Tema 2.3 - Arquitectura de una aplicación

Descargar estos apuntes pdf o html

Índice

Introducción

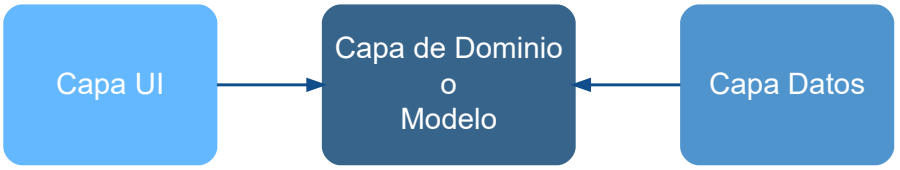
Básicamente una aquitectura software es la estructura de un sistema de software, los componentes del software, las propiedades externamente visibles de esos componentes y las relaciones entre ellos.

La idea tras las arquitecturas es tener capas que separen las responsabilidades de cada una de ellas. De esta forma se consigue que el código sea más mantenible, escalable y testeable. Además, podremos cambiar una capa sin que afecte a las demás.

Actualmente existen arquitecturas complejas de sistemas cómo por ejemplo Arquitectura Hexagonal, Microservicios, Event-driven Architecture, etc. Sin embargo, en el mundo del diseño diseño de aplicaciones para dispositivos móviles tenemos patrones arquitectónicos más sencillas cómo MVC, MVP, MVVM, etc.

> 📌 **Nota**: Puedes ver más información sobre la arquitectura propuesta por Google para Android en Guía de arquitectura de apps

Esquema básico de una aplicación Android propuesto por Google



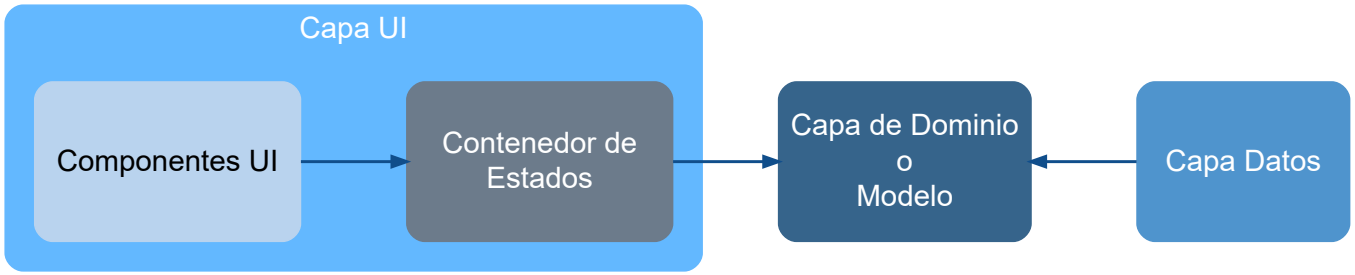
Capa UI

La función de la capa de la IU (o capa de presentación) consiste en mostrar los datos de la aplicación en la pantalla. Cuando los datos cambian, ya sea debido a la interacción del usuario (como cuando presiona un botón) o una entrada externa (como una respuesta de red), la IU debe actualizarse para reflejar los cambios.

La capa de la IU consta de los siguientes dos elementos:

- Elementos de la IU que renderizan los datos en la pantalla (puedes compilar estos elementos mediante las vistas XML o las funciones de **Jetpack Compose**)
- Contenedores de estados (como las clases **ViewModel**) que contienen datos, los exponen a la IU y controlan la lógica

> ✋ **Importante**: Documentación oficial de Android sobre la **Capa de UI** en la que más adelante profundizaremos

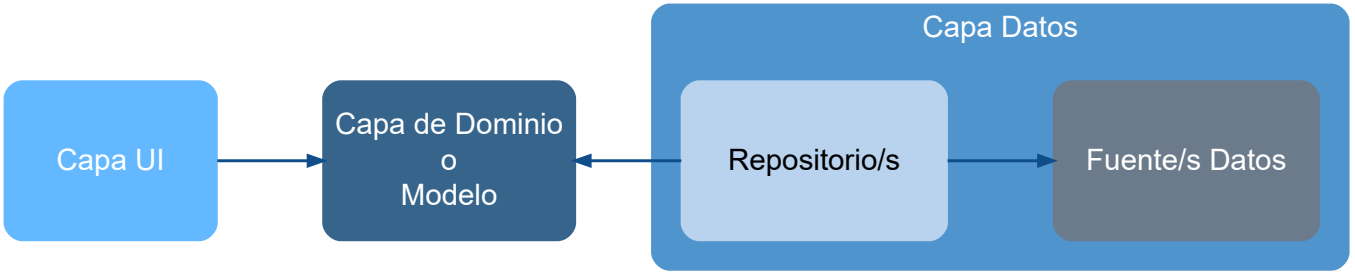


Capa de Datos

La capa de datos está formada por repositorios que pueden contener de **cero a muchas fuentes de datos**. A este tipo de diseño se le denomina **Repository Pattern** y es un patrón de diseño que se utiliza para separar la lógica empresarial de la lógica de acceso a los datos. El repositorio se encarga de obtener los datos de una fuente de datos y convertirlos en un formato con el que el resto de la aplicación pueda trabajar.

> 📌 **Nota**: Este patrón fué introducido **Martin Fowler** en su libro Patterns of Enterprise Application Architecture en el año **2002** y posteriormente desarrollado por **Eric Evans** en su libro Domain-Driven Design en el año **2003**. Hoy en día es un patrón muy utilizado en el desarrollo de software y ámpliamente utilizado en el desarrollo de aplicaciones móviles.

> ✋ **Importante**: Documentación oficial de Android sobre la **Capa de datos** en la que más adelante profundizaremos



Capa de Dominio

La capa de dominio es una **capa opcional** que se ubica entre la capa de la IU y la de datos.

La capa de dominio es responsable de encapsular la lógica empresarial compleja o la lógica empresarial simple que varios ViewModels reutilizan. Esta capa es opcional porque no todas las apps tendrán estos requisitos. Solo debes usarla cuando sea necesario; por ejemplo, para administrar la complejidad o favorecer la reutilización.

Esta terminología viene del **DDD (Domain Driven Design)** que es una metodología de diseño de software que se centra en la lógica empresarial y en la comunicación entre los expertos en el dominio y los desarrolladores.

Sin embargo, nosotros vamos a obviar esta capa y vamos a implementar una arquitectura más sencilla basada en el patrón **MVVM** en el cual esta capa se denomina **Modelo** ya que representará el modelo de datos.

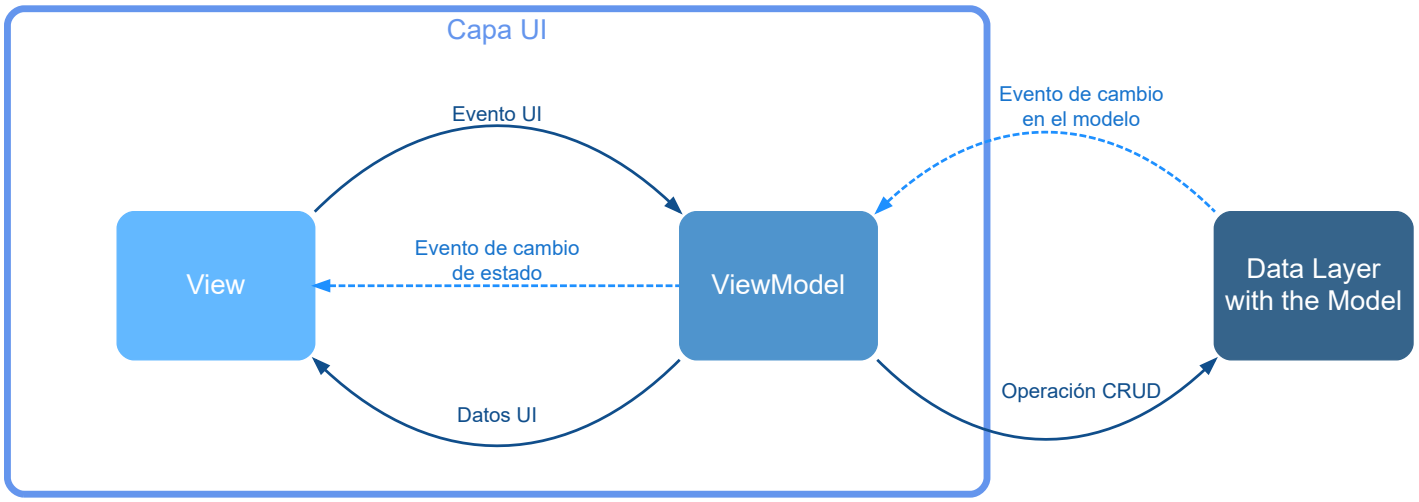
Modelo vista vista modelo MVVM

MVVM es un patrón de diseño de software que facilita la separación de la lógica de presentación de la lógica de negocio.

MVVM es una variante del patrón MVC (Modelo-Vista-Controlador) que se utiliza para la construcción de interfaces de usuario. En MVVM, el ViewModel asume la funcionalidad del "intermediario". En MVVM, toda la lógica de presentación se coloca en el ViewModel.

El ViewModel es responsable de recuperar los datos de los modelos y exponerlos a la vista. El ViewModel también se utiliza para manejar todas las interacciones del usuario. El ViewModel recupera los datos de los modelos, los formatea y los expone a la vista. El ViewModel también acepta las entradas del usuario y las valida antes de actualizar los modelos.

Implementaciones o aproximaciones a esta arquitectura hay muy diversas, pero básicamente la que nosotros vamos a implementar un esquema similar al siguiente para aproximarnos a MVVM siguiendo las recomendaciones de Google.



Donde ...

- **View** son los **Componentes UI** en nuestro caso los componentes de **Jetpack Compose** que renderizan los datos en la pantalla.
- **ViewModel** es el **Contenedor de estados** que contiene datos, los exponen a la IU y controlan la lógica.
- **Model** es una abstracción de los datos de la aplicación. En nuestro caso serán los **Repositorios** que se encargan de obtener los datos de una fuente de datos y convertirlos en un formato con el que el resto de la aplicación pueda trabajar.

Visión general de la arquitectura en un proyecto de Android

A la hora de concretar la arquitectura de una aplicación de Android, aunque hay una serie de recomendaciones por parte de Google, no hay una forma única de hacerlo. Por tanto, el equipo de desarrollo de una aplicación debe decidir algún tipo de **convenio organizativo** de paquetes, nombres de clases, etc. que se adapte a sus necesidades. De tal manera que cualquier miembro del equipo sepa encontrar fácilmente donde se encuentra la implementación de una determinada funcionalidad. Además de modificar fácilmente con el menor número de fallos y conflictos de trabajo en paralelo.

Esto último, se consigue creando diferentes módulos de la aplicación que se puedan desarrollar de forma independiente y que se puedan integrar fácilmente en el proyecto final. Sin embargo, la parte de modularización en diferentes paquetes no la vamos a abordar en este curso por falta de tiempo.

Descarga del siguiente enlace tienes un **'Cheat Sheet' de la Arquitectura** propuesta en el curso que debes descargar y tener accesible durante el presente curso.

> 📌 **Nota**: Vamos a empezar con una **estructuración inicial del proyecto** siguiendo el esquema propuesto en el cheatsheet anterior para ir aproximándonos a la arquitectura propuesta por Google y a medida que vayamos avanzando en el curso iremos concretando más sobre su implementación, las diferentes sub-capas de la arquitectura, los convenios de nombres que hemos usado en el curso, etc.

Estructurando el Modelo

Aquí vamos a crear los modelos de datos que vamos a utilizar en nuestra aplicación. En nuestro caso, vamos a crear un modelo de datos para representar un **Contacto** de una Agenda.

Para ello, siguiendo es esquema de carpetas propuesto en el cheatsheet anterior, vamos a crear un paquete `models` dentro del paquete **principal** de nuestro proyecto.

Este modelos serán clases '*generales*' o '*genéricas*' que representarán los datos de mi problema. Podremos definir roles entre ellas, así como relaciones de herencia, composición, etc. En algunos casos complejos incluso podremos definir casos de uso, etc.

Nosotros simplemente vamos a definir una clase `Contacto` que representará un contacto de una agenda. Para ello, vamos a crear un fichero `Contacto.kt` dentro del paquete `models` con el siguiente contenido.

```
data class Contacto(
    val id: Int,
    val nombre: String,
    val apellidos: String,
    val foto: String?,
    val correo: String,
    val telefono: String,
    val categorias: EnumSet<Categorias>
) {
    enum class Categorias {
        Amigos, Trabajo, Familia, Emergencias
    }
}
```

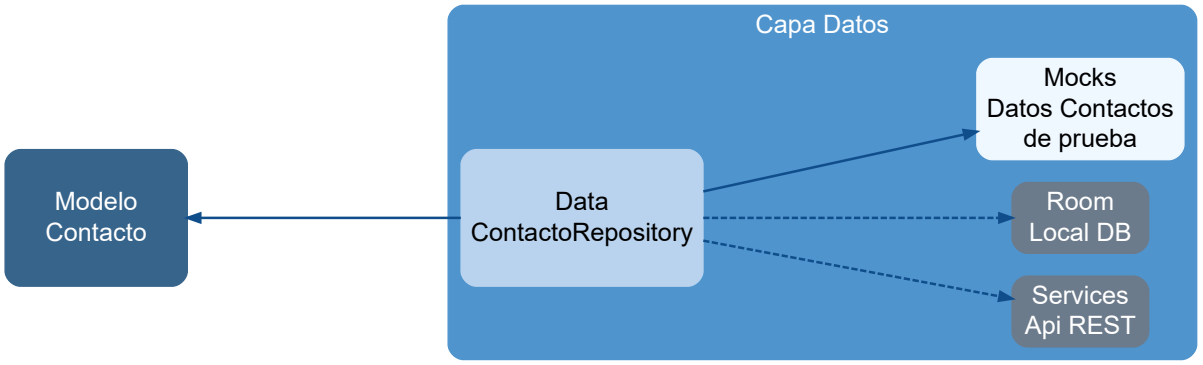
> ✋ **Importante**: El modelo no debería tener ninguna referencia o **dependencia** de la **capa de UI** ni a la **capa de datos**.

Para evitar que el modelo tenga **dependencias** por ejemplo de las bibliotecas del propio android. Fíjate que la propiedad `foto` es de tipo `String?` y no de tipo `Bitmap?` que nos obligaría a incluir el paquete `android.graphics` en el modelo. De esta manera la imagen puede ser una **url**, un **path**, un **base64**, etc. y la conversión a `Bitmap` la haremos en la capa de UI.

En nuestro caso hemos decidido que sea una cadena en **base64**. Si quieres saber más sobre que es `base64` puedes consultar el **Apéndice I** al final de este documento o bien a este enlace a la Wikipedia.

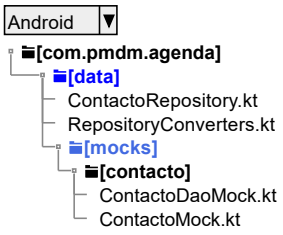
Estructurando la capa de datos

Ahondado **un poco más** el diagrama general de la capa de datos podríamos tener un esquema similar al siguiente:



Vamos a concretar una propuesta de organización en paquetes y nomenclatura para el mismo que vamos a seguir durante el curso...

1. Crearemos **paquete** `data` dentro del paquete **principal** de nuestro proyecto. En él, vamos a crear los **repositorios** que se encargarán de obtener los datos de una fuente de datos y convertirlos en un formato con el que el resto de la aplicación pueda trabajar. Además, contendrá los paquetes que van a gestionar las diferentes fuentes de datos de nuestra aplicación y esto será lo primero que vamos hacer porque sin fuente no puede haber repositorio que la encapsule.



2. Crearemos **paquete** `mocks` que definirá **colecciones de objetos de prueba** que simulan una fuente de datos. Como puede haber diferentes clases, vamos a crear un paquete `mocks.contacto` el cual definirá las clases que contienen los contactos de prueba.
   - **ContactoMock.kt** es una clase que define un objeto de prueba de tipo `Contacto`. En este caso es idéntica a la clase `Contacto` que hemos definido en el modelo. Sin embargo, en un caso real, podría ser diferente ya que el modelo es independiente de la fuente de datos.

```
data class ContactoMock(
    val id: Int,
    val nombre: String,
    val apellidos: String,
    val foto: String?,
    val correo: String,
    val telefono: String,
    val categorias: EnumSet<Categorias>
) {
    enum class Categorias {
        Amigos, Trabajo, Familia, Emergencias
    }
}
```

- **ContactoDaoMock.kt** es una clase que define una colección de objetos de prueba de tipo `ContactoMock`. En este caso, es una clase que implementa las operaciones CRUD (Create, Read, Update, Delete) sobre la colección de objetos de prueba como los siguientes...

```
class ContactoDaoMock {
    // Colección de datos de prueba
    private var contactos = mutableListOf(...)

    // Obtiene una lista de contactos
    fun get(): MutableList<ContactoMock> = contactos
    // Obtiene un contacto por Id
    fun get(id: Int): ContactoMock? = contactos.find { u -> u.id == id }
    // Inserta un contacto en la colección de contactos
    fun insert(contacto: ContactoMock) = contactos.add(contacto)
}
```

3. Dentro del paquete `data`, crearemos el fichero **RepositoryConverters.kt** que contendrá las **funciones de extensión** para transformar el **modelo** y la **fuente de datos** y viceversa. Al usar funciones de extensión, podemos agregar métodos a una clase sin tener que modificar la definición de la clase manteniendo el principio OCP (Open-Closed Principle) de SOLID. Además, evitamos que el modelo tenga **dependencias** de la fuente de datos.

```
// Convierte un un objeto Contacto del modelo en un ContactoMock
fun Contacto.toContactoMock() = ContactoMock(...)

// Convierte un un objeto ContactoMock de la fuente de datos en un Contacto
fun ContactoMock.toContacto() = Contacto(...)
```

4. entro del paquete `data`, crearemos el fichero **ContactoRepository.kt** que contendrá la clase `ContactoRepository` que implementa las operaciones CRUD (Create, Read, Update, Delete) sobre **objetos del modelo** `Contacto` y se encargará de replicarlos en la fuente de datos en nuestro caso el '***mock***' de forma transparente.

```
class ContactoRepository {
    private var dao = ContactoDaoMock()

    fun get(): MutableList<Contacto> = dao.get().map { it.toContacto() }.toMutableList()
    fun get(id: Int): Contacto? = dao.get(id)?.toContacto()
    fun insert(contacto: Contacto) = dao.get().add(contacto.toContactoMock())
}
```

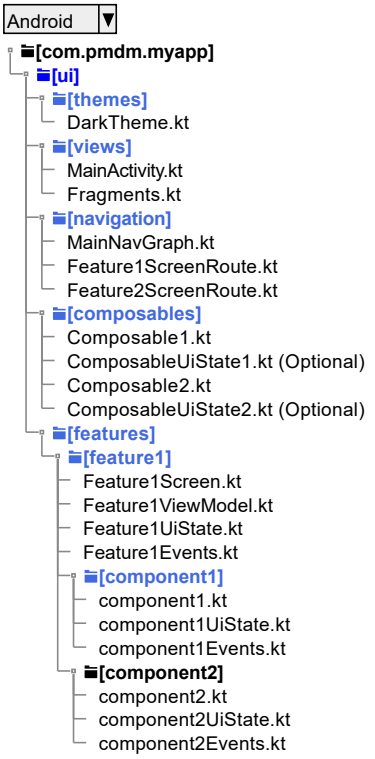
> 📣 **Resumen**: Podemos decir que es una envoltorio o '*Wapper*' que **encapsula la fuente de datos** y sus datos de forma que el resto de la aplicación pueda trabajar con ellos como un modelo genérico sin saber cual es la fuente de los mismos.
>
> Aquí tienes un **vídeo** que **te pueden ayudar a tener una idea de su funcionamiento del patrón repository** en en otros lenguajes de programación. No hace falta entender 100% el código, pero si la idea general del patrón.

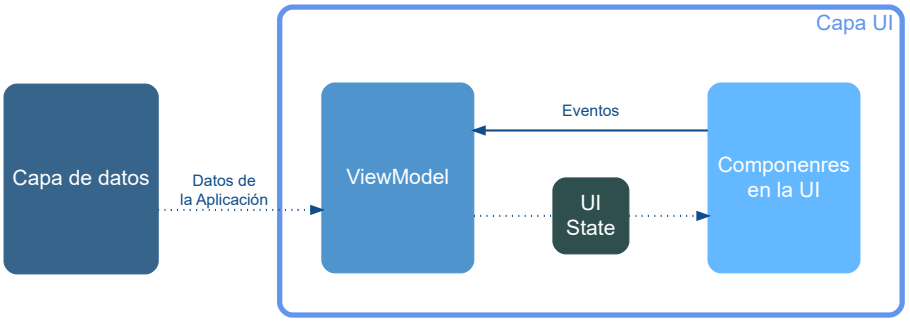
Estructurando la capa de UI

Esta va a ser la capa más compleja organizativamente. Pues vamos atener diferentes paquetes para diferenciar claramente las diferentes partes de la capa de UI.

En el **diagrama de la derecha**, tienes nuestra propuesta de plantilla organizativa para esta capa que hay en el '*cheatsheet*'. Aunque en los futuros temas vamos a abordar con detalle los ficheros que vamos a crear en cada uno de los paquetes, vamos a hacer una **primera aproximación** a la estructura de la capa de UI para tener **una visión global de la misma**.

Esta estructura respondería a la **propuesta de Google** donde existe un **contenedor de estados** que contiene los datos, los expone a la IU y controla la lógica. En nuestro caso, este contenedor de estados será el **ViewModel**.





Vamos a explicar brevemente los tres elementos de la Capa de UI y cada uno de los sub-paquetes que hemos creado en dentro del paquete `UI` que la representa.

> 📌 **Nota**: Iremos profundizando en su implementación en los siguientes temas.

Elementos que la forman:

1. `ViewModel` Como hemos comentado es el **Contenedor de estados** que contiene datos, los expone a la IU y controla la lógica. Podemos tener un **ViewModel** por cada **pantalla** de nuestra aplicación o compartirlo entre varias. Además, en nuestra propuesta de arquitectura se encargará gestionar:
   - La **gestión de los datos** de los elementos visuales a través de objetos `UiState` que representan el estado de visualización de los componentes en la UI.
   - La **lógica** de la pantalla entre los que vamos a incluir la gestión de **casos de uso** de los objetos del modelo.
   - Derivada de gestión de la lógica tendremos la **gestión de los eventos** de los componentes en la UI.
   - Gestión de la visualización de estados de **error**.
   - Transformación de los objetos del **UiState** a los objetos de **Modelo** y viceversa.
   - Comunicación con la **capa de datos** a través de las '*clases repositorio*'.
2. `UiState` estará representado por una o varias clases que representarán el **estado de visualización** de los elementos visuales de la pantalla. Podríamos tener:
   - Un `UiState` por cada **elemento visual** de la pantalla. Por ejemplo, un `UiState` de tipo booleano para representar el estado de un **icono** o si se visualiza o no un **circulo de progreso**, etc.
   - Un `UiState` por cada **grupo de elementos visuales relacionados entre si** en la pantalla. Por ejemplo, una `data class NombreCompletoUiState` con `nombre`, `apellido1` y `apellido2` a modo de `UiState` para representar el estado de **tres campos de texto** que representen el nombre completo de una persona.
   - Un `UiState` por cada **pantalla** de nuestra aplicación que a su vez este formado por UiStates de grupo de elementos visuales que se repitan entre pantallas.

3. `Componentes en la UI` son los elementos visuales que renderizan los datos en la pantalla. En nuestro caso, serán los componentes de **Jetpack Compose** que renderizan los datos en la pantalla. Sin embargo, podría sustituir de forma sencilla por componentes tradicionales de **XML** o *'Vistas'*.

Sub-paquetes de la capa de UI:

1. `ui.themes` es un paquete que contendrá los diferentes temas de la aplicación. Que contendrán los colores, tipografías, etc. que vamos a usar en nuestra aplicación.
2. `ui.views` es un paquete que contendrá las diferentes vistas de la aplicación. En nuestro caso, tendremos una única vista que será la `MainActivity` que será la actividad principal de nuestra aplicación. Solo si tuviéramos, puntos de entrada secundarios en nuestra aplicación definiríamos más '*activities*'.
3. `ui.navigation` es un paquete que contendrá el grafo de navegación de nuestra aplicación. Con las diferentes rutas de navegación entre las pantallas.
4. `ui.composables` aquí definiremos aquellos componentes personalizados de la aplicación que sean **reutilizables** entre diferentes pantallas y las clases que representen su **UiState** si fuera necesario.
5. `ui.features` es un paquete que contendrá las diferentes pantallas de nuestra aplicación. En nuestro ejemplo de plantilla, tendremos la pantalla `Feature1Screen` con un **ViewModel** para gestionarla definido en la clase `Feature1ViewModel` y su correspondiente **UiState** definido en la clase `Feature1UiState`. Además, si esta pantalla esta compuesta por diferentes grupos lógicos de elementos visuales, podríamos crear un paquete `component1` y `component2` con sus correspondientes **ViewModel** y **UiState** para gestionarlos. De esta manera si tuviéramos que reutilizarlos en otra pantalla, podríamos hacerlo fácilmente conservando toda la lógica.

Apéndice I - Codificando imágenes en Base64

En ocasiones, necesitamos codificar una imagen en Base64 para poder enviarla a un servidor o almacenarla en una base de datos. Para ello, podemos usar la siguiente función que nos permite codificar una imagen en Base64.

La **codificación base64** consiste en realizar **agrupaciones de 6 bits** y asignarle un carácter:

- Letras mayúsculas (**0-25**): ABCDEFGHIJKLMNOPQRSTUVWXYZ
- Letras minúsculas (**26-51**): abcdefghijklmnopqrstuvwxyz
- Dígitos decimales (**52-61**): 0123456789
- Símbolos especiales (**62-63**): +/

Si la cadena de bytes no contiene exactamente un múltiplo de 6 bits, se **completará con ceros**.

> **Ejemplo**: Supongamos la cadena ...
>
> | Cadena | Binario | Hexadecimal |
> |---|---|---|
> | aØ | 0110 0001 1100 0011 1001 1000 | 61 C3 98 |
>
> Si queremos codificar esta cadena de bits en Base64 tendremos que realizar **grupos de 6 bits**:
>
> | Binario | Grupos de 6 bits | Base 64 |
> |---|---|---|
> | 0110 0001 1100 0011 1001 1000 | 011000 011100 001110 01100 | YcOY |

Aquí tienes diferentes URL's de utilidad que te permiten codificar y decodificar textos e imágenes en Base64:

- https://codebeautify.org/text-to-base64-converter
- https://codebeautify.org/base64-to-text-converter
- https://codebeautify.org/image-to-base64-converter
- https://codebeautify.org/base64-to-image-converter

Funciones de utilidad para codificar y decodificar imágenes

En el siguiente código tienes una serie de funciones de utilidad para codificar y decodificar imágenes en Base64. Puedes descargar el código de las funciones de utilidad desde el siguiente enlace **Imagenes.kt**.

En nuestra propuesta organizativa de proyecto, estas funciones de utilidad las vamos a incluir en el paquete `utilities.imagenes` dentro del paquete **principal** de nuestro proyecto y a su vez. Iremos pues, creando dentro de este paquete diferentes paquetes para agrupar las diferentes funciones de utilidad que vayamos creando.

```kotlin
fun blobToBase64(byteArray: ByteArray?): String? =
    byteArray?.let { Base64.encodeToString(it, Base64.DEFAULT) }

fun base64ToBlob(base64ImageString: String?): ByteArray? =
    base64ImageString?.let { Base64.decode(it, Base64.DEFAULT) }

fun base64ToBitmap(base64ImageString: String): Bitmap {
    val decodedString = base64ImageString?.let { Base64.decode(it, Base64.DEFAULT) }
    return BitmapFactory.decodeByteArray(decodedString, 0, decodedString!!.size)
}

fun bitmapToBase64(bitmap: Bitmap): String {
    val stream = ByteArrayOutputStream()
    bitmap.compress(Bitmap.CompressFormat.PNG, 90, stream)
    val byteArray: ByteArray = stream.toByteArray()
    return Base64.encodeToString(byteArray, Base64.DEFAULT)
}
```