parents y le añade todos los resultados. En caso de ser falso, limpia la lista original de elementos y le añade todos los de la copia. Por último, notifica al adaptador que los datos han cambiado.

Parámetros	
outstandings	Boolean: verdadero para mostrar los elementos destacados, falso para mostrar todos.

Métodos heredados

onCreateParentViewHolder

```
override fun onCreateParentViewHolder(parent: ViewGroup, viewType: Int)
```

Se encarga de inflar la vista con el *layout* correspondiente a la vista expansible.

Parámetros	
parent	ViewGroup
viewType	Int

Retorno	
PViewHolder	Vista expansible del elemento del controlador.

onCreateChildViewHolder

```
override fun onCreateChildViewHolder(child: ViewGroup, viewType: Int)
```

Se encarga de inflar la vista con el *layout* correspondiente a la vista expansible.

Parámetros	
child	ViewGroup
viewType	Int

Retorno	
CViewHolder	Vista expandida del elemento del controlador.

onBindParentViewHolder

```
override fun onBindParentViewHolder (parentViewHolder: PViewHolder,
expandableType: Parent, position: Int)
```

Se encarga de configurar la vista expansible. Organiza los componentes y en el caso de que las tarjetas no sea expansibles comprueba si se muestra el botón de compartir para definir su OnClickListener, el cual se encarga de crear una instancia y la lanza para compartir las imágenes de la categoría o producto y descripción. Además, define otro OnClickListener para el que llama al presentador para que gestione el evento.

Parámetros	
parentViewHolder	PViewHolder: vista expansible.

expandableType	Parent: elemento expandible.
position	Int: posición del elemento en la lista.

onBindChildViewHolder

```
override fun onBindChildViewHolder (childViewHolder: CViewHolder, expandedType: Child, expandableType: Parent, position: Int)
```

Se encarga de configurar la vista expandida. En el caso de que la tarjeta sea expansible organiza los componentes, comprueba si se muestra el botón de compartir para definir su OnClickListener, el cual se encarga de crear una instancia y la lanza para compartir las imágenes de la categoría o producto, su título y descripción. En el caso de que la tarjeta no sea expandible la oculta.

Parámetros	
childViewHolder	CViewHolder: vista expansible.
expandedType	Child: elemento expandido.
expandableType	Parent: elemento expandible.
position	Int: posición del elemento en la lista.

CATALOGGALLERYFRAGMENT

```
class CatalogGalleryFragment: Fragment(), CatalogGalleryContract.View
```

Esta clase se encarga del comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador *CatalogGallery*.

Resumen

Variables	
View	rootView

	Interfaz de usuario que implementa el fragmento.
Boolean	showOutstandings Indica si solo se muestran los objetos destacados o todos.

Variables heredadas	
De la interfaz CatalogGalleryContract.View	
CatalogGalleryContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreateOptionsMenu (menu: Menu?) Inicializar el contenido del menú de opciones estándar de la Actividad.
void	onPrepareOptionsMenu (menu: Menu) Modifica los elementos del menú de la barra de acciones antes de mostrarlos.
Boolean	onOptionsItemSelected (item: MenuItem) Se llama cuando un elemento de la barra de acciones es pulsado para gestionar el evento.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

showOutstandings

```
private var showOutstandings
```

Si es verdadera se muestran en la vista sólo los elementos de destacados, de lo contrario se muestran todos los elementos.

Valor inicial: falso.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogGalleryContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el *layout* correspondiente al fragmento, pedir al *screen* que lo configure, habilitar el cambio de opciones a la barra de acciones y pedir los datos de configuración al presentador.

Una vez obtenidos los datos, si hay actividad muestra el título correspondiente, muestra la vista a reciclar y oculta la barra de progreso. Se establece los componentes a mostrar por defecto, tanto para expansible como para no y si se pueden obtener de los datos se sustituyen, los mismo para el número de columnas por fila.

Asimismo, se comprueba si entre los componentes a mostrar se encuentra la opción de destacados y se incorpora en la barra de acciones. Posteriormente, se configura la lista de elementos obtenidos como elementos de la clase Parent, se crea y pasa el adaptador a la vista reciclada para que adapte todos los elementos.

Parámetros	
`inflater`	`LayoutInflater`: objeto que infla cualquier vista en el fragmento.
`container`	`ViewGroup`: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
`savedInstanceState`	`Bundle`: estado anteriormente guardado del fragmento.

Retorno	
`View`?	Vista correspondiente a la interfaz de usuario del fragmento.

onCreateOptionsMenu

```
override fun onCreateOptionsMenu (menu: Menu, inflater: MenuInflater)
```

Incorpora en el menú de la barra de aplicación superior un menú inflandolo.

Parámetros	
`menu`	`Menu`: menú de opciones en el que coloca los elementos.
`inflater`	`MenuInflater`

onPrepareOptionsMenu

```
override fun onPrepareOptionsMenu (menu: Menu)
```

Prepara el menú de opciones estándar de la pantalla para que se muestre. Dependiendo de si la variable showOutstandings es verdadera o no, cambia el título de la opción del menú denominada *outstandings*.

Parámetros	
`menu`	`Menu`: el menú de opciones como se mostró por última vez o inicializado por primera vez por *onCreateOptionsMenu*.

onOptionsItemSelected

```
override fun onOptionsItemSelected (item: MenuItem)
```

Le pide al presentador que navegue a la categoría o producto correspondiente al elemento pulsado.

Parámetros	
`item`	`MenuItem`: elemento del menú de la barra superior de navegación seleccionado.

Retorno	
`Boolean`	Devuelve falso para permitir que continúe el procesamiento normal del menú, verdadero para consumirlo aquí.

3.2.5. CatalogMap

Este controlador de pantalla se encarga de mostrar un mapa con diferentes localizaciones.

CatalogMapViewModel

```
class CatalogMapViewModel: ViewModel()
```

Esta clase se encarga conservar el estado de la pantalla *CatalogMap*.

Resumen

Variables	
`MutableLiveData<CatalogMapData>`	`liveData` Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
`void`	`setData (data: CatalogMapData?)` Modifica el valor de la única variable de la clase.

CatalogMapData?	getData() Obtiene el valor de la única variable de la clase.
LiveData<CatalogMapData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogMapData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData (data: CatalogMapData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogMapData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData ()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogMapData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData ()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogMapData>	Única variable de la clase.

CATALOGMAPCONTRACT

```
interface CatalogMapContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en controlador *CatalogMap*.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.
Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	presenter Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
LiveData<CatalogMapData>	fetchScreenData () Método que busca los datos necesarios para la configuración de la pantalla.
void	onMapItemClicked (data: HashMap<String, Any?>) Método que se encarga de navegar a otro controlador al pulsar algún elemento del mapa.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogMapData>	fetchScreenData (itemId: String?, structureType: String?) Método que obtiene los datos necesarios para la configuración de la pantalla.

Router

```
interface Router
```

Interfaz que define las variables y métodos del router.

Métodos	
void	passDataToNextScreen (itemId: Int, structureType: CatalogMapItemType, itemTypeId: String?)

	Método que se encarga de pasar los datos obtenidos por parámetro al siguiente controlador.
HashMap<String,String?>	getDataFromPreviousScreen () Método que se encarga de obtener los datos pasados por el anterior controlador.
void	navigateToNextScreen (screenType: String) Método que se encarga de cambiar al siguiente controlador.

CATALOGMAPSCREEN

```
object CatalogMapScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla *CatalogMap*.

Resumen

Métodos públicos	
void	configureActivity(view: CatalogMapContract.View) Configura e inyecta las dependencias siendo la vista una FragmentActivity.
void	configureFragment(view: CatalogMapContract.View) Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogMapContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogMapContract.View`: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogMapContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogMapContract.View`: vista que se muestra.

CatalogMapRouter

```
class CatalogMapRouter: CatalogMapContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla *CatalogMap*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.

Métodos heredados	
De la interfaz CatalogMapContract.Router	
HashMap<String,String?>	getDataFromPreviousScreen () Obtiene el identificador de la categoría o producto a mostrar y qué es.
void	passDataToNextScreen (itemId: Int, itemType: CatalogMapItemType, itemTypeId: String?) Incorpora en el mediador información para el siguiente controlador.
void	navigateToNextScreen (screenType: String?) Inicia el siguiente controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

Métodos heredados

getDataFromPreviousScreen

```
override fun getDataFromPreviousScreen ()
```

Si puede obtener el mediador a partir de la actividad, obtiene el valor almacenado en la variable del identificador. Dependiendo del identificador que obtenga le asigna un tipo *product* o *category* y los incorpora en un array asociativo. Posteriormente reinicia todos los parámetros del mediador.

Retorno	
HashMap<String, String?>	Array asociativo que contiene el identificador de una categoría o producto y su tipo.

passDataToNextScreen

```
override fun passDataToNextScreen (itemId: Int, itemType: CatalogMapItemType, itemTypeId: String?)
```

Si puede obtener la actividad del fragmento y de esta el mediador, dependiendo de si recibe por parámetro una categoría o un producto almacena los identificadores pasados por parámetros en las correspondientes variables del mediador.

Parámetros	
itemId	Int: Identificador de una categoría o un producto.
itemType	CatalogMapItemType: Identifica si es una categoría o un producto.
itemTypeId	String?: Identificador de una categoría.

navigateToNextScreen

```
override fun navigateToNextScreen (screenType: String?)
```

Si puede obtener la actividad del fragmento, obtiene una instancia de controlador correspondiente al nombre pasado por parámetro y la inicializa, terminando la actividad actual.

Parámetros	
screentype	String: nombre de controlador.

CatalogMapItemType

Esta clase es un tipo de datos especial que indica que la variable de tipo *CatalogMapItemType* debe ser igual a uno de los valores predefinidos, *CATEGORY* o *PRODUCT*. Indica si los datos obtenidos pertenecen a una categoría o un producto.

CatalogMapData

```
data class CatalogMapData
```

Esta es una clase de datos que contiene la información que configura la vista.

Variables	
String	title Nombre del producto o la categoría de la cual obtenemos los datos.
ArrayList<CatalogMapItem>	itemList Lista de datos de las categorías o productos hijos del cual estamos obteniendo los datos.

CatalogMapItem

```
data class CatalogMapItem
```

Esta es una clase de datos contiene la información que configura la vista además de datos para navegación.

Variables	
String	id Identificador de un producto, una categoría o un elemento.
String	title Nombre de categoría, producto o elemento.

String?	subtitle Subtítulo de categoría, producto o elemento. Valor inicial: nulo.
Double?	latitude Coordenada de latitud. Valor inicial: nulo
Double?	longitude Coordenada de longitud. Valor inicial: nulo
CatalogMapItemType?	type Indica si es un producto, una categoría o un elemento simple. Valor inicial: nulo.
String?	typeId Identificador de una categoría. Valor inicial: nulo
String?	screenType Nombre del controlador de pantalla en el que se muestra. Valores actuales que puede tomar: – list – gallery – imageViewer – html – map – detail

CATALOGMAPMODEL

```
class CatalogMapModel (): CatalogMapContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador *CatalogMap*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogMapData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogMapModel() Construye un objeto de la clase *CatalogMapModel*.

Métodos heredados	
De la interfaz CatalogMapContract.Model	
LiveData<CatalogMapData>	fetchScreenData (itemId: String?, structureType: String?) Obtiene los datos necesarios para el controlador.

Métodos privados	
ArrayList<CatalogMapItem>	getCategoryLocations (category: Category) Convierte las localizaciones de una categoría a una lista de elementos del controlador.
ArrayList<CatalogMapItem>	getProductLocations (product: Product, category: Category) Convierte las localizaciones de un producto a una lista de elementos del controlador.
ArrayList<CatalogMapItem>	getPlacesList (places: String, type: CatalogMapItemType?, typeId: String?, screenType: String?) Convierte un texto a elementos del controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogMapData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogMapModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData (itemId: String?, structureType: String?)
```

Si obtiene un identificador por parámetro, actúa dependiendo de su tipo de una forma u otra.

- Para un producto obtiene su nombre y la lista de localizaciones.
- Para una categoría, obtiene su nombre y categorías y productos asociados, obteniendo de cada uno de ellos sus localizaciones.

Parámetros	
itemId	String: Identificador de una categoría o un producto.
structureType	String: Indica si el identificador pertenece a un producto o una categoría.

Retorno	
LiveData<CatalogMapData>	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

Métodos privados

getCategoryLocations

```
private fun getCategoryLocations (category: Category)
```

Obtiene el producto principal de la categoría pasada por parámetro, crea una lista de elementos del controlador y con el nombre del controlador con el que se muestra la categoría y el texto con los lugares obtiene elementos del controlador y los incorpora en la lista.

Retorno	
ArrayList<CatalogMapItem>	Lista de localizaciones de una categoría.

getProductLocations

```
private fun getProductLocations (product: Product, category: Category)
```

Crea una lista de elementos del controlador y con el nombre del controlador con el que se muestra el producto y el texto con los lugares y obtiene los elementos del controlador y los incorpora en la lista.

Retorno	
ArrayList<CatalogMapItem>	Lista de localizaciones de un producto.

getPlacesList

```
private fun getPlacesList (places: String, type: CatalogMapItemType?, typeId: String?, screenType: String?)
```

Crea una lista de elementos del controlador, el texto con lugares pasado por parámetro lo convierte a una lista de elementos del controlador.

Retorno	
ArrayList<CatalogMapItem>	Lista de localizaciones.

CatalogMapPresenter

```
class CatalogMapPresenter (): CatalogMapContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador *CatalogMap*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogMapViewModel	viewModel Objeto que mantiene el estado de la pantalla.
CatalogMapContract.Model	model Objeto del que se obtienen los datos para la configuración de la pantalla.
CatalogMapContract.Router	router Objeto que se encarga de la navegación.

Métodos heredados	
De la interfaz CatalogMapContract.Presenter	
LiveData<CatalogMapData>	fetchScreenData() Obtiene los datos necesarios para la configuración de la vista.
void	onMapItemClicked (item: HashMap<String,Any?>) Navegar a otro controlador al pulsar algún elemento del mapa.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogMapViewModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogMapContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogMapContract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener los datos del *ViewModel* retorna esos datos, por el contrario, si puede obtener el contexto de la aplicación, le pide al router los datos del mediador y luego le pide al modelo los datos necesarios para la configuración de la vista del controlador pasando por parámetro los datos obtenidos del router.

Retorno	
LiveData<CatalogMapData>	Variable de la clase que contiene los datos necesarios para la configuración de la vista.

onMapItemClicked

```
override fun onMapItemClicked (item: HashMap<String,Any?>)
```

Si puede obtener tipo del controlador, es decir, si es una categoría o un producto le pide al router que pase los datos al siguiente navegador y que navegue.

Parámetros	
item	HashMap<String,Any?>: Array asociativo con el identificador, tipo de elemento (categoría o producto), identificador en el caso de ser una categoría y nombre del controlador en el que se muestra.

CatalogMapActivity

```
class CatalogMapActivity: AppCompatActivity(), CatalogMapContract.View
```

Esta clase se encarga de la detección y notificación de eventos, así como la actualización y gestión de interfaz de usuario del controlador *CatalogMap*.

Resumen

Variables heredadas	
De la interfaz CatalogMapContract.View	
CatalogMapContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreate(savedInstanceState: Bundle?) Crea la vista.
De androidx.appcompat.app.AppCompatActivity	
void	onSupportNavigateUp () Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogMapContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el *layout* correspondiente a la actividad, carga el fragmento correspondiente al controlador, establece los efectos de transición entre pantallas. Y si hay, configura la barra de acciones para que muestre la navegación hacia atrás.

Parámetros	
savedInstanceState	Bundle: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

onSupportNavigateUp

```
override fun onSupportNavigateUp ()
```

Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones haciendo uso del método *onBackPressed* heredado.

Retorno	
Boolean	Es verdadero si la navegación hacia arriba se completó correctamente y esta actividad se completó, falso en caso contrario.

CatalogMapFragment

```
class CatalogGalleryFragment: Fragment(), CatalogMapContract.View,
OnMapReadyCallback, GoogleMap.OnInfoWindowClickListener
```

Esta clase se encarga del comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador *CatalogMap*.

Resumen

Variables	
View	rootView Interfaz de usuario que implementa el fragmento.

Variables heredadas	
De la interfaz CatalogMapContract.View	
CatalogMapContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.
De la interfaz OnMapReadyCallback	
void	onMapReady (googleMap: GoogleMap) Configura los datos en el mapa.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogGalleryContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el *layout* correspondiente al fragmento, pedir al *screen* que configure el fragmento y obtener el fragmento del mapa con el objetivo de saber cuándo está listo.

Parámetros	
`inflater`	`LayoutInflater`: objeto que infla cualquier vista en el fragmento.
`container`	`ViewGroup`: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
`savedInstanceState`	`Bundle`: estado anteriormente guardado del fragmento.

Retorno	
`View?`	Vista correspondiente a la interfaz de usuario del fragmento.

onMapReady

```
override fun onMapReady (googleMap: GoogleMap)
```

Una vez el mapa está listo incorpora un *Listener* a la ventana de información de cada marcador y pide los datos de configuración al presentador.

Una vez obtenidos los datos, si hay actividad muestra el título correspondiente y si la lista de elementos no está vacía, por cada elemento crea su marcador con los datos correspondientes e incorpora en la etiqueta de cada uno su identificador, tipo (categoría, producto o nulo), el identificador de la categoría y el nombre del controlador en el que se muestra.

Además, se coloca la cámara de forma que aparezcan todos los marcadores.

Parámetros	
`googleMap`	`GoogleMap`: una instancia no nula de un mapa asociado con el MapFragment.

3.2.6. CATALOGDETAIL

Este controlador de pantalla se encarga de mostrar un carrusel de imágenes, un título, un subtítulo y una descripción.

CATALOGDETAILVIEWMODEL

```
class CatalogDetailViewModel: ViewModel()
```

Esta clase se encarga conservar el estado de la pantalla *CatalogDetail*.

Resumen

Variables	
MutableLiveData<CatalogDetailData>	liveData Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
void	setData (data: CatalogDetailData?) Modifica el valor de la única variable de la clase.
CatalogDetailData?	getData() Obtiene el valor de la única variable de la clase.
LiveData<CatalogDetailData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogDetailData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData (data: CatalogDetailData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogDetailData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData ()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogDetailData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData ()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogDetailData>	Única variable de la clase.

CATALOGDETAILCONTRACT

```
interface CatalogDetailContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en controlador *CatalogDetail*.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.
Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	`presenter` Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
LiveData<CatalogDetailData>	`fetchScreenData ()` Método que busca los datos necesarios para la configuración de la pantalla.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogDetailData>	fetchScreenData (itemId: String?, structureType: String?) Método que obtiene los datos necesarios para la configuración de la pantalla.

Router

```
interface Router
```

Interfaz que define las variables y métodos del router.

Métodos	
void	passDataToNextScreen (itemId: Int, structureType: CatalogDetailItemType, itemTypeId: String?) Método que se encarga de pasar los datos obtenidos por parámetro al siguiente controlador.
HashMap<String,String>	getDataFromPreviousScreen () Método que se encarga de obtener los datos pasados por el anterior controlador.
void	navigateToNextScreen (screenType: String?) Método que se encarga de cambiar al siguiente controlador.

CatalogDetailScreen

```
object CatalogDetailScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla *CatalogDetail*.

Resumen

Métodos públicos	
`void`	`configureActivity(view: CatalogDetailContract.View)` Configura e inyecta las dependencias siendo la vista una FragmentActivity.
`void`	`configureFragment(view: CatalogDetailContract.View)` Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogDetailContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogDetailContract.View`: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogDetailContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogDetailContract.View`: vista que se muestra.

CATALOGDETAILROUTER

```
class CatalogDetailRouter: CatalogDetailContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla *CatalogDetail*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.

Métodos heredados	
De la interfaz CatalogDetailContract.Router	
HashMap<String,String>	getDataFromPreviousScreen () Obtiene el identificador de la categoría o producto a mostrar y qué es.
void	passDataToNextScreen (itemId: Int, itemType: CatalogDetailItemType, itemTypeId: String?) Incorpora en el mediador información para el siguiente controlador.
void	navigateToNextScreen (screenType: String?) Inicia el siguiente controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

Métodos heredados

getDataFromPreviousScreen

```
override fun getDataFromPreviousScreen ()
```

Si puede obtener el mediador a partir de la actividad, obtiene el valor almacenado en la variable del identificador. Dependiendo del identificador que obtenga le asigna un tipo *product* o *category* y los incorpora en un array asociativo. Posteriormente reinicia todos los parámetros del mediador.

Retorno	
HashMap<String, String?>	Array asociativo que contiene el identificador de una categoría o producto y su tipo.

passDataToNextScreen

```
override fun passDataToNextScreen (itemId: Int, itemType:
CatalogDetailItemType, itemTypeId: String?)
```

Si puede obtener la actividad de fragmento y de esta el mediador, dependiendo de si recibe por parámetro una categoría o un producto almacena los identificadores pasados por parámetros en las correspondientes variables del mediador.

Parámetros	
itemId	Int: Identificador de una categoría o un producto.
itemType	CatalogDetailItemType: Identifica si es una categoría o un producto.

itemTypeId	String?: Identificador de una categoría.

navigateToNextScreen

```
override fun navigateToNextScreen (screenType: String?)
```

Si puede obtener la actividad del fragmento, obtiene una instancia de controlador correspondiente al nombre pasado por parámetro y la inicializa, terminando la actividad actual.

Parámetros	
screentype	String: nombre de controlador.

CATALOGDETAILITEMTYPE

Esta clase es un tipo de datos especial que indica que la variable de tipo *CatalogDetailItemType* debe ser igual a uno de los valores predefinidos, *CATEGORY* o *PRODUCT*. Indica si los datos obtenidos pertenecen a una categoría o un producto.

CATALOGDETAILDATA

```
data class CatalogDetailData
```

Esta es una clase de datos contiene la información que configura la vista.

Variables	
String	itemId Identificador de un producto o una categoría.
CatalogDetailItemType	type Indica si es un producto, una categoría o un elemento simple. Valor inicial: nulo.
String?	typeId

	Identificador de una categoría. Valor inicial: nulo
Array<String>?	componentsShown Array con los nombres de los componentes a mostrar en la vista. Valores actuales que puede tener: – title – subtitle – image – description
HashMap<String,String>	carouselAttr Array asociativo con los valores de configuración del carrusel de imágenes. Valor inicial: nulo. Atributos del carrusel que contiene: - *position*: posición dentro del layout principal. Valores: *top* (parte superior), *center* (centro), *bottom* (parte inferior). - *type*: cuantas imágenes se muestran a la vez. Valores: *block* (una), *showcase* (varias). - *showCaption*: indican si se muestra el subtítulo con el número de imagen en el carrusel. Valores: verdadero si se muestra, falso en caso contrario. - *showIndicator*: indican si se muestran los indicadores. Valores: verdadero si se muestra, falso en caso contrario. - *showNavigationButtons*: indican si se muestran las flechas de navegación.

	Valores: verdadero si se muestra, falso en caso contrario. - *autoPlay*: indican si las imágenes se mueven automáticamente. Valores: verdadero si se mueven, falso en caso contrario.
String?	title Nombre de categoría o producto. Valor inicial: nulo.
String?	subtitle Subtítulo de categoría o producto. Valor inicial: nulo.
String?	description Descripción de categoría o producto. Valor inicial: nulo.
ArrayList<String>	itemImageUrls Lista de imágenes pertenecientes a un producto o una categoría. Valor inicial: vacío.

CATALOGDETAILMODEL

```
class CatalogDetailModel (): CatalogDetailContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador *CatalogDetail*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogDetailData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogDetailModel() Construye un objeto de la clase *CatalogDetailModel*.

Métodos heredados	
De la interfaz CatalogDetailContract.Model	
LiveData<CatalogDetailData>	fetchScreenData(itemId: String?, structureType: String?) Obtiene los datos necesarios para el controlador.

Métodos privados	
HashMap<String,String?>	getCarouselAttributes (strs: String) Mapea los atributos del carrusel pasados por parámetros.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogDetailData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogDetailModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData (itemId: String?, structureType: String?)
```

Si obtiene un identificador por parámetro, obtiene el producto asociado y en el caso de que sea el identificador de una categoría obtiene su producto principal. Del producto obtiene los parámetros necesarios para crear el dato que configura la pantalla.

Parámetros	
`itemId`	`String`: Identificador de una categoría o un producto.
`structureType`	`String`: Indica si el identificador pertenece a un producto o una categoría.

Retorno	
`LiveData<CatalogDetailData>`	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

Métodos privados

getCarouselAttributes

```
private fun getCarouselAttributes (strs: String)
```

Se encarga de mapear en texto obtenido a los valores de atributos del carrusel.

Retorno	
`HashMap<String,String?>`	Array con los valores de configuración del carrusel.

CATALOGDETAILPRESENTER

```
class CatalogDetailPresenter (): CatalogDetailContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador *CatalogDetail*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogDetailViewModel	viewModel Objeto que almacena el estado de la pantalla.
CatalogDetailContract.Model	model Objeto del que se obtienen los datos que configuran la vista.
CatalogDetailContract.Router	router Objeto que se encarga de la navegación entre pantallas.

Métodos heredados	
De la interfaz CatalogDetailContract.Presenter	
LiveData<CatalogDetailData>	fetchScreenData() Obtiene los datos necesarios para la configuración de la vista.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogDetailViewModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogDetailContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogDetailContract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener los datos del *ViewModel* retorna esos datos, por el contrario, si puede obtener el contexto de la aplicación, le pide al router los datos del mediador y luego le pide al modelo los datos necesarios para la configuración de la vista del controlador pasando por parámetro los datos obtenidos del router.

Retorno	
LiveData<CatalogDetailData>	Variable de la clase que contiene los datos necesarios para la configuración de la vista.

CATALOGDETAILACTIVITY

```
class CatalogDetailActivity: AppCompatActivity(), CatalogDetailContract.View
```

Esta clase se encarga de la detección y notificación de eventos, así como la actualización y gestión de interfaz de usuario del controlador *CatalogDetail*.

Resumen

Variables heredadas	
De la interfaz CatalogDetailContract.View	
CatalogDetailContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreate(savedInstanceState: Bundle?) Crea la vista.
De androidx.appcompat.app.AppCompatActivity	
void	onSupportNavigateUp () Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogDetailContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el layout correspondiente a la actividad, carga el fragmento correspondiente al controlador, establece los efectos de transición entre pantallas. Y si hay, configura la barra de acciones para que muestre la navegación hacia atrás.

Parámetros	
savedInstanceState	Bundle: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

onSupportNavigateUp

```
override fun onSupportNavigateUp ()
```

Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones haciendo uso del método *onBackPressed* heredado.

Retorno	
Boolean	Es verdadero si la navegación hacia arriba se completó correctamente y esta actividad se completó, falso en caso contrario.

CatalogDetailFragment

```
class CatalogDetailFragment: Fragment(), CatalogDetailContract.View
```

Esta clase se encarga comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador *CatalogDetail*.

Resumen

Variables	
View	rootView Interfaz de usuario que implementa el fragmento.

Variables heredadas	
De la interfaz CatalogDetailContract.View	
CatalogDetailContract.View	presenter Presentador asociado a la vista.

Métodos privados	
void	configureCarousel (imageUrls: Array<String>, carouselAttr: HashMap<String, String?>?) Se encarga de configurar el carrusel.
void	configureComponentsPosition (carouselPosition: String?, componentsShown: Array<String>) Se encarga de configurar el orden de los componentes en el *layout*.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogDetailContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos privados

configureCarousel

```
private fun configureCarousel (imageUrls: Array<String>, carouselAttr: HashMap<String, String?>?)
```

Si la lista de imágenes no está vacía va incorporando cada una de las imágenes a la lista del carrusel. Además, si hay valores de atributos, los configura.

Parámetros	
`imageUrls`	`Array<String>`: lista de imágenes del producto o categoría.
`carouselAttr`	`HashMap<String, String?>?`: array asociativo con los atributos de configuración disponibles del carrusel.

configureComponentsPosition

```
private fun configureComponentsPosition (carouselPosition: String?, componentsShown: Array<String>)
```

Organiza los componentes en el *layout* en función de la posición en la que se encuentra el carrusel.

Parámetros	
carouselPosition	String: posición del carrusel en el layout.
componentsShown	Array<String>: lista de los elementos que se van a mostrar en la pantalla.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el *layout* correspondiente al fragmento, pedir al *screen* que configure el fragmento, habilitar el cambio de opciones a la barra de acciones y pedir los datos de configuración al presentador.

Una vez obtenidos los datos, si hay actividad muestra el título correspondiente, se fija por defecto que se va a mostrar el título y la descripción y que la posición del carrusel es en la parte superior, si se obtienen datos se actualizan, y se configura cada componente del *layout* si se va a mostrar si no, se oculta.

Parámetros	
inflater	LayoutInflater: objeto que infla cualquier vista en el fragmento.
container	ViewGroup: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
savedInstanceState	Bundle: estado anteriormente guardado del fragmento.

Retorno	
View?	Vista correspondiente a la interfaz de usuario del fragmento.

3.2.7. CatalogHtml

Este controlador de pantalla se encarga de mostrar una página web.

CatalogHtmlViewModel

```
class CatalogHtmlViewModel: ViewModel()
```

Esta clase se encarga de conservar el estado de la pantalla *CatalogHtml*.

Resumen

Variables	
MutableLiveData<CatalogHtmlData>	liveData Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
void	setData (data: CatalogHtmlData?) Modifica el valor de la única variable de la clase.
CatalogHtmlData?	getData() Obtiene el valor de la única variable de la clase.
LiveData<CatalogHtmlData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogHtmlData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData (data: CatalogHtmlData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogHtmlData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData ()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogHtmlData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData ()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogHtmlData>	Única variable de la clase.

CatalogHtmlContract

```
interface CatalogHtmlContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en controlador *CatalogHtml*.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.
Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	presenter Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
LiveData<CatalogHtmlData>	fetchScreenData () Método que busca los datos necesarios para la configuración de la pantalla.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogHtmlData>	fetchScreenData (itemId: String?, structureType: String?) Método que obtiene los datos necesarios para la configuración de la pantalla.

Router

```
interface Router
```

Interfaz que define las variables y métodos del router.

Métodos	
void	passDataToNextScreen (itemId: Int, structureType: CatalogHtmlItemType, itemTypeId: String?) Método que se encarga de pasar los datos obtenidos por parámetro al siguiente controlador.
HashMap<String,String?>	getDataFromPreviousScreen () Método que se encarga de obtener los datos pasados por el anterior controlador.
void	navigateToNextScreen (screenType: String) Método que se encarga de cambiar al siguiente controlador.

CATALOGHTMLSCREEN

```
object CatalogHtmlScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla *CatalogHtml*.

Resumen

Métodos públicos	
`void`	`configureActivity(view: CatalogHtmlContract.View)` Configura e inyecta las dependencias siendo la vista una FragmentActivity.
`void`	`configureFragment(view: CatalogHtmlContract.View)` Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogHtmlContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogHtmlContract.View`: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogHtmlContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
`view`	`CatalogHtmlContract.View`: vista que se muestra.

CATALOGHTMLROUTER

```
class CatalogHtmlRouter: CatalogHtmlContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla *CatalogDetail*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.

Métodos heredados	
De la interfaz CatalogHtmlContract.Router	
HashMap<String,String>	getDataFromPreviousScreen () Obtiene el identificador de la categoría o producto a mostrar y qué es.
void	passDataToNextScreen (itemId: Int, itemType: CatalogHtmlItemType, itemTypeId: String?) Incorpora en el mediador información para el siguiente controlador.
void	navigateToNextScreen (screenType: String?) Inicia el siguiente controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

Métodos heredados

getDataFromPreviousScreen

```
override fun getDataFromPreviousScreen ()
```

Si puede obtener el mediador a partir de la actividad, obtiene el valor almacenado en la variable del identificador. Dependiendo del identificador que obtenga le asigna un tipo *product* o *category* y los incorpora en un array asociativo. Posteriormente reinicia todos los parámetros del mediador.

Retorno	
HashMap<String, String?>	Array asociativo que contiene el identificador de una categoría o producto y su tipo.

passDataToNextScreen

```
override fun passDataToNextScreen (itemId: Int, itemType: CatalogHtmlItemType, itemTypeId: String?)
```

Si puede obtener la actividad de fragmento y de esta el mediador, dependiendo de si recibe por parámetro una categoría o un producto almacena los identificadores pasados por parámetros en las correspondientes variables del mediador.

Parámetros	
itemId	Int: Identificador de una categoría o un producto.
itemType	CatalogHtmlItemType: Identifica si es una categoría o un producto.

itemTypeId	String?: Identificador de una categoría.

navigateToNextScreen

```
override fun navigateToNextScreen (screenType: String?)
```

Si puede obtener la actividad del fragmento, obtiene una instancia de controlador correspondiente al nombre pasado por parámetro y la inicializa, terminando la actividad actual.

Parámetros	
screentype	String: nombre de controlador.

CATALOGHTMLITEMTYPE

Esta clase es un tipo de datos especial que indica que la variable de tipo *CatalogHtmlItemType* debe ser igual a uno de los valores predefinidos, *CATEGORY* o *PRODUCT*. Indica si los datos obtenidos pertenecen a una categoría o un producto.

CATALOGHTMLDATA

```
data class CatalogHtmlData
```

Esta es una clase de datos contiene la información que configura la vista.

Variables	
String	id Identificador de un producto o una categoría.
CatalogHtmlItemType	type Indica si es un producto, una categoría o un elemento simple. Valor inicial: nulo.
String?	typeId

	Identificador de una categoría. Valor inicial: nulo
String?	title Nombre de categoría o producto. Valor inicial: nulo.
String?	description Descripción de categoría o producto. Valor inicial: nulo.
String?	html Texto HTML. Valor inicial: nulo.

CATALOGHTMLMODEL

```
class CatalogHtmlModel (): CatalogHtmlContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador *CatalogHtml*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity

	Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogHtmlData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogHtmlModel() Construye un objeto de la clase *CatalogHtmlModel*.

Métodos heredados	
De la interfaz CatalogHtmlContract.Model	
LiveData<CatalogHtmlData>	fetchScreenData (itemId: String?, structureType: String?) Obtiene los datos necesarios para el controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogHtmlData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogHtmlModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData (itemId: String?, structureType: String?)
```

Si obtiene un identificador por parámetro, obtiene el producto asociado y en el caso de que sea el identificador de una categoría obtiene su producto principal. Del producto obtiene los parámetros necesarios para crear el dato que configura la pantalla.

Parámetros	
itemId	String: Identificador de una categoría o un producto.
structureType	String: Indica si el identificador pertenece a un producto o una categoría.

Retorno	
LiveData<CatalogHtmlData>	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

CATALOGHTMLPRESENTER

```
class CatalogHtmlPresenter (): CatalogHtmlContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador *CatalogHtml*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogHtmlViewModel	viewModel Objeto que almacena el estado de la pantalla.
CatalogHtmlContract.Model	model Objeto del que se obtienen los datos que configuran la vista.
CatalogHtmlContract.Router	router Objeto que se encarga de la navegación entre pantallas.

Métodos heredados	
De la interfaz CatalogHtmlContract.Presenter	
LiveData<CatalogHtmlData>	fetchScreenData()

	Obtiene los datos necesarios para la configuración de la vista.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogHtmlViewModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogHtmlContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogHtmlContract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener los datos del *ViewModel* retorna esos datos, por el contrario, si puede obtener el contexto de la aplicación, le pide al router los datos del mediador y luego le pide al modelo los datos necesarios para la configuración de la vista del controlador pasando por parámetro los datos obtenidos del router.

Retorno	
LiveData<CatalogHtmlData>	Variable de la clase que contiene los datos necesarios para la configuración de la vista.

CATALOGHTMLACTIVITY

```
class CatalogHtmlActivity: AppCompatActivity(), CatalogHtmlContract.View
```

Esta clase se encarga de la detección y notificación de eventos, así como la actualización y gestión de interfaz de usuario del controlador *CatalogHtml*.

Resumen

Variables heredadas	
De la interfaz CatalogHtmlContract.View	
CatalogHtmlContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreate(savedInstanceState: Bundle?) Crea la vista.
De androidx.appcompat.app.AppCompatActivity	
void	onSupportNavigateUp ()

	Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogHtmlContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el *layout* correspondiente a la actividad, carga el fragmento correspondiente al controlador, establece los efectos de transición entre pantallas. Y si hay, configura la barra de acciones para que muestre la navegación hacia atrás.

Parámetros	
savedInstanceState	Bundle: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

onSupportNavigateUp

```
override fun onSupportNavigateUp ()
```

Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones haciendo uso del método *onBackPressed* heredado.

Retorno	
Boolean	Es verdadero si la navegación hacia arriba se completó correctamente y esta actividad se completó, falso en caso contrario.

CATALOGHTMLFRAGMENT

```
class CatalogHtmlFragment: Fragment(), CatalogHtmlContract.View
```

Esta clase se encarga del comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador *CatalogHtml*.

Resumen

Variables	
View	rootView Interfaz de usuario que implementa el fragmento.

Variables heredadas	
De la interfaz CatalogHtmlContract.View	
CatalogHtmlContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogDetailContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?,
savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el *layout* correspondiente al fragmento, pedir al *screen* que configure el fragmento, habilitar el cambio de opciones a la barra de acciones y pedir los datos de configuración al presentador.

Una vez obtenidos los datos, si hay actividad muestra el título correspondiente, se comprueba que si el dato html no es nulo se carga la página web como "*text/html* "codificado en UTF-8 (Unicode Transformation Format)[44]. En el caso de que sea nulo se carga la descripción del producto o la categoría como *"txt"* sin codificación.

Parámetros	
`inflater`	`LayoutInflater`: objeto que infla cualquier vista en el fragmento.
`container`	`ViewGroup`: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
`savedInstanceState`	`Bundle`: estado anteriormente guardado del fragmento.

Retorno	
`View`?	Vista correspondiente a la interfaz de usuario del fragmento.

3.2.8. CatalogImageViewer

Este controlador de pantalla se encarga de mostrar una imagen a la cual permite hacer zoom y compartirla.

CatalogImageViewerViewModel

```
class CatalogImageViewerViewModel: ViewModel()
```

Esta clase se encarga conservar el estado de la pantalla *CatalogImageViewer*.

Resumen

Variables	
MutableLiveData<CatalogImageViewerData>	liveData Datos necesarios para la configuración de la interfaz de usuario.

Métodos públicos	
void	setData (data: CatalogImageViewerData?) Modifica el valor de la única variable de la clase.
CatalogImageViewerData?	getData() Obtiene el valor de la única variable de la clase.
LiveData<CatalogImageViewerData>	getLiveData() Obtiene la única variable de la clase.

Variables

liveData

```
private var liveData: LiveData<CatalogImageViewerData>
```

Variable que almacena los datos necesarios para la configuración de la interfaz de usuario.

Valor inicial: nulo.

Métodos públicos

setData

```
fun setData (data: CatalogImageViewerData?)
```

Modifica el valor de la variable clase con el dato obtenido por parámetro.

Parámetros	
data	CatalogImageViewerData: dato que contiene la configuración de la interfaz de usuario.

getData

```
fun getData ()
```

Obtiene el valor de la variable única de la clase y lo retorna.

Retorno	
CatalogImageViewerData?	Valor correspondiente a la única variable de la clase.

getLiveData

```
fun getLiveData ()
```

Obtiene la variable de la clase y la retorna.

Retorno	
LiveData<CatalogImageViewerData>	Única variable de la clase.

CatalogImageViewerContract

```
interface CatalogImageViewerContract
```

Esta interfaz se encarga de definir y separar las funcionalidades de los módulos de vista, presentador, modelo y router implicados en controlador *CatalogImageViewer*.

Resumen

Interfaces	
View	Define las variables y métodos que deben estar presentes en la vista.
Presenter	Define las variables y métodos que deben estar presentes en el presentador.
Model	Define las variables y métodos que deben estar presentes en el modelo.
Router	Define las variables y métodos que deben estar presentes en el router.

Interfaces

View

```
interface View
```

Interfaz que define las variables y métodos de la vista.

Variables	
Presenter	presenter Presentador asociado a la vista.

Presenter

```
interface Presenter
```

Interfaz que define las variables y métodos del presentador.

Métodos	
LiveData<CatalogImageViewerData>	fetchScreenData (intent: Intent) Método que busca los datos necesarios para la configuración de la pantalla.

Model

```
interface Model
```

Interfaz que define las variables y métodos del modelo.

Métodos	
LiveData<CatalogImageViewerData>	fetchScreenData (intent: Intent) Método que obtiene los datos necesarios para la configuración de la pantalla.

CATALOGIMAGEVIEWERSCREEN

```
object CatalogImageViewerScreen
```

Esta clase es la encargada de configurar e inyectar las dependencias. Es decir, conecta los distintos módulos de los que se compone el controlador de pantalla *CatalogImageViewer*.

Resumen

Métodos públicos	
void	configureActivity(view: CatalogImageViewerContract.View) Configura e inyecta las dependencias siendo la vista una FragmentActivity.
void	configureFragment(view: CatalogImageViewerContract.View) Configura e inyecta las dependencias siendo la vista un Fragment.

Métodos públicos

configureActivity

```
fun configureActivity (view: CatalogImageViewerContract.View)
```

A partir de la vista que corresponde con una FragmentActivity, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el

router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
view	CatalogImageViewerContract.View: interfaz de usuario que se muestra.

configureFragment

```
fun configureFragment (view: CatalogImageViewerContract.View)
```

A partir de la vista que corresponde con un Fragment, crea el router, presentador y modelo asignándoles mediante una conexión leve la vista. Además, al presentador se le asigna el router y modelo anteriormente inicializados y el *ViewModel* que lo obtiene mediante ViewModelProvider. Finalmente, se le indica a la vista su presentador.

Parámetros	
view	CatalogImageViewerContract.View: vista que se muestra.

CATALOGIMAGEVIEWERROUTER

```
class CatalogImageViewerRouter: CatalogImageViewerContract.Router
```

Esta clase se encarga de la navegación hacia/desde el controlador de pantalla *CatalogImageViewer*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

CATALOGIMAGEVIEWERDATA

```
data class CatalogImageViewerData
```

Esta es una clase de datos contiene la información que configura la vista.

Variables	
HashMap<*,*>	item Array asociativo

CATALOGIMAGEVIEWERMODEL

```
class CatalogImageViewerModel (): CatalogImageViewerContract.Model
```

Esta clase se encarga de la gestión, almacenamiento y mantenimiento de datos necesarios en el controlador *CatalogImageViewer*.

Resumen

Variables	
WeakReference<Fragment>?	fragment

	Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogRepository	repository Repositorio del que se obtendrán los datos.
LiveData<CatalogDetailData>	liveData Objeto en el que se almacenan los datos.

Constructores
CatalogImageViewerModel() Construye un objeto de la clase *CatalogImageViewerModel*.

Métodos heredados	
De la interfaz CatalogImageViewerContract.Model	
LiveData<CatalogImageViewerData>	fetchScreenData (intent: Intent) Obtiene los datos necesarios para el controlador.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

respository

```
private var repository: CatalogRepository
```

Repositorio del que se van a obtener los datos.

liveData

```
private val liveData: MutableLiveData<CatalogImageViewerData>
```

Es una variable que puede ser observada, en la que se va a almacenar los datos requeridos por el controlador.

Valor inicial: nulo.

Constructores

CatalogImageViewerModel

```
init
```

Si puede obtener el contexto de la actividad obtiene la instancia del repositorio actualizando la variable repository.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData (intent: Intent)
```

Obtiene los extras del Intent pasado por parámetro, si hay extrae la URL de la imagen y crea el dato que configura la pantalla.

Parámetros	
item	Intent: Contenedor con la actividad.

Retorno	
LiveData<CatalogImageViewerData>	Variable de la clase que contiene los datos necesarios para la configuración del controlador.

CATALOGIMAGEVIEWERPRESENTER

```
class CatalogImageViewerPresenter (): CatalogImageViewerContract.Presenter
```

Esta clase se encarga de la gestión de eventos procedentes de la vista y la gestión de datos con la vista y el modelo necesarios en el controlador *CatalogImageViewer*.

Resumen

Variables	
WeakReference<Fragment>?	fragment Fragmento correspondiente a la vista.
WeakReference<FragmentActivity>?	activity Actividad correspondiente a la vista.
CatalogImageViewerModel	viewModel Objeto que almacena el estado de la pantalla.
CatalogImageViewerContract.Model	model Objeto del que se obtienen los datos que configuran la vista.
CatalogImageViewerContract.Router	router Objeto que se encarga de la navegación entre pantallas.

Métodos heredados	
De la interfaz CatalogImageViewerContract.Presenter	
LiveData<CatalogImageViewerData>	fetchScreenData (intent: Intent)

	Obtiene los datos necesarios para la configuración de la vista.

Variables

fragment

```
var fragment: WeakReference<Fragment>?
```

Representa una referencia leve a una parte de la interfaz de usuario.

Valor inicial: nulo.

activity

```
var activity: WeakReference<FragmentActivity>?
```

Representa una referencia a la actividad única y enfocada que el usuario puede hacer.

Valor inicial: nulo.

viewModel

```
lateinit var viewModel: CatalogImageViewerModel
```

Representa el estado de la vista.

model

```
lateinit var model: CatalogImageViewerContract.Model
```

Representa el modelo asociado al presentador.

router

```
lateinit var router: CatalogImageViewerContract.Router
```

Representa el router asociado al presentador.

Métodos heredados

fetchScreenData

```
override fun fetchScreenData ()
```

Si puede obtener los datos del *ViewModel* retorna esos datos, por el contrario, si puede obtener el contexto de la aplicación y luego le pide al modelo los datos necesarios para la configuración de la vista del controlador pasando por parámetro los datos obtenidos de la vista.

Retorno	
LiveData<CatalogImageViewerData>	Variable de la clase que contiene los datos necesarios para la configuración de la vista.

CATALOGIMAGEVIEWERACTIVITY

```
class CatalogImageViewerActivity: AppCompatActivity(), CatalogImageViewerContract.View
```

Esta clase se encarga de la detección y notificación de eventos, así como la actualización y gestión de interfaz de usuario del controlador *CatalogImageViewer*.

Resumen

Variables heredadas	
De la interfaz CatalogImageViewerContract.View	
CatalogImageViewerContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Activity a través de androidx.appcompat.app.AppCompatActivity	
void	onCreate(savedInstanceState: Bundle?) Crea la vista.
void	onCreateOptionsMenu (menu: Menu)

	Inicializa el contenido del menú de opciones estándar de la Actividad.
Boolean	onOptionsItemSelected (item: MenuItem) Se llama cuando un elemento de la barra de acciones es pulsado para gestionar el evento.
De androidx.appcompat.app.AppCompatActivity	
void	onSupportNavigateUp () Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogImageViewerContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreate

```
override fun onCreate (savedInstanceState: Bundle?)
```

Se encarga de la inicialización de la vista, carga el *layout* correspondiente a la actividad, carga el fragmento correspondiente al controlador, establece los efectos de transición entre pantallas. Y si hay, configura la barra de acciones para que muestre la navegación hacia atrás.

Parámetros	
savedInstanceState	Bundle: si la actividad se reinicializa después de que se cerró previamente, este paquete contiene los datos que proporcionó más recientemente.

onCreateOptionsMenu

```
override fun onCreateOptionsMenu (menu: Menu)
```

Infla el menú de acciones de la aplicación con el menú correspondiente.

Parámetros	
menu	Menu: menú de opciones en el que coloca los elementos.

Retorno	
Boolean	Debe devolver verdadero para que se muestre el menú; si devuelve falso, no se mostrará.

onOptionsItemSelected

```
override fun onOptionsItemSelected (item: MenuItem)
```

Si se pulsa la opción de compartir obtiene los datos de la imagen mostrada a partir de su etiqueta y crea un *Intent* para compartirla.

Parámetros	
item	MenuItem: elemento del menú de la barra superior de navegación seleccionado.

Retorno	
Boolean	Devuelve falso para permitir que continúe el procesamiento normal del menú, verdadero para consumirlo aquí.

onSupportNavigateUp

```
override fun onSupportNavigateUp ()
```

Navega hacia arriba dentro de la jerarquía de actividades de la aplicación desde la barra de acciones haciendo uso del método *onBackPressed* heredado.

Retorno	
Boolean	Es verdadero si la navegación hacia arriba se completó correctamente y esta actividad se completó, falso en caso contrario.

CATALOGIMAGEVIEWERFRAGMENT

```
class CatalogImageViewerFragment: Fragment(), CatalogImageViewerContract.View
```

Esta clase se encarga comportamiento correspondiente a la interfaz de usuario del fragmento incorporado en la actividad del controlador *CatalogImageViewer*.

Resumen

Variables	
View	rootView Interfaz de usuario que implementa el fragmento.

Variables heredadas	
De la interfaz CatalogImageViewerContract.View	
CatalogImageViewerContract.Presenter	presenter Presentador asociado a la vista.

Métodos heredados	
De android.app.Fragment	
View?	onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View? Crea la parte de la vista correspondiente al fragmento de la actividad.

Variables

rootview

```
lateinit var rootView: View
```

Interfaz de usuario que implementa el fragmento.

Variables heredadas

presenter

```
override lateinit var presenter: CatalogImageViewerContract.Presenter
```

Representa el presentador asociado a la vista.

Métodos heredados

onCreateView

```
override fun onCreateView (inflater: LayoutInflater, container: ViewGroup?, savedInstanceState: Bundle?): View?
```

Se encarga de inflar la vista con el *layout* correspondiente al fragmento, pedir al *screen* que configure el fragmento y habilitar el cambio de opciones a la barra de acciones. Si obtiene el *Intent* de la actividad y tiene extras, obtiene la transición, se la incorpora a la vista de imagen de la pantalla y pide los datos de configuración al presentador.

Una vez obtenidos los datos carga la imagen.

Parámetros	
`inflater`	`LayoutInflater`: objeto que infla cualquier vista en el fragmento.
`container`	`ViewGroup`: vista principal a la que se debe adjuntar la interfaz de usuario del fragmento.
`savedInstanceState`	`Bundle`: estado anteriormente guardado del fragmento.

Retorno	
`View`?	Vista correspondiente a la interfaz de usuario del fragmento.

CAPÍTULO VI. GUÍA DE USUARIO

En este capítulo se detalla un manual para el empleo del nuevo IUMATI Framework.

1. NUEVO IUMATI FRAMEWORK

El nuevo IUMATI Framework es un *framework* que permite a cualquier tipo de usuario, tanto con experiencia en programación como sin ella, crear aplicaciones móviles nativas de Android desde la versión 8.0 (API 26), comúnmente denominada Oreo, a la última versión disponible en el mercado, la versión 11 (API 30).

Este *framework* estructura los datos en forma de catálogo de categorías y productos y a partir de él configura una aplicación Android, lo que permite una alta escalabilidad en la aplicación. En esencia, el catálogo es una lista donde las categorías representan la jerarquía de navegación de la aplicación mientras que los productos representan el contenido de la pantalla. Esta idea se refleja de una forma más gráfica en la jerarquía de ejemplo mostrada en la Figura 28.

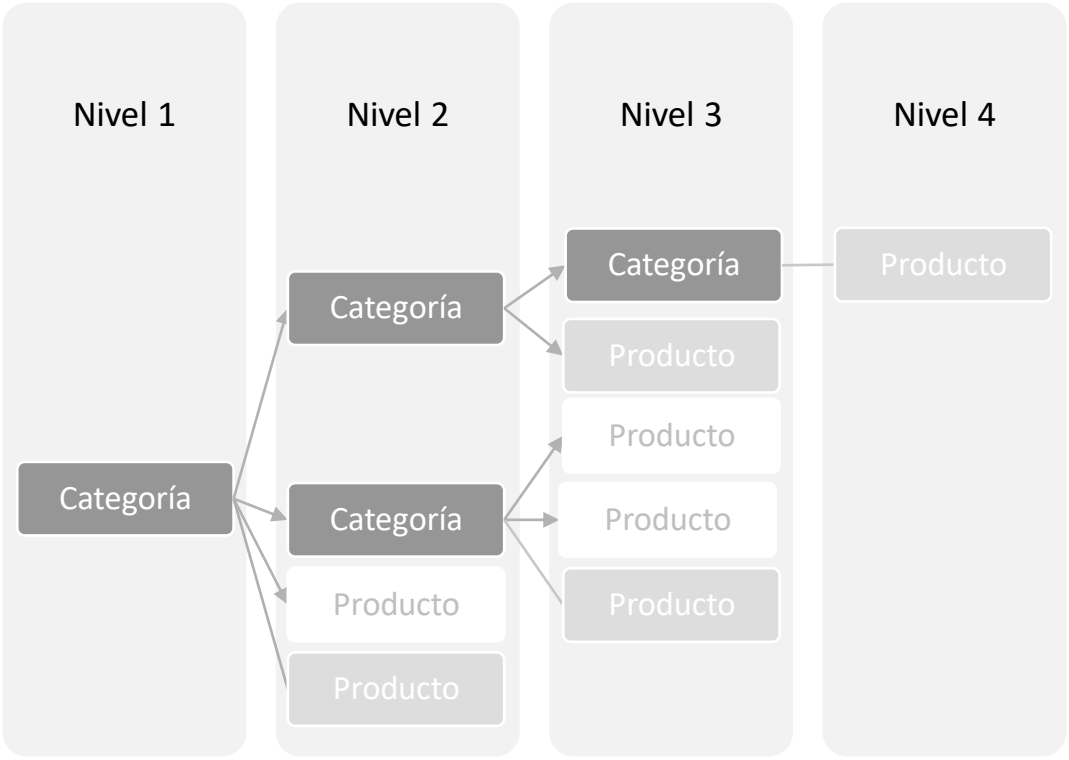

Figura 28. Ejemplo de jerarquía de un catálogo de categorías (gris oscuro) y productos (gris claro y blanco)

Como se puede ver en este ejemplo, hay cuatro niveles en la jerarquía de la aplicación, cuatro categorías y siete productos, dando un total de siete pantallas en la aplicación. La categoría del primer nivel navega entre las categorías y productos del segundo nivel y así sucesivamente tantos niveles como el usuario desee, se puede ver de forma más detallada en la Figura 29.

Asimismo, a nivel de términos específicos del *framework*, los productos que representen la pantalla de una categoría deben tener como nombre “Main Product” con el objetivo de diferenciarlos del resto de productos de una categoría.

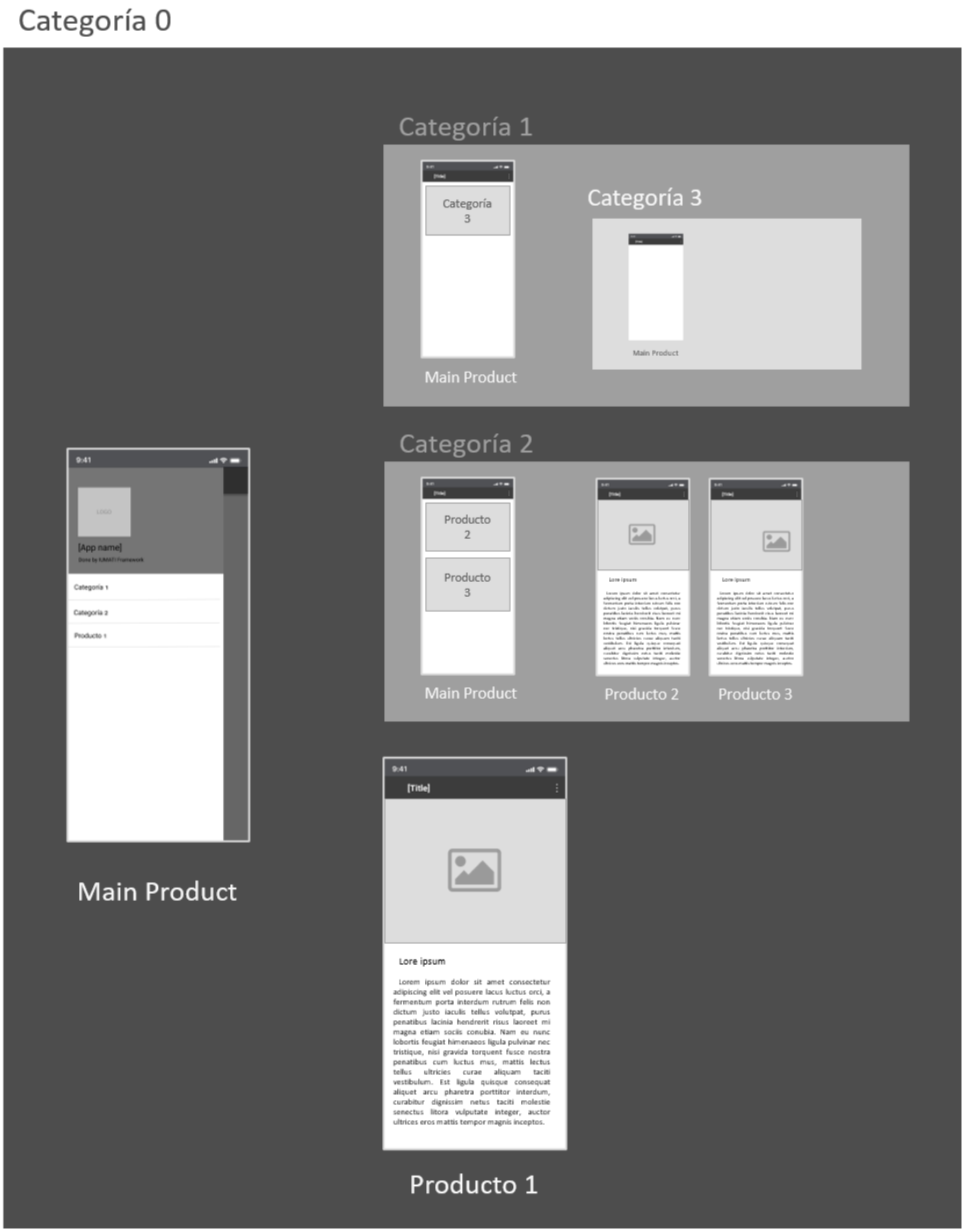

Figura 29. Ejemplo de jerarquía detallada de un catálogo de categorías y productos

Una vez indicado que son las categorías y los productos que emplea este *framework* y como se relacionan, la Figura 30 explica que elementos hay dentro de cada uno, primero indicando el nombre y luego lo que representa.

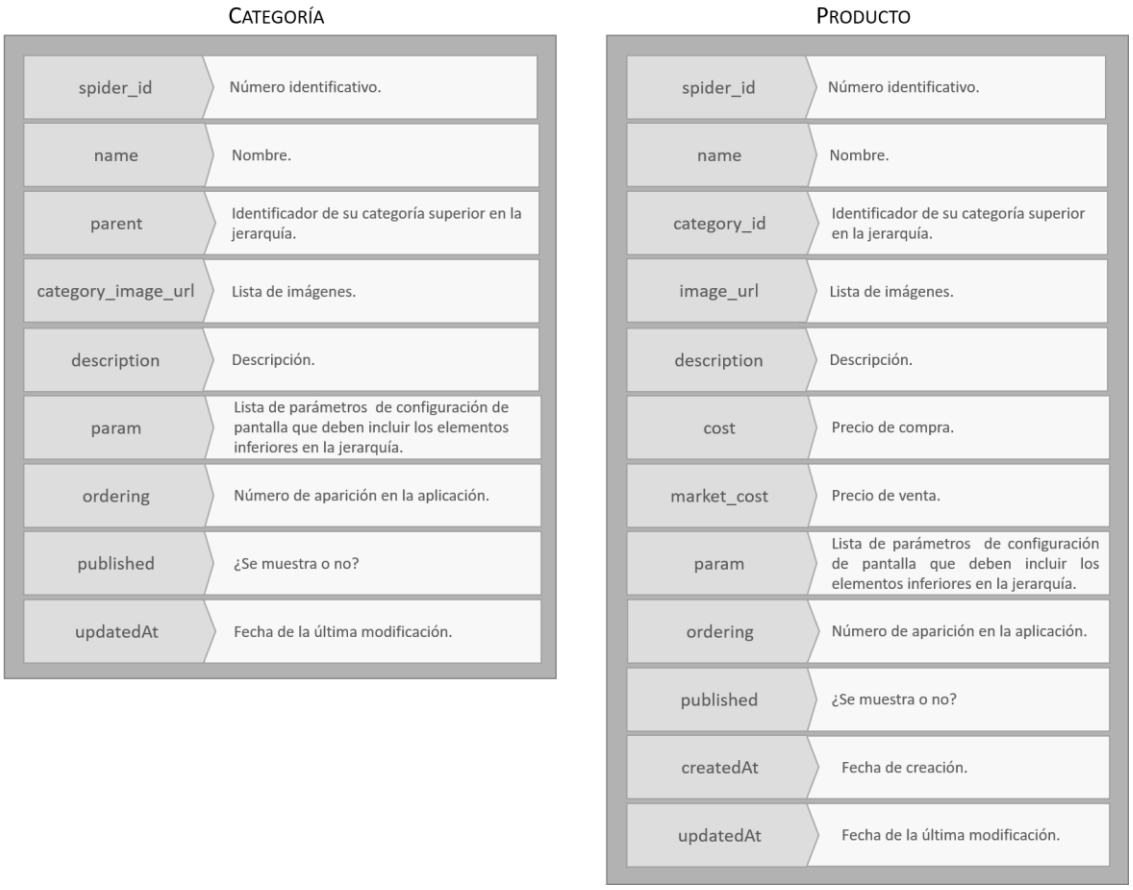

Figura 30. Elementos que hay dentro de una categoría y de un producto

2. FUNCIONAMIENTO

Una vez conocidas las características de los datos que maneja el *framework* su funcionamiento se explica en unos sencillos pasos.

En primer lugar, el usuario haciendo uso de la interfaz web del IUMATI Framework, desarrollada en WordPress con el plugin *SpiderCatalog,* diseña mediante una interfaz gráfica y sencilla su aplicación. A continuación, la interfaz mapeará los datos a un catálogo de categorías y productos que posteriormente, IUMATI Framework pedirá al servidor y se lo enviará en formato JSON.

Una vez los datos se encuentran dentro de *framework*, este pasa los datos del catálogo a una base de datos alojada en el dispositivo móvil. Finalmente, al ejecutar la aplicación e ir obteniendo la información de la base de datos el *framework* elegirá los controladores correspondientes, dependiendo de los parámetros del catálogo, para mostrarlos en la pantalla del móvil. Todo este procedimiento se muestra de forma gráfica en la Figura 31.

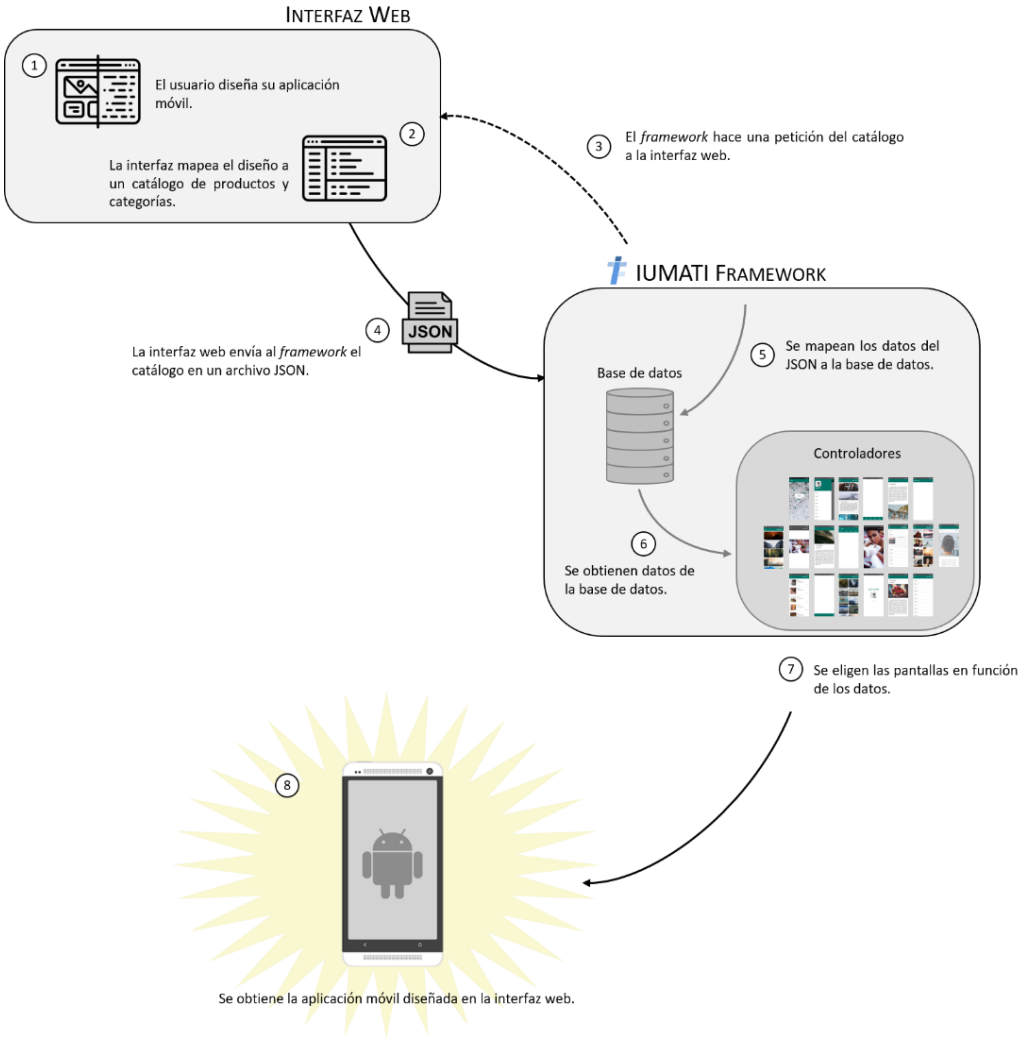

Figura 31. Funcionamiento del IUMATI Framework

3. PANTALLAS

Antes de empezar a utilizar el *framework* hay que saber las configuraciones de pantalla de las que dispone. Actualmente IUMATI Framework cuenta con ocho controladores de pantalla diferentes, de los cuales seis son modificables por parte del usuario.

3.1. PANTALLAS NO MODIFICABLES

Los controladores detallados en esta sección no son modificables por motivos de funcionalidad del *framework*, es decir, cumplen una determinada función que no está sujeta a cambios. Por tanto, no son configurables por el usuario.

3.1.1. SPLASH

Esta pantalla por excelencia es la primera en aparecer en cualquier aplicación creada con el nuevo IUMATI Framework, en ella se configura toda la aplicación. En otros términos, se lee el archivo JSON que contiene las categorías y productos, se crea la base de datos y se arranca la aplicación. Por ende, su diseño es claro y sencillo para que no desentone con la personalización del usuario en el resto de las pantallas de la aplicación.

Tal y como se puede ver en los ejemplos de la Figura 32 esta pantalla cuenta con el nombre de la aplicación y su logo, aparte de una barra de progreso para indicar al usuario que la aplicación se está configurando.

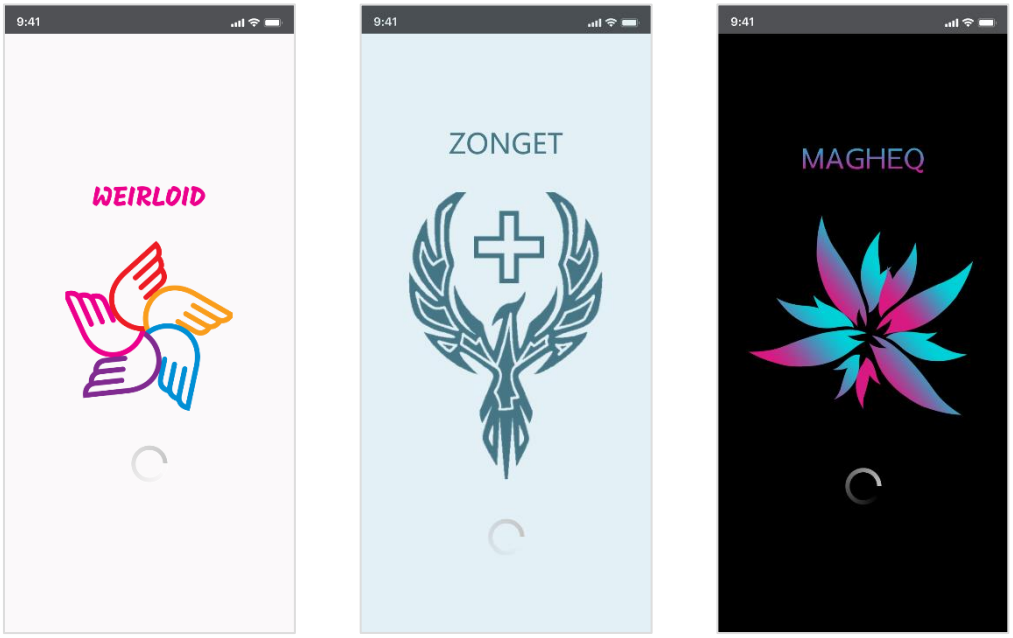

Figura 32. Ejemplos de pantalla Splash

3.1.2. IMAGE VIEWER

Este controlador de pantalla es un visor básico que se puede incluir en cualquier tipo de aplicación que el usuario desee. No es modificable por el usuario, pero su diseño minimalista lo hace encajar muy bien con cualquier estilo.

Además de cumplir con las características básicas de un visor de imágenes tiene incorporada la funcionalidad de compartir en la barra de navegación superior, como se puede ver en las imágenes de la Figura 33, lo que implica que permite enviar cómodamente cualquier imagen que se esté visualizando.

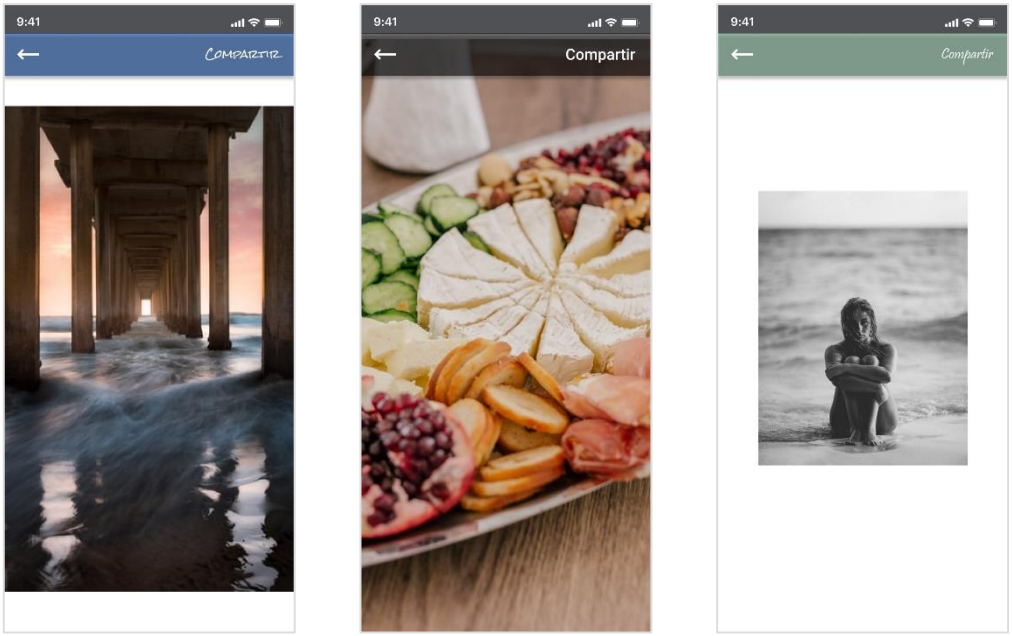

Figura 33. Ejemplos de pantallas Image Viewer

3.2. PANTALLAS MODIFICABLES

IUMATI Framework cuenta con dos pantallas que no son explícitamente configurables por el usuario como se vio en el apartado anterior, pero eso no limita este *framework* que tiene un total de seis controladores de pantalla más completamente modificables por el usuario, lo que le permite una cantidad enorme de combinaciones de estilos como veremos a continuación.

3.2.1. HOME

Este controlador se encarga de la navegación principal de la aplicación, por lo cual, para que la mayor parte de usuarios encuentren la navegación ideal IUMATI Framework cuenta con los cinco principales tipos de navegación y la combinación de dos de ellos, como se puede ver en la Figura 34 con sus respectivos nombres. Por tanto, se dispone de seis tipos de navegación principal, de los cuales, el usuario escoge uno mediante el parámetro *navType*.

A la hora de escoger la navegación, exceptuando el menú lateral que es el tipo de navegación principal escogido por defecto, el resto tiene un nombre que los identifica:

- Menú lateral.
- *abTop*: Barra de navegación superior.
- *abBottom*: Barra de navegación inferior.
- *navBottom*: Navegación inferior.

- *tabs*: Pestañas.
- *abTopNavBottom*: Barra de navegación superior y navegación inferior.

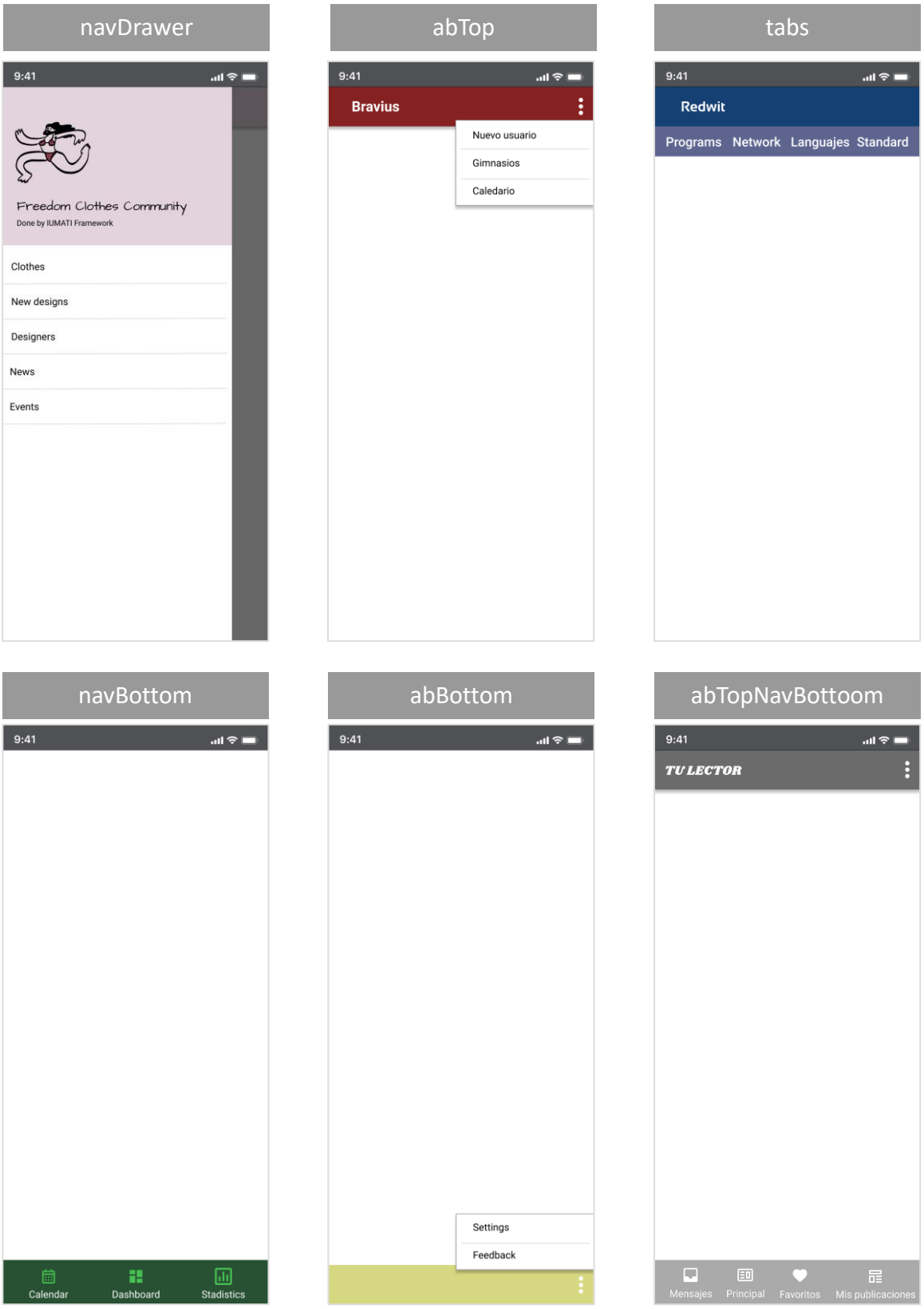

Figura 34. Tipos de navegaciones en la pantalla Home

Asimismo, como se puede intuir viendo la Figura 34 cada tipo de navegación tiene su propio menú de opciones de navegación. En el caso de la combinación *abTopNavBottom*, el menú de opciones se divide, correspondiendo dos elementos al menú de la barra superior si el menú tiene menos de cinco opciones y cuatro en el caso contrario, el resto lo incorpora la navegación inferior.

3.2.2. LIST

Esta pantalla es una de las más versátiles del *framework* ya que dispone de múltiples elementos de configuración. La pantalla, como su propio nombre indica, muestra una lista de elementos que si el usuario lo desea puede indicar que sean expandibles con el parámetro *expandable*, tal y como se puede ver en la Figura 35.

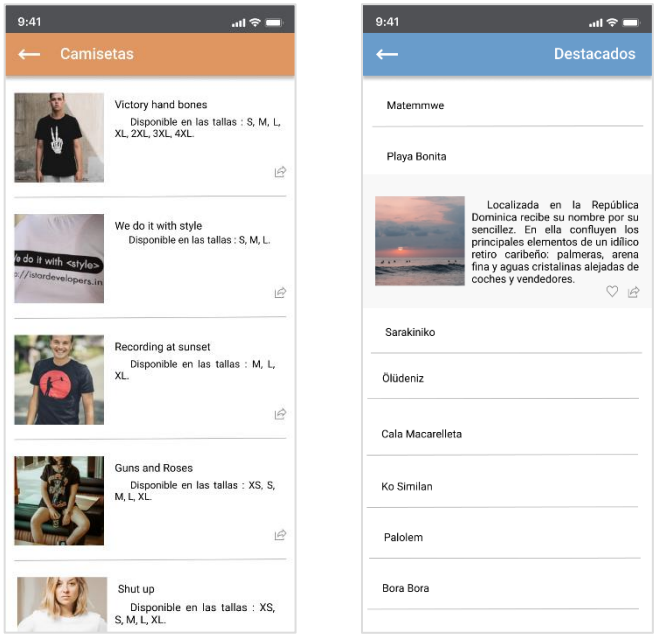

Figura 35. Pantalla List no expandible y expandible

Por otro lado, cada elemento contiene una imagen, un título, un subtítulo que corresponde con su fecha de creación y una descripción del mismo. Además, cuenta con las funcionalidades de destacados y compartir. Por tanto, todos los componentes y funcionalidades nombradas en este párrafo pueden ser mostradas o no según indique el usuario en el parámetro *componentsShown*, esto da un gran número de combinaciones posibles, algunas de ellas se muestran en la Figura 36. Asimismo, con el parámetro *itemList*, el usuario puede incorporar una lista a un producto, es decir, en vez de crear una categoría cuya pantalla sea una lista y se muestren sus productos asociados, crear un producto con una lista de nombres como el primer ejemplo de la Figura 36.

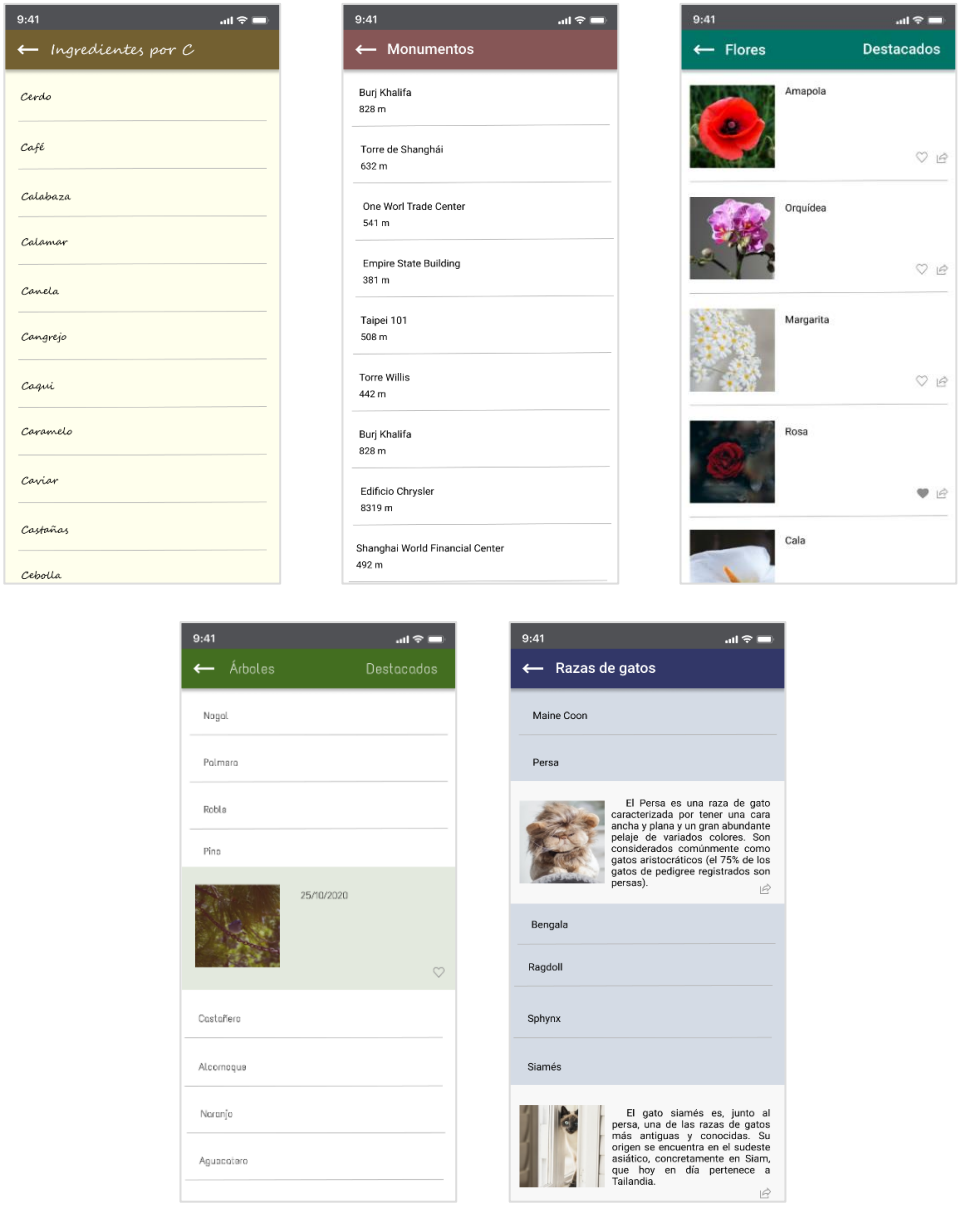

Figura 36. Ejemplos de la pantalla List

3.2.3. GALLERY

Este controlador de pantalla utiliza la misma estructura que la pantalla anterior, incorpora los mismos elementos y se configura con los mismos parámetros, pero adaptado a una vista de tarjetas. Aparte de la diferencia obvia, las pantallas tienen los componentes dispuestos de diferente forma con distinto diseño, en este controlador el usuario puede seleccionar cuantas tarjetas desea en cada fila con el parámetro *spanCount*.

A continuación, se muestran ejemplos de diferentes configuraciones en la Figura 37.

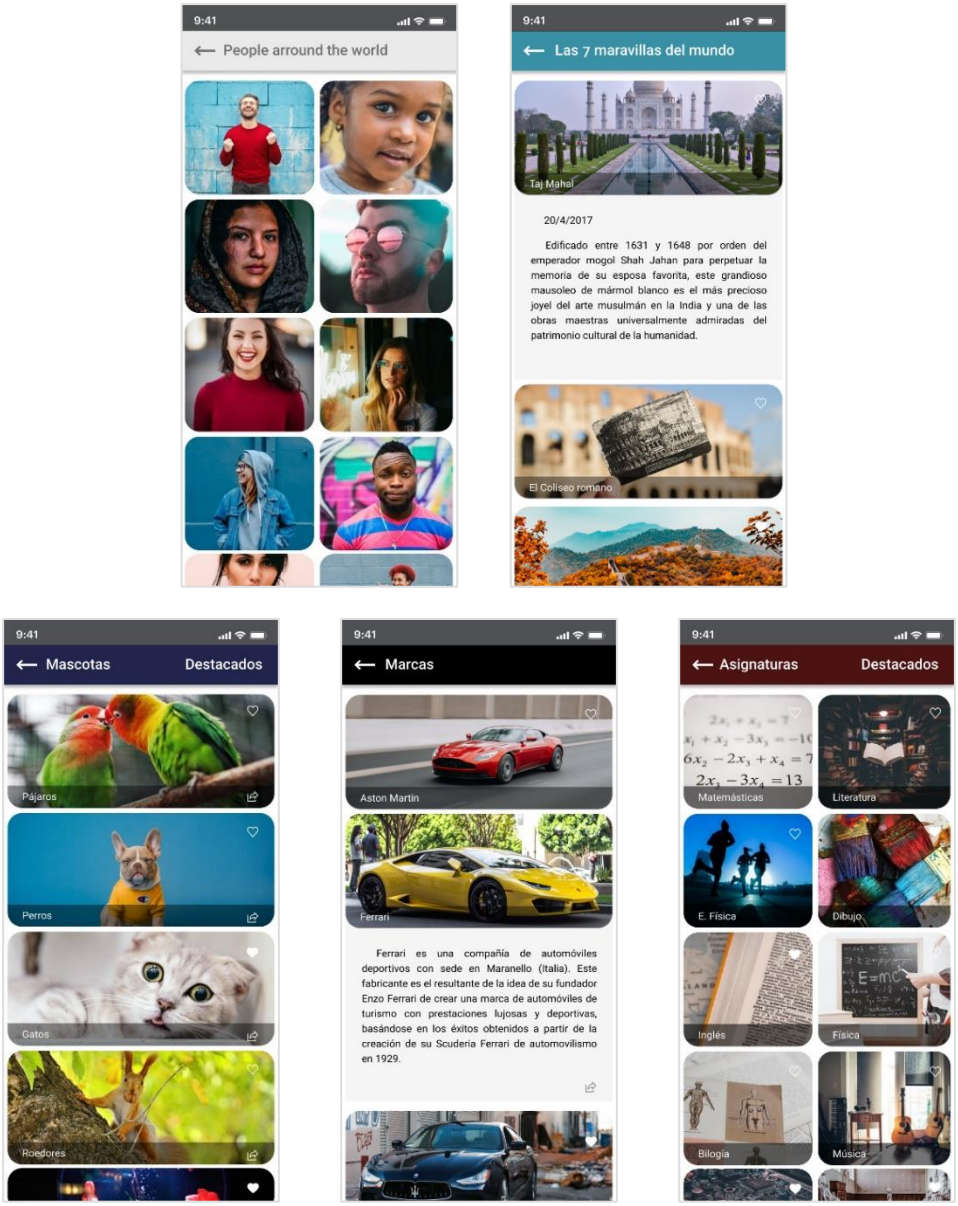

Figura 37. Ejemplos de la pantalla Gallery

3.2.4. DETAIL

Al igual que los controladores anteriores, el controlador de detalle es configurable por el usuario. Este decide si mostrar el título, el subtítulo, carrusel de imágenes o descripción que componen la pantalla con el parámetro *componentsShown*. Además de lo anterior, puede configurar donde colocar el carrusel, como se puede ver en la Figura 38, o configurar características del este a través

del parámetro *carouselAttr* como mostrar los indicadores o las flechas de navegación, y hasta elegir si rotan automáticamente las imágenes.

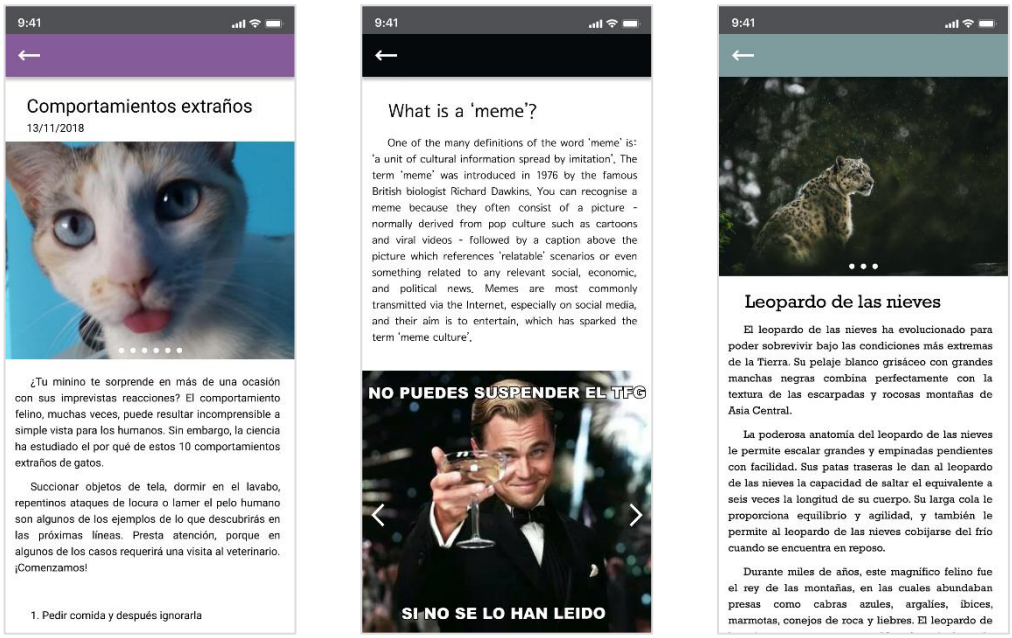

Figura 38. Ejemplos de la pantalla Detail

3.2.5. MAP

Tal y como se puede ver en la Figura 39, IUMATI Framework cuenta con una pantalla mapa en la que el usuario puede incorporar todas las localizaciones que desee con el parámetro *location*.

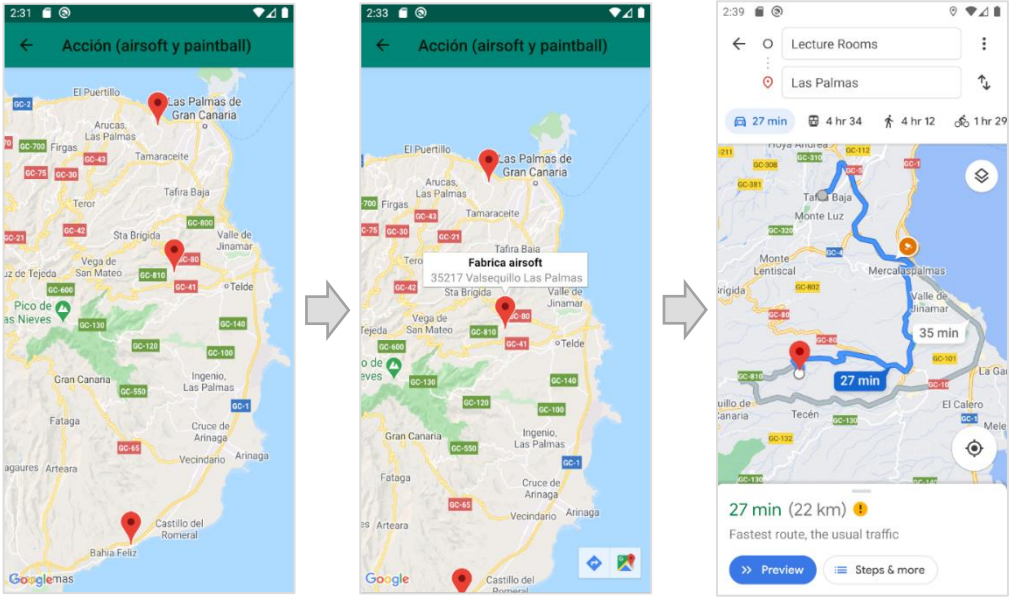

Figura 39. Ejemplo pantalla Map

Este controlador como se puede ver claramente en la Figura 39, está implementado con Maps SDK[45] para Android de Google, lo que permite de forma fácil y sencilla, aparte de visualizar las localizaciones, es acceder directamente a los mapas de Google y aprovechar toda la funcionalidad que estos aportan.

3.2.6. HTML

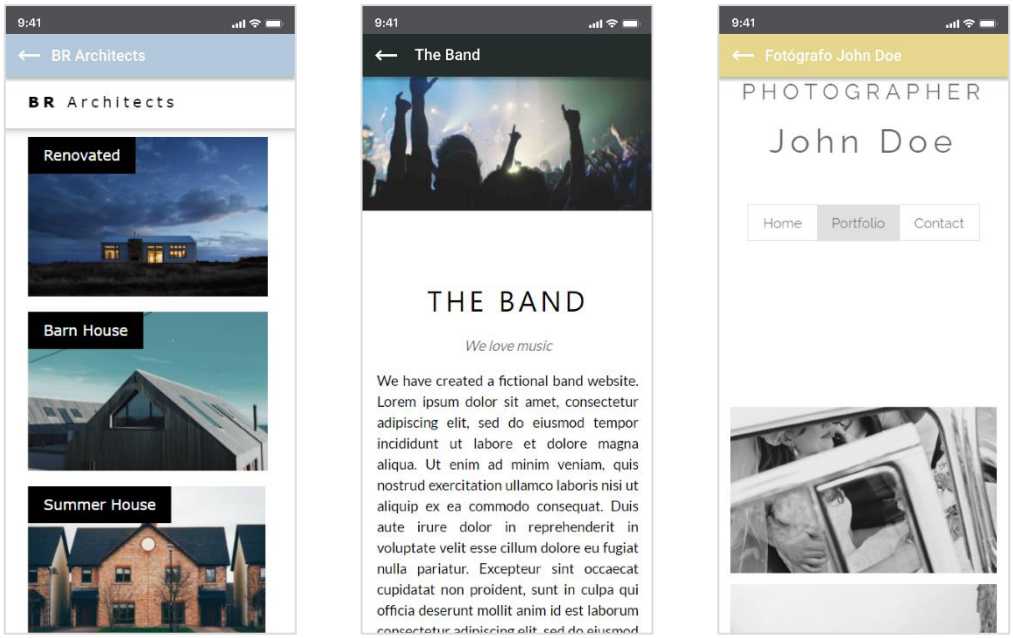

Figura 40. Ejemplos de pantalla HTML con *templates* de W3Schools[45]

El último controlador de IUMATI Framework es el denominado HTML. Su nombre de debe a que el usuario mediante el parámetro *html* puede pasarle un código HTML, es decir, una página web escrita en el lenguaje HTML como las mostradas en la Figura 40.

En conclusión, si lo que pasa el usuario por parámetro es un código HTML este tiene infinitas posibilidades de estilo ya que el HTML consta de otros componentes y otras formas de crear *layouts* diferentes. Por tanto, esta pantalla muestra todo lo que el usuario quiera ya que es él mismo el que crea el archivo HTML.

4. PARÁMETROS DE CONFIGURACIÓN

Ya introducidos todos los controladores de los que dispone IUMATI Framework y nombrar con qué parámetros se configuran, este apartado explica mediante la Tabla 3, cada uno de los valores que pueden tomar los parámetros que configuran cada una de las pantallas modificables.

Parámetros de IUMATI Framework	
Común a todos los controladores	
type	Este es el tipo de controlador a mostrar, si no se indica toma por defecto el valor *detail*. Los valores que puede tomar son: – *home* – *detail* – *html* – list – *map* – *imageViewer* – *gallery* **"par_type@@:@@map\t"**
outstanding	Indica si la categoría o producto pertenece a la categoría de destacados en el caso true o no en el caso false (por defecto). **"par_outstanding@@:@@true\t"**
Controlador Home	
navType	Indica que tipo de navegación se emplea en la pantalla. Por defecto el menú lateral es la navegación escogida, pero se puede elegir entre los siguientes valores: – *abTop*: Barra de navegación superior. – *abBottom*: Barra de navegación inferior. – *navBottom*: Navegación inferior. – *tabs*: Pestañas. – *abTopNavBottom*: Barra de navegación superior y navegación inferior. **"par_navType@@:@@abBottom\t"**

Controlador List	
componentsShown	Parámetro que indica qué componentes de la vista se van a mostrar. Por defecto, si la lista no es expandible sólo se muestra el nombre del elemento, si es expansible, el título y la descripción. Sin embargo, se puede escoger múltiples combinaciones con los siguientes elementos. - *title*: Nombre del elemento. - *subtitle*: Fecha de la última modificación del elemento. - *image*: Imagen representativa del elemento. - *description*: Descripción del elemento. - *outstandingIcon*: Muestra un icono de destacado y habilita la funcionalidad de mostrar destacados o no. - *shareButton*: Botón que implementa la funcionalidad de compartir. **"par_componentsShown@@:@@title,image,outstandingIcon\t"**
expandable	Indica si la lista es expandible o no tomando los valores *true* para el caso afirmativo y *false* para el caso negativo, siendo por defecto este último. **"par_expandable@@:@@false\t"**
itemList	Este parámetro permite crear una lista básica de sólo nombres con los datos pasados de forma rápida y sencilla. Evitando la creación de una categoría, cuya pantalla sea una lista, y sus categorías y productos asociados. **"par_itemList@@:@@500 g. Bacon, 1 Kg Papas, 200 g. Queso\t"**
Controlador Gallery	
spanCount	Representa el número de tarjetas a mostrar en cada fila de la vista de tarjetas. Su valor por defecto es uno, por lo que sólo se mostrará una tarjeta a no ser que se especifique lo contrario.

	“par_spanCount@@:@@2\t”
componentsShown	Se configura exactamente igual que el parámetro de igual nombre del controlador List.
expandable	Indica si la tarjeta es expandible o no tomando los valores *true* para el caso afirmativo y *false* para el caso negativo, siendo por defecto este último. Asimismo, esta opción sólo se permite cuando se muestra una tarjeta por fila. **“par_expandable@@:@@false\t”**
Controlador Detail	
componentsShown	Indica qué componentes de la pantalla se van a mostrar. Por defecto sólo se muestra el título y la descripción de un producto o categoría. Se puede elegir las combinaciones entre los cuatro componentes presentes en el controlador: - *title*: Nombre de la categoría o producto. - *subtitle*: Fecha de la última modificación de la categoría o producto. - *image*: carrusel de imágenes de la categoría o producto. - *description*: Descripción de la categoría o producto. **“par_componentsShown@@:@@title,image, description\t”**
carouselAttr	Este parámetro configura de carousel de imágenes mostrado en la pantalla. Para que se aplique la configuración debe tener exactamente los valores correspondientes a los siguientes campos: - *position*: Posición del carrusel en la vista. Puede tomar los valores *top* (por defecto), *center* y *bottom*. - *type*: Indica las imágenes del carrusel abarcan todo el ancho de la pantalla (valor *block*) o no (valor *showcase*). - *showcaption*: *true* para mostrar el número de imagen, *false* en caso contrario.

	- *showIndicator*: *true* si muestran los indicadores en parte inferior del carrusel, *false* en el caso contrario. - *showNavigationButtons*: indica si se muestran las flechas de navegación *true*, false en el caso contrario. - *autoplay*: *true* si las imágenes se cambian automáticamente en el carrusel, *false* en el caso contrario. **"par_carouselAttr@@:@@center,block,false,true,true,false\t"**
Controlador Map	
location	Este parámetro representa distintas localizaciones de un mapa. Para ello, se indican el nombre de la localización, la dirección y las coordenadas geográficas de cada una de ellas. **"par_location@@:@@[Vaqueria Airsoft GC, Calle Escritora Carmen Martín Gaite - n69 (35010) Ladera Alta - Las Palmas, 28.123147, -15.478113][Hangar 37, (35107) Tarajalillo - Las Palmas, 27.786303, -15.504741][Fabrica airsoft, (35217) Valsequillo - Las Palmas, 28.004957, -15.464008]\t"**
Controlador HTML	
html	Indica el código escrito en lenguaje HTML a incorporar en la pantalla. **"par_html@@:@@<!DOCTYPE HTML><html><head> <title>Cómo hacer una página web con HTML</title></head> <body> Cómo hacer una página web con HTML En el post de hoy voy a enseñarte cómo hacer una página web con HTML, pero antes … Conceptos básicos sobre páginas web ¿Cuál es entonces la diferencia entre una página web y un sitio web?… Diferencias entre una página web y un sitio web Una página web es un único documento electrónico que… </body></html>\t"**

Tabla 3. Parámetros de configuración de controladores

5. COMIENZA A CREAR UNA APLICACIÓN

Una vez conocidos todos los entresijos de IUMATI Framework se puede comenzar a utilizarlo, y para eso es hora de crear una aplicación desde cero a modo de ejemplo siguiendo unos sencillos pasos.

1. Lo primero es realizar una lluvia de ideas partiendo de conceptos generales e ir haciendo preguntas hasta descubrir la aplicación que se desea diseñar, tal y como se representa en la Figura 41.

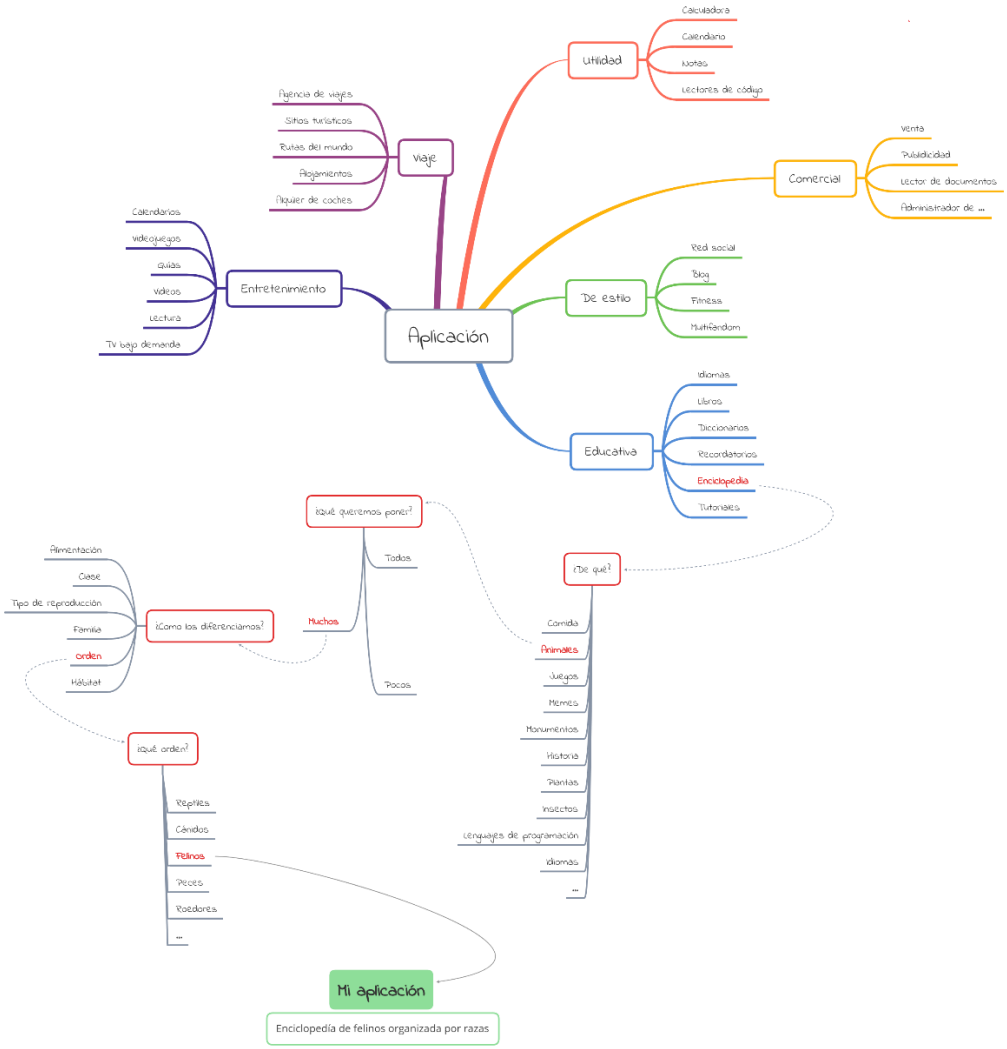

Figura 41. Aplicación de ejemplo: Elección de una aplicación

Como se puede ver al final de la Figura 41, la aplicación escogida para realizar en este ejemplo es una enciclopedia de felinos organizada por razas.

2. Una vez elegido el concepto general de la aplicación es momento de darle forma creando una lista con las pantallas a mostrar en la aplicación como la de la Figura 42.

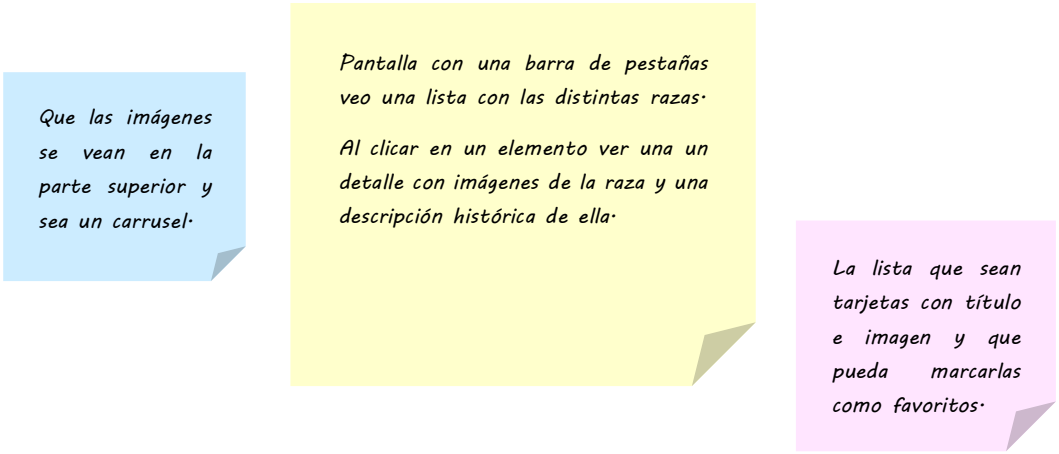

Figura 42. Aplicación de ejemplo: Ideas para las pantallas

3. Ahora que se tienen claras las ideas hay que orientarlas al IUMATI Framework, para ello lo mejor es crear un diagrama de categorías y productos como el de la Figura 43 donde las categorías, es decir, la navegación de la aplicación se representa de un gris oscuro y los productos, las pantallas, de gris claro y blanco.

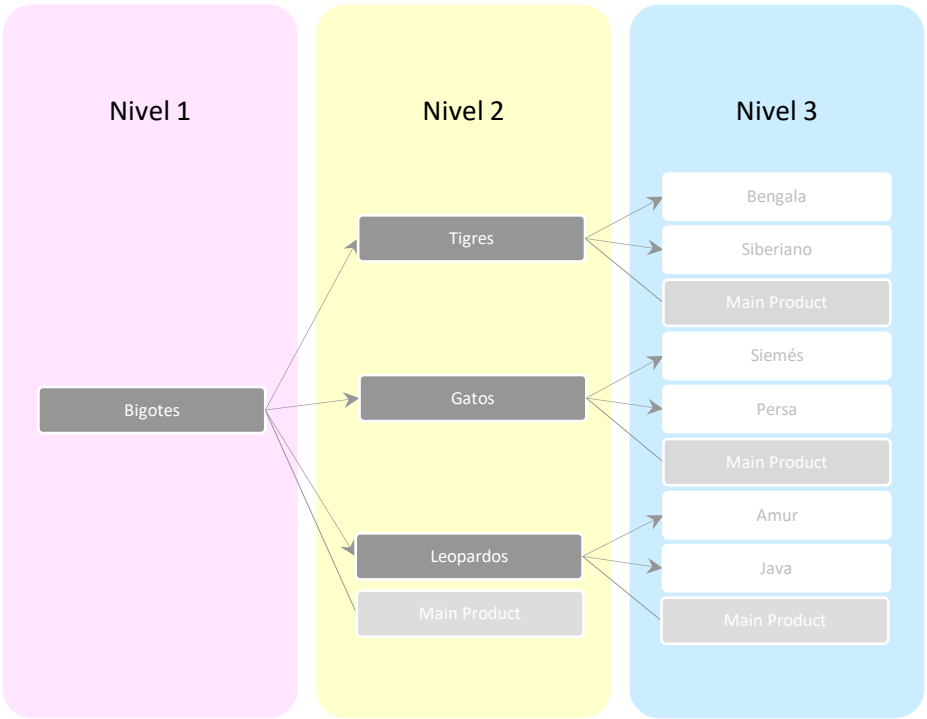

Figura 43. Aplicación de ejemplo: Esquema da catálogo de categorías y productos

4. Ahora que ya se tiene clara la jerarquía de la aplicación sólo hay que crear un diagrama, ya sea a mano o con algún tipo de herramienta como XMind[46], con los valores a poner en cada uno de los campos que componen una categoría y un producto, como en la Figura 44.

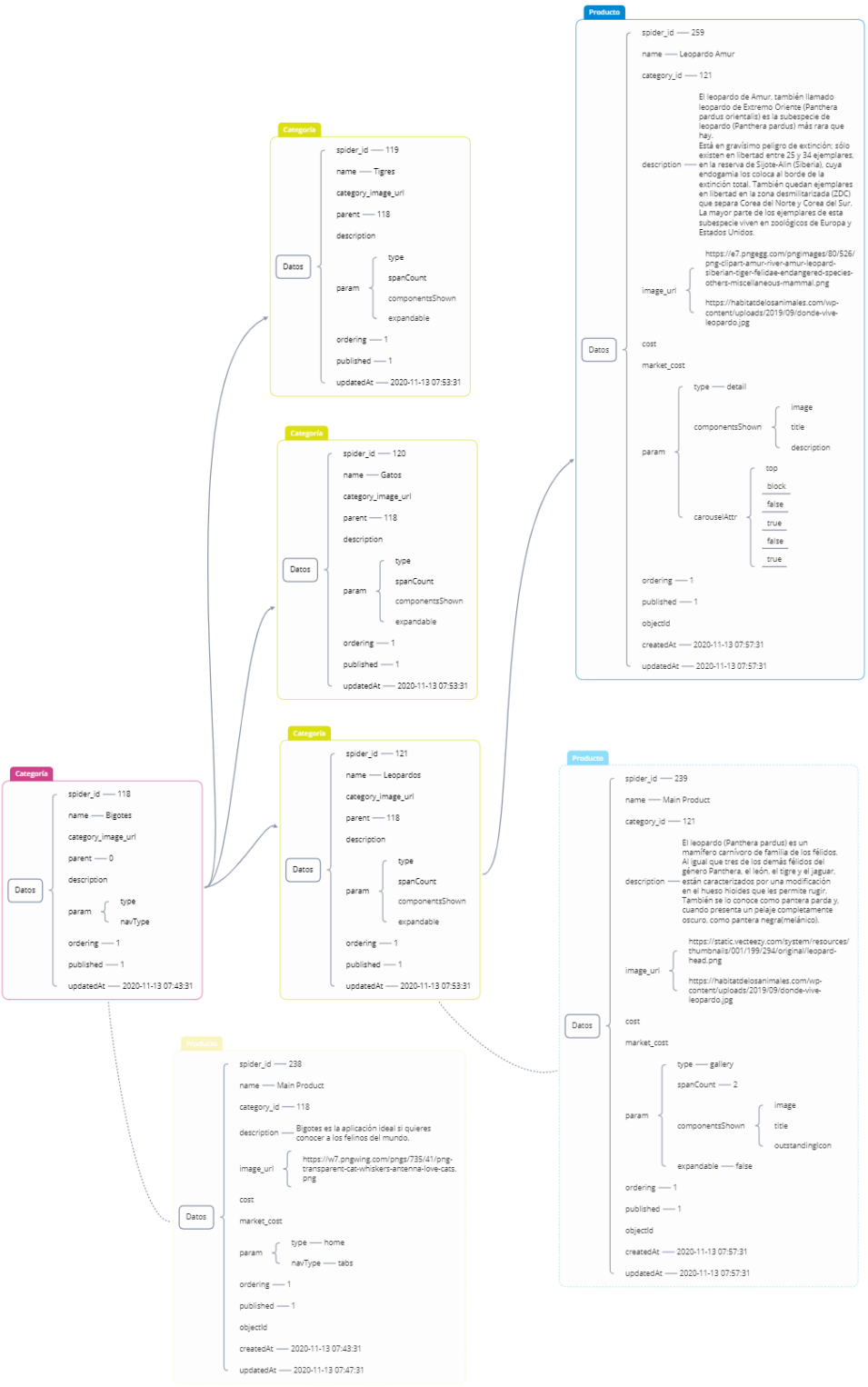

Figura 44. Aplicación de ejemplo: Diagrama completo del catálogo de categorías

5. A continuación, hay que pasar los datos del punto anterior a un archivo de formato JSON con el que trabajará IUMATI Framework para crear la aplicación, para ello se puede emplear herramientas como JSON Editor Online[47] o similares, lo que da como resultado un código como de las Figuras 45 – 46. Asimismo, los nombres de los archivos deben tener el nombre especificado en el epígrafe de las figuras anteriores.

```
{
  "status": "ok",
  "results": [
    {
      "spider_id": "118",
      "name": "Bigotes",
      "parent": "0",
      "category_image_url": "",
      "description": "",
      "param": "type, navType",
      "ordering": "1",
      "published": "1",
      "updatedAt": "2018-12-13 16:58:12"
    },
    {
      "spider_id": "119",
      "name": "Tigres",
      "parent": "118",
      "category_image_url": "",
      "description": "",
      "param": "type\tspanCount\tcomponentsShown\texpandable",
      "ordering": "2",
      "published": "1",
      "updatedAt": "2018-12-16 06:35:30"
    },
    {
      "spider_id": "120",
      "name": "Gatos",
      "parent": "0",
      "category_image_url": "******0",
      "description": "",
      "param": "type\tspanCount\tcomponentsShown\texpandable",
      "ordering": "3",
      "published": "1",
      "updatedAt": "2018-12-13 07:43:31"
    },
    {
      "spider_id": "121"
```


Figura 45. Aplicación de ejemplo: wp_spidercatalog_product_categories.json

```
{
  "status": "ok",
  "results": [
    {
      "spider_id": "238",
      "name": "Main Product",
      "category_id": "118,",
      "description": "Bigotes es la aplicación ideal si quieres
      "image_url": "https://w7.pngwing.com/pngs/735/41/png-trans
      "cost": "",
      "market_cost": "",
      "param": "par_type@@:@@home\tpar_navType@@:@@tabs\t",
      "ordering": "1",
      "published": "1",
      "objectId": "876f5714b5",
      "createdAt": "2018-12-14 07:33:38",
      "updatedAt": "2018-12-14 10:00:20"
    },
    {
      "spider_id": "259",
      "name": "Leopardo Amur",
      "category_id": "121,",
      "description": "El leopardo de Amur, también llamado leopa
      "image_url": "https://e7.pngegg.com/pngimages/80/526/png-c
      "cost": "",
      "market_cost": "",
      "param": "par_type@@:@@detail\tpar_componentsShown@@:@@ima
      "ordering": "4",
      "published": "1",
      "objectId": "d51b2d38ab",
      "createdAt": "2018-12-14 07:33:56",
      "updatedAt": "2018-12-14 10:37:26"
    },
    {
      "spider_id": "249",
      "name": "Main Product",
      "category_id": "121,",
      "description": "El leopardo (Panthera pardus) es un mamífe
```


Figura 46. Aplicación de ejemplo: wp_spidercatalog_products.json

Durante la creación del archivo JSON se deben tener en cuenta las siguientes normas de configuración para el correcto funcionamiento del *framework*:

- La categoría inicial, es decir, la categoría raíz de la aplicación siempre tiene como identificador de la categoría superior en la jerarquía el valor cero.
- Los productos que representen la pantalla de una categoría siempre tienen el nombre “Main product”.
- Las URL de imágenes se deben separar con la siguiente cadena de caracteres “*******0;;;*” siendo la primera imagen un icono representativo de la categoría o producto. Si no se indica icono dejar la primera posición vacía.
- Los parámetros tanto de categorías como de productos se separan con la cadena de caracteres “\t”.
- Para indicar que una categoría o un producto aparezca en la aplicación, el valor de *published* corresponde con un “1”, en caso contrario un “0”.

- Los valores de *spider_id*, *objectId*, *createdAt* y *updatedAt* serían creados por la interfaz web, pero al no disponer de ella el usuario debe darles un valor.
    - *spider_id* debe contener un número de valor aleatorio.
    - *objectId* será un cadena de caracteres aleatorios.
    - *createdAt y updatedAt* son fechas que siguen el siguiente formato *yyyy-mm-dd hh:mm:ss.*

6. Por último, se pasan los archivos al *framework* y en cuestión de segundos se obtiene una sencilla aplicación (Figura 47).

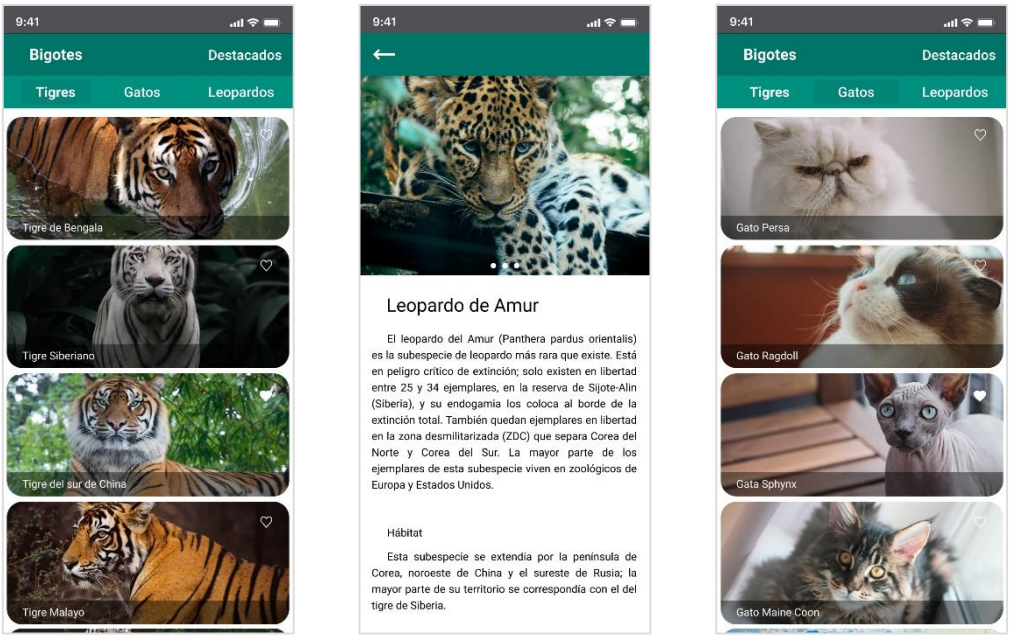

Figura 47. Aplicación de ejemplo

CAPÍTULO VII. CASO DE USO

En este capítulo se detalla una implementación real del proyecto.

1. CONTEXTO

Para testear el funcionamiento del nuevo *framework* se ha pedido a dos personas completamente ajenas al mundo de la programación y la ingeniería desarrollar con ayuda de la guía de usuario un catálogo de categorías y productos. Indicar que en condiciones normales de uso de IUMATI Framework este tipo de usuarios finales sólo interactuaría con la interfaz web, la cual generaría el catálogo en formato JSON que es el punto de partida de este Trabajo de Fin de Grado.

2. DISEÑO

En la etapa inicial los usuarios realizaron el diagrama mostrado en la Figura 48, en el cual se indica el proceso de elección de la aplicación, llegando a la conclusión de realizar una aplicación relacionada con el mundialmente famoso videojuego World of Wordcraft[48] de rol multijugador masivo en línea desarrollado por Blizzard Entertainment[49].

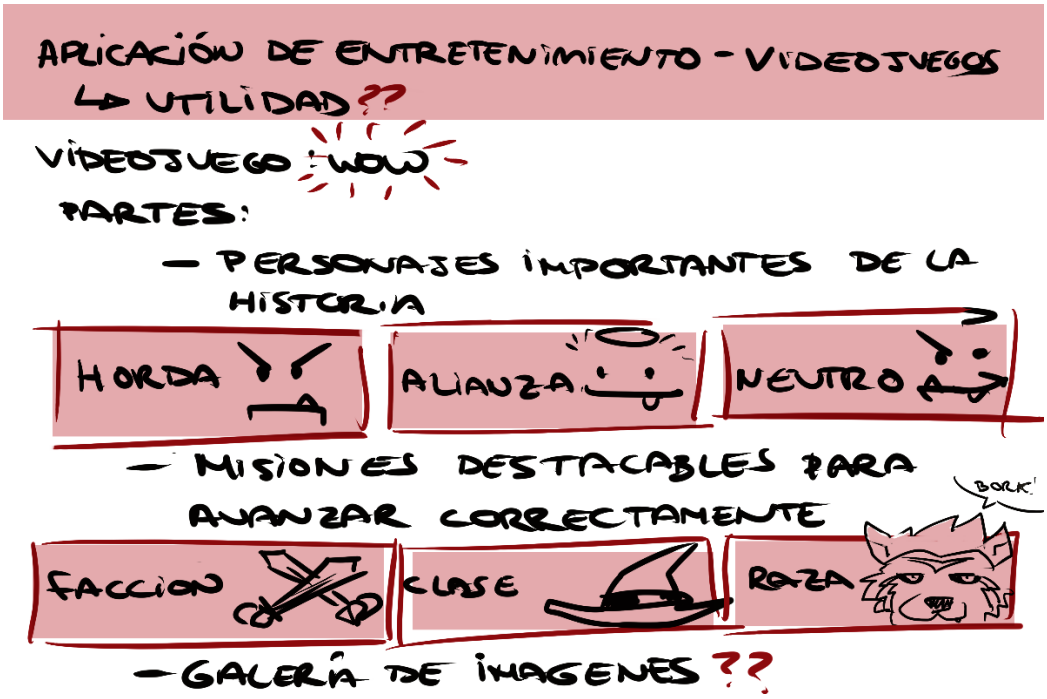

Figura 48. Caso de uso: elección de aplicación

Como se puede ver en la figura anterior, esta aplicación se compondrá de tres partes: una primera parte en la que se verán personajes importantes dentro de la aventura que desarrolla el videojuego según tres los tres parámetros escogidos por los usuario (Horda, Alianza y Neutro) ; una parte que incorpora las misiones principales para avanzar dentro de este según la facción a la que se

pertenece, la clase o raza del personaje; y finalmente, incluir una gallería de imágenes representativas del juego.

A continuación, los usuarios mediante otro diagrama (Figura 49) escogieron como representar cada una de las partes en pantallas. Como se puede apreciar, la pantalla inicial de la aplicación es la pantalla predeterminada que incluye IUMATI Framework (Splash), con el nombre de la aplicación, el cual es *Wore* con su icono representativo.

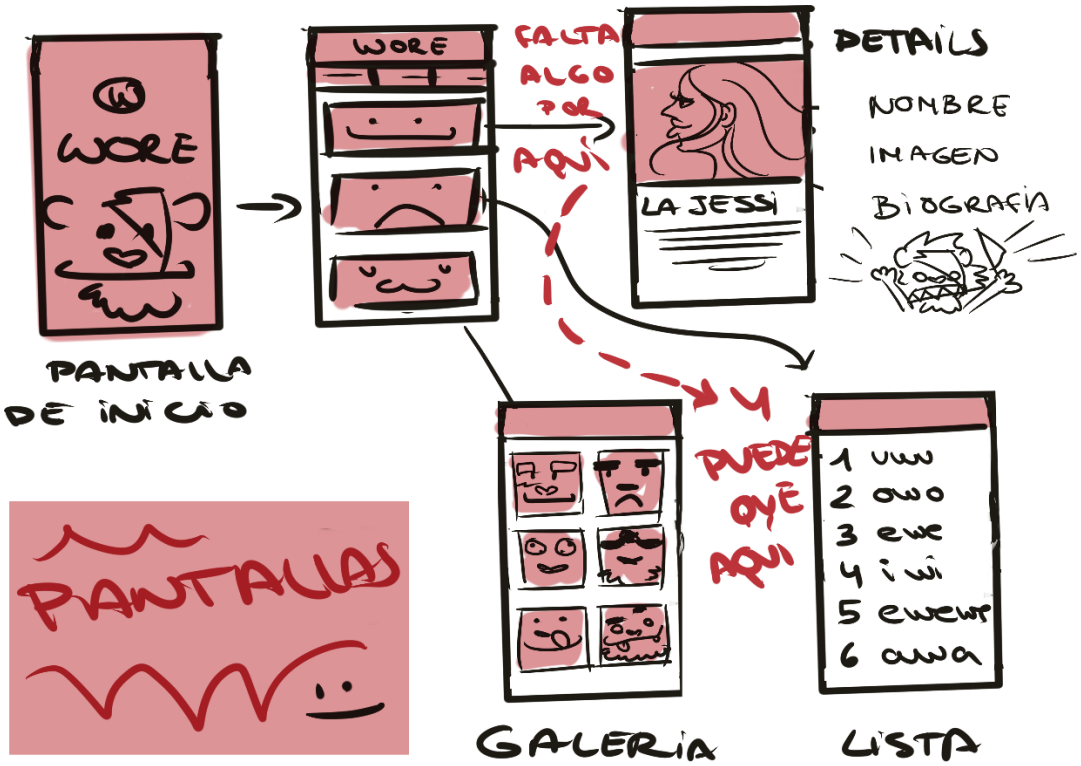

Figura 49. Caso de uso: diagrama de pantallas

Posteriormente, está la pantalla principal de la aplicación que muestra el nombre de la misma, unos botones de navegación y unas tarjetas con datos dentro. De las tarjetas se navega a unas listas (pantallas no definidas en la figura) que van a detalles en el caso de personajes y a otra lista en el caso de misiones. Asimismo, desde la pantalla principal se navega a la galería de imágenes.

Una vez finalizada la etapa de ideas los usuarios realizaron el diagrama, representado en la Figura 49, de la jerarquía final que tendría el catálogo de categorías (rojo pastel) y productos (azul y amarillo) de su aplicación. Además, indicando en el caso de los productos que tipo de pantalla, de las disponibles en en framework, serían.

Por tanto, la aplicación diseñada por los usuarios consta de nueve categorías y un total de veinticuatro productos según el diagrama mostrado a continuación.

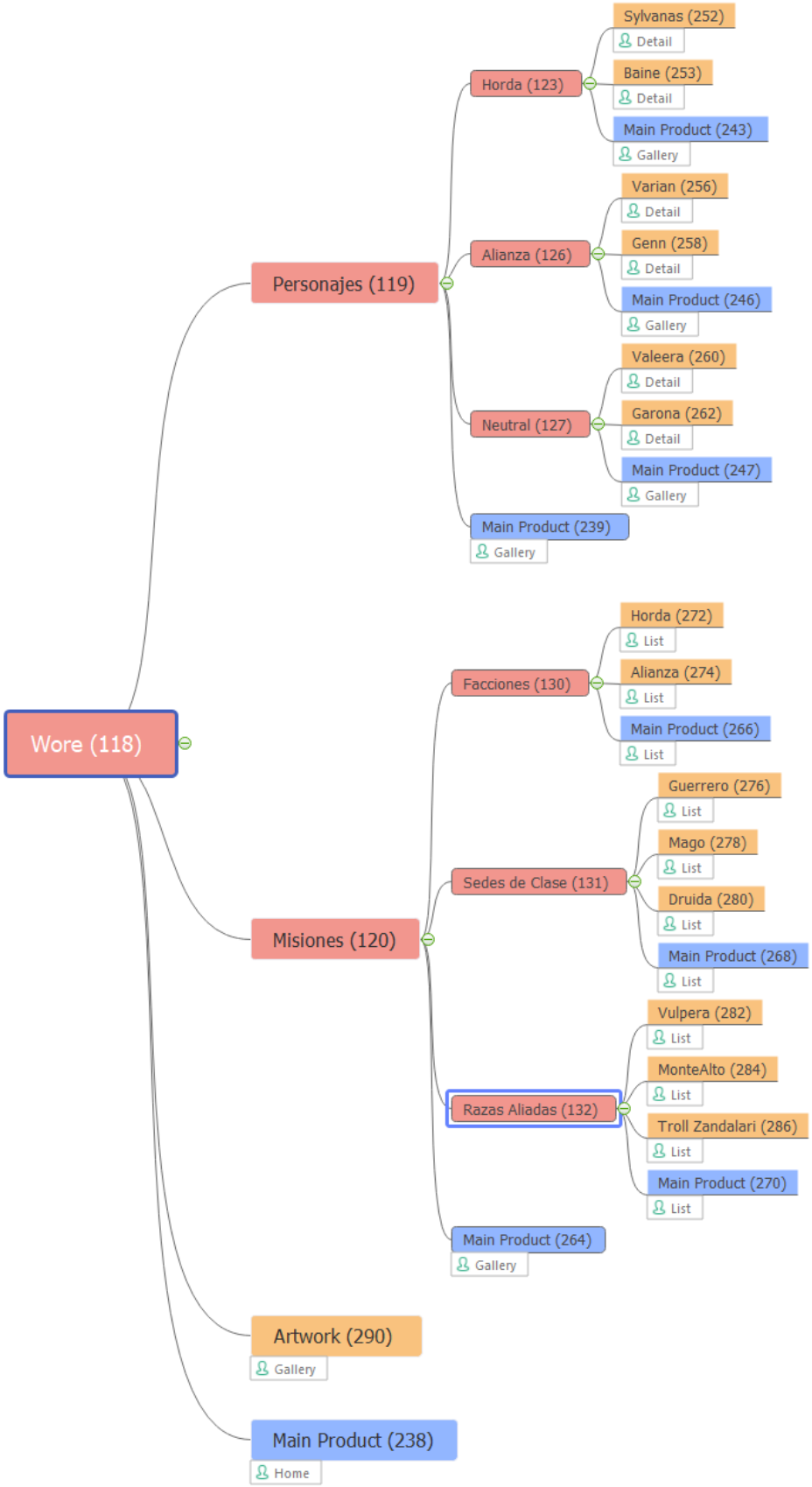

Figura 50. Caso de uso: jerarquía del catálogo de categorías y productos

Para finalizar la etapa de diseño, los usuarios crearon el catálogo de categorías y productos en dos archivos de tipo JSON con la estructura requerida por IUMATI Framework como se puede ver en las Figuras 51 – 52.

```
{
  "status": "ok",
  "results": [
    {
      "spider_id": "118",
      "name": "Wore",
      "parent": "0",
      "category_image_url": "https://www.guiaswow.com/wp-content/uploa
      "description": "Wore, tu aplicación si eres novato en word of wa
      "param": "type\tcomponentShown",
      "ordering": "1",
      "published": "1",
      "updatedAt": "2018-12-13 07:43:31"
    },
    {
      "spider_id": "119",
      "name": "Personajes",
      "parent": "118",
      "category_image_url": "https://www.guiaswow.com/wp-content/uploa
      "description": "Descubre la historia de los personajes más icóni
      "param": "type\tcomponentShown\texpandable",
      "ordering": "2",
      "published": "1",
      "updatedAt": "2018-12-16 06:35:30"
    },
    {
      "spider_id": "123",
      "name": "Horda",
      "parent": "119",
      "category_image_url": "https://www.guiaswow.com/wp-content/uploa
      "description": "La facción que siempre ha unido a las razas desp
      "param": "type\tspanCount\tcomponentShown\texpandable",
      "ordering": "6",
```


Figura 51. Caso de uso: wp_spidercatalog_product_categories.json


```
status": "ok",
results": [
 {
   "spider_id": "238",
   "name": "Main Product",
   "category_id": "118",
   "description": "Wore, tu página oficial del Wow donde encontr
   "image_url": "******0",
   "cost": "",
   "market_cost": "",
   "param": "par_type@@:@@home\tpar_navType@@:@@tabs\t",
   "ordering": "1",
   "published": "1",
   "objectId": "c8e9086582",
   "createdAt": "2018-12-13 07:33:57",
   "updatedAt": "2018-12-13 16:47:58"
 },
 {
   "spider_id": "239",
   "name": "https://www.okgrancanaria.com/wp-content/uploads/201
   "category_id": "119",
   "description": "Los mejores personajes de Worl of Warcraft.",
   "image_url": "",
   "cost": "",
   "market_cost": "",
   "param": "par_type @@:@@gallery\tpar_spanCount@@:@@1\tpar_con
   "ordering": "1",
   "published": "1",
   "objectId": "0c44d486df",
   "createdAt": "2018-12-13 16:13:09",
   "updatedAt": "2018-12-13 20:17:24"
 },
```


Figura 52. Caso de uso: wp_spidercatalog_products.json

Finalmente, los archivos JSON se incorporaron dentro del *framework* para su ejecución. Además, también se añadieron el icono de la aplicación creado por los usuarios y los colores de estilo escogidos.

3. APLICACIÓN RESULTANTE

En esta sección se detalla cómo se han configurado, mediante figuras (Figura 53 - 61), las pantallas que el nuevo IUMATI Framework a creado tras insertar el catálogo de categorías y productos creados por los usuarios y que conforman la aplicación final diseñada por el usuario.

Debido a que algunas pantallas incorporan el mismo diseño se especifica de forma global una pantalla o varias a modo de explicación para indicar lo que contienen.

La Figura 53 corresponde con la pantalla inicial de la aplicación y que emplea IUMATI Framework para cargar de catálogo categorías y productos que emplea para configurar el resto de la aplicación.

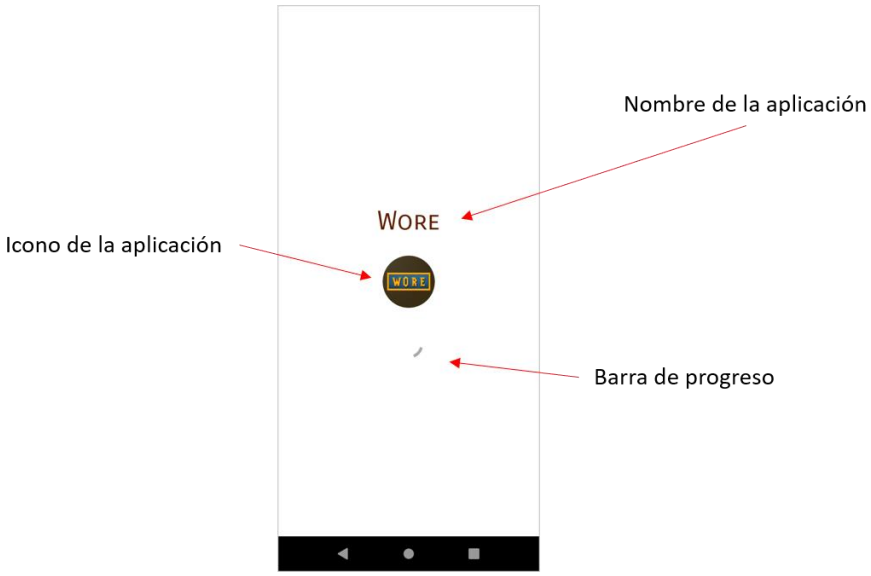

Figura 53. Aplicación *Wore*, pantalla inicial

En la pantalla principal de la aplicación (Figura 54) se ve claramente el tipo de navegación escogido por lo usuarios, menú de pestañas, y que se muestra la primera categoría *Personajes*, la cual corresponde con la vista de tarjetas con la opción de destacados.

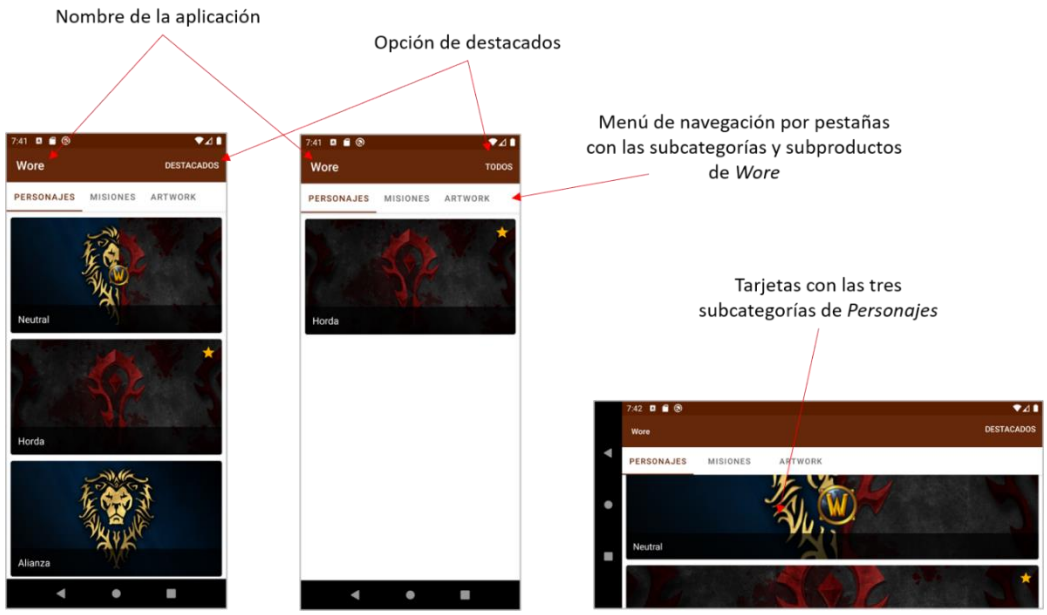

Figura 54. Aplicación *Wore*: pantalla principal

Las subcategorías de *Personajes*, mostradas en la Figura 55, constan de dos tipos de configuraciones: una correspondiente a las pantallas de *Horda* y *Alianza* en las que las tarjetas correspondientes a sus productos son expandibles, mostrando sus descripciones y que, además se pueden compartir, mientras que en la pantalla de Neutral no se dispone de esa opción, pero dispone de la funcionalidad de destacos.

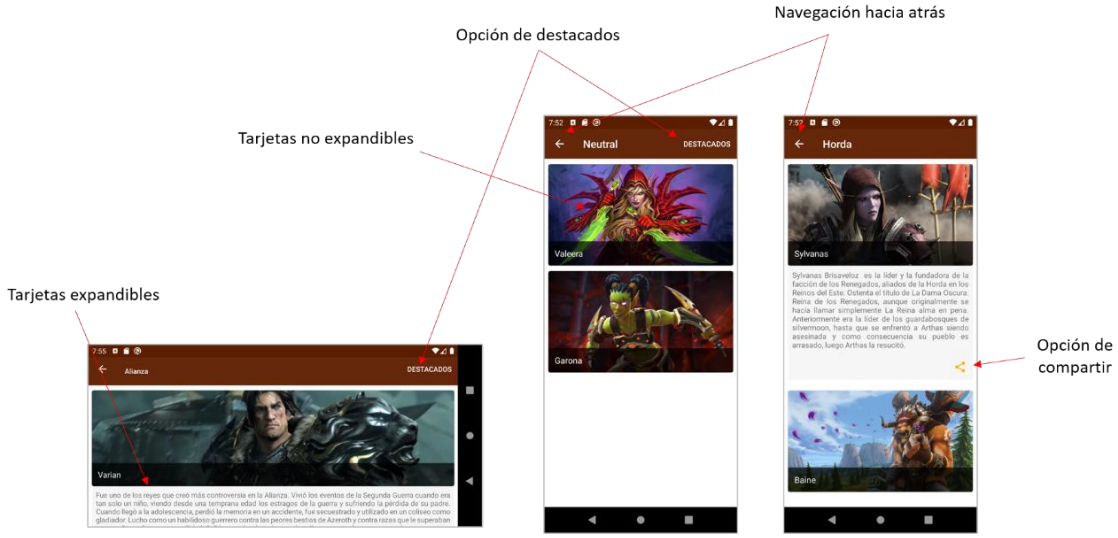

Figura 55. Aplicación *Wore*: pantallas Neutral, Horde y Alianza

Las pantallas de detalle correspondientes a los productos de la categoría *Neutral* muestran el nombre del personaje, un carrusel de imágenes y su descripción. Asimismo, entre una pantalla y otra los usuarios decidieron configurar el carrusel de dos formas diferentes.

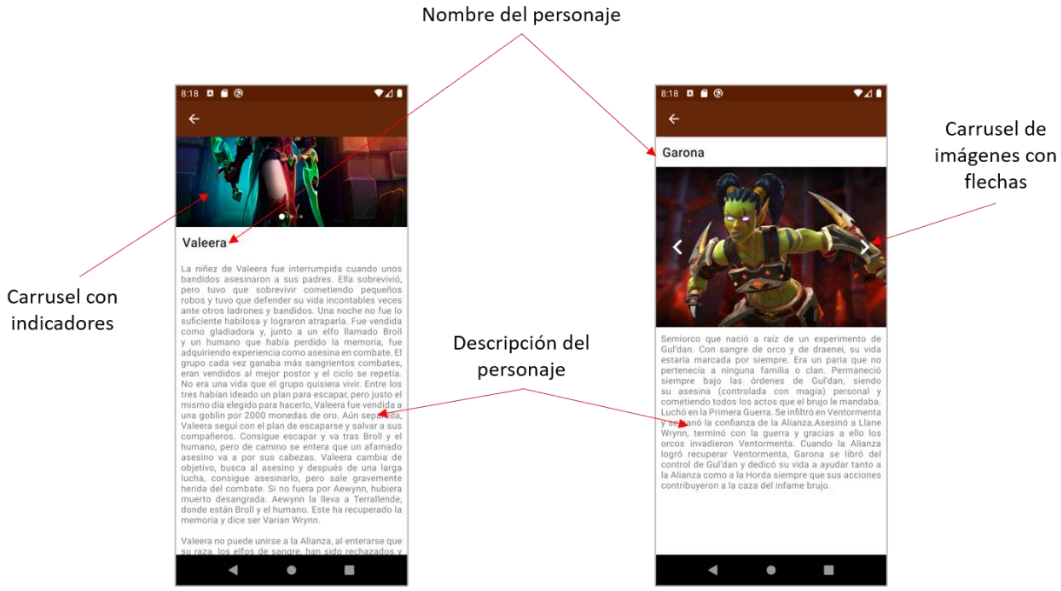

Figura 56.Aplicación *Wore*: pantallas de los personajes de nutrales

Respecto a la pantalla de *Misiones* (Figura 57), los usuarios optaron por mostrarla como una lista de tarjetas con sin opción de destacados y, además, las configuraron de forma que se vieran dos por fila.

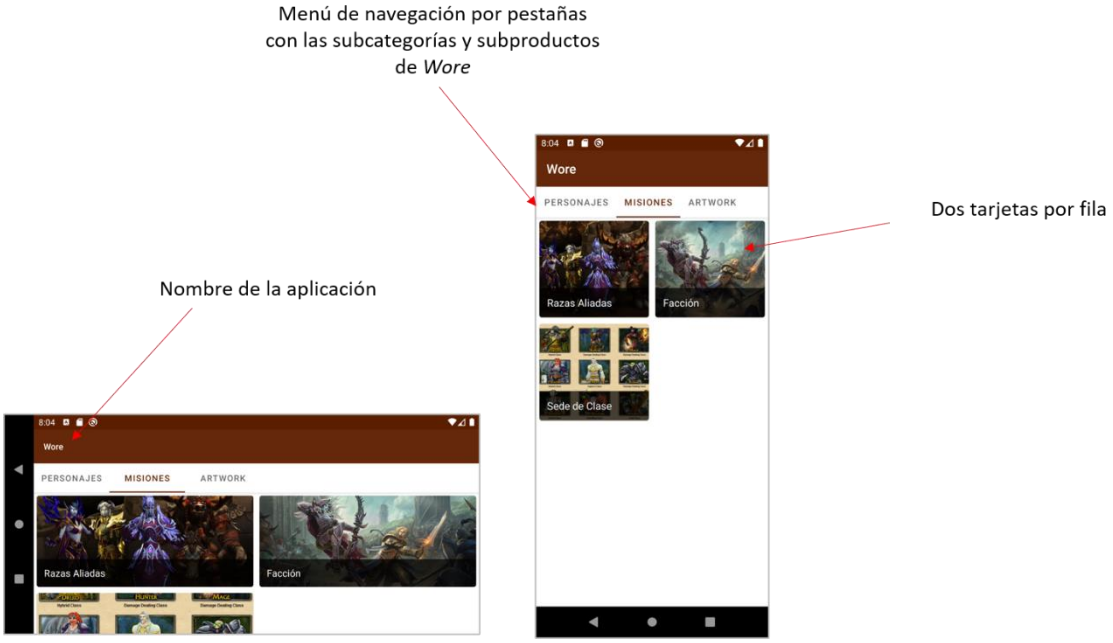

Figura 57. Aplicaión *Wore*: pantalla *Misiones*

La configuración escogida para las pantallas pertenecientes a las subcategorías de los tipos de misiones (Figura 57) es una lista con una imagen representativa además del subtipo de misión y la fecha de última modificación con el fin de conocer el último estado de las misiones.

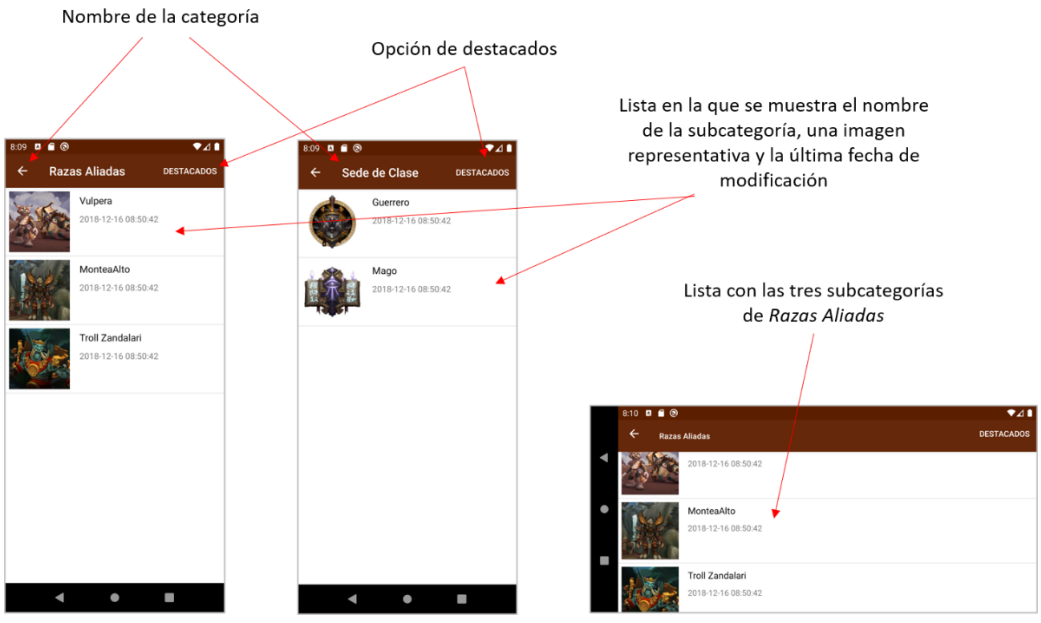

Figura 58. Aplicación Wore: pantallas de Razas Aliadas y Sede de Clase

En cuanto a las pantallas que muestran el título de las misiones disponibles en el juego, los usuarios decidieron que la mejor forma de visualizarlas es mediante una lista básica como vemos en la Figura 59.

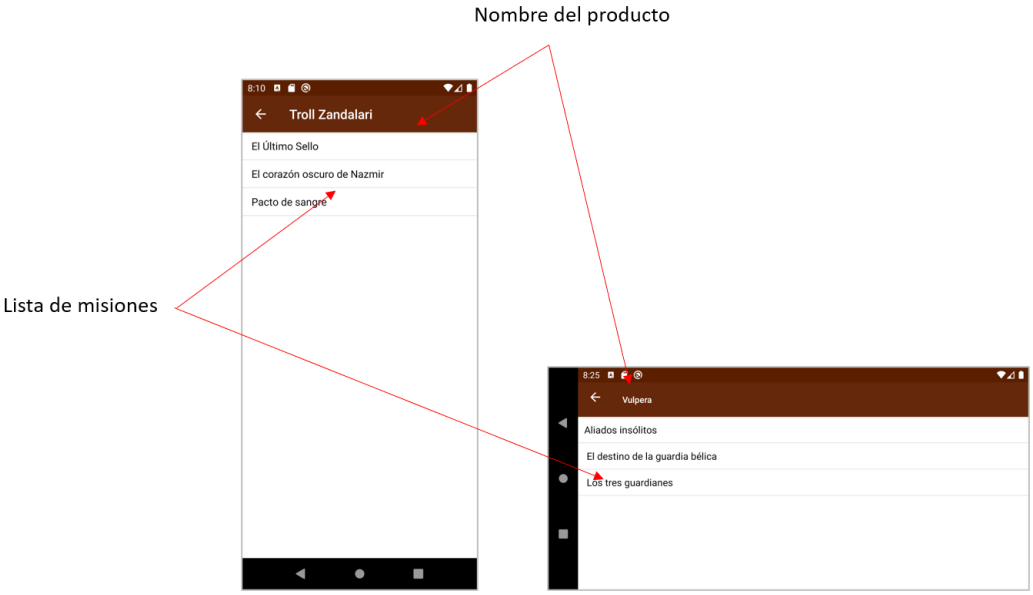

Figura 59. Aplicación *Wore*: pantallas de las subcategorías de misiones

La última opción de las visualizadas en la pantalla principal consiste en una pantalla de galería de imágenes oficiales del videojuego en la que, como se poder ver en la Figura 60, los usuarios dispusieron que se viera tres imágenes por fila.

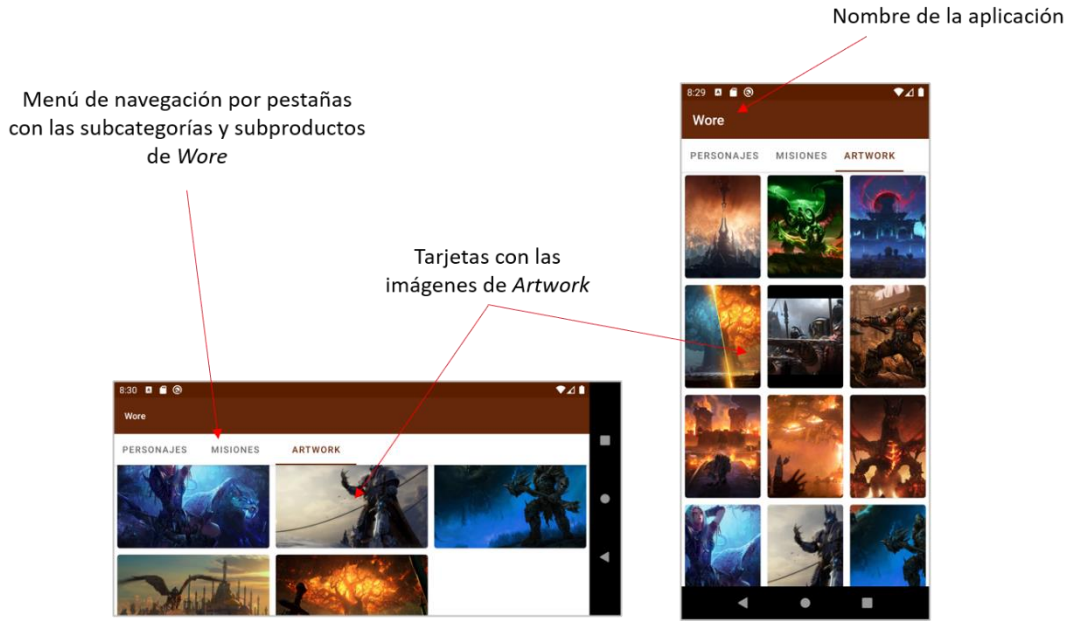

Figura 60. Aplicación *Wore*: pantalla *Artwork*

Finalmente, el último tipo de pantalla de la que dispone *Wore* es un visualizador básico con la funcionalidad de compartir la imagen y a la cual se puede hacer *zoom*.

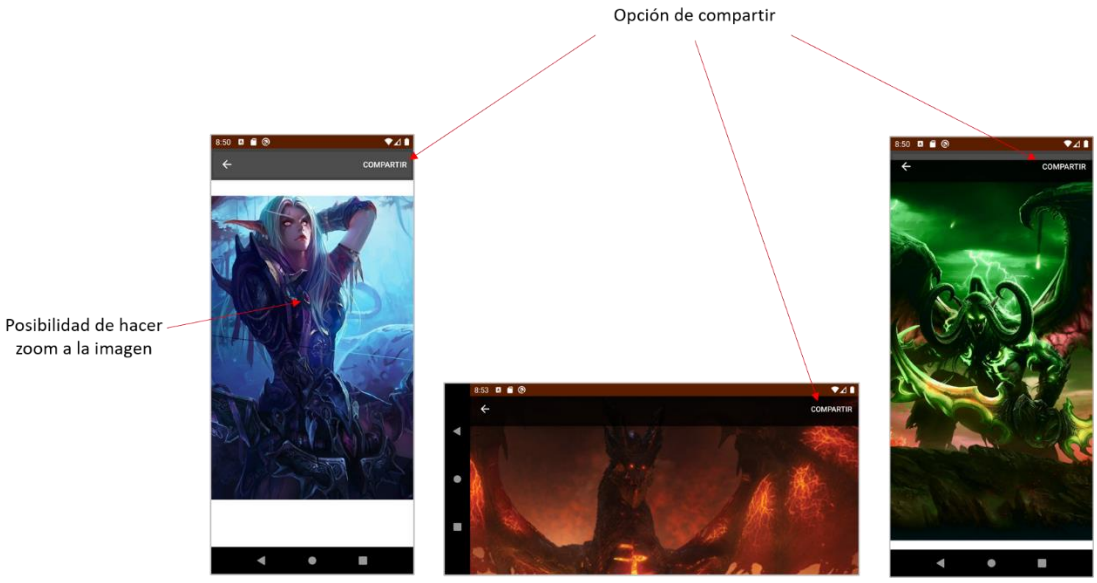

Figura 61. Aplicación *Wore*: pantalla de visualización de imágenes de la galería

4. CONCLUSIONES DE LOS USUARIOS

Una vez creada la aplicación y analizadas todas las pantallas con el objetivo de comprobar que efectivamente tienen las configuraciones que los usuarios indicaron en el catálogo de categorías y productos, se les mostró el resultado final de la aplicación que diseñaron.

La finalidad de esta última parte es comprobar el grado de satisfacción de los usuarios con el nuevo IUMATI Framework, para ello se les pidió dar sus opiniones, las cuales se muestran a continuación, respecto a su trabajo realizado y la aplicación final obtenida. Antonio Zebenzui Marrero Rojas (Pedagogo) y Gabriel Ubay Vega García (Auxiliar de enfermería) (2020) concluyeron que:

> *"La experiencia con la guía de usuario ha sido muy positiva en general. Hay conceptos que como usuarios costaban más de interpretar al no tener conocimientos previos sobre la materia, y hemos tenido la necesidad de releer varias veces ciertos aspectos tales como las diferencias entre categorías, productos y "Main Product", la nomenclatura usada a la hora de programar o cómo influyen ciertos parámetros, pero a medida que empezábamos a familiarizarnos con el procedimiento y hemos entendido todos los aspectos de la guía, ha empezado a resultar más sencillo a medida que íbamos progresando en el proyecto.*

La estructuración de la guía y la explicación de cada parámetro con detalle, unido a la posibilidad de comparar nuestro proyecto con los ejemplos nos ha ayudado mucho a la hora de saber que hacíamos los pasos de manera correcta. También hemos de comentar que el hecho de que tuviese un apartado gráfico cuidado (con fotografías, ejemplos, gráficas, etc.) ha ayudado a que la lectura fuese mucho más amena.

Tras algunas dificultades básicas como que nos faltasen comas o que nos despistásemos en ciertos parámetros, hemos logrado llevar a cabo todo el proceso satisfactoriamente con un resultado muy positivo y con el que estamos muy contentos."

CAPÍTULO VIII. CONCLUSIONES

En este capítulo se hace un breve resumen el documento y se exponen las resoluciones finales del proyecto.

1. CONTEXTO INICIAL

Tal y como se especificó en el capítulo de introducción, el objetivo principal de este Trabajo de Fin de Grado es la adaptación del IUMATI Framework a las versiones más actuales de Android, reescribiendo para ello desde cero el código de este usando el lenguaje de programación Kotlin.

Con el fin de cumplir con el objetivo anteriormente descrito, se estipularon los siguientes objetivos específico:

1. Análisis de la funcionalidad del IUMATI Framework implementado en Java, el cual configura una aplicación en función de un catálogo de categorías y productos.
2. Diseño de las pantallas de las que dispondrá el nuevo *framework* implementado en Kotlin, así como el patrón de diseño de clases que compondrá la implementación de cada pantalla.
3. Desarrollo del código necesario para implementación de las pantallas decididas en la etapa de diseño, lo cual indica desarrollar sus funcionalidades, gestión de datos y configuración de aspecto.
4. Análisis de un caso de uso del *framework* con un catálogo de categorías y productos real.

2. CONCLUSIONES

Una vez reiterados los objetivos del proyecto en el aparatado anterior, en la Tabla 4 se detalla los resultados obtenidos para cada uno de ellos.

Objetivo	Conclusión	Justificación
1	Cumplido	En el capítulo de análisis se realiza una exhaustiva y detallada investigación de la historia del IUMATI Framework, así como de su funcionamiento y la estructura de datos que emplea.
2	Cumplido	Este objetivo se logra en el capítulo de diseño con la realización de seis bocetos de pantallas y la elección de la variante del patrón de diseño MVP con *Router*, *Screen* y *ViewModel.* Además, en el capítulo de implementación se

		indica que durante dicha fase se diseñaron dos contralores más y se modificó otro ya diseñado.
3	Cumplido	En el capítulo de implementación queda evidenciado el código creado para el desarrollo de las funcionalidades de los ocho controladores de pantalla. Asimismo, también el código desarrollado para la gestión de datos desde la obtención del catálogo de categorías y productos hasta el envío de datos a los diferentes controladores.
4	Cumplido	Este objetivo se logra conjuntamente en los capítulos de guía de usuario y caso de uso donde se realiza un catálogo de categorías y productos para posteriormente introducirlo en el *framework* y obtener una aplicación completa.

Tabla 4. Conclusiones de los objetivos del proyecto.

Con la realización de este Trabajo de Fin de Grado queda demostrado la escalabilidad y capacidad de configuración de IUMATI Framework en la creación de aplicaciones móviles a partir de un catálogo de categorías y productos. Tal y como queda reflejado en el capítulo de implementación, la adaptación del *framework* a las últimas versiones de Android se completó exitosamente, así como su desarrollo en Kotlin.

Por otra parte, el en caso de uso realizado con usuarios completamente ajenos a la programación se demuestra la flexibilidad y múltiples configuraciones que permiten los ocho controladores de pantalla disponibles en el *framework*. Por consiguiente, se concluye que los objetivos y finalidad de este proyecto han sido un éxito debido a la posibilidad de que usuarios sin conocimientos de programación creen aplicaciones móviles de Android desde cero con todas las posibilidades que el nuevo IUMATI Framework pone a disposición de ellos.

3. FUTURO DEL PROYECTO

Entre las líneas futuras de este proyecto se propone, en primer lugar, posibilitar que el usuario pueda configurar desde el catálogo los estilos de color y letra, así como el nombre e icono de la

aplicación, entre otras cosas. Además, permitir que el usuario interactúe con la aplicación resultante más allá de la navegación básica incorporando botones flotantes y creando nuevos controladores de pantalla.

Por último, analizar los controladores existentes e incorporar nuevos componentes para ampliar las posibilidades de configuración, lo que aumentaría aún más su potencial de crecimiento respecto a tipos de aplicaciones generadas.

CAPÍTULO IX. BIBLIOGRAFÍA

En este capítulo se detallan las referencias empleadas en la realización de este Trabajo de Fin de Grado.

[1] “Teléfono móvil - Wikipedia, la enciclopedia libre.” [Online]. Available: https://es.wikipedia.org/wiki/Teléfono_móvil. [Accessed: 10-Oct-2020].

[2] “IBM Simon - Wikipedia, la enciclopedia libre.” [Online]. Available: https://es.wikipedia.org/wiki/IBM_Simon. [Accessed: 10-Oct-2020].

[3] “Cuál fue la primera aplicación móvil de la historia | Blog Comandia.” [Online]. Available: https://www.correosecommerce.com/blog/historia-aplicaciones-moviles/. [Accessed: 11-Oct-2020].

[4] “How Google Play Works: 2019 Google Play Public Policy Report de Google Play - Libros en Google Play.” [Online]. Available: https://play.google.com/store/books/details/Google_Play_How_Google_Play_Works?id=vC2-DwAAQBAJ. [Accessed: 11-Oct-2020].

[5] “Android and Google Play statistics, development resources and intelligence | AppBrain.” [Online]. Available: https://www.appbrain.com/stats. [Accessed: 11-Oct-2020].

[6] “Ionic - Cross-Platform Mobile App Development.” [Online]. Available: https://ionicframework.com/. [Accessed: 12-Oct-2020].

[7] “Angular.” [Online]. Available: https://angular.io/. [Accessed: 12-Oct-2020].

[8] “PhoneGap.” [Online]. Available: https://phonegap.com/. [Accessed: 12-Oct-2020].

[9] “jQuery Mobile.” [Online]. Available: https://jquerymobile.com/. [Accessed: 12-Oct-2020].

[10] “React Native · A framework for building native apps using React.” [Online]. Available: https://reactnative.dev/. [Accessed: 12-Oct-2020].

[11] “Framework7 - Full Featured Framework For Building iOS, Android & Desktop Apps.” [Online]. Available: https://framework7.io/. [Accessed: 12-Oct-2020].

[12] “GoodBarber - Crear una aplicación móvil: Aplicación de Compras y Aplicación Clásica.” [Online]. Available: https://es.goodbarber.com/. [Accessed: 12-Oct-2020].

[13] “Cómo crear una aplicación para Android y iPhone.” [Online]. Available: https://www.swiftic.com/es/. [Accessed: 12-Oct-2020].

[14] “Mobincube el mejor CREADOR gratuito de aplicaciones Android iPhone/iPad.” [Online].

Available: https://mobincube.com/es/. [Accessed: 12-Oct-2020].

[15] “Shoutem: Mobile app builder for native iOS and Android apps | Shoutem.” [Online]. Available: https://shoutem.com/?utm_expid=.-7MMwQQ4SmCmRdklPg2J0g.0&utm_referrer=https%3A%2F%2Fwww.google.com%2F. [Accessed: 12-Oct-2020].

[16] “App Builder Chosen by Industry Leaders | Mobile Roadie.” [Online]. Available: https://mobileroadie.com/. [Accessed: 12-Oct-2020].

[17] “Creador de apps para iPhone, Android y Progressive Web Apps - AppYourself.” [Online]. Available: https://appyourself.net/es/creador-de-apps/. [Accessed: 12-Oct-2020].

[18] “WordPress.com: crea un sitio web o blog gratuito.” [Online]. Available: https://es.wordpress.com/. [Accessed: 11-Nov-2020].

[19] “Plugin de Constructor de Catálogo Wordpress | Spider Catalog.” [Online]. Available: https://web-dorado.com/es/products/wordpress-catalog.html. [Accessed: 12-Oct-2020].

[20] “Introducción a Android Studio | Desarrolladores de Android.” [Online]. Available: https://developer.android.com/studio/intro?hl=es-419. [Accessed: 12-Oct-2020].

[21] “What is Kotlin? The Java alternative explained | InfoWorld.” [Online]. Available: https://www.infoworld.com/article/3224868/what-is-kotlin-the-java-alternative-explained.html. [Accessed: 12-Oct-2020].

[22] “HTML.” [Online]. Available: http://www.hipertexto.info/documentos/html.htm. [Accessed: 13-Nov-2020].

[23] “SQLite Home Page.” [Online]. Available: https://www.sqlite.org/index.html. [Accessed: 08-Nov-2020].

[24] “Biblioteca de persistencias Room | Desarrolladores de Android.” [Online]. Available: https://developer.android.com/topic/libraries/architecture/room. [Accessed: 08-Nov-2020].

[25] “Dao | Desarrolladores de Android | Android Developers.” [Online]. Available: https://developer.android.com/reference/androidx/room/Dao. [Accessed: 08-Nov-2020].

[26] “String | Desarrolladores de Android | Android Developers.” [Online]. Available: https://developer.android.com/reference/java/lang/String. [Accessed: 18-Nov-2020].

[27] “Significado de URL (Qué es, Concepto y Definición) - Significados.” [Online]. Available: https://www.significados.com/url/. [Accessed: 05-Nov-2020].

[28] “Retrofit.” [Online]. Available: https://square.github.io/retrofit/. [Accessed: 12-Nov-2020].

[29] “Context | Android Developers.” [Online]. Available: https://developer.android.com/reference/kotlin/android/content/Context?hl=en. [Accessed: 18-Nov-2020].

[30] “Boolean | Android Developers.” [Online]. Available: https://developer.android.com/reference/kotlin/java/lang/Boolean?hl=en. [Accessed: 18-Nov-2020].

[31] “Map | Android Developers.” [Online]. Available: https://developer.android.com/reference/java/util/Map?hl=en. [Accessed: 18-Nov-2020].

[32] “Intent | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/content/Intent?hl=en. [Accessed: 18-Nov-2020].

[33] “Fragment | Android Developers.” [Online]. Available: https://developer.android.com/reference/androidx/fragment/app/Fragment?hl=en. [Accessed: 18-Nov-2020].

[34] “Uri | Desarrolladores de Android | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/net/Uri. [Accessed: 18-Nov-2020].

[35] “URI - Identificadores bibliográficos (ISBN, ISSN, DOI, URI) - Biblioguías at Universidad de Extremadura. Biblioteca.” [Online]. Available: https://biblioguias.unex.es/c.php?g=572103&p=3944905. [Accessed: 05-Nov-2020].

[36] “ArrayList | Android Developers.” [Online]. Available: https://developer.android.com/reference/java/util/ArrayList?hl=en. [Accessed: 18-Nov-2020].

[37] “Definición de PNG - Significado y definición de PNG.” [Online]. Available:

https://sistemas.com/png.php. [Accessed: 06-Nov-2020].

[38] “LiveData | Android Developers.” [Online]. Available: https://developer.android.com/reference/androidx/lifecycle/LiveData?hl=en. [Accessed: 18-Nov-2020].

[39] “Activity | Desarrolladores de Android | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/app/Activity. [Accessed: 18-Nov-2020].

[40] “AppCompatActivity | Desarrolladores de Android | Android Developers.” [Online]. Available: https://developer.android.com/reference/androidx/appcompat/app/AppCompatActivity. [Accessed: 18-Nov-2020].

[41] “Bundle | Desarrolladores de Android | Android Developers.” [Online]. Available: https://developer.android.com/reference/android/os/Bundle. [Accessed: 18-Nov-2020].

[42] “MutableLiveData | Desarrolladores de Android | Android Developers.” [Online]. Available: https://developer.android.com/reference/androidx/lifecycle/MutableLiveData. [Accessed: 18-Nov-2020].

[43] “GitHub - mmobin789/Android-Dynamic-Expandable-List-Adapter: Source code for one for all expandable list adapter that provides an expandable listing feature to unlimited number of extending adapters for their individual lists.” [Online]. Available: https://github.com/mmobin789/Android-Dynamic-Expandable-List-Adapter. [Accessed: 14-Nov-2020].

[44] “What is the Unicode Transformation Format (UTF)? - Definition from Techopedia.” [Online]. Available: https://www.techopedia.com/definition/976/unicode-transformation-format-utf. [Accessed: 02-Nov-2020].

[45] “Descripción general | Maps SDK for Android | Google Developers.” [Online]. Available: https://developers.google.com/maps/documentation/android-sdk/overview. [Accessed: 16-Nov-2020].

[46] “XMind - Mind Mapping Software.” [Online]. Available: https://www.xmind.net/. [Accessed: 17-Nov-2020].

[47] “JSON Editor Online - view, edit and format JSON online.” [Online]. Available:

https://jsoneditoronline.org/. [Accessed: 17-Nov-2020].

[48] “World of Warcraft.” [Online]. Available: https://worldofwarcraft.com/es-es/. [Accessed: 22-Nov-2020].

[49] “Blizzard Entertainment.” [Online]. Available: https://www.blizzard.com/es-es/. [Accessed: 22-Nov-2020].

[50] “COIT | Colegio Oficial Ingenieros de Telecomunicación.” [Online]. Available: https://www.coit.es/. [Accessed: 19-Nov-2020].

CAPÍTULO X. PRESUPUESTO

En este capítulo se detalla la estimación monetaria para la realización del proyecto detallado en este documento.

1. DESGLOSE DEL PRESUPUESTO

La estimación del coste del proyecto redactado en este documento se basa en las pautas establecidas por el Colegio Oficial de Ingenieros Técnicos de Telecomunicación (COITT)[50] hasta el año 2008. Dichas pautas desglosan el presupuesto de un proyecto en los siguientes apartados que posteriormente serán detallados en profundidad:

- Recursos materiales.
- Trabajo tarifado por tiempo empleado.
- Costes asociados a la redacción del documento.
- Derechos del visado del COITT.
- Gastos de tramitación y envío.
- Aplicación de impuestos.

2. RECURSOS MATERIALES

Dentro de los requerimientos del COITT para la estimación económica de un proyecto se encuentran los recursos materiales empleados durante el diseño, desarrollo, testeo y ejecución del proyecto, los cuales se subdividen en recursos *hardware* y *software*, asociados estos últimos, en algunos casos, al requerimiento de licencias para su uso.

Para estimar el coste de amortización se estipula un período de cuatro años, y presuponiendo un sistema de amortización lineal en el que el material inmovilizado se deprecia de forma constante durante el período de tiempo evaluado, se obtiene la siguiente expresión:

$$Coste\ de\ amortización = \frac{Valor\ de\ adquisición - Valor\ residual}{Años\ de\ vida\ útil}$$

Ecuación 1. Coste de amortización

Debido a que el presente proyecto tiene una duración de trescientas horas distribuidas de manera aproximada en cuatro meses, siendo este período inferior a cuatro años, dicho coste serán los derivados de los cuatro meses en los que se ha desarrollado el proyecto.

3. RECURSOS SOFTWARE

Entre las herramientas *software* empleadas para el desarrollo de este Trabajo de Fin de Grado se encuentran:

- Aplicación Figma.
- Entorno de desarrollo Android Studio.
- Aplicación XMind.
- Paquete de ofimática Microsoft Office 365.

En este caso, las tres primeras herramientas son gratuitas, mientras que en la última se dispone de licencia para su uso al pertenecer a la comunidad universitaria de la ULPGC, por lo que no conlleva coste de amortización asociado. Por tanto, el coste total de los recursos *software* utilizados ha sido de cero euros (0€).

4. RECURSOS HARDWARE

Entre los recursos hardware empleados para el desarrollo del proyecto se encuentran:

- Ordenador portátil HP Omen 15-DC0004NS.
- Teléfono móvil Realme X2.
- Teléfono móvil Samsung Galaxy J7 (2016).

Recurso	Valor de adquisición (€)	Valor residual (€)	Coste de amortización (€)
HP Omen	756,06	230	43,84
Realme X2	245,57	100	12,13
Samsung Galaxy J7	230	85	13,33
		Total	69,3

Tabla 5. Coste de amortización de los recursos *hardware*

De la Tabla 5 se deduce que el coste total de recursos *hardware* utilizados en el presente proyecto es de sesenta y nueve euros con treinta céntimos (69,3 €).

5. TRABAJO TARIFADO POR TIEMPO EMPLEADO

En este trabajo se han empleado un total de 300 horas en búsqueda de información, análisis del IUMATI Framework desarrollado en Java, diseño, implementación y testeo del nuevo IUMATI Framework desarrollado en Kotlin, así como ejecución de este y elaboración de la documentación. En total un período de 4 meses.

Siguiendo las recomendaciones del COITT, el importe percibido por las horas trabajadas de un Ingeniero de Telecomunicaciones se calcula con la siguiente expresión:

$$H = C_t \cdot 74{,}88 \cdot H_n + 96{,}72 \cdot H_e$$

Ecuación 2. Honorarios

Donde:

- C_t indica un factor de corrección función del número de horas trabajadas.
- H_n indica las horas trabajas dentro de la jornada laboral.
- H_e indica las horas trabajadas fuera de la jornada laboral.

Según lo establecido por el COITT, el coeficiente tiene un valor variable en función del número de horas empleadas de acuerdo con la siguiente tabla:

Horas empleadas (H_n)	Factor de corrección (C_t)
Hasta 36	1
36 - 72	0,9
72 - 108	0,8
108 - 144	0,7
144 - 180	0,65
180 - 360	0,6
360 - 540	0,55
540 - 720	0,5
720 - 1080	0,45
Más de 1080	0,4

Tabla 6. Factor de corrección en función de las horas trabajadas

En base a la

Tabla 6 y teniendo en cuenta que no se ha trabajado fuera del horario laboral, se determina que el factor de corrección C_t tiene un valor igual a 0,6, ya que el proyecto tiene una duración comprendida entre las 180 y 360 horas.

Al aplicar el factor de corrección y las horas empleadas a la expresión anterior, esta quedaría de siguiente modo:

$$H = 0{,}6 \cdot 74{,}88 \cdot 300 + 96{,}72 \cdot 0 = 13.478{,}4\ €$$

Ecuación 3. Estimación de los honorarios

Los honorarios totales por tiempo dedicado libres de impuestos ascienden a una cuantía de trece mil cuatrocientos setenta y ocho euros y cuarenta céntimos (13.478,4 €).

6. COSTES ASOCIADOS A LA REDACCIÓN DEL DOCUMENTO

El importe de la redacción del proyecto se calcula de acuerdo con la siguiente expresión:

$$R = 0{,}07 \cdot P \cdot C_n$$

Ecuación 4. Coste de la redacción del documento

Donde:

- P es el presupuesto del documento.
- C_n es el coeficiente de ponderación en función del presupuesto.

El coeficiente C_n está determinado por el presupuesto del proyecto en el que, hasta el momento, el presupuesto total asciende a trece mil quinientos cuarenta y siete euros con setenta céntimos (13.547,7€), y el COITT establece que para un presupuesto menor de 30.050 €, el coeficiente de ponderación tiene un valor de 1. Por tanto, incorporando esta información a la Ecuación 4 se obtiene la siguiente expresión:

$$R = 0{,}07 \cdot 13.547{,}7 \cdot 1 = 948{,}34\ €$$

Ecuación 5. Estimación del coste de redacción del documento

Por lo tanto, el coste libre de impuestos derivado de la redacción del proyecto asciende a novecientos cuarenta y ocho euros con treinta y cuatro céntimos (948,34 €).

7. DERECHOS DE VISADO COITT

Los gastos de visado del COITT se tarifan mediante la siguiente expresión:

$$V = 0{,}006 \cdot P \cdot C_v$$

Ecuación 6. Gastos de visado COITT

Donde:

- P es el presupuesto del trabajo.
- C_v es el coeficiente reductor en función del presupuesto del trabajo.

El presupuesto calculado hasta el momento asciende a la suma de los costes de ejecución material (principalmente *hadware*) y de redacción, al no haber costes asociados a material fungible por no ser necesaria la impresión del documento para su evaluación obtenemos que el presupuesto actual es:

$$P = 13.547{,}7 + 948{,}34 = 14.496{,}04\ €$$

Ecuación 7. Estimación de presupuesto de costes de ejecución, material y redacción

Como se ha mencionado anteriormente, el coeficiente de ponderación para presupuestos menores de 30.050 € definido por el COITT tiene un valor de 1, por lo que el coste de los derechos de visado del trabajo asciende a:

$$V = 0{,}006 \cdot 14.496{,}04 \cdot 1 = 86{,}98\ €$$

Ecuación 8. estimación del coste de visado COITT

Por lo tanto, el coste de los derechos de visado del proyecto asciende a ochenta y seis euros con noventa y ocho céntimos (86,98 €).

8. GASTOS DE TRAMITACIÓN Y ENVÍO

Los gatos de tramitación y envío están estipulados en seis euros y un céntimo (6,01 €).

9. APLICACIÓN DE IMPUESTOS

El coste total del proyecto antes de aplicarle los correspondientes impuestos es de 14.568,23 €.

A esta cantidad hay que sumar un 7% de la misma correspondiente al Impuesto General Indirecto Canario.

Recurso	Coste (€)
Recursos materiales	69,3
Trabajo tarifado por tiempo empleado	13.478,4
Costes asociados a la redacción del documento	948,34
Derechos de visado del COITT	86,98
Gastos de tramitación y envío	6,01
Subtotal	14.589,03
Aplicación de impuestos (IGIC 7%)	1.021,23
Total	15.610,26

Tabla 7. Costes totales del proyecto

El presupuesto total de este Trabajo de Fin de Grado asciende a la cuantía de quince mil seiscientos diez euros y veintiséis céntimos (15.610,26€).

Las Palmas de Gran Canaria a 24 de noviembre de 2020

Firma:

Yguanira del Pino Vega Vega

CAPÍTULO XI. ANEXOS

En este capítulo se localiza la documentación complementaria a la memoria.

1. REDACCIÓN DE UN ARCHIVO JSON

JSON (JavaScript Object Notation) consiste en un formato de texto fácil y sencillo para almacenar e intercambiar datos.

Reglas de sintaxis JSON

La sintaxis que emplea JSON deriva de la sintaxis de notación de objetos de JavaScript, otro lenguaje de programación. Por consiguiente, JSON cumple las reglas:

- Los datos están en pares de nombre / valor.
- Los datos están separados por comas.
- Las llaves contienen objetos.
- Los corchetes contienen matrices, es decir, conjuntos de objetos.

Datos JSON

Los datos JSON se escriben como pares de nombre / valor, por ende, consta de un nombre de campo (entre comillas dobles), seguido de dos puntos, seguido de un valor:

```
"name": "Yguanira"
```

Figura 62. Ejemplo de dato JSON

Ejemplo JSON

```
{
   "asignatura": "Trabajo de Fin de Grado",
   "alumnos": [
        {
        "nombre": "Yguanira del Pino",
        "apellidos": "Vega Vega",
        "mención": "Telemática"
        },
        {
        "nombre": " William",
        "apellidos": "Bradford Shockley",
        "mención": "Electrónica"
        }
   ]
}
```

Figura 63. Ejemplo archivo JSON

En la Figura 63 se puede apreciar la estructura de un archivo JSON que contiene un objeto con dos datos: *asignatura*, cuyo valor es *Trabajo de Fin de Grado*, y *alumnos*, cuyo valor consiste es un conjunto de objetos; dentro de alumnos se encuentran dos objetos, cada uno con tres datos y sus correspondientes valores.

2. PLIEGO DE CONDICIONES

En este apartado se especifican las condiciones bajo las que se ha desarrollado este Trabajo de Fin de Grado, detallándose los requisitos hardware y software para la ejecución de este.

2.1. REQUERIMIENTOS SOFTWARE

En lo que respecta a las características y especificaciones software, entre ellas podemos encontrar:

- Microsoft Windows 10 Home.
- Android Studio 4.0.1.
- Microsoft Office 2019.

2.2. REQUERIMIENTOS HARDWARE

El ordenador con el que se ha realizado este Trabajo de Fin de Grado presenta las siguientes especificaciones hardware:

Modelo	HP OMEN 15-DC0004NS
Fabricante	HP
Procesador	Intel® Core™ i5-8300H (frecuencia de base de 2,3 GHz, hasta 4 GHz con tecnología Intel® Turbo Boost)
Memoria RAM	SDRAM DDR4-2666 de 16 GB (1 x 16 GB)
Controlador gráfico	1 TB 7200 rpm SATA + 256 GB PCIe® NVMe™ M.2 SSD
Disco duro	NVIDIA® GeForce® GTX 1050 (GDDR5 de 4 GB dedicados)

Tabla 8. Especificaciones del ordenador portátil

Respecto a los teléfonos móviles empleados durante la fase de desarrollo y testeo de proyecto cuentan con las siguientes características:

	Realme X2	Samsung Galaxy J7 (2016)
Marca	Realme	Samsung
Pulgadas	6.4''	5.5''
Tipo de pantalla	Super Amoled	
Resolución	FHD+	HD
Procesador	SnapD. 730G Octa-Core	Exynos 7870 Octa-Core
Memoria RAM	8 GB RAM	2 GB RAM
Almacenamiento	128 GB	16 GB

Tabla 9. Características de los teléfonos móviles

2.3. RECURSOS HUMANOS

Se partió de un esqueleto básico del IUMATI Framework realizado por el Dr. Luis Hernández Acosta con un catálogo JSON sencillo con el objetivo de analizar la estructura del nuevo *framework* y clarificar el patrón de diseño de los diferentes controladores de pantalla a crear.

El catálogo de categorías y productos empleado en el caso de uso, así como el trabajo previo a él fue realizado por unos amigos completamente ajenos al ámbito de la ingeniería y sin conocimientos previos de programación, con el objetivo de analizar un caso real de aplicación del nuevo IUMATI Framework.