Prueba técnica OpenBank

Partes importantes del enunciado

*Este repositorio contiene una serie de requerimientos de un Caso Práctico, que busca evaluar las capacidades técnicas del candidato con respecto a las principales funciones y responsabilidades que se requieren dentro del área de Arquitectura de Openbank.*

***¿Qué se busca evaluar?***

*Principalmente los siguientes aspectos:*⁃ *Definición de una arquitectura para la app desde cero.*⁃ *Creatividad para resolver los requerimientos.*⁃ *Calidad del código entregado (estructura y buenas prácticas).*⁃ *Eficiencia de la solución entregada.*⁃ *Familiaridad con librerías, frameworks y plataformas de desarrollo.*

***Descripción***

*Se solicita la creación de la estructura de una app que muestre un listado de los personajes Marvel y permite ver el detalle de cada uno de ellos de manera individual.*

***Funcionalidad***

⁃ *Listas el listado de personajes.*⁃ *Navegar al detalle de un personaje concreto.*

***Requisitos***

⁃ *Requerido usar Xcode/Android Studio.*⁃ *Requerido utilizar Swift/Kotlin.*⁃ *Se puede hacer uso de frameworks y librerías de terceros.*⁃ *No usar SwiftUI en iOS para la implementación de la capa de vista.*⁃ *Se valorará la implementación de tests.*⁃ *Se valorará un correcto control de errores.*⁃ *Total libertad para añadir cualquier funcionalidad extra que se considere.*

¿Qué se busca evaluar?

Arquitectura

He planteado una arquitectura desde cero utilizando el patrón de diseño VIPER.

Las arquitecturas tradicionales, como MVC (Modelo-Vista-Controlador) utilizada originariamente en el desarrollo de aplicaciones iOS, tiene como principal problema el hecho de que toda la lógica de negocio recae sobre el controlador, por lo que su tamaño crece desmesuradamente y, en consecuencia, el mantenimiento y la escalabilidad del proyecto se ven fuertemente comprometidas.

Con el fin de solventar este problema y de cumplir con los principios básicos de la *Clean Architecture*, que, en síntesis, aboga por la separación de responsabilidades y el desacoplamiento entre las distintas partes de código, se desarrollo el patrón VIPER. Éste, a su vez trata de cumplir con el principio de responsabilidad única (*Single Responsibility Principle* o SRP). Para ello, VIPER trata de dividir la lógica de la aplicación en diferentes capas o responsabilidades. Para ello, VIPER se divide en cinco capas:

* **View**: muestra la información recibida desde el **Presenter** y envía las interacciones del usuario de vuelta al **Presenter**.
* **Interactor**: recupera los datos de las **Entities** y contiene la lógica de negocio de un caso de uso particular.
* **Presenter**: se encarga de preparar los datos para ser enviados a la **View**, e intercepta las interacciones del usuario recibidas desde la **View**.
* **Entity**: representa el modelo de datos de los objetos.
* **Router**: se encarga de la lógica de navegación, gestionando qué pantallas han de mostrarse y cuando.

Al implementar VIPER, cada módulo, que corresponde con una pantalla, seguirá esta estructura. Debido a que la lógica de la app se separa en componentes más pequeños, las vistas ahora son más pequeñas y manejables, y la lógica es más sencilla de testar.

El flujo de VIPER utilizado en la app, se corresponde con el siguiente diagrama:

Imagen de la pantalla de un celular con letras

Descripción generada automáticamente con confianza media

Este diagrama muestra cómo los diferentes componentes de VIPER están relacionados entre ellos. Los componentes están unidos a través de interfaces, cada una de las cuales tienen un protocolo que define su comportamiento. Es importante establecer referencias fuertes (*strong*) y débiles (*weak*) en los lugares apropiados, con el fin de evitar referencias circulares de memoria o *retain cycles*, que puedan afectar negativamente al rendimiento de la app.

También es importante destacar que únicamente la *View* y el *Router* deberían importar *UIKit*. De esta manera, nos aseguramos de que el resto de los componentes no tengan dependencias con la interfaz gráfica.

Por lo tanto, en la app, existen dos módulos con los diferentes componentes de VIPER, uno para la pantalla de la lista de personajes y otro para la pantalla de detalle de un personaje:

Imagen que contiene Tabla

Descripción generada automáticamente

Clases Base

Para definir las partes comunes de los diferentes compontes de VIPER, se han definido una serie de clases base para cada componente. Cada una de ellas es realmente una interfaz que establece unos protocolos con los que han de conformar cada una de las clases que hereden de ellas. Debido a la simpleza de la app, únicamente las clases bases para las *Views* y los *Routers* tienen elementos dentro de sus protocolos.

Texto

Descripción generada automáticamente con confianza media

También se ha definido un *View Controller Base*, que define las funcionalidades para gestionar el *spinner* de carga que se muestra mientras se está obteniendo la información, y gestionar la disponibilidad de conexión para poder hacer las llamadas a los servicios REST.

Domain Model y View Model

La capa *Entity* se ha dividido en dos partes. Por un lado, se han definido los modelos de dominio, para cargar en ellos la información proveniente de los servicios. Y, por otro lado, se han definido los modelos de vista, que son los que se utilizan para mostrar la información en las vistas. Asimismo, se han definido los correspondientes *parsers* para traducir los modelos de dominio en los modelos de vista.

Capa de servicios

Adicionalmente a las capas de VIPER, se ha definido una capa adicional para el acceso a los servicios que, básicamente, se encarga de configurar y realizar las llamadas a los servicios REST, y traducir el resultado de esas llamadas a los modelos de dominio.

Elementos comunes

Se han definido una serie de elementos comunes, que pueden ser utilizados desde los diferentes módulos, y que albergan constantes, enumeraciones, extensiones y vistas comunes (vista para el *spinner* de carga, vista para mostrar errores y vista para mostrar que no hay conexión a internet).

Requisitos

Los requisitos que he utilizado son:

* **Xcode**
* **Swift**
* Con respecto a *frameworks* y librerías de terceros:
  + **SwiftLint** para asegurar un formateado de código correcto.
  + **ReachabilitySwift** para comprobación de la conexión a Internet.
* He implementado **test unitarios** y **test de integración**.
* He implementado una **gestión de errores**.