Documentación técnica “Cabify Mobile Challenge”

Inicio del proyecto

Challenge: challenge
Repositorio: android-cabify-challenge
Tecnología: Android
Versión de Android Studio: Android Studio Koala 2024.1.1
Lenguaje: Kotlin
Arquitectura: MVVM

- Además de ser recomendado por google, MVVM mantiene más claridad en el código, es más fácil de testear y más fácil de mantener.

Módulos creados: Como este challenge

- app
  - Módulo principal donde está la application de la app
- corenetwork
  - Módulo encargado de guardar los servicios y crear todo lo relacionado a Retrofit y llamadas a endpoints
  - Contiene también el mapping de las response que se obtengan de las mismas
- data
  - Se guardan los DTOs que se obtienen del JSON, los mapper que lo convierten a modelos, datasource de la pegada a backend y la implementación del CartRepository
- domain
  - Se guardan los modelos tanto de los productos como del carrito. Interfaz del CartRepository y caso de uso para la pegada a backend
- ui
  - Módulo donde están todas las vistas y activities. Se crea la custom view del producto y el summary
  - También se implementó componentes bases para botones, textos y tarjetas para usar siempre el mismo en toda la app, el cual tienen también sus propios estilos

Librerías agregadas:

- compose
- retrofit
- gson
- hilt
- okhttp
- coil
- mockk
- lottie
- Retrofit RxJava

Se eligieron estas librerías para facilitar el trabajo de UI, llamadas a la API, inyección de dependencias y tests. Para la UI, Compose es la librería más moderna y recomendada por Google, pero también facilita el crear componentes reutilizables, es fácil de leerlo y comprenderlo, y se integra bien con corrutinas y observables. Aunque las Previews son un poco más tediosas que las preview normal de XML. Las librerías de coil y lottie son más que nada para que se vea un poco más linda la aplicación, no pesan mucho.

Para la llamada a la API, Retrofit es la más usada en las aplicaciones, se integra bien con la librería OkHttp para usar interceptores, donde en este challenge se usó para poner un timeout, y Gson, para usar conversores y manejar los errores. También se puede extender para usar otro tipo de conversores y corrutinas, solo que para este challenge no fueron necesarios.

Para los tests se usó mockk, y creo que fue por gusto personal. Es más fácil leer y escribir el uso de every y verify.

Al principio se había agregado la librería de *RxJava* para usarlo con Retrofit pero se sacó más que nada por una sobrecarga de dependencias y complejidad en el código. Al ser una sola pegada a backend y algo simple como obtener los productos, después iba a ser más complejo el código, debuggear y testearlo.

Corenetwork

El módulo corenetwork está encargado de crear toda la instancia de Retrofit y mapear la respuesta del endpoint. Al retrofit se le agregó lo básico para que tenga el url base del challenge, un timeout de 1 minuto y use el convertidor de Gson.

Los servicios devuelven respuestas de tipo *Response*, clase propia de Retrofit, ya que son más fáciles para acceder a los headers, a los códigos de errores y los error body. El mapper de las respuestas de los endpoints recibe este tipo de clase para poder devolver un modelo, en base a un DTO. La idea del *mapResponse* es que sea genérico para cualquier tipo de respuesta que tenga la aplicación, por eso uno le especifica el DTO, el modelo y el mapper que convierte uno a otro. En caso de que el endpoint devuelva un 200 y que el body de la respuesta no sea nulo, devolvemos el modelo.

También se mide el caso en que haya un problema con la respuesta, o un problema al intentar mapear, devolviendo un error para que la activity pueda mostrar un mensaje y no pare la aplicación.

ApiResult<> es la clase sellada la cual te facilita acceder a si falló o no la llamada, y devolverte tanto el mensaje de error o el modelo a usar.

Al principio, cuando se terminó el mapResponse, se estaba usando ResponseBody, clase de okhttp3, para generalizar todas las respuestas (Response<ResponseBody>) y usar Gson para deserializar a un tipo T, pero se terminó dejando la respuestas especificando el tipo de clase mediante un DTO.

¿Qué se podría agregar?

Como dice arriba, Response puede devolverte el código o body de la respuesta de error. Esto, con la aplicación en crecimiento o en una aplicación más grande, podríamos devolverle a la activity el tipo de error que se generó al hacer la llamada al endpoint, para poder así mostrar distintos mensajes, o ejecutar ciertas acciones dependiendo de lo que haya pasado.

Data

En el módulo data se creó la implementación del datasource, el cual pide el servicio y obtiene los productos y devuelve un ApiResult gracias a la función de mapResponse.

Tiene también los mappers que obtienen el DTO obtenido del servicio y los convierte a un modelo, usados para toda la lógica del agregado, remover productos y para el carrito.

Por último tiene la implementación del repositorio del carrito, el cual se encarga de agregar y remover, crear un carrito si no existe y devolver el precio total. Tiene también un proveedor del servicio, ya con todo lo que necesita de Retrofit, creado por ServiceBuilder

Domain

En el módulo de domain se crearon los modelos de negocio usados para las vistas y repositorios. Tiene los usecase para obtener los productos, el cual usa el viewmodel de la activity, y el usecase para calcular todos los descuentos, el cual usa el repositorio del carrito.

UI

En el módulo UI se crearon todo lo relacionado a las vistas, tanto custom views para los productos y summary, como también las activities.

Componentes base

Para que en toda la aplicación no se use el mismo componente de Compose y se modifique todo el tiempo, se crearon componentes base, el cual cada uno tiene su propio estilo:

- CabifyText
- CabifyButton
- CabifyCard

Esto está pensado muy a futuro y está hecho de manera simple. La idea es que la aplicación pueda manejar todos los estilos que pueda tener, tanto tamaño de textos, fuente, colores, etc.

Factory

Cada componente (en este caso, solo existe el de producto) tiene su propio factory, el cual crea la vista y maneja lógica. La idea de manejar lógica es más a futuro, en el caso donde una vista necesite ser modificada, por el motivo que sea, porque se hizo una acción ya sea desde backend o por accionar algún otro componente. El factory recibiría eventos para así modificar su propia vista con estados mutables.

La manera de mandar eventos para que se entienda, es que el viewmodel tiene una lista de Factories que va a dibujar la vista. Si necesitas modificar una vista por el motivo que sea, el viewmodel recorreria la lista de facotires enviandoles un evento a todos. El factory que en el when esté esperando X evento (El evento puede ser una clase sellada, mas facil usarlo) ejecuta la acción que necesite