UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Aplicación de patrones de diseño para la resolución de problemas de software en el desarrollo de una aplicación móvil iOS

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Fresneda González, Sergio

Tutor: Albert Albiol, Manuela

Curso 2015/2016

Aplicación de patrones de diseño para la resolución flexible de problemas de software en el desarrollo de una aplicación móvil iOS

Resumen

Este Proyecto nace de la necesidad de desarrollar una *app* móvil en *iOS* para la empresa en la que me encontraba realizando las prácticas. Esta aplicación tendría que tener un mantenimiento lo más sencillo posible, ya que los requerimientos irían cambiando de forma continua según la empresa fuera creciendo y con ella la funcionalidad de la *app*.

El reto al que me enfrentaba era no solo de poder desarrollar un código de fácil lectura para mi o cualquier otro desarrollador que trabajará con él en un futuro, sino utilizar un lenguaje y un entorno en el cual no había desarrollado ningún proyecto de forma profesional.

El proyecto se ha desarrollado durante mi periodo de prácticas en empresa y la posterior contratación.

Palabras clave: Patrones de diseño, aplicación móvil, iOS, reutilización, flexibilidad, mantenibilidad, Swift

Tabla de contenidos

1. Introducción

Imagen 1. Logo de la aplicación.

En este proyecto se tratará el uso de los patrones de diseño aplicados al desarrollo de una aplicación móvil en la plataforma iOS.

La aplicación se llama Tendfy y el logo podemos verlo en la **Imagen 1**. Se trata de una aplicación de localización y compra de zapatos de manera *online/offline*. Esta aplicación es la base de la Startup con el mismo nombre en la que realicé mi periodo de prácticas.

La elección de la plataforma sobre la cual han desarrollado las primeras versiones de la aplicación (*iOS*) no fue algo aleatorio. Mediante una serie de estudios y encuestas, la aplicación se enfocó principalmente a un tipo de público con un poder adquisitivo alto. Este tipo de público utiliza dispositivos *iOS*.

El entorno de desarrollo de la *app* móvil ha sido *XCode* de Apple, este es el entorno de programación por defecto y oficial para la creación de aplicaciones para iOS. Los lenguajes de programación seleccionados para el desarrollo son Swift 2.2/3.0 y *Objective-C*.

La aplicación también tiene una parte *Back-end* apoyada sobre un servidor Parse que le ofrece todos los servicios/peticiones necesarias para el correcto funcionamiento.

De manera externa se ha utilizado un *Scrapper* llamado import.io para extraer la información de los zapatos de distintas tiendas online.

Utilizando conocimientos dados en la asignatura de integración e interoperabilidad del grado y mediante una serie de *API's,* se consiguió automatizar la extracción de datos de las tiendas y la catalogación para su posterior inserción en *Parse*.

Una vez finalizada la versión de la aplicación, esta debería publicarse en la tienda de aplicaciones. Para ello, la subiremos a la AppStore. Al final de la memoria explicaremos los pasos a seguir para publicar una aplicación en la Store.

1.1 Motivación

El buscar el calzado ideal puede llegar a ser una tarea ardua y muchas veces molesta, tener que ir a visitar una o varias tiendas para encontrar esos zapatos que necesitamos conlleva tiempo, del cual muchas veces no disponemos o que no queremos dedicarle a esta tarea.

Con nuestra aplicación, este problema queda solucionado de una manera rápida y sencilla para el usuario, ya que permite localizar el tipo de calzado que busca el usuario en unos pocos segundos desde un sólo sitio y de una forma muy intuitiva.

Por otra parte, se hizo un análisis de mercado, nos percatamos que no existía una aplicación que realizara las mismas funcionalidades. Esto es una gran oportunidad para iniciar una *StartUp* con una idea nueva y rompedora.

Como en cualquier inicio de *StarUp* existen competidores pero centrados en otros factores que no considerábamos principales para el usuario. Teníamos que potenciar el factor diferenciador para poder captar la atención de usuarios que necesitaran solventar el problema que pretendíamos solucionar.

Mediante redes sociales y encuestas a pie de calle, comenzamos a mover la idea para conocer el interés de los potenciales usuarios por la aplicación. El resultado fue muy positivo ya que muchos usuarios nos pedían que lanzáramos la aplicación lo más pronto posible, estaban deseando poder utilizarla, esto resultó muy motivante para el equipo.

1.1.1 App Móvil

Hoy en día todo el mundo dispone de un dispositivo móvil es una plataforma en auge la cual sirve como caldo de cultivo perfecto para cualquier *StartUp*, permite a una empresa jóven visibilidad de manera rápida y usuarios activos. Todos utilizamos diferentes aplicaciones en nuestros *SmartPhones*, a diario descargamos aplicaciones que nos puedan ayudar en la resolución de problemas que necesitamos solventar. Esto convierte el mercado de las aplicaciones en el sitio perfecto en el que comenzar un proyecto.

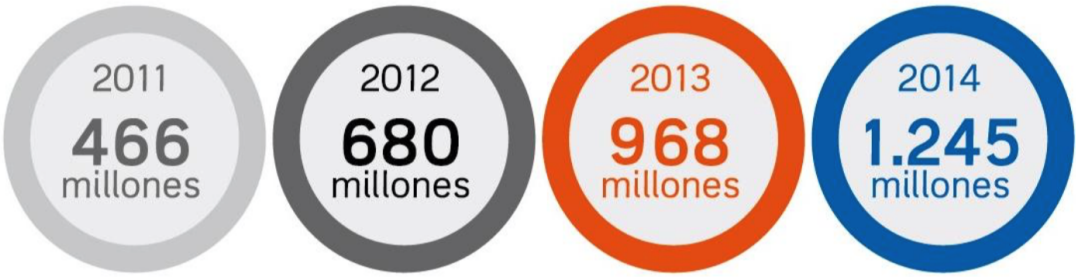


Imagen 2. Evolución de los números de smartphones vendidos en el mundo (unidades)

Normalmente, un usuario cualquiera utiliza una media de 13 aplicaciones durante el día, esto resulta una barrera importante ya que cualquier app quiere estar en ese selecto grupo. Pertenecer a él significa que resulta muy útil al usuario y que quiere que estés siempre accesible cuando este necesite que solventes su problema.

Un estudio realizado en españa a usuarios entre 15 y 50 años dice que el 50% de las apps descargadas por los usuarios son borradas inmediatamente después de su descarga, por no cumplir las expectativas.

Al realizar una aplicación necesitamos atraer la atención del usuario para que utilice y permanezca en la aplicación el mayor tiempo posible. Esto la hace valiosa y evitamos el caer en ese porcentaje de aplicaciones borradas al instante.

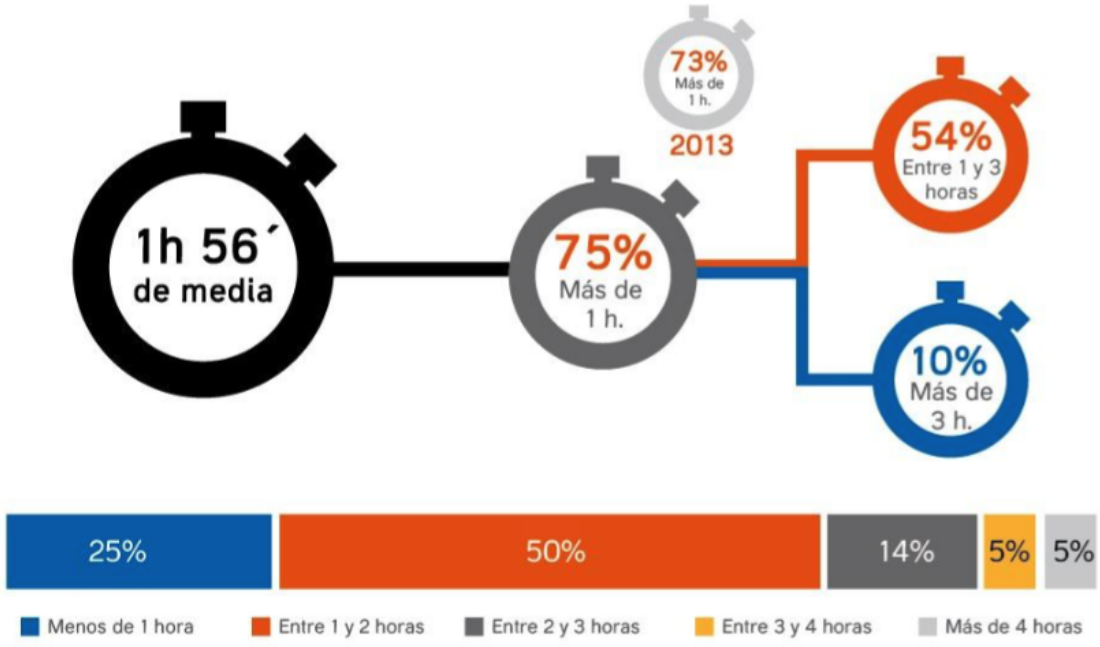


Imagen 3. Tiempo de uso medio diario de una aplicación móvil

1.1.2 Aplicaciones de Moda

El uso de aplicaciones móviles relacionadas especialmente con la moda es un sector que está creciendo de manera exponencial, gracias a la expansión de tiendas online que han dado el saldo de la web a las *apps* móviles. También las redes sociales como Instagram se convierten en foco de difusión para famosos o marcas que puedan mostrar productos relacionados con la moda.

La idea principal de Tendfy no es hacer un escaparate más de productos para que el usuario suba y baje viendo fotografías de calzados. La aplicación tiene como parte

principal la interacción de usuario con cada uno de los productos que le puedan interesar. Utilizar la aplicación se convierte en una especie de juego.

Mediante cartas, el usuario irá seleccionando los productos que le parecen interesantes y los que no. Esto le da información a la *app* para conocer los gustos que tiene el usuario y poder mostrarle productos en función de sus gustos, al contrario que una aplicación de escaparate en la cual vemos decenas o cientos de productos y simplemente nos vamos moviendo entre secciones para verlos.

Imagen 4. App tipo Escaparate

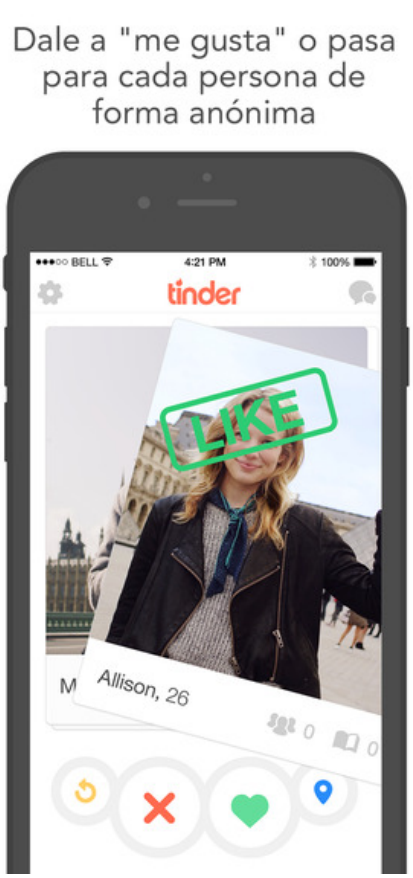


Imagen 5. App tipo Cartas

Además gracias al acceso a internet contínuo del que disponen la mayoría de los *SmartPhones* en la actualidad, el usuario puede acceder a la información de la *app* en cualquier momento. Además de la posibilidad de utilizar la geolocalización para indicar la posición de un producto al usuario.

1.1.3 Aplicación sobre zapatos

La moda es un sector que no desaparecerá ya que todos necesitamos ropa/calzado para cualquier situación de nuestra vida cotidiana, esto lo convierte en una apuesta segura para desarrollar una idea.

Tendfy se centra en la idea de la moda del calzado, especialmente en el calzado femenino. La compra de calzado femenino es mucho mayor que en el sector masculino, por lo que lo hace un nicho atractivo en el cual focalizar la idea de aplicación.

1.2 Objetivos del proyecto

La realización de este proyecto era un reto más que interesante para mí, aunque no fue sencillo y el esfuerzo siempre fue en mayor escala, la motivación por realizarlo consiguió que lo finalizara de forma exitosa.

Los objetivos son los siguientes:

- Como objetivo principal se desea conseguir un MVP *(Minimum Viable Product),* con el cual poder lanzar una primera versión de la *app* en la *AppStore*, con esto conseguir financiación para la *StartUp*.
- El otro objetivo es el poder tener un código organizado y preparado para su mantenimiento en un futuro. Mediante el uso de patrones de diseño, podemos desarrollar un código estructurado y flexible.

1.3 Inicios del equipo y del proyecto

El proyecto se comenzó a desarrollar con otro desarrollador, el cual disponía de más conocimientos sobre el lenguaje principal utilizado (*Swift*). En el inicio de la aplicación las tareas que me ocupaban era la programación de la Vista y la Lógica de Negocio, dejando la conexión con el Back-End para el otro programador.

El tiempo que trabajamos juntos intentamos utilizar una metodología ágil para la asignación de tareas (*Scrum*). Mediante un panel nos preparamos una serie de objetivos en un tiempo estimado de una semana, con el fin de poder saber en todo momento en que estaba trabajando cada uno sin pisarnos. Esto hacía que el desarrollo de la aplicación fuera más rápido y sencillo.

Como nexo para el código utilizamos un repositorio en *Bitbucket*, el cual nos permitía unir nuestro código en cualquier momento y poder corregir posibles conflictos en poco tiempo. Todo esto nos facilitó el desarrollo en paralelo de la aplicación.

2. Estudio del proyecto

Antes de iniciar un proyecto de StartUp, es necesario valorar el entorno sobre el cual irá dirigido dicho proyecto. Aquí hablaremos sobre lo que se tendría que tener en cuenta al iniciar algo de este tipo

2.1 Competidores

En este punto, veremos *apps* competidoras directas e indirectas de Tendfy para después mostrar una comparación de puntos favorables y negativos entre nuestra *App* y los competidores.

Como idea principal no hay otra *App* que permita encontrar calzado de la misma manera que lo hace la nuestra, pero sí existen otras que permiten ver contenido de la misma manera. Estas otras aplicaciones será sobre las que haremos un pequeño análisis.

Como ya se ha dicho anteriormente, no hay ninguna App que permita jugar al Truc como la nuestra, pero sí que existen otros juegos para jugar al Truco (la versión Sudamericana). Estas serán sobre las que haremos un pequeño análisis.

2.2 Aplicaciones similares

2.2.1 Mencanta

La primera aplicación que vamos a analizar se llama Mencanta. Un rival sin duda muy potente que cuenta con más de 1 millón de descargas. Desarrollado por otra StartUp Valenciana, premiada por empresas como Telefónica. Esta aplicación lleva publicada más de dos años.

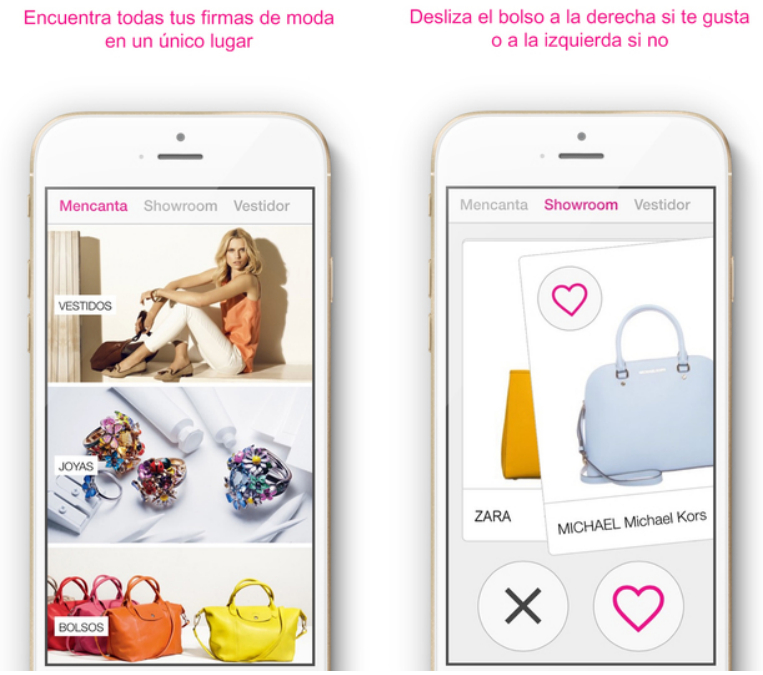


Imagen 6. Mencanta

Imagen 7. Mencanta

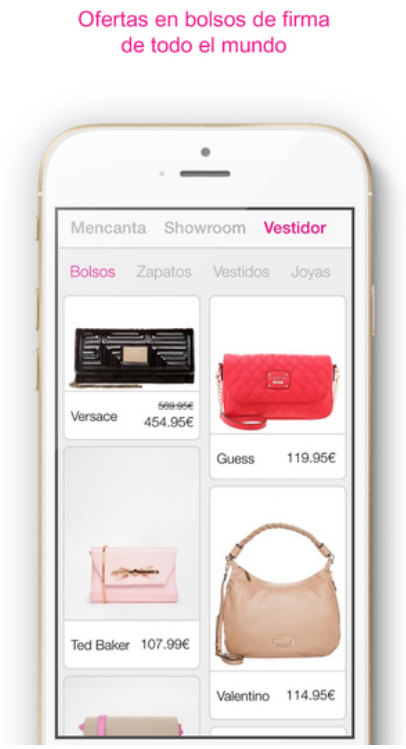


Imagen 8. Mencanta

Imagen 9. Mencanta

Mencanta no se centra en un solo sector de la moda, simplemente recoge los productos de todo tipo y los muestra en diferentes formatos. La aplicación se divide en tres secciones principales:

- **Mencanta**: Nos permite ver un *feed* de noticias sobre tendencias (imagen 6), descuentos etc... También podemos filtrar en este *feed* con una categorías según el producto del que prefiramos leer noticias.
- **Showroom**: Aquí reside el factor que sitúa la aplicación como competidor de manera más cercana. Se trata de un selector de productos los cuales se pueden filtrar y ver información sobre estos. El modo de previsualización de los productos es en modo carta (imagen 5 y 7). También desde esta sección permite la compra del producto, pero siempre mandando al usuario a la web del fabricante.
- **Vestidor**: Muestra los productos que el usuario ha marcado como favorito y los muestra en un formato escaparate para poder ver la información sobre el producto o acceder a su compra (imagen 8 y 9).

En las imágenes no se puede apreciar pero es una aplicación híbrida, es decir que no se ha realizado de forma nativa en los dispositivos para los que se lanzó. Probablemente los creadores tomaron la decisión de seguir este camino para desarrollar su aplicación debido al prisa por lanzar el producto. Esto tiene una serie de inconvenientes como la

lentitud de carga de la aplicación, un estilo no definido por el sistema para el que va dirigido, lo que hace que se vea de manera extraña al resto del dispositivo.

La aplicación permite la compra de productos pero siempre dirigiendo al usuario a la web del vendedor, no vendiendo directamente. Esto tiene inconvenientes como que el usuario tiene que registrarse y configurar su cuenta para poder comprar productos de varias marcas diferentes.

2.2.2 Stylect

Continuamos analizando la competencia directa con Stylect, esta aplicación todavía superar el nivel de potencial que Mencanta. Cuenta con más de 3 millones de descargas y ha sido financiada en varias ocasiones permitiendo que el proyecto crezca todavía más. La empresa creadora es británica con tres años a las espaldas. Al igual que Tendfy se enfoca en el calzado femenino.

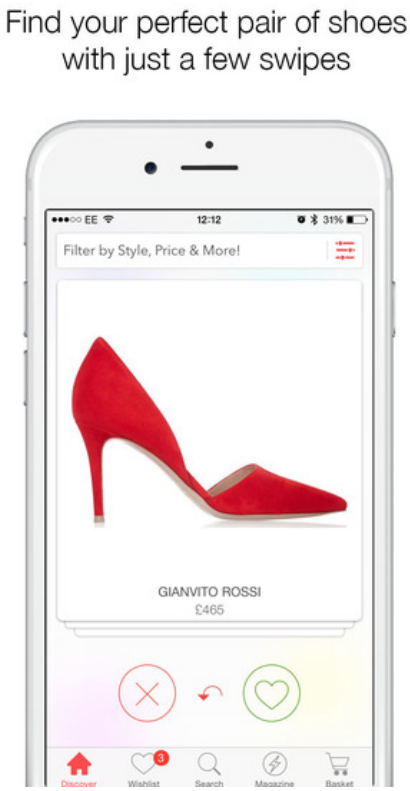


Imagen 10. Stylect

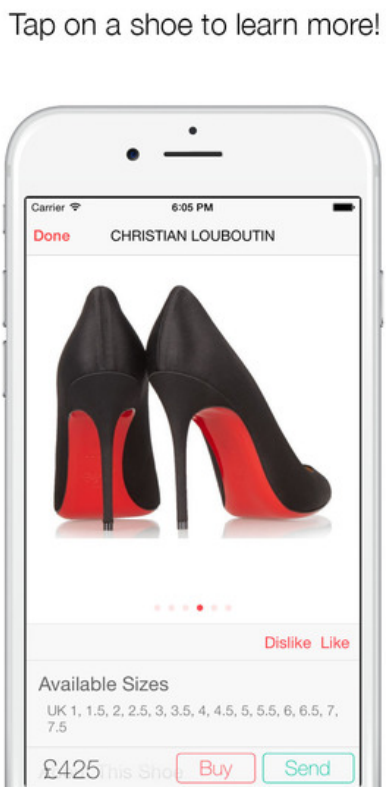


Imagen 11. Stylect

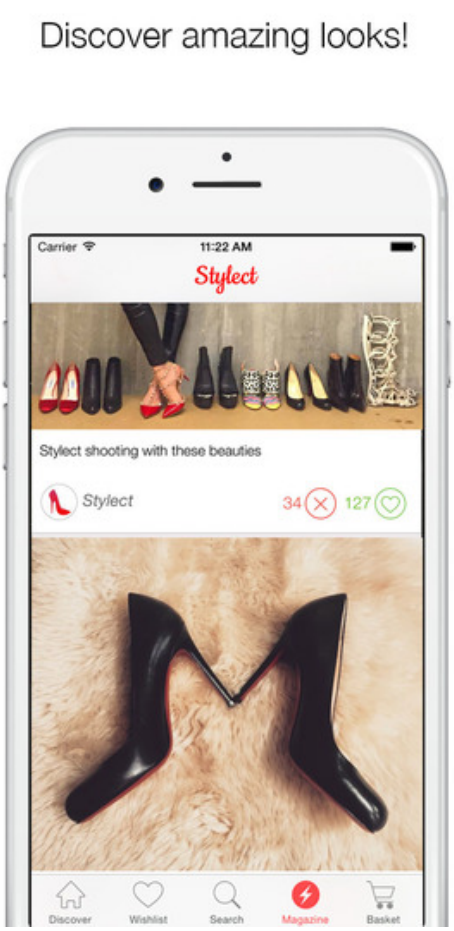


Imagen 12. Stylect

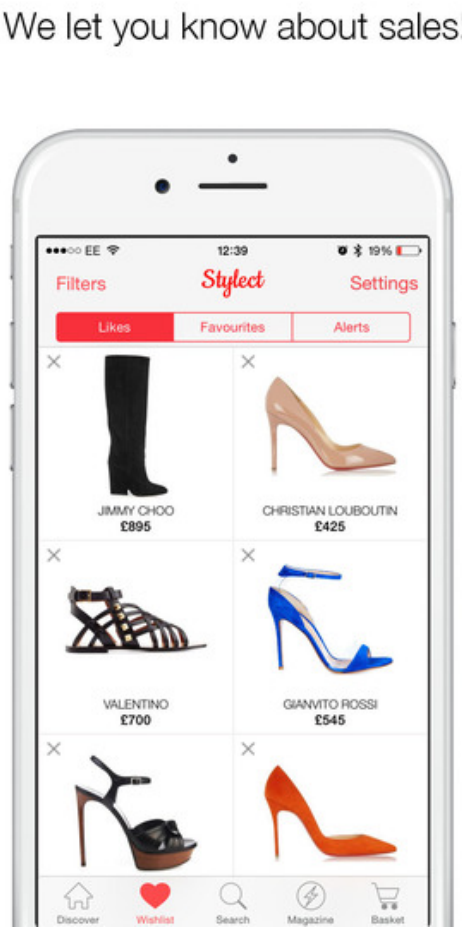


Imagen 13. Stylect

Stylect como ya se mencionó en la descripción de la *App*, se centra en el sector del calzado. Ellos también optan por la opción de las cartas como punto principal de encuentro con los productos. La aplicación se divide en cinco secciones principales:

- **Discover**: Desde esta sección volvemos a ver el selector de cartas (imagen 10) para interactuar con los productos, deslizándose hacia un lado u otro, dependiendo de si nos gusta o no.
- **Shop**: Muestra los contenidos de la sección discover, pero en forma de escaparate, implementando las mismas funcionalidades (imagen 13).
- **Magazine**: Nos permite ver un *feed* de noticias sobre tendencias (imagen 12), descuentos etc... También podemos filtrar en este *feed* con una categorías según el producto del que prefiramos leer noticias.
- **Wishlist**: Muestra los productos que el usuario ha marcado como favorito y los muestra en un formato escaparate para poder ver la información sobre el producto o acceder a su compra (imagen 13).

La aplicación es completamente nativa por lo que la calidad es mayor que en la aplicación antes descrita.La aplicación permite la compra de productos pero siempre dirigiendo al usuario a la web del vendedor, no vendiendo directamente. Esto tiene

ciertos inconvenientes, como que el usuario tiene que registrarse y configurar su cuenta para poder comprar productos de varias marcas diferentes.

2.3 Ventajas sobre los competidores

De estos dos competidores más directos que hemos analizado, es obvio que el más potente es Stylect. Es una *App* muy completa de la cual se sacaron varias ideas.

Veamos una comparación de nuestra aplicación y la de estos competidores. Analizaremos qué características tiene Tendfy frente a ellos. Pero cabe destacar que tenemos una ventaja exclusiva sobre todos que nos coloca por encima: nuestra aplicación es la única que permite la guía al usuario hasta el local más cercano donde comprar esos zapatos que busca, ya que muchos usuarios prefieren probarse el calzado antes de comprarlo, sobretodo cuando el caldazo tiene un precio elevado.

	Tendfy	Mencanta	Stylect
App Nativa	✅	❌	✅
Compra Online o Reserva Online	✅	✅	✅
Social	✅	❌	❌
Interacción Directa con el Producto	✅	✅	✅
Feed Noticias	❌	✅	✅
Filtro	✅	✅	✅
Geolocalización	✅	❌	❌

Leyenda: ✅ Disponible ❌ No Disponible

Como se puede observar, Tendfy tiene la mayoría de características que posee nuestro competidor más potente, añadiendo algunos extras que le dan mayor potencial viral y útil a la aplicación. Por lo que se concluye con este primer análisis que, Tendfy es el producto más completo del mercado.

2.4 Análisis/Estudio de Modelo de Negocio

Vamos a mostrar el estudio del proyecto. Empezaremos por el Lean Canvas, seguido del DAFO y analizaremos la proyección económica de la App para finalizar.

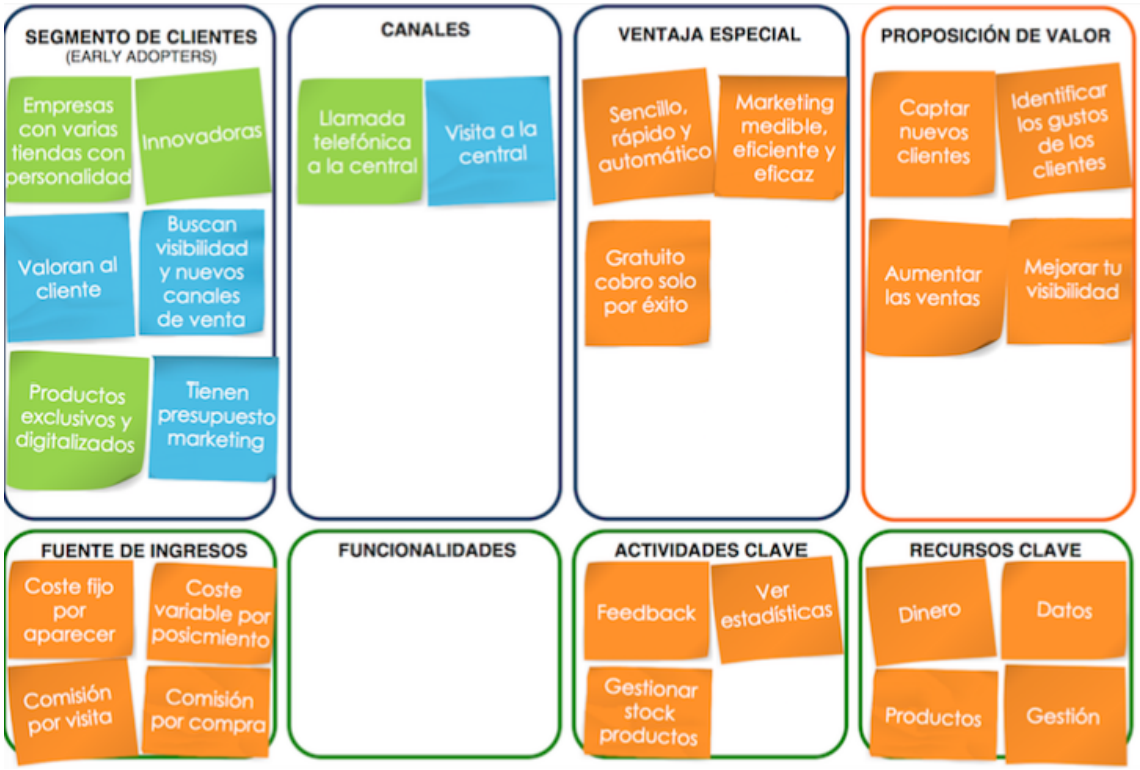
Primero definiremos brevemente qué son:

- **Lean Canvas**: Esta es una de las herramientas más recurridas por *StartUps*, permite crear y comunicar modelos de negocio de manera rápida y eficaz. Introduciendo un poco de historia sobre Lean Canvas, podemos decir que su creador fué Ash Maurya, inspirado en Canvas Business Model y la metodología Lean.

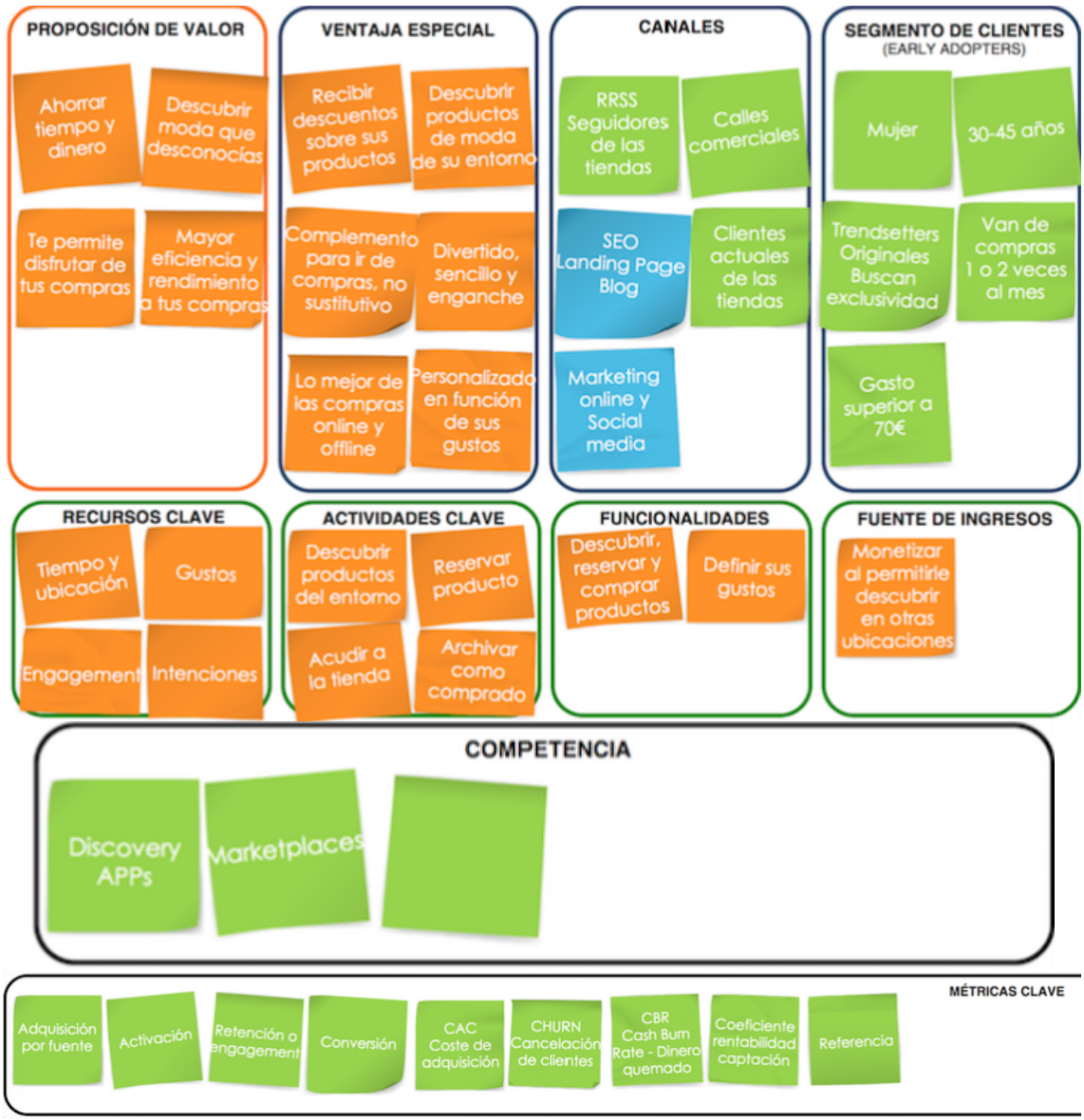
- **Análisis DAFO**: Definida como Debilidades Amenazas Fortalezas y Oportunidades, analiza tanto las características internas como las externas. Esta herramienta muestra la situación real en la que se haya una empresa o proyecto. Con esta herramienta determinamos las ventajas competitivas de la empresa bajo análisis.

- **Proyección Económica**: Se trata de un análisis en dos puntos: ingresos y tiempo. Su objetivo principal es estimar cuáles pueden ser los ingresos del proyecto y si puede salir rentable en un periodo de tiempo determinado, así como prever el crecimiento/decrecimiento de los ingresos.

2.4.1 Lean Canvas




PROBLEMAS
Han caído sus ventas, han perdido clientes
Mala visibilidad online
Escasa variedad de productos
Pérdida de tiempo y desplazamts innecesarios
SOLUCIÓN
Una aplicación móvil
Te permite descubrir la moda que te gusta
de las tiendas más exclusivas de tu ciudad
mientras disfrutas yendo de compras
ESTRUCTURA DE COSTES
Asesrmiento legal y fiscal
Métricas
Alojamiento y dominio
Marketing


PROPOSICIÓN DE VALOR
Ahorrar tiempo y dinero
Descubrir moda que desconocías
Te permite disfrutar de tus compras
Mayor eficiencia y rendimiento a tus compras
VENTAJA ESPECIAL
Recibir descuentos sobre sus productos
Descubrir productos de moda de su entorno
Complemento para ir de compras, no sustitutivo
Divertido, sencillo y enganche
Lo mejor de las compras online y offline
Personalizado en función de sus gustos
CANALES
RRSS Seguidores de las tiendas
Calles comerciales
SEO Landing Page Blog
Clientes actuales de las tiendas
Marketing online y Social media
SEGMENTO DE CLIENTES
(EARLY ADOPTERS)
Mujer
30-45 años
Trendsetters Originales Buscan exclusividad
Van de compras 1 o 2 veces al mes
Gasto superior a 70€
RECURSOS CLAVE
Tiempo y ubicación
Gustos
Engagement
Intenciones
ACTIVIDADES CLAVE
Descubrir productos del entorno
Reservar producto
Acudir a la tienda
Archivar como comprado
FUNCIONALIDADES
Descubrir, reservar y comprar productos
Definir sus gustos
FUENTE DE INGRESOS
Monetizar al permitirle descubrir en otras ubicaciones
COMPETENCIA
Discovery APPs
Marketplaces
MÉTRICAS CLAVE
Adquisición por fuente
Activación
Retención o engagement
Conversión
CAC Coste de adquisición
CHURN Cancelación de clientes
CBR Cash Burn Rate - Dinero quemado
Coeficiente rentabilidad captación
Referencia

2.4.1 DAFO

Imagen 14. Análisis DAFO

Como se observa las fortalezas del proyecto pueden ayudar de manera cuantiosa a los ingresos de la empresa, lo cual permitiría hacer más grande el equipo y absorber las debilidades de las que puede pecar.

Las principales debilidades vienen principalmente por el tamaño de la empresa lo que nos impide tener a gente dedicada al sector especializados como las *RRSS*, comerciales para captar empresas que quieran aparecer en la aplicación.

En cuanto a las oportunidades la fácil adaptación que se planteó en un inicio a la posibilidad de cambiar de tecnología es existente gracias a la ya nombrada planificación que se realizó en un principio.

2.4.2 Proyección Económica

Retruque! — **Primer año de vida de la App**

Costos Fijos

Descripcion	Detalle	Gasto Mensual	Gasto Anual
Alta desarrollador Apple	**Darse de Alta como desarrollador Apple**	0,00	**100,00**
Diseñador Gráfico	**Realizar diseños interfaz de la aplicación**	0,00	**50,00**
Publicidad Online	**Coste por click**	**20,00**	**240,00**
	TOTAL	**20,00**	**390,00**

Ingresos

Descripcion	Detalle	Ingreso Mensual	Ingreso Anual
Descargas	0,02 por cada descarga (100 mensuales)	20,00	240,00
Publicidad	(100000 impresiones, 5% CTR, 0,12/click)	80,00	960,00
	TOTAL	80,00	1200,00
	TOTAL Ingreso - Gastos	60,00	885,00

Imagen 15. Proyección Económica Primer Año

En este primer periodo nos encontramos como aplicación nueva, la cual no obtendrá un gran número de descargas y que tendrá una inversión inicial.

Los beneficios recibidos serán de mano de la publicidad y las descargas que obtenga nuestra aplicación, por lo que es esencial la expansión de esta para recuperar los gastos y obtener beneficio.

3. Contexto Tecnológico

En esta sección se describen las técnicas y tecnologías de las que se han hecho uso en el proyecto. Por un lado presentamos los patrones de diseño, que se han utilizado para lograr un diseño de la aplicación que satisfaga los criterios de calidad de mantenimiento y legibilidad que se marcaron como objetivo para el desarrollo de la aplicación. Por otro lado presentamos distintas tecnologías y lenguajes de programación utilizados para la implementación de la app.

El proyecto ha llevado aproximadamente 7 meses de desarrollo hasta su versión *MVP*, un año y medio desde su inicio. A continuación se explicará en que se ha invertido todo este tiempo.

La idea surgió cuando Álvaro el fundador se unió con su actual socio Quique, con el sueño de ser emprendedores. Buscaban un producto novedoso, que les gustara hacer y que fuera completamente emprendedor. Tras pensar en varias alternativas, decidieron hacer una empresa sobre moda centrada en el sector del calzado.

Durante los primeros meses se dedicaron a realizar encuestas a personas de la calle para ver qué necesidades tenían con el sector de la moda y cómo podría solucionarlo. Hasta dar por fin con la fórmula que buscaban focalizaron la idea en una empresa que sirviera como nexo para encontrar calzado de mujer mayoritariamente.

Después de estos análisis de mercado se presentaron a una lanzadera la cual accedió a realizar una financiación inicial para comenzar con la idea de proyecto. En este momento fue cuando entré a trabajar en el proyecto mientras terminaba la carrera. Entre como desarrollador *iOS* en *Swift*, con ayuda del desarrollador del que ya disponían también socio de la empresa.

Teniendo claras las tecnologías que queríamos utilizar y cómo tenía que ser nuestra *App*, comenzamos finalmente a programar *Xcode*. Este proceso dura desde esta etapa hasta la publicación de la App en la *AppStore*.

Poco después se decidió mover la empresa al centro de emprendimiento de la UPV, donde podríamos trabajar con otros emprendedores y aprender aún más sobre el sector.

Decidimos comenzar registrando a los usuarios mediante Facebook ya que permitía un registro rápido y nos daba toda la información necesaria de los usuarios, este paso fue costoso, ya que tuvimos que aprender cómo funcionaba *Facebook Apps* y cómo podríamos comunicarnos con su API.

Ya claro el funcionamiento de esta tecnología, fuimos avanzando hasta conseguir terminar el inicio de sesión, carga de datos de usuario y cerrado de sesión, ahora queríamos abordar la obtención de datos del calzado para mostrarlo en la aplicación.

En un principio en planteamiento de esta parte del proyecto parecía muy compleja, sin embargo, nos costó bastante menos de lo que creíamos, debido a los conocimientos

que habíamos aprendido los dos desarrolladores de las tecnologías web en antiguos trabajos.

Cuando se consiguió acabar la visualización de contenidos del servidor dentro de nuestra app, incluimos también un sistema de localización, para poder saber donde encontrar el producto que nos estaba mostrando la aplicación, así como una mejora notable de la interfaz gráfica con animaciones.

Finalmente implementamos un sistema de reservas muy rústico, pero completamente funcional, basado en varias tecnologías que se explicarán más adelante.

Cuatro meses después de entrar en el proyecto ya disponíamos de un producto usable para el usuario final, el cual se publicó en la tienda de aplicaciones de Apple. Tras esta publicación, nos centramos en el mantenimiento de la aplicación y la refactorización del código de esta, en el cual se centra este proyecto.

Durante este último periodo decidí realiza una reestructuración de la aplicación, utilizando patrones de diseño. Utilizando estos diseños de software buscaba un mejor código, que no solo afectara a la aplicación en ese momento, sino que mejorara su mantenimiento en un futuro. Consiguiendo evitar la reiteración de búsquedas de soluciones a complejos que pudieran ir apareciendo, estandarizando el lenguaje para que no solo el desarrollador que hizo el código pudiera entenderlo fácilmente.

3.1 Patrones de Diseño

Un patrón de diseño es la descripción de un problema que ocurre de manera reiterada en nuestro entorno, así como una solución a ese problema, de tal modo que se pueda aplicar esta solución un millón de veces, sin repetir lo mismo dos veces.

Esta definición puede ser aplicada no solo al diseño de software si no a cualquier tipo de patrón en cualquier situación.

3.1.1 ¿Qué compone un Patrón de Diseño?

Un Patrón de Diseño debe tener estos elementos esenciales:

- **Nombre**: Es la parte que permite describir al patrón, normalmente se utilizan una o dos palabras para ello. Al utilizar nombres de este tipo, nos permite hablar de ellos, documentarlos. De esta manera, resulta más fácil pensar en estos diseños y poder transmitirlos a otros. Quizás esta sea la parte más difícil de la creación de un patrón de diseño.
- **Problema**: Debe describir cuándo aplicar el patrón. Explica el problema y en el contexto en el que se haya. Puede ser la descripción de un problema en concreto de diseño, así como estructuras de clases u objetos que puedan ser síntoma de un diseño poco flexible. El problema puede incluir una serie de condiciones que deben darse para que el patrón tenga sentido.
- **Solución**: Describe los elementos que forman el diseño, las relaciones, responsabilidades y colaboraciones. No describe un diseño o una implementación concreta, sino que un patrón es como una plantilla que se puede aplicar en muchas diferentes situaciones. El patrón proporciona una

descripción abstracta de problemas de diseño y cómo se resuelve una disposición general de elementos.

- **Consecuencias**: Son los resultados, tanto las ventajas como inconvenientes de aplicar el patrón. Las consecuencias son fundamentales para poder evaluar las posibles alternativas de diseño y comprender los costes y beneficios de aplicarlo. Ya que la reutilización suele ser uno de los factores del diseño orientado a objetos, las consecuencias que tiene un patrón incluyen su impacto sobre la flexibilidad, extensibilidad y portabilidad de este en un sistema.

3.1.2 ¿Cómo se describe un Patrón de Diseño?

Utilizar anotaciones gráficas, no son suficientes a la hora de describir un patrón, aunque sean importantes para su descripción. Para la reutilización de un diseño, es necesario hacer constar las decisiones, alternativas y ventajas e inconvenientes que dieron lugar al patrón. También es importante dar ejemplos concretos, que ayuden a ver el diseño aplicado a una situación real.

En la descripción de patrones existe un formato consistente. Cada patrón será dividido en secciones de acuerdo con la plantilla que explicaré a continuación. Esta estructura le dará uniformidad a la información, haciendo que el patrón sea más fácil de aprender y utilizar.

Plantilla:

- **Nombre del patrón y clasificación**: El nombre del patrón debe describir de manera clara el sentido del patrón. Su clasificación depende de su nivel de abstracción, relacionando estas clasificaciones asignadas como familias entre patrones.
- **Propósito**: Esta parte debe ser breve, y debe responder a qué función realiza el patrón, en qué se basa y cuál es el problema concreto que soluciona.
- **También conocido como…**: Informa sobre posibles variaciones en el nombre que permita el reconocimiento de este.
- **Motivación**: Es un escenario ilustrativo sobre un problema de diseño y como con el uso de estructuras de clases y objetos del patrón, se puede resolver el problema. Este escenario ayuda en la comprensión de la aplicabilidad.
- **Aplicabilidad**: Tiene que resolver las cuestiones de qué situaciones se puede utilizar el patrón, qué ejemplos de mal diseño puede resolver y cómo se pueden reconocer estas situaciones.
- **Estructura**: Es una representación gráfica de las clases del patrón utilizando una notación basada en el Modelado de Objetos (OMT). También mediante el uso de diagramas de interacción para mostrar posibles consecuencias de peticiones y colaboraciones entre los distintos objetos.
- **Participantes**: Clases y objetos participantes en el patrón, junto con sus responsabilidades en este.
- **Colaboraciones**: De qué manera colaboran los participantes para realizar sus responsabilidades.
- **Consecuencias**: Responde a de qué manera consigue completar sus objetivos el patrón, cuales son las ventajas e inconvenientes y el resultado que ofrece el

uso del patrón, también qué aspectos de la estructura del sistema está permitido modificar de manera independiente.
- **Implementación**: La dificultad, trucos o técnicas que se deben tener en cuenta cuando se aplique el patrón y si existen cuestiones específicas del lenguaje.
- **Código de ejemplo**: Fragmentos de código que muestre cómo se implementa el patrón.
- **Usos conocidos**: Posibles entornos reales en los que podemos utilizar el patrón.
- **Patrones relacionados**: La relación que tiene el patrón con otros patrones y cuales son las diferencias con estos. También con qué otros patrones se debería de utilizar.

También cabe la posibilidad de añadir un apéndice que de información adicional para ayudar a entender el patrón.

3.1.4 Patrones que utilizaremos

Los patrones que se han aplicado al diseño de software de la aplicación están basados en el libro Gang of Four (GoF) de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides. Estos patrones que se describen en el libro son los más importantes y fundamentales en el desarrollo de software moderno. Además también se aplica el conocido patrón arquitectónico MVC y el patrón Object Template ambos explicados en el libro *Pro Design Patterns in Swift* de Adam Freeman.

Se decidió estudiar y utilizar patrones para el desarrollo de una aplicación móvil como es la que trata esta memoria. Todos estos patrones serán orientados hacia el lenguaje Swift, que es con el que se ha desarrollado la aplicación

Primero se nombrará las características y utilidades de cada uno de los patrones que se han utilizado, posteriormente el código existente antes de la refactorización, finalmente la demostración de cómo se ha aplicado el patrón.

3.1.4.1 Patrón Objeto Plantilla (Object Template)

3.1.4.1.1 ¿Qué es?

Este patrón utiliza una clase o estructura como especificación del tipo de datos y la lógica que recibe este tipo de dato. Los objetos son creados utilizando una plantilla, los valores de estos datos son insertados durante su inicialización, ya sea mediante el uso de valores por defecto en la plantilla o proporcionados por el componente de la clase o inicializador de la estructura, también conocido como constructor.

3.1.4.1.2 ¿Qué beneficios tiene?

Proporcionar la base para agrupar los valores de los datos y la lógica de quien los manipula en conjunto, este método también es conocido como encapsulación. La encapsulación permite a un objeto representar una API para sus consumidores al tiempo que oculta la implementación privada de esa API. Esto ayuda a prevenir el acoplamiento de componentes.

3.1.4.1.3 ¿Cuándo utilizar este patrón?

En cualquier proyecto, aunque no es recomendable en los que sean muy simples, ya que añadirían una complejidad extra, que no es necesaria. Para este tipo de proyectos simples se pueden utilizar las tuplas de Swift, ya que son una característica interesante, pero cuando el proyecto tiene una envergadura mayor pueden presentar un problema de mantenimiento a largo plazo. Utilizar este patrón sólo se requiere un poco de trabajo extra para crear una clase muy simple o una estructura.

3.1.4.1.4 ¿Cuándo evitar este patrón?

Debido a su simpleza, no habría porqué evitarlo, salvo en el ejemplo que puse anteriormente de un proyecto muy simple.

3.1.4.1.5 ¿Cómo saber que se está implementando de forma correcta?

Cuando se pueden realizar cambios en la implementación privada de una clase o estructura sin hacer los cambios en todos los componentes que lo utilizan.

3.1.4.1.6 Código previo al patrón

Para explicar este patrón utilizaremos el objeto usuario, el cual tiene múltiples atributos con distintos tipos de datos.

El concepto más básico de estructura que podemos utilizar para este caso es una tupla, la cual permite almacenar datos de múltiples tipos en un solo lugar y poder ser accesibles en un solo bloque.Ante esto podemos encontrarnos con el problema que suele conllevar el desarrollo de software, se trata de ampliar funcionalidades.

Al hacer esto necesitaremos realizar cambios en cualquier parte del código que utilice esta tupla con la información del usuario.

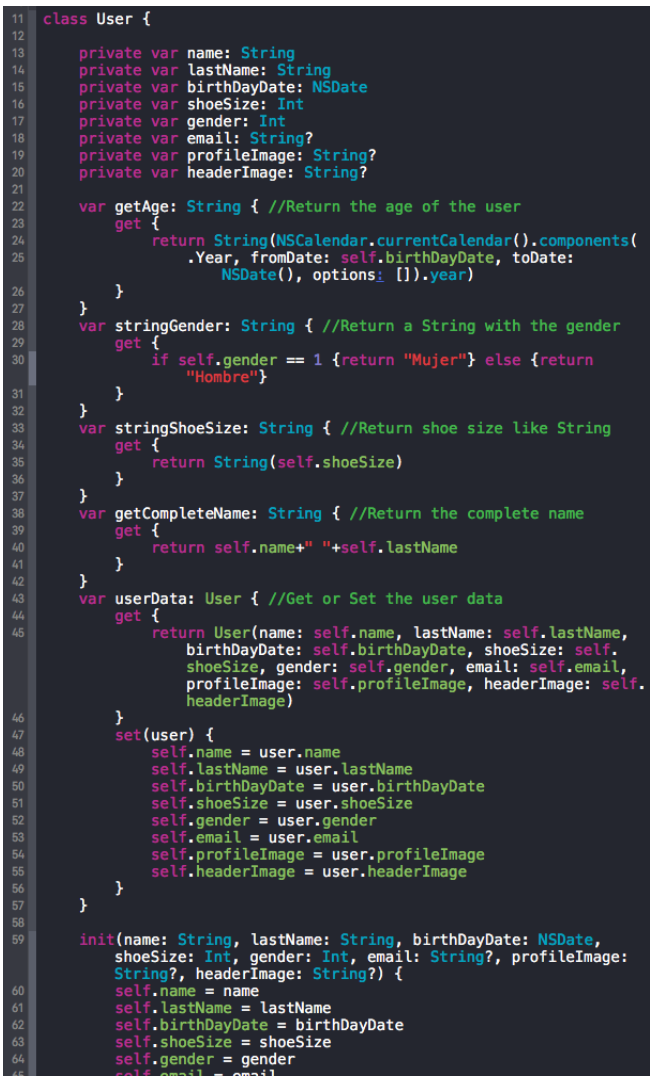
```
9
10  var user = [
11      ("Sergio", "Fresneda", "05-11-1990", 41, "Hombre", "mail@mail.es",
            "profileImage.jpg", "headerImage.jpg")
12  ]
13
```

Imagen 16. Tupla usuario

No sólo es la dificultad de modificación que tiene este tipo de código, si no también lo dificultoso que resulta su lectura cuando no se ha diseñado desde su inicio.

3.1.4.1.7 Demostración del patrón

Utilizando el patrón Objeto Plantilla hemos creado un objeto mucho más complejo con las mismas propiedades que el anterior, permitiendo el acceso a sus propiedades mediante unos *get* que realizan la función de API.



```
class User {

    private var name: String
    private var lastName: String
    private var birthDayDate: NSDate
    private var shoeSize: Int
    private var gender: Int
    private var email: String?
    private var profileImage: String?
    private var headerImage: String?

    var getAge: String { //Return the age of the user
        get {
            return String(NSCalendar.currentCalendar().components(
                .Year, fromDate: self.birthDayDate, toDate:
                    NSDate(), options: []).year)
        }
    }
    var stringGender: String { //Return a String with the gender
        get {
            if self.gender == 1 {return "Mujer"} else {return
                "Hombre"}
        }
    }
    var stringShoeSize: String { //Return shoe size like String
        get {
            return String(self.shoeSize)
        }
    }
    var getCompleteName: String { //Return the complete name
        get {
            return self.name+" "+self.lastName
        }
    }
    var userData: User { //Get or Set the user data
        get {
            return User(name: self.name, lastName: self.lastName,
                birthDayDate: self.birthDayDate, shoeSize: self.
                shoeSize, gender: self.gender, email: self.email,
                profileImage: self.profileImage, headerImage: self.
                headerImage)
        }
        set(user) {
            self.name = user.name
            self.lastName = user.lastName
            self.birthDayDate = user.birthDayDate
            self.shoeSize = user.shoeSize
            self.gender = user.gender
            self.email = user.email
            self.profileImage = user.profileImage
            self.headerImage = user.headerImage
        }
    }

    init(name: String, lastName: String, birthDayDate: NSDate,
        shoeSize: Int, gender: Int, email: String?, profileImage:
        String?, headerImage: String?) {
        self.name = name
        self.lastName = lastName
        self.birthDayDate = birthDayDate
        self.shoeSize = shoeSize
        self.gender = gender
```

Imagen 17. Patrón Objeto Plantilla

No solo la lectura del código ha mejorado, también mediante la encapsulación hemos reducido el número de modificaciones que tienen que realizarse en caso de ampliar la funcionalidad de este objeto en un futuro. El objeto se vuelve más rico en definición, en todo momento se puede conocer el tipo de datos que utiliza y de qué manera se almacena, sin necesidad de ningún tipo de documentación.

3.1.4.2 Patrón Prototipo (Prototype)

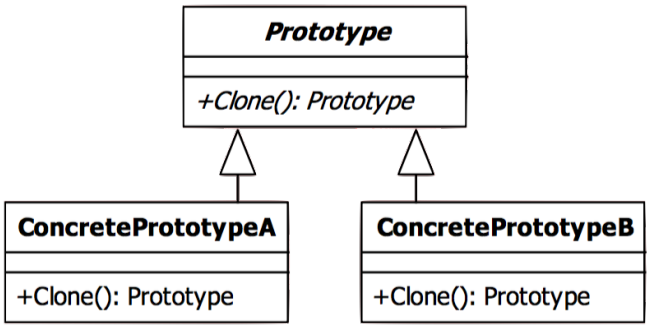


Imagen 18. Diagrama patrón prototipo

Este patrón se puede complementar con el objeto plantilla, lo demostraremos con un ejemplo.

3.1.4.2.1 ¿Qué es?

El patrón prototipo crea nuevos objetos copiando un objeto existente, conocido como prototipo.

3.1.4.2.2 ¿Qué beneficios tiene?

El principal beneficio es poder ocultar el código que crea objetos a partir de los componentes que utilizan; esto significa que los componentes no necesitan saber qué clase o estructura se requiere para crear un nuevo objeto, no necesitan saber los detalles de inicializadores, y no tienen que cambiar cuando se crean y se instancian subclases. Este patrón también se puede utilizar para evitar la repetición de las inicializaciones cada vez que un nuevo objeto de un tipo específico que sea.

3.1.4.2.3 ¿Cuándo utilizar este patrón?

Resulta muy útil cuando se está escribiendo un componente que se necesita para crear nuevas instancias de objetos sin crear una dependencia en el inicializador de la clase.

3.1.4.2.4 ¿Cuándo evitar este patrón?

No habría porqué evitar este patrón.

3.1.4.2.5 ¿Cómo saber que se está implementando de forma correcta?

La aplicación de este patrón es correcta, cuando al cambiar el inicializador de la clase o estructura utilizada como objeto prototipo, no requiere un cambio correspondiente en el componente que crea clones. Otra forma de verificarlo, sería crear una subclase de la clase de prototipo y comprobar de nuevo de que el componente puede clonar sin requerir ningún cambio.

3.1.4.2.6 Código previo al patrón

Para el ejemplo de este patrón podemos tomar como código previo el mismo ejemplo que en el patrón Objeto plantilla con algunas modificaciones.

```
86
87  var anonimousUser = [(name:"", lastname:"", birthDayDate: NSDate(), shoeSize:0,
        gender: 0, email:"", profileImage: "", headerImage:"")]
88
89  print("Complete name: \(anonimousUser[0].name) \(anonimousUser[0].lastname)")
90  //"Complete name:  \n"
91
92  var registeredUser = [(name:"Sergio", lastname:"Fresneda", birthDayDate: NSDate(),
        shoeSize:41, gender: 0, email:"mail@mail.es", profileImage:
        "profileImage.jpg", headerImage:"headerImage.jpg")]
93
94  print("Complete name: \(registeredUser[0].name) \(registeredUser[0].lastname)")
95  //"Complete name: Sergio Fresneda\n"
```

Imagen 19. Tupla usuario anónimo y registrado

Como se puede observar, si queríamos crear varios tipos de usuario teníamos que repetir la estructura base una y otra vez por cada usuario generado. Esto ensucia mucho el código con repeticiones que pueden evitarse utilizando el patrón prototipo.

3.1.4.2.7 Demostración del patrón

Con este patrón podemos continuar el código demostrativo del ejemplo anterior con el patrón objeto plantilla. Ahora continuaremos su implementación para demostrar la utilidad de este patrón.

```
65
66  let anonimousUser : User = User(name: "", lastName: "", birthDayDate: NSDate(),
        shoeSize: 0, gender: 0, email: "", profileImage: "", headerImage: "")
67  print(anonimousUser.getCompleteName)
68  ////"Complete name: \n"
69
70  let registeredUser : User = anonimousUser
71
72  registeredUser.name = "Sergio"
73  registeredUser.lastName = "Fresneda"
74  registeredUser.birthDayDate = NSDate()
75  registeredUser.shoeSize = 41
76  registeredUser.gender = 0
77  registeredUser.email = "mail@mail.es"
78  registeredUser.profileImage = "profileImage.jpg"
79  registeredUser.headerImage = "headerImage.jpg"
80  print("Complete name: \(registeredUser.getCompleteName)")
81  //"Complete name: Sergio Fresneda\n"
82
```

Imagen 20. Patrón Prototipo

Implementando un usuario anónimo o de cualquier otro tipo, podemos clonar este objeto y modificarlo para crear un usuario de otro tipo como se observa en la imagen. No sólo permite el acceso a los atributos del objeto, sino también a sus funciones internas como se muestra con *getCompleteName*. Esto agiliza el desarrollo de objetos que tienen una base igual, pero pueden tener un rol distinto dependiendo de cómo estén formados.

3.1.4.3 Patrón Estrategia (Strategy)

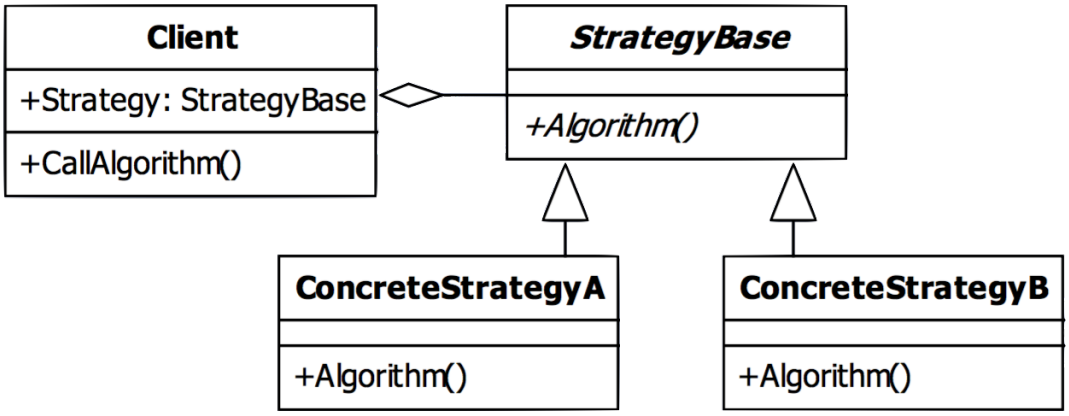


Imagen 21. Diagrama patrón estrategia

3.1.4.3.1 ¿Qué es?

Estrategia se utiliza para crear clases que se pueden extender sin modificación, mediante la aplicación de objetos que se ajustan a un protocolo bien definido.

3.1.4.3.2 ¿Qué beneficios tiene?

Permite a los desarrolladores de terceros cambiar el comportamiento de las clases sin modificarlos directamente, y permiten cambios de bajo costo para hacerse en proyectos que tengan procedimientos de gran coste y longitud para clases específicas.

3.1.4.3.3 ¿Cuándo utilizar este patrón?

Se puede utilizar este modelo cuando sean necesarias clases que se pueden extender sin ser modificadas.

3.1.4.3.4 ¿Cuándo evitar este patrón?

No hay una razón por la que necesitemos evitar este patrón.

3.1.4.3.5 ¿Cómo saber que se está implementando de forma correcta?

Estará implementado correctamente siempre y cuando se pueda extender el comportamiento de una clase mediante su definición y aplicación de una nueva estrategia sin necesidad de realizar ningún cambio en la propia clase.

3.1.4.3.7 Demostración del patrón

Este patrón no dispone de un código previo a esta estrategia debido a lo común que es el uso en este lenguaje. Así que se ilustrará el patrón con un ejemplo y posteriormente se mostrará de qué manera se ha aplicado en la aplicación.

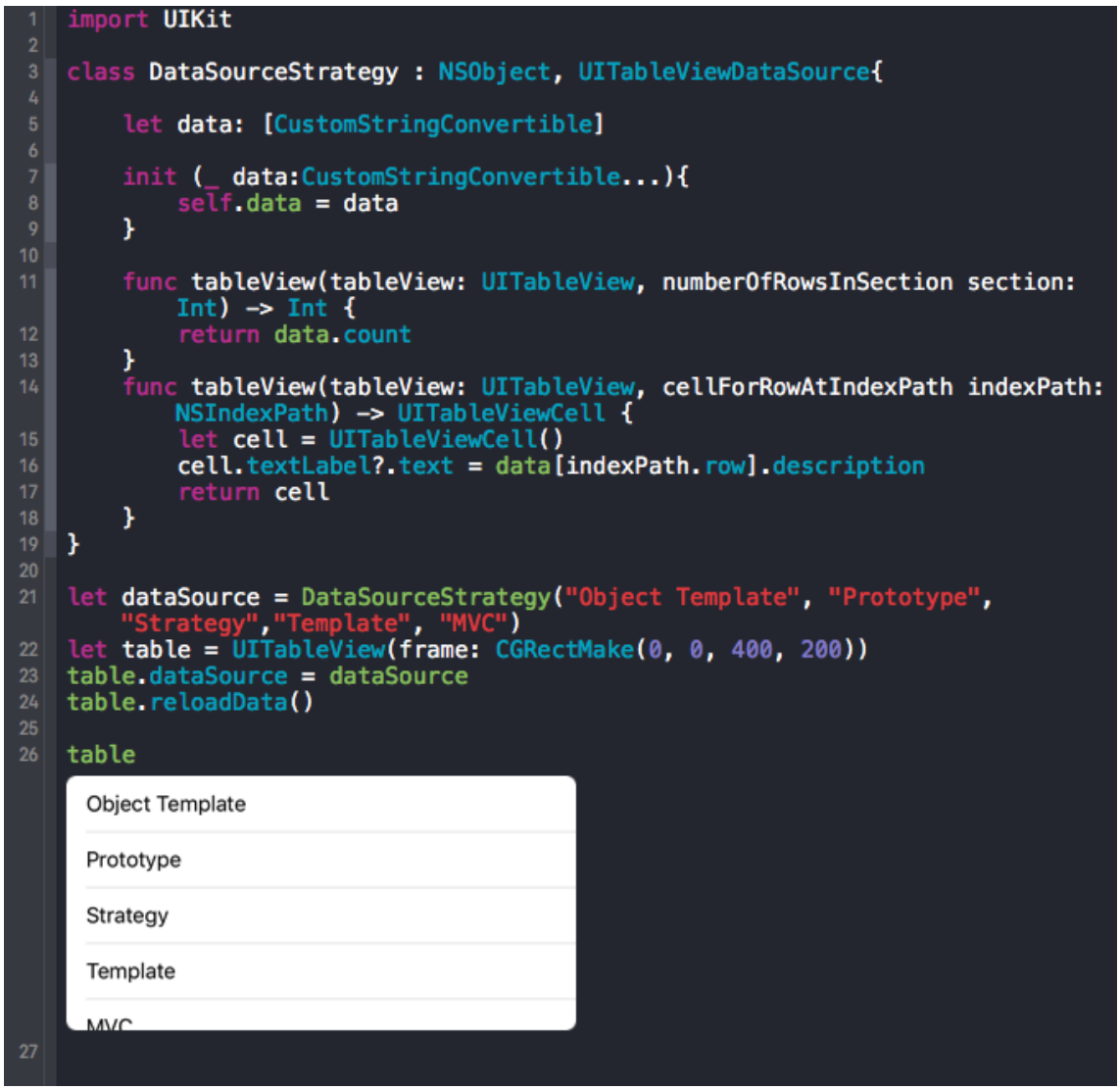


Imagen 22. Patrón estrategia extracto del libro *Pro Design Patterns in Swift*.

Hemos definido una clase llamada DataSourceStrategy que conforma el protocolo de UITableViewDataSource e implementa los dos métodos necesarios para proporcionar valores con un *Array* de objetos CustomStringConvertible. Se ha creado una instancia de la clase de estrategia y se utiliza como *source* para un objeto UITableView, produciendo el resultado se muestra en la parte inferior del código.

En la aplicación se aplicó de una manera muy similar a la mostrada en el ejemplo anterior.

```
class DetailViewController: ViewWithREDNavTemplate, UITableViewDelegate, UITableViewDataSource, MKMapViewDelegate, CLLocationManag

    @IBOutlet weak var productImageView: UIImageView!
    @IBOutlet weak var locationLabel: UILabel!
    @IBOutlet weak var priceLabel: UILabel!
    @IBOutlet weak var discountLabel: UILabel!
    @IBOutlet weak var descriptionLabel: UILabel!
    @IBOutlet weak var mapView: MKMapView!
    @IBOutlet weak var tableView: UITableView!
    @IBOutlet weak var trademarkLogoImageView: UIImageView!

    @IBOutlet weak var closeButton: UIButton!
    @IBOutlet weak var shareButton: UIButton!

    var model: DetailModel = DetailModel()

    var object: Shoe {
        get {return self.model.shoe!}
        set(shoe) {self.model.shoe = shoe}
    }

    // MARK: - Table view data source
    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {

        let cell: DetailTableViewCell = self.tableView.dequeueReusableCellWithIdentifier("DetailCell") as! DetailTableViewCell
        cell.createMenu(titleValue: (self.model.contentDetailArray[indexPath.row]))

        if indexPath.row%2 != 0 {
            cell.singleLabel.textColor = UIColor(rgba: "#b9b9b9")
            cell.customSeparatorView.hidden = true
        }

        return cell
    }
```

Imagen 23. Patrón estrategia en el controlador de la vista de detalle

3.1.4.4 Patrón Plantilla

Este patrón se puede complementar con el estrategia, se mostrará utilizando la misma clase detalle de vista que en el patrón anterior.

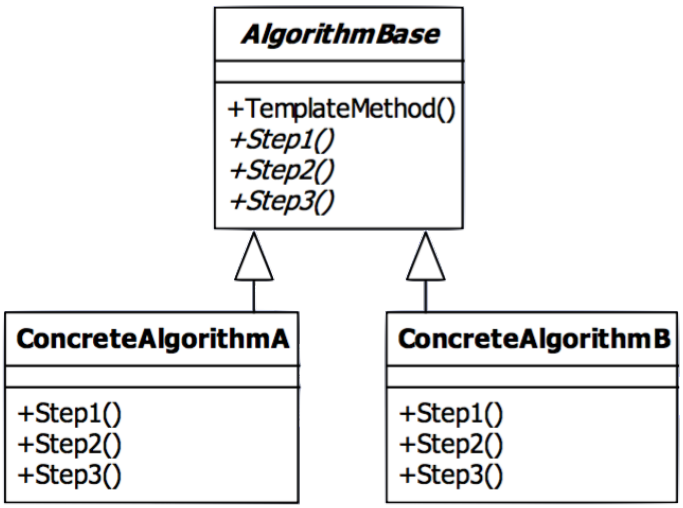


Imagen 24. Diagrama del patrón plantilla

3.1.4.3.1 ¿Qué es?

Este patrón permite modificar un algoritmo para ser sustituido por las implementaciones proporcionadas por un tercero, ya sea mediante la especificación de las funciones como clausuras o creando una subclase.

3.1.4.3.2 ¿Qué beneficios tiene?

Permite que un framework se pueda extender y personalizar por otros desarrolladores que lo incorporen en su aplicación.

3.1.4.3.3 ¿Cuándo utilizar este patrón?

Siempre y cuando necesitemos utilizar el contenido de un algoritmo sin modificar la clase original que lo contiene.

3.1.4.3.4 ¿Cuándo evitar este patrón?

Cuando se necesite modificar el algoritmo original.

3.1.4.3.5 ¿Cómo saber que se está implementando de forma correcta?

Este patrón se implementa correctamente cuando utiliza un algoritmo sin necesidad de modificar la clase que define al propio algoritmo.

3.1.4.3.6 Código previo al patrón

Antes de aplicar este patrón, la repetición de código en distintas clases para realizar la misma función era de un tamaño considerable. Esto no solo dificulta la lectura del código, sino que aumenta el coste de creación de cada una de las clases que necesiten estas modificaciones.

Para ilustrar este paso previo al patrón mostraremos dos controladores distintos que tienen parte de su comportamiento en común.

```
class MainViewController: UIViewController {

    @IBOutlet var filterButton: UIBarButtonItem!
    @IBOutlet weak var kolodaView: CustomKolodaView!

    let sizeImagePulse: CGFloat = UIScreen.mainScreen().bounds.width/3
    let model: MainModel = MainModel()

    override func viewDidLoad() {
        super.viewDidLoad()
        //change style of statusBar
        UIApplication.sharedApplication().statusBarStyle = .LightContent

        //modify navigationBar appearance
        let navigationBarAppearance = UINavigationBar.appearance()
        navigationBarAppearance.tintColor = UIColor.whiteColor()
        navigationBarAppearance.barTintColor = Constants().NAVIGATOR_BAR_COLOR
        navigationBarAppearance.translucent = false

        //create a custom navigation item with Tendfy Logo
        let image = UIImage(named: "logo.png")
        let imageView = UIImageView(frame: CGRect(x: 0, y: 0, width: 170, height: 68))
        imageView.image = image
        imageView.contentMode = .ScaleAspectFit
        navigationItem.titleView = imageView

        //erase all shadows on NavigationBar
        var stop: Bool = false
        if self.navigationController != nil {
            for parent in self.navigationController!.navigationBar.subviews {
                for childView in parent.subviews {
                    if(childView is UIImageView && stop == false) {
                        childView.removeFromSuperview()
                        stop = true
                    }
                }
            }
        }

        //clear the text on backButton
        let backItem = UIBarButtonItem()
        backItem.title = ""
        navigationItem.backBarButtonItem = backItem
```

Imagen 25. Controlador de la vista principal

```swift
class DetailViewController: UIViewController, UITableViewDelegate, MKMapViewDelegate, CLLocationManagerDelegate {

    @IBOutlet weak var productImageView: UIImageView!
    @IBOutlet weak var locationLabel: UILabel!
    @IBOutlet weak var priceLabel: UILabel!
    @IBOutlet weak var discountLabel: UILabel!
    @IBOutlet weak var descriptionLabel: UILabel!
    @IBOutlet weak var mapView: MKMapView!
    @IBOutlet weak var tableView: UITableView!
    @IBOutlet weak var trademarkLogoImageView: UIImageView!

    @IBOutlet weak var closeButton: UIButton!
    @IBOutlet weak var shareButton: UIButton!

    var model: DetailModel = DetailModel()

    var object: Shoe {
        get {return self.model.shoe!}
        set(shoe) {self.model.shoe = shoe}
    }

    override func viewDidLoad() {
        super.viewDidLoad()

        //change style of statusBar
        UIApplication.sharedApplication().statusBarStyle = .LightContent

        //modify navigationBar appearance
        let navigationBarAppearance = UINavigationBar.appearance()
        navigationBarAppearance.tintColor = UIColor.whiteColor()
        navigationBarAppearance.barTintColor = Constants().NAVIGATOR_BAR_COLOR
        navigationBarAppearance.translucent = false

        //create a custom navigation item with Tendfy Logo
        let image = UIImage(named: "logo.png")
        let imageView = UIImageView(frame: CGRect(x: 0, y: 0, width: 170, height: 68))
        imageView.image = image
        imageView.contentMode = .ScaleAspectFit
        navigationItem.titleView = imageView

        //erase all shadows on NavigationBar
        var stop: Bool = false
        if self.navigationController != nil {
            for parent in self.navigationController!.navigationBar.subviews {
                for childView in parent.subviews {
                    if(childView is UIImageView && stop == false) {
                        childView.removeFromSuperview()
                        stop = true
                    }
                }
            }
        }

        //clear the text on backButton
        let backItem = UIBarButtonItem()
        backItem.title = ""
        navigationItem.backBarButtonItem = backItem
    }
```

Imagen 26. Controlador de la vista de detalle

Como vemos que el comportamiento inicial de ambos controladores es el mismo, esto nos genera una cantidad enorme de código duplicado en la aplicación. Pero también implica que si en un futuro se decide cambiar este comportamiento en estos controladores u otros nuevos, se tendrá que ir uno a uno modificando para que queden igual.

3.1.4.3.7 Demostración del patrón

Para ver la aplicación de este patrón primero se mostrará la estrategia de la cual extenderán los dos controladores que acabamos de ver.

```swift
import UIKit

class ViewWithREDNavTemplate: UIViewController, UINavigationBarDelegate {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)

        //modify navigationBar appearance
        let navigationBarAppearance = UINavigationBar.appearance()
        navigationBarAppearance.tintColor = UIColor.whiteColor()
        navigationBarAppearance.barTintColor = Constants().NAVIGATOR_BAR_COLOR
        navigationBarAppearance.translucent = false

        //create a custom navigation item with Tendfy Logo
        let image = UIImage(named: "logo.png")
        let imageView = UIImageView(frame: CGRect(x: 0, y: 0, width: 170, height: 68))
        imageView.image = image
        imageView.contentMode = .ScaleAspectFit
        navigationItem.titleView = imageView

        //erase all shadows on NavigationBar
        var stop: Bool = false
        if self.navigationController != nil {
            for parent in self.navigationController!.navigationBar.subviews {
                for childView in parent.subviews {
                    if(childView is UIImageView && stop == false) {
                        childView.removeFromSuperview()
                        stop = true
                    }
                }
            }
        }

        //clear the text on backButton
        let backItem = UIBarButtonItem()
        backItem.title = ""
        navigationItem.backBarButtonItem = backItem
    }
    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(true)

        //change style of statusBar
        UIApplication.sharedApplication().statusBarStyle = .LightContent

    }
}
```

Imagen 27. Patrón Plantilla

El contenido de esta nueva clase, contiene ese código que podíamos ver duplicado en los controladores mostrados anteriormente. Este controlador es del mismo tipo que las clases del ejemplo anterior UIViewController.

Ahora veremos el resultado tras aplicar el patrón a las clases antes mencionadas.

```
13  class MainViewController: ViewWithREDNavTemplate {
14
15      @IBOutlet var filterButton: UIBarButtonItem!
16      @IBOutlet weak var kolodaView: CustomKolodaView!
17
18      let sizeImagePulse: CGFloat = UIScreen.mainScreen().bounds.width/3
19      let model: MainModel = MainModel()
20
21      override func viewDidLoad() {
22          super.viewDidLoad()
23
24      }
```

Imagen 28. Patrón Plantilla controlador vista principal

```
13  class DetailViewController: ViewWithREDNavTemplate, UITableViewDelegate, MKMapViewDelegate, CLLocationManagerDelegate {
14
15      @IBOutlet weak var productImageView: UIImageView!
16      @IBOutlet weak var locationLabel: UILabel!
17      @IBOutlet weak var priceLabel: UILabel!
18      @IBOutlet weak var discountLabel: UILabel!
19      @IBOutlet weak var descriptionLabel: UILabel!
20      @IBOutlet weak var mapView: MKMapView!
21      @IBOutlet weak var tableView: UITableView!
22      @IBOutlet weak var trademarkLogoImageView: UIImageView!
23
24      @IBOutlet weak var closeButton: UIButton!
25      @IBOutlet weak var shareButton: UIButton!
26
27      var model: DetailModel = DetailModel()
28
29      var object: Shoe {
30          get {return self.model.shoe!}
31          set(shoe) {self.model.shoe = shoe}
32      }
33
34      override func viewDidLoad() {
35          super.viewDidLoad()
36
37      }
```

Imagen 29. Patrón Plantilla controlador vista detalle

Las diferencias que podemos encontrar a primera vista es la reducción drástica de código en los controladores. Como segundo cambio aplicado, podemos observar que se cambió el tipo de controlador de UIViewController al tipo ViewWithRedNavTemplate. Con este cambio permitimos a cada controlador reutilizar el mismo código, dejándolo más limpio y optimizado.

En caso de necesitar hacer cambios sobre el comportamiento de ambos controladores, sólo tendríamos que modificar la clase de la que entienden estas clases.

3.1.4.5 Patrón MVC

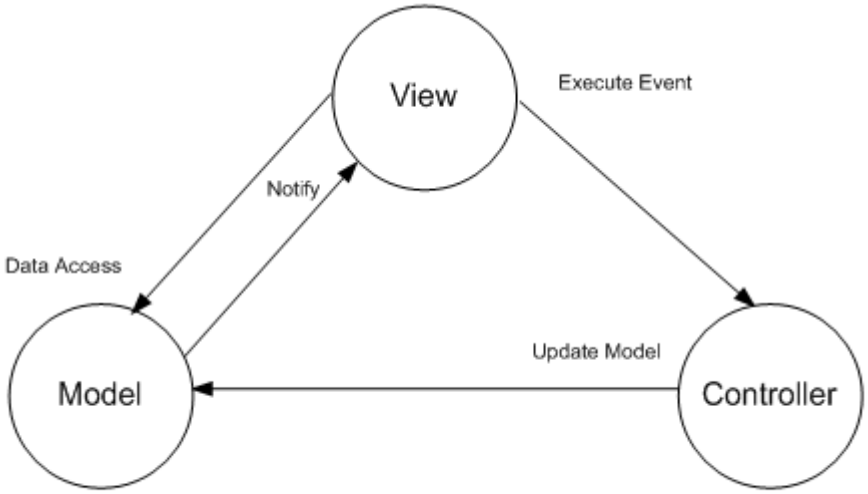


Imagen 30. Diagrama patrón MVC

3.1.4.5.1 ¿Qué es?
El patrón MVC añade estructura a toda una aplicación en lugar de a un componente individual.

3.1.4.5.2 ¿Qué beneficios tiene?
Cada parte de una aplicación puede ser desarrollada, testeada y mantenida con mayor facilidad.

3.1.4.5.3 ¿Cuándo utilizar este patrón?
Siempre que se trabaje sobre un proyecto complejo.

3.1.4.5.4 ¿Cuándo evitar este patrón?
Depende de la planificación y la infraestructura necesaria para implementar este patrón no se recomienda para proyectos de corta vida o simples.

3.1.4.5.5 ¿Cómo saber que se está implementando de forma correcta?
La implementación correcta de este patrón tiene que ver con las preferencias personales de cada desarrollador, no existe una explicación de cuando el patrón se implementa correctamente. En términos generales, las secciones individuales de la aplicación (modelo, vistas y controladores) deben estar muy poco acoplados con el fin de poder aislarlos fácilmente para realizar pruebas, también realizar cambios en una parte de la aplicación sin que sea necesario realizar los cambios en las otras secciones.

3.1.4.5.6 Código previo al patrón
Una de las principales señales que nos dirá que no se ha aplicado este patrón es la visualización de todo el código de un caso de uso en el mismo fichero. Esto también

depende de la complejidad del caso de uso, no siempre es interesante usar un modelo, cuando el controlador tiene poca lógica que manejar.

Lo más común al no utilizar este patrón es encontrar variables y funciones que almacenan o tratan datos de forma desacoplada del controlador o la vista, pero encontrándose en el mismo fichero.

```
class SettingsTableViewController: UIViewController, UITableViewDataSource, MFMailComposeViewControllerDelegate {

    var viewName = "Settings View"

    var settings = ["Compártelo con tus amigos", "Enviar una Sugerencia", "Perfil", "Sobre Nosotros"]//"Valorar en la App Store",
    let segueIdentifier = ["", "", "AccountView", "AboutUSView"]//"",
    let msg = "Descarga Tendfy para tu móvil. Descárgala ya desde aquí "

    @IBOutlet weak var tableView: UITableView!


    override func viewDidLoad() {
        super.viewDidLoad()

        let leftAddBarButtonItem: UIBarButtonItem = UIBarButtonItem(title: "Cerrar", style: UIBarButtonItemStyle.Plain, target: self, action:
            #selector(SettingsTableViewController.closeModal(_:)))
        self.navigationItem.setLeftBarButtonItem(leftAddBarButtonItem, animated: true)

        self.tableView.tableFooterView = UIView(frame: CGRect.zero)
        self.navigationController!.navigationBar.tintColor = UIColor(red: 190.0 / 255.0, green: 67.0 / 255.0, blue: 88.0 / 255.0, alpha: 1.0)

        self.navigationItem.title = "Ajustes"
        animateTable()
    }

    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(animated)
        UIApplication.sharedApplication().statusBarStyle = .Default
    }
```

Imagen 31. Diagrama patrón MVC

Al no utilizar el patrón también podremos encontrarnos con ficheros de gran volumen a nivel de líneas de código, esto es debido no siempre a la compleja lógica que almacena, sino al gran acoplamiento del código.

3.1.4.5.7 Demostración del patrón

El uso del patrón permite que la lógica de negocio del controlador funcione de forma indistinta a los datos que se almacenan en el modelo. En caso de que el controlador necesite un dato para mostrarlo en la vista o realizar alguna tarea con los datos, se lo solicitará al modelo y recogerá el resultado.

```
import Foundation

class SettingsModel {
    let settingsTextCells: [String] = ["Compártelo con tus amigos", "Enviar una Sugerencia", "Perfil", "Sobre Nosotros", "Valorar en la App Store"]
    let segueIdentifierForCells: [String] = ["", "", "EditProfileSegue", "AboutUsSegue", ""]

    func shareTendfy() {

        let objectsToShare = [msg, MyStringItemSource()]
        let vc = UIActivityViewController(activityItems: objectsToShare, applicationActivities: nil)


        //New Excluded Activities Code
        vc.excludedActivityTypes = [UIActivityTypePrint,
                                    UIActivityTypeCopyToPasteboard,
                                    UIActivityTypeAssignToContact,
                                    UIActivityTypeSaveToCameraRoll,
                                    UIActivityTypeAddToReadingList,
                                    UIActivityTypeAirDrop]
        //

        self.presentViewController(vc, animated: true, completion: nil)
    }

}
```

Imagen 32. Patrón MVC clase modelo de ajustes

```swift
import UIKit
import MessageUI

class SettingsTableViewController: ViewWithWHITENavTemplate, UINavigationBarDelegate, UITableViewDelegate, MFMailComposeViewControllerDelegate {
    let model: SettingsModel = SettingsModel()

    @IBOutlet weak var tableView: UITableView!
    override func viewDidLoad() {
        super.viewDidLoad()

        addCloseButton()
        self.tableView.tableFooterView = UIView()
        self.navigationItem.title = "Ajustes"
    }

    // MARK: - Table view data source
    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return model.settingsTextCells.count
    }

    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell = tableView.dequeueReusableCellWithIdentifier("SettingsCell", forIndexPath: indexPath) as! SettingsTableViewCell
        cell.textLabel?.text = model.settingsTextCells[indexPath.row]
        return cell
    }
```

Imagen 33. Patrón MVC clase controlador de ajustes

Como se puede observar en el controlador, obtiene el modelo, mediante el cual realiza la obtención y almacenamiento de datos, desacoplando así parte del código.

Este es el patrón de diseño más común en la aplicación, ya que la gran mayoría de las pantallas requerían de un modelo para almacenar información acerca de la vista.

La estructura de carpetas típica en este proyecto, sigue el MVC y es la siguiente:

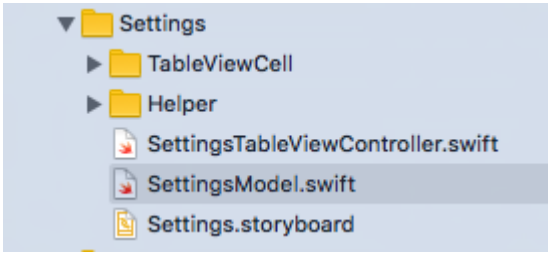


Imagen 34. Estructura de carpetas MVC

3.2 Tecnologías

En el proyecto hemos utilizado muchas y variadas tecnologías. Crear una aplicación para iOS no es tarea sencilla. A continuación las describiré.

3.2.1 iOS

3.2.1.1 Características de los SmartPhones

Hoy en día resultaría una tontería explicar lo que es un SmatPhones, ya que forma parte de nuestras vidas tanto como lo puede ser Internet. Un SmartPhone es un dispositivo de tamaño variable, en un principio la característica principal era la capacidad de miniaturización que tenían. Cuando aparecieron las primeras pantallas táctiles en dispositivos móviles, la tendencia del tamaño cambió en la dirección contraria. La necesidad de una pantalla mayor con la que poder ver e interactuar con el contenido multimedia del que disponían se hizo mayor. El acceso a internet mediante infraestructuras WIFI o 3G comenzaron a estandarizarse en estos dispositivos y con ello la posibilidad de poder crear contenido propio que utilizara ese acceso a la red de redes.

Entre sus características más destacables de los SmartPhones actuales:

- Dispositivo centrado en la pantalla, la cual es táctil y cuanto mayor sea el ratio tamaño - manejo, mejor.
- Especificaciones de hardware a nivel de ordenadores miniaturizados con una gran capacidad de computación.
- Capacidad de realizar fotografías de gran calidad y almacenarlas.
- Autonomía de 1 o 2 días, debido al gran consumo de batería por parte de las antenas de red y la pantalla.
- Servicios de conectividad inalámbrica.
- Interacción con el usuario adaptada.

Algunas de las tecnologías que posibilitan el desarrollo de este tipo de dispositivos son:

HSDPA, ARM, 4G, WIFI, Memoria Flash,OLED, Multitouch, Pantallas capacitivas, Bluetooth, Cloud computing, GPU, Web services, GPS/AGPS

El éxito de los dispositivos móviles es la capacidad flexible para ofrecer funciones y servicios a los usuarios.

3.2.1.2 Definición

iOS es un sistema operativo diseñado con el objetivo de abstraer el hardware y facilitar el desarrollo de aplicaciones para dispositivos de Apple, generalmente dispositivos móviles.

Hablamos de una plataforma cerrada, es decir que sólo se puede instalar en los dispositivos que Apple quiere, al contrario que Android el sistema operativo de Google, que permite la instalación en casi cualquier dispositivo.

Define un conjunto de software, incluyendo el sistema operativo, middleware, soporte de ejecución de aplicaciones.

Se apoya en el uso de un núcleo Darwin al igual que el resto de sistemas operativos de Apple, basado en Unix.

Proporciona un conjunto de APIs y herramientas de desarrollo, compilación y depuración.

Inicialmente, iOS se llamaba iPhoneOS, presentado en 2007 junto al primer SmartPhone más parecido a los que tenemos hoy en día. El sistema no tenía soporte a aplicaciones externas al sistema operativo. Disponía de una colección variada y útil de herramientas. Un año después tras el lanzamiendo de iPhoneOS 2.0, se lanzó la primera versión de XCode con soporte para iPhoneOS y con ello la posibilidad de desarrollar aplicaciones para el sistema operativo y poder venderlas en la AppStore.

Imagen 35. Evolución de iOS desde 2007-2013

Actualmente el sistema se encuentra en la versión 9.2, aunque ya se presentó el pasado mes de Junio de 2016 la siguiente versión 10 del sistema operativo.

3.2.1.3 Capas de iOS

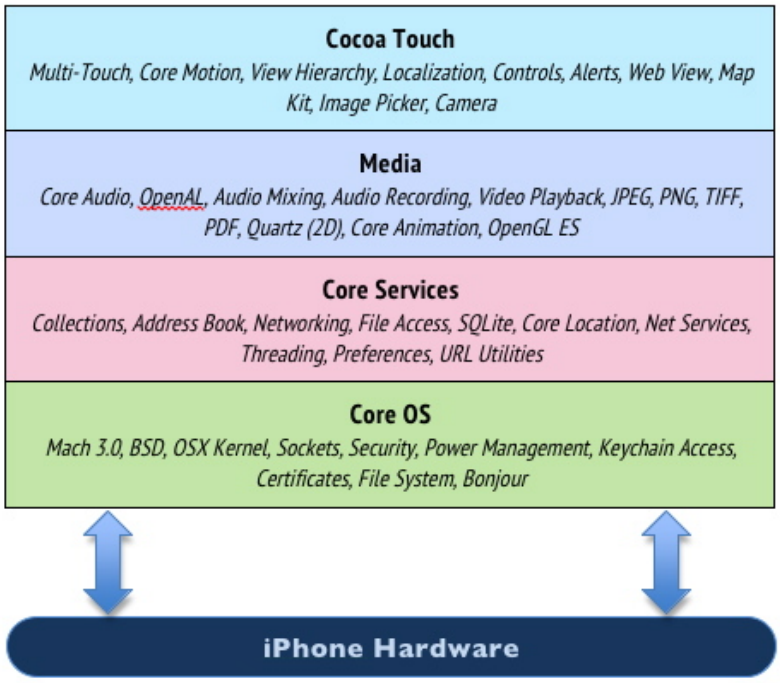


Imagen 36. Capas del sistema operativo

Como cualquier sistema operativo en la actualidad, el sistema se haya dividido en capas, cada una de ellas encargadas de una serie de tareas. Todas ellas son dependientes una de la otra, aunque estén definidamente separadas.

- **Core OS**: Esta es la capa más a bajo nivel del sistema. Contiene las características más básicas como: manejo de ficheros del sistema, manejo de memoria, seguridad, drivers del dispositivo…

- **Core Services**: Contiene servicios fundamentales del sistema que usan todas las aplicaciones para el almacenamiento de datos...
- **Media**: Provee los servicios de gráficos y multimedia, como puede ser el renderizado de imágenes, reproducción de música...
- **Cocoa Touch**: Esta es la capa más importante para el desarrollo de aplicaciones iOS. Contiene un conjunto de Frameworks que proporciona Cocoa para desarrollar aplicaciones. Esta capa está formada por dos Frameworks fundamentales:
    - **UIKit**: contiene las clases que necesita el desarrollador para el diseño de la interfaz de usuario.
    - **Foundation Framework**: define las clases básicas, acceso y manejo de objetos, servicios del sistema operativo.

3.2.1.4 Kernel iOS

iOS está basado en el kernel Unix, este adopta el nombre de Darwin. Funciona con el mismo kernel que el sistema operativo de los dispositivos de mesa con macOS dotando de gran fiabilidad. iOS aprovecha el kernel para la mejora su seguridad, la gestión de memoria con ARC, introducido junto a la presentación de la versión 5 del sistema, hasta entonces la gestión de la memoria era trabajo del programador.

iOS utiliza el modelo de controladores, proporcionando soporte a nuevos accesorios a través de estos controladores. Separando el hardware del software mediante una capa de abstracción.

3.2.1.5 Frameworks de iOS

Hay un conjunto de frameworks básicos desde las cuales se pueden implementar nuevos frameworks o simplemente utilizar los existentes para el desarrollo de aplicaciones. Pasamos a explicar algunos de ellos:

- **UIKit**: Este framework se encarga de la generación de objetos de interfaz de la aplicación, desde vistas, hasta simples etiquetas o botones.
- **Foundation**: Mediante Foundation podremos realizar el manejo de objetos o servicios del sistema operativo. El uso de este framework es completamente necesario para el desarrollo de cualquier tipo de aplicación, ya tenga esta una parte de interfaz o no.
- **Core Data**: Con Core Data podremos manejar la persistencia de la aplicación mediante una base de datos del tipo *SQLite*.
- **Core Location**: Gracias a este framework disponemos de acceso a los servicios de geolocalización del dispositivo, pudiendo manejarlos para la necesidades que tengamos en el desarrollo de una aplicación.

3.2.1.6 Entorno de las aplicaciones

Este entorno proporciona una plataforma de desarrollo premitiendo la reutilización de componentes, cada aplicación muestra las capacidades que tiene y se utilizan unas a otras en base a ellas mismas.

Con el acceso a estos frameworks, el desarrollador puede tener acceso al hardware de los dispositivos, para poder gestionar el ciclo de vida de la aplicación.

Imagen 37. Elementos de gráficos de UIKit

3.2.1.7 Nivel de aplicación

Mediante esta capa se ubican las aplicaciones preinstaladas por Apple y las que instalará el usuarios. Las aplicaciones se desarrollan en Objective-C / C / C++ /Swift utilizando el SDK (Software Development Kit) de Apple, llamado XCode.

En cuanto a aplicaciones por defecto tenemos *App Store*, que ofrece al resto de *Apps* desarrolladas por terceros.

Imagen 38. Logo de Swift

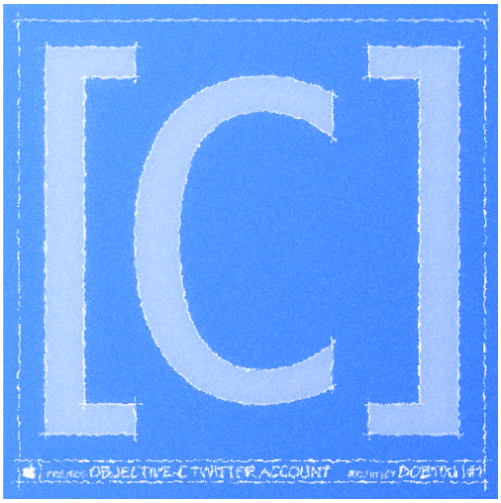


Imagen 39. Representación de Objective-C

3.2.1.8 Ciclo de vida

A la hora de desarrollar aplicaciones para cualquier sistema operativo, es necesario el ciclo de vida de las aplicaciones en el sistema para el que vayamos a desarrollarlas. El ciclo de vida se podría dividir en dos grupos principales:

- **Estado Activo**: Este estado es en el cual la aplicación aparece en pantalla de forma completa o parcialmente completa. Durante todo este proceso la

aplicación puede acceder o realizar una serie de tareas dentro de las limitaciones del sistema.

- **Estado Background**: Siempre y cuando una aplicación pierda el foco, ya sea debido a que se cambia de aplicación, se cierra o se suspende debido al bloqueo del dispositivo, este pasa a modo background, el cual tiene unas limitaciones del sistema mayores y más restrictivas.

Debido a que el sistema es tan cerrado, las aplicaciones en estado background tienen dos opciones; finalizar el proceso que están ejecutando o piden un tiempo de prórroga para acabar lo que estaban haciendo y detenerse.

Es muy común en aplicaciones que descargan contenidos dentro de la aplicación, si se suspende la aplicación mientras se descarga una canción, la *App* debe pedir una prórroga al sistema para terminar de descargar la canción y luego detenerse. En caso de no ser suficiente la prórroga, el sistema la detendrá automáticamente.

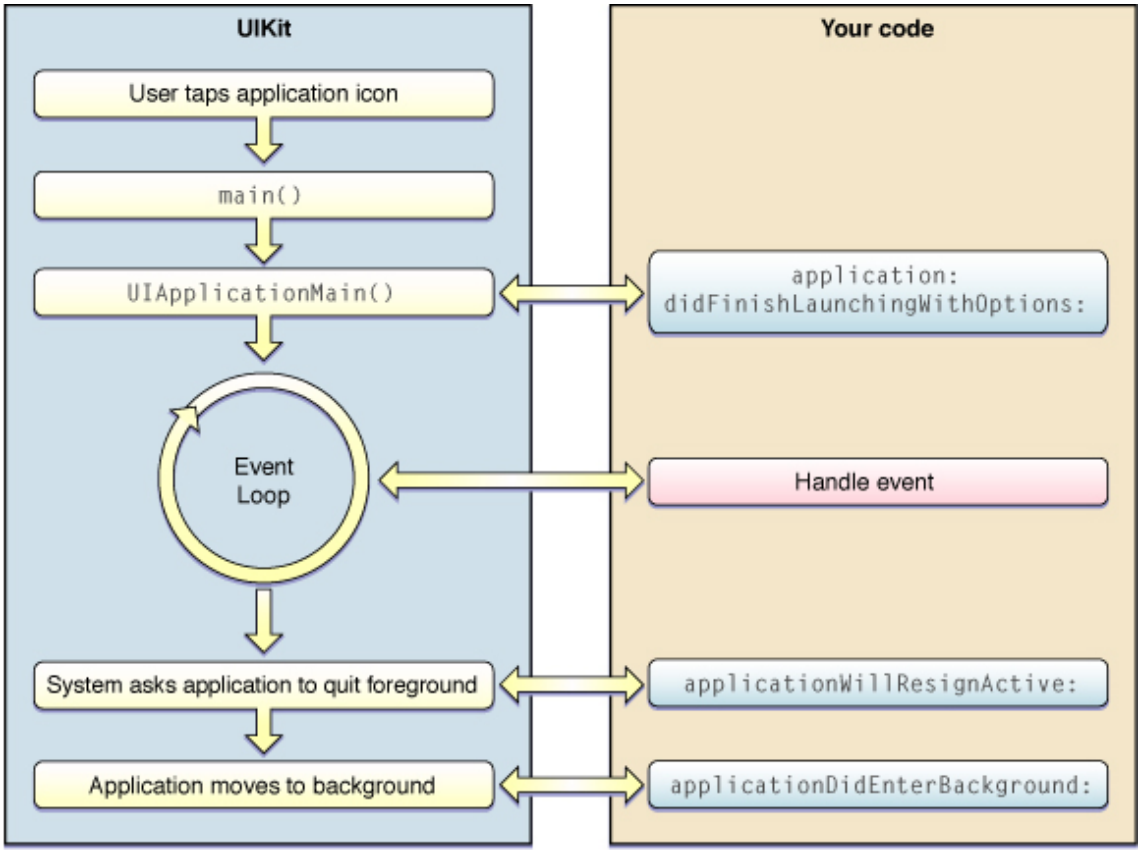


Imagen 40. Ciclo de vida de una aplicación

3.2.1.9 Procesos y su gestión

Es común pensar en la multitarea como un conjunto de procesos ejecutándose al mismo tiempo y al devolver el foco a uno de estos continuar con lo que estaba haciendo.

En iOS la multitarea es existente y a la vez no lo es. El proceso de gestión de procesos del sistema es distinta al que se podría encontrar en Android, en esa plataforma se podrían tener varias aplicaciones funcionando realmente, refrescando contenidos y realizando sus tareas sin un sistema que intenta que dejen de hacer su trabajo, lo que

conocemos como una multitarea real. Sin embargo en iOS funciona de una manera distinta.

Cuando un proceso pasa a background, como se ha explicado anteriormente, este tiene un tiempo determinado para acabar su trabajo y almacenarse en memoria, si transcurre ese tiempo y no se almacena, se desecha. El proceso al almacenarse en memoria se detiene completamente, no puede ser capaz ni de modificar la interfaz.

Cuando el proceso se requiere que vuelva a la actividad, este se recoge de memoria y se continúa por el mismo instante por donde se quedó suspendido. La ventaja que se obtiene es una mayor ahorro de recursos del sistema, ya no solo en procesador si no en el consumo de la batería.

3.2.2 XCode

XCode es el *IDE* oficial para el desarrollo de aplicaciones en iOS/macOS/tvOS/watchOS. Este es el único entorno de desarrollo de aplicaciones en los lenguajes nativos del sistema. Aunque se puede desarrollar en otras plataformas como Xamarin, escogimos el IDE por excelencia de Apple. Xcode es un software propio de Apple, inicialmente orientado al desarrollo de aplicaciones para macOS, pero actualmente es el nexo del desarrollo para los distintos operativos de Apple.

Imagen 41. XCode

3.2.2.1 Estructura de Datos

Por defecto, Xcode muestra los archivos del proyecto en la la barra lateral izquierda. Esta vista nos proporciona un acceso rápido a los archivos fuente y nos muestra la estructura de nuestro proyecto.

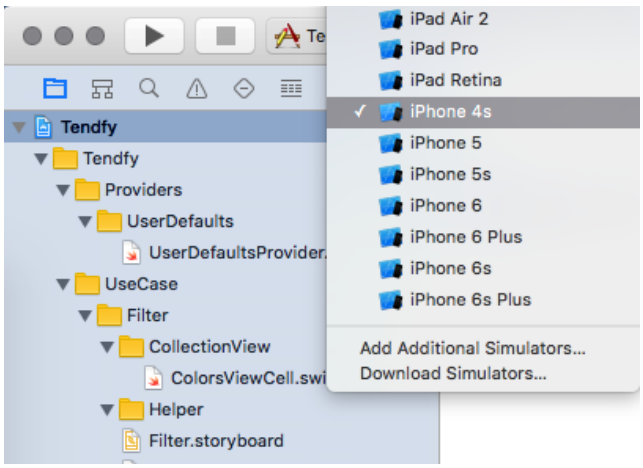


Imagen 42. Vista del proyecto

Esta vista muestra con un icono azul el fichero principal del proyecto, desde el que nace el resto del contenido de la aplicación, el cual se puede estructurar según lo desee el desarrollador.

Se pueden encontrar varios tipos de ficheros dentro de esta vista, nombraremos ahora a los principales:

- **.swift**: son ficheros de código fuente que utiliza el lenguaje *Swift*. En estos ficheros es donde se implementa el código que ejecutará la aplicación según se interactúe con el.
- **.storyboard:** este tipo de fichero es con el que se puede realizar la interfaz de la aplicación mediante una serie de herramientas visuales que permiten generar las vista de la *App*, de manera muy sencilla y cómoda sin tener que recurrir a código.
- **.xcassets:** se le podría considerar un directorio especial en lugar de un tipo de fichero. Con los *xcassets* almacenamos como su propio nombre indica los *Assets* de la aplicación, es decir todos los gráficos de tipo imagen que serán mostrados en la aplicación.
- **.plist:** en cuanto a estos ficheros suelen almacenar opciones de configuración de la aplicación, como puede ser la versión del sistema que tiene como *target* la *App*.

3.2.2.2 iOS Builder

Como podemos apreciar en la parte derecha de la imagen 22, tenemos una pequeña herramienta que nos permite la compilación, ejecución y test de la aplicación. El sistema también nos proporciona la opción de elegir sobre que dispositivo queremos ejecutar la aplicación, esto se explicará más extensamente a continuación en el punto 6.2.4.

3.2.2.3 Depuración y rendimiento

XCode nos proporciona un gran número de mejoras para la depuración y ayudar a mejorar el rendimiento de nuestro código, mediante un asistente para la emulación de dispositivos móviles, depuración línea a línea y herramientas de análisis del rendimiento. XCode también permite la activación de *BreakPoints* en el código en tiempo de ejecución, es decir, podemos añadir puntos críticos en nuestro código mientras se está ejecutando la aplicación en el simulador, permitiéndonos tener mayor control sobre nuestra aplicación mientras se está desarrollando.

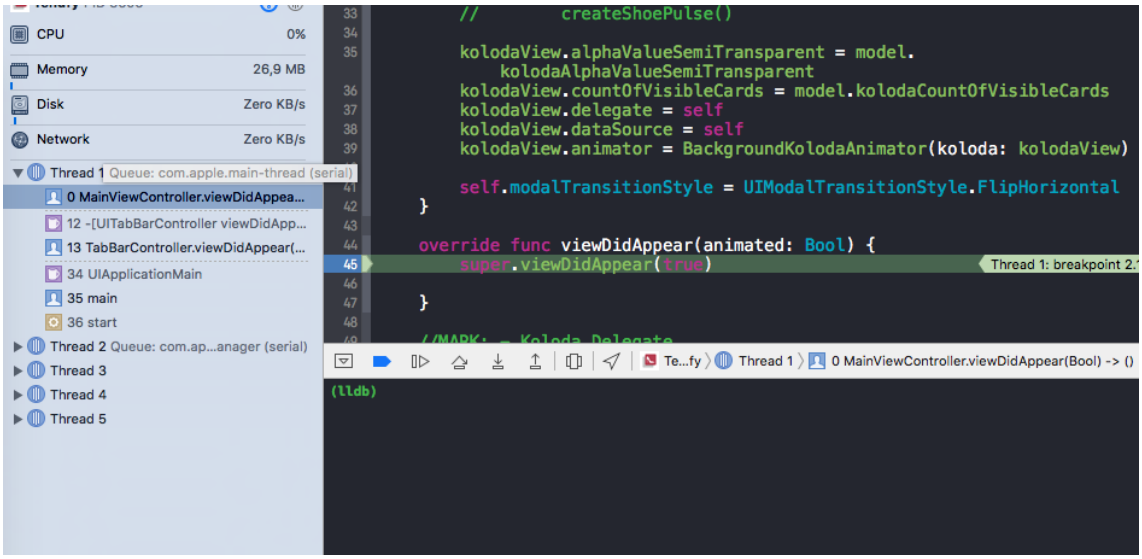


Imagen 43. BreakPoint de XCode

3.2.2.4 iOS Simulator

El simulador pone a nuestro servicio una máquina virtual que simula un dispositivo. Este simulador permite ejecutar cualquier dispositivo con *iOS* para el *target* al que va dirigido nuestra aplicación, incluyendo las tablets. El simulador te permite ejecutar cualquier función del sistema a excepción de notificaciones push y la cámara de fotos. En caso de necesitar estas dos funciones es necesario un dispositivo real.

También es posible la simulación de una geolocalización, incluso movimiento en vivo por un mapamundi, además existe la posibilidad de simular el movimiento del dispositivo, el uso de la huella dactilar o la tecnología *3D Touch*.

3.2.2.5 Monitorizar el rendimiento de nuestra App

Xcode también provee de un apartado de monitorización de rendimiento de la aplicación, este se ejecuta siempre que lanzamos la aplicación desde el *IDE*. Mientras la aplicación se ejecuta en el simulador o en el dispositivo real con el que se testee la aplicación, podremos ver las siguientes características mediante una serie de gráficos:

- **CPU**: Indica el consumo de procesador que tiene la aplicación durante la ejecución. Esto puede darnos pistas de qué partes de la aplicación tienen más

carga y poder asegurarlas ante posibles problemas de calentamiento del dispositivo cuando se publique la aplicación.

- **Memory**: Como su nombre indica nos muestra el consumo de memoria RAM del dispositivo, esto puede ser un factor muy a tener en cuenta cuando desarrollamos una *App*, ya que un consumo tanto de *CPU* como de memoria indica un mayor consumo de batería y posibles salidas inesperadas de la aplicación debido a que el sistema cierra una aplicación cuando considera que consume más memoria de la que debería.
- **Disk**: Podemos ver el uso de escrituras que hacemos el disco del dispositivo. Esto suele ser muy útil cuando utilizamos CoreData y el acceso a los datos de la base de datos es continuo.
- **Network**: Otro factor a tener muy en cuenta cuando desarrollamos una aplicación es el uso que hacemos de la red, ya que un uso excesivo puede acabar con la tarifa de datos del usuario. Por este motivo podemos optimizar los contenidos descargados con la aplicación que estamos desarrollando.

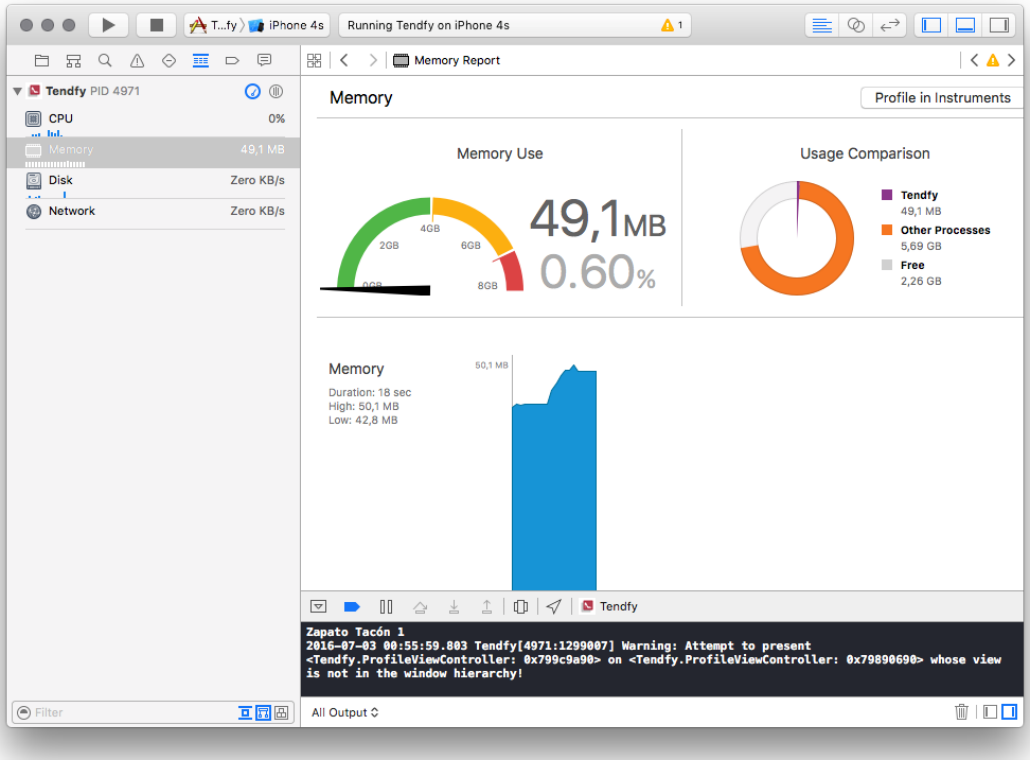


Imagen 44. Monitor de rendimiento

3.2.3 Bitbucket

Bitbucket es una plataforma de alojamiento para trabajo colaborativo de software, utilizando el sistema de control de versiones, permite que varios desarrolladores trabajen sobre el mismo proyecto. El código se almacena de forma privada o pública,

dependiendo del fin que le quiera dar el equipo de desarrollo. El uso de esta plataforma es gratuito para la mayoría de los proyectos.

Imagen 45. Bitbucket

Bitbucket aloja tu código y te da herramientas como sourceTree para agilizar el uso del repositorio en equipo.

También puedes contribuir a mejorar los proyectos de otros usuarios que han publicado de manera pública. Bitbucket ofrece funcionalidades para hacer *fork* y solicitar *pulls*.

Para realizar un *fork* tendríamos que clonar el repositorio con el que queremos colaborar (genera una copia en tu cuenta), podemos solucionar algún bug añadiendo funcionalidades al código. Una vez realizados los cambios puedes lanzar un *pull* al dueño del proyecto original. Éste valorará los cambios que se han realizado, y puede considerar tu contribución, adjuntándolo al repositorio original.

Este es un manejador de versiones muy potente y útil para equipos de trabajo. Tanto en entornos profesionales, como para equipos de desarrollo independiente. Xcode también nos da soporte a la sincronización de nuestros proyectos con Bitbucket o Github. Podemos descargarnos nuestro repositorio desde el propio *IDE* sin necesidad de instalar otra aplicación externa, esto mantendrá actualizado nuestro proyecto local con el código del repositorio en la nube y en caso de existir conflictos (dos usuarios han cambiado los mismo ficheros), podremos hacer un *merge* y fusionar ambos documentos. Realizadas las modificaciones, subiremos al repositorio a la nube con nuestros cambios.

También tenemos la posibilidad de crear *branches* (ramas), con las que podremos alojar diferentes versiones de nuestro código sobre el mismo repositorio, facilitando así tanto el desarrollo como el mantenimiento y el testing de cada módulo por separado.

Podremos subir a Bitbucket tantos proyectos como necesitemos, tanto públicos como privados.

3.2.4 Parse

Parse es un servidor de almacenamiento escalable y orientado al desarrollo de aplicaciones móviles.

Imagen 46. Parse

Mediante un *SDK* dirigido al sistema al que se va a desarrollar, permite el uso y explotación de una base de datos en la nube al que puede consultar nuestra aplicación para descargar información útil para el funcionamiento de esta.

El servicio en un inicio es gratuito hasta superar un tamaño máximo de nuestra base de datos almacenada o un número máximo de consultas y accesos al servidor. A partir de entonces hay que adjuntarse a un plan de pago.

3.2.4.1 Instalación de Parse SDK

Para poder utilizar Parse como servidor de nuestra aplicación, el primer paso es descargar el *SDK* desde su página web e instalarlo en nuestra aplicación. La configuración de la aplicación tras la instalación del *SDK* es mínima, simplemente añadiendo unas pocas líneas y añadiendo las acreditaciones para el acceso a nuestros datos, ya tendremos todo configurado para usar Parse.

3.2.4.2 ¿Cómo funciona Parse?

Parse utiliza un sistema de consultas como se podría utilizar en un sistema de base de datos *SQL*. Mediante estas consultas el servidor nos devuelve la información que corresponde a la consulta solicitada.

Aquí tenemos un ejemplo de consulta con *Swift* sobre la puntuación de un usuario:

```
// Using PFQuery
let query = PFQuery(className: "GameScore")
query.whereKey("playerName", equalTo: "Dan Stemkoski")
let scoreArray = query.findObjects()
```

Código 1. Query en Parse

Explicando la sintaxis mostrada en el código que aparece arriba, tendremos lo siguiente:

- Realizamos una consulta sobre la tabla *GameScore*.
- Pedimos que sobre esa tabla nos de todos los datos del usuario con el nombre “Dan Stemkoski”.

Como se observa, la sintaxis es muy sencilla y como se comentó al inicio de este apartado es similar a una consulta en *SQL*.

No solo podemos recibir y almacenar datos, Parse además puede recibir extensiones mediante *Scripts* que nos permiten realizar cálculos o incluso añadirle inteligencia

artificial a la propia base de datos. Esto último se puede utilizar para realizar sugerencias de datos según el uso que le demos a la aplicación.

La plataforma fue comprada por Facebook tras el rápido crecimiento que tuvo en 2014. Tras un año después de la compra, la plataforma anunció su cierre para Enero de 2017. A la vez que publicó su cierre, publicó su código fuente y una serie de herramientas en Github para que todos aquellos usuarios de Parse no quedaran huérfanos y pudieran migrar sus servidores a otros distintos utilizando la misma tecnología y evitando así que las aplicaciones que ya se apoyaban sobre este servicio no dejaran de dar servicio.

3.2.4.3 Migración de Parse

La migración del servidor es relativamente sencilla, mediante el código disponible en el repositorio público que se puede encontrar en Github, simplemente se tendrá que instalar en otro servidor como Amazon AWS o Heroku, ambos servidores tienen un ayudante de instalación, haciéndolos atractivos para aquellos usuarios que necesitan migrar su servidor sin tener grandes conocimientos sobre el *deploy* de este tipo servidores.

Imagen 47. Amazon AWS

Imagen 48. Heroku

3.2.5 Import.io

Import.io es un una herramienta web de extracción de información de otros sitios web, lo que se conoce como un *Scrapper*.

Imagen 49. Import.io

Esta herramienta permite extraer grandes cantidades de información de un sitio web, mediante la automatización del proceso de *scrapeo*. La información sustraída la almacena en una base de datos que se ha configurado previamente.

Una vez obtenidos los datos necesarios es posible exportarlos mediante su API para el usuario que necesitemos. En el caso de Tendfy, se extraía información sobre el calzado de distintos sitios webs y tras un tratamiento del que hablaremos posteriormente, se almacenaba en el servidor Parse antes nombrado.

3.2.6 Back-End PHP

Tras el uso de Import.io, nació la necesidad de exportar y tratar esos datos que a veces eran almacenados de forma desordenada o incompleta, debido a la automatización de la herramienta antes nombrada.

Para el tratamiento de estos datos, se implementó un *Back-End* desarrollado en PHP, el cual utilizando la API de Import accedía a la información de cada uno de los productos que había capturado. Una vez recibida esta información, procedía al ordenamiento y autocompletado de datos, para su posterior almacenamiento en el servidor final.

Un ejemplo de este tratamiento de datos sería el siguiente:

Nombre Producto	Categoría Producto	Colores Producto	Talla Producto	Material Producto	Descripción Producto
Zapato Tacón	Tacón	-	36, 36.5, 37, 38	-	Zapato de tacón de color rojo hecho con piel natural

Datos recibidos de Import.io

El resultado tras el tratamiento de la información sería de la siguiente manera:

Nombre Producto	Categoría Producto	Colores Producto	Talla Producto	Material Producto	Descripción Producto
Zapato Tacón	Tacones	Rojo	36,37,38	Pieles	Zapato de tacón de color rojo hecho con piel natural

Datos recibidos de Import.io

Esto es posible mediante la dotación de inteligencia al software.

El proceso de tratamiento de los datos comienza por la lectura de los campos incompletos o con un formato no válido.

Mediante la lectura de los campos que están completos, se reconocen palabras clave como los colores o materiales tanto si están escritos en plural, singular, con mayúsculas

o minúsculas. Generando unas cadenas de texto estándar que utilizarán todos los productos que sean almacenados en la base de datos final. Esto aporta una homogeneidad de datos.

Un ejemplo del código de tratamiento de los datos es el siguiente:

```php
<?php
//Material exists?
if(!$shoesArray["material"])
    $shoesArray["material"] = -1;

//Fill the field with Materials Strings
$shoesArray["material"] = giveMeMaterial($shoesArray["material"], $shoesArray["name"], $shoesArray["description"]);

//Add this Materials to the Object
if($shoesArray["material"])
    $object->setArray("material", $shoesArray["material"]);

//Obtain Materials Strings with the Strings Template
function giveMeMaterial($material, $title, $description){
    $array = array();
    $materials = array();
    $array = readMeFile("material.csv");

    $materials = array_merge(findMeSimilarities($title, $material, $array), findMeSimilarities($description, $material, $array));
    $material = array_map("unserialize", array_unique(array_map("serialize", $materials)));
    $orderedMaterial = array_values($material);
    return $orderedMaterial;
}

//Find matches of fields data with CSV Strings
function findMeSimilarities($data1, $data2, $source){

    $arrayData1 = array();
    $arrayData2 = array();
    $datas = array();

    $arrayData1 = preg_split('/[,.\s;:-]+/', $data1);
    if(is_array($data2)){
        foreach($data2 as $data){
            $arrayData2 = array_merge($arrayData2, preg_split('/\b\s*[!?.,;:-]*/', $data));
        }
    }else
        $arrayData2 = preg_split('/\b\s*[!?.,;:-]*/', $data2);

    foreach($source as $dataKey => $data){
        foreach($arrayData1 as $key=>$value){
            $value = cleanMeAccent($value);
            if(in_array(strtolower($value), array_map('strtolower', $data))){
                array_push($datas, $data[0]);
            }
        }
    }
    foreach($source as $dataKey => $data){
        foreach($arrayData2 as $key=>$value){
            $value = cleanMeAccent($value);
            if(in_array(strtolower($value), array_map('strtolower', $data))){
                if($data[0] == 'MULTICOLOR'){
                    //..do something
                }
                array_push($datas, $data[0]);
            }
        }
    }
    return $datas;
}
?>
```

Imagen 50. Parte del código de tratamiento de productos

El código trata de obtener los materiales de un calzado, mediante la lectura de la descripción y el nombre de este.

Como se puede observar el código es en PHP y realiza el siguiente proceso:

1. Comprueba que tiene el campo categoría capturado para poder rellenarlo. (**líneas 3-4**)
2. Pide recibir los materiales de zapatos mandando la descripción y el nombre del zapato. (**línea 7**)
3. Lee un fichero con una plantilla de materiales escritos con palabras claves y pide que busque similitudes entre los campos recibidos y las palabras del fichero leído. (**líneas 15-17**)
4. Trata el texto para que esté escrito de la misma manera las palabras obtenidas en el paso anterior (minúsculas y sin acentos).(**líneas 32 y 38**)
5. Una vez encontradas las coincidencias, comprueba que no existan repeticiones de materiales y lo devuelve para que sea enviada la información al servidor. (**líneas 40-59 y 10-11**)

4. Diseño de Tendfy utilizando patrones

En esta parte de la memoria se explicará en qué consiste cada una de las partes de la aplicación, qué función tiene y de qué manera se han aplicado los patrones de diseño tras la refactorización del código.

Al comenzar la refactorización del proyecto, lo primero a tener en cuenta era la estructura base de módulos y carpetas que debería tener. Esta está dividida en Casos de Uso (CU). Cada uno de estos *CU*, representa una vista en la aplicación, la cual a su vez está dividida en tres partes debido a las necesidades del proyecto.

Estas partes son las siguientes:

- **Vista o Presentación**: Representa todo aquello que se refiere a la vista que puede ver el usuario cuando utiliza la aplicación. El diseño utilizado, la disposición de botones u objetos con interacción...
- **Lógica de Negocio**: Trata las interacciones del usuario con la vista, definiendo el comportamiento que debe tener la aplicación cuando se está utilizando.
- **Conexión con el Back-End**: Es básicamente el proveedor que se encarga de conectar con la *API* del servidor de datos. Envía y recibe datos, según la lógica de negocio le indica.

Antes de entrar en detalle con la estructura de la aplicación, veremos qué ventanas y apartados tiene la aplicación, es decir el Front-End de la *App*.

Comenzaremos viendo la navegación entre las diferentes vistas de la aplicación, describiendo la funcionalidad de cada una de ellas, así como de qué manera nos moveremos de una a otra. Explicaremos también cómo se comportan estas vistas ante un cambio en el ciclo de vida de la app.

Continuaremos con un análisis de los objetos que se utilizan en cada una de las vistas, y comentaremos también por qué se decidió usar ese diseño. También tendremos en cuenta la usabilidad y explicaremos las librerías utilizadas. También veremos qué ocurre al interactuar con estos componentes, por ejemplo el efecto *drag & drop* de las cartas de los zapatos.

4.1 Vistas de la aplicación

Parte del código ha sido eliminado y substituido por comentarios por motivos de confidencialidad con la empresa.

El número total de vistas de la aplicación asciende a 12, con la siguiente estructura:

- Pantalla de carga
    - Login
        - Pantalla de Aviso Legal
    - Pantalla principal (Cartas)
        - Pantalla de Filtro

- Pantalla de selección de Marcas
- Pantalla de perfil de usuario
  - Pantalla de Ajustes
    - Pantalla de edición de perfil
    - Pantalla AboutUs
- Pantalla de favoritos
  - Pantalla de detalle de producto

4.1.1 Pantalla de Carga

La pantalla de cargando está compuesta por dos elementos:

1. UImageView
2. UImageView

UImageView es el componente que utiliza iOS para representar imágenes.

Imagen 51. Pantalla de carga

Como se puede observar en la imagen 29, aparece el logo de la empresa y el nombre de esta en medio de la pantalla.

El motivo por el que se creó una pantalla de carga además de la que trae el sistema por defecto, era la comodidad de aplicarle lógica a esta vista y redirigir al usuario depende de su estado, si estaba registrado o no.

4.1.1.1 Estructura y patrones

La estructura interna de carpetas de esta vista al igual que el *Login* y la pantalla de aviso legal, no requiere del patrón *MVC* (Modelo Vista Controlador). Ya que tiene una poca lógica de negocio muy pobre.

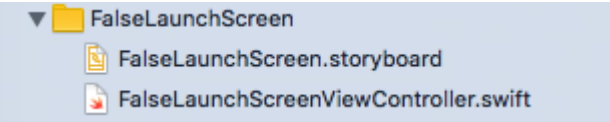


Imagen 52. Estructura de carpetas de la vista de carga

```
import UIKit

class FalseLaunchScreenViewController: UIViewController {

    override func viewDidLoad() {super.viewDidLoad()}
    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(true)

        if(UserDefaultsProvider.existsUserData()) {
            self.performSegueWithIdentifier("mainSegue", sender: nil)
        } else {
            self.performSegueWithIdentifier("loginSegue", sender: nil)
        }

    }
}
```

Imagen 53. Lógica de negocio de la vista de carga

Esta vista utiliza un proveedor que comprueba si el usuario ha sido *logeado* con anterioridad en la aplicación y en caso de ser así o no, lanza una vista u otra. Este proveedor será explicado en la siguiente sección.

4.1.2 Pantalla de Login

La pantalla de Login es una puerta de acceso a las funcionalidades de la aplicación.

Para iniciar sesión se eligió la red social Facebook como punto entrada, de esta manera era mucho más sencillo el registro de nuevos usuarios, y el acceso de los ya existentes. Si no se dispone de una cuenta de Facebook, se puede acceder de forma anónima, pero la gran mayoría de las funcionalidades están bloqueadas, invitando al usuario a registrarse para poder utilizar la aplicación en su totalidad.

Imagen 54. Login

Imagen 55. Login Facebook

La pantalla está formada por un UImageView con el nombre de la *App*, otro UImageView para el fondo de la vista, un UIButton con un aspa blanca en la zona superior derecha para el acceso anónimo a la aplicación, otro UIButton en la parte inferior de color azul pare acceder a la aplicación mediante Facebook. Finalmente otro UIButton que nos llevará al aviso legal.

Esta pantalla tan solo será mostrada en dos ocasiones. La primera será cuando el usuario abra la aplicación por primera vez tras su descarga. La segunda ocasión sería cuando el usuario cierre sesión, acción que veremos más adelante.

4.1.2.1 Estructura y patrones

Como ya se indicó en la pantalla de carga, esta vista también tiene una poca lógica de negocio muy pobre.

Esta clase Login, simplemente lanza el login con Facebook del sistema mediante su SDK y el proveedor almacena estos datos dentro de lo que se conoce como NSUserDefaults, que permite la persistencia de datos muy básicos.

```swift
import UIKit

class LoginViewController: UIViewController {

    @IBOutlet weak var loginFacebookButton: UIButton!
    @IBOutlet weak var legalAdvertiseButton: UIButton!
    @IBOutlet weak var dimissLoginButton: UIButton!

    override func viewDidLoad() {
        super.viewDidLoad()
        UIApplication.sharedApplication().statusBarStyle = .LightContent
    }

    @IBAction func loginFacebook(sender: AnyObject) {
        FacebookSDK.Login()
        self.dismissViewControllerAnimated(true, completion: nil)
    }

    @IBAction func showLegalAd(sender: AnyObject) {
        self.performSegueWithIdentifier("legalSegue", sender: nil)
    }
}
```

Imagen 56. Lógica del Login

4.1.3 Pantalla de Aviso Legal

La pantalla de aviso legal es un navegador embebido que apunta al aviso legal presente en la web de la empresa.

Para evitar crear una vista más compleja se suele utilizar este pequeño truco, que reduce mucho el coste en tiempo de desarrollar una pantalla nueva que muestre información que puede variar, por este motivo si no se hiciera de esta manera, habría que actualizar la aplicación cada vez que se decidiera realizar cambios en el aviso legal.

La pantalla está formada por un UIWebView que carga la página de aviso legal nada mas aparecer por pantalla, esta vista también dispone de una UINavigationBar , esta es la barra de navegación entre vistas por defenco por el sistema. En esta barra de navegación se se ha colocado un UIButton con un aspa para poder cerrar la vista cuando el usuario lo desee.

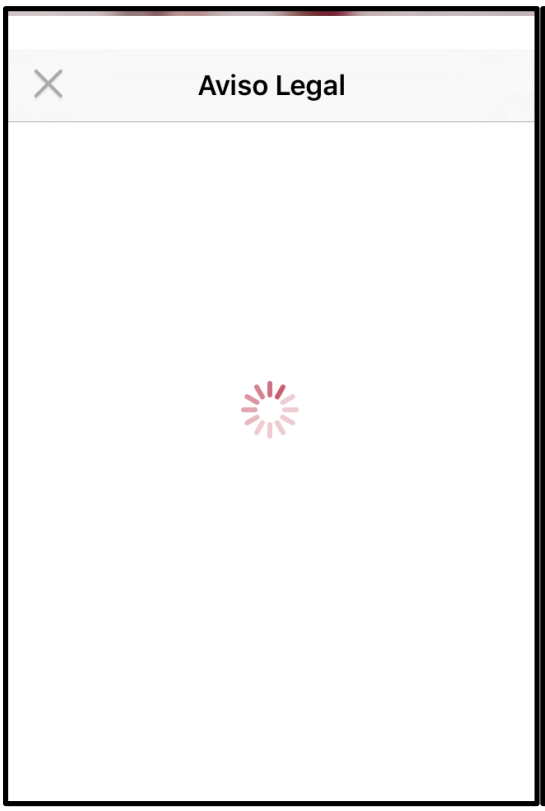


Imagen 57. Carga del aviso Legal

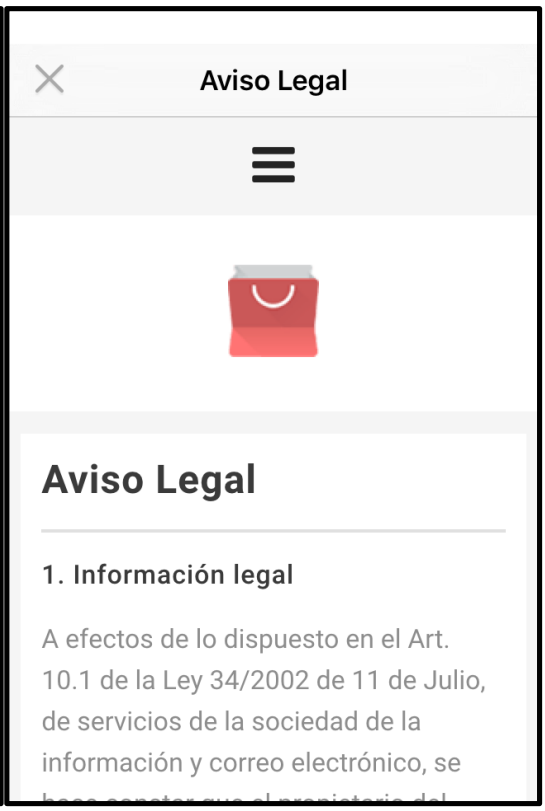


Imagen 58. Aviso legal

Mientras se realiza la carga de la página web, se muestra un UIActivityIndicatorView, que básicamente es el icono de carga por defecto del sistema. Indica al usuario que la aplicación sigue en funcionamiento en el caso de que se demore mucho la carga de la página.

4.1.3.1 Estructura y patrones

Esta clase de aviso legal, quizás sea la vista con menos trabajo a nivel lógico, ya que simplemente tiene que cargar una web externa a la *App* al iniciarse.

```
import UIKit

class LegalAdvertiseViewController: UIViewController, UIWebViewDelegate {
    @IBOutlet weak var webView: UIWebView!
    @IBOutlet weak var closeModalViewButton: UIBarButtonItem!
    @IBOutlet weak var activityIndicator: UIActivityIndicatorView!

    override func viewDidLoad() {
        super.viewDidLoad()
        webView.loadRequest((NSURLRequest(URL: NSURL(string: "http://tendfy.com/aviso-legal/")!)))
        activityIndicator.startAnimating()
        webView.delegate = self
    }

    @IBAction func closeView(sender: AnyObject) {
        self.dismissViewControllerAnimated(true, completion: nil)
    }
    func webViewDidFinishLoad(webView: UIWebView) {
        self.activityIndicator.stopAnimating()
        self.activityIndicator.hidden = true
    }
}
```

Imagen 59. Lógica de aviso Legal

A pesar de este motivo, esta vista ya utiliza el patrón estrategia para el uso de los delegados del UIWebView. Lo que permite interactuar y re-implementar las funciones definidas por UIKit para este objeto.

4.1.4 Pantalla Principal

La pantalla principal encapsula la gran parte de la funcionalidad de Tendfy, proporcionando un punto de acceso a todos y cada uno de sus casos de uso. Esta vista viene embebido dentro de dos tipos de vista, un UINavigationBar que ya vimos en la vista de aviso legal, pero también tiene un UITabBar. Este último permite el movimiento entre vistas mediante unos botones que siempre están presentes en la parte inferior de la vista.

Imagen 60. Pantalla principal

La vista está formada por un UINavigationBar y un UITabBar como ya mencionamos antes, pero también tiene dos UIButtons en la parte superior que nos permiten acceder al Filtro (icono superior izquierda) o interactuar con la carta que se encuentra en el centro de la vista de la que más tarde hablaremos (icono superior derecha).

Los botones inferiores permiten la navegación al perfil de usuario, la propia pantalla principal y la sección de favoritos.

Al ser la vista principal que el usuario verá nada más abrir la aplicación tiene un aspecto llamativo y focalizando de inmediato el fin por el que un usuario instalaría la aplicación. Esto permite que el usuario pueda comenzar a utilizarla nada más abrirla sin tener que hacer que se tenga que desplazar entre demasiadas vistas.

4.1.4.1 Estructura y patrones

Esta quizás sea la vista con más lógica de negocio, ya que permite no solo la navegación hasta cualquier otro caso de uso, si no que aquí es donde se haya la funcionalidad completa de la aplicación.

La estructura de carpetas de esta pantalla también es la más compleja, ya no solo definida por el patrón de diseño *MVC*, si no que necesita ciertos ayudadores (Helpers), para poder realizar la función de interacción con las cartas.

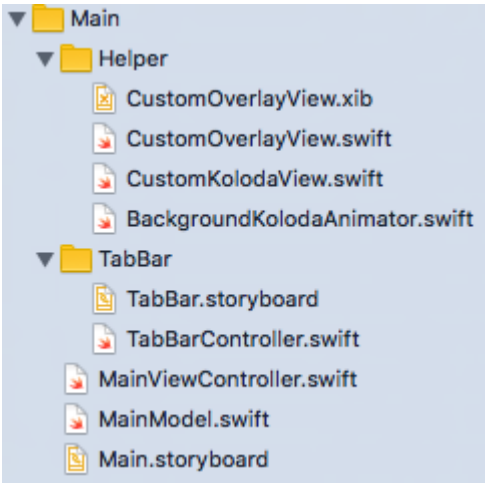


Imagen 61. Estructura de la pantalla principal

Como podemos apreciar cada vista de cada uno de los casos de uso, algunos como esta pantalla dispone de sus propios *.Storyboard*, esto es una decisión de diseño, permite que la carga de estos ficheros que como comenté nos muestran la interfaz de la aplicación sea mucho más rápida, que utilizando un sólo fichero *.Storyboard*. Además nos permite agilizar el trabajo mediante repositorio, impidiendo que varios desarrolladores modifiquen uno de estos ficheros y produzca un conflicto grave al unir el código.

Además de utilizar el patrón Modelo Vista Controlador, también dispone del patrón plantilla. Utilizando una plantilla de UIViewController con unos pre-ajustes necesarios. Esto es muy útil cuando necesitamos que varias vistas utilicen o tengan ciertas propiedades, evitamos repetirlas en cada una de estas pantallas, creando una plantilla y creando pantallas que la utilicen.

Como ya se mencionó esta vista utiliza el patrón *MVC*, de esta manera podemos separar la lógica y facilitar tareas como el testing.

```
import UIKit
import Koloda
import pop

class MainViewController: ViewWithREDNavTemplate {

    @IBOutlet var filterButton: UIBarButtonItem!
    @IBOutlet weak var kolodaView: CustomKolodaView!

    let sizeImagePulse: CGFloat = UIScreen.mainScreen().bounds.width/3
    let model: MainModel = MainModel()

    override func viewDidLoad() {
        super.viewDidLoad()
        model.numberOfCards = model.shoes.count
        if revealViewController() != nil {
            filterButton.target = revealViewController()
            filterButton.action = #selector(SWRevealViewController.revealToggle(_:))
            revealViewController().rightViewRevealWidth = 150
        }
        kolodaView.alphaValueSemiTransparent = model.kolodaAlphaValueSemiTransparent
        kolodaView.countOfVisibleCards = model.kolodaCountOfVisibleCards
        kolodaView.delegate = self
        kolodaView.dataSource = self
        kolodaView.animator = BackgroundKolodaAnimator(koloda: kolodaView)

        self.modalTransitionStyle = UIModalTransitionStyle.FlipHorizontal
    }

    //MARK: - Koloda Delegate
    func kolodaDataSource() -> [UIImage] {
        var array: Array<UIImage> = []
        for index in 0..<model.shoes.count {
            array.append(UIImage(named: model.shoes[index].image)!)
        }
        return array
    }
    @IBAction func leftButtonTapped() {
        kolodaView?.swipe(SwipeResultDirection.Left)
        //Unlike Action
    }
    @IBAction func rightButtonTapped() {
        kolodaView?.swipe(SwipeResultDirection.Right)
        //Like Action
    }
    @IBAction func undoButtonTapped() {
        kolodaView?.revertAction()
        //Undo Action
    }
}
```

Imagen 62. Lógica del Controlador

Aquí podemos ver el contenido del modelo, el cual se encarga de almacenar los datos de la aplicación que recoge el controlador de la vista. Los datos sobre el calzado se han rellenado con datos falsos para poder mostrar cómo funciona.

Esta vista también utiliza el patrón Objeto para la rápida generación de grupos de propiedades, como podemos observar con el objeto *Shoes*.

```swift
import Foundation
import MapKit

class MainModel {
    var numberOfCards: Int = 0
    let frameAnimationSpringBounciness: CGFloat = 9
    let frameAnimationSpringSpeed: CGFloat = 16
    let kolodaCountOfVisibleCards = 2
    let kolodaAlphaValueSemiTransparent: CGFloat = 0.1


    let shoes: [Shoe] = [
        Shoe(image: "shoe1", shoeName: "Zapato Tacón 1", price: 79.49, discountPrice: 59.49, trademark: "Menbur", size: [36, 37, 38, 39],
            color: ["Negro", "Marrón"], description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at ipsum lectus.
            Suspendisse faucibus velit nec ipsum sagittis, sit amet volutpat dolor tincidunt. ", material: ["Piel", "Cuero"], gender: 1,
            distance: 2.3, geoPosition: CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0)),
        Shoe(image: "shoe2", shoeName: "Zapato Tacón 2", price: 79.49, discountPrice: 59.49, trademark: "Menbur", size: [36, 37, 38, 39],
            color: ["Rojo", "Blanco"], description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at ipsum lectus.
            Suspendisse faucibus velit nec ipsum sagittis, sit amet volutpat dolor tincidunt. ", material: ["Piel", "Cuero"], gender: 1,
            distance: 2.3, geoPosition: CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0)),
        Shoe(image: "shoe3", shoeName: "Zapato Tacón 3", price: 79.49, discountPrice: 59.49, trademark: "Menbur", size: [36, 37, 38, 39],
            color: ["Verde", "Beige"], description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at ipsum lectus.
            Suspendisse faucibus velit nec ipsum sagittis, sit amet volutpat dolor tincidunt. ", material: ["Piel", "Cuero"], gender: 1,
            distance: 2.3, geoPosition: CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0)),
        Shoe(image: "shoe4", shoeName: "Zapato Tacón 4", price: 79.49, discountPrice: 59.49, trademark: "Menbur", size: [36, 37, 38, 39],
            color: ["Negro"], description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at ipsum lectus. Suspendisse
            faucibus velit nec ipsum sagittis, sit amet volutpat dolor tincidunt. ", material: ["Piel", "Cuero"], gender: 1, distance: 2.3,
            geoPosition: CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0)),
        Shoe(image: "shoe5", shoeName: "Zapato Tacón 5", price: 79.49, discountPrice: 59.49, trademark: "Menbur", size: [36, 37, 38, 39],
            color: ["Rosa", "Blanco"], description: "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Curabitur at ipsum lectus.
            Suspendisse faucibus velit nec ipsum sagittis, sit amet volutpat dolor tincidunt. ", material: ["Piel", "Cuero"], gender: 1,
            distance: 2.3, geoPosition: CLLocationCoordinate2D(latitude: 0.0, longitude: 0.0))
    ]


}
```

Imagen 63. Modelo de la vista

4.1.5 Pantalla Filtro

El filtro es la otra sección con mayor parte de lógica de la aplicación, es la vista encargada de realizar petición de productos al servidor mediante una serie de parámetros que pasan mediante la interfaz. Esta pantalla se muestra de una forma peculiar comparada con las demás, ya que no utiliza una navegación nativa del sistema, si no que se tiene que implementar de manera separada. La disposición de esta vista realiza un barrido de la pantalla principal, mostrando el filtro debajo de esta.

Además el filtro dispone de múltiples tipos de objetos gráficos, para cada uno de los tipos de filtro que podemos aplicar a los productos. El filtro está formado por los siguientes componentes:

- **UITableView**: Es una tabla en forma de lista que muestra los tipos de productos que podemos encontrar en la aplicación. Para utilizar esta lista, tendremos que pulsar en el tipo por el que deseamos filtrar y este nos responderá con un *check* indicando lo que hemos seleccionado.
- **UITextField**: Podemos encontrar dos objetos de este tipo en el filtro, que son utilizados para que el usuario indique el precio mínimo y máximo de los productos deseados. Ambos campos están limitados a que sean de tipo numérico, por lo que solo se mostrará el teclado numérico cuando interactuemos con ellos, también dispone un máximo de carácteres.
- **UICollectionView**: Utilizado para mostrar una serie de cuadros pintados con cada uno de los colores por los que podemos filtrar el calzado.
- **UIButton**: Se encuentran dos de estos elementos, el primero para navegar hasta la vista de marcas y el siguiente de color amarillo en la parte inferior para resetear el filtro de la aplicación.

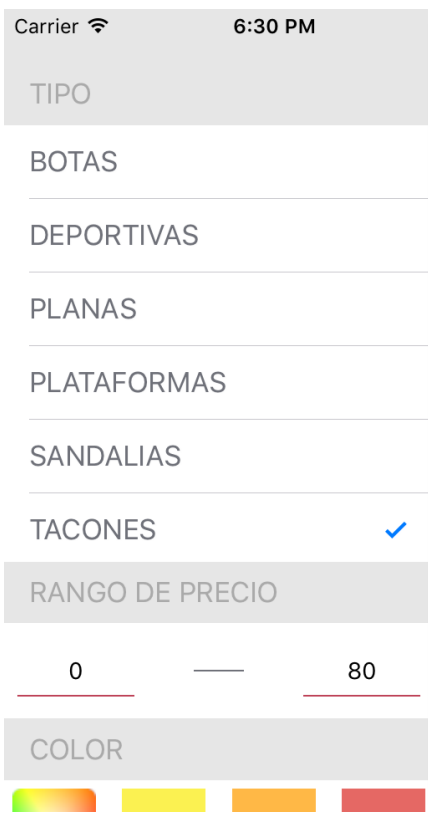


Imagen 64. Parte superior Filtro

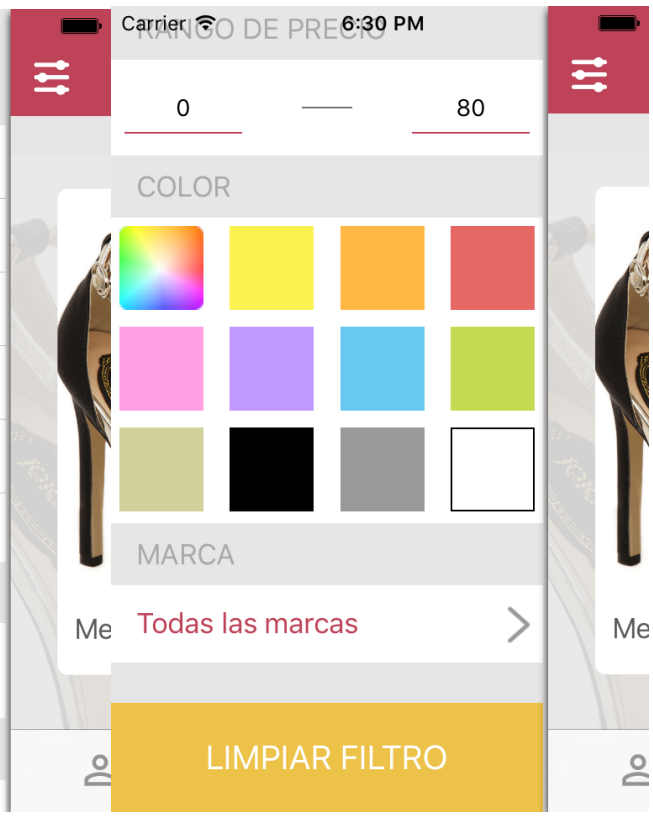


Imagen 65. Parte inferior Filtro

4.1.5.1 Estructura y patrones

El MVC también está presente en esta pantalla, su estructura de ficheros sigue la línea de otras vistas que disponen de la aplicación de este patrón.

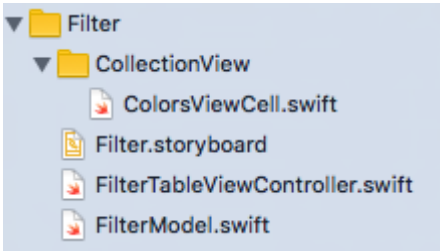
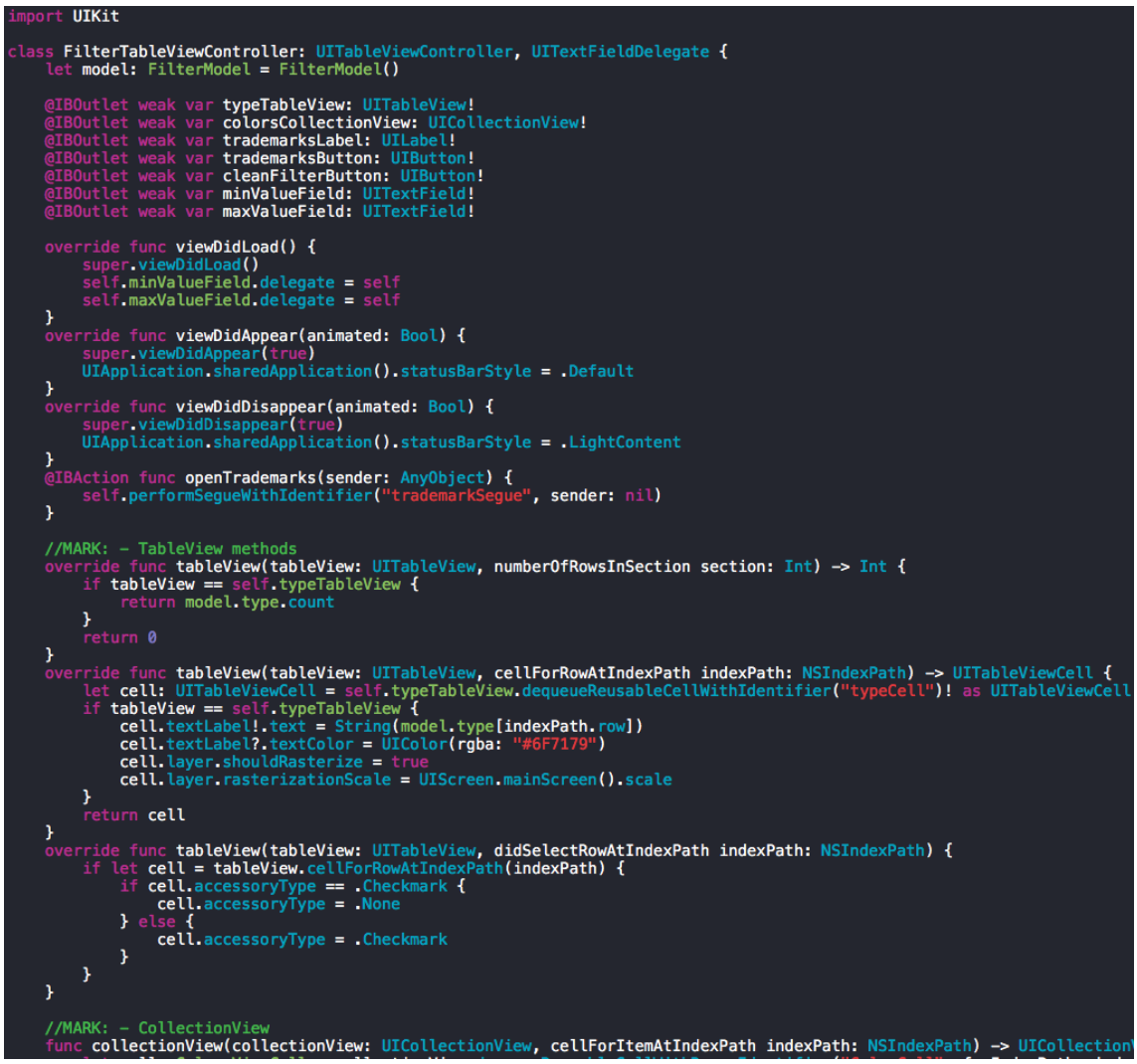


Imagen 66. Estructura del filtro

La lógica del modelo de esta vista quizás sea la más sofisticada de toda la aplicación, ya que tiene que tratar numerosos datos de distintos objetos distintos para realizar la acción de filtrado.

El patrón estrategia es utilizado para el acceso de delegados externos a la propia vista.



```swift
import UIKit

class FilterTableViewController: UITableViewController, UITextFieldDelegate {
    let model: FilterModel = FilterModel()

    @IBOutlet weak var typeTableView: UITableView!
    @IBOutlet weak var colorsCollectionView: UICollectionView!
    @IBOutlet weak var trademarksLabel: UILabel!
    @IBOutlet weak var trademarksButton: UIButton!
    @IBOutlet weak var cleanFilterButton: UIButton!
    @IBOutlet weak var minValueField: UITextField!
    @IBOutlet weak var maxValueField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()
        self.minValueField.delegate = self
        self.maxValueField.delegate = self
    }
    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(true)
        UIApplication.sharedApplication().statusBarStyle = .Default
    }
    override func viewDidDisappear(animated: Bool) {
        super.viewDidDisappear(true)
        UIApplication.sharedApplication().statusBarStyle = .LightContent
    }
    @IBAction func openTrademarks(sender: AnyObject) {
        self.performSegueWithIdentifier("trademarkSegue", sender: nil)
    }

    //MARK: - TableView methods
    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        if tableView == self.typeTableView {
            return model.type.count
        }
        return 0
    }
    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell: UITableViewCell = self.typeTableView.dequeueReusableCellWithIdentifier("typeCell")! as UITableViewCell
        if tableView == self.typeTableView {
            cell.textLabel!.text = String(model.type[indexPath.row])
            cell.textLabel?.textColor = UIColor(rgba: "#6F7179")
            cell.layer.shouldRasterize = true
            cell.layer.rasterizationScale = UIScreen.mainScreen().scale
        }
        return cell
    }
    override func tableView(tableView: UITableView, didSelectRowAtIndexPath indexPath: NSIndexPath) {
        if let cell = tableView.cellForRowAtIndexPath(indexPath) {
            if cell.accessoryType == .Checkmark {
                cell.accessoryType = .None
            } else {
                cell.accessoryType = .Checkmark
            }
        }
    }

    //MARK: - CollectionView
    func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollection
```

Imagen 67. Lógica del controlador

Gran parte del código que podemos observar en este controlador sirve para la composición y rellenado de cada uno de los objetos de la vista. Esta vista no utiliza una plantilla ya que es una vista especial como ya se comentó, por lo que no sería necesario generarla.

En el modelo sería donde podemos encontrar la parte carnosa del código de la vista.

Mediante una serie de enumeraciones podemos controlar qué tipo de acción se está utilizando sobre el filtro y sus objetos. Primero podemos observar una serie de *Arrays* que nos dan los datos básicos para poder rellenar los datos de la vista

```
import Foundation

class FilterModel {
    let type = ["BOTAS", "DEPORTIVAS", "PLANAS", "PLATAFORMAS", "SANDALIAS", "TACONES"]
    let colors = [
        ["name": "MULTICOLOR", "uicolor": UIColor(rgba: "#fff"), "id": 0],
        ["name": "AMARILLO", "uicolor": UIColor(rgba: "#fbf151"), "id": 1],
        ["name": "NARANJA", "uicolor": UIColor(rgba: "#ffb846"), "id": 2],
        ["name": "ROJO", "uicolor": UIColor(rgba: "#e56864"), "id": 3],
        ["name": "ROSA", "uicolor": UIColor(rgba: "#ff9fe4"), "id": 4],
        ["name": "MORADO", "uicolor": UIColor(rgba: "#c299ff"), "id": 5],
        ["name": "AZUL", "uicolor": UIColor(rgba: "#68c8f2"), "id": 6],
        ["name": "VERDE", "uicolor": UIColor(rgba: "#c3db51"), "id": 7],
        ["name": "MARRON", "uicolor": UIColor(rgba: "#d0d09b"), "id": 8],
        ["name": "NEGRO", "uicolor": UIColor(rgba: "#000"), "id": 9],
        ["name": "GRIS", "uicolor": UIColor(rgba: "#999"), "id": 10],
        ["name": "BLANCO", "uicolor": UIColor(rgba: "#fff"), "id": 11],
        ]

    var filterArray: [[String]] = [
        [],
        [],
        ["0", "2000"],
        []
    ]

    enum typeValue {
        case COLOR
        case TYPE
        case PRICE
        case TRADEMARK
    }
    enum actionValue {
        case DELETE
        case ADD
        case EDIT
    }
    enum typePrice: Int {
        case MIN = 0
        case MAX = 1
    }

    func manageValue(value: String, type: typeValue, action: actionValue, priceType: typePrice?) {
        var destination = -1

        switch type {
        case .COLOR:
            destination = 0
            break
        case .TYPE:
            destination = 1
            break
        case .PRICE:
            destination = 2
            break
        case .TRADEMARK:
            destination = 3
            break
        }
        switch action {
```

Imagen 68. Modelo de la vista

Como podemos observar en la parte inferior del código tenemos un manejador de datos el cual se encarga de operar con cada valor independientemente del tipo que sea, mediante los parámetros que reciba el método, este reconoce la clase de objeto de la interfaz sobre el que estamos trabajando y mandará la respuesta adecuada.

El código de esta vista se ha abstraído de manera que si en un futuro se desea añadir nuevos valores de filtrado, se puedan añadir con la mínima modificación de código.

4.1.6 Pantalla de selección de marcas

La selección de marcas es sólo accesible desde la pantalla de filtro. Selección de marcas nos permite seleccionar las marcas de los productos disponibles en la aplicación. Estas marcas se muestran en forma de lista al igual que los tipos de calzado de la vista de filtro, pero esta vez ocupando toda la pantalla de la aplicación. También tenemos accesible una barra de búsqueda que nos permite encontrar de manera más ágil las marcas deseadas.

El componente estrella en esta vista es el UITableView, para mostrar cada una de las marcas.

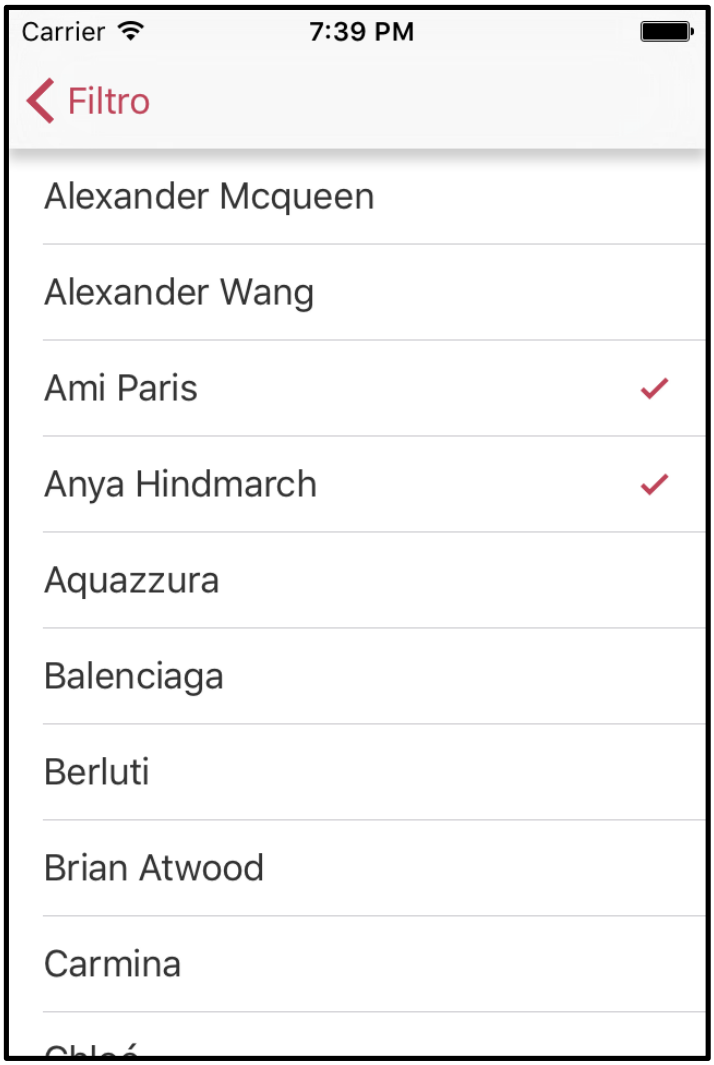


Imagen 69. Selección de marcas

4.1.6.1 Estructura y patrones

Debido al poco contenido que tiene esta vista, se decidió no generar un modelo. Se utilizó el patrón estrategia para el uso de los delegados del UISearch.

```swift
import UIKit
import Parse

class TrademarkTableViewController: UITableViewController, UISearchResultsUpdating {
    var selected: [Int] = []
    var trademarks: [String] = []
    var selectedTrademarks: [String] = []

    var indicator: UIActivityIndicatorView = UIActivityIndicatorView()
    var filteredTableData = [String]()
    let searchController = UISearchController(searchResultsController: nil)
    var resultSearchController: Bool = false

    override func viewDidLoad() {
        super.viewDidLoad()
        self.prepareSearchBar()
        self.tableView.tableHeaderView = searchController.searchBar
    }

    override func viewDidAppear(animated: Bool) {
        self.loadTrademarks()

        self.indicator.activityIndicatorViewStyle = UIActivityIndicatorViewStyle.WhiteLarge
        self.indicator.color = UIColor(rgba: "#BE4358")
        self.indicator.center = self.view.center
        self.view!.addSubview(indicator)
        self.indicator.bringSubviewToFront(self.view!)
    }

    //Mark: - TableView DataSource
    override func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        if self.resultSearchController == true {
            return self.filteredTableData.count
        } else {
            return self.trademarks.count
        }
    }

    override func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell: TrademarkTableViewCell = self.tableView.dequeueReusableCellWithIdentifier("Cell")! as! TrademarkTableViewCell

        if self.resultSearchController == true {
            cell.trademarkLabel!.text = String(self.filteredTableData[indexPath.row]).capitalizedString
        } else {
            cell.trademarkLabel!.text = String(self.trademarks[indexPath.row]).capitalizedString
        }

        cell.trademarkLabel?.textColor = UIColor(rgba: "#333")
        let pos = self.whereIsThisTrademark(cell,trademarkLabel.text!)

        if self.selected[pos] == 0 {
            cell.accessoryType = .None
        } else {
            cell.accessoryType = .Checkmark
        }

        cell.layer.shouldRasterize = true
        cell.layer.rasterizationScale = UIScreen.mainScreen().scale

        return cell
    }
```

Imagen 70. Lógica del controlador

La lógica de este controlador se encarga de recibir los nombres de las marcas que tiene almacenadas el servidor y mostrarlas mediante una tabla. El usuario selecciona las marcas que desea y pulsa el botón atrás para terminar. En este último paso la vista de selección envía al filtro toda aquella información que ha marcado el usuario.

4.1.7 Pantalla de perfil de usuario

Aquí encontramos una vista que muestra la información del usuario, esta pantalla está preparada para una futura implementación relacionada con una parte social.

La parte superior dispone UIButton que nos llevaría hasta la vista de ajustes. También tenemos dos UImageView, que muestran tanto la imagen de portada como la de perfil de Facebook del usuario, esta primera se trata con un *blur* que desenfoca la imagen. Continuando hacia abajo, encontramos un UILabel, que sería una etiqueta de texto la cual indica el nombre del usuario. Finalmente un UITableView con la información más específica.

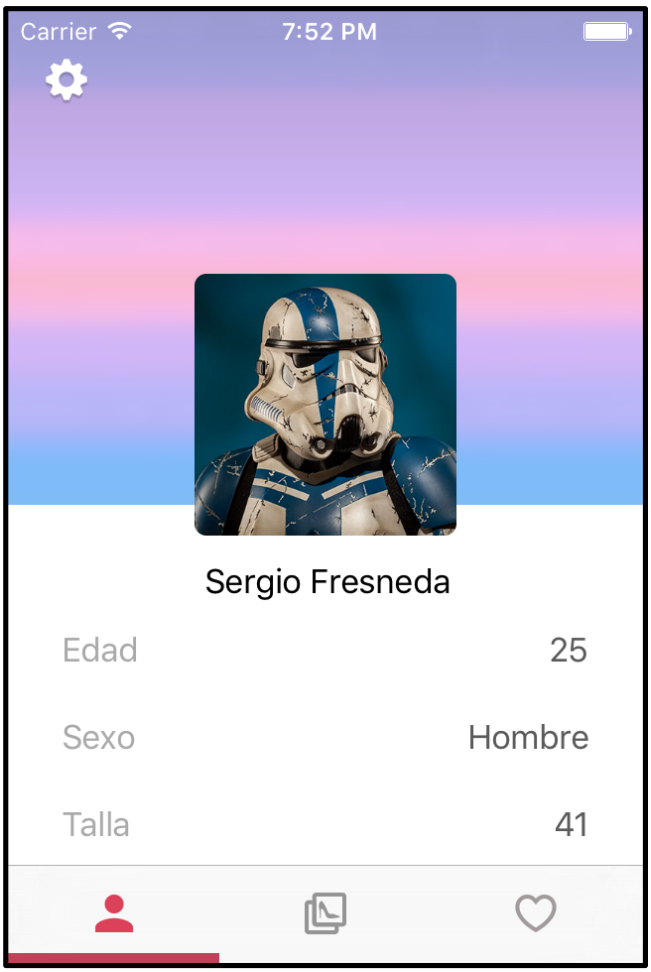


Imagen 70. Pantalla de perfil de usuario

Todos los datos representados en esta pantalla a excepción de la talla de calzado, los proporciona la red social Facebook, con la cual nos registramos en la aplicación.

4.1.7.1 Estructura y patrones

La estructura de carpetas ya delata el uso del *MVC*, además del estrategía una vez más para el acceso a los delegados del UITableView. En esta ocasión se decidió por diseño generar un *helper* o ayudador por la futura posibilidad de extender esta vista a un modelo más social, con acceso a perfiles de amigos.

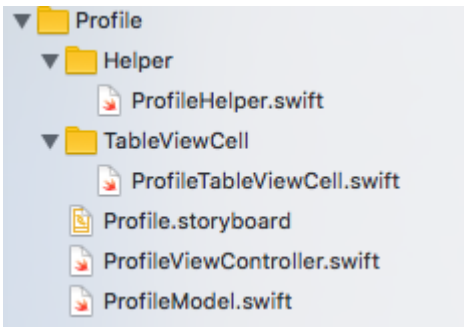


Imagen 71. Estructura de perfil de usuario

Utilizando el modelo como almacén de datos, el controlador accede en cualquier momento que necesite información para rellenar la vista.

```
class ProfileViewController: UIViewController, UITableViewDelegate {

    @IBOutlet weak var settingsButton: UIButton!
    @IBOutlet weak var headerImageView: UIImageView!
    @IBOutlet weak var profileImageView: UIImageView!
    @IBOutlet weak var usernameLabel: UILabel!

    @IBOutlet weak var tableView: UITableView!


    let model: ProfileModel = ProfileModel()


    override func viewDidLoad() {
        super.viewDidLoad()
        self.model.dataForTableView = ProfileHelper.prepareDataForTableView(model)

        self.tableView.delegate = self
        self.tableView.tableFooterView = UIView()

        self.decoreView()
    }

    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(true)
    }
    override func viewWillAppear(animated: Bool) {
        super.viewWillAppear(true)
        self.model.updateUserData = UserDefaultsProvider.loadUserData()
        self.decoreView()
        UIApplication.sharedApplication().statusBarStyle = .LightContent
        self.tableView.reloadData()
    }

    @IBAction func goToSettingsView(sender: AnyObject) {
        self.presentViewController(UIStoryboard(name: "Settings", bundle:nil).instantiateViewControllerWithIdentifier("SettingsNav") as!
            UINavigationController, animated:true, completion:nil)
    }

    // MARK: - TableView

    func tableView(tableView: UITableView, cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
        let cell: ProfileTableViewCell = self.tableView.dequeueReusableCellWithIdentifier("ProfileCell") as! ProfileTableViewCell
        cell.titleLabel.text = self.model.headersTableView[indexPath.row]
        cell.valueLabel.text = self.model.dataForTableView[indexPath.row]

        return cell
    }

    func tableView(tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return self.model.dataForTableView.count
    }

    func decoreView() {
        ProfileHelper.setProfileImageView(self.model, profileImageView: self.profileImageView, headerImageView: self.headerImageView)
        self.usernameLabel.text = self.model.userData.getCompleteName
    }
}
```

Imagen 72. Lógica del controlador

```
import Foundation

class ProfileModel {

    var userData: User = UserDefaultsProvider.loadUserData()
    let headersTableView: [String] = ["Edad", "Sexo", "Talla"]
    var dataForTableView: [String] = []

    var updateUserData: User {
        get {
            return userData
        }
        set (user) {
            userData = user
        }
    }
}
```

Imagen 73. Modelo de la vista

Una vez más volvemos a encontrarnos con el proveedor de datos NSUserDefaults, en este caso para cargar los datos recibidos de la red social.

```
import UIKit

class ProfileHelper {

    class func prepareDataForTableView(model: ProfileModel) -> [String] {
        var array: [String] = []
        array.append(model.userData.getAge)
        array.append(model.userData.stringGender)
        array.append(model.userData.stringShoeSize)

        return array
    }
    class func setProfileImageView(model: ProfileModel, profileImageView: UIImageView, headerImageView: UIImageView) {
        if model.userData.profileImage != nil {
            profileImageView.image = UIImage(named: model.userData.profileImage!)
        }
        if model.userData.headerImage != nil {
            headerImageView.image = UIImage(named: model.userData.headerImage!)
        }
        OutletsTreatment.roundImageView(profileImageView, radius: 6)
        OutletsTreatment.blurImage(headerImageView, radius: 6, iterations: 10)
    }
}
```

Imagen 74. Helper de la vista

Mediante el uso del ayudante de la vista se prepara la información para ser mostrada, también se realiza cierta decoración de la vista, como son la presentación de la imagen de perfil y portada del usuario. Todo ello con el fin de no acumular código en el controlador y una vez más tener un código más limpio y manejable.

4.1.8 Pantalla de ajustes y AboutUs

Ambas vistas las explicaré en el mismo punto debido al poco contenido que aportan a nivel de código y arquitectura. Simplemente se mostrará lo que permite cada una de estas, describiendo cómo están formadas.

Comenzamos con la ventana de ajustes, se trata de una vista que aparecerá de manera modal, es decir que se superpone de manera obligatoria a la vista anterior. Utilizando el patrón plantilla para la decoración de la vista. Simplemente muestra las secciones de ajustes de la aplicación como podemos ver en la imagen 54. La más interesante es editar perfil, que se explicará detalladamente más adelante.

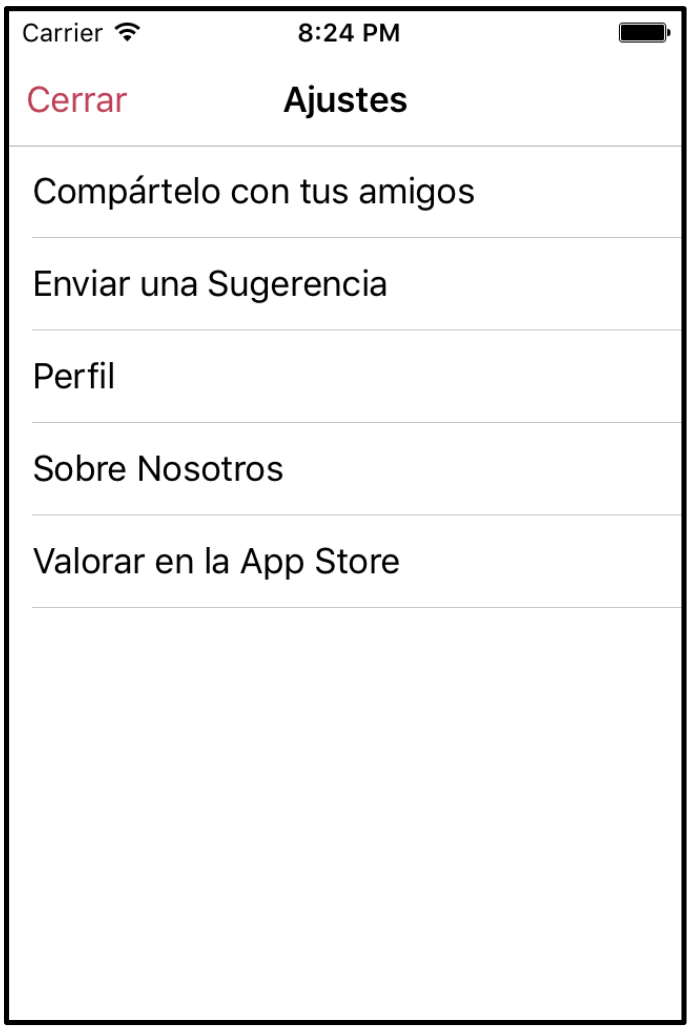


Imagen 75. Vista de ajustes

Cada uno de los componentes de la lista nos invita a navegar hacia una función o vista distinta de la aplicación, pasaré a comentar cada una de ellas a continuación:

- **Compártelo con tus amigos**: Lanzará lo que conocemos como un UIAlert para compartir la aplicación en las redes sociales.
- **Enviar una Sugerencia**: Nos abrirá de manera modal una vista para mandar un correo a la dirección de correo de soporte de Tendfy.
- **Perfil**: Nos llevará hacia la vista de edición de perfil, la cual trataremos en el siguiente punto.
- **Sobre Nosotros (AboutUs)**: Muestra una vista con información sobre la empresa.
- **Valorar en la App Store**: Abre la página de la AppStore donde se encuentra la aplicación de Tendfy para poder valorarla.

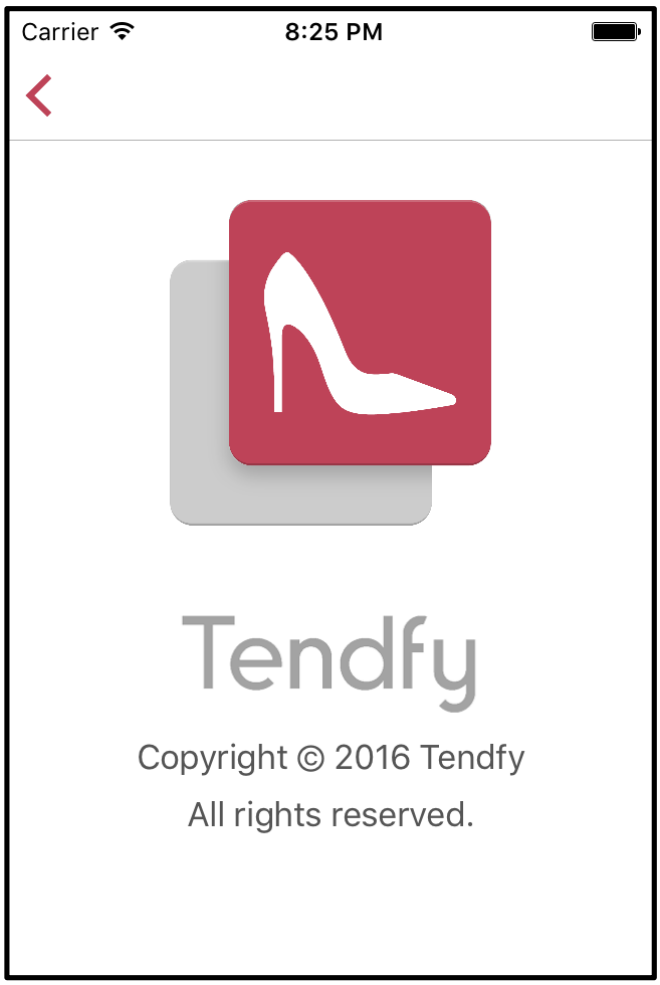


Imagen 76. Vista de AboutUs

4.1.9 Pantalla de edición de perfil

Desde esta vista tendremos acceso a casi todos los datos de los que dispone la aplicación sobre el usuario, para poder realizar modificaciones. La disposición de los elementos de esta vista ha sido muy variado a lo largo del periodo de vida de la aplicación. Todo esto debido a las necesidades de la empresa.

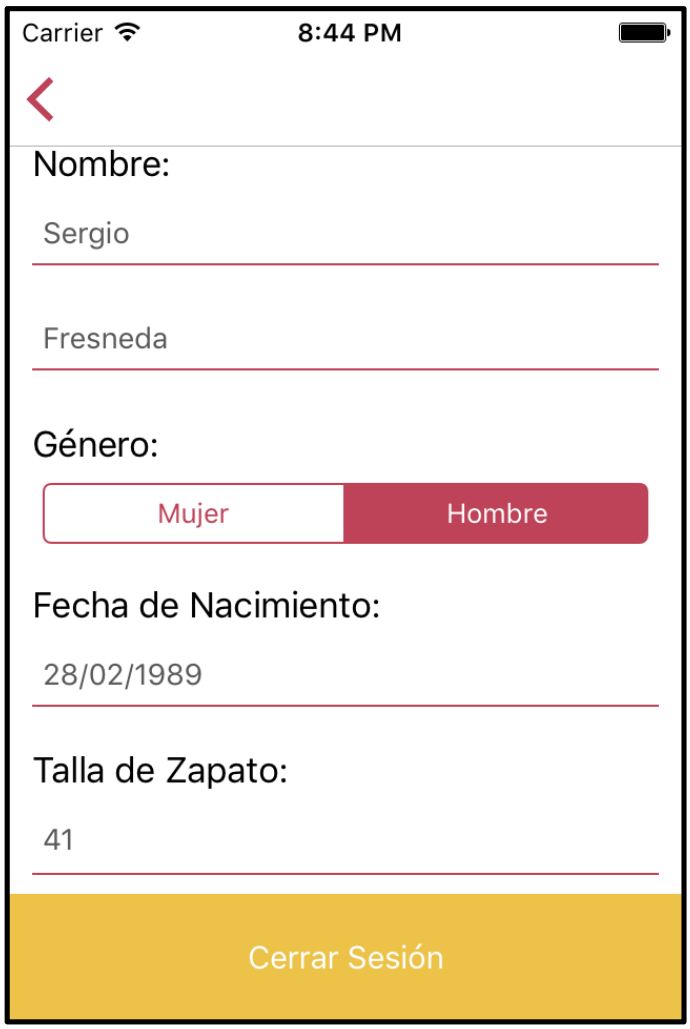


Imagen 77. Vista de edición de perfil

Finalmente se estableció un diseño y disposición final la cual se va a comentar a continuación, hay que indicar que todos los campos del perfil son del tipo UITextField, a excepción del UISegmented que indica el género del usuario y el UIButton que permite cerrar sesión en la *App*:

- **Campos de nombre**: Ambos disponen de un *placeholder* indicativo cuando el usuario no tiene rellenos los campos, en los que le indica qué tipo de contenido tiene que introducir dentro de estos. El contenido de estos campos serán los que posteriormente rellenen el nombre del usuario en la vista de perfil. Ambos campos están limitados por número de carácteres.
- **Selector de género**: Mediante un tono rojo este elemento de la interfaz indica el valor seleccionado. Este no permite la selección de más de un valor, es decir que sólo es posible la selección del género hombre o mujer. El género indicado hará variar el tipo de productos que recibiremos en la pantalla de main, filtrándose por género.

- **Campo fecha de nacimiento**: Este campo de texto al igual que el de talla, tienen la peculiaridad de que no muestran un teclado al pulsar sobre ellos, sino que muestran un selector conocido como UIPicker. En este caso la fecha mostraría un *picker* del tipo UIDatePicker como podemos ver en la imagen 56. Este campo tiene una limitación que impide que la fecha de nacimiento sea futura a la actual, o que se puedan tener un número máximo de años.
- **Campo talla**: Aquí podemos indicar la talla de calzamos que tiene el usuario. Al igual que el género este campo también interacciona con el filtro para mostrar productos que concuerden con nuestra talla. Como se mencionaba antes, este campo no muestra un teclado al entrar en él para su edición, sino que obtendremos un UIPicker, pero con unos números que indican la talla.

Cabe mencionar que todo el formulario de edición de perfil cuenta con un manejador que permite moverse entre los distintos campos de texto, sin la necesidad de ocultar el teclado en ningún momento. Esto es debido a una librería externa que se comentará en la siguiente sección.

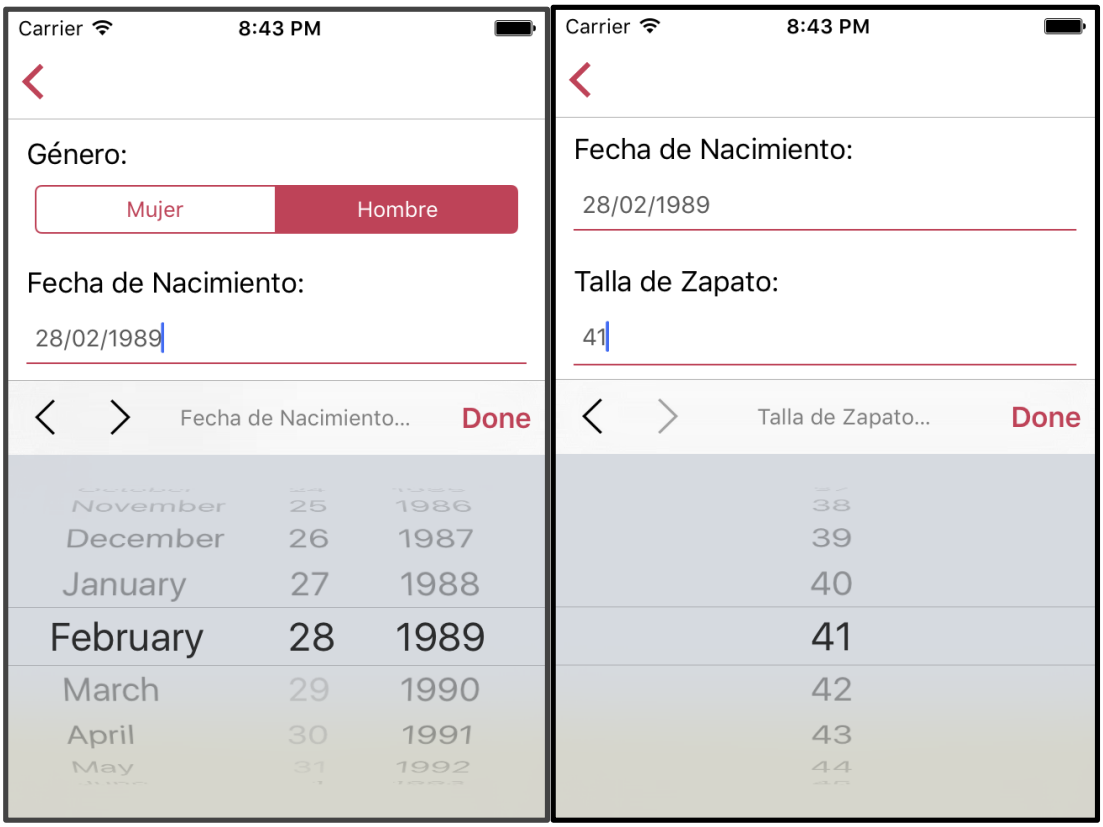


Imagen 78. UIDatePicker

Imagen 79. UIPicker talla

4.1.9.1 Estructura y patrones

Una vez más el patrón MVC se muestra en la estructura de carpetas de este caso de uso.

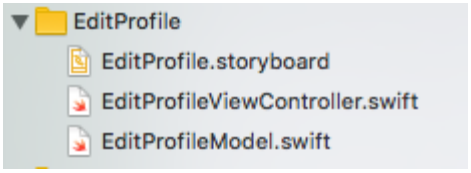


Imagen 80. Estructura de edición de perfil

También la utilización del patrón plantilla, esta se comparte con toda la sección de ajustes de la aplicación, principalmente para la decoración de la vista. El patrón Objeto es visible desde el controlador, mediante el uso del objeto *User*, el cual utiliza para manejar los datos del formulario y cargarlos/almacenarlos en el NSUserDefaults para su posterior almacenamiento en el servidor.

```
import UIKit

class EditProfileViewController: ViewWithWHITENavTemplate, UITextFieldDelegate, UIPickerViewDelegate, UIPickerViewDataSource {

    let model: EditProfileModel = EditProfileModel()

    let birthPickerInput = UIDatePicker()
    let pickerInput = UIPickerView()

    @IBOutlet weak var nameField: UITextField!
    @IBOutlet weak var lastNameField: UITextField!

    @IBOutlet weak var genderSegmentedControl: UISegmentedControl!

    @IBOutlet weak var birthdayField: UITextField!
    @IBOutlet weak var shoeSizeField: UITextField!

    override func viewDidLoad() {
        super.viewDidLoad()

        self.pickerInput.delegate = self
        self.birthPickerInput.datePickerMode = UIDatePickerMode.Date
        self.birthPickerInput.addTarget(self, action: #selector(EditProfileViewController.datePickerValueChanged), forControlEvents:
            UIControlEvents.ValueChanged)

        self.birthdayField.inputView = self.birthPickerInput
        self.shoeSizeField.inputView = self.pickerInput

        self.prepareDelegates()
        self.fillAllInputs()
    }

    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(true)
    }
    override func viewWillDisappear(animated: Bool) {
        super.viewWillDisappear(animated)
        UserDefaultsProvider.updateUserData(User(name: self.model.nameUser!, lastName: self.model.lastNameUser!, birthDayDate: self.model.
            birthDayUser!, shoeSize: self.model.shoeSize!, gender: self.model.genderUser!, email: nil, profileImage: nil, headerImage: nil))
    }

    //MARK: - UIPicker Data Source
    func numberOfComponentsInPickerView(pickerView: UIPickerView) -> Int {
        return 1
    }
    func pickerView(pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int {
        return self.model.pickOption.count
    }
    func pickerView(pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) -> String? {
        return self.model.pickOption[row]
    }
    func pickerView(pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int) {
        self.shoeSizeField.text = self.model.pickOption[row]
    }

    func datePickerValueChanged(sender: UIDatePicker) {
        self.birthdayField.text = sender.date.shortDate
    }
```

Imagen 81. Lógica del controlador

En cuanto al modelo, cabe mencionar que es utilizado para la carga del contenido del picker de talla y el contenido de los campos de texto del formulario.

```
import Foundation

class EditProfileModel {

    let pickOption: [String] = ["35", "36", "37", "38", "39", "40", "41", "42", "43", "44", "45"]

    var nameUser: String?
    var lastNameUser: String?
    var genderUser: Int?
    var birthDayUser: NSDate?
    var shoeSize: Int?
}
```

Imagen 82. Modelo de la vista

4.1.10 Pantalla de favoritos

La pantalla de favoritos muestra todos aquellos productos que el usuario ha marcado como interesantes para él, desde la vista principal de la aplicación. La forma de mostrar estos productos es mediante la ya nombrada anteriormente vista escaparate, formada por una UICollectionView, que muestra cada uno de los productos marcados como favorito en una celda distinta, indicando la información básica de este para su fácil reconocimiento. Este mismo tipo de diseño ya se nombró en los competidores, ya que es la manera más fácil de navegar, y tampoco es tan necesario el tipo de interacción que realiza el usuario con estas.

Como ya se mencionó la vista está formada en su totalidad por una UICollectionView, la cual nos permite listar datos de una manera gráfica y cómoda.

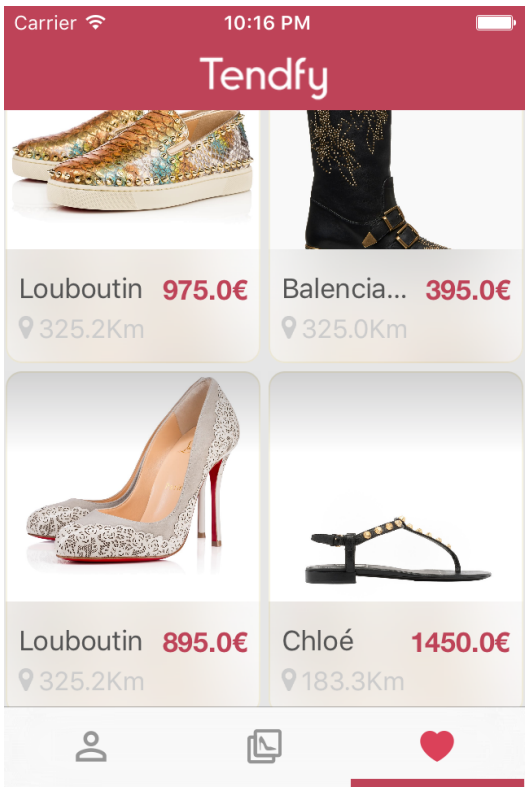


Imagen 83. Pantalla de favoritos

4.1.10.1 Estructura y patrones

El patrón *MVC* se repite en esta vista también, recordemos que separar la lógica permite la lectura y la modificación en un futuro de esa misma sea mucho más ágil y sencilla.

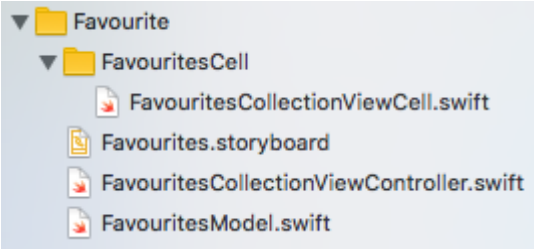


Imagen 84. Estructura de favoritos

El patrón plantilla actúa añadiendo funcionalidades sobre el estilo y los atributos de la barra de navegación.

```
import UIKit

class FavouritesCollectionViewController: CollectionViewWithREDNavTemplate {

    let model: FavouritesModel = FavouritesModel()
    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func viewDidAppear(animated: Bool) {
        super.viewDidAppear(animated)

        self.setCollectionViewLayout()
        self.checkTabBarVisibility()
    }
    override func viewWillDisappear(animated: Bool) {
        super.viewWillDisappear(animated)
    }
    //MARK: - CollectionView Methods
    override func collectionView(collectionView: UICollectionView, numberOfItemsInSection section: Int) -> Int {
        return self.model.favShoes.count
    }
    override func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath indexPath: NSIndexPath) -> UICollectionViewCell {
        let cell = collectionView.dequeueReusableCellWithReuseIdentifier(self.model.reuseIdentifier, forIndexPath: indexPath) as!
            FavouritesCollectionViewCell
        cell.layer.cornerRadius = 10
        cell.layer.masksToBounds = true

        cell.layer.borderColor = Constants().GOLD_COLOR.colorWithAlphaComponent(0.1).CGColor
        cell.layer.borderWidth = 1

        cell.mainImageView.image = UIImage(named: self.model.favShoes[indexPath.row].image)
        cell.titleLabel.text = self.model.favShoes[indexPath.row].trademark
        cell.priceLabel.text = String(self.model.favShoes[indexPath.row].discountPrice!)+"€"
        cell.locationLabel.text = String(self.model.favShoes[indexPath.row].distance)+"Km"
        cell.object = self.model.favShoes[indexPath.row]

        return cell
    }

    //MARK: - Scroll Delegate
    override func scrollViewDidScroll(scrollView: UIScrollView) {
        let verticalIndicator: UIImageView = (scrollView.subviews[(scrollView.subviews.count - 1)] as! UIImageView)
        verticalIndicator.backgroundColor = Constants().NAVIGATOR_BAR_COLOR

    }
    override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
        if (segue.identifier == "showDetail") {
            UIView.animateWithDuration(0.2, animations: {
                self.tabBarController?.tabBar.frame = CGRectMake((self.tabBarController?.tabBar.frame.origin.x)!, (self.tabBarController?.
                    tabBar.frame.origin.y)!+100, (self.tabBarController?.tabBar.frame.size.width)!,
                    (self.tabBarController?.tabBar.frame.size.height)!)
                self.model.hidden = true
            })

            let destination = segue.destinationViewController as! DetailViewController
            let cell = sender as! FavouritesCollectionViewCell

            destination.object = (cell.object)!
        }
    }
```

Imagen 85. Lógica del controlador

La pantalla de favoritos no realiza grandes acciones en su lógica, debido a que sólo debe recibir información sobre los zapatos marcados como favoritos del servidor y los muestra en una tabla. La segunda función que realiza es el traspaso de información sobre un producto, para mostrarlos en la vista de detalle, que explicaremos en el siguiente punto.

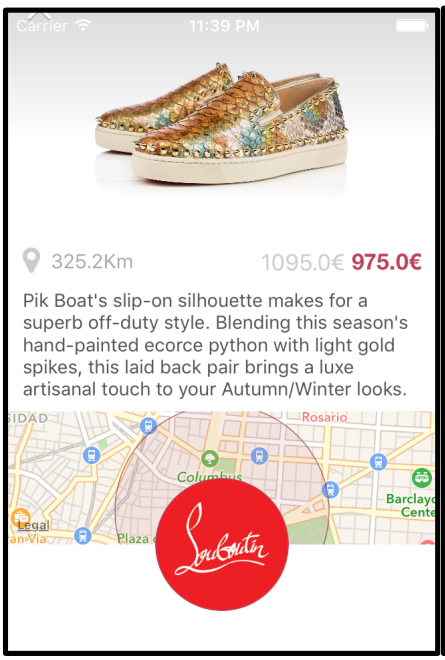
```
import Foundation
import MapKit

class FavouritesModel {

    let reuseIdentifier: String = "FavouriteCell"
    var hidden = false
    let favShoes: [Shoe] = []
}
```

Imagen 86. Modelo de la vista

El modelo de favoritos se encarga de mantener los datos de los zapatos recibidos del servidor como favoritos, para que posteriormente el controlador pueda acceder a ellas.

4.1.11 Pantalla detalle de producto

En esta pantalla podremos consultar en detalle la información sobre un producto. Desde la ubicación de la tienda más cercana donde lo venden, hasta las tallas disponibles, colores o materiales de los que está fabricado el calzado.



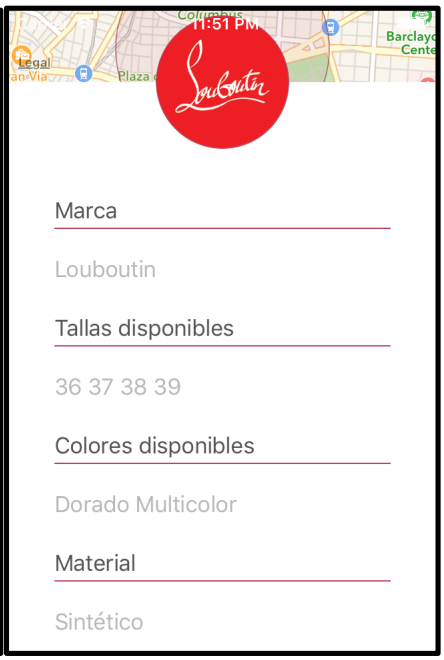


Imagen 87. Vista detalle parte superior **Imagen 88. Vista detalle parte inferior**

4.2 Librerías

Ahora vamos a ver algunas de las librerías con que hemos añadido al proyecto. Librerías completamente externas a iOS y que facilitarion extremadamente la implementación de ciertas funcionalidades.

Imagen 89. Cocoapods

Para la instalación de librerías externas, se utilizó CocoaPods, se trata de un repositorio de librerías de libre acceso. Primero generando un fichero binario, indicando que librerías deseas añadir a tu proyecto y guardándolo en la raíz del proyecto, después mediante el terminar navegar hasta la carpeta del proyecto y ejecutar el comando "*pod install*". Esto descargará e instalará las librerías de manera muy sencilla y limpia.

Ahora haremos un repaso a las librerías externas que se han instalado en la aplicación y para qué se han utilizado:

- **BWWalkthrough**: Esta librería permite la creación de un Walkthrough, es decir un tutorial iniciar para explicarle al usuario las funciones de la aplicación antes de entrar en materia.
- **FBSDKCoreKit**: Núcleo del SDK de Facebook para iOS, es la base para permitir el acceso a la API de la red social.
- **FBSDKLoginKit**: Extensión del SDK original que permite el login entre la app y la red social.
- **Parse**: SDK de Parse, permite la conexión con el servidor.
- **IQKeyboardManagerSwift**: Esta librería trabaja cuando el teclado aparece en pantalla, permite moverse entre los distintos campos de una vista añadiendo unos botones flotantes junto al teclado.
- **Koloda**: Es la librería que nos permite generar las cartas para mostrar los productos.

5. Publicación de la aplicación

En este punto comenzaremos contando un poco de historia sobre la tienda online de aplicaciones de iOS. Posteriormente como registrarse como desarrollador y los pasos a seguir para publicar una aplicación en la tienda.

5.1 AppStore

Imagen 90. AppStore

La AppStore nacida en 2007 junto a iPhoneOS 2.0 es una plataforma de distribución digital de aplicaciones móviles para los dispositivos con sistema operativo iOS. Esta aplicación permite a los usuarios navegar y descargar aplicaciones y juegos entre otros.

Las aplicaciones se pueden encontrar disponibles tanto de forma gratuita, como de pago. Estas son revisadas de manera exhaustiva por parte del equipo de aprobación de Apple. Con este proceso aseguran que las aplicaciones publicadas en su tienda funcionan de manera correcta y no realizan funciones sin que el usuario lo sepa.

5.2 Ser desarrollador

Lo primero que deberemos hacer es registrarnos en la web de desarrolladores de Apple, para darnos de alta como tal. Este registro es gratuito, pero no con ello tenemos acceso a publicar aplicaciones en la tienda, para ello tendremos que pagar una subscripción anual de 100€ en caso de ser un particular y 200€ en caso de ser una empresa. Hasta hace relativamente poco tiempo para poder testear una aplicación en un dispositivo real, era necesaria la suscripción anual, actualmente está permitida aún siendo una cuenta gratuita.

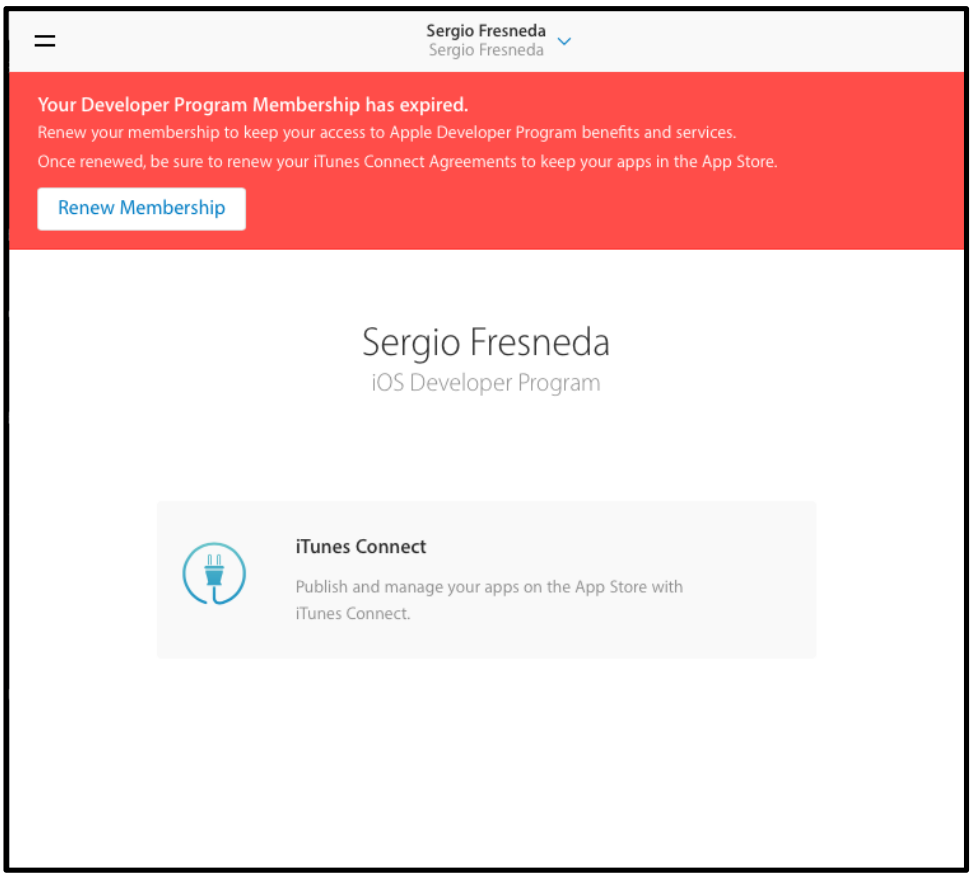


Imagen 91. Cuenta de desarrollador

5.3 Publicar la aplicación

Una vez disponemos de una cuenta de desarrollador suscrita, podremos subir nuestras aplicaciones.

Estos son los pasos a seguir:

1. Primero tendremos que generar el paquete de nuestra aplicación desde XCode. Desde el menú *Product>Archive*.
2. Esto nos abrirá una nueva ventana tras haber sido compilada nuestra aplicación sin errores. Ahora seleccionaremos la versión del paquete a subir en caso de tener varias. Pulsaremos el botón azul situado a la derecha.
3. A continuación nos preguntará con qué cuenta de desarrollador deseamos subirla en caso de tener varias configuradas en el *IDE*.
4. Una vez seleccionada nos indicará qué aplicación se va a subir a la tienda, aceptamos este paso y XCode comenzará a empaquetar la aplicación y a subirla.
5. Una vez finalizada la subida de nuestra *App*, nos mostrará un mensaje indicando que se ha subido correctamente.
6. Ahora tendremos que esperar aproximadamente dos días hasta que los revisores de aplicaciones de Apple den el visto bueno a nuestra aplicación.

7. Mientras los revisores realizan este paso, podremos rellenar la información necesaria de la aplicación, como pueden ser capturas de pantalla, descripción, icono de la aplicación etc...
8. Una vez haya pasado el filtro la aplicación, podremos seleccionar en qué países deseamos que aparezca nuestra aplicación.
9. Finalmente la publicaremos.

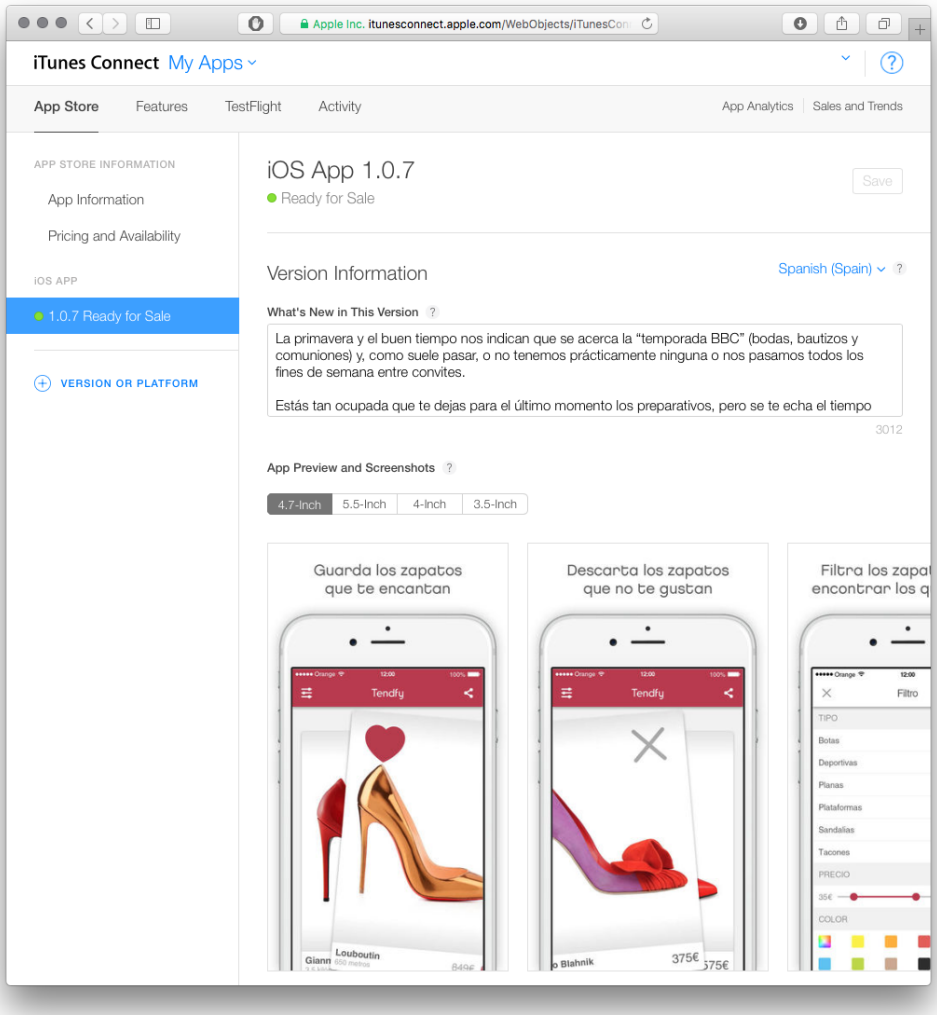


Imagen 92. Panel de administración de la App

5.4 Primeros resultados

La aplicación fue publicada en la AppStore en su primera versión en el mes de marzo de 2016. Desde entonces, se han subido distintas actualizaciones, con correción de fallos o adición de nuevas funcionalidades. Actualmente se encuentra en la versión 1.0.7, a esperas de publicar la versión que he desarrollado durante esta memoria.

5.5 Número de descargas

Durante el primer mes se consiguió alcanzar 127 descargas repartidas entre España, México y Estados Unidos. Durante los siguientes meses, las descargas se estabilizaron en una media de 175 al mes.

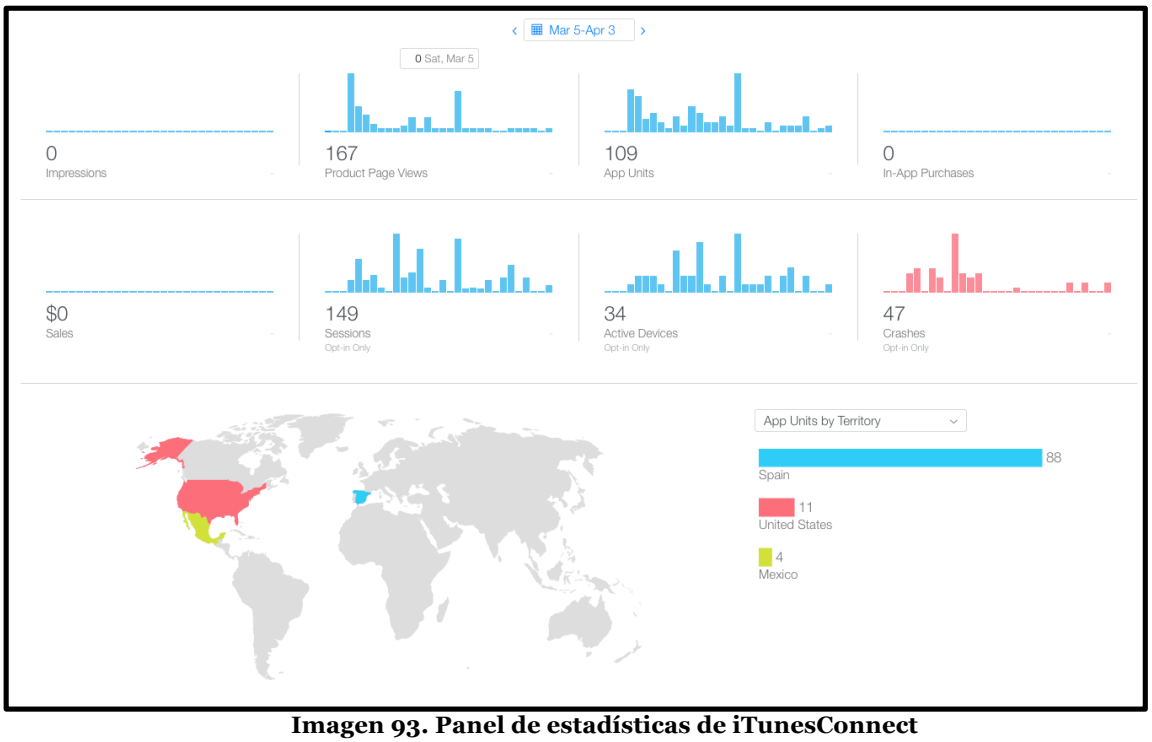


Imagen 93. Panel de estadísticas de iTunesConnect

5.6 Usuarios Activos

Actualmente la aplicación tiene unos 40-60 usuarios diarios según las estadísticas que nos ofrece el servidor Parse.

Pese a no haber gastado dinero en publicidad y dar a conocer mediante las redes sociales y el boca a boca, el resultado no ha sido nada malo.

6. Conclusión

Llegamos al final de la memoria del proyecto, en el que hablaremos de las conclusiones que hemos sacado tras finalizar la aplicación y de todo el proceso que ha conllevado.

6.1 Resultado del proyecto

Tras varios meses de trabajo, el objetivo principal era conseguir desarrollar una aplicación útil y atractiva para el usuario. Esto ha sido complicado pero se ha podido conseguir una primera versión de la *App* muy bien hecha, incluyendo características que inicialmente no estaban planificadas.

El código en un pincipio alcanzó una longitud aproximada de 12.000 líneas de código, eso no quiere decir que el proyecto fuera más complejo, ya que después de su refactorización se consiguió ahorrar hasta 4.000 líneas de código.

Una de las mayores motivaciones después de completar el proyecto era ver como se iban obteniendo un número mayor de descargas. Con el análisis de estas métricas, concluimos diciendo que el proyecto ha sido un éxito y que puede llegar ser mucho mayor.

6.2 Todo lo aprendido

Para poder llegar a completar el proyecto de que trata esta memoria he tenido que aprender muchísimo sobre las distintas tecnologías que intervinieron en el desarrollo de la aplicación. Mi experiencia como desarrollador de iOS, antes de trabajar en un proyecto real desde su principio hasta el final de la primera versión, ha sido una aventura agridulce en algunos momentos de esta, pero completamente satisfactoria una vez finalizada. He tenido que adquirir y expandir conocimientos que tenía para entender mejor el entorno iOS y el diseño de patrones.

La investigación sobre las tecnologías a utilizar para cada una de las partes del proyecto, me ha ayudado a tener más claro cómo funcionan otros entornos y tomar ideas para futuros proyectos propios.

Además de las tecnologías relacionadas directamente con el entorno iOS, aprendí a trabajar en equipos de desarrollo con entornos como bitbucket.

Podría decir que, además de un Proyecto Final de Grado, esto ha sido una gran experiencia para mi, que me resultará muy útil para mi futuro profesional. Actualmente estoy comenzando en otros proyectos relacionados con iOS y aplicando los conocimientos que explico en esta memoria, me doy cuenta de que ahora soy mejor desarrollador y me encuentro más a gusto con este entorno de desarrollo.

6.3 Opinión personal

Para finalizar esta memoria, propondré mi valoración personal del proyecto.

Mi valoración en general es muy positiva en todos los sentidos. Todo esto me ha ayudado a crecer como persona y como programador, lo que me enorgullece enórmemente.

Cuando Álvaro (CEO Tendfy) se puso en contacto conmigo para ayudarles con su StartUp, no me creía capaz de poder llevar su idea a algo material. Ahora después de haber completado todo lo que he contado con esta memoria, me doy cuenta de todo el camino que he recorrido a pasos agigantados. Tantas horas de duro trabajo, han asentando las bases de lo que he estado estudiando durante estos años en la carrera. Una gran experiencia que no podré olvidar.

Este proyecto me ha servido para poder darme cuenta del interés que despierta en mí el mundo de los dispositivos móviles, el cual siempre me ha gustado, pero desconocía que pudiera ser hasta este nivel. Quiero enfocar mi carrera profesional al diseño de software móvil.

7. Bibliografía

Patrones de Diseño. Erich GAMMA

Pro Design Patterns in Swift. Adam Freeman

Gang of Four. Erich GAMMA

http://www.blackwasp.co.uk/GangOfFour.aspx

Transparencias y contenidos de la asignatura de "Diseño de Software", de 3º curso del Grado de Ingeniería Informática de la Universidad Politécnica de Valencia.

Transparencias y contenidos de la asignatura de "Procesos de Software", de 4º curso del Grado de Ingeniería Informática de la Universidad Politécnica de Valencia.

iOS Parse Guide.

https://parse.com/docs/ios/guide

Import.io API reference.

http://api.docs.import.io/

iOS Developer Library.

https://developer.apple.com/library/ios/navigation/

StackOverFlow.

http://stackoverflow.com/

Swift Tutorials with RayWenderlich

https://www.raywenderlich.com/115253/swift-2-tutorial-a-quick-start