Guía de arquitectura de apps

En esta guía, se incluyen las prácticas y la arquitectura recomendadas (#recommended-app-arch) para desarrollar apps sólidas y de calidad.

Nota: En esta página, se asume que tienes conocimientos de los aspectos básicos del framework de Android. Si no tienes experiencia en el desarrollo de apps para Android, consulta el Curso de aspectos básicos de Android (https://developer.android.com/courses/android-basics-kotlin/course?hl=es-419) para introducirte en el tema y obtener más información sobre los conceptos que se mencionan en esta guía.

Experiencias del usuario de apps para dispositivos móvile

Una app de Android típica consta de varios componentes de la app (https://developer.android.com/guide/components/fundamentals?hl=es-419#components), como actividades (https://developer.android.com/guide/components/activities/intro-activities?hl=es-419), fragmentos (https://developer.android.com/guide/fragments?hl=es-419), servicios (https://developer.android.com/guide/components/services?hl=es-419), proveedores de contenido (https://developer.android.com/guide/topics/providers/content-providers?hl=es-419) y receptores de emisión (https://developer.android.com/guide/components/broadcasts?hl=es-419). A la mayoría de estos componentes los declaras en el manifiesto de la app (https://developer.android.com/guide/topics/manifest/manifest-intro?hl=es-419). Luego, el SO Android usa ese archivo para decidir la integración de tu app a la experiencia del usuario general del dispositivo. Dado que una app para Android típica puede contener varios componentes y que los usuarios suelen interactuar con diferentes apps en poco tiempo, las apps deben adaptarse a distintos tipos de tareas y flujos de trabajo controlados por los usuarios.

Ten en cuenta que los dispositivos móviles tienen restricciones de recursos, de manera que, en cualquier momento, el sistema operativo podría cerrar algunos procesos de app a fin de hacer lugar para otros.

Según las condiciones de este entorno, es posible que los componentes de tu app se inicien de manera individual y desordenada, además de que el usuario o el sistema operativo podrían destruirlos en cualquier momento Debido a que no puedes controlar

estos eventos, no debes almacenar ni mantener en la memoria ningún estado ni datos de la aplicación en los componentes de tu app, y estos elementos no deben ser interdependientes.

Principios comunes de arquitectura

Si se supone que no deberías usar los componentes de la aplicación para almacenar datos y estados, ¿cómo deberías diseñarla?

A medida que crece el tamaño de las apps para Android, es importante definir una arquitectura que permita que esta crezca, aumente su solidez y facilite su prueba.

La arquitectura de una app define los límites entre sus partes y las responsabilidades que debe tener cada una. A fin de satisfacer las necesidades que se mencionaron antes, debes diseñar la arquitectura de tu app para que cumpla con algunos principios específicos.

Separación de problemas

El principio más importante que debes seguir es el de separación de problemas (https://en.wikipedia.org/wiki/Separation_of_concerns). Un error común es escribir todo tu código en una `Activity` (https://developer.android.com/reference/android/app/Activity?hl=es-419) o un `Fragment` (https://developer.android.com/reference/android/app/Fragment?hl=es-419). Estas clases basadas en IU solo deberían contener lógica que se ocupe de interacciones del sistema operativo y de IU. Si mantienes estas clases tan limpias como sea posible, puedes evitar muchos problemas relacionados con el ciclo de vida de los componentes y mejorar la capacidad de prueba de estas clases.

Ten en cuenta que las implementaciones de `Activity` y `Fragment` no son de tu propiedad, sino que estas solo son clases que representan el contrato entre el SO Android y tu app. El SO puede destruirlas en cualquier momento en función de las interacciones de usuarios y otras condiciones del sistema, como memoria insuficiente. Para brindar una experiencia del usuario satisfactoria y una experiencia de mantenimiento de apps más fácil de administrar, recomendamos reducir la dependencia de esas apps.

Cómo controlar la IU a partir de modelos de datos

Otro principio importante es que debes controlar la IU a partir de modelos de datos, preferentemente que sean de persistencia. Los modelos de datos representan los datos de una app. Son independientes de los elementos de la IU y otros componentes de la app. Por

lo tanto, no están vinculados a la IU ni al ciclo de vida de esos componentes, pero se destruirán cuando el SO decida quitar el proceso de la app de la memoria.

Los modelos de persistencia son ideales por los siguientes motivos:

- Tus usuarios no perderán datos si el SO Android destruye tu app para liberar recursos.
- Tu app continúa funcionando cuando una conexión de red es débil o no está disponible.

Si basas la arquitectura de tu app en clases de modelos de datos, mejorarás la capacidad de prueba y la solidez de tu app.

Arquitectura de app recomendada

En esta sección, se muestra cómo estructurar la app según las prácticas recomendadas.

Nota: Las sugerencias y las prácticas recomendadas de la página se pueden aplicar a un amplio espectro de apps para hacer ajustes, mejorar la calidad y la solidez, y facilitar las pruebas. Sin embargo, debes tratarlas como lineamientos y adaptarlas a tus requisitos según sea necesario.

Teniendo en cuenta los principios de arquitectura comunes que se mencionaron en la sección anterior, cada aplicación debe tener al menos dos capas:

- La *capa de la IU* que muestra los datos de la aplicación en la pantalla.
- La *capa de datos* que contiene la lógica empresarial de tu aplicación y expone sus datos.

Puedes agregar una capa adicional llamada *capa de dominio* para simplificar y volver a utilizar las interacciones entre la IU y las capas de datos.

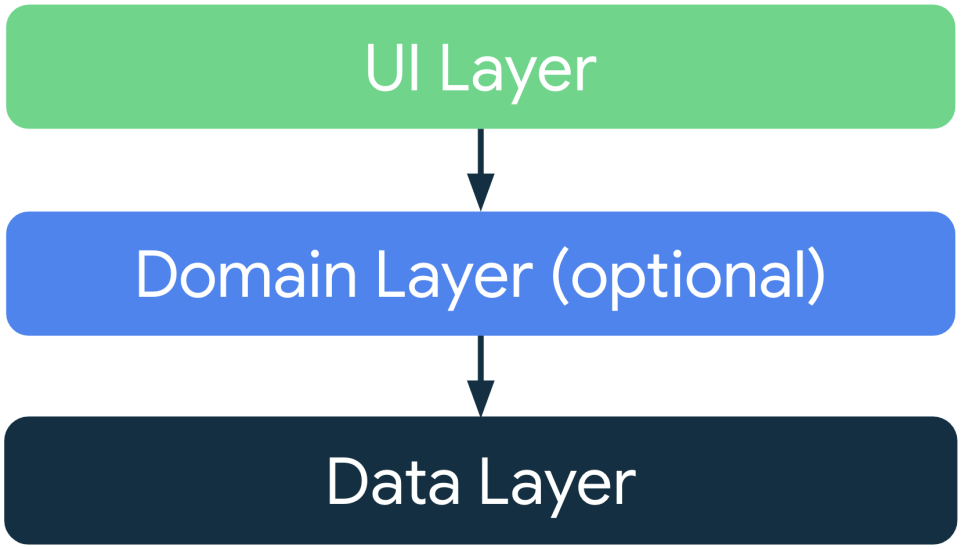

Figura 1: Diagrama de una arquitectura de app típica

Nota: Las flechas de los diagramas de esta guía representan dependencias entre clases. Por ejemplo, la capa de dominio depende de las clases de capa de datos.

Capa de la IU

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 o las funciones de Jetpack Compose (https://developer.android.com/jetpack/compose?hl=es-419))
- Contenedores de estados (como las clases ViewModel (https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419)) que contienen datos, los exponen a la IU y controlan la lógica

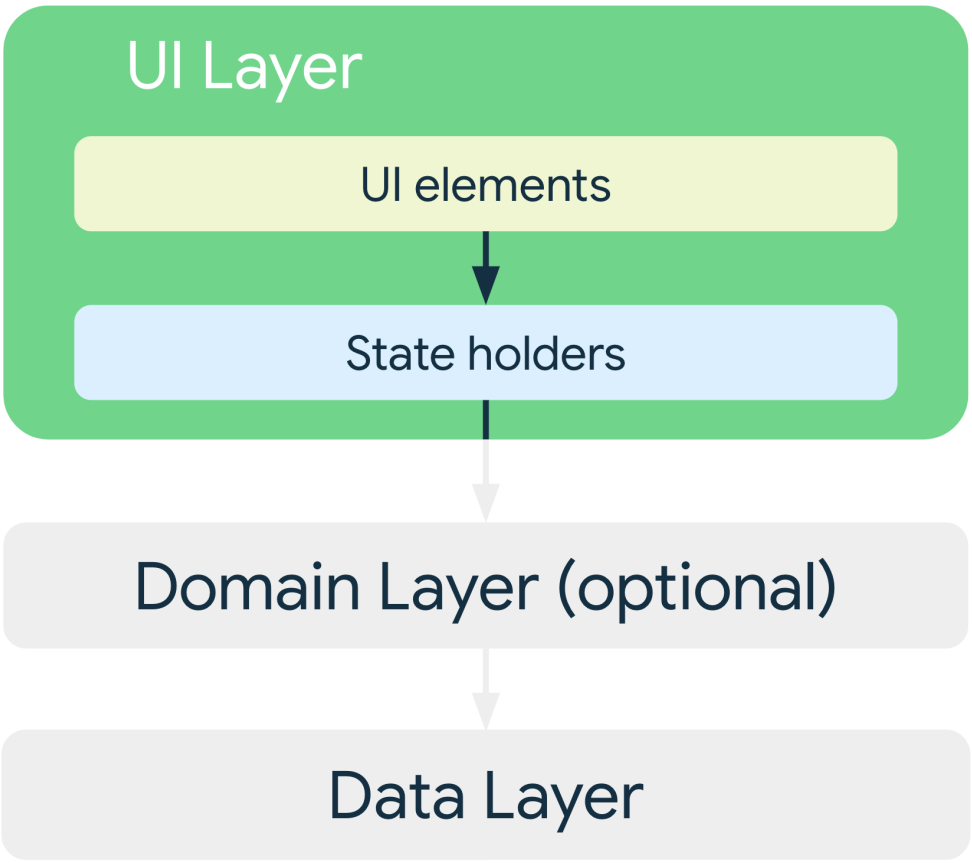

Figura 2: La función de la capa de la IU en la arquitectura de la app

Para obtener más información sobre esta capa, consulta la página sobre la capa de la IU (https://developer.android.com/jetpack/guide/ui-layer?hl=es-419).

Capa de datos

La capa de datos de una app contiene la *lógica empresarial*. Esta lógica es lo que le da valor a tu app. Además, está compuesta por reglas que determinan cómo tu app crea, almacena y cambia datos.

La capa de datos está formada por *repositorios* que pueden contener de cero a muchas *fuentes de datos*. Debes crear una clase de repositorio para cada tipo de datos diferente que administres en tu app. Por ejemplo, puedes crear una clase `MoviesRepository` para datos relacionados con películas o una clase `PaymentsRepository` para datos relacionados con pagos.

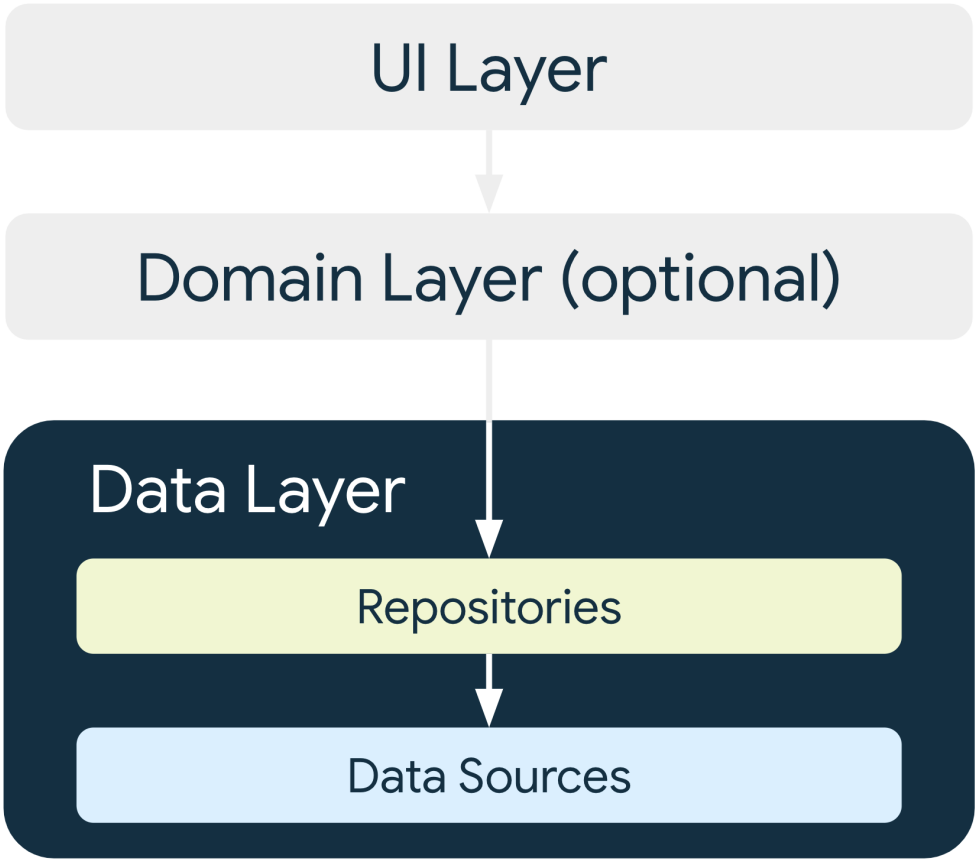

Figura 3: La función de la capa de datos en la arquitectura de la app

Las clases de repositorio son responsables de las siguientes tareas:

- Exponer datos al resto de la app
- Centralizar los cambios en los datos
- Resolver conflictos entre múltiples fuentes de datos
- Abstraer fuentes de datos del resto de la app
- Contener la lógica empresarial

Cada clase de fuente de datos debe tener la responsabilidad de trabajar con una sola fuente de datos, que puede ser un archivo, una fuente de red o una base de datos local. Las clases de fuente de datos son el puente entre la aplicación y el sistema para las operaciones de datos.

Para obtener más información sobre esta capa, consulta la página sobre la capa de datos (https://developer.android.com/jetpack/guide/data-layer?hl=es-419).

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.

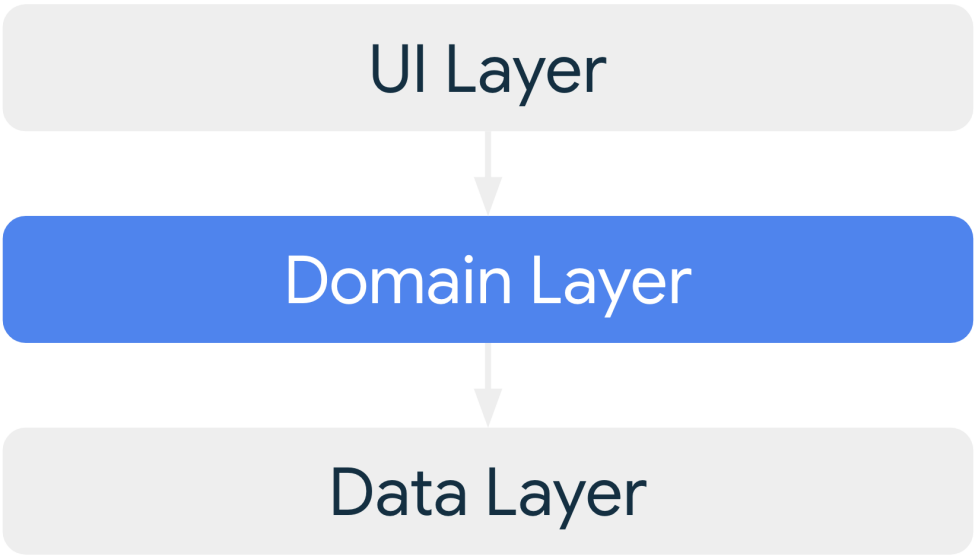

Figura 4: La función de la capa de dominio en la arquitectura de la app

Las clases de esta capa se denominan *casos de uso* o *interactores*. Cada caso de uso debe tener responsabilidad sobre una funcionalidad *única*. Por ejemplo, tu app podría tener una clase `GetTimeZoneUseCase` si varios ViewModels dependen de las zonas horarias para mostrar el mensaje adecuado en la pantalla.

Para obtener más información sobre esta capa, consulta la página sobre la capa del dominio (https://developer.android.com/jetpack/guide/domain-layer?hl=es-419).

Cómo administrar dependencias entre componentes

Las clases de tu app dependen de otras para funcionar correctamente. Puedes usar cualquiera de los siguientes patrones de diseño para recopilar las dependencias de una clase en particular:

- Inserción de dependencia (DI) (https://developer.android.com/training/dependency-injection?hl=es-419): Permite que las clases definan sus dependencias sin construirlas. En el tiempo de ejecución, otra clase es responsable de proporcionar estas dependencias.
- Localizador de servicios (https://en.wikipedia.org/wiki/Service_locator_pattern): Su patrón brinda un registro en el que las clases pueden obtener sus dependencias en lugar de construirlas.

Estos patrones te permiten hacer un escalamiento del código, ya que proporcionan patrones claros para administrar dependencias sin duplicar el código ni aumentar la complejidad. Además, te permiten cambiar rápidamente entre las implementaciones de prueba y de producción.

Te recomendamos seguir los patrones de inserción de dependencia y usar la biblioteca Hilt (https://developer.android.com/training/dependency-injection/hilt-android?hl=es-419) **en las apps para Android.** Hilt construye automáticamente objetos mediante un recorrido del árbol de dependencias, proporciona garantías de tiempo de compilación sobre dependencias y crea contenedores de dependencias para clases de marco de trabajo de Android.

Prácticas recomendadas generales

La programación es una disciplina creativa y crear apps de Android no es una excepción. Hay muchas maneras de resolver un problema: puedes comunicar datos entre varias actividades o fragmentos, recuperar datos remotos y conservarlos a nivel local para el modo sin conexión, o bien controlar cualquier cantidad de situaciones comunes con las que pueden encontrarse las apps no triviales.

Aunque las siguientes recomendaciones no son obligatorias, en la mayoría de los casos, si las sigues, tu código base será más confiable, tendrá mayor capacidad de prueba y será más fácil de mantener a largo plazo.

No almacenes datos en los componentes de la app

Evita designar los puntos de entrada de tu app (receptores de transmisiones, servicios y actividades) como fuentes de datos. En cambio, solo deben coordinar con otros componentes para recuperar el subconjunto de datos relevante para ese punto de entrada. Cada componente de la app tiene una duración relativamente corta, según la interacción que el usuario tenga con su dispositivo y el estado general del sistema en ese momento.

Reduce las dependencias de clases de Android

Los componentes de tu app deben ser las únicas clases que dependan de las API del SDK de framework de Android, como `Context` (https://developer.android.com/reference/android/content/Context?hl=es-419) o `Toast` (https://developer.android.com/guide/topics/ui/notifiers/toasts?hl=es-419). La abstracción de otras clases en tu app fuera de ellas ayuda con la capacidad de prueba y reduce el acoplamiento (https://en.wikipedia.org/wiki/Coupling_(computer_programming)) dentro de la app.

Crea límites de responsabilidad bien definidos entre varios módulos de tu app.

Por ejemplo, no extiendas el código que carga datos de la red entre varias clases o paquetes en tu código base. Del mismo modo, no definas varias responsabilidades no relacionadas, como caché de datos y vinculación de datos, en la misma clase. Podría ser útil que siguieras la arquitectura de la app recomendada (#recommended-app-arch).

Expón lo mínimo indispensable de cada módulo

Por ejemplo, no caigas en la tentación de crear un acceso directo que exponga un detalle interno de la implementación de un módulo. Quizás ahorres algo de tiempo a corto plazo, pero tendrás más probabilidades de que se generen problemas técnicos a medida que tu código base evolucione.

Concéntrate en aquello que hace única a tu app para que se destaque del resto

No desperdicies tu tiempo reinventando algo que ya existe ni escribiendo el mismo código estándar una y otra vez. En cambio, enfoca tu tiempo y tu energía en aquello que hace que tu app sea única y deja que tanto las bibliotecas de Jetpack como las otras recomendadas se ocupen del código estándar repetitivo.

Piensa en cómo lograr que cada parte de tu app se pueda probar por separado

Por ejemplo, una API bien definida para obtener datos de la red facilitará las pruebas que realices en el módulo que conserve esa información en la base de datos local. En cambio, si combinas la lógica de estos dos módulos en un solo lugar, o bien si distribuyes el código de red por todo tu código base, será mucho más difícil (y quizás hasta imposible) ponerlo a prueba eficazmente.

Conserva la mayor cantidad posible de datos relevantes y actualizados

De esa manera, los usuarios podrán aprovechar la funcionalidad de tu app, incluso cuando su dispositivo esté en modo sin conexión. Recuerda que no todos tus usuarios cuentan con una conexión de alta velocidad de manera constante y, si lo hacen, pueden tener una mala recepción en lugares muy concurridos.

Ejemplos

En los siguientes ejemplos de Google, se demuestra una buena arquitectura de la app. Explóralos para ver esta guía en práctica:

- iosched (https://github.com/google/iosched), la app de Google I/O
- Sunflower (https://github.com/android/sunflower)

- Trackr (https://github.com/android/trackr)
- Jetnews (https://github.com/android/compose-samples/tree/main/JetNews) (que se implementó con Jetpack Compose)
- Ejemplos de arquitectura (https://github.com/android/architecture-samples)

Last updated 2022-01-05 UTC.