🚀 Presentación: Desarrollo Avanzado de Aplicaciones Android en Java

1. Interacción con Servicios Web y APIs 🌐

Concepto	Teoría	Ejemplo Clave
APIs REST	Son un conjunto de principios de arquitectura que define cómo los sistemas de red se comunican sin estado. Son esenciales para aplicaciones móviles, ya que permiten la comunicación con servidores (p. ej., para obtener o enviar datos) usando métodos HTTP estándar (GET, POST, PUT, DELETE).	Peticiones a un servidor para obtener el listado de películas de la semana.
Llamadas de red con Retrofit	Es una librería "Type-safe HTTP client" para Android y Java que simplifica enormemente las llamadas de red. Permite definir la	Definir una interfaz ApiService.java con un método anotado con @GET("users/{id}") para obtener un usuario. Se

	API como una **interfaz Java**, usando anotaciones para especificar los métodos HTTP (@GET, @POST), las URL y los parámetros.	debe incluir el permiso de internet en el Manifest.
Parsing de datos JSON con Gson	Gson es una librería de Google que se usa para serializar (objeto Java a JSON) y deserializar (JSON a objeto Java) datos. Automáticamente mapea la estructura JSON a **clases de modelo Java**.	Usar Gson.fromJson(jsonString, MyClass.class) para convertir una respuesta JSON en un objeto de la clase MyClass.
Autenticación y seguridad	Se usan **tokens de autenticación** (como JWT) para validar que el usuario tiene permiso para acceder a ciertos recursos del servidor. El token se envía típicamente en la cabecera HTTP (Authorization: Bearer <token>).	Implementar un **Interceptor de OkHttpClient** (que usa Retrofit internamente) para adjuntar automáticamente el token guardado a todas las peticiones salientes.

2. Concurrencia e Hilos de Ejecución ⚙️

Concepto	Teoría	Ejemplo Clave
Hilos y Procesos	El **hilo principal (UI Thread)** es el responsable de manejar la interfaz de usuario. Si se bloquea con operaciones largas (red, base de datos), la aplicación se congela ("Application Not Responding" o ANR). Los **hilos de trabajo** se usan para ejecutar estas tareas en segundo plano.	Tarea pesada: Descargar un archivo grande. El código de la descarga se ejecuta en un **hilo de trabajo** para no bloquear el **UI Thread**.
Manejo de Tareas Asíncronas	**AsyncTask** (aunque obsoleto, es fundamental para entender el concepto) permitía realizar operaciones en segundo plano (doInBackground()) y publicar resultados en el hilo principal (onPostExecute()) de forma	Cargar 1000 imágenes desde disco en doInBackground(). Mostrar un ProgressBar en onPreExecute() y ocultarlo en onPostExecute().

	segura y sencilla.	
Programación con ExecutorService	Es una herramienta más avanzada y flexible para gestionar **pools de hilos**. Permite controlar el número de hilos de trabajo activos y reutilizarlos eficientemente para evitar la sobrecarga de crear y destruir hilos constantemente.	Usar un Executors.newFixedThreadPool(4) para limitar a 4 el número de peticiones a una base de datos que se ejecutan simultáneamente.
Manejo de errores	Es crucial capturar y manejar **excepciones** dentro de los hilos de trabajo, ya que si un hilo de trabajo lanza una excepción no capturada, puede detener la aplicación.	Usar bloques try-catch dentro del código del hilo de trabajo para registrar (Log.e()) o informar de fallos de red/datos al hilo principal para mostrar un mensaje de error al usuario.

3. Arquitectura y Patrones de Diseño 🧬

Concepto	Teoría	Ejemplo Clave
Patrón M.V.V.M.	**Model-View-ViewModel**. Es un patrón de diseño que separa la interfaz de usuario (View/Activity) de la lógica de negocio (Model) y utiliza un **ViewModel** como intermediario para exponer los datos a la View, simplificando las pruebas y manteniendo los datos incluso después de cambios de configuración (p. ej., rotación de pantalla).	La **Activity/Fragment (View)** observa un **LiveData** en el **ViewModel**. El **ViewModel** llama al **Repository** para obtener datos.
Componentes de la Arquitectura de Android (AAC)	**ViewModel** y **LiveData** son componentes clave: **ViewModel** almacena y gestiona datos relacionados con la UI de forma que sobreviven a los cambios de configuración.	El GameViewModel expone un MutableLiveData<String> (currentScrambledWord) que la Activity observa y actualiza un TextView en la UI cuando el valor cambia.

	LiveData es un contenedor de datos observable que notifica a los *observers* (View) cuando los datos cambian, respetando el ciclo de vida.	
Inyección de dependencias (DI)	Es un principio de diseño donde un objeto recibe otros objetos de los que depende (sus dependencias). Librerías como **Dagger** o **Hilt** automatizan la creación y gestión de estas dependencias, haciendo el código más modular, reutilizable y fácil de probar.	Usar @Inject constructor() en una clase (UserRepository) para que Dagger sepa cómo crear y proporcionar una instancia de esta clase cuando se solicite.
Patrones de repositorio	El **Repository** es una capa que centraliza la lógica de acceso a datos, decidiendo si obtener la información de la red o de una base de datos local (caché). Aísla el ViewModel de la fuente de datos real.	El UserRepository tiene dos métodos: getUsersFromNetwork() y getUsersFromDatabase(). El ViewModel solo llama a userRepository.getUsers(), sin saber de dónde provienen los datos.

4. Optimización y Rendimiento ⚡

Concepto	Teoría	Ejemplo Clave
Optimización de layouts	Reducir la complejidad y la profundidad de la jerarquía de vistas para acelerar el tiempo de dibujo. **ConstraintLayout** es la herramienta principal, ya que permite crear diseños complejos y planos (sin anidamiento excesivo de LinearLayouts).	Reemplazar LinearLayout anidados por un solo ConstraintLayout para reducir el "overdraw" y acelerar el renderizado.
Carga eficiente de imágenes	Librerías como **Glide** se utilizan para descargar imágenes asíncronamente, manejar la **caché** (en memoria y en disco) y escalar imágenes al tamaño del ImageView de destino, reduciendo el consumo de memoria.	Usar Glide.with(context).load(url).into(imageView); para cargar una imagen remota directamente en un ImageView.

Análisis de memoria	**Android Profiler** es la herramienta clave para monitorear el consumo de CPU, red, batería y memoria en tiempo real. Permite detectar **fugas de memoria** (memory leaks) causadas por referencias persistentes a objetos que ya no deberían existir (p. ej., una Activity que se destruyó).	Usar el **Memory Profiler** para forzar un *Garbage Collection* y observar si la memoria ocupada por una Activity se libera después de que el usuario sale de ella.
Pruebas de rendimiento	Medición de métricas clave como el **tiempo de inicio** de la aplicación y la fluidez de la UI (FPS). La caída de *frames* (baja tasa de FPS, idealmente 60) indica un trabajo excesivo en el UI Thread.	Usar el **CPU Profiler** para identificar los métodos que consumen más tiempo durante el inicio de la app y optimizarlos.

5. Temas Avanzados 🌟

Concepto	Teoría	Ejemplo Clave
Notificaciones push	**Firebase Cloud Messaging (FCM)** es el servicio de Google para enviar notificaciones a los dispositivos Android. Distingue entre mensajes de **notificación** (manejados por la bandeja del sistema) y mensajes de **datos** (manejados por el código de la app).	Crear un servicio que extienda FirebaseMessagingService y anular el método onMessageReceived() para recibir y procesar mensajes de datos cuando la app está en primer o segundo plano.
Publicación en Google Play Store	El proceso requiere generar un **APK** o **AAB (Android App Bundle)** firmado con una clave privada (keystore). El **AAB** es el formato recomendado, ya que permite a Google Play optimizar el tamaño de la descarga para cada	Usar la opción "Generate Signed Bundle/APK" en Android Studio y subir el AAB a Google Play Console para iniciar el proceso de revisión y lanzamiento.

	dispositivo.	
Monetización	Se refiere a las estrategias para generar ingresos, siendo las más comunes: **AdMob** (publicidad dentro de la app, como *banners* o anuncios intersticiales) y **compras dentro de la aplicación (In-App Purchases)**, como suscripciones o ítems virtuales.	Integrar el SDK de **AdMob** y colocar un componente AdView en el layout de una Activity para mostrar un *banner* publicitario.
Integración de Firebase	Además de FCM, Firebase ofrece otras herramientas valiosas: **Analytics** (para medir el uso de la aplicación y el comportamiento del usuario) y **Crashlytics** (para el seguimiento y reporte en tiempo real de *crashes* y errores no fatales).	Integrar **Crashlytics** para recibir automáticamente informes de errores con el *stack trace* en la consola de Firebase, ayudando a diagnosticar fallas en producción.