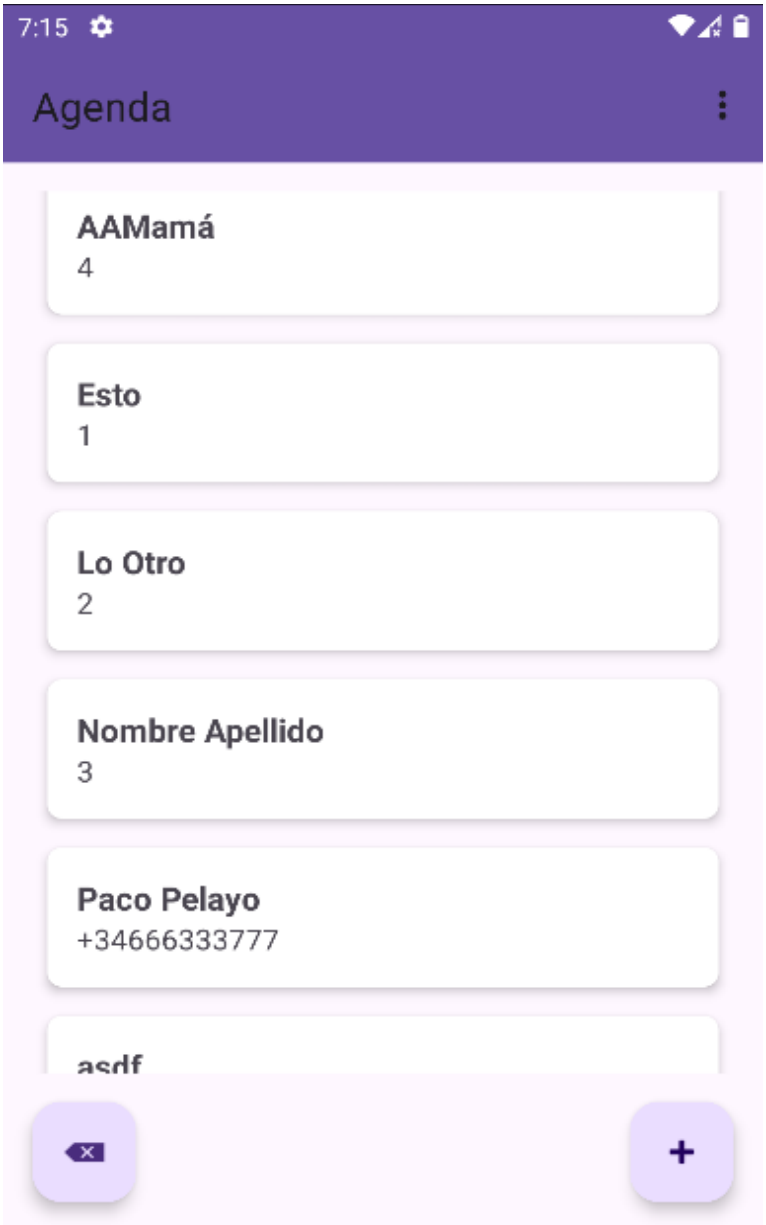
Agenda



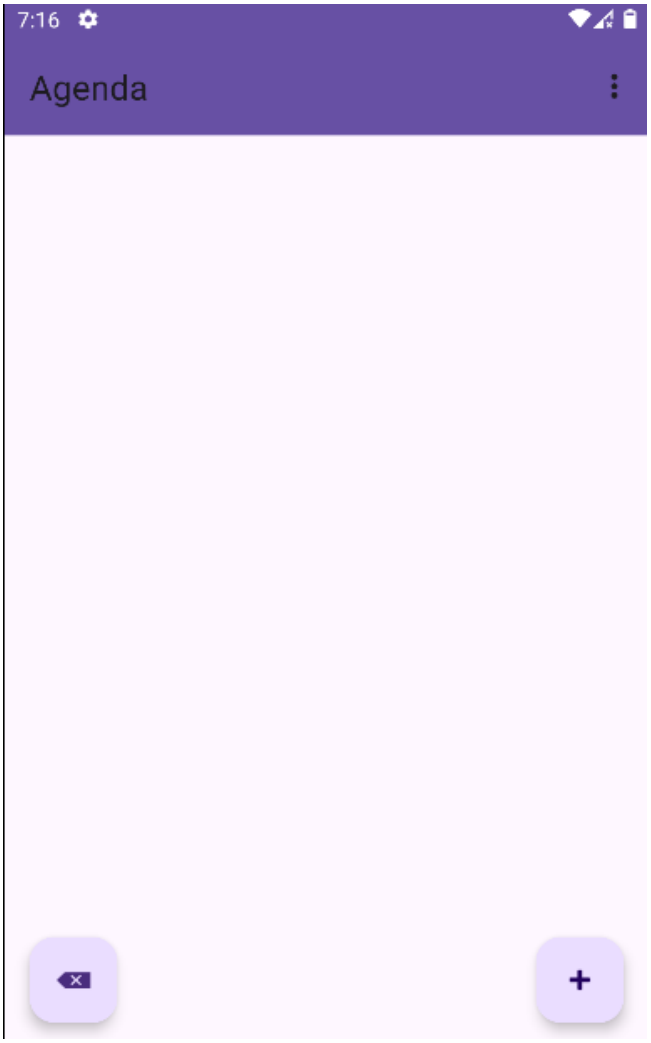
Img. 1 Pantalla principal (ejemplo)

Índice

Funcionamiento

La aplicación consiste en una pantalla principal que es la Activity principal (“MainActivity” que contiene el Fragment inicial “ContactListFragment”), una segunda pantalla (“ContactDetailFragment”) con los detalles del elemento seleccionado (se accede al hacer click sobre el elemento), los detalles sobre el Contact son nombre, teléfono y ubicación (que se muestra en un mapa de Google Maps como Fragment mediante el uso de API keys), varias pantallas que son Fragments y una pantalla que es una segunda Activity (“AboutActivity”) que muestra información acerca de la App.

Al abrir la aplicación, lo primero que veremos será una pantalla en blanco (ya que no hay elementos creados en la base de datos Room), dos FAB (Floating Action Button) para añadir elementos (derecha) o para eliminar todos los elementos (izquierda) y un menú (arriba a la derecha).



Img. 2 Pantalla inicial

Al presionar el FAB (+) se abrirá un DialogFragment (“AgregarContactoDialog”) que nos pedirá nombre del contacto, teléfono y ubicación. Al hacer click en “seleccionar ubicación” se abrirá otro DialogFragment (“SeleccionarUbicacionFragment”) que únicamente contendrá un mapa y un botón para confirmar la ubicación seleccionada. En caso de no gustarnos, podemos volver a hacer click en las coordenadas elegidas para abrir de nuevo el mapa y seleccionar otra ubicación.

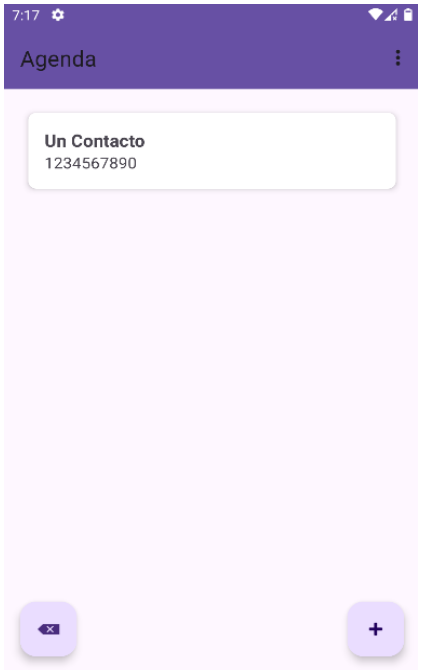
Al final tenemos dos botones (Cancelar y Guardar). Si no queremos crear el elemento, cancelaremos; en caso contrario, al darle a “Save” el elemento se añadirá a la base de datos y se mostrará en la lista de la primera pantalla.

Img. 3 Pantalla Agregar Contacto

Img. 4 Mapa Seleccionar Ubicación

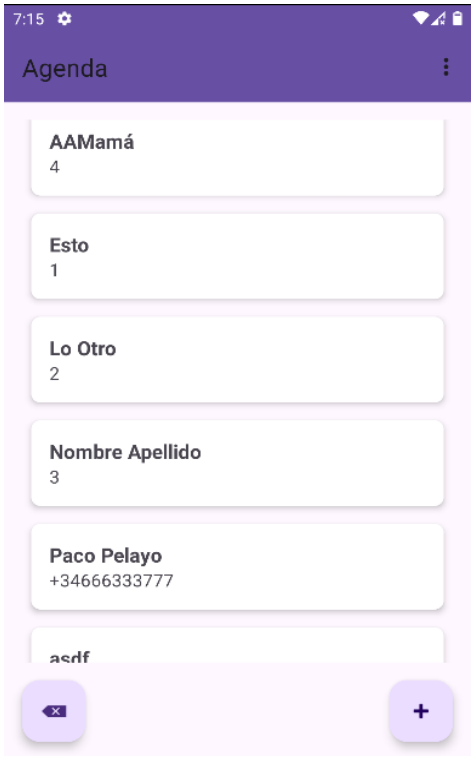
Cancelar y Guardar

Img. 5 Pantalla tras cancelar



Img. 6 Al guardar, se añade el elemento

Tras añadir varios elementos, la lista quedaría así:



Img. 7 Pantalla principal tras añadir varios contactos

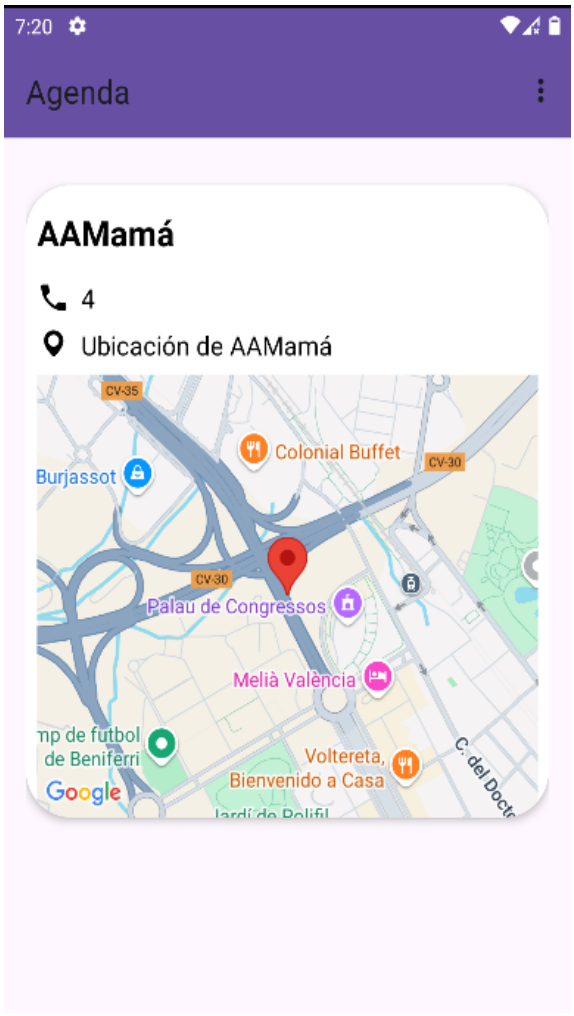
Como se puede apreciar, se han añadido varios elementos para usarlos de ejemplo:

AAMamá, Esto, Lo Otro, Nombre Apellido, Paco Pelayo…

Y aunque no se ve, se puede intuir que hay otro elemento bajo Paco. Bueno, esto es gracias a que se usa una RecyclerView para la lista, lo que permite mostrar una lista "infinita" reciclando los elementos que se muestran.

Si hacemos scroll hacia abajo, veremos que hay más elementos más: asdf entre ellos.

Al hacer click en un elemento de la lista (AAMamá, por ejemplo) se abre el ContactDetailFragment que muestra los datos del Contacto (nombre, teléfono y ubicación en un pequeño mapa).



En un futuro se puede añadir en la parte inferior un botón para editar el contacto, controles al mapa e incluso la opción de compartir el mapa con Intent implícito para abrir la navegación en coche en Google Maps.

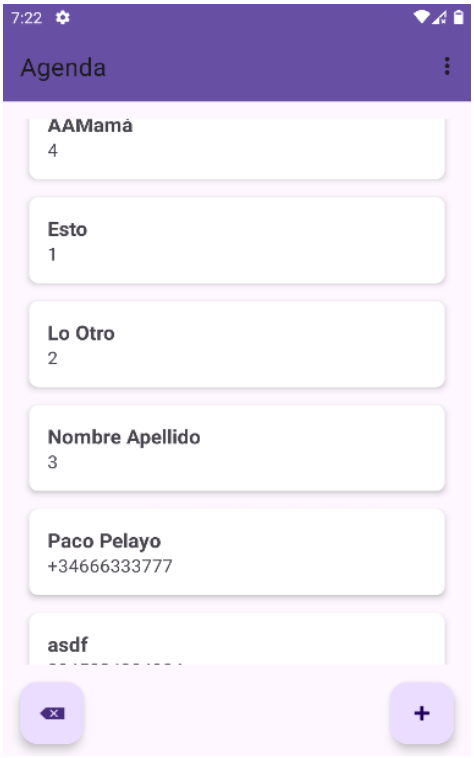
Por el momento es un ejemplo docente bastante sencillo y únicamente muestra los datos añadidos en la fase de creación del Contacto.

Los datos del contacto está dentro de una CardView para darle un estilo “smooth” y agradable, además de tener los bordes redondeados.

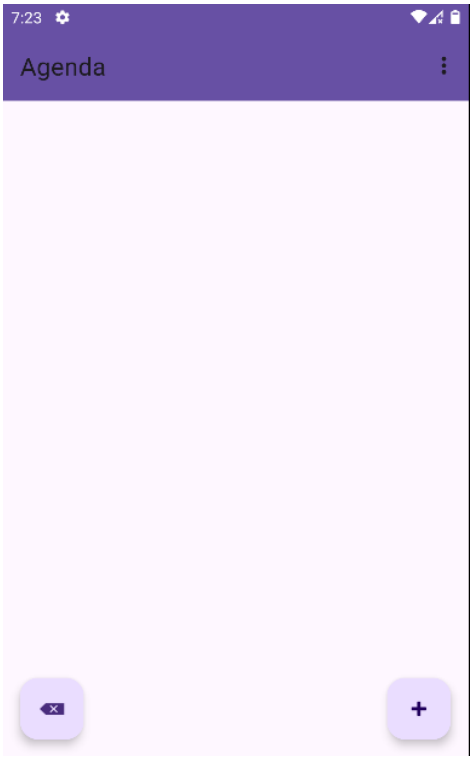
Img. 8 Pantalla Detalles de contacto

Opté por este diseño en lugar de alargar el mapa para ofrecer un diseño armonioso y agradable. Además, a la hora de extender la funcionalidad de la aplicación es mejor aprovechar el espacio añadiendo elementos que rediseñar la interfaz reduciendo el tamaño del mapa para añadir los nuevos elementos.

Eliminar todos los elementos



Img. 9 Lista de elementos de ejemplo



Img. 10 Al borrar, se eliminan los elementos

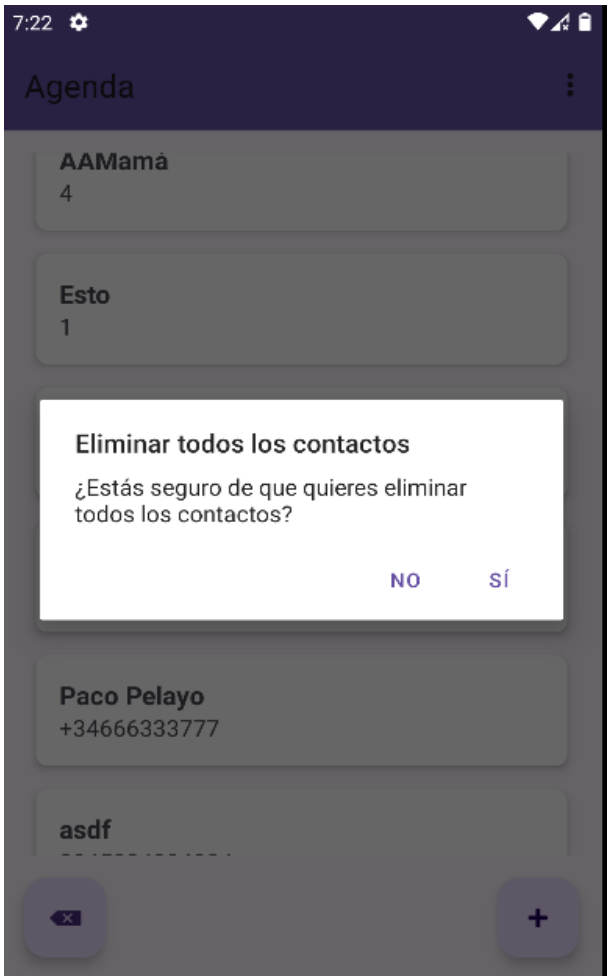
Al pulsar en el botón de borrar (FAB de la izquierda), se abre un DialogFragment que nos pide confirmar que queremos borrar todos los elementos.

Esto se ha hecho porque borrar todos los elementos de la base de datos es una opción irreversible y es una buena práctica solicitar confirmación por parte del usuario.

Y, aunque no se ha implementado el uso del borrado uno a uno de los contactos, a nivel de código sí está implementada esta opción y bastaría con añadir el/los elemento/s correspondiente/s y la llamada adecuada para implementar dicha funcionalidad.

Nuevamente, al tratarse de un ejemplo docente, he optado por simplificar la funcionalidad.

Mensaje de confirmación (eliminar contactos)



En caso de pulsar “no” la lista permanece sin cambios.

En caso de pulsar “sí” se eliminarán todos los elementos de la base de datos y, por tanto, de la lista que se muestra.

Al tratarse de una opción irreversible, se solicita confirmación.

Img. 11 Pantalla Info con la información

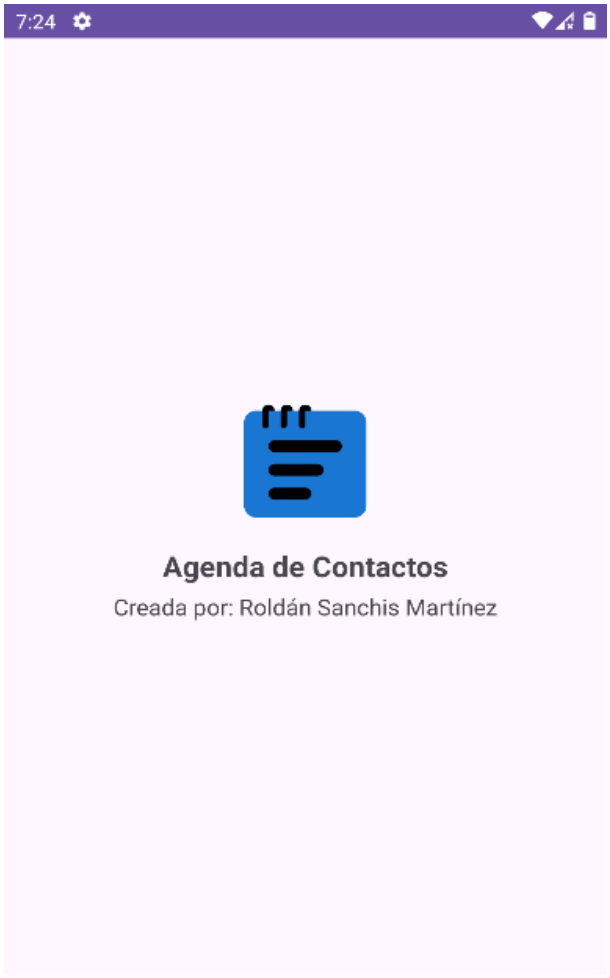
NOTA: Es importante recordar (aunque pueda sonar redundante) que es una buena práctica no realizar cambios drásticos e irreversibles en bases de datos sin añadir un filtro adicional, como es, en este caso, una confirmación para estar seguros de querer realizar la acción que se va a realizar. Imagina que le das sin querer al botón que borra TODOS los datos de una base de datos sin querer y que no existe una breve confirmación de “sí o no” como parapeto final.

Acerca de

Finalmente, la última pantalla está compuesta por una sencilla Activity (“AboutActivity”) con un Layout propio, que contiene un ImageView con el icono de la aplicación y dos TextView, uno con el título de la aplicación “Agenda de Contactos” y el otro con el texto “Creada por: Roldán Sanchis Martínez”.

Se trata de una pantalla muy sencilla que podría haberse resuelto con una Activity sin Layout propio y creada mediante código únicamente, o incluso mediante un Fragment, pero para cumplir con los requerimientos del ejemplo docente, he optado por hacerlo así.

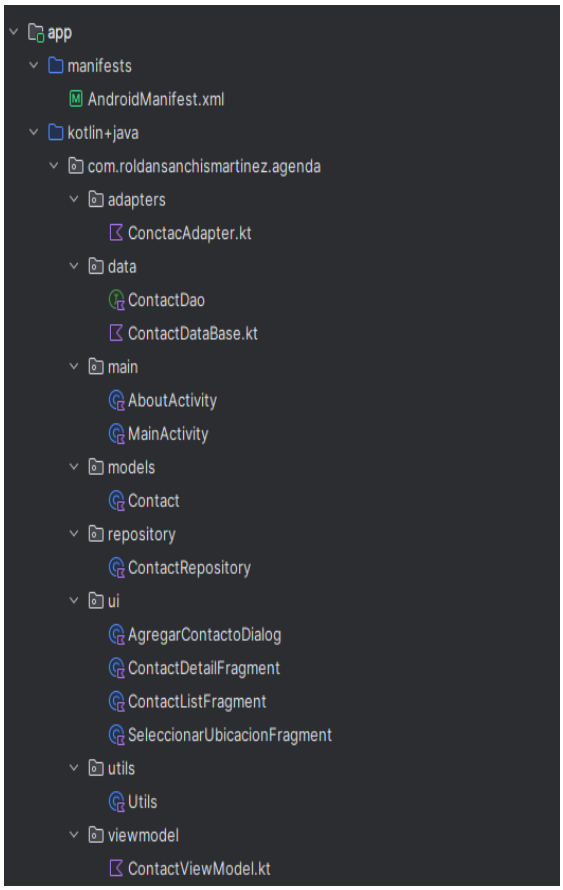
Cabe destacar que el icono de la aplicación se ha cambiado editando el Manifest.



Img. 12 Pantalla Acerca de con la información de la App

Justificación del código

Se ha optado por separar el código según la capa que ocupa/utiliza (p.e. UI para componentes visuales). Y se han implementado varios patrones de arquitectura y diseño de software.



Img. 13 Estructura del proyecto

Se puede apreciar en la imagen (img. 13) la estructura del proyecto dividida por funcionalidad:

El código relacionado con la UI, el modelo, la Base de datos, las Activities (main), el adaptador, el repositorio (mejora la escalabilidad).

Además de un package extra (utils) para añadir funciones adicionales (como por ejemplo parsear una ubicación de String a Pair<Double, Double> para posteriormente convertirse en LatLng y emplearse en el mapa.

Aunque existen diversas formas de estructurar el código dentro de un proyecto, he optado por ésta.

El patrón principal y que mejor se aprecia es el patrón arquitectónico MVVM (Model-View-ViewModel) que permite separar las capas del modelo (datos), las vistas (UI) y el modelo de vista (o ViewModel) que actúa como intermediario entre ambos mediante datos observables (LiveData) que notifican de forma automática estos cambios y a los que se puede asociar eventos. Como podemos ver en los siguientes fragmentos (img 14 y 15):

```
// Observa los cambios en la lista de contactos y actualiza el adapter y la vista
contactViewModel.allContacts.asLiveData().observe(viewLifecycleOwner) { contacts ->
    // Cuando la lista de contactos cambie, actualizamos el adapter
    adapter.setContacts(contacts)
}
```

Img. 14 ViewModel observando LiveData para mantener la UI actualizada con los cambios.

El fragmento mostrado (img. 14) se asegura de que la lista (UI) se actualice cada vez que la lista de elementos cambie, lo que permite que la lista mostrada siempre esté actualizada.

```kotlin
package com.roldansanchismartinez.agenda.viewmodel

import ...

/** ViewModel para la lista de contactos */
class ContactViewModel(private val repository: ContactRepository) : ViewModel() {


    val allContacts: Flow<List<Contact>> = repository.allContacts

    fun insert(contact: Contact) = viewModelScope.launch(Dispatchers.IO) {
        repository.insert(contact)
    }

    fun delete(contact: Contact) = viewModelScope.launch(Dispatchers.IO) {
        repository.delete(contact)
    }

    fun deleteAll() = viewModelScope.launch(Dispatchers.IO) {
        repository.deleteAll()
    }
}

/** Factory para crear instancias de ContactViewModel (Patrón Singleton) */
class ContactViewModelFactory(private val repository: ContactRepository) :
    ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(ContactViewModel::class.java)) {
            @Suppress( ...names: "UNCHECKED_CAST")
            return ContactViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

Img. 15 Clase ContactViewModel

Como se puede apreciar en el código, el ViewModel implementa un patrón Singleton para asegurar que exista una única instancia del mismo durante toda la ejecución del programa.

En otras palabras, se crea un ViewModel la primera vez que se llama, y el resto de veces se devuelve la instancia del mismo.

Otro patrón empleado y muy común en programación y desarrollo es el patrón de diseño Singleton (que ya hemos visto), que se asegura de que una clase tenga una única instancia y proporciona un punto de acceso global a ésta. Y es importante mencionarlo porque no se usa únicamente en el ViewModel, sino que también en otro elemento importante como es la base de datos.

A continuación podemos observar su implementación en la base de datos en la siguiente imagen (imagen 16):

```kotlin
package com.roldansanchismartinez.agenda.data

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.roldansanchismartinez.agenda.models.Contact

@Database(entities = [Contact::class], version = 2, exportSchema = false)
/** Clase abstracta que define la base de datos */
abstract class ContactDatabase : RoomDatabase() {

    abstract fun contactDao(): ContactDao

    /** Patrón Singleton para obtener la instancia de la base de datos */
    companion object {
        @Volatile
        private var INSTANCE: ContactDatabase? = null

        fun getDatabase(context: Context): ContactDatabase {
            return INSTANCE ?: synchronized( lock: this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    ContactDatabase::class.java,
                     name: "contact_database"
                ).fallbackToDestructiveMigration().build()
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Img. 16 Ejemplo de patrón Singleton.

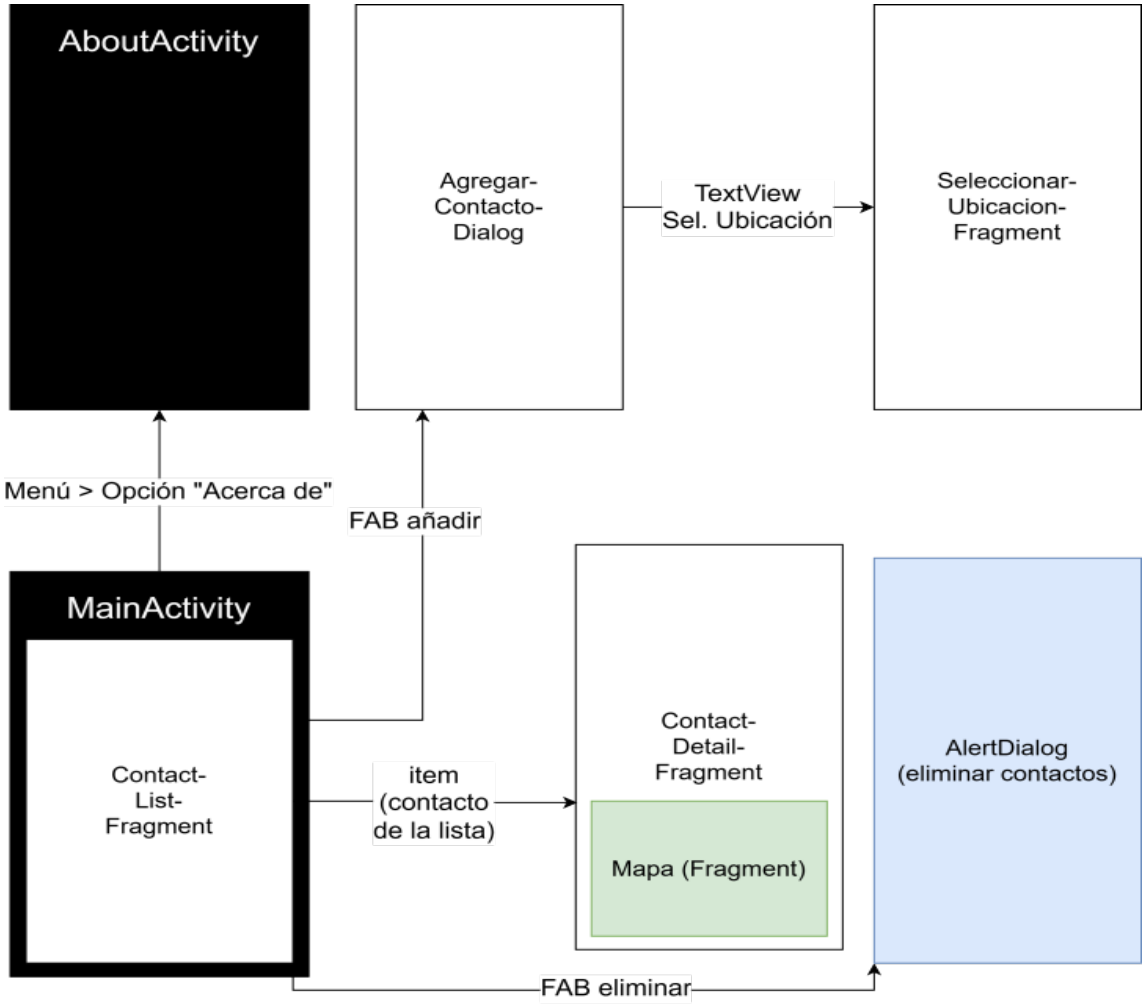
En cuanto a la decisión de usar Fragments en lugar de Activities, surge en base a que la vista principal es una lista mostrada con una RecyclerView y era más sencillo y eficiente de implementar mediante Fragments. Al final como en los requisitos de creación de la aplicación estaban las dos Activities, se optó por añadir un menú de opciones con un botón para abrir la pantalla “Acerca de” y hacer que fuera una Activity en lugar de otro Fragment.

Se ha intentado seguir una filosofía Clean Code a la hora de programar e implementar varios patrones de diseño/arquitectura.

Se ha intentado seguir las convenciones generales en cuanto a nombres de métodos, clases, variables, carpetas y demás elementos.

Finalmente, se muestra la estructura de la aplicación en un breve diagrama (img. 17).

Estructura de la app



Img. 17 Diagrama de la estructura de la aplicación