Weather App

Joaquín Lamela

https://github.com/joaquinlamela/WeatherApp

Alcance 2
Control de cambios y de versiones 2
Uso de comentarios en commits 3
Decisiones de Diseño 4
Diseño de la arquitectura de la aplicación móvil 4
Descripción general - MVVM Architecture 4
View 5
ViewModel 5
Model 5
Comunicación entre View y ViewModel 5
LiveData 5
Comunicación entre ViewModel y Model 6
Inyección de dependencias 7
Separación de problemas 7
Single Activity Architecture 7
Navegación 8
Librerías utilizadas 9
KTX 9
Navigation 9
Constraint Layout 9
Material 9
Retrofit 9
Mapas 10
Permisos 10

Alcance

El alcance de la aplicación está dado por las siguientes características:

En primera instancia, la aplicación tendrá un solo rol de cara a los usuarios, debido a que la misma se centra en cualquier persona residente en cualquier parte del mundo, y no busca distinguir roles o promociones diferenciadas.

Todos los usuarios podrán acceder a las dos funcionalidades presentes dentro de la aplicación, la cual consta de una Bottom Navigation Bar, en donde el usuario podrá seleccionar la pantalla a visualizar. Dichas pantallas son:

- Visualizar el clima para el día de hoy y para los siguientes 7 días para la ubicación actual.

- Visualizar un mapa en el cual el usuario pueda seleccionar un punto del mismo y se muestre el clima para ese marcador específico.

Control de cambios y de versiones

El uso de ramas, para administrar y diferenciar el trabajo, y a su vez para identificar distintos ambientes (producción, pruebas, desarrollo) son de las principales utilidades que dispone Git, esto nos ayuda a llevar un mejor control del código y del proyecto.
Una rama se trata de una bifurcación del estado del código que crea un nuevo camino de cara a la evolución del código, en paralelo a otras ramas que se puedan generar. En nuestro caso, decidimos hacer uso de Git mediante 2 ramas fijas (master y develop) y otros 3 tipos de ramas: features (funcionalidades), bugfix (bugs) y refactor.

La rama master se utiliza cada vez que se llega a una etapa estable que tuviera algún flujo completamente cerrado. Sería como la rama de producción de nuestro proyecto.

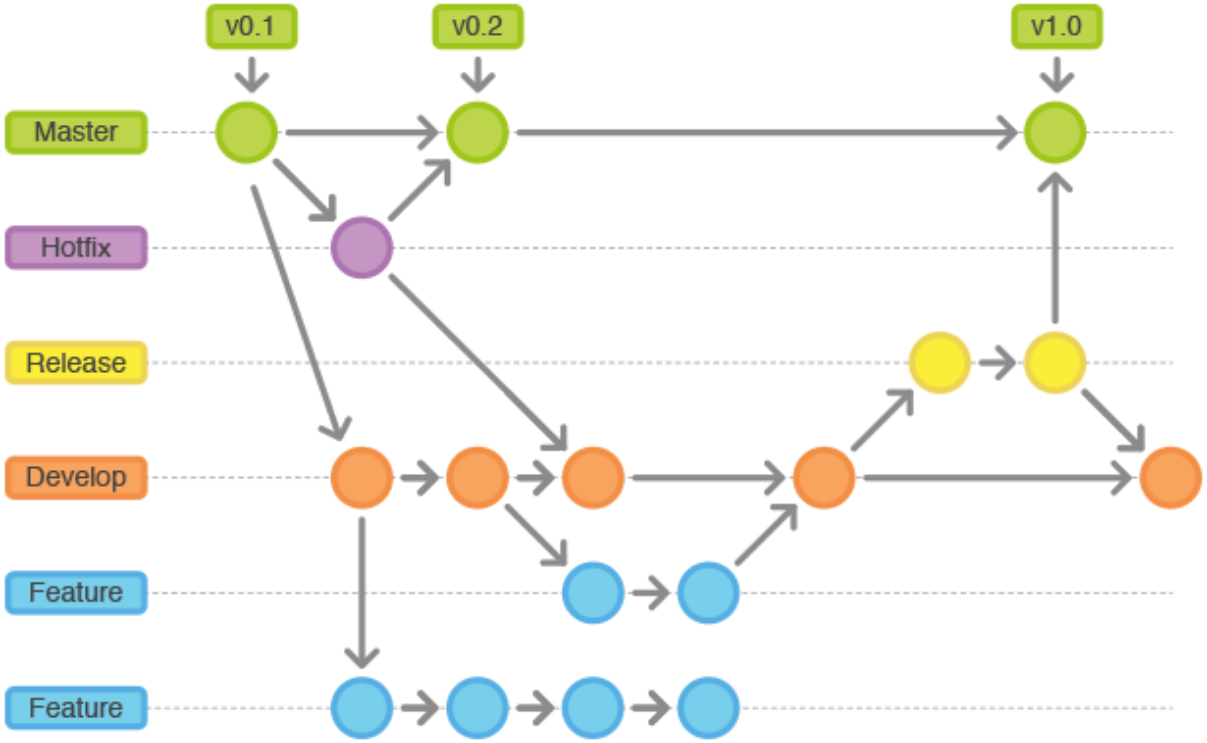
La rama develop es donde mergeamos cuando una de las ramas features/bugfix/refactor fue revisada y no tiene errores (por lo menos al probar manualmente el desarrollador). Sería como la rama de pruebas para que el cliente pueda ver la última versión del producto y pueda probar.

Para cada nuevo trabajo que realizamos, creamos una rama que partiera de develop y cuando el trabajo está finalizado y testeado, subimos el branch y creamos un pull request.

Luego está el proceso de validación del pull request creado, se encarga de hacer una revisión al código. En caso de que considere que todo está funcionando correctamente, se aprueba el pull request y se hace un merge de la rama sobre develop. Luego el flujo se reinicia. En caso contrario, se revisan los errores.

En cuanto a las ramas refactor, se crean con los objetivos de: mejorar la facilidad de comprensión del código y eliminar código muerto. En el proceso de refactorización, no se arreglan errores ni incorporamos funcionalidades, sino que alteramos la estructura interna del código sin cambiar el comportamiento externo.

Las principales ventajas de utilizar este modelo son, entre otras, disminuir la posibilidad de mezclar ramas, independizar partes del código para el momento que entregamos a producción, el equipo de pruebas recibe más rápido código para probar y así evitamos que muchos bugs se "pasen por alto" o que lleguen a producción por accidente.



Uso de comentarios en commits

Un commit es la acción de guardar o subir archivos a un repositorio. Los mensajes de commit son una herramienta muy útil en el desarrollo del software, ya que nos permiten encontrar cambios concretos en la historia de nuestro repositorio.
Cuando realizamos un commit, nos enfocamos en que el mensaje sea conciso y claro, de forma que brinde la suficiente información como para determinar qué cambios se realizaron en el commit.

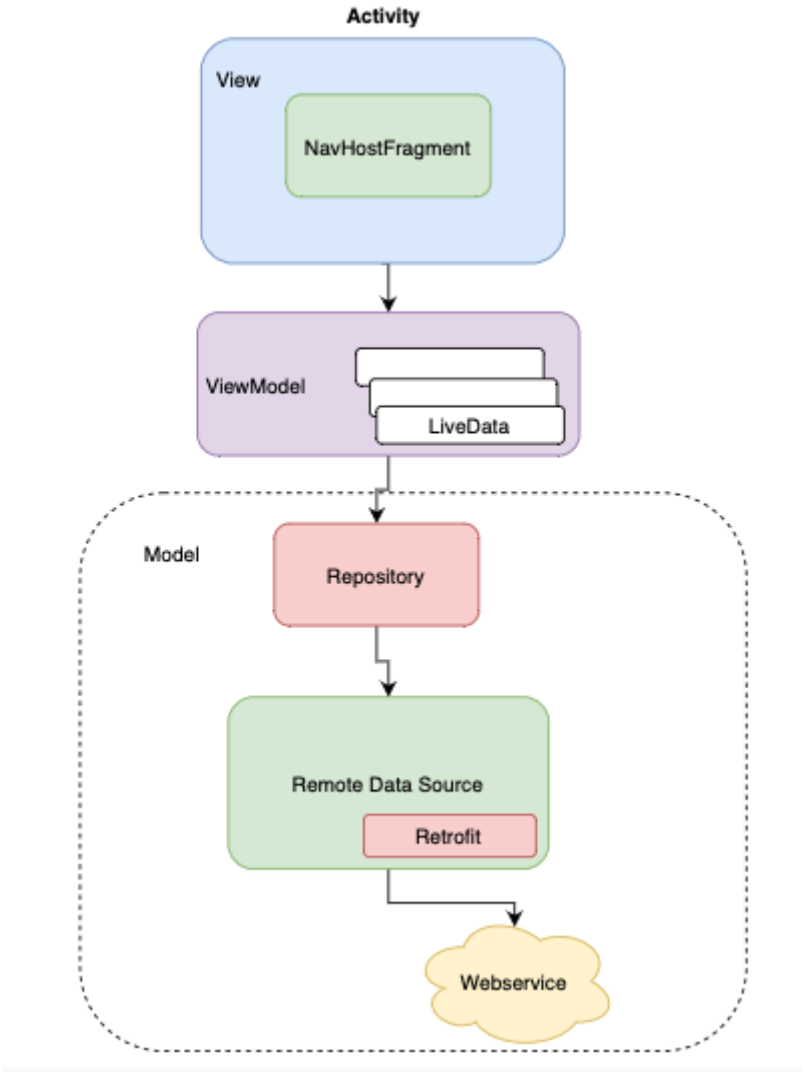
Decisiones de Diseño

Diseño de la arquitectura de la aplicación móvil

Para diseñar la arquitectura de la aplicación móvil, se siguieron las prácticas y arquitecturas recomendadas que permiten el desarrollo de aplicaciones sólidas y de calidad.

Descripción general - MVVM Architecture

En la figura que se muestra a continuación, se observa una arquitectura general de cómo se ha desarrollado la aplicación móvil, la cual se conoce como MVVM (Model-View-ViewModel). Se puede observar cómo cada componente solo depende del componente que está un nivel más abajo (sólo tienen referencia por observables). Por ejemplo, las actividades y los fragmentos sólo dependen de un modelo de vista. El repositorio es la única clase que depende de otras clases. En la aplicación, el repositorio depende de una fuente de datos proveniente desde el backend.



A continuación, se describen a grandes rasgos cada componente de esta arquitectura.

View

Consiste en el código de interfaz de usuario: el fragmento y el diseño XML. Cuando se necesita información, o el usuario realiza una determinada acción, se envía esa petición al ViewModel, pero no se obtiene la respuesta directamente. Para obtener la respuesta, la vista debe suscribirse a los observables que el ViewModel le expone.

ViewModel

Funciona como un puente entre la View y el Model. Un ViewModel proporciona los datos para un componente de UI específico, como un fragmento o una actividad, y también incluye lógica empresarial de manejo de datos para comunicarse con el modelo. ViewModel no conoce los componentes de UI, de manera que no se ve afectado por los cambios de configuración.
Básicamente, lo que hace es interactuar con el modelo y exponer el observable que puede ser observado por la View.

Model

El Model representa los datos y la lógica de negocios en la aplicación Android. En nuestro caso, contiene: modelos, repositorios y fuentes de datos.

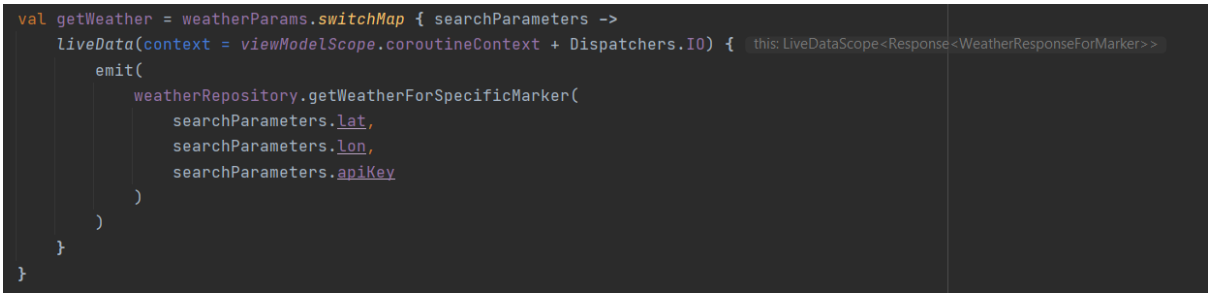
Comunicación entre View y ViewModel

Una vez tenemos por un lado la View y por el otro lado el ViewModel vamos a explicar como realizar la comunicación entre ambos componentes.
El ViewModel va a comunicarse con el Model para traer la data necesaria. Cuando esa data cambia, se le debe informar de alguna forma al fragmento, y en ese momento entra en juego un componente llamado LiveData.

LiveData

LiveData es una clase que retiene datos observables. Otros componentes de nuestra aplicación pueden supervisar cambios en objetos que usan este titular sin crear rutas de dependencia explícitas y rígidas entre ellos.

Usando LiveData, cada uno de nuestros fragmentos pueden recibir notificaciones cuando se actualizan los datos. Para recibir esas actualizaciones, un determinado fragmento debe observar de ese LiveData, para que cuando se actualicen los datos enterarse, y de esa forma, poder actualizar la UI.



Una ventaja que tiene el hecho de utilizar LiveData es que el mismo está optimizado para los ciclos de vidas, lo que implica que las referencias se borran automáticamente cuando ya no son necesarias.

Comunicación entre ViewModel y Model

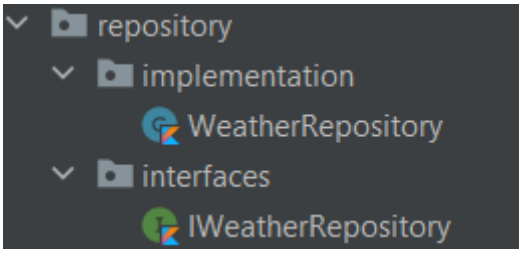
Vamos a explicar la forma en que se comunican los componentes ViewModel con el Model.
Nuestra aplicación cuenta con la utilización de una API externa. En ese sentido, usamos la biblioteca Retrofit para poder utilizar dicha API externa.
En función de eso, tenemos un WebService que lo que hace es comunicarse con los diferentes endpoints que expone la API REST, esto se puede observar en el paquete endpoints.

Seguidamente, vimos cómo diseñar la comunicación entre los WebServices y los ViewModels. Una idea inicial para implementar ViewModel podría consistir en llamar directamente a Webservice a fin de recuperar los datos y asignarlos a nuestro objeto LiveData. Aunque este método funciona, el mantenimiento de nuestra aplicación se complica a medida que crece. Estaríamos asignando demasiada responsabilidad a los ViewModel, lo que implica que no estaríamos cumpliendo el principio de separación de problemas. Además, el alcance de un ViewModel está vinculado a un ciclo de vida de Activity o Fragment, lo que significa que los datos del Webservice se pierden cuando finaliza el ciclo de vida del objeto de UI asociado. Este comportamiento crea una experiencia del usuario no deseable.

En cambio, nuestro ViewModel delega el proceso de obtención de datos a un nuevo módulo, un repositorio.

Los módulos de repositorio manejan las operaciones de datos. Proporcionan una API limpia para que el resto de la app pueda recuperar estos datos fácilmente. Saben de dónde obtener los datos y qué llamadas de API deben hacer cuando se actualizan los datos.

Como se puede observar en la figura que se presenta a continuación de los paquetes y clases de la solución, se nota que tenemos un paquete repositories.implementation y otro repositories.interfaces, es decir, que tenemos interfaces de los repositorios, y para cada uno de ellos implementaciones concretas, ya que de esta forma podemos lograr que el ViewModel dependa de una interfaz de repositorio y no del repositorio concreto, lo que implica que ante un cambio en el repositorio concreto, el ViewModel no se verá afectado debido a que está dependiendo de una abstracción y no de una implementación concreta. Es decir, los ViewModel no saben cómo se adquieren los datos, de forma que en un futuro podemos proporcionarle información de varias implementaciones de obtención de datos diferentes.



Cada una de las implementaciones de los repositorios, usan instancias de los WebServices para poder obtener los datos que se requieren, para eso decidimos utilizar inyección de dependencias de forma manual.

Inyección de dependencias

Las implementaciones de repositorio, necesitan de una instancia de WebService para obtener los datos. Podríamos simplemente crearla, pero para hacerlo, también necesitaríamos conocer las dependencias de la clase Webservice. Esta situación obliga a duplicar el código, ya que cada clase que necesita una referencia a Webservice debe saber cómo construirla con sus dependencias. Si cada clase crea un WebService nuevo, nuestra aplicación tendría un alto consumo de recursos.
Para solucionar este problema, se usa inyección de dependencias (en este caso de forma manual), permitiendo así, que las clases definan sus dependencias sin construirlas. En el tiempo de ejecución, otra clase es responsable de proporcionar estas dependencias.

Separación de problemas

Uno de los principios más importantes que se tomó fue el de la separación de problemas. Un error bastante usual que se comete al desarrollar aplicaciones es el de escribir todo el código en una Activity o Fragment.

Estas clases basadas en UI únicamente deberían contener lógica que se ocupe de interacciones del SO y de la UI. Al mantener estas clases tan limpias como sea posible, podemos evitar muchos problemas vinculados con el ciclo de vida.

Es de suma importancia considerar que nosotros como desarrolladores no somos propietarios de las implementaciones de Activity y Fragment, sino que ellas son clases que representan el contrato entre el SO Android y nuestra app. En función de esto, el S.O puede finalizarlas en cualquier momento, por ejemplo, por tener memoria insuficiente. En ese sentido, nosotros lo que intentamos hacer es reducir la dependencia de estas clases, teniendo únicamente en ellas lo que es la interacción entre el S.O y la UI para poder brindar una experiencia de usuario satisfactoria y mantenible en el tiempo.

Single Activity Architecture

Las actividades son componentes a nivel del sistema que facilitan una interacción gráfica entre tu app y Android. La clase de actividad también permite que nuestra app reaccione a los cambios de Android, como cuando la IU de nuestra app ingresa al primer plano o sale de él, rota, etc.

Si bien las actividades son los puntos de entrada proporcionados por el sistema en la interfaz de usuario de su aplicación, su inflexibilidad cuando se trata de compartir datos entre sí y transiciones las ha convertido en una arquitectura que no se recomienda para construir nuestra aplicación.

Dentro del contexto de la app, las actividades deben servir como un host para la navegación y deben contener la lógica y el conocimiento de cómo hacer la transición entre pantallas,

pasar datos, etc. Sin embargo, siguiendo esta arquitectura, consideramos administrar los detalles de la UI en una parte más pequeña y reutilizable, como son los fragmentos.

Por lo cual se tiene una activity la cual contiene un host de navegación, que es un contenedor vacío en el que se intercambian los destinos a medida que un usuario navega por la aplicación. De esa forma, la main activity se asocia con un gráfico de navegación, y contiene un NavHostFragment que es responsable de intercambiar los destinos según sea necesario.

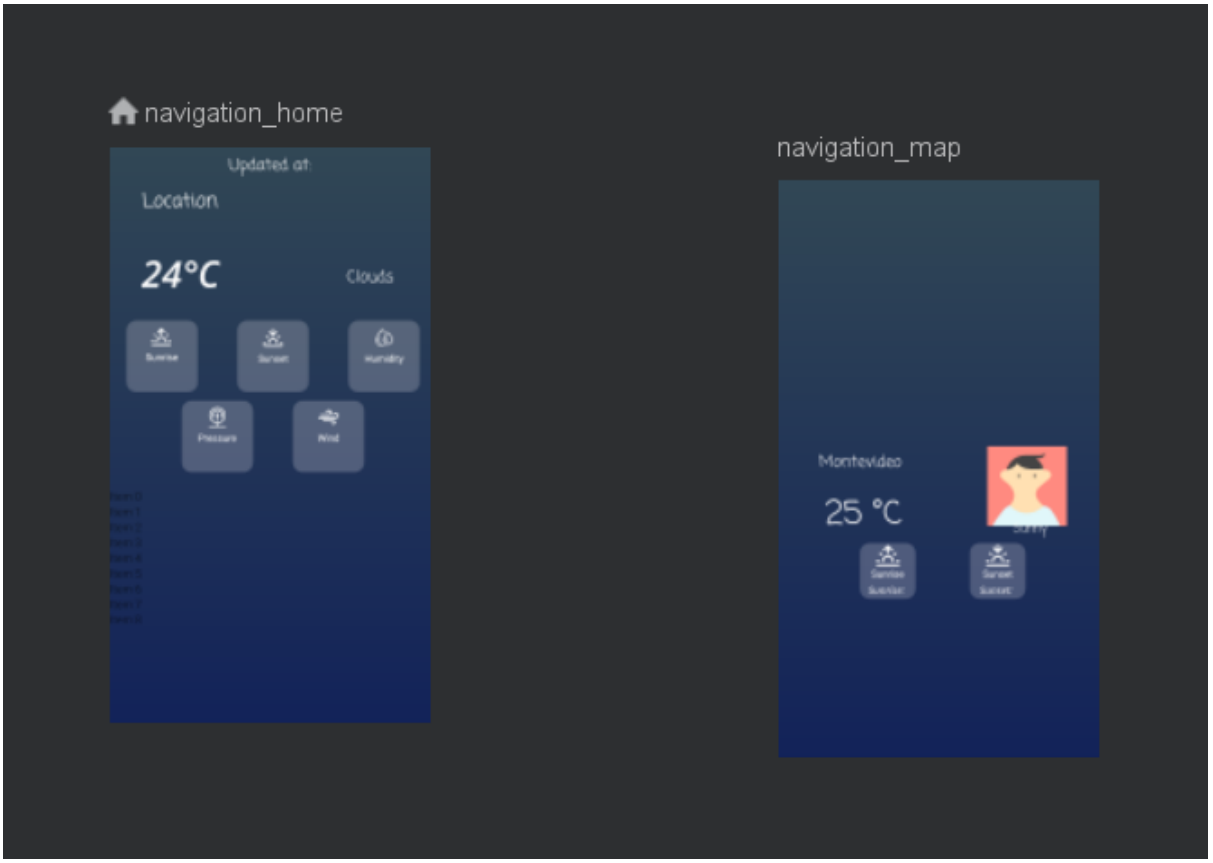
Navegación

Como se mencionó, la aplicación cuenta de una única activity que contiene un NavHostFragment que se va intercambiando con los diferentes fragmentos a medida que el usuario va navegando por la aplicación.

La navegación ocurre entre los destinos de la aplicación, es decir, en cualquier lugar de la aplicación en el que los usuarios pueden navegar. Estos destinos están conectados a través de acciones.

Para diseñar las navegaciones entre los diferentes fragmentos de nuestra aplicación, utilizamos el gráfico de navegación, en donde el mismo contiene todos los destinos y acciones, representando todas las rutas de navegación de la aplicación.

El grafo de navegación de la aplicación, se puede observar en la figura que se presenta a continuación.



Librerías utilizadas

KTX

Android KTX es un conjunto de extensiones de Kotlin que se incluyen con Android Jetpack y otras bibliotecas de Android. Las extensiones KTX proporcionan Kotlin conciso a Jetpack, la plataforma de Android y otras API.

- implementation 'androidx.core:core-ktx:1.3.2'
- implementation 'androidx.appcompat:appcompat:1.2.0'

Navigation

El componente Navigation de Android Jetpack te permite implementar la navegación, garantizando una experiencia del usuario coherente y predecible, ya que se adhiere a un sistema establecido de conjunto de principios. Para eso, se incluyeron las dependencias que se indican a continuación:

- implementation 'androidx.navigation:navigation-fragment-ktx:nav-version-alpha07'
- implementation 'androidx.navigation:navigation-ui-ktx:nav-version'
- implementation 'androidx.navigation:navigation-fragment-ktx:2.3.5'
- implementation 'androidx.navigation:navigation-ui-ktx:2.3.5'

Constraint Layout

Para crear diseños responsive y que los contenidos se adapten a los diferentes tamaños de pantalla, hicimos uso de Constraint Layout, incluyendo la dependencia que se indica a continuación.

- implementation 'androidx.constraintlayout:constraintlayout:2.0.4'

Material

Material es un sistema de diseño creado por Google para ayudar a los equipos a crear experiencias digitales de alta calidad. Para hacer uso de ella, se incluyó la siguiente dependencia.

- implementation 'com.google.android.material:material:1.3.0'

Retrofit

Retrofit es un cliente de servidores REST para Android, que permite hacer peticiones al servidor tipo: GET, POST, PUT, PATCH, DELETE y HEAD, y gestionar diferentes tipos de parámetros, parseando automáticamente la respuesta a un tipo de datos. Para eso, requerimos de las siguientes dependencias.

- implementation 'com.squareup.retrofit2:retrofit:2.9.0'
- implementation 'com.squareup.retrofit2:converter-gson:2.6.0'
- implementation 'com.squareup.okhttp3:logging-interceptor:4.5.0'

Mapas

Para hacer uso de los mapas de Google y poder acceder a la ubicación del usuario, se agregaron las dependencias que se indican a continuación.

- implementation 'com.google.android.gms:play-services-maps:17.0.1'
- implementation 'com.google.android.gms:play-services-location:18.0.0'

Permisos

Para gestionar los permisos del usuario, se hizo uso de la librería EasyPermissions, ya que es un wrapper que simplifica la lógica básica de permisos del sistema. Para hacer uso de la misma, se agregó la dependencia que se indica a continuación.

- implementation 'com.vmadalin:easypermissions-ktx:1.0.0'