Escuela Politécnica Superior de Jaén

Grado en Ingeniería Informática

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

ANÁLISIS, DISEÑO Y DESARROLLO DE UNA APLICACIÓN MÓVIL EN ANDROID Y SU RENTABILIDAD

Alumno: Luis Navidad Cobo

Tutor: Prof. D. Francisco Mata Mata
Dpto: Departamento de Informática

Septiembre, 2018

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Francisco Mata Mata , tutor del Proyecto Fin de Carrera titulado: Análisis, diseño y desarrollo de una aplicación móvil en Android y su rentabilidad, que presenta Luis Navidad Cobo, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, 3 de Septiembre de 2018

El alumno: Los tutores:

Luis Navidad Cobo Francisco Mata Mata

Índice

1. Introducción

1.1. Motivación

Es difícil recordar alguna creación tecnológica que haya impactado más en nuestro día a día que el teléfono móvil. Desde los modelos más antiguos, que servían *sólo* para realizar llamadas y mandar mensajes de texto, hasta el actual Smartphone, que básicamente es un ordenador de bolsillo que podemos utilizar para prácticamente cualquier cosa.

Es por esto que millares de aplicaciones han salido al mercado diseñadas específicamente para ser ejecutadas en un dispositivo móvil. Como todos sabemos, dentro de los usuarios de Smartphone, el grupo más potente tanto en cantidad como en intensidad de uso, es la gente joven. Una mezcla de tiempo libre y de búsqueda continua de actividades de ocio tienen la culpa.

Es por esto por lo que, para este trabajo de fin de grado, he decidido crear una aplicación Android donde poder compartir y buscar fiestas y eventos de los alrededores. La aplicación está pensada para ser un punto de encuentro entre organizadores de eventos o dueños de locales y gente que busque algo de marcha. Una plataforma perfecta para el ocio en general, con un modelo de rentabilidad y financiación que sobre todo prima captación de usuarios.

Este trabajo de fin de grado contiene, además de la complejidad inherente al diseño e implementación de la aplicación, un detallado análisis de rentabilidad económica, basado en un sistema de pago por subscripciones donde el usuario final (el que entra a la aplicación para buscar fiestas) no tiene que invertir ni un céntimo. Es el usuario que publica y publicita sus fiestas o eventos el que asume el coste, siendo este menor cuanto más se pretenda invertir en la aplicación.

1.2. Objetivos

Los objetivos propuestos desde la Escuela politécnica superior de Jaén para el TFG elegido y desarrollado son los siguientes:

Elegir un ámbito donde desarrollar una aplicación móvil en Android en busca de su rentabilidad. Como se ha comentado brevemente en la introducción del presente documento y se ampliará en siguientes páginas, el ámbito de este TFG es el ocio orientado a eventos festivos.

Estudiar las modalidades o estrategias de marketing móvil para lograr rentabilizar la aplicación. Actualmente en el mercado existen múltiples maneras obtener beneficio económico de una aplicación, de las cuales se estudiaron varias (pago por descarga, publicidad, usuarios premium…). Finalmente se decidió utilizar un modelo de subscriptor-publicador, donde es el publicador el que invierte dinero para poder captar posibles clientes de los subscriptores. Se explicará más detalladamente durante el desarrollo de este documento.

Definir los requerimientos funcionales y no funcionales, así como la especificación de la aplicación móvil. El proceso de análisis y planificación de requisitos siempre es necesario en un desarrollo serio de software, aunque bien es cierto que como de costumbre fueron apareciendo nuevos requisitos y desapareciendo antiguos tras la planificación inicial (por aumentar y/o disminuir la funcionalidad prevista)

Realizar el análisis y diseño de la aplicación móvil. Dicho proceso consistió en una elección iterativa de la funcionalidad a implementar, funcionalidad que después que se dividió en vistas funcionales y finalmente, el diseño de las mismas.

Desarrollar un prototipo de la aplicación móvil

Redactar una memoria que recoja todo el trabajo desarrollado, así como los manuales de instalación y usuario.

1.3. Metodología

Como metodología de desarrollo de software se decidió utilizar **SCRUM**, una de las metodologías ágiles más utilizadas en el mundo laboral. Es algo complicado definir roles y describir tareas específicas en un proyecto de una sola persona, pero es cierto que se ha intentado desde el principio limitar las actuaciones a necesidades específicas que el proyecto tenía en cada fase.

En una **fase inicial**, se inició un proceso algo difuso en el tiempo de modelo de negocio. Se descartaron varias ideas que, si bien podían ser rentables, parecían poco novedosas e imaginativas. Tras un largo periodo de reflexión, se optó por un modelo de negocio dinámico, adaptado a las necesidades actuales y orientado al usuario más habitual.

Una vez se tenía decidido el propósito general de la aplicación, se prosiguió con la **acotación funcional** de la misma. Que cosas iba a hacer la aplicación y qué cosas se iban a quedar fuera, a veces por complejidad y otras por considerar que no aportaban nada positivo.

Decidido ya que tipo de aplicación se iba a desarrollar, se inició el proceso de **definición de *historias de usuario*** (en una metodología ágil, la unidad mínima funcional que nos permite añadir valor a la aplicación). Aunque lo ideal es que este proceso se realice al principio y se modifique lo mínimo posible, he de admitir que fue el proceso por el que más se iteró, ya que la funcionalidad objetivo de la aplicación fue aumentando y disminuyendo continuamente.

Con todo el trabajo de negocio y de análisis funcional hecho, sólo quedaba lo más difícil: **desarrollar la aplicación**. Usando Fibonacci, se estimaron las historias de usuario en relación al tiempo semanal que podía invertir yo, único desarrollador, cada semana en el proyecto. Una vez hecho esto, se asignaron historias de usuario en cada iteración o sprint (de 1 semana de duración) y se empezó a crear código.

1.4. Estructura del proyecto

En los siguientes apartados continuaremos con un análisis, diseño e implementación como podemos observar en la tabla de contenido donde además se incluyen los subapartados. Está desarrollado de forma secuencial, pero el desarrollo real del diseño y la implementación han sido de forma iterativa dado que hemos seguido una metodología SCRUM para un desarrollo ágil.

En el apartado 2 trataremos sobre el análisis donde daremos una propuesta de solución, seguida de la obtención de requisitos junto una planificación de tareas y el modelado del proyecto. Es decir, definiremos totalmente el qué debe hacer nuestro proyecto. A comentar que la planificación seguirá algunas técnicas de SCRUM.

El siguiente apartado, número 3, está dedicado al diseño donde se especificará el cómo se realizará, determinaremos la estructura y funcionamiento que tendrá nuestro software a partir de las herramientas obtenidas en el análisis. Este apartado se ha desarrollado de forma iterativa, por lo tanto, comentaremos los cambios que han podido surgir en su desarrollo.

En el apartado 4 encontraremos los detalles de la implementación. En él comentaremos los elementos que se incluyen en la implementación, cómo se relacionan y la función que cumplen. El siguiente paso es entrar en detalle de implementación donde hablaremos en profundidad sobre los mismos y describiremos las iteraciones en las que se han llevado a cabo.

Como último apartado, el apartado 5 daremos las conclusiones tanto del cumplimiento de los objetivos, como reflexiones personales. Sin olvidarnos, de las posibles mejoras tras su desarrollo.

Finalmente adjuntamos un anexo del manual de instalación y descripción de contenidos suministrados junto a la memoria, a los que llamaremos apéndice I y II correspondientemente.

2. Análisis

2.1. Análisis preliminar

Vivimos en el pleno auge de la tecnología. Nos rodea miremos a donde miremos: desde un frigorífico hasta prácticamente cualquier reloj de muñeca. Es un mundo cambiante donde, sin embargo, el núcleo mercado sigue estando en el mismo sitio que lo ha estado siempre: el ocio.

Aunque esto es prácticamente lo único que no ha cambiado. Por ejemplo, las formas de buscar visibilidad como marca no son las mismas, ni las estrategias de marketing están enfocadas en los mismos aspectos, ni por supuesto el tipo de ocio es el mismo hoy que hace, por ejemplo, 20 años.

Vivimos una época de continuo avance tecnológico, que además es bastante diverso: puedes encontrar aplicaciones para prácticamente cualquier cosa (compartir coche, pedir comida a domicilio, consultar tu cuenta bancaria…) pero aun así, el sector del ocio sigue estando disperso digitalmente hablando. No hay un solo punto de encuentro para decidir que se hará este fin de semana, en lugar de eso acudimos a varios a la vez.

Uno de los lugares más comunes a los que los internautas suelen acudir en búsqueda de un plan para el fin de semana son las redes sociales. No obstante, en las redes sociales no recibes toda la información disponible (sólo la de los organizadores que sigas) y, además, en un entorno saturado de notificaciones como este, suelen pasar desapercibidas.

Es aquí donde *KELOKE* llega con la intención de ocupar un hueco que existe en el mercado. Congregar todas las fiestas en un solo punto que puede ser consultado por cualquier persona en cualquier momento. Fiestas y eventos de todo tipo: desde chupitos gratis en el antro más olvidado de Jaén, hasta conciertos multitudinarios en escenarios espectaculares.

Así que se parte de un objetivo bastante claro, el de presentar un único punto de información, pero también se cuenta con una plataforma ideal: una aplicación móvil. Desde luego cuesta pensar en un medio mejor de distribuir un producto que en el propio móvil, ya que es un elemento prácticamente obligatorio en el bolsillo de cualquier miembro de nuestra sociedad.

Como todos sabemos existen varios sistemas operativos a elegir, además de la opción de una aplicación híbrida, pero según numerosos estudios [1] el sistema operativo dominante es Android, superando (aunque no por mucho, hay que decirlo) a IOs. Además de ser el sistema operativo más utilizado, es la plataforma de desarrollo con más comunidad de desarrolladores y, por lo tanto, con más soporte, lo tiene mucho peso a la hora de embarcarse en un proyecto con vistas a estar en producción. El lenguaje utilizado es una adaptación de Java a las librerías propias de Google para Android, con un paradigma orientado a eventos asíncronos, por lo cual la curva de aprendizaje es relativamente pequeña para un estudiante de este grado.

La opción de una aplicación híbrida fue estudiada y descartada por dos principales motivos: tiene una penalización bastante fuerte en rendimiento, y lo más importante, añade dificultad extra al tener que combinar varios sistemas interconectados, desarrollados en distintos lenguajes y frameworks. Lenguajes y frameworks que, en lo personal, ni domino ni conozco en profundidad.

2.2. Propuesta de Solución.

Esta aplicación se divide a su vez en 3 pequeños sistemas que la componen. No todos se localizan físicamente en el mismo sitio, aprovechando así el hecho de tener computación distribuida en cliente y servidor.

El primero de los elementos es la aplicación cliente, es decir, la propia aplicación Android. Instalada en el dispositivo móvil del usuario final, será la que compute la lógica de negocio de la aplicación, evitando así saturar el servidor. Como la lógica de negocio es bastante sencilla, no hay que temer por temas de rendimiento o de batería. Si la aplicación creciese de manera que se necesitase de más recursos para poder funcionar correctamente (teniendo en cuenta tiempos de espera y gasto de energía), podríamos plantear separar la lógica cliente del cómputo, pero no es el caso. En esta aplicación se utiliza Android como lenguaje, el paradigma es orientado a eventos y se utiliza de manera repetida las librerías de Google que nos posibilitan cosas bastante útiles como tener mapas en la aplicación.

Para comunicarnos con el exterior lo hacemos a través de mensajes REST a una API que se encarga de procesarlos, comunicarse con el backend y devolver una respuesta en JSON, todo esto ya en el lado del servidor. Esta API es un Spring Boot con uso de JPA. Está desarrollada en Java 8 (utilizando funcionalidades de Java 8 como Streams y expresiones lambdas) y utiliza plugins muy conocidos entre programadores como Lombok.

La API utiliza Spring Framework por innumerables motivos: es el framework MVC más utilizado en Java, inversión e inyección de dependencias, simplicidad a la hora de generar código con anotaciones… Además, se decidió usar REST sobre SOAP no sólo por ser la tecnología predominante en la comunicación entre sistemas (lo cual te asegura más soporte de la comunidad), si no por no tener un contrato preestablecido como tenía SOAP, y, aún más importante, por no tener estado, lo que nos permite un escalado horizontal sólo limitado por el número de máquinas que tengamos en producción. A su vez, utilizamos JSON como representación de datos sobre XML por la libertad que da JSON respecto a los parámetros, por simplicidad de presentación y por supuesto, por ser el predominante en el mercado actual.

Por último, esta API se conecta a una base de datos que guarda el modelo y el estado de la aplicación. Ya que es un simple prototipo, se ha elegido usar H2 en vez de cualquier otro tipo de base de datos por la simplicidad. H2 es una base en memoria (es decir, no guarda estado entre ejecuciones) muy utilizada en fases de desarrollo y pruebas, pero que desde luego no te permite ir a producción al carecer de persistencia real en el tiempo. Habría que tener en cuenta esto si se decide avanzar con este prototipo.

2.3 Requisitos del Sistema

Cabe aquí comentar que normalmente para recoger los requisitos del sistema de una forma más efectiva, lo más adecuado sería un proceso intenso pero necesario de comunicación directa con el cliente final: qué es exactamente lo que se quiere, qué es lo que no se quiere, limitaciones del software, conversaciones frecuentes, entrevistas y presentación de prototipos funcionales y de diseño de manera incremental hasta que cliente y equipo de desarrollo tengan muy claro qué es lo que se debe obtener tras el desarrollo.

No obstante, al ser el cliente y el equipo de desarrollo la misma persona, este proceso no ha sido necesario. Además del mundo de la informática, también me considero un amante de la noche, la fiesta y la juerga en general. Respecto a personas jóvenes y estudiantes, creo saber perfectamente qué información quieren, en qué formato la quieren y cómo debe ser presentada. Y si me equivoco siempre puedo achacarlo a que soy desarrollador y no analista funcional.

2.3.1 Requisitos Funcionales

Como la aplicación está claramente dividida funcionalmente en dos: usuario normal y publicador, creo que lo mejor será hacer una división de requisitos atendiendo dicha característica:

Como publicador:

- El sistema debe permitir al usuario registrarse con sus datos.
- El sistema debe ser capaz de permitir realizar login a un usuario previamente registrado.
- El sistema debe permitir consultar mis datos como publicador (cantidad de fiestas disponibles, fiestas por mes, etc.)
- El sistema debe permitir crear nuevas fiestas y que estas se puedan ver por cualquier usuario.
- El sistema debe permitir editar una fiesta ya creada, y dicha actualización será visible por cualquier usuario.
- El sistema debe permitir eliminar una fiesta ya creada.

Como usuario final:

- El sistema debe proporcionar filtrado de fiestas de distintos tipos:
  - Por ciudad: se introduce una localidad y se muestran las fiestas en esa localidad.
  - Por distancia: se introduce una cantidad de metros/kilómetros y se muestran las fiestas que están en ese radio.
  - Por geolocalización: el móvil nos facilita la ciudad por geolocalización y se muestran fiestas de esa localidad.
- El sistema debe permitir la correcta visualización de los datos de una fiesta específica: nombre, fecha, mapa, descripción…

Además de todos estos, la aplicación debe permitir los idiomas Inglés y Español.

2.3.1 Requisitos no Funcionales

Al ser una aplicación móvil, los requisitos no funcionales son muy parecidos a cualquier otra aplicación de estas características, a saber:

- No concentrar todo el cómputo en el cliente: aprovechar el paradigma cliente-servidor para no saturar el dispositivo.
- Comunicaciones rápidas y eficaces, a tiempo real, que consigan una experiencia de usuario positiva.
- Alta disponibilidad de los sistemas en servidor: no es aceptable un retardo en la conexión demasiado alto ni periodos de indisponibilidad.
- La interfaz debe seguir los principios de adaptabilidad para una mejor visualización en general.
- La validación de los datos de entrada es obligatoria en absolutamente toda interfaz a la que el usuario final tenga acceso.

2.4 Planificación inicial de tareas – Historias de Usuario

Al elegir para el desarrollo del proyecto una tecnología ágil como SCRUM, la planificación de tareas se ha resuelto mediante la elaboración y desarrollo de historias de usuario.

Se ha elegido una metodología ágil por encima de un desarrollo en cascada tradicional por varios motivos. Es fácil distinguir aspectos de este proyecto que puedan inclinarnos por utilizar una metodología ágil: continuo cambio de requisitos y requerimientos según el sistema evoluciona, equipo de trabajo de pocas personas y con horarios bastante indefinidos en principio, necesidad de adaptación al cambio permanente… pero me gusta verlo de otra forma: no hay absolutamente ningún motivo para utilizar un modelo tradicional.

Desde hace ya algún tiempo se ha demostrado (no sólo con teorías, si no con datos) que las metodologías tradicionales no están preparadas para un mundo cambiante como el actual, y que cualquier tipo de metodología ágil es más eficiente y eficaz tanto para cliente como para equipo de desarrollo. Como podemos ver en algunos artículos [2] ya en 2017 la mayoría de los equipos de desarrollo del mundo habían abrazado metodologías ágiles.

Pero digo más, esta vez abalado por mi trayectoria profesional. Sin duda alguna el mayor problema que presentan las tecnologías ágiles es la migración desde una metodología tradicional. Todos los actores suelen mostrar resistencia a este cambio, sobre todo por no entender las implicaciones del mismo, y puede llegar a crear situaciones incómodas e incluso discusiones feroces que destruyen equipos.

Pero por suerte yo sólo he trabajado con metodologías ágiles: ni siquiera me planteo volver atrás y trabajar con waterfall (o desarrollo en cascada). Estoy totalmente adaptado a este modelo de trabajo y es en el que más cómodo me siento. Por lo tanto, la mayor desventaja desaparece por completo e incluso se convierte en una razón de más para utilizar agile.

Pasemos pues a listar las historias de usuario definidas inicialmente, junto con la estimación que el equipo de desarrollo dio en su momento:

Historia de usuario	Criterios de aceptación	Puntuación
Como usuario **Quiero** poder ver una lista de fiestas en la página principal **De manera** que vea las fiestas filtradas por un criterio concreto	1) Debo visualizar una tabla con fiestas 2) Es posible hacer scroll en la tabla	5
Como usuario **Quiero** poder filtrar las fiestas que veo por distancia **De manera** que vea sólo las que estén a menos de la distancia introducida	1) No deberá verse ninguna fiesta que esté más lejos de la distancia introducida 2) Se deberán visualizar todas las fiestas dentro del rango seleccionado 3) En la pantalla principal podrá verse la distancia que se usa para filtrar	13

Como usuario **Quiero** poder filtrar las fiestas que veo por ciudad **De manera** que vea sólo las que se celebran en la ciudad seleccionada	1) Cualquier ciudad del mundo debe poder ser elegible por este filtro concreto. 2) Se mostrarán todas las fiestas de dicha ciudad que hayan sido publicadas 3) No se mostrarán fiestas de otras ciudades.	13
Como usuario **Quiero** poder filtrar las fiestas que veo por geolocalización **De manera** que vea sólo las que se celebran en la ciudad donde esté mi dispositivo	1) El sistema de geolocalización debe reconocer la posición del usuario 2) Tras reconocer la localización, se debe hacer una traducción a la ciudad y cumplir con el filtro por ciudad	13
Como usuario **Quiero** poder ver la aplicación es español **De manera** que todos los textos se vean en dicho idioma	1) Sólo se utilizará español en la aplicación	5
Como usuario **Quiero** poder ver la aplicación es inglés **De manera** que todos los textos se vean en dicho idioma	1) Sólo se utilizará inglés en la aplicación	5

Como usuario **Quiero** poder registrarme en la aplicación. **De manera** que al final del proceso obtengo un usuario de tipo publicador, con una subscripción concreta asignada a mi cuenta.	1) El mismo usuario no puede registrarse dos veces. Para comprobar que dos usuarios son el mismo, nos fijaremos en el nick. 2) El usuario debe rellenar todos los campos para completar el registro 3) El registro tendrá efecto en el sistema servidor,	8
Como usuario **Quiero** poder logearme en la aplicación. **De manera** que si tengo un usuario previamente registrado, pueda acceder a la pantalla de publicador	1) El login se realizará mediante usuario y contraseña 2) Se debe introducir la misma contraseña que al registrarse 3) Tras un login correcto se redireccionará a la pantalla de publicador 4) Tras un login fallido un mensaje de error aparecerá	8
Como usuario publicador **Quiero** poder visualizar mis datos en la pantalla de publicador **De manera** que sea visibles mis fiestas actuales, así como los datos generales de mi subscripción y opciones como publicador.	1) El publicador podrá hacer logout 2) El publicador podrá comprar fiestas 3) El publicador puede ver cuantas fiestas tiene en total y por mes 4) Las fiestas mostradas en formato tabla serán sólo las relativas al publicador	21

Como usuario publicador **Quiero** poder comprar más fiestas **De manera** que se añadan a mi cómputo de fiestas.	1) Habrá una estrategia de márketing detrás del precio y cantidad de fiestas. 2) Las fiestas se añaden inmediatamente al total.	5
Como usuario publicador **Quiero** poder editar una fiesta ya creada **De manera** que los cambios se hagan efectivos tanto para mi como para el resto de usuarios	1) Podrá editarse cualquier información de la fiesta 2) Habrá un botón para cancelar los cambios y volver al estado inicial 3) Este editado tendrá su reflejo en base de datos	8
Como usuario publicador **Quiero** poder borrar una fiesta ya creada **De manera** que la fiesta desaparece y ningún usuario será capaz de visualizarla más.	1) El borrar una fiesta no modifica la cantidad de fiestas de las que se dispone. 2) Este borrado tendrá su reflejo en base de datos	5
Como usuario **Quiero** poder ver los detalles de una fiesta clicando en ella **De manera** que se navegue a una nueva pantalla con los datos de la fiesta.	1) Se visualizará el nombre, fecha, lugar, descripción y foto de la fiesta 2) Esta opción estará disponible desde cualquier sitio donde se muestre una fiesta en la aplicación	21

2.4 Planificación inicial de tareas – Planificación de los Sprints

Como sabemos, en SCRUM la puntuación de tareas puede realizarse siguiendo los criterios numéricos que desee el equipo, ya que este es un mero ejercicio de estimación de esfuerzo. Es decir, la puntuación que yo he asignado a las tareas podría ser la establecida en el punto anterior, o con números más grandes o más pequeños: al final no importa si ajustamos de igual manera la velocidad del equipo por iteración.

Se ha elegido una duración de iteración bastante larga, ya que de haber elegido una iteración demasiado corta prácticamente ninguna de las historias de usuario hubiera cabido en una iteración, lo cual nos hubiera obligado a dividirlas en historias de usuario más pequeñas. Esto se hace para poder confiar en que al final de cada iteración el producto ha sufrido una mejora incremental, por pequeña que esta sea, en vez de asumir una historia de usuario demasiado grande que con toda seguridad no podrá ser realizada en un solo sprint. De haber contado con un equipo ya experimentado en Android posiblemente las iteraciones hubieran sido más cortas.

A toda la planificación inicial hay que añadir desviaciones constantes en la velocidad del equipo de trabajo. Estas desviaciones se deben a dos principales motivos: primero a una mala planificación inicial al carecer de conocimientos técnicos suficientes en Android como para discernir lo que es realmente costoso de lo que es trivial; y segundo el aumento de velocidad natural que experimenta un equipo con el tiempo (por adquisición de conocimiento, experiencia, agilidad a la hora del desarrollo, etc.)

Por lo tanto, se han elegido iteraciones de 4 semanas, en las cuales se estima una velocidad del equipo de desarrollo de 25 puntos. Si hacemos la suma de los puntos de historia que tenemos en el backlog, tenemos un total de 130 puntos, es decir algo más de 5 iteraciones. Se decidió planificarlo en 6 iteraciones para poder estar preparado antes posibles contratiempos. En total serán algo menos de 6 meses de trabajo, planificados de la siguiente manera:

Iteración	Historias de usuario	Puntos totales
Iteración 1	Fiestas en principal Filtro por distancia	18
Iteración 2	Filtro por ciudad Filtro por geolocalización	26
Iteración 3	Español Inglés Registro de usuario Login de usuario	26
Iteración 4	Pantalla de publicador	21
Iteración 5	Comprar fiestas Editar fiesta Borrar fiesta	18
Iteración 6	Detalles de una fiesta	21

2.6 Estudio de viabilidad

A la hora de embarcarse en un proyecto de cualquier tipo, conviene realizar previamente un estudio de viabilidad, comparando costes y posibles beneficios para poder discernir si el proyecto es viable o no. La correcta realización de dicho estudio, sabiendo estimar los costes con precisión y los posibles beneficios con algo de prudencia, es muchas veces el punto clave para el éxito o fracaso del proyecto. Es por esto que se ha realizado un estudio sencillo, pero exhaustivo, teniendo en cuenta todos los costes reales que vamos a tener. Incluso se ha tenido en cuenta ciertos parámetros, como el sueldo del desarrollador, que realmente no son necesarios ya que yo voy a desarrollar la aplicación de manera gratuita. Más vale prevenir que curar.

2.6.1 Gastos de Hardware

Para la realización de este proyecto se van a utilizar dos ordenadores distintos: Un ordenador de sobremesa valorado en 800€, y un ordenador portátil valorado en 960€. Además, se utilizará un monitor amplio valorado en 150€. Por último, un teclado valorado en 60€ y dos ratones, uno valorado en 40€ y el otro en 10€. Por lo tanto, el desembolse inicial en gastos asociados al hardware asciende a una suma de 1.960€.

2.6.2 Gastos de licencias

Absolutamente todo el software utilizado para la realización del presente trabajo de fin de grado es bajo licencia open source, por lo cual habría que estimar un gasto en licencias de 0€ en total.

2.6.3 Gastos en servicios externos

Aquí podemos tener en cuenta dos servicios externos imprescindibles para la realización del proyecto: Una conexión a internet, a 40€ el mes (con una duración estimada de 6 meses de desarrollo) y un gasto energético, que hemos calculado en 0,26€ por hora. La estimación en horas de los 6 meses de trabajo es de unas 60 horas/mes, por lo que, sumando el gasto de internet, podemos estimar un gasto en servicios externos de 333,60€ en total.

2.6.4 Gastos de transporte

Prácticamente la entera realización del trabajo de fin de grado se realizará bajo tutoría online, ya que el equipo de desarrollo se encuentra en Madrid y el tutor en Jaén. Es por esto que no cabe destacar ningún tipo de gasto en transporte, 0€ en total.

2.6.5 Gastos en personal

Entramos por fin en el verdadero núcleo de gasto del proyecto. Al ser una aplicación relativamente sencilla, podríamos decir que, con un mes de trabajo de un analista funcional, que desglose los requisitos del proyecto y elabore las historias de usuario correspondientes, añadidos a los 6 meses estimados de desarrollo con un solo programador, tendríamos todos los gastos de personal cubiertos. Según la tabla salarial del convenio colectivo de Telefónica, un analista funcional tiene un salario mensual de 2555.29€/mes, y un programador Senior 2166.79€/mes. Cabe destacar que estos datos son totalmente discutibles y que el salario de sendos roles varía muchísimo en función de factores demográficos y sociológicos. Por ejemplo, en Andalucía ambos roles pueden llegar a cobrar menos de la mitad, mientras que en Madrid los salarios pueden incluso duplicarse.

No obstante, para intentar realizar una estimación lo más objetiva posible, nos fijaremos en los salarios que establece el BOE, antes citados. Teniendo en cuenta el tiempo total de trabajo y los salarios correspondientes, el coste en personal del proyecto asciende a 15.556,03€.

2.6.6 Total de gastos

Para calcular el coste total del proyecto nos bastará con sumar las cantidades anteriormente estimadas.

Tipo de gasto	Coste
Gasto en hardware	1.960€
Gasto en licencias	0€
Gastos en servicios externos	333,60€
Gastos en transporte	0€
Gastos en salarios	15556,03€
Total:	**17549,63€**

Es decir, aproximadamente 18.000€ en total. Es un gasto absolutamente ridículo comparado con el costo que tiene crear cualquier tipo de aplicación hoy en día, sin duda, pero desde luego es inasumible para un joven estudiante sin ninguna fuente de ingresos más que su salario mensual. Es por eso que el proyecto necesita un buen modelo de financiación.

Adicionalmente al desglose de gastos, presentamos un diagrama de Grantt que nos permitirá visualizar las etapas del proyecto en el tiempo y el tiempo estimado para cada fase. Hay que decir que esta planificación, realizada al inicio del TFG, ha sido desmentida por la realidad, ya que se han tardado algo más de dos años en la realización del proyecto.

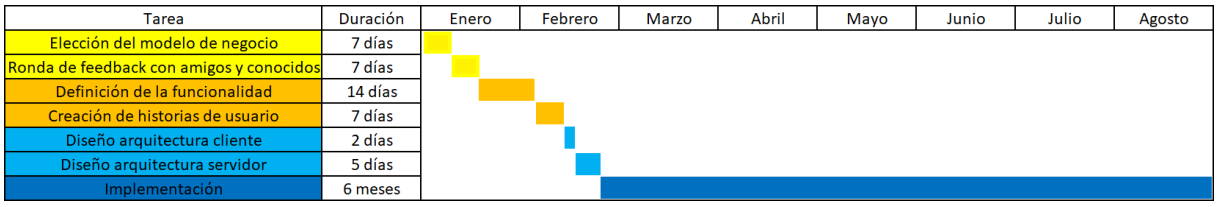


Figura 1: Diagrama de grantt

2.6.7 Modelo de negocio

Como se ha mencionado anteriormente este proyecto se ha desarrollado con una especial atención en el estudio de rentabilidad. Es por esto que el modelo de negocio se ha estudiado cuidadosamente y planificado en consecuencia.

Para intentar rentabilizar la inversión inicial y, por qué no, poder obtener beneficios con la puesta del producto en el mercado, se ha establecido un modelo de financiación por subscripción.

La aplicación cuenta con dos tipos diferenciados de usuarios: usuarios publicadores, que representarán un porcentaje ínfimo de los usuarios reales; y usuarios finales, que serán los masivos en el mercado.

Un usuario final no va a desembolsar ni un euro para poder utilizar la publicación. El uso que este usuario la da a la aplicación es de lo más sencillo: ver las fiestas ordenadas por el criterio que desee.

Sin embargo, el usuario publicador va a tener que asumir los costes. Como sabemos, hoy en día tener visibilidad es una ventaja competitiva importantísima por la que los negocios ya pagan una buena cantidad de dinero (campañas de publicidad, carteles, anuncios en televisión…). Esta aplicación se presenta como la plataforma perfecta para obtener visibilidad, además, esta visibilidad no se obtiene entre usuarios al azar (como el caso de la publicidad), sino que se centra en dar visibilidad al cliente objetivo de un negocio de ocio: los usuarios de la aplicación.

Y es que el 100% de las personas que acaben visualizando una publicación de un usuario publicador en *KELOKE* lo hacen con el objetivo único de informarse sobre fiestas. Es decir, absolutamente todo usuario alcanzado con una publicación de un usuario publicador, es potencialmente cliente del organizador de eventos.

Por lo tanto, el subscriptor estará más predispuesto a emplear parte de su presupuesto de publicidad en aplicaciones como esta. Mucho más si *KELOKE* consigue tener una comunidad de usuarios representativa.

Teniendo todo esto en cuenta, se ha diseñado un sistema de subscripciones de la siguiente manera:

- Para dar la oportunidad al usuario publicador de probar el alcance real de la aplicación, existirá un tipo de subscripción totalmente gratuita, que te permite publicar una sola fiesta en total.
- El resto de subscripciones serán por tiempo. La primera de ellas es una subscripción durante un mes, en el cual se pueden publicar hasta 3 fiestas. El coste es de 9’90€.
- El segundo tipo de subscripción te da acceso a 3 meses, con un máximo de 5 fiestas por mes. El coste es de 14’90€
- El tercer y último tipo de subscripción te da acceso por 6 meses, sin límites de fiestas por mes. El coste es de 29’45€.

Como podemos observar el sistema de subscripciones anima a cualquier usuario publicador a que pruebe la aplicación, ya que cuenta con una subscripción gratuita. La idea es que un organizador pueda ver qué efecto tiene *KELOKE* en el total de gente que acude al evento, y confiando en que este efecto será positivo, pueda animarse a pagar por nuestros servicios.

Una vez que un usuario se decide a dar el paso y a invertir en la aplicación, el sistema de subscripciones premia la confianza. Es decir, si se sigue sin estar seguro de si la aplicación es rentable o no para un organizador de eventos, lo lógico es elegir una duración de subscripción no muy larga en el tiempo. No obstante, si se decide optar por una subscripción más duradera en el tiempo, el precio tanto por mes como por fiesta publicada baja considerablemente.

Así se intenta crear un sistema de fidelización de usuarios por rentabilidad mensual. Si el usuario publicador hace un rápido cálculo mental, verá que mantener una subscripción mensual es comparativamente muy caro: sólo se dispone de 3 fiestas y cuesta más de 3€ cada fiesta. En el punto opuesto, si se decide optar por la subscripción premium, se podrán publicar cuantas fiestas se quiera durante 6 meses, a un precio ridículo si lo comparamos en duración, e infinitamente mejor si tenemos en cuenta la cantidad de fiestas disponibles.

Al estar hablando de cantidades inversivas bastante pequeñas (de 10€ a 30€, más o menos) para lo que un presupuesto en marketing se refiere, es fácil que este sistema de subscriptores anime al publicador a contratar la opción más duradera en el tiempo. No es un desembolse grande de dinero y la rentabilidad respecto al resto de subscripciones es obvia. De esta manera se intenta captar a usuarios activos y duraderos en el tiempo, que nutran nuestra aplicación de fiestas para los usuarios finales.

Además del sistema de subscripciones, dentro de la aplicación hay otro sistema de financiación. Y es que un usuario publicador puede ampliar el número total de fiestas.

Obviamente si se tiene una opción premium, con un número ilimitado de fiestas por mes, esta opción no tiene sentido. Pero es posible que un usuario publicador haya elegido cualquiera de las otras subscripciones y, tras ver que la aplicación funcione, eventualmente no disponga de más fiestas en el mes actual. Para poder dar la opción al usuario de seguir publicando, se venden fiestas individuales, a razón de:

- 1 fiesta adicional este mes, 2’95€
- 3 fiestas adicionales este mes, 4’65€
- 5 fiestas adicionales este mes, 7’95€

Vemos como el precio de una fiesta individual es disparatadamente alto si lo comparamos con las fiestas mensuales que se obtendrían con subscripciones más ambiciosas. Obviamente cuantas más fiestas se deseen comprar el precio por fiesta se abarata, pero nunca es más rentable comprar fiestas a parte que subscribirse con una subscrición de mayor duración en el tiempo.

De nuevo con este sistema se busca fidelizar usuarios. Cualquier usuario que se haya subscrito, quedado sin fiestas y decidido comprar más, podrá darse cuenta de que le ha salido más cara esta opción que subscribirse con un tipo de subscripción más duradera en el tiempo. De esta manera se intenta conseguir el máximo de subscripciones premium posibles.

2.7 Modelo de dominio

En el desarrollo de negocio de esta aplicación se identifican pocas identidades relevantes. Procedamos a presentar y explicar con detalle el modelo de dominio:

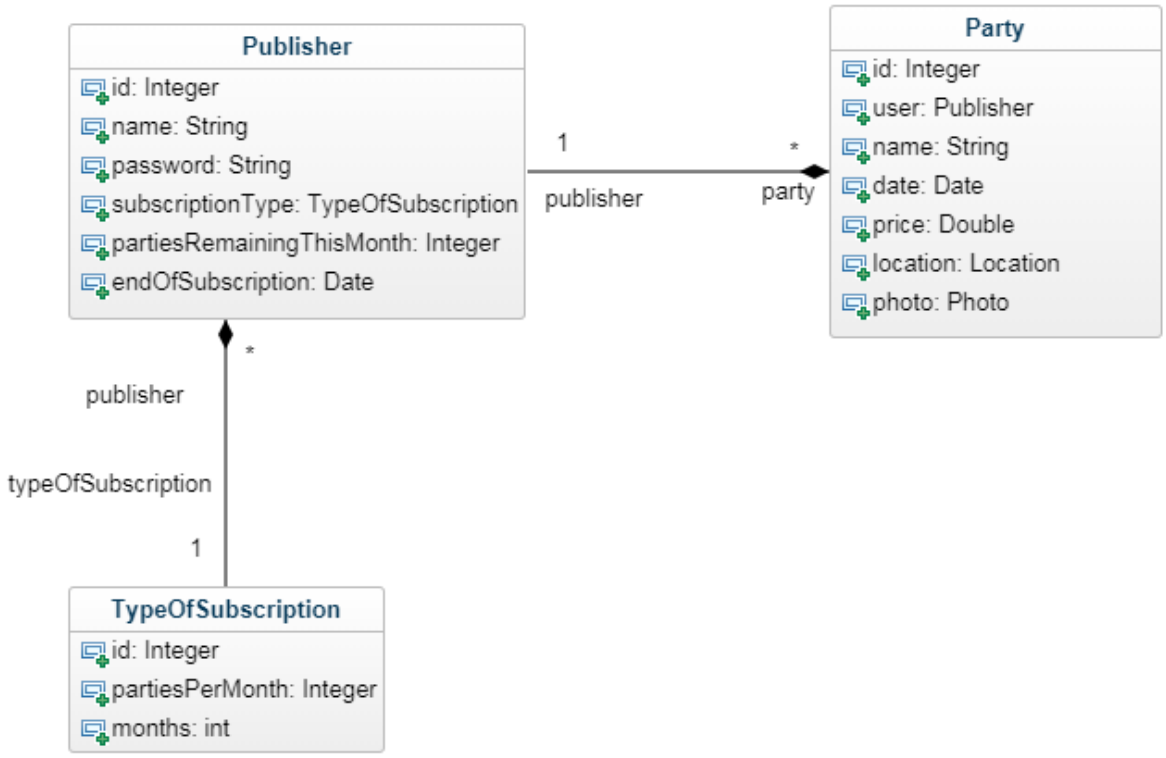


Figura 2: Modelo de dominio

Como podemos ver en la aplicación sólo se tiene en cuenta un tipo de usuario: el publicador. Esto es así porque podemos mantener al sistema agnóstico de los usuarios que no publican fiestas: no realizan ninguna acción en la aplicación más que consultas a la base de datos.

Un publicador, entre otros atributos, tiene un tipo de subscripción. Este tipo de subscripción es un enumerado que define dos parámetros principales: las fiestas de las que se dispone por mes y la duración de la subscripción. Un publicador sólo puede tener un tipo de subscripción, que puede ser compartido por varios publicadores.

Por otro lado, tenemos la entidad principal de la aplicación: la fiesta. Una fiesta tiene todos los atributos necesarios para la correcta visualización de su información. Tiene una relación de N a 1 con publicador, es decir, una fiesta pertenece a 1 y sólo a 1 publicador, pero un publicador puede tener de 0 a N fiestas.

3. Diseño

3.1. Diagrama Entidad-Relación

En el siguiente diagrama ER estableceremos todas las interacciones entre las entidades que componen el sistema:

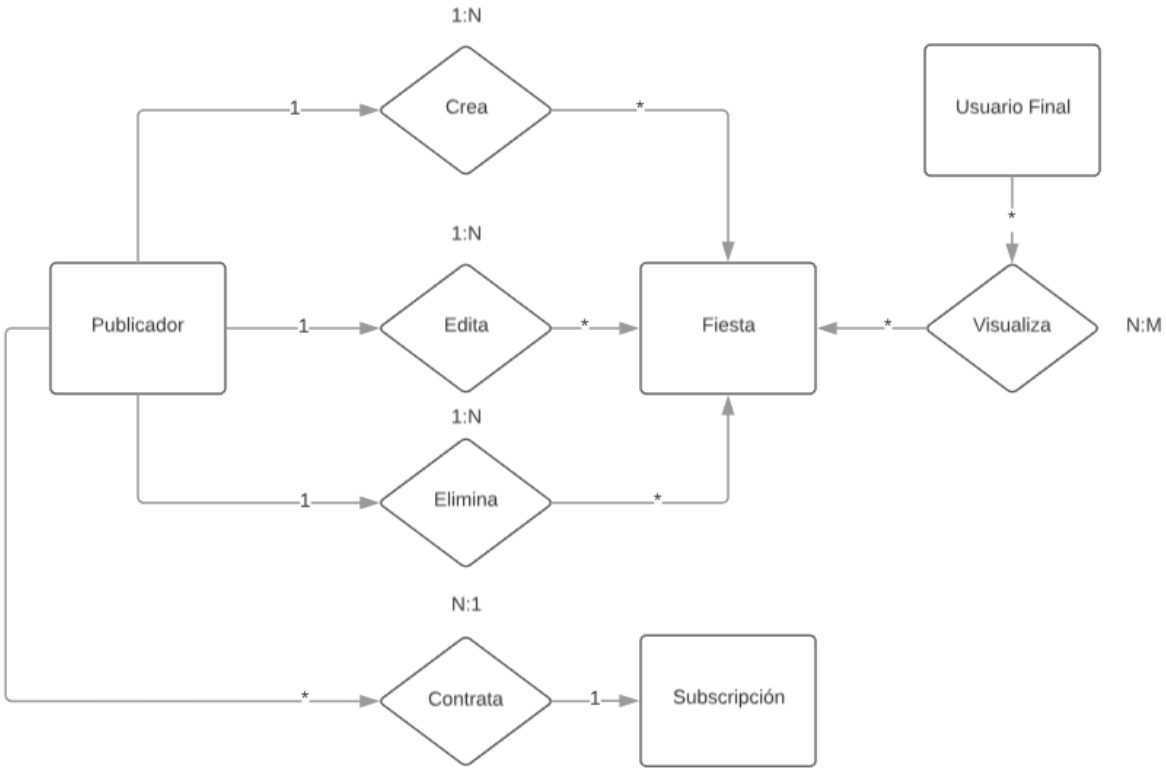


Figura 3: Diagrama ER

Como podemos ver, la entidad principal sigue siendo la fiesta. Un usuario publicador establece una relación de N a 1 con la subscripción que contrata: es única para dicho usuario, pero esta subscripción puede pertenecer a muchos más usuarios.

A su vez, puede crear tantas fiestas como le permita su subscripción. Mientras tenga una fiesta creada, tiene la posibilidad de editarla cuantas veces desee, aunque sólo puede borrarla una vez. Todas estas operaciones se declaran de 1 a N ya que un usuario puede tener potencialmente N fiestas, no obstante, cada fiesta pertenece a un único usuario.

Por último, los usuarios finales visualizan una cantidad ilimitada de fiestas. A su vez, estas fiestas pueden ser visualizadas por una cantidad ilimitada de usuarios. Por lo tanto, la relación entre usuarios finales y fiestas es de N a M.

3.2 Diagrama de clases

En el siguiente diagrama vamos a describir las entidades que componen nuestra aplicación en forma de clases. Como en realidad tenemos dos aplicaciones, una en el cliente y otra en el servidor, hay dos diagramas que presentar ya que las responsabilidades funcionales están divididas y aisladas.

3.2.1 Aplicación Servidor

Empecemos con la aplicación servidor, una API REST que se encarga de realizas operaciones CRUD (Creation, Read, Update, Delete) sobre nuestro modelo de datos:

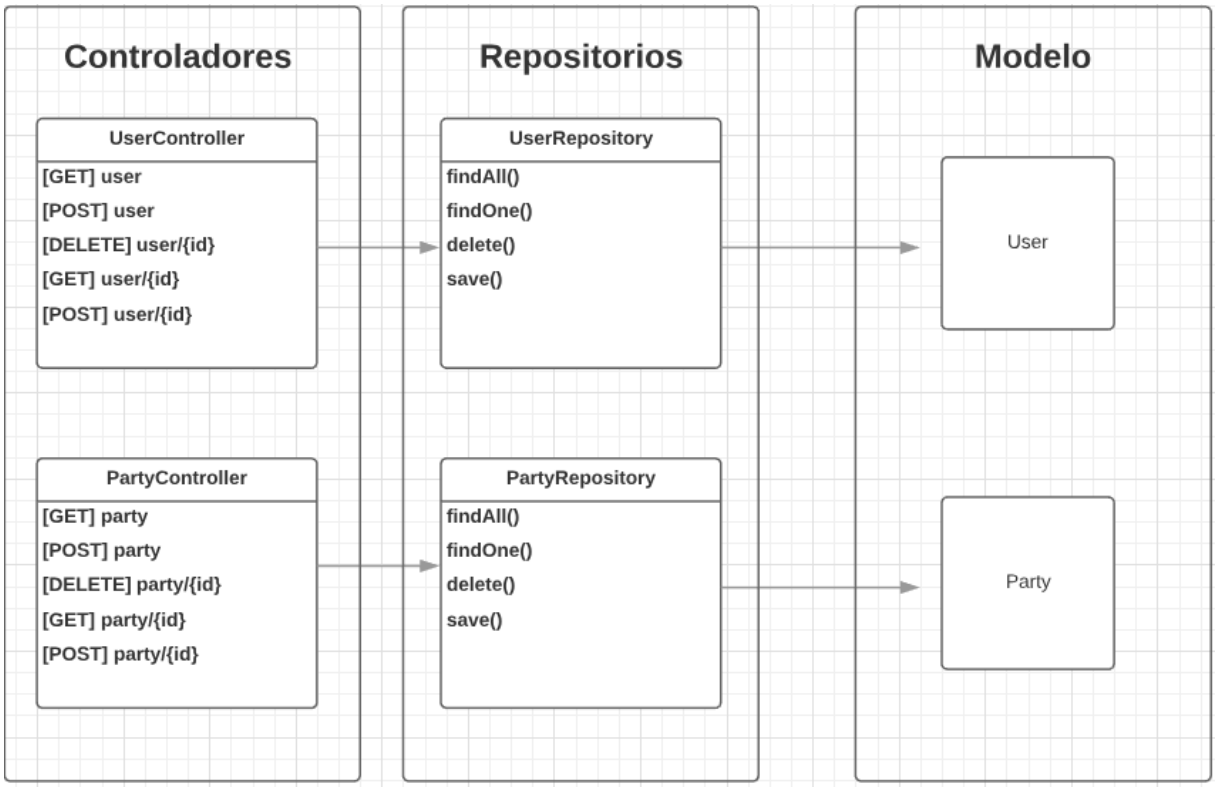


Figura 4: Diagrama de clases de la API

Como vemos nuestra API presenta un flujo funcional diseñado para ser sencillo. La aplicación se divide en 3 bloques: los controladores, que son los responsables de exponer endpoints al exterior vía HHTP, por los cuales nuestra aplicación cliente podrá comunicarse con nosotros. Por otro lado, tenemos los repositorios, concretamente implementaciones de repositorios de JPA. Un repositorio es el encargado de manipular de manera efectiva nuestros datos: será el responsable de hacer las consultas, modificaciones, inserciones y eliminaciones en nuestra base de datos. Por último, tenemos el modelo, clases POJO (Plain Old Java Object) que sólo contienen datos y lógica para consultarlos y modificarlos (métodos Get y Set)

3.2.2 Aplicación Cliente

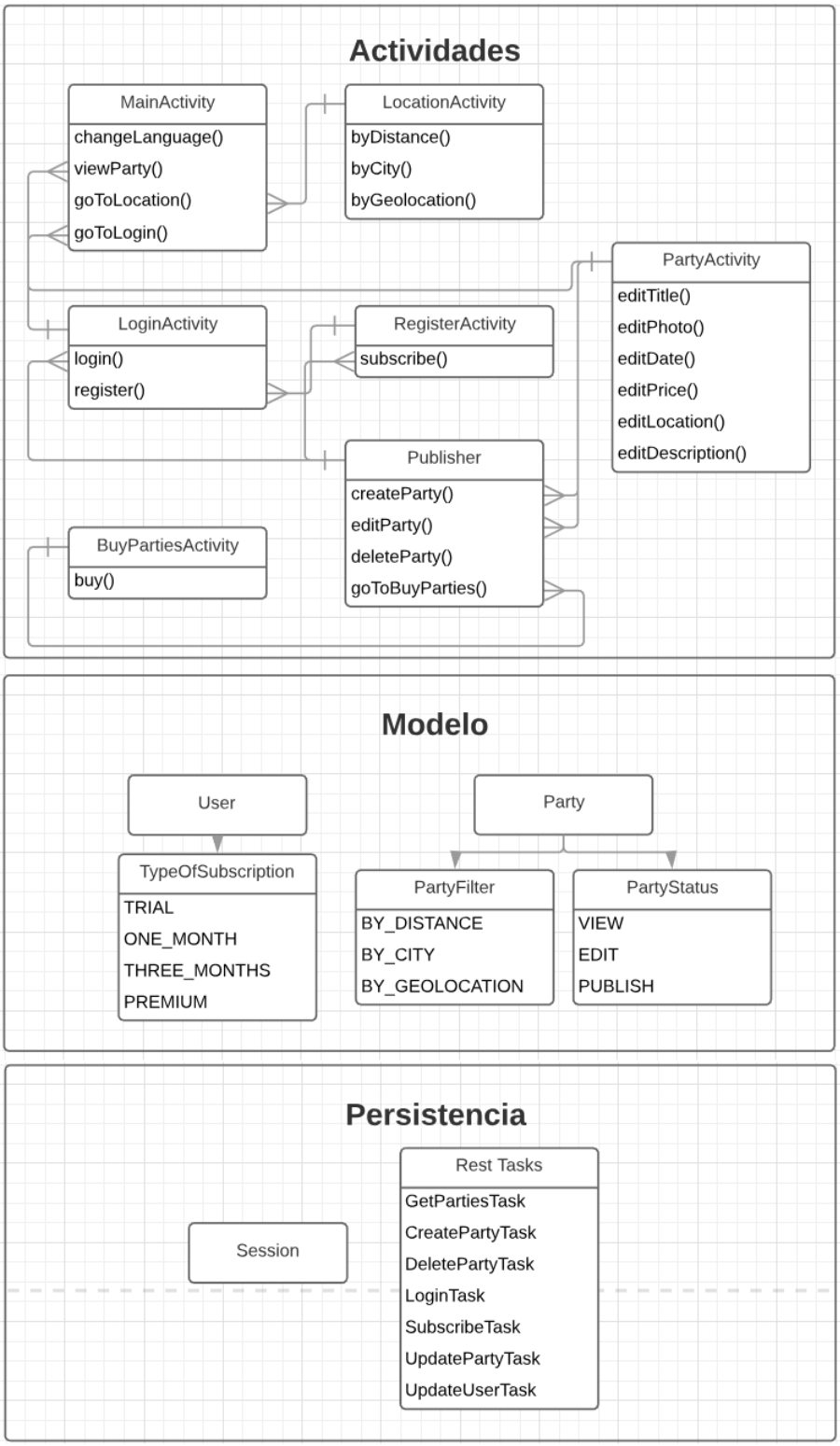


Figura 5: Diagrama de clases de KELOKE

Este esquema es bastante más complejo que el anterior. En la aplicación cliente podemos observar un diagrama de clases muy ligado al paradigma orientado a eventos de Android, funcionalmente orientado a actividades.

Como podemos observar las clases se dividen en tres conjuntos diferenciados: las clases actividad, las clases pertenecientes al modelo, y las clases relativas a la persistencia de datos. Pasemos a explicar cada grupo detalladamente.

Empecemos con las **clases de actividad.** En Android una actividad es una modelación lógica de una vista (o pantalla) en el móvil. Es la clase que contiene todo el código necesario para que una vista sea capaz de visualizar datos dinámicos y procesar peticiones. Es por esto que el número de actividades es igual que el número de vistas que contiene la aplicación. Cada actividad expone una serie de funcionalidad, correspondiente a lo que un usuario es capaz de hacer en cada vista.

Las **clases del modelo** son clases pertenecientes a nuestro modelo de datos, nuestro dominio de negocio. Hay más clases de las que hay en la API porque no todo nuestro modelo necesita ser persistido en la base de datos: el filtro que se use para buscar fiestas o el estado de *PartyActivity* son simples modelaciones que nos permiten escribir código más legible, pero carecen de vida más allá de la sesión local del usuario.

Por último, las clases de persistencia son las que nos permiten manipular el estado de los datos de nuestra aplicación. Se dividen en dos grupos, el primero de ellos, las actividades REST, que nos permiten realizar comunicaciones asíncronas a través de HTTP. Hay una clase por cada tipo de petición que se realiza en la aplicación, como buscar una fiesta concreta, crearla, crear un usuario nuevo, etc. Por otro lado, la clase Session nos permite almacenar información local que necesitamos compartir entre actividades y que no necesitamos persistir en la base de datos. Esta clase sigue un patrón Singleton [3] que nos asegura que sólo será instanciada una vez por aplicación cliente.

3.3 Diagramas de secuencia

En este apartado presentaremos diagramas de secuencia de los principales flujos de la aplicación. No todos los flujos posibles están contemplados, ya que potencialmente se requeriría decenas de diagramas de secuencia: nos vamos a centrar en las funcionalidades más representativas y que mejor pueden hacer entender la comunicación existente dentro del sistema.

3.2.1 Diagrama de secuencia: Login

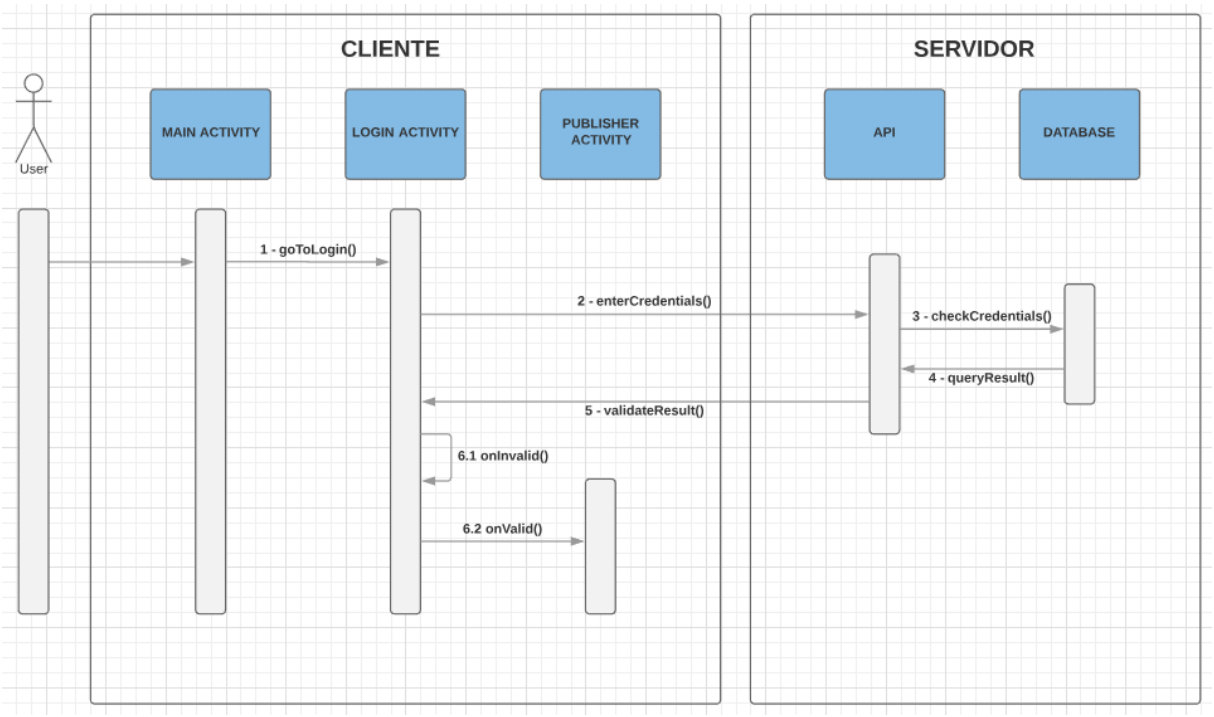


Figura 6: Diagrama de secuencia del login

Como podemos ver el flujo comienza cuando el usuario accede a la vista de login. Una vez en esta vista, deberá introducir sus credenciales (usuario y contraseña) para poder continuar. En este momento la aplicación realiza una llamada HTTP asíncrona para comprobar si las credenciales introducidas pertenecen a algún usuario. En esta llamada la API hace una consulta a la base de datos y devuelve los datos del usuario si es que se han introducido unas credenciales válidas. Si es así, la aplicación redireccionará a la vista de publicador, mostrando todos sus datos. Si las credenciales son inválidas, la aplicación nos devolverá a la vista de login mostrando un mensaje de error, dándole al usuario la oportunidad de que vuelva a introducir sus credenciales correctamente.

3.2.2 Diagrama de secuencia: Visualización de fiestas filtradas

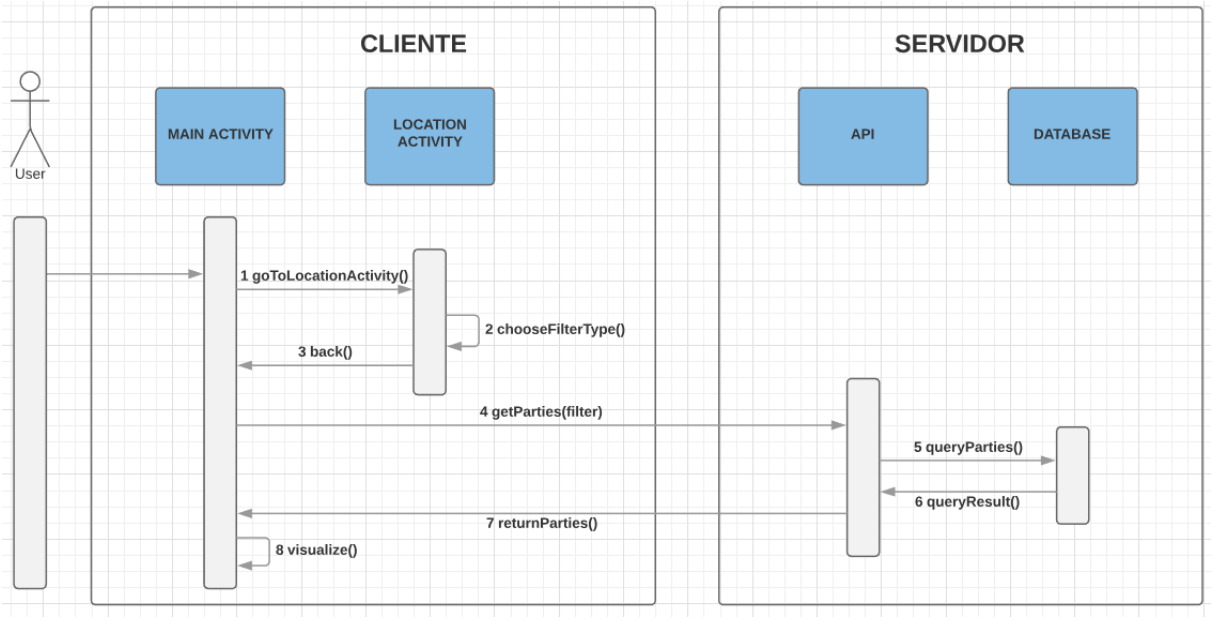


Figura 7: Diagrama de secuencia de la visualización de fiestas

Este diagrama representa un flujo que, aun siendo bastante sencillo y fácil de explicar, es el flujo principal de la aplicación: usuarios finales visualizando fiestas filtradas. El flujo comienza cuando el usuario, desde la vista principal, decide establecer un criterio de búsqueda, recordamos, por distancia (en kilómetros), introduciendo una ciudad en el mapa, o dejando que el dispositivo detecte nuestra ciudad actual por geolocalización. Un botón desde la actividad principal hasta la actividad de localización facilitará todo esto.

Una vez que se ha elegido criterio de filtrado y se vuelve a la vista principal, se realiza una llamada HTTP asíncrona a la API. Esta se encargará de realizar la query en la base de datos con los criterios establecidos por el usuario, y devolverá un listado de fiestas que cumplen las condiciones. Tras recibir estos datos, la aplicación cliente mostrará todas las fiestas disponibles en una tabla.

3.2.3 Diagrama de secuencia: Publicador

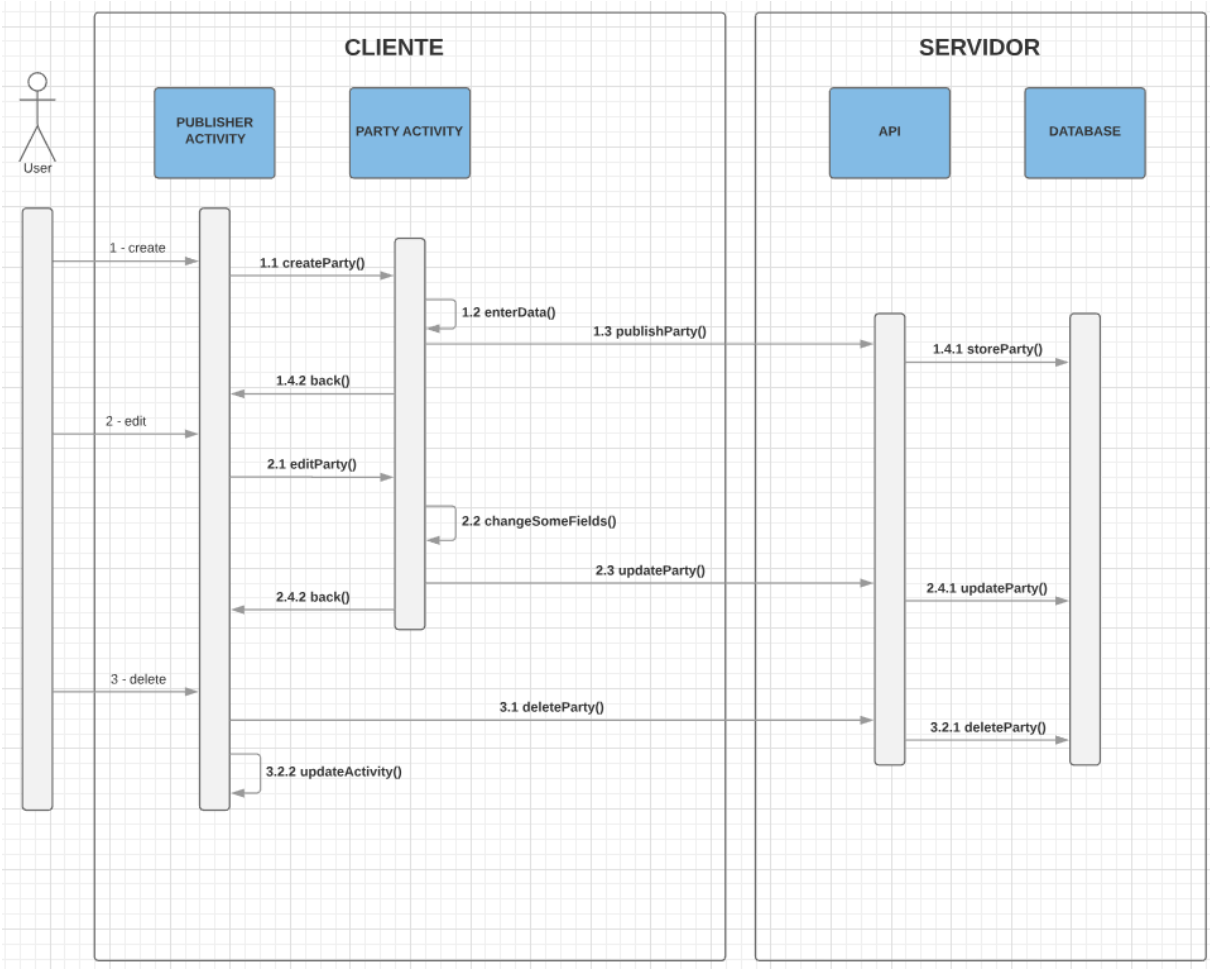


Figura 8: Diagrama de secuencia de publicador

En este diagrama de secuencia se detallan los flujos más relevantes que puede realizar un usuario subscrito a la aplicación. Hay algunos flujos que se han obviado, como comprar fiestas, para evitar generar infinidad de diagramas o diagramas demasiado largos. Este diagrama contiene 3 flujos distintos, crear, editar y borrar una fiesta.

El flujo de **crear una fiesta** comienza cuando el publicador, desde la vista del publicador, hace click en el botón de crear fiesta. Si el usuario no tiene fiestas disponibles este mes, un mensaje aparecerá invalidando la acción e invitando al usuario a comprar más fiestas. Si el usuario sí que dispone de fiestas, la aplicación transiciona hacia la vista de fiesta, donde aparece una fiesta con datos de ejemplo. El usuario en este momento deberá modificar los datos de ejemplo para reflejar todo lo relativo a su fiesta: el nombre, la ubicación, el precio… etc.

Cuando haya finalizado dicho proceso, hará click en el botón de publicar y se realizará una llamada HTTP asíncrona a la API. Ésta guardará la información de la nueva fiesta en la base de datos. Por otro lado, aprovechando la asincronía, la aplicación devolverá al usuario a la vista del publicador.

El flujo de **editar una fiesta** comienza cuando el publicador hace click en el botón de editar sobre una de las fiestas anteriormente creadas. La aplicación redireccionará hacia la vista de fiesta, donde aparecerán todos los datos de la fiesta. Tras modificar cuantos datos considere oportuno, el usuario deberá hacer click en el botón de guardar, lo que desencadenará una petición HTTP asíncrona a la API. De igual manera que en el anterior flujo, a la vez que la API realiza el update en la base de datos, la aplicación cliente redicreccionará a la vista del publicador.

Por último, el flujo de **eliminar una fiesta** es el más sencillo de todos. Comienza cuando un usuario, desde la vista del publicador, hace click en el botón de borrar sobre una fiesta anteriormente creada. Un PopUp aparecerá pidiéndonos confirmar la acción. Tras confirmar, se realizará una petición HTTP a la API, que borrará la fiesta de la base de datos, a la vez que la aplicación recarga la vista de publicador para actualizar la información que se visualiza.

3.4 Diseño de la interfaz

En este apartado presentaremos un Story Board de la aplicación. El esquema se ha realizado con pantallazos de la aplicación real, y recoge todas las vistas de la aplicación y las interacciones entre ellas. Como podemos observar es un esquema realmente útil para entender el flujo funcional de la aplicación de un vistazo.

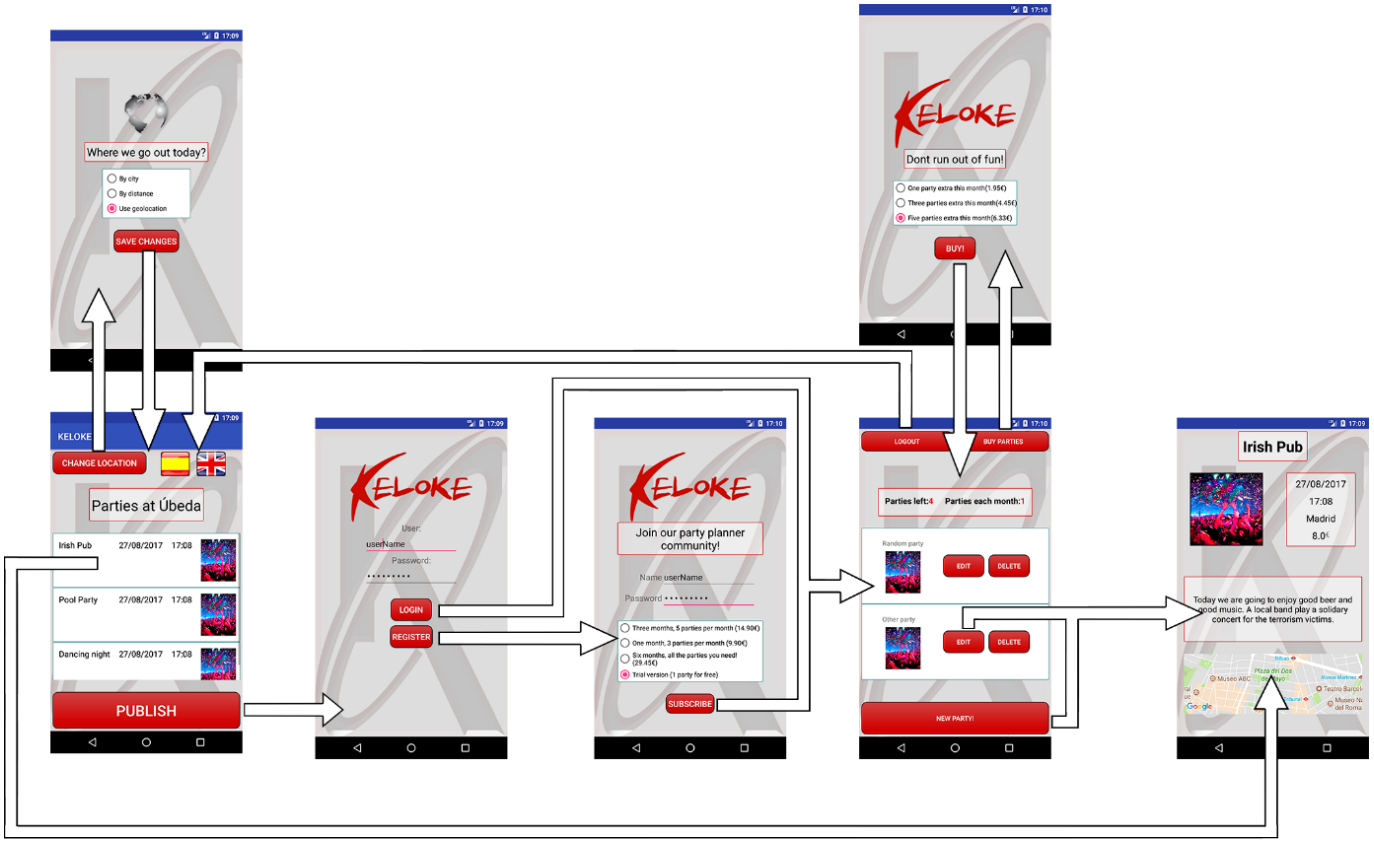


Figura 9: Story Board

4. Implementación

En el presente apartado pasaremos a detallar los detalles más técnicos del proyecto. En primer lugar, se describirá la arquitectura diseñada tanto para la aplicación cliente como para la aplicación servidor. Posteriormente se procederá a detallar de manera exhaustiva cómo funciona la aplicación por dentro, donde se explicarán diferentes paradigmas y frameworks utilizados.

4.1. Arquitectura

4.1.1 API REST (Aplicación servidor)

Como se ha comentado anteriormente, en el lado del servidor tendremos una API REST y una base de datos. La finalidad de la base de datos será, obviamente, persistir estados tanto de fiestas como de usuarios. Por otro lado, la API REST se encargará de proporcionar una interfaz HTTP para que cualquier cliente (en nuestro caso, la aplicación Android) pueda acceder o modificar los datos.

Para la **base de datos** se exploraron dos opciones: bases de datos no relacionales y bases de datos relacionales. Android parecía un entorno bastante proclive a utilizar bases de datos no relacionales, especialmente Firebase [4]. Firebase tiene una integración fácil con Android, además Google proporciona un espacio de memoria en la nube de manera gratuita para desarrolladores. Aun teniendo todo esto en cuenta, en el diseño inicial se valoró por encima la robustez de una base de datos relacional, ya que los datos son más difícilmente corrompibles. Además, elegir firebase implicaría sobrecargar la aplicación cliente de abstracciones que actualmente realiza la API.

Dentro de todas las opciones disponibles entre las bases de datos relacionales, por practicidad se eligió H2. H2 es una base de datos relacional que sólo persiste su estado en memoria, es decir, no guarda estados entre ejecución y ejecución. Esto es un problema mayúsculo para una aplicación en producción, ya que en el momento en el que se desconecte la base de datos se perdería todo su contenido. Este gran hándicap fue valorado como poco relevante debido a que esta aplicación no ha sido diseñada para estar en producción, por lo menos de momento. Por otro lado, H2 nos proporciona una integración muy sencilla con código java y JPA. Además, la curva de aprendizaje para un estudiante de esta carrera es prácticamente nula.

Para la **API REST** se decidió utilizar java como lenguaje de programación. El principal motivo es que es el lenguaje que más se ha utilizado en la carrera y el que practico en mi vida laboral. Por otro lado, se utilizó el plugin Lombok [5] para generar código de manera menos verbosa.

Para realizar la comunicación HTTP en base a un modelo MVC, se eligió Spring por encima de cualquier otro framework. Los motivos son inumerables: es el framework más utilizado de lejos lo que nos asegura más soporte de la comunidad y eso es muy importante para un estudiante. Por otro lado, Spring nos daba la posibilidad de utilizar Spring Boot, que es un framework de Pivotal [6] que facilita muchísimo toda la configuración de la aplicación. Genera varios beans por defecto muy útiles y tiene integración con prácticamente cualquier sistema externo: base de datos, colas RabbitMQ, Kafka…

Vale la pena comentar que durante el desarrollo de la API se hizo un upgrade de la versión de Java, de la JDK6 a la JDK8. Esto se hizo para poder aprovechar las ventajas de las features de Java 8, como los Streams o las Lambdas, que acercan más a Java hacia un paradigma funcional e invita a abandonar el lenguaje declarativo. Este refactor se realizó porque personalmente creo que los lenguajes funcionales son el futuro laboral de los desarrolladores de software.

4.1.2 KELOKE (Aplicación cliente)

La aplicación cliente, por supuesto, se desarrolló en Android. Era imposible desarrollarla en Kotlin ya que este lenguaje no era compatible con Android cuando se empezó el proyecto. Android Studio, que fue el entorno de programación seleccionado, proporciona fácil adaptación a la creación de código java para la lógica interna, integrado por XML con la interfaz gráfica. A pesar de que, a posteriori, considero que no es la mejor de las opciones, ya que una interfaz programada con Javascript parece bastante más razonable, prácticamente no había otra opción al inicio del proyecto, ya que mis conocimientos sobre Javascript eran y siguen siendo limitadísimos.

Dentro de la aplicación se utiliza un framework, concretamente Spring. No ha sido desarrollado en su plenitud ya que sólo ha sido utilizado para programar la comunicación REST con el servidor, al considerarse que las librerías de Spring son las más claras y fáciles de usar con este propósito.

Además del framework, se han utilizado varias librerías que merece la pena destacar, como puede ser la de Google Maps y Google Places. Ambas se han utilizado para poder mostrar mapas, elegir a través de la interfaz de Google y tratar programáticamente ubicaciones de todo el mundo.

Dentro del proyecto, podemos encontrar dos paquetes importantes: el paquete java y el paquete resources. En el paquete java se encuentran las clases y en resources, podemos encontrar los elementos XML.

Para organizar el código XML, se hicieron dos divisiones principales: los *layouts*, que son archivos que definen interfaces gráficas, y los *drawables*, que son elementos independientes que son visualizados en varias partes de la aplicación. Esto archivos están tanto en XML como en otros formatos propios de imágenes como PNG o JPG.

Para organizar el código ejecutable en una vista, se utilizó el paradigma de **Activity** de Android. Una *Activity* es una clase que se encarga de recibir peticiones y mandar respuestas a la interfaz gráfica. De esta manera cada vista de la aplicación tiene una clase asociada que contiene toda la lógica de dominio necesaria para la correcta ejecución de las funcionalidades disponibles en dicha vista.

Para realizar peticiones REST, la aplicación utiliza una utilidad propia de Android llamada **AsyncTask**. Como su propio nombre indica, nos permite realizar tareas asíncronas durante la ejecución de la aplicación. Obligados por el paradigma Android, que es no bloqueante por naturaleza, se implementaron varios tipos de peticiones REST que la aplicación necesita realizar en tareas de este tipo.

Por otro lado, tenemos al conjunto de clases del paquete de **entidades.** Como se ha mencionado anteriormente representan los objetos de dominio de nuestra aplicación, que son simples POJOs que carecen de lógica interna, más allá de la recuperación y asignación de valores.

Finalmente, hay una clase que merece una mención aparte ya que no encaja en ninguno de los grupos mencionados, la clase **Session**. Nuestra aplicación tiene la particularidad de que necesita persistir muchos menos datos de los que maneja. Hay datos que sólo son relevantes en la ejecución en curso, como por ejemplo que criterio de búsqueda utiliza un usuario, o el idioma en el que prefiere visualizar la aplicación. Para guardar todos estos datos y poder consultaros, utilizamos esta clase, que sólo permite una instancia por ejecución.

4.2 Detalles sobre la implementación

En este apartado entraremos de fondo en detalles específicos de implementación. Podremos ver parte del código que se ha desarrollado, que nos ayudará a comprender el sistema que se ha construido. De nuevo, dividimos esta sección en dos, ya que realmente se han desarrollado dos aplicaciones.

4.2.1 API REST (Aplicación servidor)

La aplicación está basada en Spring Boot, lo que facilita mucho tanto la configuración necesaria para utilizar Spring como los procesos del desarrollo, ya que Spring Boot cuenta con un servidor Tomcat embebido. La aplicación consta de una clase principal que establece la configuración general.

```
@ComponentScan
@EnableAutoConfiguration
@EnableJpaRepositories
@EnableSwagger2
@Configuration
public class KelokeApiApp {

    public static void main(String[] args) {
        SpringApplication.run(KelokeApiApp.class, args);
    }

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
                .select()
                .apis(RequestHandlerSelectors.any())
                .paths(PathSelectors.any())
                .build();
    }
}
```

Figura 10: Clase de configuración de Spring Boot

Un detalle interesante es que cuenta con integración con Swagger [7], lo que ha ayudado mucho en la fase de pruebas. Con Swagger podemos olvidarnos de herramientas que quizás sean algo obsoletas como Postman [8]. Aparte de esto, la clase cuenta con un sencillo método main() que, junto con las anotaciones de la clase, nos genera el esqueleto necesario para una API REST sencilla como es esta.

Como se ha descrito anteriormente, esta aplicación tiene un sencillo diseño orientado a operaciones CRUD. Este diseño se divide en controladores, repositorios y entidades de dominio.

Empecemos primero con las clases de dominio. Estas clases son POJOs sin lógica de negocio. Al utilizar Lombok, podemos comprobar que el código resultante es mucho más sencillo y legible (siempre que se tenga familiaridad con Lombok, hay que decirlo)

```
@Entity
@Getter
@Setter
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    @Enumerated(EnumType.ORDINAL)
    private TypeOfSubscription typeOfSubscription;
    private String name;
    private String pass;
    private Integer partiesRemainingThisMonth;
    private Date subscriptionEndDate;
}
```

Figura 11: Entidad usuario

Además de Lombok, en la implementación de las entidades de dominio se utilizan anotaciones de JPA para poder integrarnos de manera sencilla con una base de datos SQL como es H2. La anotación @Id establece el atributo con el cual JPA identificará a un recurso. Además, establecemos una autogeneración de IDs, y restringimos la generación para que sea única con @GeneratedValue.

Por último, la anotación @Enumerated nos permite persistir enumerados en la base de datos de manera sencilla, por un número generado en este caso, aunque se puede guardar el nombre de la instancia.

Por otro lado, para poder manipular los objetos de dominio se han utilizado implementaciones de los repositorios de JPA. Dichos repositorios ofrecen una abstracción de las consultas SQL más habituales en operaciones CRUD, sin tener nosotros que escribir una sola línea de código. Si necesitásemos realizar consultas complejas podríamos programarlas de distintas maneras, pero no ha sido el caso en este proyecto.

```
@Repository
public interface UserRepository extends CrudRepository<User, Integer> {

    @Override
    List<User> findAll();

    @Override
    User findOne(@Param("id") Integer id);

    @Override
    User save(User user);

    @Override
    void delete(User user);
}
```

Figura 12: Repositorio JPA

Simplemente indicando la entidad del repositorio (en el caso de la imagen 10, la entidad Usuario) podemos disfrutar de la funcionalidad que describe la interfaz de CrudRepository sin tener que programar nada. Nuestro repositorio sólo necesita 4 métodos, uno para listar los usuarios, otro para obtener la información de un usuario, y otros dos métodos para guardar y borrar usuarios. Este último no es invocado en la aplicación cliente, ya que no hay opción de borrar cuenta, pero formaba parte de lo que se consideró como mínimo para seguir los principios RESTful.

Por último, para poder acceder a todas estas operaciones, exponemos endpoints en Controllers REST. Estas clases son las que exponen a través del tomcat una serie de peticiones HTTP que cualquier cliente REST puede realizar. En el diseño se tuvo en cuenta los principios RESTful para exponer recursos de una manera estándar, ya que, aunque REST no cuenta con contrato como es el caso de SOAP, es conveniente seguir algunas buenas prácticas que facilitan la colaboración entre profesionales.

```
@RestController
@RequestMapping(value = "/user", produces = MediaType.APPLICATION_JSON_VALUE)
public class UserController {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private PartyRepository partyRepository;

    @RequestMapping(method = RequestMethod.GET)
    public Iterable <User> getUsers() { return userRepository.findAll(); }

    @RequestMapping(method = RequestMethod.POST)
    public User createUser(@RequestBody User user) {...}

    @RequestMapping(value = "/{id}", method = RequestMethod.POST)
    public User createUser(@RequestBody User user, @PathVariable("id") int id) {...}
    @RequestMapping(value = "/{id}", method = RequestMethod.GET)
    public User getUser(@PathVariable("id") Integer id) { return userRepository.findOne(id); }

    @RequestMapping(value = "/{id}", method = RequestMethod.DELETE)
    public User deleteUser(@PathVariable("id") Integer id) {...}

    @RequestMapping(value = "/{id}/parties", method = RequestMethod.GET)
    public Iterable<Party> getUserParties(@PathVariable("id") Integer id) {...}
}
```

Figura 13: Controller REST

Como podemos ver utilizamos el contexto de Sprint para injectar dependencias, concretamente, los repositorios necesarios para ejecutar nuestras operaciones. Hacemos todo esto gracias a la anotación @Autowired, que puede ir a nivel de línea o a nivel de constructor.

4.1.2 KELOKE (Aplicación cliente)

Como se ha mencionado anteriormente, en la aplicación cliente se ha utilizado dos lenguajes: Android para la lógica de negocio y XML para el diseño de la interfaz.

En Android se pueden desarrollar interfaces de manera nativa usando el paradigma de Layout. Un layout es una abstracción que permite diseñar interfaces gráficas con comportamientos dinámicos, apoyándose en las Activitys de Android. El proyecto cuenta con más layouts que vistas, ya que dentro de una vista se pueden combinar diferentes layouts.

Un ejemplo sencillo sería los pop ups que aparecen en la aplicación, tanto para confirmar operaciones como para notificar al usuario.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:background="@drawable/popup_style">

    <RelativeLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:gravity="center"
        android:layout_marginTop="0dp"
        android:background="@drawable/popup_gradient">

        <TextView
            android:id="@+id/welcomeText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Welcome! Lets get into hot water!"
            android:layout_centerHorizontal="true"
            android:textColor="#FFFFFF"
            android:textSize="18sp"
            android:padding="15dp"/>

        <Button
            android:id="@+id/beginButton"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_below="@id/welcomeText"
            android:text="BEGIN NOW!"
            android:textColor="#000000"
            android:background="@drawable/popup_button_style"
            android:textSize="24sp"/>

    </RelativeLayout>
</RelativeLayout>
```

Figura 14: Layout pop up de bienvenida

En la figura anterior podemos ver uno de los layouts más sencillos de la aplicación, un pop up que muestra un mensaje de bienvenida cuando un usuario realiza una subscripción. Asociando IDs a los componentes del layout, podremos referenciaros en una clase Java actividad (o de cualquier otro tipo) y establecer comportamientos programáticos a acciones como pinchar, introducir texto, etc.

Dejando de lado la parte XML, nos adentramos ahora en la lógica de negocio que se ejecuta en la aplicación. Empezando este bloque tiene sentido explicar el principal tipo de clase de KELOKE, que son, por supuesto, las actividades. A riesgo de ser repetitivo, una clase actividad contiene la lógica asociada al comportamiento que expone la interfaz gráfica. Veamos la actividad asociada a la vista inicial de la aplicación, donde se visualizan las fiestas:

```
public class MainActivity extends AppCompatActivity {

    private static final String MEASUREMENT = "km";
    public static final int GPS_REQUEST_TIME = 0;
    public static final int GPS_REQUEST_DISTANCE = 0;
    public static final int LOCATION_REQUEST_CODE = 2;

    private TextView cityText;
    private TableLayout layoutPartiesTable;
    private LocationManager locationManager;
    private LocationListener locationListener;
    private Geocoder geocoder;

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    private void setSearchFilter() {...}

    private void getLocationFromGPS() {...}

    private void setLocationIfPermission() {...}

    @Override
    public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {...}

    @Override
    protected void onPause() {...}

    private void fillPartiesTable() {...}

    public void setSpanishLanguage(View v) {...}

    public void setEnglishLanguage(View v) {...}


    public void locationButton(View v) { startActivity(new Intent( packageContext: this, LocationActivity.class)); }

    public void publishButton(View v) {...}
```

Figura 15: Main Activity

Esta clase extiende de AppCompatActivity, lo que nos permite definir comportamientos asíncronos, necesarios para la interacción entre la interfaz y la lógica de negocio. Como vemos contiene como objetos los diferentes componentes dinámicos de la vista principal. El método onCreate() establece el comportamiento que debe ejecutar la actividad cuando se carga la vista, es decir, cuando se abre la aplicación. Entre otras cosas, realiza la primera consulta a la API para poder visualizar una lista de fiestas con los criterios que el usuario haya establecido.

Además, podemos ver una serie de métodos encapsulados que son utilizados por la clase para realizar operaciones aisladas, como puede ser setSpanishLenguage() y setEnglishLanguage(), cuya funcionalidad se puede intuir. Otro método nos permite rellenar la lista de las fiestas con datos nuevos (utilizado para ello las llamadas asíncronas) o el comportamiento que se ejecuta cuando se hace click en el botón de publicar.

Ya que considero la clase actividad la más importante dentro del código desarrollado, pasemos a exponer otro ejemplo con más detalle. Podría pensarse que una clase llamada Main Activity sería la actividad principal de la aplicación, pero esto es sólo medio verdad. Lo es para un usuario que sólo busque y visualice fiestas, pero para el usuario publicador, que representa un porcentaje mayor del código desarrollado, la actividad principal es Publisher Activity.

```
public class PublisherActivity extends Activity {

    private TextView partiesRemainingThisMonthView;
    private TextView partiesPerMonthView;
    private TableLayout layoutPartiesTable;
    private LayoutInflater layoutInflater;
    private PopupWindow popupWindow;
    private View popupView;
    private Button logoutButton;
    private Button yesButton;
    private Button noButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    private void setUserInfo() {...}

    private void fillPartiesTable() {...}

    public void publishButton(View view) {...}

    public void logoutButton(View view) {...}

    private void goToMainActivity() { startActivity(new Intent( packageContext: this, MainActivity.class)); }

    public void goToBuyParties(View view) {...}

    public boolean onKeyDown(int keyCode, KeyEvent event) {...}

    public boolean onKeyUp(int keyCode, KeyEvent event) {...}
}
```

Figura 16: Publisher Activity

De igual manera que en el caso de Main Activity, Publisher Activity contiene objetos privados que nos permiten manipular programáticamente los componentes de la vista de publicador, como por ejemplo la tabla de fiestas, o los botones de acción NO y SÍ que aparecen en el pop up de borrar una fiesta.

De nuevo, el método onCreate() nos permite ejecutar una lógica inicial, previa a que la interfaz de usuario cargue esta vista. En el caso de Publisher Activity, se guarda la instancia de ejecución (que nos permite acceder al estado de la aplicación), se inicializan los objetos asociándolos a su respectivo componente visual y se muestran los datos del publicador. Para conseguir esto, hay dos métodos distintos que vale la pena comentar:

Uno de ellos, setUserInfo(), establece los datos de la subscripción, como el número de fiestas por mes, o el tiempo de subscripción. Este método no necesita una llamada REST ya que en la vista de login ya se consiguieron estos datos, lo que hacemos es apoyarnos en la clase Session para guardarlos en el login y consultarlos cuando nos sea necesario.

El segundo, fillPartiesTable() es el en cargado de poblar la tabla de fiestas del publicador. Este método sí que requiere de una llamada REST, que es gestionada a través de las AsyncTask, clases que detallaremos más adelante.

El resto de métodos son sencillamente asociables a comportamientos de la interfaz: publishButton(), que nos pasa a la vista de fiesta, logoutButton(), que nos devuelve a la actividad principal, y dos métodos más de transición por vistas.

Los métodos onKeyUp() y onKeyDown() nos permiten establecer comportamiento cuando el usuario pulsa el botón de atrás, y cuando deja de pulsarlo, para iniciar la transición en el momento que se desee y además, definir hacia dónde va la transición. Si no se programan, el botón de atrás puede permitir transiciones que nuestra lógica de negocio no debería permitir, por ejemplo, un usuario podría loguearse, hacer logout, y después volver a la pantalla de publicador sin volver a hacer login, lo que afectaría a la seguridad de la aplicación

Habiendo sido ya mencionada, es el momento de explicar con más detalle la clase Session. El patrón de diseño que sigue nos permite asegurar que, por ejecución, es decir por dispositivo móvil, sólo se realizará una instancia. Esto nos permite una cierta robustez, ya que no tiene sentido tener dos sesiones a la vez en la aplicación: la funcionalidad actual no lo contempla.

```
public class Session {

    private static Session instance;
    private static SimpleDateFormat dateFormat;
    private static SimpleDateFormat hourFormat;
    private static FilterType filterType;
    private static String city;
    private static Location location;
    private static String geolocationCity;
    private static Float distance;
    private static User user;
    private static Boolean geolocationEnabled;

    private Session() {...}

    public static Session getInstance() {...}

    public SimpleDateFormat getDateFormat() { return dateFormat; }

    public void setDateFormat(SimpleDateFormat dateFormat) { Session.dateFormat = dateFormat; }

    public SimpleDateFormat getHourFormat() { return hourFormat; }

    public void setHourFormat(SimpleDateFormat hourFormat) { Session.hourFormat = hourFormat; }

    public FilterType getFilterType() { return filterType; }

    public void setFilterType(FilterType filterType) { Session.filterType = filterType; }

    public String getCity() { return city; }

    public void setCity(String city) { Session.city = city; }

    public Location getLocation() { return location; }

    public void setLocation(Location location) {...}

    public String getGeolocationCity() { return geolocationCity; }
```

Figura 17: Session

Para poder aplicar el patrón singleton, debemos establecer el constructor o constructores de la clase como privados. De esta manera sólo dentro de la propia clase se podrán realizar instancias. La manera de acceder a *Sesión* es utilizar un método estático, getInstance(). Este método se fija en el objeto llamado *instance* de tipo *Sesión* dentro de la clase: si es nulo, lo crea con el constructor privado y lo devuelve. Si no es nulo, se simplemente se devuelve.

```
public static Session getInstance() {
    if (instance == null) {
        instance = new Session();
    }
    return instance;
}
```

Figura 18: Método getInstance()

A partir de aquí, como podemos ver el resto de la case sigue el mismo esquema que cualquier otro POJO: almacena datos que son necesarios modificar y consultar en algún flujo funcional de la aplicación. Por ejemplo, guarda los datos del Usuario, sólo en el caso de que se haya realizado un login. También guarda el criterio de búsqueda que ha establecido el usuario, o el formato de las fechas que se prefiere visualizar.

Por último, pasemos a explicar las clases utilizadas para la comunicación REST con la API: las **AsyncTasks**. Estas clases utilizan un paradigma asíncrono, que nos permite realizar peticiones mientras la aplicación cliente ejecuta lógica de negocio. Hay algunas tasks que, inevitablemente, detienen la ejecución de la aplicación cliente, como puede ser la consulta de fiestas. Necesariamente tenemos que esperar a la recepción de este tipo de información, ya que la necesitamos para visualizar la tabla de fiestas, no tenemos alternativa. Sin embargo, hay otras actividades, como la de comprar fiestas, que podemos realizar totalmente en paralelo. Sólo iremos a la API para informar que el usuario dispone de más fiestas, pero la aplicación esto ya lo sabe y puede seguir su flujo de manera totalmente independiente.

```
public class CreatePartyTask extends AsyncTask<Void, Void, Void> {

    private static final String BASE_URL = "http://10.0.2.2:8080/party";

    private Party party;
    private RestTemplate restTemplate;

    public CreatePartyTask(final Party party) {
        this.party = party;
        this.restTemplate = new RestTemplate();
    }

    @Override
    protected Void doInBackground(Void... params) {
        try {
            HttpHeaders headers = new HttpHeaders();
            headers.setContentType(MediaType.APPLICATION_JSON);
            HttpEntity<Party> request = new HttpEntity<>(party, headers);
            restTemplate.getMessageConverters().add(new MappingJackson2HttpMessageConverter());
            restTemplate.exchange(BASE_URL, HttpMethod.POST, request, Party.class);
        } catch (Exception e) {
            Log.e( tag: "PartiesTask", e.getMessage(), e);
        }
        return null;
    }
}
```

Figura 19: AsyncTask para crear una fiesta

Las tareas asíncronas exponen dos métodos que podemos sobrescribir, el método doInBackground() y onPostExecute(). En el primero de ellos se establece el comportamiento de la tarea asíncrona, que en este caso consiste en la creación y ejecución de la petición HTTP POST guardando los datos de una fiesta que ha sido creada. El método onPostExecute() no es necesario en esta task, pero si necesitamos programar comportamiento en relación a la respuesta HTTP que nos de la API, deberíamos programarla aquí. Es lo que se hace en otras AsyncTasks como la de consultar fiestas, consultar datos de un usuario, etc.

5. Conclusiones

En la presente memoria se ha realizado un estudio comercial de la obtención de rentabilidad en una aplicación Android, se ha propuesto un diseño de una aplicación que conecta a organizadores de eventos o fiestas y a usuarios con interés en el mundo de la diversión, y se ha desarrollado dicha aplicación con una serie de arquitecturas y frameworks elegidos según las necesidades del proyecto.

En el mundo del desarrollo actual, las aplicaciones móviles presentan un futuro apasionante. Hay ciertas personas en la sociedad actual, especialmente los jóvenes, que sólo accede a internet a través de su teléfono móvil, desplazando al ordenador de sobremesa o portátil. Y los móviles cada vez son capaces de más cosas, llegando incluso a competir en requisitos con los ordenadores más modestos. Cabe esperar que el desarrollo de software se adapte a esta situación, ofreciendo aplicaciones desarrolladas en lenguajes nativos, que permiten aprovechar al máximo nuestro Smartphone.

Se ha comprobado además las ventajas de aplicar una metodología de desarrollo ágil, que ha permitido responder de mejor manera a los cambios, y establecer una planificación dinámica del desarrollo, cuyo contenido se ha decidido una vez inicialmente y se ha modificado después de cada iteración o sprint.

La arquitectura final, que incluye 2 aplicaciones y una base de datos, ha resultado perfecta para los objetivos del proyecto. Siempre se podrían haber explorado otras opciones más simples, como tener todo en el lado del cliente, pero no nos hubiera permitido adentrarnos de fondo en el mundo de las comunicaciones servidor-cliente que tan utilizado en el mundo de desarrollo de software. Se puede argumentar cierta sobre-ingeniería en este aspecto, pero se justifica por lo aprendido por el camino.

Por último, cabe destacar que mi situación personal y laboral ha impactado de manera brutal en el desarrollo. Cuando empecé a programar, a la vez comencé en mi primer trabajo, y hoy, casi dos años después, puedo identificar infinidades de mejoras implementadas que no estaban previstas inicialmente, pero que fueron fruto de lo aprendido en el trabajo.

5.1. Mejoras y trabajos futuros.

Como es normal en cualquier proceso de desarrollo, tras la finalización de la primera versión de KELOKE conlleva una larga lista de posibles mejoras identificadas.

El aspecto más importante sin duda es el sistema de pagos: no se ha implementado ninguno. El motivo es que para poder tramitar pagos en Android se necesita realizar una subscripción no gratuita con alguna pasarela de pago como Redys, y dado que el fin de la aplicación aún no es estar en producción, se decidió mantener el pago fuera de la primera versión de KELOKE.

La segunda más importante sería sustituir la base de datos. Una base de datos H2 no es aceptable en producción, ya que no persiste más allá de la ejecución actual. Necesitaríamos implementar otra solución, siendo las bases de datos tipo Oracle la más adecuada pues la integración sería casi inmediata.

Otro aspecto que me parece muy relevante es la interfaz de usuario. Siendo honesto, no puedo decir que sea de lo que más orgulloso me siento. No soy un diseñador ni maquetador, me dedico más a la parte de la lógica de dominio, normalmente ejecutada en el lado del servidor, lo cual se ha visto reflejado en el resultado. Para una siguiente versión, lo mejor sería abandonar el paradigma de layouts y XML, y pasar a implementar una interfaz en Javascript.

Uno de los aspectos peligrosos que plantea la aplicación, si deseamos seguir adelante con ella, es la ausencia total de test unitarios. Estos test, tanto en la aplicación cliente como en la servidor, son muy importantes de cara a seguir desarrollando, ya que nos permiten comprobar si con el nuevo código que creamos se rompe alguna funcionalidad ya existente.

Dentro de detalles más técnicos y de arquitectura, considero que el diseño de la aplicación cliente, KELOKE, no es el mejor. Se ha optado por seguir fielmente el paradigma de actividad de Android, aunque quizás una arquitectura orientada a servicios [9] nos permitiría un dividido funcionalmente con una estructura más legible y mantenible.

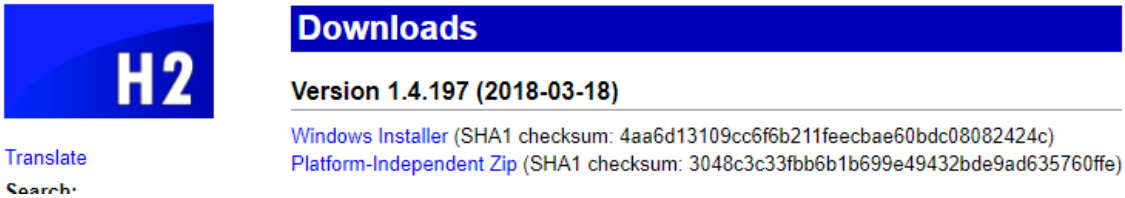
6. Bibliografía

[1] *IOs contra Android, ¿quién gana la batalla?* Artículo online disponible pinchando aquí.
[2] *Agile vs Watterfall again? Now in numbers*. Artículo online disponible pinchando aquí.
[3] Patron Singleton, explicación online pinchando aquí.
[4] Firebase, base de datos para aplicaciones móviles no relacional.
[5] Lombok, plugin de generación de código en base a anotaciones.
[6] Pivotal, empresa dueña de Spring y Spring Boot
[7] Swagger, herramienta para exposición y testeo de APIs.
[8] Postman, cliente REST.
[9] SOA, Service Oriented Architecture

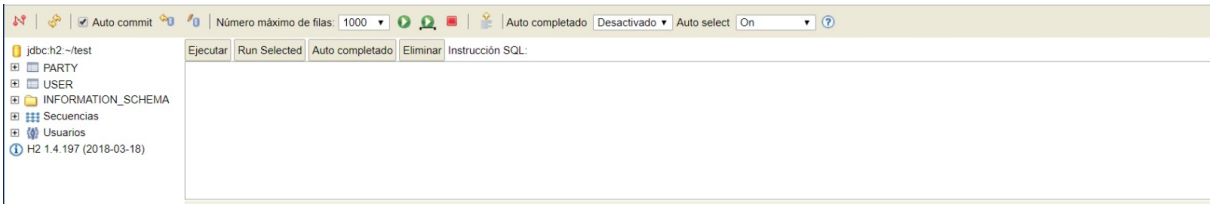
Apéndice I. Manual de instalación del sistema

En este apéndice listaremos los requisitos técnicos necesarios para poder ejecutar el código desarrollado en un ordenador. Para ello necesitaremos arrancar 3 distintos tipos de soluciones de software: una aplicación cliente, una aplicación servidor y una base de datos en memoria.

Empecemos por la base de datos. Para arrancar una base de datos H2, no tenemos más que descargar el instalador a través del siguiente enlace.



Tras descargar el ejecutable, con un simple doble click entraremos en la consola de H2. Con los valores por defecto, pinchamos en conectar y ya tendremos la base de datos levantada y escuchando consultas.



Sigamos por la API. Para poder ejecutarla necesitaremos dos cosas, la JDK8 de java y el entorno de programación IntelliJ. En el caso de la JDK sólo se asegura compatibilidad con la 8, no habiendo probado el funcionamiento con JDKs superiores y asegurando incompatibilidad con JDKs anteriores. Respecto a IntelliJ, en principio no es necesario, podríamos utilizar otro entorno de desarrollo, pero en el que se ha usado durante todo el proyecto es IntelliJ.

Podemos descargar la JDK en el siguiente enlace desde la página web de Oracle. Deberemos elegir la versión que sea acorde con nuestro sistema operativo.

Java SE Development Kit 8u181

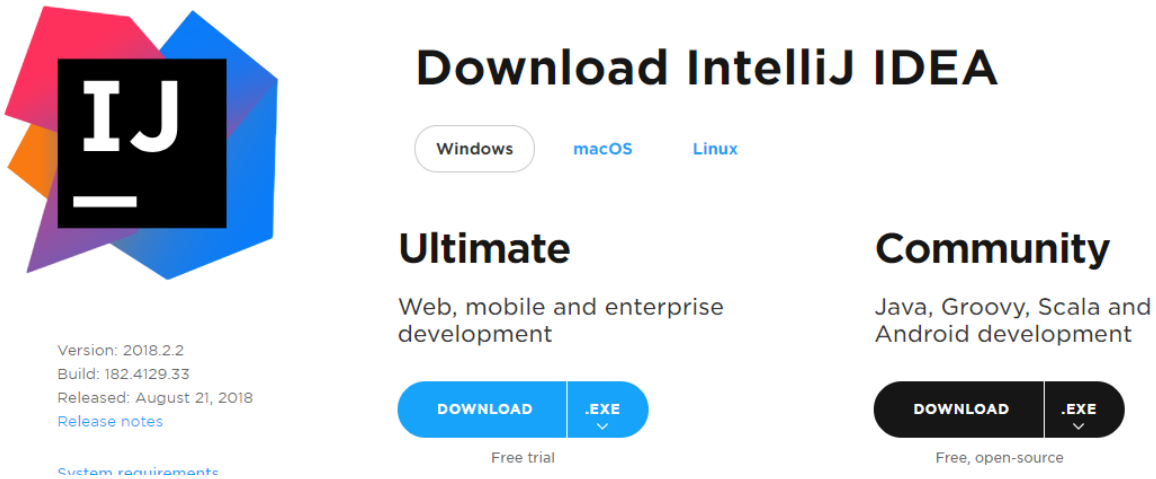
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement Decline License Agreement

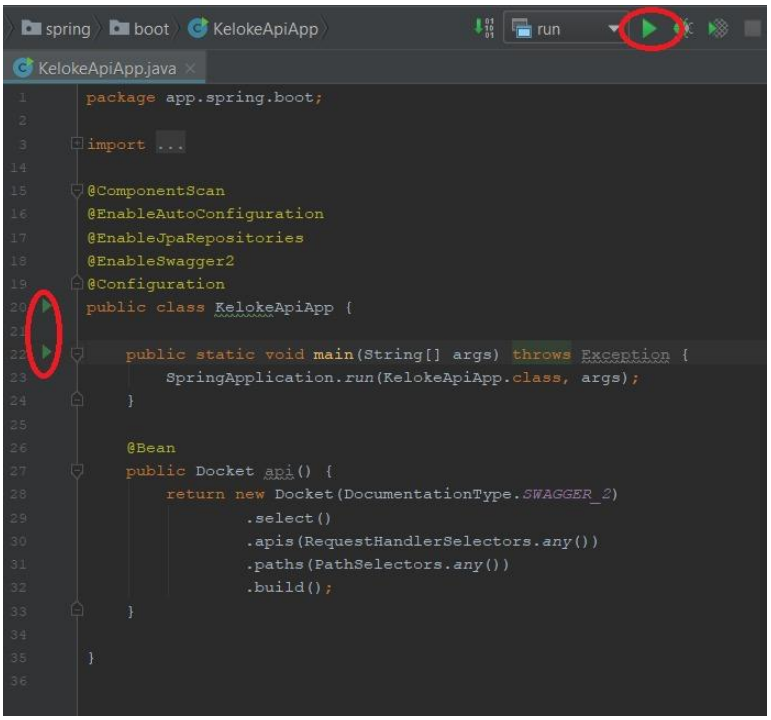
Product / File Description	File Size	Download
Linux ARM 32 Hard Float ABI	72.95 MB	jdk-8u181-linux-arm32-vfp-hflt.tar.gz
Linux ARM 64 Hard Float ABI	69.89 MB	jdk-8u181-linux-arm64-vfp-hflt.tar.gz
Linux x86	165.06 MB	jdk-8u181-linux-i586.rpm
Linux x86	179.87 MB	jdk-8u181-linux-i586.tar.gz
Linux x64	162.15 MB	jdk-8u181-linux-x64.rpm
Linux x64	177.05 MB	jdk-8u181-linux-x64.tar.gz
Mac OS X x64	242.83 MB	jdk-8u181-macosx-x64.dmg
Solaris SPARC 64-bit (SVR4 package)	133.17 MB	jdk-8u181-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	94.34 MB	jdk-8u181-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	133.83 MB	jdk-8u181-solaris-x64.tar.Z
Solaris x64	92.11 MB	jdk-8u181-solaris-x64.tar.gz
Windows x86	194.41 MB	jdk-8u181-windows-i586.exe
Windows x64	202.73 MB	jdk-8u181-windows-x64.exe

.

También de manera gratuita, podemos descargarnos la versión de prueba de IntelliJ pinchando aquí.

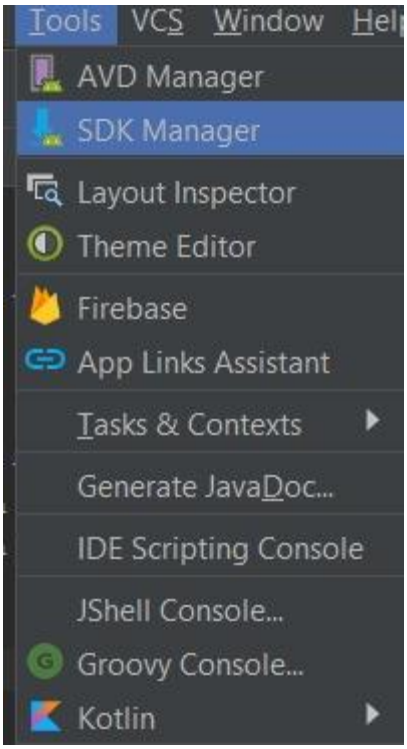


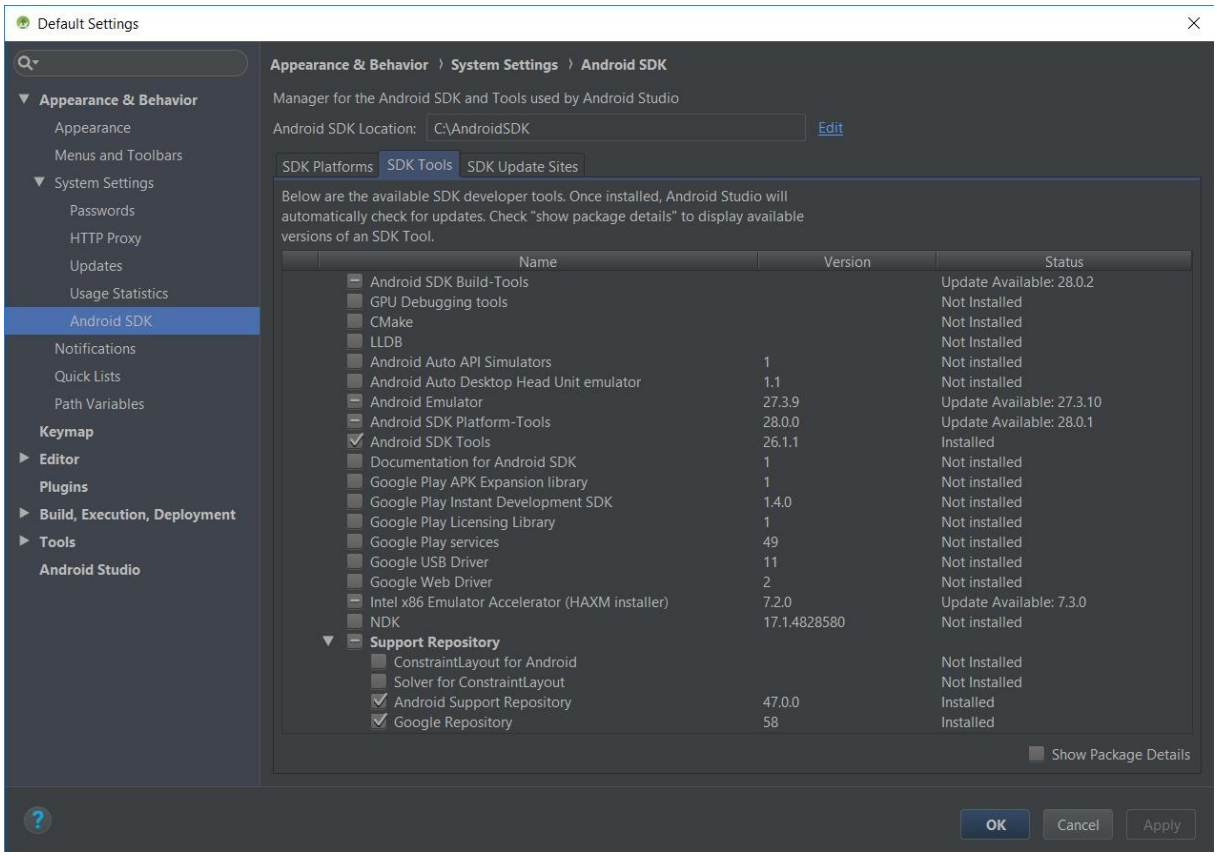
Una vez hecho esto, pasamos a abrir con IntelliJ el código fuente incluido en API.rar, suministrado en el CD del TFG. Establecemos nuestra JDK e IntelliJ debería autoconfigurar un botón de ejecución automática. Si no es así, podemos ir a nuestra clase KelokeApiApp, y hacer click en el método main.



Tras darle al play, spring boot arrancará un servidor tomcat que ejecutará el código de la API.

Tras hacer esto, tendremos API y BBDD conectadas, nos falta levantar un cliente que las ataque. Para ello necesitaremos descargar Android Studio y la SDK26. Podemos descargar el entorno desde aquí, lugar desde donde también podemos descargar la SDK o, si lo preferimos, podremos descargarla desde Android Studio siguiendo la interacción de las siguientes imágenes:

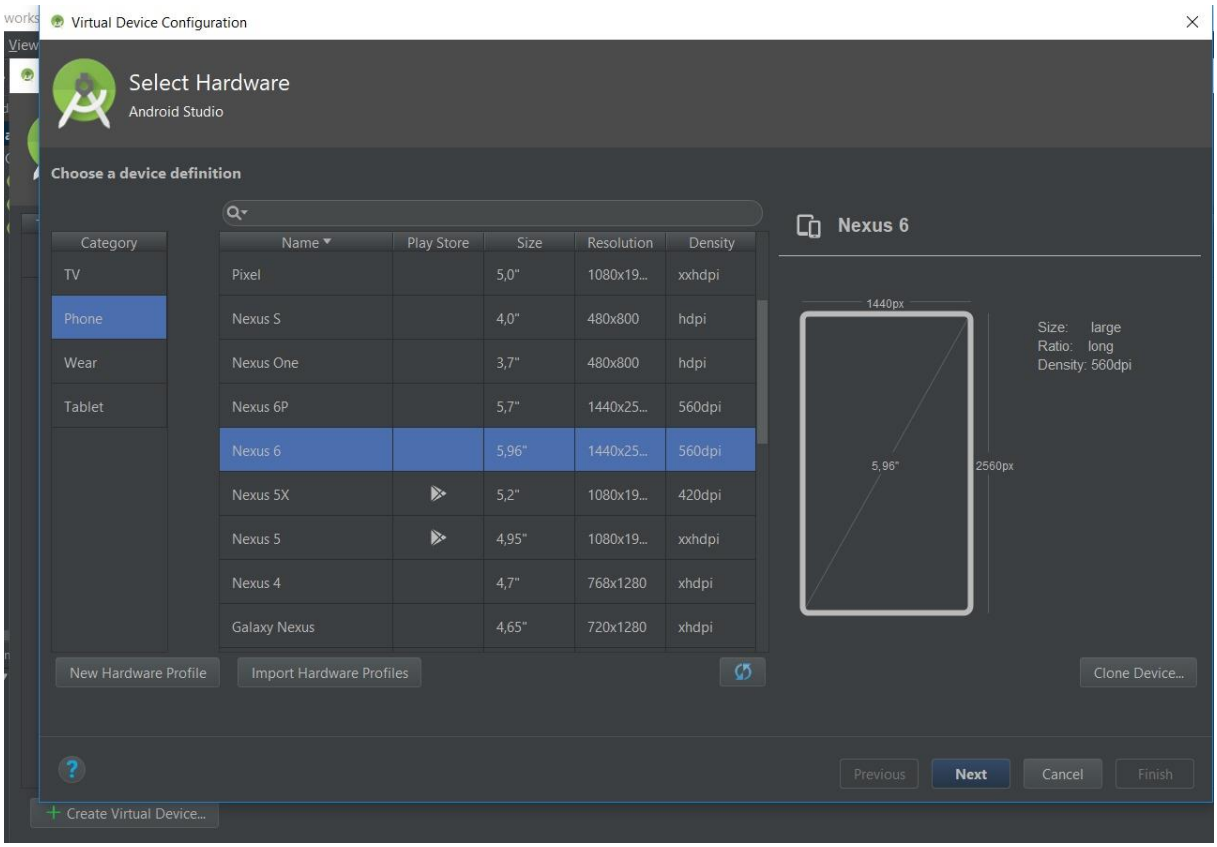

Tools
VCS
Window
AVD Manager
SDK Manager
Layout Inspector
Theme Editor
Firebase
App Links Assistant
Tasks & Contexts
Generate JavaDoc...
IDE Scripting Console
JShell Console...
Groovy Console...
Kotlin


Default Settings
Appearance & Behavior
Appearance
Menus and Toolbars
System Settings
Passwords
HTTP Proxy
Updates
Usage Statistics
Android SDK
Notifications
Quick Lists
Path Variables
Keymap
Editor
Plugins
Build, Execution, Deployment
Tools
Android Studio
Appearance & Behavior › System Settings › Android SDK
Manager for the Android SDK and Tools used by Android Studio
Android SDK Location: C:\AndroidSDK
Edit
SDK Platforms
SDK Tools
SDK Update Sites
Below are the available SDK developer tools. Once installed, Android Studio will automatically check for updates. Check "show package details" to display available versions of an SDK Tool.
Name
Version
Status
Android SDK Build-Tools
Update Available: 28.0.2
GPU Debugging tools
Not Installed
CMake
Not Installed
LLDB
Not Installed
Android Auto API Simulators
1
Not installed
Android Auto Desktop Head Unit emulator
1.1
Not installed
Android Emulator
27.3.9
Update Available: 27.3.10
Android SDK Platform-Tools
28.0.0
Update Available: 28.0.1
Android SDK Tools
26.1.1
Installed
Documentation for Android SDK
1
Not installed
Google Play APK Expansion library
1
Not installed
Google Play Instant Development SDK
1.4.0
Not installed
Google Play Licensing Library
1
Not installed
Google Play services
49
Not installed
Google USB Driver
11
Not installed
Google Web Driver
2
Not installed
Intel x86 Emulator Accelerator (HAXM installer)
7.2.0
Update Available: 7.3.0
NDK
17.1.4828580
Not installed
Support Repository
ConstraintLayout for Android
Not Installed
Solver for ConstraintLayout
Not Installed
Android Support Repository
47.0.0
Installed
Google Repository
58
Installed
Show Package Details
OK
Cancel
Apply

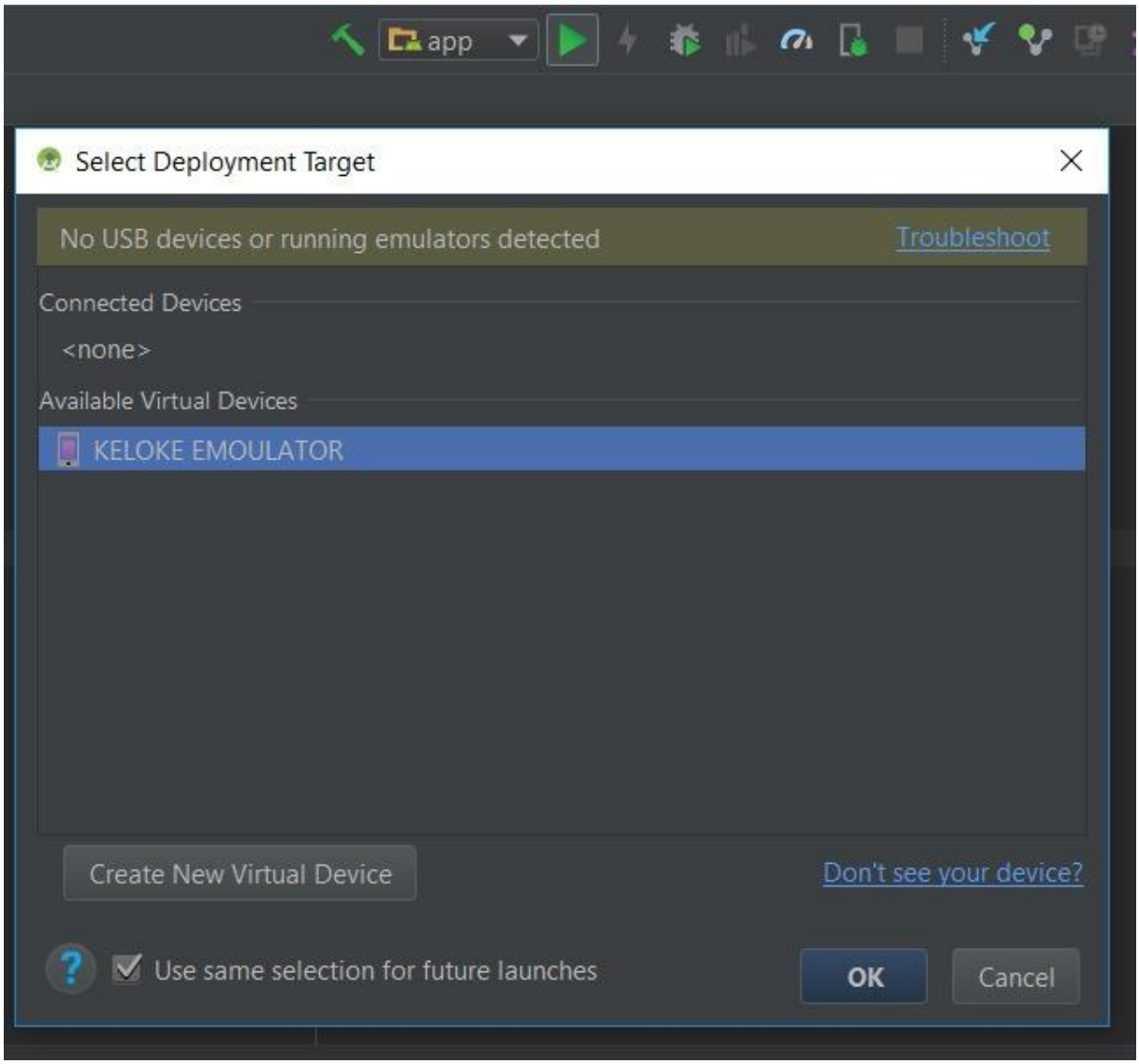
Es importante que la SDK sea la 26, ya que hay algunas restricciones relativas a la versión de la SDK en la aplicación. Por ejemplo, si la SDK elegida es anterior a la 21, la gestión de permisos de geolocalización fallará ya que se utilizan funcionalidades desarrolladas a partir de la SDK 26. Este aspecto es importante y ha sido decisivo en el proyecto, ya que las versiones de la SDK son inexplicablemente no retro compatibles, alejándose de la filosofía de Java. Cualquier otra SDK permitirá ejecutar la aplicación, pero el correcto procesamiento de las funcionalidades no está asegurado.

Ya tendremos Android Studio listo para ejecutar nuestro código. Lo único que necesitamos ahora es un dispositivo donde hacerlo. Durante el desarrollo de todo el proyecto se ha utilizado el emulador Android que tiene integrado Android Studio para ejecutar las pruebas. No se asegura una correcta ejecución utilizando otros dispositivos, como un dispositivo real conectado por USB, ya que no se ha tenido en cuenta esta posibilidad en el desarrollo.

Para crear un nuevo dispositivo, vamos a Tools -> AVD Manager. Clicamos en nuevo dispositivo y rellenamos los datos de la SDK que queremos ejecutar. El emulador nos permite utilizar distintos tipos de dispositivos físicos, de entre los cuales se ha utilizado Nexus 6, que es el que se recomienda.



Tras esto, damos click a siguiente de manera repetitiva hasta que el emulador haya sido creado. Es importante seleccionar la SDK adecuada para la ejecución. Ahora tenemos todo lo necesario para ejecutar la aplicación cliente, así que lo hacemos desde Android Studio, que nos pedirá que seleccionemos un emulador de los disponibles.



Deberemos seleccionar el dispositivo creado anteriormente, o crear uno nuevo en caso de que no se haya hecho, y ya podemos “jugar” con la aplicación. Cabe destacar que el proceso debe seguirse en este orden: la API necesita a la BBDD levantada para poder arrancar, y la aplicación Android hará peticiones a la API, por lo que si ejecutamos la aplicación si tener la API levantada nada funcionará.

Apéndice II. Descripción de los contenidos suministrados

En el CD suministrado se encuentran los siguientes contenidos:

- **Memoria.pdf** – La memoria del TFG.
- **KELOKE.rar** – Código fuente de la aplicación cliente.
- **API.rar** – Código fuente de la aplicación servidor.
- **Video demo.mkv** – Vídeo de demostración de la aplicación.

NOTA: Para una correcta instalación del sistema leer apéndice I.