UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Comparativa Kotlin y Java en desarrollo Android

Android development comparison between Kotlin and Java

Realizado por
Francisco Sánchez Rueda

Tutorizado por
Luis Manuel Llopis Torres

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, NOVIEMBRE DE 2019

Fecha defensa: __ de diciembre de 2019

Fdo. El/la Secretario/a del Tribunal

UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA

D/Dª: Francisco Sánchez Rueda, con DNI 77182397D, estudiante del Grado/Máster en Ingeniería del Software, de la Universidad de Málaga.

DECLARO QUE:

El Trabajo Fin de Grado/Máster denominado: Comparativa Kotlin y Java en desarrollo Android

es de mi autoría, inédito (no ha sido difundido por ningún medio, incluyendo internet) y original (no es copia ni adaptación de otra), no habiendo sido presentado anteriormente por mí ni por ningún otro autor ni en parte ni en su totalidad. Así mismo, se ha desarrollado respetando los derechos intelectuales de terceros, para lo cual se han indicado las citas necesarias en las páginas donde se usan, y sus fuentes originales se han incorporado en la bibliografía. Igualmente se han respetado los derechos de propiedad industrial o intelectual que pudiesen afectar a cualquier institución o empresa.

Para que así conste, firmo la presente declaración en Málaga, a 18 de Noviembre de 2019.

Fdo.: D/Dª Francisco Sánchez Rueda

Resumen

El objetivo principal del trabajo fin de grado es la realización de un análisis profundo del lenguaje Kotlin como alternativa a Java para el desarrollo de aplicaciones móviles. Se realizará un estudio comparativo de ambos en aquellas cuestiones de programación que los diferencia. Como ejemplo de utilización del lenguaje Kotlin, se ha desarrollado una aplicación móvil basada en realidad aumentada.

La aplicación tendrá como propósito encontrar las ofertas de las tiendas que tenemos a nuestro alrededor mediante geolocalización, sensores de movimiento y realidad aumentada. La aplicación, si reconoce una tienda en la imagen mostrada mediante las técnicas anteriormente mencionadas, tratará de buscar ofertas exclusivas de dicha tienda y si las hay las mostrará en pantalla para que así el usuario sepa seleccionar a qué tiendas ir si va escaso de tiempo y poder incluso comprarlas navegando al enlace de la web oficial de la tienda facilitado. A parte de tener esta funcionalidad, la aplicación dispondrá un panel de administración web para propietarios en el que se podrá añadir ofertas a unas tiendas en específico creadas con anterioridad en relación a la marca del propietario. Adicionalmente a esto, el usuario tendrá una lista de sus tiendas y ofertas favoritas para poder así visitarlas con una mayor accesibilidad.

Palabras clave: ofertas, realidad aumentada, geolocalización, Android, Kotlin, Java

Abstract

The main purpose of this project is to develop a document where we could consider differences, advantages and disadvantages on the main subjects associated to the most prevailing Android programming languages: Kotlin and Java; including examples, coming to a conclusion and conducting a Kotlin's application about augmented reality.

The application's intent is to spot the offers of the shops around us using geolocation, motion sensors and augmented reality. If a shop is recognized, the application will try to find sole sales and if so, it will show them on the display. This could be useful if we have little time and will allow us to buy them using the link provided by the shop. Besides this functionality, the application will have a web management dashboard where the owner could add the offers. In addition, the user will have a list with his favorites offers and shops, giving him a better accessibility.

Keywords: sales, augmented reality, geolocation, Android, Kotlin, Java

Índice

1

Introducción

1.1 Motivación

Hace años, prácticamente todos los proyectos nativos Android estaban desarrollados en lenguaje Java y podíamos generalizar un poco diciendo que el número de desarrollos en Java era muy elevado. En el ámbito general, Java ha sido sustituido actualmente por muchos de los lenguajes emergentes que mejoran en situaciones específicas por bastante a este lenguaje orientado a objetos; podemos destacar Python y JavaScript, entre otros. Ya hablando del ámbito del desarrollo nativo Android, en los últimos años ha habido un declive de Java a favor de un auge por parte de un nuevo lenguaje emergente de JetBrains y apoyado ampliamente por Google para este propósito: Kotlin. Aun siendo Java un lenguaje maduro y con bastante versatilidad para ser utilizado en múltiples situaciones, hoy en día queda por detrás en algunos o bastante aspectos frente a este nuevo competidor.

El documento de comparación de este TFG nace de la necesidad de muchos desarrolladores Android que necesitan un cambio fresco hacia un lenguaje menos verboso, más seguro en cuanto a lanzamiento de excepciones y muchas cosas más que veremos en este documento mencionado. Pero, sobre todo, y es lo que va a ser clave en su crecimiento, que permita la interoperabilidad con su antiguo compañero: Java.

La aplicación en este caso sirve a todas aquellas personas que carecen de tiempo cuando van de compras y no les sirve el comercio on-line textil, ya sea por dificultades de encontrar la talla correcta, observación de diferencias entre las imágenes expuestas del producto en un modelo y el producto real en el cuerpo propio o cualquier otro motivo relacionado con la compra en línea y sus procesos. Para todos ellos será más fácil conocer las ofertas de las tiendas a su alrededor y así deambular por ellas con una mejor idea y de forma más eficiente.

1.2 Objetivos

En lo relativo al documento, los objetivos principales son los de: dar a conocer el nuevo lenguaje Kotlin como propósito general, aunque centrándonos un poco en el propósito de desarrollo nativo en Android. Esto se hace viendo dónde se sitúa en la actualidad realizando una comparativa general y realizando algunas comparaciones con lenguajes que estén en auge, como ya hemos dicho, en cualquier propósito ya que como veremos Kotlin nace como un posible lenguaje multiparadigma. Posterior a esto, veremos cuáles son los puntos fuertes y débiles de este lenguaje realizando una comparación exhaustiva con Java. Los puntos documentados tocan palos de todo tipo intentando abarcar conceptos generales de programación como son el tratamiento de datos nulos, tipado de datos y su manejo, programación asíncrona, así como otros temas más específicos relacionados con la programación Android o, sin ir más lejos, otros apartados en los que se ha visto diferencias entre ambos lenguajes o en los que se podrían destacar datos importantes.

Por otro lado, hablando de la aplicación desarrollada, los objetivos son los de realizar una aplicación nativa en lenguaje Kotlin que sirva como ejemplo para ver las ventajas y beneficios tanto del desarrollo nativo como de la elección del lenguaje. El objetivo que se quiere alcanzar con esta aplicación es la facilitación para el usuario en las compras. Podrá acceder fácilmente y de una forma interactiva (realidad aumentada) a las ofertas de las tiendas que le rodean, haciendo así la compra mucho más eficiente en cuanto a tiempo y divertida. Esta aplicación beneficiará también a las empresas y propietarios de estas, que verán que sus tiendas y ofertas aparecen en la aplicación y son visitadas con redirecciones a sus propias páginas para que ellos tengan la información y la gestión de la compra final del producto.

1.3 Estructura de la memoria

Al estar este trabajo de fin de grado dividido en dos grandes partes: documento teórico de comparación y desarrollo de la aplicación de ofertas, la estructura del mismo también quedará dividida en dos grandes partes:

Documento teórico, comparación Kotlin y Java: En este apartado introduciremos y pondremos un poco en contexto el documento de comparativa entre ambos lenguajes y los motivos por los que se ha llevado a cabo, así como el porqué de los puntos tratados en el escrito. Posterior a eso se expondrá todo el contenido teórico.

Desarrollo de la aplicación: Veremos explicados brevemente los motivos de la aplicación y una antesala para los subapartados siguientes que vendrán a rellenar la parte restante de la memoria.

- **Metodologías y fases del trabajo:** Veremos qué metodologías y procedimientos se han seguido tanto para la preparación como para el desarrollo del proyecto, incluyendo en este término tanto el documento de comparación como la aplicación, y el motivo de cada uno de ellos. También destacaremos las fases de trabajo para realizar cada una de las partes.
- **Especificación de requisitos, casos de uso y flujo del sistema:** En este subapartado concretaremos todos los requisitos del sistema, tanto funcionales como los no-funcionales derivados de los anteriores. Detallaremos descripción, dependencias, prioridades y algún comentario de forma individual de los presentes en esta iteración y nombraremos los que se quedan para futuras actualizaciones.
  De estos requisitos funcionales surgirán los casos de uso del sistema, que asociaremos a cada actor y veremos cuáles son las dependencias entre cada uno de ellos.
  Después de todo esto veremos cuál sería el flujo que debería de tener el usuario en nuestra aplicación móvil con una secuenciación de acciones del sistema.
- **Diseño y modelado de datos:** Se explicará cómo se ha realizado el modelo de los datos y se desarrollará el porqué de cada una de las instancias y propiedades del diseño.
- **Elección e implementación de tecnologías:** En este apartado veremos cuáles han sido las tecnologías elegidas, sus motivos y cómo se han implementado. Analizaremos tanto las tecnologías utilizadas para la aplicación móvil como las aplicadas en el panel de administración y el servidor API.
- **Desarrollo de la aplicación por requisitos:** Comprobaremos cómo se han desarrollado cada uno de los requisitos en la aplicación, si se han realizado finalmente o no y cuál ha sido la experiencia al implementarlos.
- **Actualizaciones futuras:** Veremos cuáles podrían ser las actualizaciones futuras de la aplicación y el motivo de ellas intentando localizar cuáles serían más imprescindibles y se harían con la mayor brevedad posible y cuáles, en contraposición, podrían llevarse a cabo en un futuro más lejano.
- **Conclusión:** Analizaremos cuáles han sido los valores y conocimientos aportados desde el principio hasta el final de la realización de este proyecto, así como las experiencias que se han tenido al realizar todos los procesos.

2

Comparación Kotlin y Java

2.1 Introducción

El principal motivo de este trabajo fin de carrera era ver cómo estaba la situación en la programación Android y evaluar la incidencia de la entrada de Kotlin en el panorama de la programación nativa. Para ello se decidió realizar un documento teórico donde poder realizar un análisis de este lenguaje. Esto derivó en un documento de comparación de Kotlin y Java en muchos de los aspectos destacables de la programación donde se intentaría, de forma imparcial, analizar las diferencias en muchos de los aspectos programáticos como la programación asíncrona, tratamiento de objetos nulos, casteos y tipados, etc. Los temas han ido cambiando y disminuyendo debido a la gran multitud de temas que se pusieron encima de la mesa desde un principio, aunque al final se ha intentado comprimir el contenido sin dejar de lado información importante.

Además, el documento contiene una introducción al lenguaje Kotlin situando contextualmente su entrada al desarrollo y se analizan otros puntos interesantes de la programación nativa como concepto, como puede ser el tema de *Aplicaciones nativas vs Aplicaciones híbridas* o *Kotlin como lenguaje de propósito general.*

Todas las comparaciones con su compañero Java, van adjuntadas con ejemplos a modo de capturas de código para lograr un mayor entendimiento y verificación del caso teórico.

Por último, decir que este documento puede considerarse la pieza principal del proyecto habiéndose llevado a cabo un extenso trabajo de investigación y conglomeración de conceptos. Aun así, el trabajo más ingenieril viene del lado de la aplicación de ofertas, la cual introduciremos en el siguiente apartado.

Todos hemos sido conscientes del auge de la telefonía móvil en las últimas décadas y, en particular, de los smartphones en estos últimos años. Actualmente una inmensa mayoría de las personas en el mundo poseen un teléfono móvil y una gran parte de ellos tiene un smartphone.

A continuación, vemos una gráfica (1) de cómo han aumentado las inscripciones (compras) de telefonía móvil en los últimos 40 años y de cómo continuarán en años venideros:

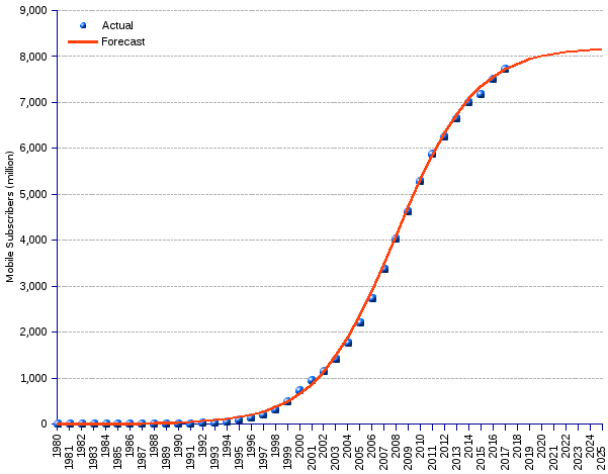

Figura 1: Número de usuarios con móvil por año

Este incremento ha producido cambios de todo tipo en el campo de la tecnología, pero el que nos atañe principalmente es el asociado al uso de smartphones para navegar por Internet o el fenómeno de hace pocos años: las aplicaciones web.

¿Y dónde vamos con todo esto?

La incorporación continua de nuevas funcionalidades para los smartphones y de facilidades para el usuario de estos, ha hecho que el principal uso de Internet (ya sea en forma de web o de aplicación) provenga de los teléfonos móviles.

Esto cambió radicalmente la forma de desarrollar las webs hace unos años, teniendo que adaptar el diseño y funcionalidad de estas al tamaño y uso de móviles lo que posteriormente derivaría en las aplicaciones web actuales, como hemos dicho anteriormente.

Aquí vamos a ver un gráfico (2) que demuestra cómo ha crecido el número de páginas web servidas y adaptadas a los smartphones en la última década:

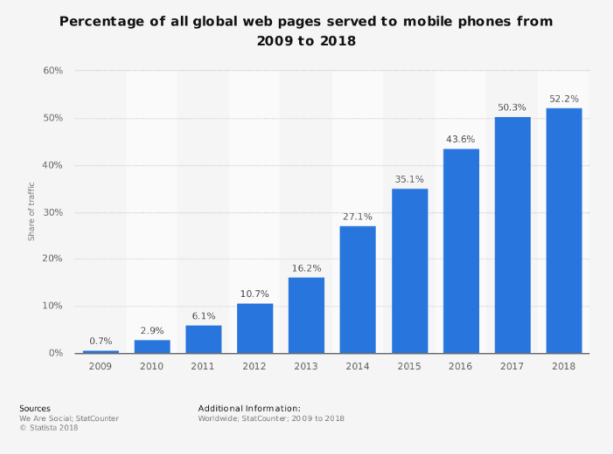

Figura 2: Porcentaje global de páginas web visitadas en móvil

Nuestro principal punto se centra en la aparición de las aplicaciones móviles (que no web). Contenido especializado para teléfonos móviles producido para ser publicado en un mercado, principalmente asociado al sistema operativo del móvil (Play Store o App Store), y que mejora la usabilidad, el rendimiento y la accesibilidad en móviles respecto a las páginas web.

En este marco llega el desarrollo específico para estas aplicaciones móviles y más específico para cada sistema operativo: Java para Android y Objective-C para iOS.

Dejando a un lado iOS y metiéndonos un poco más en materia, la mayoría de las aplicaciones de Android están desarrolladas y programadas en Java. La estructura del sistema operativo Android se compone de estas aplicaciones que se ejecutan en un marco de trabajo (framework) Java sobre un núcleo de bibliotecas Java.

Android desde su versión 5.0, es capaz de interpretar y ejecutar directamente instrucciones expresadas en código binario Java (Java bytecode), y digo desde la versión 5.0 porque hasta ese entonces Android utilizaba una máquina virtual Dalvik con compilación en tiempo de ejecución, esto cambia a partir de la versión 5.0 en la

que Android introduce el entorno Android Runtime, con el que compila el bytecode de Java en el proceso de instalación de la aplicación.

Es por esto último que el lenguaje más utilizado para la programación de aplicaciones Android es Java, el más cercano a su entorno y máquina virtual y es también por esto, que en 2011 surge un nuevo lenguaje intentando solucionar algunos de los problemas de Java e intentando igualar las ventajas que este lenguaje tiene en el desarrollo nativo Android: **Kotlin**.

2.2 Comparativa general

JetBrains decide crear un nuevo lenguaje que se ejecute sobre la máquina virtual de Java anteriormente nombrada y decide crearlo con el objetivo de que compile tan deprisa como el propio Java en su máquina virtual.

El líder de JetBrains en el año 2011 comentó que ningún lenguaje tenía las características que él buscaba (3) y creó Kotlin basándose en algunos pilares clave como los son (4):

- **Concisión** frente a la verbosidad presente en un lenguaje con años como es Java. Un ejemplo claro es la creación de clases con los métodos habituales get, set, equals, hashcode y copy en una sola línea con la sentencia "data class".
- **Seguro** intentando evitar errores en tiempo de compilación con su notada característica Null Safe, gestionando los valores nulos de manera segura garantizando así la no aparición de NullPointerException (NPE) de Java. La única forma de obtener NPE en Kotlin es: llamando la excepción explícitamente, usando el operador !! (se verá más adelante), inconsistencia de datos o con la interoperabilidad con Java, otro pilar clave.
- **Interoperabilidad**, sin duda la más importante**,** la facilidad de transformación de proyectos Java a Kotlin y su posible convivencia en una misma aplicación. No tienen una sintaxis compatible, pero Kotlin sí está diseñado para interoperar con Java. Kotlin además depende de la biblioteca de clases de Java.

Por esto y mucho más que veremos más adelante, muchas aplicaciones son ahora desarrolladas en este nuevo lenguaje más que emergente y, además, son muchas las empresas que deciden trasladar su código a Kotlin por sus novedades y por su fácil traslado e interoperabilidad, como hemos dicho antes.

En la siguiente gráfica (5) vemos cómo ha aumentado el número de desarrolladores de Kotlin en los últimos dos años frente a los desarrolladores Java, todo esto refiriéndonos a aplicaciones Android y no como lenguaje de propósito general (tema a tratar más adelante):

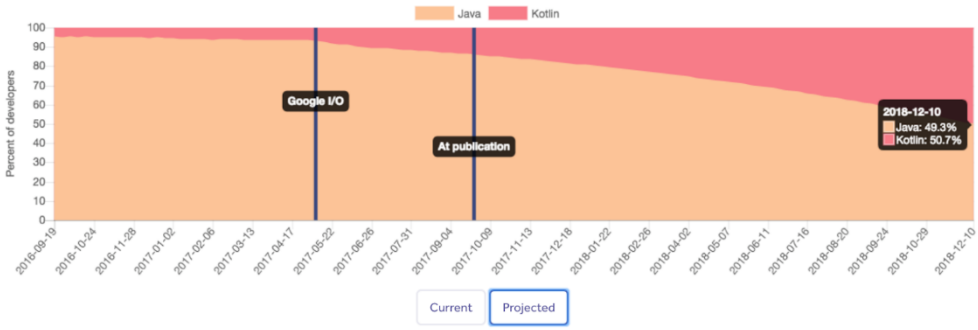

Figura 3: Porcentaje de desarrolladores que programan en un lenguaje u otro por año

Otra gráfica (6) que demuestra el auge de este lenguaje se representa en la siguiente gráfica referente a la actividad de Kotlin en Github, StackOverflow y el uso de plugins programados en este lenguaje:

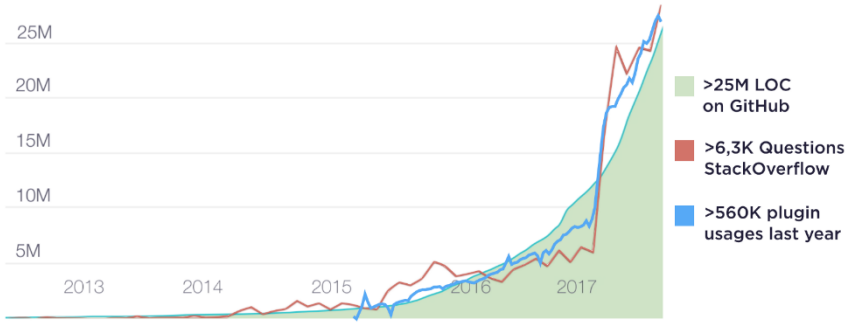

Figura 4: Hilos en GitHub, preguntas en StackOverflow y uso de plugins el último año

2.3 Aplicaciones híbridas

En el mismo marco en el que nos situamos, pero con un recorrido paralelo al desarrollo nativo, se encuentran las aplicaciones híbridas. Hablamos de desarrollo web adaptado a móvil en este caso, se utilizan lenguajes que nacieron con orientación a la web como son HTML, CSS y Javascript, estamos hablando de que estas aplicaciones se ejecutan en webView: las vistas muestran una página web usando el mismo motor que un navegador. Algunos frameworks conocidos son Cordova, Ionic y Xamarin entre otros. Aun así, gracias a algunos frameworks, desde hace poco, existentes son capaces de acceder hasta a funcionalidades nativas de nuestro teléfono móviles, es el caso de React Native o Ionic Native. Al fin y al cabo, estas aplicaciones híbridas acaban accediendo a plugins nativos que están una capa por encima en abstracción respecto a las librerías o componentes nativos de cada sistema operativo y pueden llegar incluso a tener el mismo rendimiento y velocidad al acceder a estas funcionalidades que las propias aplicaciones nativas.

Estas aplicaciones pueden llegar a tener muchas ventajas, destacando la principal (7): un solo desarrollo y un solo lenguaje para todo tipo de dispositivos móviles y cualquier sistema operativo, ya sea iOS, Android o Windows Phone en su momento, lo que se conoce como “app multiplataforma”. Esto es posible ya que realmente lo que se está construyendo es una web, una aplicación web, solo que posteriormente es compilada tanto para Android como iOS de forma independiente… O no.

Otra opción que está emergiendo actualmente es el término **PWA**: aplicaciones web que asemejan su comportamiento al de una aplicación móvil mediante el uso de Service Workers que permiten seguir ejecutando la aplicación en segundo plano y pueden ser instaladas en el móvil desde el navegador como si fueran una aplicación del marketplace. Algunas de las funcionalidades destacadas pueden ser la posibilidad de lanzar notificaciones push, funcionamiento sin conexión (aun siendo, como hemos dicho, una aplicación web), menor peso que una aplicación nativa o híbrida proviniendo del store de nuestro sistema operativo y, sobre todo, no tener que compilar la aplicación para ningún sistema operativo: es una web, sin más.

En principio parecen ventajas de lo más significativas, la duda viene cuando se mide el rendimiento de estas aplicaciones híbridas en juegos, ya sean 3D o HD, animaciones, en todo lo referente a aplicaciones con gran requerimiento de gráficos y, sobre todo, donde todo esto pueda unirse para aunar un proyecto, como puede ser en el campo de la **realidad aumentada**, donde son necesarios multitud de accesos a hardware y plugins nativos, así como gráficos: cámara, geolocalización, giroscopio, acelerómetro, sensor de movimiento, modelado 3D… Esto unido a que existen algunas características que no pueden ser accesibles mediante estos plugins nativos comentados y que requieren de implementación propia desencadenando en mayor complejidad de desarrollo hacen que **el desarrollo nativo decaiga en aplicaciones más simples, pero no en otras más complejas** con los rasgos descritos.

2.4 Kotlin como lenguaje de propósito general

Como último apartado en esta introducción, me gustaría destacar a Kotlin como lenguaje de propósito general. Ya hemos hablado un poco de la utilidad de Kotlin en el desarrollo nativo Android, pero cabe señalar algunas características de Kotlin fuera de ese ámbito.

En el ámbito desde desarrollo web, Kotlin tiene un amplio abanico de posibilidades. Desde desarrollar servidores con **Sprint**, que te permite aplicar todas las ventajas de Kotlin respecto a Java y sin ningún problema de compatibilidad con las tecnologías Java existentes, hasta el desarrollo tanto de parte de cliente como de servidor con el framework Ktor aplicando el uso asincronía y co-rutinas para una mayor escalabilidad.

También podemos destacar que al igual que Kotlin corre sobre la máquina virtual de Java con prácticamente la misma eficiencia que el lenguaje del que toma el nombre, Kotlin también puede ser compilado a código fuente de **Javascript** proveyendo optimización, código de fácil lectura e interoperabilidad con los módulos de Javascript. Es por esto que también es posible desarrollar aplicaciones web e incluso permite el desarrollo multiplataforma, aplicando los conceptos explicados más arriba sobre aplicaciones híbridas. Puede ser usado tanto de lado de cliente como de lado de servidor, interactuando con elementos del DOM (Document Object Model) y con elementos gráficos en las páginas web, incluso creándolos; así como interactuar con el lado del servidor como NodeJS.

En adición a todo esto, existe **Kotlin/Native** una tecnología que permite el desarrollo en cualquier tipo de dispositivo, el código programado puede ser compilado a código binario siendo posible ejecutarse sin máquina virtual (como así lo hace en Android ejecutándose en la JVM). Esto aporta a Kotlin de

interoperabilidad tanto para que el compilador cree como para que tenga compatibilidad con las librerías de las diferentes plataformas existentes: WebAssembly, Linux (MIPS, Raspberry Pi…), Windows, Android evidentemente y sistemas Apple, tanto para proyectos Swift como en Objective-C. Así podemos decir que se pueden programar sistemas empotrados, webs, aplicaciones y no se excluye a iOS ni macOS del proceso. Sí, podemos programar para el sistema operativo de la manzana con Kotlin gracias al Apple Framework de Kotlin.

Por último destacar que Kotlin permite la compartición de código entre todas las plataformas para las que se puede programar tanto por parte del propio lenguaje Kotlin como por parte de Kotlin/Native; así tenemos un enlace entre las distintas plataformas para las que puede servir Kotlin: Android y JVM, aplicaciones web (servidor y cliente: Javascript, CSS, HTML…), iOS y MacOS y el lado nativo (4).

Analizaremos, y de ello va este texto, las diferencias, coincidencias, ventajas y desventajas de estos dos lenguajes de programación en el resto del documento.

2.5 Null safety

Para empezar, podemos definir qué es algo nulo en cuanto a programación se refiere: "null" es algo que no hace referencia a ningún sitio, literalmente, no apunta a nada. Esto es bastante normal verlo en programación como también es normal intentar operar con este tipo de datos. No ocurre nada cuando algo adquiere este valor, pero sí cuando intentamos acceder a él o a parte de él.

En un lenguaje con gestión manual de memoria y aritmética de punteros como puede ser C o C++, el acceder a una propiedad de un objeto nulo puede ser un problema más de los muchos que nos pueden aparecer a la hora de tener que gestionar las posiciones de memoria y el acceso a punteros como hemos dicho antes.

Java fue diseñado en principio para liberar al programador del "código defensivo": comprobaciones continuas sobre el valor de objetos o propiedades a las cuales queremos acceder. A gran escala lo consigue, porque nos libra de todo lo mencionado antes presente en lenguajes como C (los strings hacen que no nos tengamos que preocupar por buffers, la máquina virtual de Java intenta liberar objetos no utilizados...), pero "null" sigue existiendo y eso hace que tengamos que seguir haciendo continuas comprobaciones (8) (9).

El error lanzado por Java en particular cuando esto ocurre es el conocido Null Pointer Exception y puede ocurrir por acceder a un método de una instancia no creada o de una instancia de un objeto nulo, acceder o modificar un campo de un objeto nulo…

El problema de esto es que esta excepción es lanzada en tiempo de ejecución y no de compilación, esto hace que no sepamos en qué momento puede producirse este acceso nulo hasta que el programa llegue a ejecutarse.

Consciente de esto, Kotlin inventa un sistema de tipado capaz de eliminar el peligro que supone el acceso a referencias nulas, traducido a Java: evitar el Null Pointer Exception. Esta excepción solo es posible obtenerla en Kotlin por las siguientes causas (4):

- La llamada explícita a la propia excepción: *throw NullPointerException();*
- Uso del operador `!!` que veremos a continuación.
- Inconsistencia en los datos y por tanto en la inicialización de los mismos.
- Interoperabilidad con Java

KOTLIN

Para la explicación se adjuntarán ejemplos básicos para el correcto entendimiento de la teoría explicada, estos ejemplos son escalables y reflejan lo que puede ocurrir en cualquier línea de cualquier programa por complejo que sea.

En primer lugar, conocer que en Kotlin existen las referencias que soportan valores nulos y las que no. La declaración de un objeto o propiedad de la forma ordinaria en un lenguaje orientado a objetos como es Java, en Kotlin se traduce en una variable NO NULA y cuando se accede a ella y esta adopta el valor null por alguna razón, el compilador de Kotlin nos lanza un error de que esta variable no puede adoptar el valor nulo, y esto es importante: el compilador nos lanza el error **sin necesidad de que hayamos accedido a ninguna propiedad de esta variable nula** (4)**.**

NON-NULL TYPE

```
fun main() {
    var a: String = "Hola";
    /* Some code */
    a = null; //By any reason
    /* Some code */
    println(a.length);
}
```

Figura 5

```
NullSafety.kt:4:9: error: null can not be a value of a non-null type String
    a = null; //By any reason
        ^
```

Figura 6

Como vemos en la imagen que representa el error, este se lanza en la línea 4, justo donde le damos el valor null a nuestra variable, no le importa si accedemos o no a esta variable posteriormente y nos dice eso: la variable que hemos declarado de la forma NON-NULL no puede adoptar dicho valor por definición.

Esto es bastante potente cuando queremos que en un sistema un objeto, propiedad o variable no queremos que adopte el valor null de ninguna forma.

Sin embargo, es posible que queramos esa versatilidad de poder tener un valor null en una propiedad, para esto existe el tipo de referencia que posibilita un valor nulo (nullable reference). En este caso, se podrá asignar el valor null sin problemas tal y como ocurre en otros lenguajes. A diferencia de otros lenguajes, Kotlin avisará en

tiempo de compilación sobre el acceso a este tipo de propiedades cuando adopten el valor nulo.

NULLABLE TYPE

```
fun main() {
    var a: String? = "Hola";
    /* Some code */
    a = null; //By any reason
    /* Some code */
    println(a.length);
}
```

Figura 7

```
NullSafety.kt:6:14: error: only safe (?.) or non-null asserted (!!.) calls are allowed
 on a nullable receiver of type String?
    println(a.length);
            ^
```

Figura 8

Vemos en la imagen que ahora el error ha saltado en la línea 6 que es cuando accedemos a la propiedad *length* siendo la variable *a* nula y nos dice que no es posible acceder a la propiedad del objeto nulo con el ".", es necesario hacer accesos seguros como "?." Y "!!.".

Ahora la cuestión es cómo evitar este error de compilación cuando queremos declarar una variable que pueda ser nula y por algún motivo consiga serlo:

- **Condiciones**: al igual que en Java, en Kotlin podemos evitar el acceso nulo mediante condiciones. Este método resulta tedioso y provoca un código denso donde predominan las sentencias if/else con asiduidad. Con Java este método de comprobación es inevitable pero no en Kotlin, donde tenemos más formas simples y menos verbosas de chequeo.

```
/* Conditions */
if(a != null) println(a.length);
else println(-1);
/* It will print -1 */
```

Figura 9

- **Llamada segura**: sin duda la más potente y utilizada dentro del lenguaje, esta llamada nos permite poder realizar, sin temor alguno, accesos a objetos o propiedades posiblemente nulas. En el momento en el que el objeto al que pertenece la propiedad a buscar sea nulo y queramos acceder, Kotlin inmediatamente evalúa la sentencia a null en lugar de intentar acceder a nuestro objeto nulo.

```
/* Safe access */
println(a?.length);
/* It will print null because the property "a" is null */
```

Figura 10

En el ejemplo podemos ver que, aunque nuestra variable “a” sea nula, se imprimirá por pantalla “null” por acceder de manera segura y no hará el acceso a la propiedad “length” especificado.

- **Operador Elvis**: nombrado así por su parecido al mirar el operador de derecha a izquierda -> `?:`, es una mezcla entre las comprobaciones anteriormente nombradas. En el caso de la llamada segura, se devolverá “null” al acceder al objeto o propiedad nula sí o sí. Pues con este operador controlamos el valor de esa devolución: lo usaremos puntualmente cuando queramos devolver un valor en específico y su sintaxis es la siguiente:

```
/* Elvis operator, remix of Conditions and Safe access */
println(a?.length ?: -1);
/* It will print -1 because the property "a" is null */
```

Figura 11

Vemos como se realiza una llamada segura, el operador Elvis y el valor que queremos devolver: en este caso -1.

- **Operador `!!`:** aquí viene la excepción nombrada en el comienzo del apartado. Acceder a una propiedad de esta forma lanzará la excepción NullPointer cuando este sea nulo, se podría decir que simula el acceso ordinario en Java. El por qué es simple, este operador que convierte cualquier valor en no-nulo: sí, podemos declarar la variable como posiblemente nula, que obtenga dicho valor y posteriormente escribir, en caso de que la variable se llame “b”, `b!!` para convertir “b” en no-null, por lo que al transformar el tipado de esta y haber obtenido el valor “null” anteriormente, ¿qué nos queda? NullPointerException.

```
/* !! operator */
println(a!!.length);
/* It will throw NullPointerException because it's the same
    Java access */
```

Figura 12

Existe en Kotlin la excepción NullPointer, pero hay que buscarla explícitamente y esta se lanza en tiempo de ejecución tal y como ocurre en Java.

```
Exception in thread "main" kotlin.KotlinNullPointerException
        at NullSafetyKt.main(NullSafety.kt:21)
        at NullSafetyKt.main(NullSafety.kt)
```

Figura 13

EJEMPLO COMPLETO

```
fun main() {
    var a: String? = "Hola";
    /* Some code */
    a = null; //By any reason
    /* Some code */

    /* Conditions */
    if(a != null) println(a.length);
    else println(-1);
    /* It will print -1 */

    /* Safe access */
    println(a?.length);
    /* It will print null because the property "a" is null */

    /* Elvis operator, remix of Conditions and Safe access */
    println(a?.length ?: -1);
    /* It will print -1 because the property "a" is null */

    /* !! operator */
    println(a!!.length);
    /* It will throw NullPointerException because it's the same
        Java access */
}
```

Figura 14

Los dos tipos de datos nombrados anteriormente son igualmente aplicables a colecciones de datos (arrays). Podemos tener una colección de datos nulables y otra colección de datos no nulables, las cuales se declararán de la misma forma que declaramos una variable de tipo primitivo o cualquier otra:

```
val listaNulable: List<String?> = listOf("hola", null);
val listaNoNulable: List<String> = listOf("hola", null);
```

Figura 15

El comportamiento es el mismo que para los valores individuales, evidentemente se lanzará un error en tiempo de compilación sobre la inicialización de la lista "listaNoNulable" al contener un valor de tipo String nulo.

```
NullSafety.kt:5:40: error: type inference failed. Expected type mismatch: inferred type is
 List<String?> but List<String> was expected
    val listaNoNulable: List<String> = listOf("hola", null);
                                       ^
```

Figura 16

Más tarde en el apartado de Casteos, comprobaremos cómo transformar variables nulables a no-nulables y viceversa, analizando con detenimiento la potencia y utilidad de ello.

A pesar de todas estas buenas noticias, existe un inconveniente en los tipos no-nulables que viene por definición para el cual el lenguaje ofrece también una solución.

Las propiedades no-nulables deben ser inicializadas ya sea de forma global o en el constructor. Muchas veces esto no es posible o no nos conviene ya que puede haber variables que queramos inicializar por inyección de dependencias, en una función de inicialización aparte, etc. Para esto podemos marcar una propiedad con el modificador "**lateinit**", lo cual nos permitirá inicializar la variable posteriormente. Si accedemos a esta propiedad antes de su inicialización posterior, se lanzará una excepción característica que identificará la propiedad a la que se ha accedido y el por qué no se ha inicializado correctamente.

```
class SalesActivity : AppCompatActivity() {

    var salesList: ListView
            Property must be initialized or be abstract
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_sales)

        salesList = findViewById(R.id.listView)
    }
```

Figura 17

Nos encontramos en un ejemplo de programación Android: queremos inicializar la lista en el método que se ejecuta al crear la actividad en cuestión ya que es ahí donde tenemos acceso a los elementos de la interfaz y no antes. Si no declaramos la variable con el modificador "lateinit" el compilador nos lanza el error de inicialización de variable no-nulable.

```
lateinit var salesList: ListView

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_sales)

    salesList.itemsCanFocus
    salesList = findViewById(R.id.listView)
}
```

Figura 18

```
Caused by: kotlin.UninitializedPropertyAccessException: lateinit property salesList has not been initialized
    at com.example.fsanchez.salesapp.SalesActivity.onCreate(SalesActivity.kt:22)
```

Figura 19

Aquí hemos declarado la variable con el modificador "lateinit" y accedido a una de sus propiedades previamente a su inicialización. Como hemos comentado, se lanza una excepción característica en tiempo de ejecución especificando la variable causante.

JAVA

En Java, como hemos comentado anteriormente, no queda otra que realizar continuas comprobaciones con las sentencias if/else para no llegar a la excepción NullPointer, esta es una de las cosas que hace a Java un lenguaje verboso y denso. Vemos un ejemplo:

```java
public class NullSafety {
    Run | Debug
    public static void main(String[] args) {
        String a = "Hola";
        /* Some code */
        a = null; //By any reason
        /* Some code */
        System.out.println(a.length());
    }
}
```

Figura 20

Evidentemente, este trozo de código lanzará en tiempo de ejecución la excepción NullPointer al intentar acceder a la propiedad "length" sin dar ningún previo aviso en la compilación.

```
Exception in thread "main"
java.lang.NullPointerException
        at NullSafety.main(NullSafety.java:7)
```

Figura 21

La solución a esto parte de lo ya comentado anteriormente: condiciones.

```java
public class NullSafety {
    public static void main(String[] args) {
        String a = "Hola";
        /* Some code */
        a = null; //By any reason
        /* Some code */
        if(a != null) {
            System.out.println(a.length());
        }
    }
}
```

Figura 22

2.6 Casteo y comprobación de tipos

En primer lugar: ¿qué es el casteo? Castear es una forma de informar al compilador explícitamente que pretendes cambiar el tipado de un dato o, también, realizar una conversión de un tipo a otro teniendo en cuenta las consecuencias que acarree (4).

¿Cómo se castea de la forma más simple posible en los dos lenguajes que analizamos? Para ello vamos a tomar el mismo ejemplo, queremos realizar un típico casteo de tipo cadena de caracteres a tipo Object (clase padre) y viceversa, en caso de Kotlin no sería 'Object' si no 'Any'.

KOTLIN

```kotlin
fun main() {
    var s: String = "hola";
    var res: Any = s as Any; //CASTEO DE STRING A OBJETO(ANY) (UPCAST)
    println("Any " + res);
    var o: Any = "adios";
    var res2: String = o as String; //CASTEO DE OBJETO(ANY) A STRING (DOWNCAST)
    println("String " + res2);
}
```

Figura 23

JAVA

```java
public class Cast {
    public static void main(String[] args) {
        String s = "hola";
        Object res = (Object) s; //CASTEO DE STRING A OBJETO (UPCAST)
        System.out.println("Objeto " + res);
        Object o = "adios";
        String res2 = (String) o; //CASTEO DE OBJETO A STRING (DOWNCAST)
        System.out.println("String " + res2);
    }
}
```

Figura 24

Podemos ver cómo el operador 'as' es el equivalente al casteo típico en Java:

KOTLIN: x as String => JAVA: (String) x

Estos dos casteos realizados tienen el nombre de:

- **Upcast**: se representa como el casteo desde un tipo más genérico, en este caso el tipo String, a otro menos genérico, en este caso el tipo Object o Any, es el menos peligroso ya que nunca se produce error al ser siempre posible por ir a una capa de abstracción mayor y ser un objeto padre en la jerarquía orientada a objetos el tipo del resultado del casteo.
- **Downcast**: es el casteo de un tipo menos genérico a otro más genérico, en este caso de tipo Object a tipo String, es de los dos el más peligroso y puede llevar a excepción (ClassCastException) como vamos a ver en los siguientes ejemplos:

  JAVA

```
Object o = "adios";
Integer res2 = (Integer) o; //CASTEO DE OBJETO A STRING (DOWNCAST)
System.out.println("Integer " + res2);
```

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
        at Cast.main(Cast.java:7)
```

Figura 25

KOTLIN

```
var o: Any = "adios";
var res2: Int = o as Int; //CASTEO DE OBJETO(ANY) A INT (DOWNCAST)
println("Int " + res2);
```

Figura 26

```
Exception in thread "main" java.lang.ClassCastException: java.lang.String cannot be cast to java.lang.Integer
        at CastKt.main(Cast.kt:6)
        at CastKt.main(Cast.kt)
```

Figura 27

Estamos intentando hacer un downcast de un objeto Object o Any inicializado con una cadena de caracteres a Integer o Int y como no es posible realizarlo, el resultado es el lanzamiento de la excepción en tiempo de ejecución.

Dejando a un lado la explicación del término y clasificación, podemos analizar y comparar el casteo de ambos lenguajes:

En cuanto al casteo típico en Java se puede considerar bastante potente ya que permite castear variables numéricas de diferentes tipos variando el resultado de este casteo:

```java
int i;
double d = 54.45;
i = (int) d;//CASTEO DE DECIMAL A ENTERO
System.out.println("Entero " + i);
d = (double) i;//CASTEO DE ENTERO A DECIMAL
System.out.println("Decimal " + d);
```

Figura 28

```
Entero 54
Decimal 54.0
```

Figura 29

Conociendo las consecuencias y los resultados de nuestros casteos, podemos hacer cosas realmente potentes con esto y muchas casteos más complejos. Esto puede llevar sin embargo a excepciones inesperadas de las cuales el compilador no avisa y que puede ser fruto de la confianza al realizar casteos en Java, un ejemplo puede ser el querer castear un objeto de tipo Date a String sin utilizar el método implementado toString() o querer obtener por cualquier motivo un tipo Integer del tipo Date anteriormente mencionado, y todo esto con su consecuente ClassCastException.

Kotlin, sin embargo, con el casteo básico (al que Kotlin llama 'unsafe cast') no actúa de la misma forma que Java en el ejemplo de Double a Integer, resultará en la excepción ClassCastException y dando un warning previo avisando de que el casteo nunca se producirá:

```
var daux: Double = 54.45;

var i: Int = daux as Int;
println("Entero " + i);
```

Figura 30

```
C:\Users\sanxe\Documents\TFGWorkspace\Casts\Kotlin>kotlinc Cast.kt -include-runtime -d Cast.jar
Cast.kt:5:23: warning: this cast can never succeed
    var i: Int = daux as Int;
                      ^

C:\Users\sanxe\Documents\TFGWorkspace\Casts\Kotlin>java -jar Cast.jar
Exception in thread "main" java.lang.ClassCastException: java.lang.Double cannot be cast to java.lang.Integer
        at CastKt.main(Cast.kt:5)
        at CastKt.main(Cast.kt)
```

Figura 31

Para realizar la misma funcionalidad que Java, deberíamos llamar a una función y que esta nos transformara el número a entero, es razonable ya que el objetivo del casteo no es modificar el valor que se tiene previamente, si no tiparlo sin perder la información previa.

Un mayor acercamiento a esto es dar un tipo nulable a las variables como vimos en el apartado Null Safety, esto aun así dará como resultado 'null': le ha sido imposible realizar el casteo, pero el 'safe cast' no produce la excepción y devuelve null.

Aquí vemos las dos situaciones nombradas:

```
var d: Double? = 54.45;

var i1: Int? = d as? Int?; //SAFE CAST
println("Entero " + i1);
var i2: Int? = d?.toInt(); //FUNCIÓN DOUBLE->INT
println("Entero " + i2);
```

Figura 32

```
Entero null
Entero 54
```

Figura 33

Dicho todo esto, ¿cómo se comprueban los tipos en ambos lenguajes y por qué es esto importante en el casteo?

Dado que estamos hablando de dos lenguajes fuerte y estáticamente tipados, nos facilitan bastante el conocer qué tipo tiene cada variable en cada momento, un ejemplo contrario podría ser Javascript. Sin embargo, puede haber algún momento en el que queramos conocer el tipo de una variable por haber pasado por una serie de casteos, por haber pertenecido esta a una clase padre anteriormente, etc.

En este caso, vamos a ver cómo comprueba esto Java y Kotlin:

JAVA

```
Object o = "hola";
Object o2 = 14;
String s = "adios";

if(o instanceof String) System.out.println("Object con valor String es String");
if(o2 instanceof String) System.out.println("Object con valor Int es String");
if(s instanceof String) System.out.println("String es String");
```

Figura 34

```
Object con valor String es String
String es String
```

Figura 35

KOTLIN

```
var a: Any = "Hola";
var a2: Any = 14;
var s: String = "ADios";

if(a is String) println("Any con valor String es String");
if(a2 is String) println("Any con valor Int es String");
if(s is String) println("String es String");
```

Figura 36

```
Any con valor String es String
String es String
```

Figura 37

En estos ejemplos podemos ver qué sentencias utilizan cada lenguaje y realmente poco más ya que los resultados son prácticamente triviales, solo destacar la primera comprobación que como vemos compara una clase padre con una hija cuyo resultado es verdadero ya que el valor de la variable con tipo Object corresponde a un valor String válido. Es por ello que ambos lenguajes coinciden en que es intancia del tipo String.

¿Qué relación más allá de la obvia tiene la comprobación de tipos con el casteo?

Esta pregunta viene por una de las cualidades de Kotlin. Este lenguaje ofrece un casteo inteligente que mezcla ambos conceptos (casteo y comprobación) en la sentencia 'is' al que llaman 'smart cast' y que tiene muchas más cualidades a parte de la mostrada en el siguiente ejemplo:

KOTLIN

```
var s: Any = "Hola";

if(s is String) println("Muestro: " + s.length);
```

Figura 38

```
Muestro: 4
```

Figura 39

En principio no parece nada especial que no hayamos contado ya... Pero vamos a probar en Java a ver qué ocurre:

```
Object s = "hola";

if(s instanceof String) System.out.println("Entro: " + s.length());
```

Figura 40

```
Cast.java:5: error: cannot find symbol
        if(s instanceof String) System.out.println("Entro: " + s.length());
                                                                 ^
  symbol:   method length()
  location: variable s of type Object
1 error
```

Figura 41

Efectivamente, no podemos acceder a la propiedad 'length' porque hemos declarado la variable en cuestión con el tipo Object y no con el tipo String desde un principio. La diferencia respecto a Kotlin es que este realiza un 'downcast' explícito si comprueba que realmente es instancia del tipo señalado en la sentencia 'is', es decir, comprueba que la variable es tipo String y si lo es, castea la variable: **comprobación y casteo posterior**. Es por ello por lo que podemos acceder a la propiedad 'length' sin ningún problema.

¿Qué tendríamos que hacer en Java para semejar dicho comportamiento?

```
Object s = "hola";

if(s instanceof String) {
    String s2 = (String) s;
    System.out.println("Entro: " + s2.length());
}
```

Figura 42

```
Entro: 4
```

Figura 43

Hay que especificar y programar un casteo previo y asignarle este a una nueva variable a la cual accederemos con la propiedad 'length': **comprobación y casteo posterior**.

Yendo más allá sobre la potencia que nos aporta la sentencia 'is' en Kotlin y viendo algún sobre ello, podemos aplicar esto al tipado característico de este lenguaje y convertir tipos nulables en tipos no-nulables solo con esta sentencia:

```
var s: Any? = "Hola";

if(s is String?) println("Muestro: " + s?.length);
if(s is String) println("Muestro: " + s?.length);
```

Figura 44

```
Cast.kt:5:44: warning: unnecessary safe call on a non-null receiver of type Any?
    if(s is String) println("Muestro: " + s?.length);
                                           ^
```

Figura 45

Podemos ver cómo, a propósito, intentamos acceder con el 'safe access' (?.) a la variable en cuestión dentro del if que comprueba si la variable es tipo String no-nulable y, efectivamente, verificamos que se ha tipado a String no-nulable ya que el compilador nos lanza el warning de que no es necesario el 'safe access' en el objeto. Aunque ponga de tipo 'Any?' en el mensaje del compilador, realmente está haciendo referencia al nuevo tipado String non-null casteado por la sentencia 'is'.

De igual manera en la primera sentencia, el objeto pasa a ser de tipo Any? a tipo String?, el resultado, por cierto, es trivial explicado ya el desarrollo:

```
Muestro: 4
Muestro: 4
```

Figura 46

2.7 Interoperabilidad

En la introducción se ha hablado ya de Kotlin como propósito general y la interoperabilidad con todo tipo de lenguajes: hemos visto que la interoperabilidad

con **Java** ha sido un pilar fundamental en la creación del lenguaje en todo momento para facilitar el paso de los desarrolladores de este último al novedoso lenguaje de JetBrains, de hecho existen traductores Java-Kotlin en la mayoría de los entornos de desarrollo para facilitar este paso; así como la posibilidad de interactuar con módulos de **Javascript** en la programación más orientada a web, y el desarrollo nativo interactuando con lenguajes como **C**, **Objective-C**, **Swift**, etc (4).

Centrándonos un poco en la interoperabilidad fundamental: **Java**, sabemos que la interoperabilidad completa es posible ya que Kotlin se compila en la máquina virtual de Java. La interoperabilidad es, como decimos, completa hasta tal punto que podemos conciliar ambos lenguajes en un mismo proyecto e incluso en un mismo paquete.

Existen algunos problemas con esto y uno de ellos son las palabras reservadas por ambos lenguajes: imaginemos que tenemos una clase escrita en Java y queremos llamarla desde un archivo Kotlin, algo que se da bastante a menudo. Ahora uno de los métodos de esa clase en Java tiene el nombre de una palabra reservada en Kotlin, por ejemplo: 'in', 'object', 'is', 'as'… Estas palabras no están reservadas en el lenguaje Java, pero sí en Kotlin, por lo que es posible llamar a una función de esa forma en Java, pero en Kotlin no. Cuando intentamos llamar a ese método del objeto en Kotlin, el compilador no lo permite. Para ello se utiliza el método de *escaping* con el cual escribimos las palabras entre tildes invertidas tal que así: `ìn`, `object`, `is`, `as`. De esta forma el compilador no reconoce la palabra reservada y la máquina, al compilar el código reconoce la llamada a la función implementada en Java.

De la misma forma, es posible ofrecer información a Kotlin desde los archivos Java mediante notaciones con el carácter @. Un ejemplo lo tenemos en el tratamiento de los valores nulos. Kotlin toma todos los valores de archivos Java como valores nulables puesto que este lenguaje no contiene nada que lo impida. Sin embargo, si sabemos que estos valores nunca van a tomar el valor nulo, podemos hacer que Kotlin los pase a no-nulables en su código mediante la notación @NotNull delante de cualquier variable, función, etc. Incluso podemos crear nuestras propias anotaciones personalizadas que creen un comportamiento u otro en cualquier lenguaje.

Estas son una de las muchas cualidades de interoperabilidad que ofrece Kotlin con Java, pero no nos olvidemos que fuera de estas cualidades específicas, existe una interoperabilidad clave entre los dos lenguajes a nivel de proyecto. Antes de decíamos que era habitual llamar desde un archivo Kotlin a otro Java, y es que todavía hay muchas librerías de Android que todavía no existen en lenguaje Kotlin o que todavía no han migrado su código al lenguaje de JetBrains y siguen en Java, por lo que, si se quiere dar uso de ellas en lenguaje nativo, se ha de llamar a estos archivos Java y por lo tanto, esta interoperabilidad es indispensable para la transacción continua de un lenguaje a otro. Esto es lo que hace a Kotlin especial, el no despreciar a su lenguaje antecesor, si no arroparlo y unirse a él para beneficiar a la comunidad novel y no defraudar a aquellos que sigan programando en Java pero que temen el cambio a este nuevo lenguaje.

2.8 Programación asíncrona (co-rutinas)

Como podemos conocer por Java, Kotlin se compila y ejecuta de manera procedural, es decir, de forma síncrona. A diferencia de otros lenguajes basados en bucles de

eventos (event-loop) como Javascript, en Kotlin el código se compila y ejecuta línea a línea, por lo que puede producirse un bloqueo en el flujo del programa o aplicación cuando se intenta llevar a cabo un comportamiento más costoso temporalmente hablando de lo habitual.

La **programación asíncrona** evita precisamente eso, el bloqueo del programa por el alto tiempo de ejecución de una de sus partes ya sea por tareas de alto consumo o por el tratamiento de datos masivo. Si sabemos localizar estas partes, podemos hacer que nuestro programa prosiga sin haber obtenido resultado alguno de la función costosa y recogerlo posteriormente para su uso.

Este bloqueo es ciertamente molesto sobre todo en aplicaciones con interfaz, en las que un usuario está en todo momento esperando una respuesta, ya sea visual, auditiva o de cualquier tipo, por parte de la aplicación.

Veamos un ejemplo de un problema que nos puede ocurrir habitualmente:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val saleService = Retrofit.Builder()
        .baseUrl( baseUrl: "http://192.168.1.44:3000/api/")
        .addConverterFactory(MoshiConverterFactory.create())
        .build()
        .create(SaleService::class.java)

    saleService.all().execute()
```

Figura 47

En este fragmento de código hemos preparado una llamada a una API externa y la hemos ejecutado (última línea), el código posterior a esta última línea esperará hasta que termine su ejecución independientemente del tiempo que lleve, llegamos en esa línea pues a un estado de bloqueo de nuestra aplicación.

Actualmente la programación asíncrona puede abarcarse de 5 diferentes formas en lenguajes como Java o Kotlin, vamos a analizar cada una de ellas y posteriormente veremos por qué solución opta Kotlin y Java y la comparación entre ambos lenguajes (10).

Para ejemplificar cada una de las siguientes soluciones se utilizará una llamada a una API externa, es igualmente aplicable a tareas de alto consumo o al tratamiento de grandes cantidades de datos.

- **Hebras:** el flujo principal del programa se ejecuta en una y solo una hebra principal si no hacemos nada al respecto, si esta se bloquea todo el programa y la interfaz queda bloqueada, por lo que lo que podemos hacer es ejecutar esta función que sabemos que costará en términos de tiempo de ejecución en otra hebra paralela y liberar la principal, siguiendo esta su curso con normalidad.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    Thread {
        val saleService = Retrofit.Builder()
            .baseUrl( baseUrl: "http://192.168.1.44:3000/api/")
            .addConverterFactory(MoshiConverterFactory.create())
            .build()
            .create(SaleService::class.java)

        val sales = saleService.all().execute()
    }.start()
```

Figura 48

Creamos la hebra y especificamos su contenido, este será el comportamiento que hemos comprobado que podría bloquear la aplicación. Una vez hecho esto, la ejecutamos con el método start() en la hebra principal de nuestro programa.

A pesar de que esta solución sea de las más utilizadas en la programación asíncrona de muchos lenguajes y en específico lenguajes como Java, tiene muchas desventajas entre las que destacamos:

- Evidentemente no existe un número ilimitado de hebras y normalmente están limitadas por el sistema operativo en el que se esté ejecutando el programa, la creación de muchas de ellas puede resultar, en algunas ocasiones, en problemas de rendimiento por cuellos de botella.
- Limita la compatibilidad con otros lenguajes, viéndose reducida en plataformas como Javascript que no dan soporte a hebras.
- La detección de errores es costosa cuando tratamos con hebras, suelen producirse condiciones de carrera: situaciones en las que se llevan a cabo más de una operación al mismo tiempo, si no sabemos controlar el número de hebras y las relaciones entre ellas podemos tener problemas graves.

- **Callbacks:** bastante utilizados en lenguajes como Javascript, un callback es una función que se pasa por parámetro a otra función para así poder llamarla una vez finalizado su comportamiento.

```
fun requestAPI(saleService: SaleService, callback: (Response<List<Sale>>) -> Unit) {
    val salesResult = saleService.all().execute()
    callback(salesResult)
}
```

Figura 49

```
val saleService = Retrofit.Builder()
    .baseUrl( baseUrl: "http://192.168.1.44:3000/api/")
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
    .create(SaleService::class.java)

requestAPI(saleService) {
    sales -> print(sales.toString())
}
```

Figura 50

En primer lugar creamos la función requestAPI(), la cual recibe por parámetro la función callback que queremos que se ejecuta posterior al contenido costoso en tiempo, después de esto llamamos a esta función en la hebra principal pasándole como función callback la función print().
Vemos que el código se simplifica bastante y pasamos a no tener problemas con el rendimiento ni problemas relaciones con el hardware del dispositivo en el que se ejecuta el programa como sí ocurre con las hebras, pero ¿qué desventajas tiene?:

- Cuando anidamos diferentes callbacks, no es para nada raro, el código se vuelve engorroso y de difícil comprensión.
- El anidamiento de los diferentes callbacks puede producir errores y la captación de estos produce aún más anidamiento.

Hemos comentado al principio que los callbacks son muy utilizados en lenguajes donde predomina la programación asíncrona, como Javascript con arquitecturas de bucle eventual (event-loop), pero cada vez caen más en desuso por una solución más limpia que, al menos, solucionan el problema del anidamiento.

- **Promesas:** el objetivo de este tipo de solución es que al hacer una llamada a la función costosa, esperamos que nos retorne un objeto (promesa) con el que podrá operarse a posteriori. La ventaja es que podemos realizar una cadena de promesas sin anidamiento, un punto a favor frente a los callbacks.

```
fun requestAPI(saleService: SaleService): Promise<Response<List<Sale>>> {
    return saleService.all().execute()
}

val saleService = Retrofit.Builder()
    .baseUrl( baseUrl: "http://192.168.1.44:3000/api/")
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
    .create(SaleService::class.java)

requestAPI(saleService).thenAccept { sales ->
    print(sales.toString())
}
```

Figura 51

Nuestra función costosa requestAPI ahora devuelve una promesa, esto es una desventaja de esta solución ya que cualquier función costosa de la que posteriormente quieras obtener su resultado, debes devolver una promesa con ese resultado. En este caso, devolvemos una promesa con un valor Response<List<Sale>>. Posteriormente, hacemos uso de la librería de promesas y obtenemos el valor de la función asíncrona con el 'then' característico.
Toda la sintaxis y métodos que nos ofrece esta librería de promesas va a ser explicada en el apartado del lenguaje Java de programación asíncrona un poco más adelante, dicho esto podemos ir directamente a las ventajas y desventajas de esta solución.
A pesar de que son bastante usadas actualmente y que se han convertido en una solución popular a los callbacks, las promesas tienen algunas desventajas que conviene repasar:

- Se cambia bastante la forma de programar de una forma imperativa a otra basada en cadena de promesas donde, por ejemplo, los bucles no tienen mucha cabida.
- Cuando se encadenan varias promesas el manejo de errores se hace complicado produciéndose errores de propagación y encadenamiento como el no saber que eslabón ha sido el que ha producido el error al capturarlo, dificultad de pasar valores de un eslabón a otro no consecutivo, etc.
- Ya hemos comentado antes sobre la necesidad de devolver un tipo en específico de manera obligatoria, en lugar de devolver el dato del tipo que queremos se ha de devolver un objeto Promise con el 'subtipo' que queramos recoger.
- Por último y la más importante, la necesidad de aprendizaje de una nueva API que ha de ser importada de forma externa prácticamente en todos los lenguajes.

- **Reactive Extensions (Rx):** la programación reactiva fue introducida en C# y utilizada en el framework .NET y obedece al patrón de diseño Observador. Fue el equipo de desarrollo de Netflix el que trasladó esta solución al lenguaje Java para mejorar su API y, posterior a esto, se ha trasladado a múltiples lenguajes y plataformas.

  El objetivo de esta solución es mejorar el concepto que ofrecen las promesas. La sintaxis es muy parecida entre ambas soluciones: sustituimos el tipo Promise por Observable en los retornos, .next del observador para devolver el valor en lugar de .resolve y .error del observador para lanzar un error en lugar de .reject; después a la hora de obtener el resultado lo obtenemos con .subscribe del Observable, nos suscribimos al Observable en el tiempo, en lugar del .then de las promesas.

Dado que esta programación reactiva viene como sustitución a las promesas, podemos ver unas pocas mejoras de las muchas que nos aporta frente a estas y ver si solucionan las desventajas anteriormente mencionadas:

- Mientras que con las promesas solo podemos devolver un valor, con los Observables es posible la devolución de múltiples valores ya que lo que devolvemos es un stream, secuencia de múltiples datos, a los cuales nos suscribimos (subscribe) para escucharlos todos y cómo cambian en el tiempo.
- Los Observables son perezosos. A la hora de construir una promesa el constructor de esta ejecuta la función que se le pasa por parámetro, mientras que los Observables solo ejecutan esa función cuando en algún punto del código existe una suscripción a dicho Observable. Esto ayuda al rendimiento pues no tenemos multitud de promesas ejecutándose y consumiendo recursos sin ser llamadas en nuestro programa.
- Los Observables dan la posibilidad de ser cancelados. Si tenemos un comportamiento de larga duración y entramos en una situación en la cual ya no requerimos del resultado de dicho comportamiento, podemos cancelarlo si se trata de un Observable pero no si se trata de una Promesa, la cual seguiría ejecutando su función hasta terminar.

Hemos visto muchas funcionalidades y mejoras de los Observables pero, ¿solucionan algo de lo anteriormente mostrado? Pues seguimos teniendo que: cambiar nuestra forma imperativa de programar en el lenguaje Kotlin, devolver el valor requerido mediante un tipo en específico, en este caso, un Observable y llevar un aprendizaje extra sobre la API de los Observables. Lo que si hemos mejorado ha sido la obtención de múltiples valores que soluciona el tedioso paso de datos entre promesas solucionando así los errores de encadenamiento pero manteniendo los errores de propagación que permanecen tanto en callbacks como en promesas.

- **Co-rutinas:** precisamente del problema del cambio de modelo de programación a la hora de implementar tareas asíncronas, nacen las co-rutinas. Esta solución es específica de Kotlin y no está disponible en Java. Están implementadas en un conjunto de librerías con la idea de la computación suspendible: una función puede ser suspendida en cualquier lugar y momento para ser retomada posteriormente sin ningún problema. Como hemos dicho antes, uno de los pilares en los que se basa esta solución es en su implementación en librerías y no de forma nativa del lenguaje como si lo hace por ejemplo C#, y también en mantener el modelo programático síncrono y de arriba a abajo, cosa que no consiguen callbacks y promesas.

```
suspend fun requestAPI(saleService: SaleService): Response<List<Sale>>{
    return saleService.all().execute()
}

val saleService = Retrofit.Builder()
    .baseUrl( baseUrl: "http://192.168.1.44:3000/api/")
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
    .create(SaleService::class.java)

GlobalScope.launch { this: CoroutineScope
    val sales = requestAPI(saleService)
    print(sales)
}
```

Figura 52

Como vemos el bloque programático que tendríamos en caso de que la función requestAPI no fuera costosa sería el mismo, solo hemos añadido el tener que llamar en la hebra principal a la función suspendible dentro de un bloque 'launch', que lanzará nuestra co-rutina sin bloquear la hebra principal. La co-rutina será ejecutada, pausada, reanudada o cancelada cuando así lo sea el trabajo que dentro de ella se realiza, especificado como vamos a comentar ahora.
Otra variante que hemos tenido que añadir ha sido el prefijo 'suspend' que informa de que esa función puede ser ejecutada, pausada y posteriormente reanudada. Esto puede ser especificada en un bloque de código llamando a la función suspendCoroutine en la propia función suspendible, como vemos en el siguiente ejemplo:

```
suspend fun requestAPI(saleService: SaleService): Response<List<Sale>>{
    return suspendCoroutine { continuation ->
        /* Especificación */
        val sales = saleService.all().execute()
        if(sales.isSuccessful) continuation.resume(sales)
        else continuation.resumeWithException(throw Exception())
    }
}
```

Figura 53

El objeto continuation controla el comportamiento post-pausa de la función y vemos que en el caso de que la petición se haga correctamente, reanudamos la función suspendible con el resultado y si, por el contrario, obtenemos un error esta será reanudada con una excepción.
Veamos, en definitiva, qué ventajas nos ofrecen las co-rutinas frente a las demás soluciones nombradas anteriormente:

- Como hemos nombrado anteriormente, una de las mayores ventajas frente a las demás soluciones es el hecho de no tener que cambiar de modelo de programación pues continuamos con la forma síncrona top-bottom de Kotlin.
- Todo esto es posible sin muchos cambios en nuestro código, solo hay que encapsular las llamadas en la hebra principal con 'launch' desde

un contexto y la adición del prefijo 'suspend' para la función suspendible.
- No es necesario el aprendizaje de una nueva API como así ocurre con las promesas.
- Esta solución es independiente de la plataforma en la que se ejecute y no afecta a la multitud de compatibilidades que tiene Kotlin, como si afectaba la solución de hebras en compatibilidades como Javascript.
- Volvemos a poder utilizar bucles, ya que nuestro modelo programático no cambia, y el manejo de errores vuelve a ser sostenible tal y como era desde un principio en el propio lenguaje.
- Los problemas de propagación, presentes en las soluciones con callbacks, ya no están presentes pues solo existe el único anidamiento que produce el 'launch'.

Como vemos, las desventajas que hemos ido teniendo a lo largo de toda la explicación de las diferentes soluciones a la asincronía se han solucionado con las co-rutinas, es por esto por lo que actualmente **es la solución más utilizada y recomendada en Kotlin.** Las co-rutinas no son un concepto inventado por JetBrains para su lenguaje, ya se usan de forma muy parecida en lenguajes como Go, lo ventajoso en Kotlin es que toda esa funcionalidad está exportada en librerías que pueden ser importadas o no dependiendo si nos son necesarias, con todo lo bueno que eso conlleva.

2.8.1 ¿Y en qué lugar nos deja esta explicación dentro de nuestra comparación entre Kotlin y Java?

La comparación entre ambos lenguajes ya está hecha, solo queda ver qué solución o soluciones de las nombradas anteriormente es por la que opta Java.

Actualmente, Java 8 ha realizado multitud de avances en lo referente a la programación asíncrona. Previo a esta actualización, las soluciones eran la creación de hebras y, yendo un poco más allá, la interfaz Future.

En cuanto a la utilización de **hebras** nos vale como explicación la realizada más arriba en el apartado de Kotlin, la utilización es la misma, así como las ventajas y desventajas de la solución.

Future, sin embargo, es una interfaz que te permite realizar una llamada asíncrona, obtener su resultado e incluso cancelar la ejecución de la tarea (característica destacada de la programación reactiva). El problema llega a la hora de obtener el resultado, tenemos solo un método al que llamar (get) que bloquea el resto de la ejecución, por lo que la asincronía pasa a tener poco sentido y se necesitaría implementar individualmente y de forma específica cada escenario y cada llamada asíncrona (11).

La evolución de Java 8 llega con una nueva implementación basada en las promesas de Javascript que implementa las interfaces Future y CompletionStage, llamada CompletableFuture.

CompletionStage hace posible la obtención del resultado de forma asíncrona, solventando así el problema de la interfaz Future, pudiendo añadir funciones callback tras la ejecución de la llamada. Ahora es posible obtener el resultado sin bloquear la hebra principal.

```
//No se utiliza el resultado de la función asíncrona
CompletableFuture.runAsync(() -> System.out.println("FIN"))
//Capturamos error en la función asíncrona
.exceptionally(err -> new Exception("ERROR"));

//Utilizamos el resultado de la función asíncrona
CompletableFuture.supplyAsync(() -> "Hola")
//Capturamos error en la función asíncrona
.exceptionally(err -> new Exception("ERROR"))
//Transforma el resultado
.thenApply(s -> s + " a todos")
//Aplica un método sin transformar el resultado
.thenAccept(s -> System.out.println("Saludo: " + s))
//No devuelve nada ni utiliza el resultado
.thenRun(() -> System.out.println("FIN"));
```

Figura 54

Como vemos en el ejemplo, con runAsync y supplyAsync podemos obtener el resultado de forma asíncrona. El primero no utiliza el resultado que nos proporciona nuestra función costosa y el segundo lo utiliza para, posteriormente, transformarlo mediante diferentes métodos similares a los que ya teníamos, y hemos explicado, con las reactive extensions, RxJava en este caso:

- **thenApply**: sería el equivalente a la función 'map' de los streams y de las reactive extensions (RxJava), transforma el resultado obtenido de la función costosa y lo devuelve transformado.
- **thenAccept**: su equivalente sería el 'foreach' de los streams y de las reactive extensions (RxJava), obtiene el resultado de la función costosa y aplica un método sin transformarlo.
- **thenRun**: ni utiliza ni transforma el resultado, solo espera a que la función costosa concluya para ejecutar la acción especificada.

Todos estos métodos se ejecutan de forma síncrona, pero en caso de que queramos que se ejecutan de manera asíncrona también tienen su variante.

```
//Utilizamos el resultado de la función asíncrona
CompletableFuture.supplyAsync(() -> "Hola")
//Capturamos error en la función asíncrona
.exceptionally(err -> new Exception("ERROR"))
//Transforma el resultado de forma asíncrona
.thenApplyAsync(s -> s + " a todos")
//Aplica un método sin transformar el resultado de forma asíncrona
.thenAcceptAsync(s -> System.out.println("Saludo: " + s))
//No devuelve nada ni utiliza el resultado de forma asíncrona
.thenRunAsync(() -> System.out.println("FIN"));
```

Figura 55

También cabe destacar la función '**exceptionally**', cuando es lanzado un error desde la función costosa esta función lo captura y maneja el error con el código

especificado en el callback. Sería el equivalente al 'catch' en las promesas de Javascript.

Cuando se requiere de un comportamiento más complejo se produce el anidamiento, característica en los callbacks. Para no caer en el mismo error, esta librería nos ofrece varios métodos para solucionarlo: thenCompose y thenCombine.

```
//Utilizamos el resultado de la función asíncrona
CompletableFuture.supplyAsync(() -> "Hola")
//Encadenamos dos llamadas asíncronas con thenCompose
.thenCompose(s -> CompletableFuture
.supplyAsync(() -> s + " mundo"));
```

Figura 56

Con **thenCompose** podemos realizar una cadena de promesas tal y como podemos hacerlo en Javascript sin problemas de anidamiento.

thenCombine tan solo es una función que puede combinar dos CompletableFuture para un mejor manejo de errores y una mayor eficiencia.

En adición a esta solución basada en promesas, se encuentra la ya nombrada varias **RxJava**, la librería de programación reactiva del lenguaje. Ya hemos nombrado todas las ventajas y desventajas de esta implementación en el apartado de Kotlin, pues se ha hecho un análisis genérico y no especializado en el lenguaje.

Con la versión 8 de Java se ha avanzado en el campo de la asincronía y, actualmente, se barajan dos soluciones viables a la hora de abarcarla: promesas y reactive extensions. Kotlin, sin embargo, ha ido un poco más allá con las co-rutinas. Así que, finalmente, nuestra comparación entre Kotlin y Java se ve limitada al análisis y comparación de las soluciones más viables y actuales de cada lenguaje: promesas y programación reactiva por parte de Java, y co-rutinas por parte de Kotlin. Por lo tanto, utilizamos lo ya escrito en el apartado de Kotlin sobre estas soluciones para dar por terminado este importante apartado.

2.9 Concisión vs. verbosidad

Uno de los aspectos clave de Kotlin y donde JetBrains ha hecho bastante hincapié ha sido en la concisión de su lenguaje frente a Java. De la misma forma, sabemos de la carencia de esta concisión en el lenguaje de Oracle, a esto se le suele llamar verbosidad. En este apartado veremos cómo consigue esta concisión Kotlin y, si es el caso, en qué basan sus ideas comparando cada una de estas con Java y viendo si realmente son relevantes o no.

2.9.1 Clases y clases de datos

En primer y más destacado lugar se encuentra la creación de objetos (clases) y sus métodos predeterminados. Veamos con un ejemplo cómo se desenvuelven ambos lenguajes a la hora de crear una clase básica:

KOTLIN

```
class Person(var name: String, var age: Int);
```

Figura 57

JAVA

```
public class Person {

    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

Figura 58

- Centrémonos primero en la ausencia de getters y setters en la clase representada en Kotlin porque esta va más allá de la creación de clases. Cuando queremos crear una variable en Java, estamos creando un "**campo**" (o field) mientras que lo que creamos en Kotlin es una "**propiedad**" (o property).

  Campo en Java

  ```
  int age;
  ```

  Figura 59

  Propiedad en Kotlin

  ```
  var age: Int
  ```

  Figura 60

  La diferencia entre ambas es que la propiedad creada en Kotlin contiene los métodos getter y setter compilados en sí mismos. Estos dos métodos están implementados en su forma más básica. Para lograr un acercamiento a esto

que hace Kotlin, en Java deberíamos de implementarlos nosotros mismos en la clase como hemos visto en el primer ejemplo de la clase Persona.

En caso de que queramos modificar dichos métodos, se haría de una forma muy sencilla dentro de la propia clase:

```
var age: Int
    get() = field * 12
    set(value) {
        field = value + 1
    }
```

Figura 61

En este trozo de código hemos modificado el método get() de la propiedad "age" para que devuelva la edad en meses y el método set() clásico para que añadamos un año más a la persona en cuestión.

- El siguiente caso por analizar es la ausencia de constructor. Al igual que en el caso de getters y setters existe una implementación predominante la cual es la más básica posible de estos métodos (se han visto en los primeros ejemplos), en el constructor pasa prácticamente lo mismo. Kotlin incluye un constructor básico predominante en todas sus clases, normalmente suele ser así:

```
class Person() {
    var name: String
    var age: Int

    constructor(name: String, age: Int) {
        this.name = name;
        this.age = age;
    }
};
```

Figura 62

Para no tener un constructor implementado de la misma forma en todas las clases, Kotlin decide tomar una sintaxis más concisa eliminando el constructor, pero manteniendo el formato constructor para la clase ya que muchas veces pasamos valores a esta mediante dicho constructor:

```
class Person(var name: String, var age: Int);
```

Figura 63

Con esto podremos crear un objeto Persona de la misma forma que llamamos en Java, incluso omitiendo la palabra "new":

```
var person: Person = Person("Francisco", 23)
```

Figura 64

La sintaxis anterior nos permite también crear clases con propiedades inmutables (clase inmutable), con propiedades mutables e inmutables o solo con propiedades mutables:

```
//Clase mutable
class Person(var name: String, var age: Int);

//Clase con propiedades mutables e inmutables
class Person(val id: Int, var name: String, var age: Int)

//Clase inmutable
class Person(val id: Int, val dni: String)
```

Figura 65

Con el modificador "val" podemos crear variables inmutables, solo pueden inicializarse una vez y no podrán modificar su valor.

Así nos queda una clase útil tal y como es utilizada en Java y concisa, pero ¿y si queremos definir otros constructores propios? Es posible tal y como lo hacemos en Java, así tendremos tantos constructores como queramos definir más el dominante proporcionado por Kotlin:

```
class Person(var name: String, var age: Int) {
    constructor(name: String): this(name, 18)
};
```

Figura 66

Como decíamos en el inicio del apartado, queremos centrarnos en los métodos predeterminados de las clases. Nos faltan varios métodos muy utilizados en Java y en Programación Orientada a Objetos: "equals", para comprobar si dos objetos son iguales, "toString", usado para representar claramente un objeto, y "hashCode", necesario para mantener objetos en dos tipos diferentes en el momento de hashearlos (Set o Map).

¿Cómo se representan las clases incluyendo estos métodos?

KOTLIN

En Kotlin, todas las clases tienen en común una superclase (o clase padre) llamada Any a la que pertenecen y la cual contiene los tres métodos básicos nombrados anteriormente. Una clase creada como hemos visto anteriormente heredará estos tres métodos.

```
class Person(var name: String, var age: Int);

val person = Person("Francisco", 23)

println(person); //Imprime: ConcisenessKt$main$Person@1b6d3586
println(person.toString()); //Imprime: ConcisenessKt$main$Person@1b6d3586

println(person.equals(Person("Francisco", 23))) //Imprime: false
println(person.equals(person)) //Imprime: true

println(person.hashCode()) //Imprime: 460141958
```

Figura 67

Como vemos, el método “toString” realiza una representación bastante pobre del objeto basando su último número en el hashCode de la clase con la forma: “nombre_de_la_clase”$”función_que_la_ejecuta”$”nombre_de_la_clase”$”hashCode_HEX”, y el método “equals” compara las referencias de ambos objetos. En esto profundizaremos más adelante con el comportamiento en Java, pues es muy similar y nos sirve para explicar la herencia de métodos y la sobreescritura de estos.

La innovación de Kotlin en este aspecto viene aquí:

```
data class Person(var name: String, var age: Int);
```

Figura 68

Efectivamente, solo al poner la palabra “data” delante de la declaración de la clase, Kotlin incluye una implementación básica de los métodos mencionados anteriormente y otras características (12):

- Métodos toString: realiza una representación básica del objeto y sus propiedades.

```
var person = Person("Francisco", 23)

print(person) //Imprime: Person(name=Francisco, age=23)
print(person.toString()) //Imprime: Person(name=Francisco, age=23)
```

Figura 69

- Método equals: para el método “equals” aplica el mismo método a cada una de las propiedades del objeto, si todas las comprobaciones son verdaderas, los objetos son iguales.

```
val person = Person("Francisco", 23)

print(person.equals(Person("Paco", 25))) //Imprime: false
print(person.equals(Person("Francisco", 23))) //Imprime: true
```

Figura 70

- Deconstrucción: permite la separación en forma de tupla en diferentes variables de cada una de las propiedades del objeto en cuestión.

```
val person = Person("Francisco", 23)
val (name, age) = person

print(name) //Imprime: "Francisco"
print(age) //Imprime: 23
```

Figura 71

- Método copy: equivalente al método “clone” en Java, te ofrece una forma de copiar el objeto con modificaciones en sus propiedades que facilita así la inmutabilidad de los objetos, muy utilizado en programación funcional.

```
val person = Person("Francisco", 23)
val personCopy = person.copy(age = 25)

print(person) //Imprime: Person("Francisco", 23)
print(personCopy) //Imprime: Person("Francisco", 25)
```

Figura 72

Al fin y al cabo, lo que Kotlin nos ofrece es una implementación básica basada en la experiencia y utilidad que han mostrado los usuarios tras años y años de utilización de Java.

JAVA

Ya sabemos que en Java hay que implementar estos métodos manualmente. Vamos a intentar ir más allá.

Antes hemos visto la superclase Any y los métodos que implementan todas las clases hijas que creamos en Kotlin. En Java pasa algo parecido, todos los objetos que creemos en este lenguaje heredarán de la clase Object, esta a su vez tiene una implementación de los métodos nombrados anteriormente (toString, equals y hashCode), es por eso que al añadir una implementación manual en nuestra clase de estos métodos es correcto utilizar la anotación @Override, ya que indica al compilador que vamos a modificar un método de la clase padre. Vamos a analizar si esta "implementación padre" es suficiente para tratar con datos relacionados a la clase que creemos o es totalmente necesaria su implementación manual.

- Método toString: el método toString de la clase Object imprime el nombre de la clase seguido del hashCode en forma hexadecimal. Así que se nos hace bastante necesaria su reimplementación.

```
Person person = new Person("Francisco", 23);
System.out.println(person.toString()); //Imprime Person@15db9742
```

Figura 73

- Método equals: realiza una comparación superficial comprobando si las referencias a dos objetos son iguales, por lo tanto, si queremos hacer una comparación teniendo en cuenta los campos de las clases creadas, no es imposible. Se necesitaría una reimplementación.

```
Person person = new Person("Francisco", 23);
Person personCopy = new Person("Francisco", 23);
System.out.println(person.equals(personCopy)); //Imprime false
System.out.println(person.equals(person)); //Imprime true
```

Figura 74

Como vemos, a pesar de darle a los dos objetos los mismos valores en sus campos, el método "equals" de la clase Object no los tiene en cuenta y devuelve falso.

- Método clone: el equivalente al método copy de Kotlin es el clone de la clase Padre, al ser un método protegido habría que implementar la interfaz Cloneable en nuestra clase Person para así poder acceder y clonar el objeto en nuestra clase aplicación.

```java
public class Person implements Cloneable {

    public Object clone()throws CloneNotSupportedException{
        return super.clone();
    }
```

Figura 75

```java
Person person = new Person("Francisco", 23);
Person personCopy = (Person)person.clone();

//toString
System.out.println(person.toString()); //Imprime Person@15db9742
System.out.println(personCopy.toString()); //Imprime Person@6d06d69c

//equals
System.out.println(person.equals(personCopy)); //Imprime false
System.out.println(person.equals(person)); //Imprime true
```

Figura 76

Podemos deducir, como era de esperar, que es necesaria una implementación de los métodos "toString" y "equals" en la clase a representar, así como la implementación de una interfaz perteneciente a la clase Object para poder utilizar el método clone, equivalente al copy de Kotlin.

Aun así, esta demostración ha servido para ilustrar que, aunque a priori estos métodos no sean muy útiles, existen y el hecho de implementarlos en tu clase implica la sobreescritura de estos que provienen de la clase Object.

2.9.2 Sobrecarga de operadores

Una característica que también aumenta el grado de concisión en cualquier lenguaje es la sobrecarga en operadores. La versatilidad de la utilización de un operador u otro, por ejemplo, en la igualdad: las clases u objetos pueden compararse en Kotlin tanto con el operador "==" como con el método "equals", ambos ofrecen el mismo resultado. Podemos encontrar muchos más operadores regulados por una convención establecida por Kotlin como puede ser: "++" traducido a "inc", "/" traducido a "div", "in" traducido a "contains" para operaciones con cadena de caracteres o "+=" traducido a "plusAssign" entre muchos otros.

Pero no solo es eso, también podemos sobrecargar nosotros mismos los operadores básicos aplicados a tipos de datos en específico, y con tipo de datos me refiero a clases propias.

```kotlin
data class Coordenada(x: Float, y: Float);

operator fun Coordenada.inc() = Coordenada(x++, y++)

var coord = Coordenada(10, 10)

println(coord.inc()) //Imprime: Coordenada(x=11, y=11)
```

Figura 77

En este caso lo que hemos querido hacer es que al aplicar la función de incremento a nuestra clase "Coordenada", se incrementen ambas propiedades (ejes de coordenadas).

Aquí es donde Kotlin se distancia de Java en este subapartado. **Java no soporta la sobrecarga de operadores definida por el programador** como acabamos de hacer, solo la sobrecarga ya definida por el propio lenguaje.

2.9.3 When vs. Switch

La sentencia "when" puede considerarse una mejora en cuanto a concisión y funcionalidad del conocido por todos "switch" (4). Dado un argumento, la sentencia "when" ejecuta una serie de casos (en la sentencia "switch": cases) y hace algo en consecuencia si se cumple dicho caso. Esta sentencia no solo compara el valor pasado por argumento con otro dado por el programador, ejecuta un caso y este puede ser una expresión booleana, una función aplicada al argumento, etc. Esta sería una forma bastante básica de utilización de esta nueva sentencia:

```
when(cadena) {
    "ratón", "raton" -> println(cadena.toUpperCase())
    "RATÓN", "RATON" -> println(cadena.toLowerCase())
    else -> println("Error")
}
```

Figura 78

Podemos poner varias condiciones de éxito en la parte izquierda separadas por comas, mientras que en el "switch" vamos a ver cómo habría que escribir varias sentencias "case" y cómo el código, para una funcionalidad tan básica, se nos hace algo extenso:

```
switch (cadena) {
    case "ratón":
    case "raton":
        System.out.println(cadena.toUpperCase());
        break;
    case "RATÓN":
    case "RATON":
        System.out.println(cadena.toLowerCase());
        break;
    default:
        System.err.println("Error");
        break;
}
```

Figura 79

Lo novedoso del operador "when", y es por lo que hablábamos en un principio de funcionalidad, es la posibilidad de realizar comprobaciones de todo tipo en la parte izquierda de los "casos". Podemos llamar a funciones y aplicar todo tipo de operadores en la parte izquierda como no pasa con el operador "switch" de Java.

```
when(nota) {
    in 0..4 -> println("Suspenso")
    in 5..6 -> println("Aprobado")
    in 7..8 -> println("Notable")
    in 9--10 -> println("Sobresaliente")
    else -> println("Error")
}
```

Figura 80

Vemos como el argumento en este caso podría ser de tipo entero y los casos devuelven un valor booleano, es decir, podemos operar con dicho argumento y no solo limitarnos a comparar su valor con otro.

```
switch (cadena) {
    case cadena.contains("ra"):
    case "raton":
        System.out.println(cadena.toUpperCase());
        break;
```

Figura 81

```
App.java:7: error: incompatible types: boolean cannot be converted to String
            case cadena.contains("ra"):
                                 ^
```

Figura 82

Podemos ver como la sentencia “when” da un paso más allá en cuanto a funcionalidad permitiendo la operación con el argumento. En cuanto a concisión todos sabemos de lo engorroso que es tratar con una sentencia “switch” por la necesidad de la ruptura de bloque (break) y el no poder congregar varios casos posibles en una sentencia “case” y más que trivial, visto lo visto, que este problema es solucionado con la sentencia “when”.

2.9.4 Patrón Singleton

Existen muchas formas de implementar muchos de los patrones existentes. El patrón Singleton es uno de ellos y, sin duda, es uno de los más utilizados en la programación orientada a objetos. Vayamos a lo básico, el patrón de diseño Singleton permite restringir la creación de objetos pertenecientes a una clase a un único objeto, restringe a que una clase solo pueda tener una sola instancia la cual, normalmente, es punto de acceso global en el código.

Como hemos dicho existen múltiples formas de implementar este patrón, algunas de ellas no compaginan bien en sistemas concurrentes o son susceptibles a ataques de serialización. Aun así vamos a ver una de las formas más básicas de su representación en el lenguaje Java:

```
public class Singleton {
    private static Singleton instancia = new Singleton();
    private Singleton() {}
    public static Singleton getInstancia() { return instancia; }
}
```

Figura 83

Vemos como creamos un constructor privado de forma que podamos controlar las instancias que se crean de nuestra clase Singleton y creamos la instancia única que queremos de forma privada. De la misma forma construimos un getter de esta instancia para que clases externas puedan acceder a ella.

Dicho esto, vamos a ver cómo lo puede llegar a hacer Kotlin:

```
object Singleton {}
```

Figura 84

Una línea, dos palabras. Hay un gran contraste con la forma de crear clases en Java, tanto por la densidad de código como por la palabra "object" en lugar de "class". Realmente estamos definiendo un solo objeto, que es semánticamente lo mismo que definir una clase Singleton: una clase que siempre de la que siempre tendremos uno y solo un objeto.

Un apunte para ambos lenguajes es que tanto en Java como en Kotlin puedes elegir entre hacer inmutable o mutable tu objeto / clase Singleton.

No nos quedamos aquí, volvamos a Java. Porque existe una forma poco conocida que cabe mencionar para crear un Singleton.

```
enum Singleton {
    INSTANCIA
}
```

Figura 85

Sí, **un enumerado de un solo tipo**. Un enumerado, por definición, es un clase que limita la creación de objetos a los especificados dentro del enumerado (INSTANCIA en nuestro caso). En adición a esto, si el enumerado tuviera constructor, este debería de ser privado para que no se puedan crear nuevos objetos. Esta última frase nos huele a Singleton. Además, los enumerados están exentos a cualquier ataque por serialización, lastre que llevan las clases Singleton creadas mediante "class".

Evidentemente, podemos añadir propiedades y métodos tanto al objeto en Kotlin como al enumerado en Java.

Con esto, Java realiza un aproximamiento casi extremo a la implementación en Kotlin, y esto es algo que no hemos podido decir en muchos apartados que hemos tratado.

2.9.5 Programación Android: findViewById

En esta ocasión vamos a analizar un caso más específico, centrándonos en la programación Android, el propósito más conocido y útil de este lenguaje. En concreto, una funcionalidad bastante utilizada en la programación Android es el findViewById. Esta función se encarga de recuperar un elemento de la interfaz de la actividad o vista especificada, lo que nos permite asociarla a una variable en nuestro código Java o Kotlin y operar con dicho elemento. Es de las funciones más utilizadas en la programación Android, pero tiene varios inconvenientes.

```
name = findViewById(R.id.editText3);
number = findViewById(R.id.editText4);
```

Figura 86

Resulta tedioso tener que realizar la misma llamada y asignación para todos y cada uno de los elementos con los que queramos operar. Esto conlleva tanto problemas con la lectura (volvemos a la verbosidad) como problemas de rendimiento.

El problema de concisión es trivial, cada vez que queramos operar con algún elemento de la interfaz tenemos que acceder a él de la forma vista anteriormente mínimo una vez (se puede guardar en una variable).

```
TextView textView1 = dialogView.findViewById(android.R.id.text1);
textView1.setText(word);
```

Figura 87

Con estas dos sentencias estamos obteniendo un texto de la vista "DialogView" y modificándolo con el contenido de la variable "word". Veamos cómo podría ser en Kotlin:

```
dialogView.text1.text = word
```

Figura 88

El propio id del elemento de la interfaz (en este caso "text1") es una propiedad de la vista en cuestión, y a su texto podemos acceder como si de una propiedad se tratase.

Es solo un ejemplo de lo que Kotlin puede ofrecernos en cuanto a versatilidad y facilidad al programar para este propósito.

Gracias a un plugin llamado Kotlin Android Extension que nos ofrece el IDE de JetBrains, el cual viene preinstalado al iniciar el proyecto y solo tenemos que habilitar, podemos hacer cosas como esta. JetBrains, al igual que el propio lenguaje Kotlin, vuelve a fijarse en los comportamientos más usados y engorrosos de su antecesor y logra añadir nueva funcionalidad; ya sea formando parte del propio lenguaje o, si no es posible, en su propio IDE para que programar, en este caso Android, en su lenguaje vuelva a ser mucho menos engorroso que con su "antecesor".

Otro punto por destacar, comentado anteriormente, es el del rendimiento. Poco se puede hacer en ese aspecto ya que, a pesar de que el plugin de Kotlin nombrado anteriormente soluciona problemas de concisión, no lo hace en cuanto a rendimiento pues el compilador a fin de cuentas termina entendiendo un findViewById detrás de todo eso.

Imaginemos una interfaz con unos cuantos elementos, digamos 10 o 20. Tenemos que acceder a varios de ellos en un momento del ciclo de vida de nuestra actividad. Android por cada llamada findViewById, recorre todo el árbol jerárquico de la interfaz XML para buscar nuestro elemento y todos conocemos del posible coste de esa operación en determinados qué casos. Sumándole que probablemente queramos modificar el valor de alguna propiedad de este elemento buscado (como hemos visto anteriormente), aumentamos aún más la complejidad de nuestra funcionalidad.

La solución a este problema no está en uno u otro lenguaje, nos la ofrece Android independientemente del lenguaje con el que se complemente y no es más que utilizar una función "findViewTraversal" que no tendrá que recorrer el árbol jerárquico tantas veces como queramos acceder a dicho elemento en nuestro código. En caso de que este elemento se utilice varias veces podemos guardar el elemento en una variable miembro para poder así utilizarla tantas veces como se quiera (13).

2.10 Otros apartados

Tratados todos estos temas en relativa profundidad, solo nos queda comentar algunos casos que se nos hacen triviales o que no requieren de demasiada explicación y que contribuyen a la concisión de Kotlin igual o tanto como los anteriormente mencionados: la omisión del punto y coma siempre que sea necesario choca con la obligatoriedad de Java para escribirlo, dotando a Kotlin de nuevo con mayor ligereza a la hora de programar; hablando anteriormente de sentencias de control como el when-switch no podemos dejar atrás la sintaxis sencilla de los bucles "for" en Kotlin frente a la sintaxis repetitiva de Java; el ya nombrado y explicado apartado del Smart-cast deja bastante clara la balanza en este aspecto a favor de Kotlin; por último, decir que el hecho de que Kotlin no sea un lenguaje tan fuertemente tipado como Java, da un respiro y versatilidad al programador sin dejar de lado la robustez que te ofrece el tipado de datos.

2.11 Conclusión

Como conclusión, y rompiendo una lanza a favor de Java, podemos decir que Kotlin ha obtenido toda esta concisión a través de años estudiando la usabilidad que se le da al lenguaje que ha venido a sustituir. Ha sabido corregir de manera correcta uno de los puntos flacos de Java, que es su verbosidad, a través del estudio de sus usuarios y lo que constantemente llevan pidiendo años y años pero que Java por su antigüedad y presencia no es capaz de dar.

Al igual que a día de hoy estamos hablando de esta diferencia de concisión entre Java y Kotlin, en un futuro y no muy lejano el papel de perdedor podría tenerlo Kotlin contra un posible sucesor llevado ahí por la experiencia.

3

Metodologías y fases del trabajo

La aplicación desarrollada nace de la necesidad por exponer en un ejemplo real las características de Kotlin nombradas, ya que los ejemplos de código del documento teórico pueden discernir de la realidad. Sabiendo esto, se barajó la posibilidad de desarrollar una aplicación de mucho menor tamaño y hacerlo en ambos lenguajes para ver las diferencias entre ambos. Pero al final se acabaría pecando de igual forma en la simpleza de los ejemplos, por lo que se decidió hacer una aplicación de mayor tamaño y centrarse en las características propias de Kotlin dejando la comparativa solo al documento teórico.

Como primer acercamiento se decidió realizar una aplicación móvil únicamente para así centrarnos en las cualidades del lenguaje y poder profundizar un poco más en las propiedades nativas. Pero después se decidió hacer un esfuerzo extra por realizar un desarrollo completo en torno a una aplicación cliente-servidor de administración para los propietarios de las tiendas además de la aplicación móvil para los clientes y así poder realizar un proceso de ingeniería más completo y que los datos de la aplicación no estuvieran almacenados localmente.

Por lo tanto, el proyecto queda dividido en la aplicación nativa, programada en lenguaje Kotlin, a la cual accederá el usuario para ver las ofertas de las tiendas que le rodean, y una aplicación cliente-servidor que servirá como administración a los propietarios de dichas tiendas, programada en NodeJS y Angular 7.

Podemos empezar hablando sobre la **organización del trabajo** y como hemos ido llevando a cabo las tareas.

Para empezar, podemos hablar sobre la organización en el trabajo realizado. Como ya hemos dicho, las fases de trabajo serían:

- **Documento de comparación teórico**, que incluye: la preparación de investigación previa a la escritura de todos los temas a tratar y otras ayudas para abarcar de una mejor forma estos temas elegidos, la redacción del documento explicativo para ambos lenguajes en el que se desarrollará por cada tema la explicación en Kotlin y posteriormente la explicación para el lenguaje Java, y la implementación de los ejemplos aclarativos para la mejor compresión de los apartados con su previa codificación y su captura posterior.
- **Desarrollo de la aplicación**, que incluye: la especificación de requisitos, casos de uso y diagramas de flujo del sistema, el diseño y modelado de los datos, la preparación de las tecnologías y los recursos elegidos, la elaboración de los diagramas correspondientes y, finalmente, la implementación de la aplicación móvil y la plataforma de administración.

3.1 Desarrollo de la aplicación

Una vez aclaradas cuáles han sido las fases del trabajo, queda ver cuál ha sido la metodología para desarrollar esas fases. Para el **desarrollo de la aplicación** se ha aplicado la metodología ágil Scrum adaptada a trabajos "individuales". El desarrollo ágil basa sus fundamentos en el desarrollo iterativo e incremental, los requisitos y demás componentes del proyecto van variando en cantidad y complejidad. Scrum te

da las instrucciones para aplicar este tipo de desarrollo a tu proyecto y, en este caso, vamos a tener un backlog (lista) de tareas ordenadas por prioridad de la cual se van a ir extrayendo y desarrollando de mayor prioridad a menos, pudiendo quedar algunas de estas últimas incluso sin hacer en la iteración correspondiente.

Figura 89: Método MoSCoW

En el caso de este proyecto, se ha aplicado un método especializado para la extracción y priorización de requisitos: el **método MOSCOW** (14). Este método intenta dividir los requisitos en 4 partes principales: M (must) son requisitos indispensables para el proyecto sin los cuales no podría existir el mismo, S (should) son requisitos importantes pero que no se consideran vitales para el proyecto, C (could) son requisitos que no aportan tanto valor como los que están en las dos categorías anteriores pero que aun así representarían una buena incorporación al proyecto y, por último, W (wont o wish) son requisitos que no son considerados parte de la iteraciones por su poco valor y a los que se acudirán en caso de tener tiempo sobrante. Por lo tanto, si comenzamos una extracción de requisitos con este método, al terminar los tendremos priorizados en 4 ramas y posteriormente se extraerán las tareas de cada uno de estos.

Las tareas obtenidas de estos requisitos forman el backlog mencionado anteriormente, del cual se irán extrayendo tareas y siendo realizadas con la mayor brevedad posible.

Una vez acabados los requisitos-tareas de una de las listas, se han realizado reuniones para ver cómo abordar el siguiente bloque y si el realizado cumple con las características pensadas y especificadas en un principio.

3.2 Documento teórico

Por otra parte, para la redacción del **documento teórico** se iban acordando qué apartados tenían mayor prioridad dependiendo del peso programático, así como de las nuevas aportaciones del lenguaje Kotlin. Una vez acordados los puntos a tratar en la reunión, se redactaban con la información obtenida previa a la redacción y se implementaban los ejemplos junto a la explicación hasta la siguiente reunión.

En cuanto a la experiencia con este método ha sido buena. Al haberse aplicado con anterioridad a proyectos personales e inclusos de empresa, no ha habido problemas con la adaptación a un proyecto pequeño e individual. Las reuniones han transcurrido con normalidad y se han ido exponiendo los problemas que se han ido teniendo a lo largo del desarrollo con su correspondiente solución.

4

Especificación de requisitos, casos de uso y flujo del sistema

4.1 Método

Como hemos nombrado anteriormente, se ha implementado el método MOSCOW para la extracción de requisitos que se ha realizado en una reunión inicial con el tutor, explicándole al mismo previamente de qué iba el método para facilitar la extracción. Por lo que la tabla de requisitos ya priorizada quedaría de esta forma:

Roles:

- Usuario (utiliza la aplicación móvil para ver las ofertas)

- Propietario (utiliza una aplicación de administración para inscribir sus ofertas en la plataforma)

MÉTODO MOSCOW			
M (Must)	**S (Should)**	**C (Could)**	**W (Wish)**
El usuario podrá ver la vista previa de la cámara al iniciar la aplicación	Los datos visibles por los usuarios están almacenados en línea	El usuario podrá ver un tutorial al iniciar la aplicación como guía de uso	Se deberá redirigir con la información adecuada el tráfico aplicación-web de la empresa
El usuario podrá ver las ofertas ayudándose de la geolocalización	El propietario podrá iniciar sesión en la plataforma de administración	El usuario podrá iniciar sesión en la aplicación	El usuario podrá guardar las ofertas como favoritas
El usuario podrá navegar entre las ofertas de las diferentes tiendas	El propietario podrá cerrar su sesión en la plataforma	El usuario podrá cerrar sesión en la aplicación	El usuario podrá ver las ofertas guardadas en un apartado a parte de la aplicación
El usuario podrá ver una oferta en concreto y sus datos relacionados de una tienda	El propietario podrá gestionar sus tiendas (crear, ver, modificar y borrarlas)	El propietario podrá ver los datos de su empresa asociada y modificarlos	El propietario podrá registrarse en la plataforma
El usuario podrá navegar hacia el enlace externo de la oferta, que se encontrará en web de la tienda	El propietario podrá gestionar las ofertas asociadas a sus tiendas (crear, ver, modificar y borrarlas)	El propietario podrá cambiar la contraseña de su cuenta	

Los datos visibles por el usuario estarán almacenados localmente			

Tabla 1: Método MoSCoW aplicado

*** Nótese que hasta este punto no se dividen los requisitos en funcionales y no funcionales*

4.2 Requisitos funcionales

Una vez tenemos todos los requisitos extraídos y priorizados queda separarlos en requisitos funcionales y no funcionales. En primer lugar, vamos con los requisitos funcionales ya que la mayoría de los requisitos no funcionales derivarán de estos. Además, dividiremos los requisitos funcionales en los que han sido cubiertos para la exposición de este TFG y los que se dejarán para posteriores actualizaciones.

Requisitos funcionales cubiertos	Código
El usuario podrá ver la vista previa de la cámara al iniciar la aplicación	RF-1
El usuario podrá ver las ofertas ayudándose de la geolocalización	RF-2
El usuario podrá navegar entre las ofertas de las diferentes tiendas	RF-3
El usuario podrá ver una oferta en concreto y sus datos relacionados de una	RF-4
El usuario podrá navegar hacia el enlace externo de la oferta, que se encontrará en web de la tienda	RF-5
El propietario podrá iniciar sesión en la plataforma de administración	RF-6
El propietario podrá cerrar su sesión en la plataforma	RF-7
El propietario podrá gestionar sus tiendas (crear, ver, modificar y borrarlas)	RF-8
El propietario podrá gestionar las ofertas asociadas a sus tiendas (crear, ver, modificar y borrarlas)	RF-9

Tabla 2: Requisitos funcionales cubiertos

Vamos a pasar a definir cada uno de ellas detallando dependencias (tanto de requisitos funcionales como de no funcionales), descripción, prioridad en el método MOSCOW y algún comentario adicional si es necesario (15).

RF-1	El usuario podrá ver la vista previa de la cámara al iniciar la aplicación
Dependencias	RNF-1: El usuario tendrá que aceptar el acceso a la cámara RNF-2: El usuario tendrá que aceptar el acceso a la geolocalización
Descripción	Al iniciar la aplicación, y después de haber aceptado los permisos requeridos, el usuario verá la vista previa de la cámara con la que podrá ver las tiendas mediante la realidad aumentada.
Prioridad	M (Must)
Comentarios	
RF-2	El usuario podrá ver las ofertas ayudándose de la geolocalización
Dependencias	RNF-2: El usuario tendrá que aceptar el acceso a la geolocalización
Descripción	El usuario al situarse cerca de las coordenadas de una tienda, le saldrá un aviso sobre si quiere ver las ofertas a la que se está acercando.
Prioridad	M (Must)
Comentarios	

Tabla 3: Requisitos funcionales 1 y 2

RF-3	El usuario podrá navegar entre las ofertas de las diferentes tiendas
Dependencias	RF-2: El usuario podrá acceder a una tienda al estar cerca de su localización
Descripción	El usuario verá una lista con las diferentes ofertas de la tienda seleccionada con datos suficientes y podrá seleccionar cualquiera de ellas.
Prioridad	M (Must)
Comentarios	

Tabla 4: Requisito funcional 3

RF-4	El usuario podrá ver una oferta en concreto y sus datos relacionados de una tienda
Dependencias	RF-3: El usuario podrá navegar entre las ofertas de las diferentes tiendas

Descripción	El usuario al seleccionar la oferta verá todos los datos relacionados con esta, así como un botón para acceder al enlace externo de la oferta.
Prioridad	M (Must)
Comentarios	
RF-5	El usuario podrá navegar hacia el enlace externo de la oferta, que se encontrará en web de la tienda
Dependencias	RF-4: El usuario podrá ver una oferta en concreto y sus datos relacionados de una tienda
Descripción	El usuario al pinchar en el botón de compra será redireccionado externamente a la oferta que se encuentra en la página web de la tienda y podrá acceder a comprarla desde su plataforma.
Prioridad	M (Must)
Comentarios	

Tabla 5: Requisitos funcionales 4 y 5

RF-6	El propietario podrá iniciar sesión en la plataforma de administración
Dependencias	
Descripción	El usuario propietario de una tienda podrá iniciar sesión en una plataforma de administración habilitada para gestionar todos los datos referentes a la aplicación móvil.
Prioridad	S (Should)
Comentarios	

Tabla 6: Requisito funcional 6

RF-7	El propietario podrá cerrar su sesión en la plataforma
Dependencias	RF-6: El propietario podrá iniciar sesión en la plataforma de administración
Descripción	El usuario propietario de una tienda podrá cerrar la sesión iniciada con anterioridad en la plataforma de administración.
Prioridad	S (Should)
Comentarios	

Tabla 7: Requisito funcional 7

RF-8	El propietario podrá gestionar sus tiendas (crear, ver, modificar y borrarlas)
Dependencias	RF-6: El propietario podrá iniciar sesión en la plataforma de administración
Descripción	El usuario podrá crear tiendas asociadas a su marca, así como ver todos sus datos, modificarlos y borrar cualquier tienda. La operación de borrado también elimina todas las ofertas asociadas a esa tienda.
Prioridad	M (Must)
Comentarios	Finalmente, no se almacenarán imágenes (o enlace a imágenes) en la base de datos para las tiendas, por lo que la propiedad "logo" de la tienda queda anulada hasta una posterior iteración.

Tabla 8: Requisito funcional 8

RF-9	El propietario podrá gestionar las ofertas asociadas a sus tiendas (crear, ver, modificar y borrarlas)
Dependencias	RF-6: El propietario podrá iniciar sesión en la plataforma de administración RF-8: El propietario podrá gestionar sus tiendas (crear, ver, modificar y borrarlas)
Descripción	El usuario podrá crear ofertas asociadas a una tienda en propiedad, así como ver todos sus datos, modificarlos y borrar cualquier oferta.
Prioridad	S (Should)
Comentarios	Finalmente, no se almacenarán imágenes (o enlace a imágenes) en la base de datos para las ofertas, por lo que la propiedad "imagen" de la oferta queda sustituida por un enlace a la imagen de la oferta que tiene la marca en su página web propia.

Tabla 9: Requisito funcional 9

Ahora veremos qué requisitos se han quedado, por falta de tiempo y escasez de prioridad, fuera de nuestro proyecto actual pero que en un futuro serán candidatos para entrar:

Requisitos funcionales sin cubrir	Código
El usuario podrá ver un tutorial al iniciar la aplicación como guía de uso	RF-10
El usuario podrá iniciar sesión en la aplicación	RF-11
El usuario podrá cerrar sesión en la aplicación	RF-12
El usuario podrá guardar las ofertas como favoritas	RF-13
El usuario podrá ver las ofertas guardadas en un apartado a parte de la aplicación	RF-14
El propietario podrá ver los datos de su empresa asociada y modificarlos	RF-15
El propietario podrá cambiar la contraseña de su cuenta	RF-16
El propietario podrá registrarse en la plataforma	RF-17

Tabla 10: Requisitos funcionales no cubiertos

Vemos como todos los requisitos pertenecen a las iteraciones C (could) y W (wont o wish) que como comentamos en el apartado *Metodologías y fases del trabajo,* constan de requisitos que no son importantes para versiones del proyecto. Podemos decir, por lo tanto, que el método MOSCOW se ha realizado perfectamente y, aunque normalmente no pasa, ha especificado desde un principio correctamente las prioridades y especificación de los requisitos y esto se debe, en parte, a la poca complejidad y extensión del proyecto. En proyectos de mayor calibre, la prioridad de los requisitos suele variar y cambiar de una fase a otra. De hecho, es una de las cualidades del desarrollo ágil, la capacidad de cambiar en cada iteración del proyecto.

4.3 Requisitos no funcionales

Como se dijo anteriormente, ha habido requisitos no funcionales derivados de los funcionales. De hecho, están reflejados como dependencias en las tablas de los requisitos funcionales cubiertos. También se ha añadido un requisito no funcional derivado del propio proyecto (RNF-5) y otro expuesto por nosotros al cliente que requeriría de información de nuestra aplicación para datos estadísticos (RNF-6).

Requisitos no funcionales	Código
El usuario tendrá que aceptar el acceso a la cámara	RNF-1
El usuario tendrá que aceptar el acceso a la geolocalización	RNF-2
Los datos visibles por el usuario estarán almacenados localmente	RNF-3
Los datos visibles por los usuarios están almacenados en línea	RNF-4
La aplicación deberá desarrollarse en lenguaje nativo	RNF-5
Se deberá redirigir con la información adecuada el tráfico aplicación-web de la empresa	RNF-6

Tabla 11: Requisitos no funcionales

Los requisitos RNF-3 y RNF-4 son excluyentes el uno del otro. Cuando RNF-4 se complete, el requisito RNF-3 ya no será necesaria y quedará apartado del proyecto como un requisito obsoleto.

4.4 Casos de uso

De estos requisitos que hemos analizado, nacen los casos de uso del sistema. En primer lugar, analicemos los casos de uso asociados a la aplicación móvil de ofertas. Para ello hemos utilizado la herramienta MagicDraw que proporciona un potente creador de diagramas de todo tipo entre los que se encuentran los utilizados en este proyecto: diagrama de caso de uso, diagrama de secuencia y diagrama de clases.

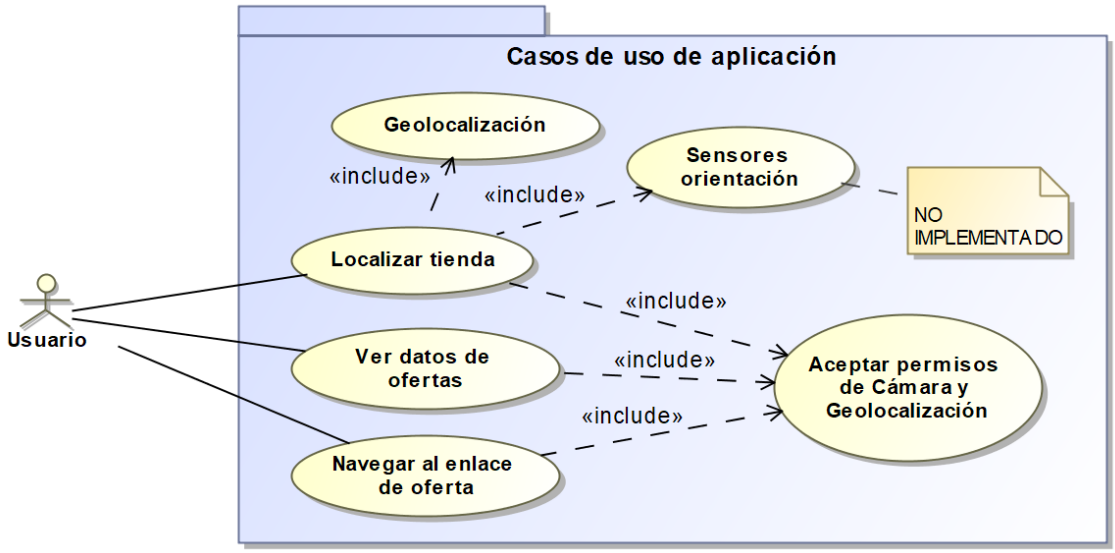

Figura 90: Diagrama de casos de uso de aplicación

Vemos cómo los requisitos principales de la aplicación (CU-1, CU-2 Y CU-3) dependen de la aceptación de permisos por parte del usuario (CU-4) y, en específico,

localizar una tienda depende de la geolocalización y de los sensores de orientación para una actualización futura (CU-5 y CU-6 respectivamente). Como único actor tenemos al usuario, cliente de las tiendas y participante de la aplicación sin necesidad de registro.

Ahora asociaremos los casos de uso representados con los requisitos del sistema:

Código	Casos de uso de aplicación	Requisitos
CU-1	Localizar tienda	RF-2
CU-2	Ver datos de las ofertas	RF-3 / RF-4
CU-3	Navegar al enlace de la oferta	RF-5
CU-4	Aceptar permisos de cámara y geolocalización	RNF-1 / RNF-2
CU-5	Geolocalización	RF-2
CU-6	Sensores de orientación	-

Tabla 12: Casos de uso de aplicación

Ya en el apartado de administración, el diagrama comprende a dos actores, aunque en la especificación de requisitos solo se tuvo en cuenta a uno, el empresario. El administrador general de la plataforma deberá hacerse cargo de la gestión de estos dos actores (usuario y empresa) en la plataforma.

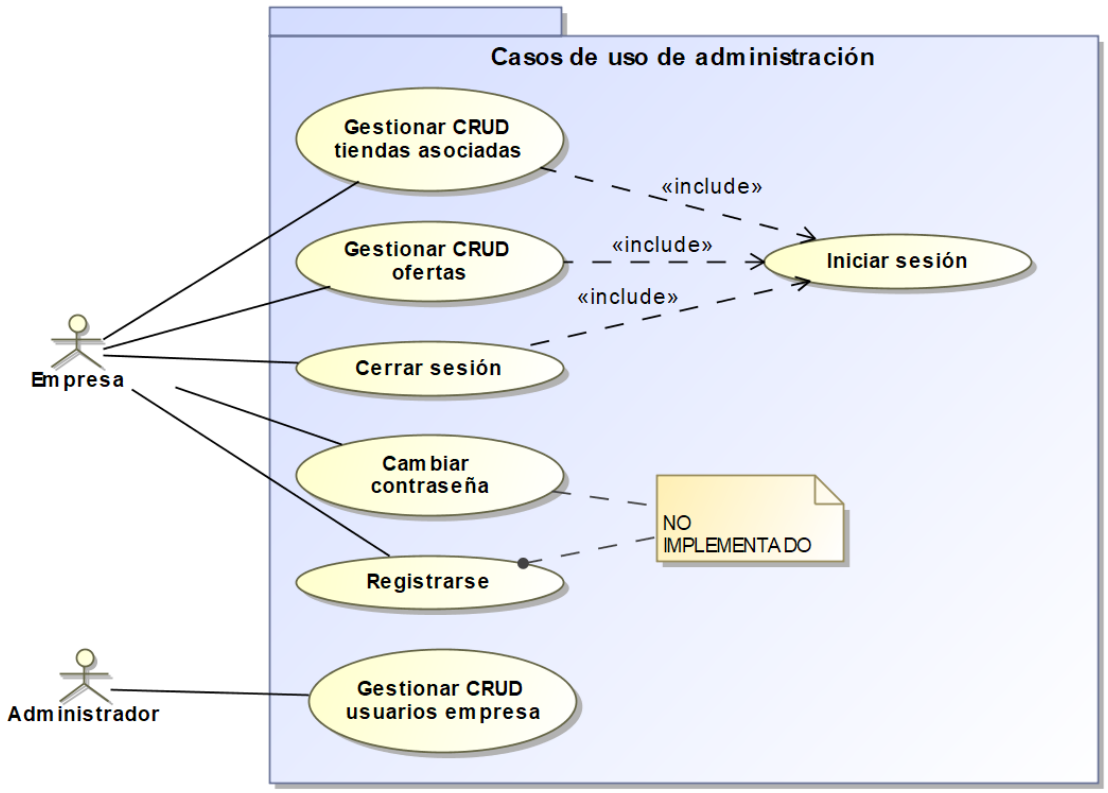

Figura 91: Diagrama de casos de uso de administración

Se aprecia como, de nuevo, los casos de uso principales de la plataforma de administración dependen de *Iniciar sesión* (CU-10). Asociaremos ahora los casos de uso mostrados arriba con los requisitos correspondientes. CU-13 quedará sin requisito asociado pues este no requiere de implementación, aunque si requiere de tiempo de desarrollo.

Código	Casos de uso de administración	Requisitos
CU-7	Gestionar CRUD de tiendas asociadas	RF-8
CU-8	Gestionar CRUD de ofertas	RF-9
CU-9	Cerrar sesión	RF-7
CU-10	Iniciar sesión	RF-6
CU-11	Cambiar contraseña	RF-16
CU-12	Registrarse	RF-17
CU-13	Gestionar CRUD de usuarios empresa	-

Tabla 13: Casos de uso de administración

4.5 Secuenciación y flujo del sistema

Al ser la aplicación móvil el principal motivo del desarrollo y dado que el panel de administración no deja de ser una aplicación cuyas bases de usabilidad ya están más que establecidas, vamos a llevar a cabo el diagrama de secuencia de la aplicación móvil, del cómo el usuario pasa de iniciar la aplicación por primera vez a ser redirigido a la oferta de forma externa por nuestra aplicación.

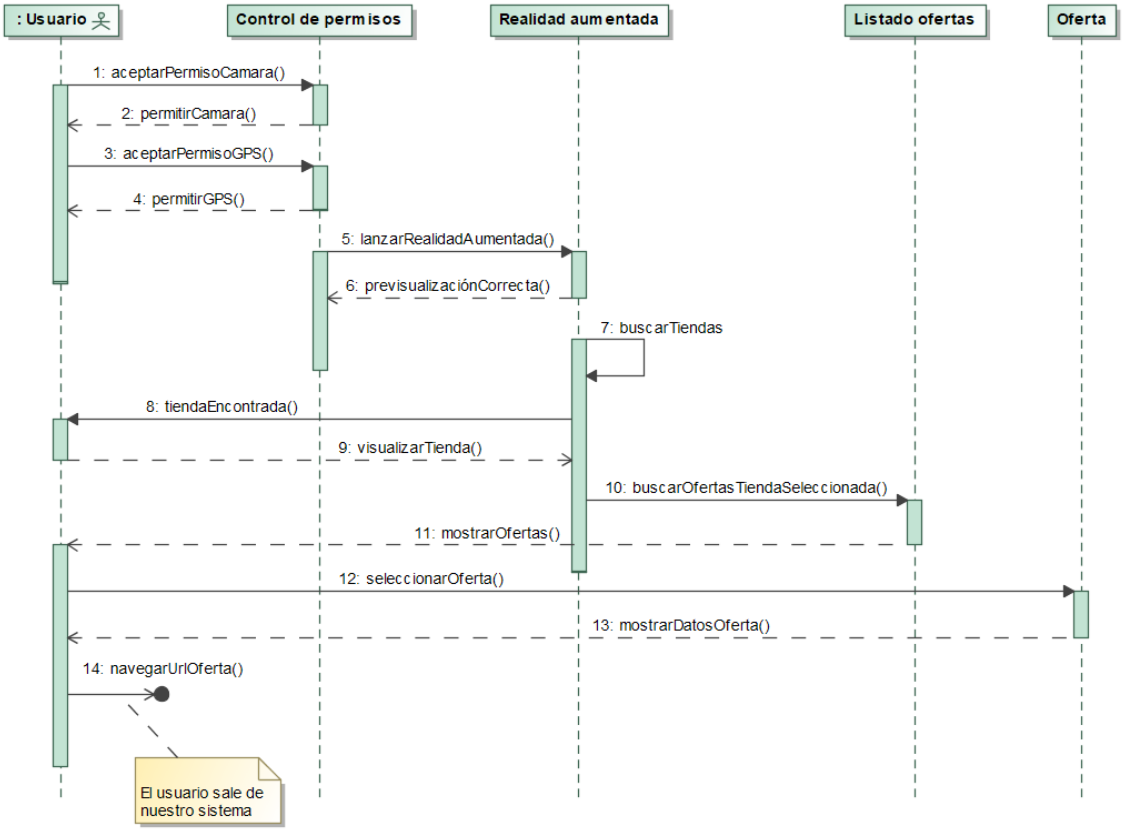

Figura 92: Diagrama de secuencia de aplicación

En primer lugar, el usuario debe aceptar los permisos de acceso a cámara y a geolocalización. En el diagrama no está contemplado el rechazo de estos pues la aplicación se cerraría ya que no puede trabajar sin estos permisos.

Posteriormente a esto la aplicación lanza la realidad aumentada y busca las tiendas cercanas al usuario, cuando la tienda es encontrada el usuario confirma la selección y visualiza las ofertas de la tienda seleccionada. Una vez mostradas las ofertas, el usuario señala una y se muestran todos sus datos, así como un botón con un enlace para navegar externamente hacia la página web de la tienda especificando la oferta seleccionada. En este punto perdemos el mensaje y la secuencia de acciones del usuario queda de manos de la aplicación web de la tienda asociada, saliendo así de nuestro sistema.

5

Diseño y modelado de datos

Comenzar con el análisis de datos diciendo que se ha utilizado una **base de datos relacional** y es por varios motivos: esta nos facilita un esquema exacto de los datos que vamos a almacenar, es el único formato de base de datos compatible con el framework utilizado para la implementación del servidor API y, el principal motivo, es más escalable para actualizaciones futuras.

Las bases de datos no relacionales suelen tener una finalidad más científica al no tener una estructura clara de los datos y almacenar todo tipo de datos sin filtro. Esto permite tener almacenado todos estos datos independientemente de su tipado y su origen, para su posterior análisis y procesamiento. No es nuestro caso.

Nosotros realizamos un análisis de datos previo y conocemos la estructura de nuestros datos y aportamos escalabilidad al proyecto tanto con el análisis como con la elección del tipo de base de datos.

Ahora se va a adjuntar el diagrama de clases implementado y vamos a pasar a explicar cada uno de los modelos, atributos y relaciones:

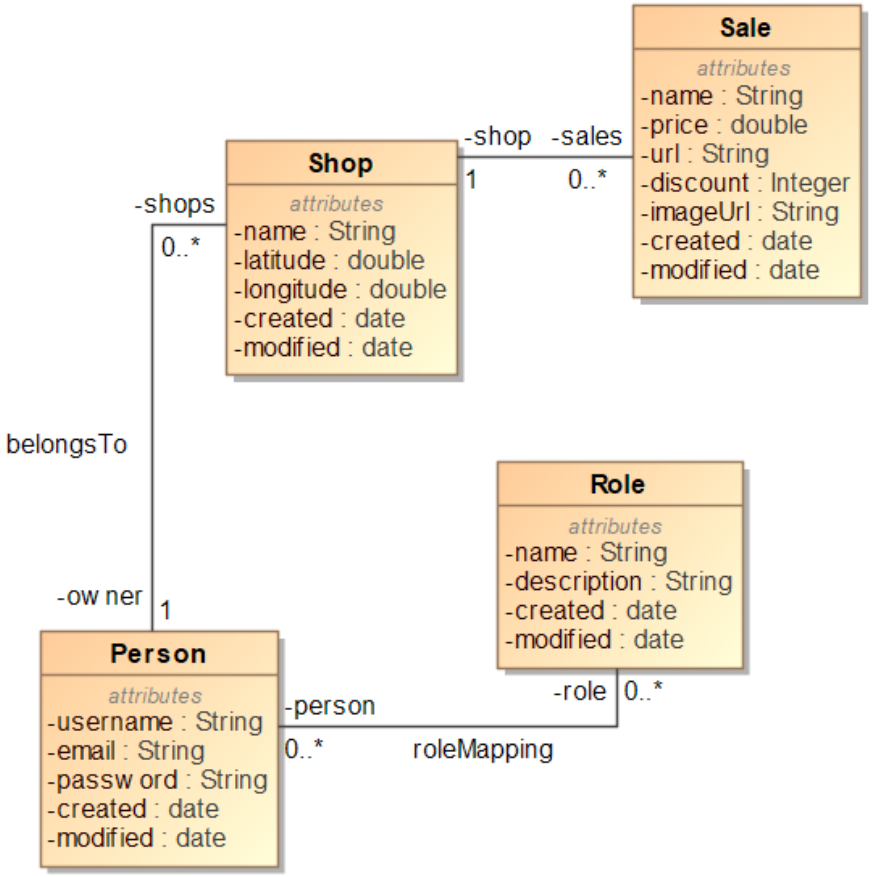

Figura 93: Diagrama de clases del sistema

*** Todos los modelos tienen un id único para identificar de forma inequívoca a cada instancia de la clase que será autoincrementable y que, por lo tanto, no es un dato manipulable por el cliente.*

Tengamos en cuenta en primer lugar al modelo **Sale** (oferta): cada oferta tiene su nombre, precio, descuento en forma de número que posteriormente, en la aplicación, mostraremos en forma de porcentaje, la url a la oferta original de la página web de la tienda donde se navegará de forma externa, la url de la imagen mostrada en la aplicación y dos fechas de actualización y creación de la instancia para ámbito estadístico o de información. Todos estos campos son introducidos, a excepción del mencionado *id* a pie de diagrama, por la empresa en el panel de administración y visualizados posteriormente por el usuario en la aplicación móvil.

Esta clase tiene, además, una relación de muchos a uno con la clase Shop (tienda): una oferta pertenece a una tienda y una tienda puede tener muchas ofertas.

El modelo **Shop** (tienda) tiene como atributos un nombre para ser identificada, una latitud y longitud con el objetivo de ser localizada por el usuario con la aplicación móvil y las fechas de actualización y creación ya mencionadas anteriormente. Estos atributos de geolocalización (latitud y longitud) son leídos por la aplicación móvil y son los que utiliza la función de comparación de coordenadas en Kotlin para comprobar que estamos cerca de la tienda en cuestión.

Así mismo, esta clase tiene una relación de pertenencia de muchos a uno con la clase Person (persona) la cual hace referencia a que una tienda pertenece a una persona y una persona puede ser propietaria de muchas tiendas.

No se almacenan datos de los usuarios de la aplicación móvil, pero los usuarios propietarios que manejen los datos de administración de las tiendas deben ser registrados en la plataforma y almacenados en la base de datos. Pasamos a hablar del modelo **Person** (persona) y sus datos: tiene atributos como el nombre de usuario para facilitar el inicio de sesión en la plataforma, el correo que será un atributo único en el modelo que utilizarán para iniciar sesión también, una contraseña para el mismo uso y las fechas de actualización y creación.

Por lo tanto, tenemos que las personas registradas en esta clase van a ser los propietarios de las tiendas. Antes de hablar de su relación con la clase Rol, se tiene que comentar que existe una tabla implementada internamente por el framework utilizado que gestionar los tókenes de acceso, por la cual una persona tiene relacionado un token de acceso que se almacena en base de datos con un id aleatorio al iniciar sesión una persona en la plataforma de administración (16). Con este token de acceso, el propietario podrá realizar las peticiones consideradas a la API sin ningún problema… o sí.

La clase Person (person) tiene relación con una clase Role (rol) mediante la cual el servidor sabe si la persona que realiza la petición está registrada en la plataforma y, sobre todo, tiene un rol de propietario. Por lo tanto, la instancia persona necesitará del token de acceso y de tener rol de propietario para tener acceso a las peticiones de administración de ofertas y tiendas. Así mismo, para gestionar las ofertas de una tienda, la persona tiene que ser propietaria de la tienda en cuestión.

Los usuarios no registrados y que hacen uso de la aplicación móvil, accederán a puntos de la API abiertos a usuarios no autenticados con el fin de que todo el mundo pueda retraer la información de las tiendas y ofertas de la base de datos sin restricciones.

Por último, el modelo Role (rol) que contiene el nombre del rol (en este caso, propietario), una posible descripción del rol y las fechas de actualización y creación. Como hemos dicho esta clase contendrá una relación con Person (persona) de muchos a muchos implementada con una tabla intermedia llamada RoleMapping, que no ilustramos en el diagrama. Tanto este modelo como el de AccessToken (token de acceso) son expuestos por el framework utilizado y es por ello que se ha utilizado, aunque ya se explicará con más detenimiento en el apartado de *Elección de tecnologías* (16).

6

Elección e implementación de tecnologías

Dividiendo en dos de nuevo el análisis, ahora de las tecnologías elegidas, nos queda en primer lugar y más importante la **aplicación móvil**. Con la aplicación móvil no se han tenido dudas. Desde un principio se quiso hacer en Kotlin porque de ello va el proyecto completo. En lo que si había cuestiones era en si hacer un desarrollo paralelo igual con el lenguaje Java.

Como ya dijimos en el apartado *Introducción al desarrollo de la aplicación,* se barajó la posibilidad inicialmente de desarrollar la aplicación en ambos lenguajes. Pero debido a que tampoco era el tema principal del proyecto y que no se le dedicarían tantas horas como para realizar un desarrollo lo suficientemente extenso como para comprobar diferencias prácticas entre ambos lenguajes, se decidió aportar un mayor número de ejemplos sencillos en el apartado teórico y dejar la aplicación móvil como desarrollo Kotlin únicamente.

Es por eso que esta parte del desarrollo tampoco requiere de mucho análisis pues la elección de este lenguaje para la aplicación formaba parte del propio proyecto y no ha habido nunca discusión sobre ello.

Destacando un poco por encima, la experiencia ha sido bastante más llevadera que programando con Java. La obtención de vistas del layout por ids es mucho más concisa y en general es un lenguaje mucho más limpio Gracias a la interoperabilidad con Java se ha aprovechado alguna que otra librería implementada en Java y el sistema de tipado de Kotlin ha resultado ser todo un descubrimiento en la programación Android.

Gracias a la programación nativa se han podido implementar de forma eficiente y controlada el gestor de permisos, tanto de cámara como de geolocalización, y, sobre todo, la vista de la cámara; con un lenguaje híbrido hubiera sido bastante tedioso y probablemente bastante más ineficiente. Te da bastante seguridad el acceder a librerías nativas donde sientes tener el control del más mínimo detalle.

Aun así, el análisis con las diferencias entre ambos lenguajes aplicados a la programación Android se encuentra en el documento teórico situado en el anexo.

Por otra parte, tenemos tanto el servidor API como la aplicación de administración en los cuales se ha utilizado el lenguaje JavaScript y en concreto TypeScript en la aplicación de administración, un JavaScript tipado con semejanzas a Java por tener clases, interfaces, etc. Desgranando un poco, en el **servidor API** como decimos se ha utilizado JavaScript, en concreto un entorno de ejecución de JavaScript orientada a eventos asíncronos llamado NodeJS (17). Al ser NodeJS un entorno de ejecución, se necesita un framework para la creación de un servidor API. Uno de los frameworks más conocidos para este tipo de uso es Express, te da las herramientas necesarias para poder crear una API desde cero y poder manipular totalmente la estructura de tu servidor. Pero no es nuestro caso. Finalmente se ha decidido utilizar Loopback, otro framework que al igual que Express, te facilita el poder crear tu API REST sin ningún problema. La ventaja de Loopback es que ya te proporciona una estructura base en la que está implementado un sistema de roles y tokens de acceso, así como estructuras para modelos de Usuario y gestión de contraseñas. También te facilita una interfaz para poder ver los métodos de tu API REST que utiliza Swagger y, por si fuera poco, te da todos los métodos básicos para cada modelo que crees desde cero: get, put, post, delete, patch, head… A todo esto, se le suma que tiene un CLI (o terminal) propio en el que podemos gestionar, con comandos específicos, la

creación y modificación de modelos y relaciones, roles y accesos a los métodos de los modelos, creación de nuevos métodos, modificación de los ya existentes, servidores proxy, etc.

Este framework nos proporciona escalabilidad y facilidad para la implementación de nuestra API. La documentación de Loopback es bastante extensa y es un software implementado por IBM, empresa de las más grandes y exitosas del sector por lo que tiene una gran comunidad y solución a los problemas que plantean los desarrolladores de la propia comunidad. En el proyecto se utiliza la versión 3 de este framework, pero actualmente está en producción la versión 4 que pasa a utilizar TypeScript y hace más robusta tu servidor API con el lenguaje tipado y más controlable con una nueva estructura basada en módulos.

Así pues, la elección de la base de datos es fácil una vez que se ha elegido el framework. Loopback tiene compatibilidad con bases de datos relacionales, a su vez tiene dos conectores potentes en este momento: PostgreSQL y MySQL. Por experiencia en el trato, se ha decidido utilizar MySQL que es una base de datos relacional que te proporciona un entorno de trabajo llamado MySQL Workbench con el que puedes tratar los datos con una interfaz gráfica más cómodamente. Aun así, y como se ha comentado ya en el apartado de *Diseño y modelado de datos*, se hubiera elegido una base de datos relacional por la escalabilidad que esta proporciona y por conocer en todo momento la estructura de los datos introducidos, leídos, modificados o borrados, cualidad que casa bastante bien con el lenguaje TypeScript que vamos a nombrar a continuación.

En la **aplicación de administración** hemos utilizado el framework Angular que desde su versión 2 utiliza TypeScript como lenguaje. Se ha desarrollado con Angular 7 que para el momento en el que se desarrolló la aplicación era la penúltima versión estable. Angular es un framework que te ofrece una terminal con comandos propios y que te da una estructura y una serie de librerías preestablecidas para poder crear tu aplicación mayor facilidad. Debido a lo denso que es, es complicado dominar al máximo todos y cada uno de sus aspectos. Aun así, se ha decidido tomar este framework por experiencia y porque es de los frameworks de creación de aplicaciones web más estables y que más favorecen la escalabilidad de todos los actuales, cuyos rivales pueden ser React o VueJS. Estos dos últimos utilizan JavaScript y el más parecido a la orientación que tiene Angular puede ser VueJs, ya que es un framework que te ofrece una serie de cualidades y estructuras que React al ser una librería de JavaScript no aporta. La escalabilidad de Angular es proporcionada principalmente por el lenguaje con el que se utiliza. TypeScript aporta el tipado de datos, interfaces y clases que son tres cualidades bastante importantes a la hora de llevar a cabo proyectos grandes para llevar un control de tus datos en todo momento. La estructuración y, a la vez, la versatilidad, ya que no excluye las cualidades de JavaScript, que te aporta Angular y TypeScript a la vez que la experiencia con el framework hace que haya sido el elegido como herramienta de desarrollo para la aplicación de administración.

Por último, comentar que, para la elaboración de los diagramas de casos de uso, secuencia y de clases se ha utilizado el lenguaje UML como lenguaje más conocido y utilizado actualmente con el software MagicDraw que, por experiencia, da un buen

nivel de detalles a los diagramas y aporta bastantes funciones que poder implementar.

7

Desarrollo de la aplicación por requisitos

Podemos comenzar con el desarrollo de la **aplicación móvil** Android en Kotlin.

El primer requisito a abarcar fue el de intentar mostrar la realidad aumentada en la aplicación asociado a **RF-1**, para eso utilizamos la vista previa de la cámara y la implementamos en una clase a parte que implementaba la interfaz SurfaceView, interfaz que permite crear vistas de todo tipo en tu aplicación. Se utilizaron ideas de algunas páginas web y todas estaban en lenguaje Java, por lo que tuvimos que implementar la traducción a Kotlin, problema que ha sido recurrente durante el desarrollo del proyecto. Ya hemos dicho que Java puede convivir perfectamente con Kotlin, pero la idea de la aplicación era de hacerla completamente en Kotlin.

Con esto finalizamos la implementación de la vista previa de la cámara añadiendo funciones de pausa y reanudación de la aplicación en la actividad principal para liberar la cámara a otras aplicaciones o que nuestra aplicación pueda obtenerla sin tener que arrancarla desde cero una y otra vez.

Lo siguiente a analizar es la implementación de los permisos de cámara y geolocalización. La gestión de permisos se ha llevado de forma síncrona: primero hemos lanzado el aviso de autorización para operar con la cámara, la cual hemos hecho que siempre se muestre en orientación vertical, y después el permiso para la localización. La gestión de estos permisos en móvil siempre es una implementación complicada debido a las numerosas posibles respuestas que puede dar el usuario (rechazar siempre, rechazar esta vez, permitir en uso, permitir en segundo plano…).

Figura 94: Permiso de cámara

Para el permiso de cámara asociado a **RNF-1**, si la respuesta era afirmativa, daba igual que fuera en primero o en segundo plano inclusivo pues liberamos la cámara cuando la aplicación no está en primer plano, procedíamos a lanzar la autorización de permiso de geolocalización. En caso negativo, cerramos la aplicación, pues nuestra aplicación no tiene sentido sin poder tener este acceso.

Figura 95: Permiso de geolocalización

Para el permiso de geolocalización asociado a **RNF-2**, tampoco nos importa que la respuesta afirmativa sea para primer plano o segundo plano inclusive, pues la aplicación intentará localizar las tiendas cuando pueda independientemente del plano en el que se encuentre la aplicación. Si es negativa, procederemos a mostrar la visión de la cámara, pero evidentemente no será localizada ninguna tienda y, en caso de que haya rechazado solo una vez la solicitud, se mostrará un aviso de que la localización es necesaria para el correcto funcionamiento de la aplicación. En este caso, si el usuario quiere seguir haciendo uso de la aplicación, deberá dar permisos a la aplicación a través de Ajustes. En cambio, si es afirmativa, mostraremos de igual forma la vista previa de la cámara y se formarán nuevamente dos posibilidades. La función GPS puede estar encendida o no,

en caso de que esté encendida se mostrará el mensaje predeterminado de Android para encender la ubicación precisa desde la aplicación y poder seguir funcionando, si se enciende la aplicación funcionará perfectamente y si no, el usuario deberá encenderla manualmente desde los Ajustes del teléfono.

Posterior a esto, obtenemos todas las tiendas de nuestro servidor mediante una llamada HTTP a nuestro servidor API al recurso '/tiendas'. Este recurso está abierto a cualquier llamada, no es necesaria autenticación para obtener las tiendas de la base de datos. Esto lo realizamos de forma asíncrona mediante una co-rutina. Aunque ya ha sido explicado en el documento teórico, la co-rutina va a ejecutar la llamada a la API REST mientras el usuario lleva a cabo la gestión de permisos asíncronamente, es decir, paralelamente.

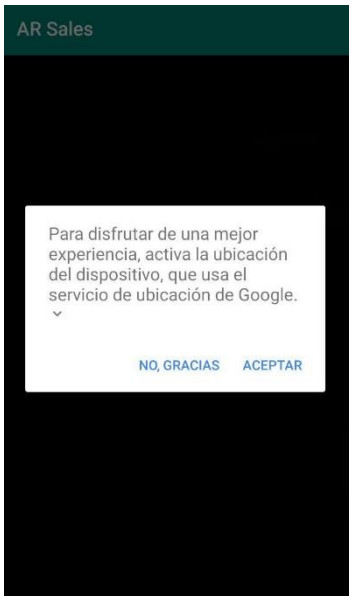

Figura 96: Geolocalización

Una vez obtenidos los datos del servidor, nos ayudamos de la geolocalización conseguida previamente para comparar latitud y longitud de las tiendas cercanas con latitud y longitud del dispositivo del usuario, asociado a **RF-2**. La comparación se realiza con el valor absoluto de la resta tanto de latitudes como de longitudes en decimales. Si esta resta es menor que 0.0003, que es el equivalente en decimales de 1 segundo, tanto en latitud como en longitud, es mostrado un mensaje con la tienda en cuestión para poder acceder si el usuario quiere a las ofertas de dicha tienda. En el caso de que se pulse 'CANCELAR', el cuadro de texto desaparecerá y se volverá a intentar identificar las tiendas cercanas con el mismo método. En caso de pulsar 'ACEPTAR', mostramos el listado de las ofertas de la tienda seleccionada lanzando la actividad con Intent.

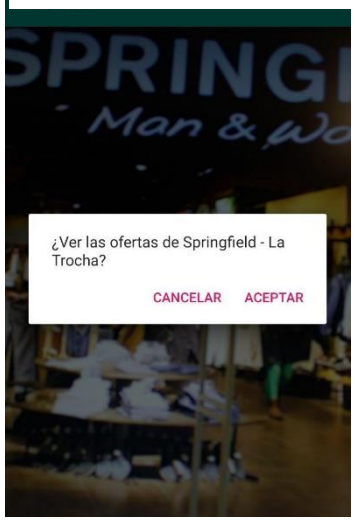

Figura 97: Tienda encontrada

Una vez seleccionamos la tienda en cuestión, como decíamos, mostramos todas las ofertas actuales en forma de lista con ListView de la tienda encontrada, asociado a **RF-3**. Podremos ver la información suficiente y podremos navegar entre ellas. Las ofertas son obtenidas mediante otra llamada a nuestro servidor API, esta vez al recurso '/tiendas/:id/ofertas', donde 'id' es el identificador único de la tienda anteriormente seleccionada y 'ofertas' las ofertas asociadas a la tienda en cuestión.

Figura 98: Ofertas

Una vez seleccionada una de las ofertas, veremos todos los detalles de esta en una nueva actividad lanzada y la imagen de esta en mayor tamaño. Esta llamada que se realiza al servidor para retraer toda la información de la oferta se ejerce mediante el recurso

‘/tiendas/:id/ofertas/:fk’, donde ahora ‘fk’ es el identificador único de la oferta seleccionada. Esta acción está asociada al requisito **RF-4**.

Figura 99: Oferta

Cuando el botón de compra es pulsado, la aplicación redirige su flujo al enlace externo de la oferta, introducido por el propietario mediante el panel de administración, por lo que el usuario será llevado a la web oficial de la tienda en su navegador predeterminado. Hecho esto nuestra aplicación para de obtener información acerca del usuario y el flujo queda de manos de la aplicación de la tienda. Esta acción está asociada al requisito **RF-5**.

Todas y cada una de estas acciones descritas han sido parte de una iteración posterior en la que se ha hecho uso de una base de datos en línea administrada por nuestro servidor. Previamente a esta versión, los datos eran retraídos de una base de datos SQLite la cual se implementaba con un manejador (DBHandler) que contenía todos los métodos necesarios que traducían a lenguaje SQL la llamada y con un creador de instancias y modelos (DBHelper) que creaba, ayudándose de la interfaz SQLiteOpenHelper los modelos e instancias necesarias para hacer las pruebas localmente.

Esta primera implementación local obedece al requisito no funcional **RNF-3** y la segunda implementación en línea se relaciona con **RNF-4**.

Pasaremos ahora a analizar el desarrollo del servidor API y de la **aplicación de administración**.

En cuanto al servidor API ha sido una implementación básica referente al framework Loopback en el que se han creado modelos y relaciones con la CLI propia. Se han habilitado los recursos de obtención de tiendas y ofertas para todo tipo de usuarios mediante el sistema de ACLs de Loopback especificado en el archivo JSON de cada modelo. Para los métodos de administración, se ha creado un rol de administración para que los propietarios puedan gestionar tiendas y ofertas propias. Por ello, estos recursos de administración pasan todos por el modelo Persona: se debe conocer el id de la persona y ver si es administrador, además de ver si es propietario de la tienda a la que está intentando acceder.

Por el lado de la aplicación de administración, en primer lugar, comentar que se ha implementado un inicio de sesión que en cuanto a servidor se ha utilizado los métodos ya implementados por Loopback para nuestro modelo Persona, que extiende del modelo Usuario predeterminado de Loopback el cual tiene ya todos los métodos necesarios para la autenticación de usuarios. Por el lado de cliente, en Angular se ha diseñado un inicio de sesión básico con un formulario “Reactive form” de Angular y se ha llamado al servidor mediante una petición HTTP al recurso ‘/login’ del modelo Persona. Se inicia sesión con el correo electrónico del propietario, el cual tiene un validador para que el usuario no introduzca valores que

no sean correo electrónico, y la contraseña, que puede ser visible al escribir. Esta acción queda asociada al requisito funcional **RF-6**.

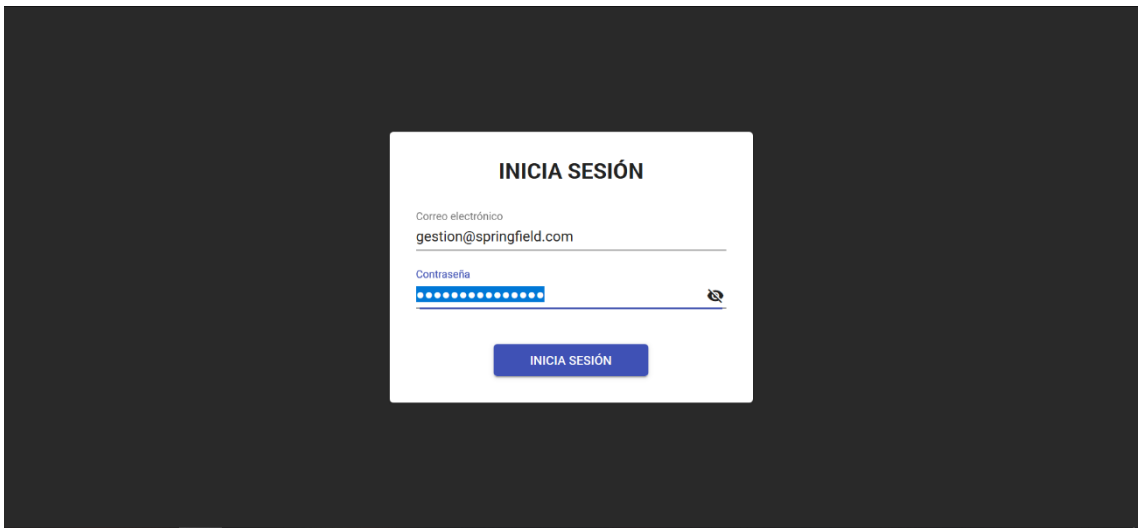

Figura 100: Inicio de sesión en administración

Ya dentro de la plataforma de administración podemos listar tanto tiendas como ofertas. En cuanto a las tiendas, listamos todas de las que el usuario que ha iniciado sesión es propietario llamando a la API con el recurso '/people/me/shops' donde 'me' es un literal que Loopback entiende como el id de la persona que realiza la petición pasándosele por cabecera HTTP ese id. Hecho esto, los datos se representan con una lista de Angular diseñada por Material Design incluyendo los datos básicos para su diferenciación.

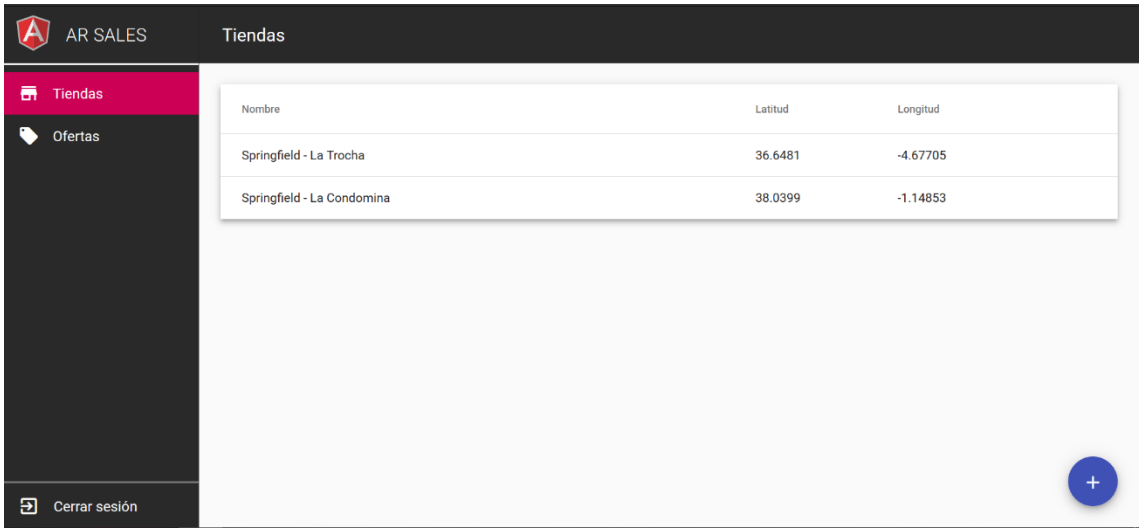

Figura 101: Administración de tiendas

Para listar las ofertas el diseño es el mismo y se quiere representar los datos para que sean lo suficientemente diferenciables en la tabla. Para listarlas se llamará al método '/people/me/sales' que trae todas las ofertas de las tiendas de las que es

propietario el usuario. Realmente la llamada atraviesa 3 tablas: la tabla Persona, desde donde se realiza la llamada, la tabla Tienda, por donde tiene que pasar para saber de qué tiendas es propietario, y la tabla Oferta.

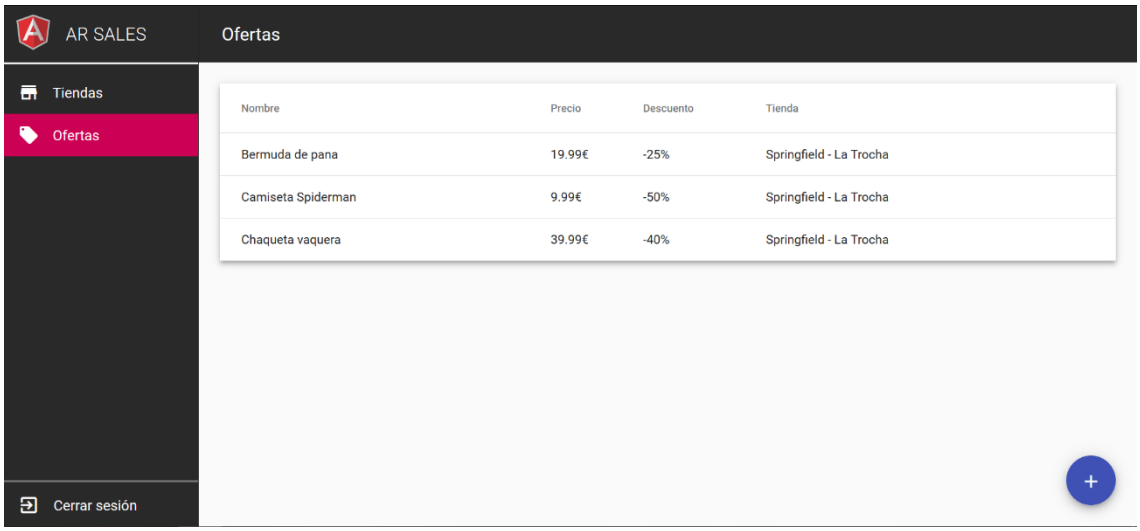

Figura 102: Administración de ofertas

En cuanto a la creación de ambas entidades, se pulsará el botón de debajo a la derecha y se introducirán los datos llamando al recurso POST '/people/me/shops' o '/people/me/sales' cuando proceda. Se utilizará un 'Reactive form' con validación de latitud y longitud personalizada.

La modificación se hará pulsando la entrada de la tabla que se quiera editar que nos llevará a un formulario igual que al de la adición de instancia. Sin embargo, en este estarán los datos de la tienda u oferta seleccionada ya introducidos y se podrán modificar. La llamada a la modificación es al recurso PUT '/people/me/shops' o '/people/me/sales' según proceda.

También se podrá borrar cada una de estas instancias dentro de la página de edición de tienda u oferta pulsando el botón Borrar. Para esto se hace una llamada HTTP de nuevo al recurso DELETE '/people/me/shops' o '/people/me/sales' y la instancia queda borrada de la base de datos.

Figura 103: Administración de tienda

Figura 104: Administración de oferta

Esta gestión tanto de tiendas como de ofertas quedan asociadas a los requisitos funcionales **RF-8** y **RF-9** respectivamente.

El último requisito por cubrir es el **RF-7**: podemos cerrar sesión de la plataforma en la parte de abajo a la izquierda, en la barra de navegación. Esto borrará el token creado por el inicio de sesión del almacenamiento local y redireccionará a la página de inicio de sesión vista previamente.

8

Actualizaciones futuras

Gracias al método MOSCOW, conocemos las posibles mejoras del sistema que fueron requeridas desde un principio. En este caso, habría mejores tanto para la aplicación como para el panel de administración. Podemos enumerar esos requisitos especificados de forma inicial ordenados por prioridad:

- El usuario podrá ver un tutorial al iniciar la aplicación como guía de uso
- El usuario podrá iniciar y cerrar sesión en la aplicación
- El propietario podrá ver los datos de su empresa asociada y modificarlos
- El propietario podrá cambiar la contraseña de su cuenta
- Se deberá redirigir con la información adecuada el tráfico aplicación-web de la empresa
- El usuario podrá guardar las ofertas como favoritas
- El usuario podrá ver las ofertas guardadas en un apartado a parte de la aplicación
- El propietario podrá registrarse en la plataforma

A parte de estas mejoras ya especificadas, podemos añadir otras más realizando una tormenta de ideas y filtrando algunas por su poco peso y valor dentro del sistema:

- Se mejorará el algoritmo de realidad aumentada para que sea más preciso y eficiente, siendo en eso en lo que se basa la aplicación
- Se aumentará la cantidad de información que obtenemos de las ofertas, aportando más fotos, tallaje y algunas valoraciones de clientes
- Se implementará una pasarela de pago para que se puedan comprar las ofertas desde la aplicación de forma rápida y segura
- Mejora del diseño para adaptarlo a cada tienda seleccionada
- El usuario podrá iniciar sesión con Google o Facebook en la aplicación

11

Conclusión

Acostumbrado a trabajar en grupos de desarrollo tanto en el ámbito estudiantil como en el laboral, este proyecto ha sido todo un reto al tener que afrontarlo de manera individual solo con la ayuda del tutor.

El ámbito más costoso ha sido sin duda el teórico. Aun sin ser un gurú de la programación nativa ni de Kotlin, he aprendido bastante de ellos a la hora de realizar esta comparación. He visto, hablando de manera más global, la diferencia entre dos lenguajes de distinta época independientemente de que estemos hablando de Kotlin y Java en específico porque, al final, las principales diferencias se concluyen en una diferenciación en la época de aparición de estos. Java nació mucho antes que Kotlin y eso puede notarse en prácticamente cualquier lenguaje al avanzar tan rápido la tecnología.

El trabajo de investigación realizado ha sido bastante fructífero y, acostumbrado a investigar para el desarrollo, esta forma de investigación teórica ha resultado ser un descubrimiento para mí y ser bastante satisfactorio. Si logras abarcar los grandes puntos de la programación, a la hora de ponerte a programar, no tendrás muchas dificultades complejas o de estructuración.

El documento no empezó a escribirse si no tras un extenso proceso de investigación y de estructuración para ver qué temas se iban a tratar, de qué forma y analizar lo más profundamente posible estos. Los ejemplos de código fueron un gran acierto pues me ayudaron incluso a mí mismo a entender ciertos puntos expuestos.

Decir que con este análisis he llegado a la conclusión que esperaba llegar desde que se me ocurrió este proyecto. ¿Estaría bien que cualquier desarrollador rompa un poco su monotonía y se abarque en un análisis de los nuevos lenguajes que llegan para así intentar mejorar su producto final? Está claro que sí, pero con estos dos lenguajes pienso que se acentúa aún más, que es donde quería llegar. Kotlin actualmente es un portento en el desarrollo nativo por los motivos ya expuestos, pero, sobre todo, como ya hemos dicho unas cuantas veces a lo largo del documento, lo es por no hacer de su competidor un enemigo, si no un aliado en el que apoyarse para crecer. La interoperabilidad es la clave de Kotlin y chapó para JetBrains.

Por otro lado, en el desarrollo de la aplicación de ejemplo, he asentado conocimientos y métodos que ya tenía para el análisis de requisitos por motivos laborales. La extracción de requisitos y la conversión a clases, casos de uso, secuencias y finalmente código, ha hecho que sea bastante fructífero y satisfactorio para mí todo el apartado previo a la programación: “ingeniería”.

En el desarrollo en sí, por parte de servidor y la aplicación de administración he de decir que al utilizar tecnologías ya asentadas por motivos laborales y de ocio, tampoco ha sido un desafío como si lo ha podido ser en algún momento la aplicación móvil. Aun así, he conseguido asentar conocimientos y he cambiado un poco la estructura del proyecto de administración a como lo hago usualmente. Siempre hay que mejorarse.

En el ámbito de la programación de la aplicación nativa, fue todo un desafío implementar la gestión de permisos y la visión de realidad aumentada. Sobre todo, la gestión de permisos fue bastante tedioso por venir de otros lenguajes híbridos como TypeScript y su framework Ionic, ya que estos te abstraen bastante de toda esta gestión. Finalmente te das cuenta del valor de toda esa implementación y la versatilidad que existe en el desarrollo nativo. Aunque estos frameworks híbridos

intentan aprovechar al máximo sus librerías y acercarse al máximo también a lo nativo, por experiencia en ambos lados debo decir que cuando haces uso de librerías nativas, estas librerías híbridas no te ofrecen muchas veces todo lo que querrías. Estas librerías, además, no las suelen hacer la propia empresa, si no desarrolladores de la comunidad que se vuelcan en el código libre y compartido.

Al fin y al cabo, si tu aplicación depende de acceso nativo en gran parte, te acabarás viendo en la necesidad de realizar desarrollos nativos diferenciados. Así que el poder de la programación nativa, evidentemente, sigue ahí.

12

Referencias

Referencias: sitios web

1. **ITU, International Telecommunications Union.** Mobile Phone Market Forecast - 2017. *Areppim.* [Online] 19 Diciembre 2018. [Cited: 16 Agosto 2019.] https://stats.areppim.com/stats/stats_mobilex2017.htm.

2. **Clement, J.** Percentage of all global web pages served to mobile phones from 2009 to 2018. *Statista.* [Online] 22 Julio 2019. [Cited: 16 Agosto 2019.] https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/.

3. **Wikipedia.** Kotlin (lenguaje de programación). *Wikipedia.* [En línea] 28 de Mayo de 2017. [Citado el: 16 de Agosto de 2019.] https://es.wikipedia.org/wiki/Kotlin_(lenguaje_de_programaci%C3%B3n).

4. **Kotlin.** Learn Kotlin - Documentación. *Kotlin.* [Online] 25 Septiembre 2018. https://kotlinlang.org/docs/reference/.

5. **Cimpanu, Catalin.** Kotlin Expected to Surpass Java as Android Default Programming Language for Apps. *Bleeping Computer.* [Online] 14 Octubre 2017. [Cited: 16 Agosto 2019.] https://www.bleepingcomputer.com/news/mobile/kotlin-expected-to-surpass-java-as-android-default-programming-language-for-apps/.

6. **Rusu, Dan.** Kotlin: A massive leap forward. *Pro Android Dev.* [En línea] 9 de Febrero de 2018. [Citado el: 16 de Agosto de 2019.] https://proandroiddev.com/kotlin-a-massive-leap-forward-78251531f616.

7. **Patro, NC.** Choose the best — Native App vs Hybrid App. *Code burst IO - Medium.* [Online] 26 Marzo 2018. [Cited: 22 Agosto 2019.] https://codeburst.io/native-app-or-hybrid-app-ca08e460df9.

8. **Engel, Joshua.** Are null pointer exceptions really that much of an issue? *Quora.* [Online] 16 Abril 2018. [Cited: 23 Agosto 2019.] https://www.quora.com/Are-null-pointer-exceptions-really-that-much-of-an-issue.

9. **Hoare, Tony.** Null References: The Billion Dollar Mistake. *InfoQ.* [Online] 25 Agosto 2009. [Cited: 23 Agosto 2019.] https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/.

10. **Kotlin.** Kotlin Tutorials - Documentación. *Kotlin.* [Online] 25 Septiembre 2018. [Cited: 20 Septiembre 2019.] https://kotlinlang.org/docs/tutorials/.

11. **Steendam, Lisa.** Parallel and Asynchronous Programming in Java 8. *Foreach.* [Online] 17 Abril 2018. [Cited: 21 Septiembre 2019.] https://www.foreach.be/blog/parallel-and-asynchronous-programming-in-java-8.

12. **Moskala, Marcin.** The character of Kotlin: Conciseness. *Blog Kotlin Academy - Medium.* [Online] 28 Febrero 2018. [Cited: 2 Octubre 2019.] https://blog.kotlin-academy.com/the-character-of-kotlin-conciseness-2c70bc141985.

13. **Leiva, Antonio.** Kotlin Android Extensions: Adiós al findViewById (KDA 04). *DevExperto.* [En línea] 8 de Octubre de 2016. [Citado el: 10 de Octubre de 2019.] https://devexperto.com/kotlin-android-extensions/.

14. **Korolev, Sergey.** MoSCoW Method: How to Make the Best of Prioritization. *Railsware.* [Online] 6 Marzo 2019. [Cited: 11 Noviembre 2019.] https://railsware.com/blog/moscow-prioritization/.

15. **Junta de Andalucía.** Atributos de los requisitos. *Marco de Desarrollo de la Junta de Andalucía.* [En línea] 12 de Abril de 2011. [Citado el: 11 de Noviembre de 2019.] http://www.juntadeandalucia.es/servicios/madeja/contenido/recurso/409.

16. **StrongLoop by IBM.** Making authenticated requests. *Loopback IO.* [Online] 29 Noviembre 2016. [Cited: 13 Noviembre 2019.] https://loopback.io/doc/en/lb3/Making-authenticated-requests.html.

17. **Nodejs.** Acerca de Node.js. *Nodejs.* [En línea] 23 de Julio de 2016. [Citado el: 13 de Noviembre de 2019.] https://nodejs.org/es/about/.

Apéndice A

Manual de Instalación

Requerimientos

Para poner en marcha el proyecto es necesario tener instalado el entorno de desarrollo **NodeJS** globalmente, el gestor de paquetes **NPM** y la CLI de **Angular**, que es instalada mediante NPM.

NodeJS se instala mediante un instalable que puede ser descargado de su propia página web, con esto automáticamente tendremos el gestor de paquetes NPM instalado también en nuestro ordenador.

Angular se instala mediante NPM utilizando el comando "npm install -g @angular/cli".

Una vez instalado tenemos el entorno listo, extraemos el zip de nuestro proyecto y abrimos una terminal en el directorio raíz.

Si queremos iniciar el servidor, tendremos que instalar los paquetes necesarios yéndonos previamente a la carpeta de nuestro servidor API. Para ello introducimos "cd sales-api && npm install". Hecho esto, iniciaremos nuestro servidor sin movernos de carpeta con el comando "node .".

Si queremos iniciar la aplicación de administración, tendremos que instalar los paquetes necesarios yéndonos previamente a la carpeta de nuestra aplicación. Para ello introducimos "cd sales-dashboard && npm install". Hecho esto, iniciaremos nuestra aplicación sin movernos de carpeta con el comando "ng serve".

Para la aplicación móvil, hará falta tener **Android Studio** instalado. Dentro de él, abrir el proyecto llamado sales-app, que es nuestro proyecto Android. Compilamos dentro del entorno de desarrollo y ejecutamos teniendo un dispositivo Android conectado previamente, con acceso a transferencia de archivos y con la depuración USB activada. Hecho esto, al ejecutar, la aplicación se ejecutará en el dispositivo Android. En su defecto, se puede utilizar un emulador instalado mediante Android Studio previamente.

Un dato **importante** es que tanto el ordenador donde se ejecuta el servidor como el dispositivo móvil donde se ejecuta la aplicación móvil deben estar conectados a la misma red. Además, se deberá conocer la IP de nuestro PC en esa red accediendo al

comando “ipconfig” en la terminal del ordenador y esa IP debe ser cambiada en el código de las 3 llamadas que se hacen a la API dentro de la aplicación móvil. Todas estas al principio de cada archivo y se encuentran en las actividades: MainActivity.kt, SalesActivity.kt y SaleActivity.kt. SOLO hay que cambiar la IP, no el puerto.