Laboratorio Final.
Github: https://github.com/Yee404/Plataformas_LaboratorioFinal

RESUMEN DE LA ELABORACIÓN DEL PROYECTO

(con ayuda de chatgpt, grabaciones de clases, repositorios del curso y de proyectos)

Resumen.

Resumen Completo de la Conversación

Inicio del Proyecto

El usuario está desarrollando una aplicación de Android llamada **CryptoTracker**, utilizando **Jetpack Compose** y tecnologías como **Room**, **Ktor**, y **DataStore**. El objetivo de la aplicación es mostrar un listado de criptomonedas, obtener datos de una API (CoinCap), y trabajar con persistencia local en Room. Desde el principio, el proyecto enfrentó problemas de configuración en Gradle y errores al implementar las clases necesarias.

Estructura Inicial

Estructura Original del Proyecto

```
com
└── example
    └── cryptotracker
        ├── data
        │   ├── local
        │   │   ├── AppDatabase.kt (vacío)
        │   │   ├── AssetDao.kt (vacío)
        │   │   └── AssetEntity.kt (funciona)
        │   ├── remote
        │   │   ├── ApiService.kt (vacío)
        │   │   ├── AssetResponse.kt (vacío)
        │   │   └── HttpClientFactory.kt (funciona parcialmente)
        │   └── repository
        │       └── AssetRepository.kt (errores)
        ├── domain
        │   └── Asset.kt (vacío)
        ├── ui
        │   ├── components (vacío)
        │   ├── navigation
        │   │       └── NavGraph.kt (funciona)
        │   ├── screens
        │   │       ├── detail
        │   │       │       └── AssetDetailScreen.kt (funciona)
        │   │       ├── list
        │   │       │       └── AssetListScreen.kt (errores)
        ├── utils
        │       └── Resource.kt (vacío)
        └── MainActivity.kt (errores)
```

Problemas Iniciales

- Muchas clases como `AppDatabase`, `AssetDao`, `AssetResponse`, y `Asset.kt` estaban vacías o mal configuradas.
- La clase `AssetRepository` tenía múltiples errores relacionados con dependencias ausentes y métodos incorrectos.
- El `MainActivity` generaba errores porque no se configuró correctamente la instancia de `ApiService`.

Cambios Realizados

1. `AssetEntity`

Se creó correctamente para representar los datos de cada criptomoneda, definiendo campos como `id`, `name`, `symbol`, y datos adicionales como `priceUsd`, `supply`, y `marketCapUsd`.

2. `HttpClientFactory`

Se implementó para configurar un cliente **Ktor** con soporte para:

- **Logging**: Para depurar tráfico de red.
- **Content Negotiation**: Aunque inicialmente esta parte fue ignorada, se completó después.
- **Default Request**: Se configuró el tipo de contenido JSON por defecto.

3. `AssetRepository`

Se corrigió para que interactuara con **ApiService** y **AssetDao**, manejando tanto datos remotos como locales:

- Se usó `Flow` para emitir estados (`Loading`, `Success`, `Error`).
- Se agregó persistencia local en Room.

4. `AppDatabase`

Se definió para usar **Room** y manejar la entidad `AssetEntity`:

- Configuración singleton para garantizar una única instancia.
- Declaración del DAO correspondiente (`AssetDao`).

5. `ApiService`

Se implementó para realizar solicitudes HTTP utilizando **Ktor**, interactuando con la API de CoinCap para obtener datos de criptomonedas.

6. `AssetListScreen`

Se corrigió para mostrar un listado de criptomonedas:

- Se conectó a un `AssetListViewModel` para manejar el estado.
- Se agregó un `LazyColumn` para renderizar los datos.

7. `AssetListViewModel`

Se implementó para manejar la lógica de negocio:

- Uso del patrón **MVVM**.
- Interacción con `AssetRepository`.

8. Navegación (NavGraph)

Se configuró un sistema de navegación para conectar las pantallas:

- Pantalla de listado (`AssetListScreen`).
- Pantalla de detalle (`AssetDetailScreen`).

Cambios en la Estructura del Proyecto

Después de implementar las clases necesarias, la estructura quedó así:

```
com
└── example
    └── cryptotracker
        ├── data
        │   ├── local
        │   │   ├── AppDatabase.kt (completo)
        │   │   ├── AssetDao.kt (completo)
        │   │   └── AssetEntity.kt (completo)
        │   ├── remote
        │   │   ├── ApiService.kt (completo)
        │   │   ├── AssetResponse.kt (completo)
        │   │   └── HttpClientFactory.kt (completo)
        │   └── repository
        │       └── AssetRepository.kt (completo)
        ├── domain
        │   └── Asset.kt (completo)
        ├── presentation
        │   ├── viewmodels
        │   │       └── AssetListViewModel.kt (completo)
        ├── ui
        │   ├── components (vacío)
```

```
│   ├── navigation
│   │       └── NavGraph.kt (completo)
│   ├── screens
│   │       ├── detail
│   │       │       └── AssetDetailScreen.kt (completo)
│   │       ├── list
│   │       │       └── AssetListScreen.kt (completo)
├── utils
│       └── Resource.kt (completo)
└── MainActivity.kt (funciona)
```

Problemas y Soluciones

Errores en Gradle

1. **Dependencias faltantes**:
    a. **`androidx.core:splashscreen`** y **`kotlinx-serialization-json`** generaron errores porque las versiones configuradas no existían.
    b. Solución: Se actualizaron a versiones estables:
    `implementation("androidx.core:core-splashscreen:1.0.1")`
    `implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.5.1")`

2. **`jcenter` obsoleto**:
    a. `jcenter()` fue eliminado del proyecto y reemplazado por `mavenCentral()`.
3. **Compatibilidad con Gradle**:
    a. Se actualizó **Gradle Wrapper** a la versión `8.0`:
    `distributionUrl=https\://services.gradle.org/distributions/gradle-8.0-bin.zip`

4. **Configuración de Repositorios**:
    a. Se ajustaron los repositorios en `settings.gradle.kts`:

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
    }
}
```

Errores Persistentes

Incluso después de resolver múltiples problemas, el proyecto aún enfrentó dificultades para construir el APK debido a configuraciones incompatibles y errores en la sincronización de dependencias. Esto se atribuyó a:

- Malas configuraciones en dependencias.
- Incompatibilidades entre las versiones de plugins, Gradle y las librerías usadas.

Estado Final

El proyecto logró configurar correctamente:

- **Navegación entre pantallas.**
- **Conexión a la API CoinCap.**
- **Persistencia local con Room.**
- **Interfaz funcional con Jetpack Compose.**

Sin embargo, el APK no se pudo generar debido a errores en dependencias específicas y configuraciones avanzadas de Gradle.

Guía Paso a Paso para el Laboratorio Final: CryptoTracker

Esta guía detalla cómo desarrollar la aplicación **CryptoTracker**, asegurando claridad y especificidad en cada paso. Cada acción incluye dónde realizarla, cómo configurarla y qué propósito cumple.

1. Configuración Inicial

1.1. Crear el Proyecto

1. Abre Android Studio.
2. Selecciona **"New Project"**.
3. Escoge la plantilla **"Empty Compose Activity"**.
4. Configura:
    a. **Nombre del proyecto**: `CryptoTracker`.
    b. **Lenguaje**: Kotlin.
    c. **SDK Mínimo**: API 21 o superior (recomendado para Jetpack Compose). //API 21, Android 5.0
    d. **Directorio base**: Selecciona una ubicación adecuada.
5. Haz clic en **"Finish"** para crear el proyecto.

1.2. Configuración de Dependencias en `build.gradle`

Archivo: `build.gradle (:app)`

1. Abre el archivo en `app/build.gradle`.
2. Actualiza las dependencias con las siguientes librerías requeridas:

```
    plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'kotlin-kapt'
}

dependencies {
    // Jetpack Compose
    implementation "androidx.compose.ui:ui:1.x.x"
    implementation "androidx.compose.material3:material3:1.x.x"
    implementation "androidx.lifecycle:lifecycle-viewmodel-compose:2.x.x"

    // Ktor para networking
    implementation "io.ktor:ktor-client-core:2.x.x"
    implementation "io.ktor:ktor-client-android:2.x.x"
```

```
    implementation "io.ktor:ktor-client-logging:2.x.x"

    // Room para base de datos local
    implementation "androidx.room:room-runtime:2.x.x"
    kapt "androidx.room:room-compiler:2.x.x"

    // Navegación en Compose
    implementation "androidx.navigation:navigation-compose:2.x.x"

    // Accompanist (indicadores visuales como carga)
    implementation "com.google.accompanist:accompanist-
systemuicontroller:0.x.x"
}
```

3. Sincroniza el proyecto haciendo clic en **"Sync Now"**.

Archivo: `build.gradle (Proyecto)`

1. Asegúrate de que esté configurada la versión correcta de Kotlin y Compose.

```
ext {
    compose_version = '1.x.x'
}
```

1.3. Permisos en `AndroidManifest.xml`

1. Abre `AndroidManifest.xml` en `app/src/main`.
2. Agrega el permiso de internet dentro del nodo `<manifest>`:

```
<uses-permission android:name="android.permission.INTERNET" />
```

1.4. Organización del Proyecto

1. Navega a la carpeta `java/com.example.cryptotracker`.
2. Crea los siguientes paquetes:
    a. `data`: Para manejo de datos.
        i. Subpaquetes:
            1. `network`: Para Ktor.
            2. `database`: Para Room.
    b. `domain`: Para modelos y lógica de negocio.
    c. `presentation`: Para pantallas y UI.
    d. `navigation`: Para rutas y configuración de navegación.

2. Configuración de Networking con Ktor

2.1. Implementación del Cliente HTTP

1. En el paquete `data/network`, crea un archivo llamado `HttpClientFactory.kt`.
2. Configura el cliente HTTP:

```
object HttpClientFactory {
    fun create(): HttpClient {
        return HttpClient(Android) {
            install(Logging) {
                level = LogLevel.ALL
            }
        }
    }
}
```

2.2. Definir la Interfaz de API

1. En el mismo paquete, crea `ApiService.kt`.
2. Define los métodos para consumir el API:

```
interface ApiService {
    suspend fun getAssets(): List<Asset>
    suspend fun getAssetById(id: String): Asset
}
```

2.3. Implementar el API

1. En el mismo paquete, crea `ApiServiceImpl.kt`.
2. Implementa la interfaz:

```
class ApiServiceImpl(private val client: HttpClient) : ApiService {
    override suspend fun getAssets(): List<Asset> {
        return client.get("https://api.coincap.io/v2/assets")
    }

    override suspend fun getAssetById(id: String): Asset {
        return client.get("https://api.coincap.io/v2/assets/$id")
    }
}
```

3. Configuración de la Base de Datos Local con Room

3.1. Crear la Entidad

1. En data/database, crea un archivo AssetEntity.kt.
2. Define la entidad para la tabla:

```
@Entity(tableName = "assets")
data class AssetEntity(
    @PrimaryKey val id: String,
    val name: String,
    val symbol: String,
    val priceUsd: String,
    val changePercent24Hr: String,
    val timestamp: Long
)
```

3.2. Definir el DAO

1. En el mismo paquete, crea AssetDao.kt.
2. Agrega las funciones necesarias:

```
@Dao
interface AssetDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAll(assets: List<AssetEntity>)

    @Query("SELECT * FROM assets")
    fun getAllAssets(): Flow<List<AssetEntity>>

    @Query("SELECT * FROM assets WHERE id = :id")
    fun getAssetById(id: String): Flow<AssetEntity>
}
```

3.3. Configurar la Base de Datos

1. En el mismo paquete, crea AppDatabase.kt.
2. Configura Room:

```
@Database(entities = [AssetEntity::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun assetDao(): AssetDao
}
```

3.4. Singleton de Room

1. En `data/database`, crea `DatabaseModule.kt`.
2. Implementa el singleton:

```
object DatabaseModule {
    private var INSTANCE: AppDatabase? = null

    fun getDatabase(context: Context): AppDatabase {
        return INSTANCE ?: synchronized(this) {
            Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "crypto_tracker_db"
            ).build().also { INSTANCE = it }
        }
    }
}
```

4. Implementación de UI con Jetpack Compose

4.1. Pantalla de Listado de Assets

1. En `presentation/screens`, crea `AssetListScreen.kt`.
2. Implementa la pantalla con un `LazyColumn` para mostrar cada criptomoneda:

```
@Composable
fun AssetListScreen(navController: NavController, viewModel:
AssetListViewModel) {
    val state by viewModel.state.collectAsState()

    when (state) {
        is AssetState.Loading -> CircularProgressIndicator()
        is AssetState.Success -> {
            LazyColumn {
                items((state as AssetState.Success).assets) { asset ->
                    AssetCard(asset) {

navController.navigate("asset_detail/${asset.id}")
                    }
                }
            }
        }
        is AssetState.Error -> Text("Error loading assets")
    }
```

```
}
```

4.2. Pantalla de Detalle del Asset

1. En el mismo paquete, crea `AssetDetailScreen.kt`.
2. Implementa la pantalla para mostrar detalles adicionales:

```
@Composable
fun AssetDetailScreen(navController: NavController, assetId: String) {
    // Carga los detalles del asset usando el ID
}
```

4.3. Configuración de Navegación

1. En `navigation`, crea `NavGraph.kt`.
2. Configura el `NavHost`:

```
@Composable
fun NavGraph(startDestination: String = "asset_list") {
    val navController = rememberNavController()

    NavHost(navController, startDestination) {
        composable("asset_list") { AssetListScreen(navController) }
        composable("asset_detail/{id}") { backStackEntry ->
            val id = backStackEntry.arguments?.getString("id") ?: ""
            AssetDetailScreen(navController, id)
        }
    }
}
```

5. Finalización y Pruebas

1. **Verifica la funcionalidad**:
    a. Datos online desde Ktor.
    b. Sincronización offline con Room.
2. **Optimiza la UI**:
    a. Asegúrate de mostrar indicadores visuales adecuados para estados de carga y error.

Con estos pasos, tendrás una aplicación funcional que cumple con todos los requisitos del laboratorio.

Análisis Detallado de los Archivos ZIP

A continuación, se analiza cada archivo proporcionado en términos de estructura, funcionalidad, y recursos relevantes para el laboratorio **CryptoTracker**.

1. `proyecto-app-movil.zip`

Estructura

- **`data`**:
  - `network`:
    - Clases como `ApiClient`, `safeCall`, y `NetworkError`.
    - Implementaciones para realizar solicitudes HTTP de manera segura.
  - `database`:
    - Configuración de Room: entidades, DAO y base de datos.
- **`presentation`**:
  - Pantallas diseñadas con Jetpack Compose.
  - Ejemplos de `State` y manejo de eventos.
- **`navigation`**:
  - Uso de `NavHost` para la navegación entre pantallas.

Funcionalidad

- Ejemplo funcional de una aplicación basada en API con Ktor y Room.
- Manejo de errores de red con clases reutilizables.
- Arquitectura MVVM aplicada correctamente.

Relevancia

- Contiene todos los recursos necesarios para implementar las recomendaciones del laboratorio:
  - `safeCall` para manejar errores.
  - Configuración de Room.
  - Manejo de estados y navegación en Compose.

2. `MelodyMaster.zip`

Estructura

- **`media`**:
  - Manejo de audio y video.
- **`ui`**:

    - Pantallas básicas con Jetpack Compose.
- **`navigation`**:
    - Navegación simple con `NavHost`.

Funcionalidad

- Ejemplo básico de una aplicación multimedia.
- Implementaciones específicas para reproducir audio y video.
- Uso limitado de Room y Ktor.

Relevancia

- Relevancia baja para el laboratorio.
- Algunos patrones de navegación podrían ser útiles, pero no contiene implementaciones avanzadas de Ktor ni Room.

3. `uvg-rickmorty.zip`

Estructura

- **`data`**:
    - **`network`**:
        - Cliente Ktor con configuración reutilizable.
        - DTOs (`CharacterDto`) para transformar respuestas del API.
    - **`database`**:
        - Configuración completa de Room con entidades (`CharacterEntity`) y DAO.
- **`presentation`**:
    - Pantallas bien estructuradas con Jetpack Compose.
    - Manejo de estados: `Loading`, `Success`, `Error`.
- **`navigation`**:
    - Uso avanzado de `NavHost` para pasar parámetros entre pantallas.

Funcionalidad

- Aplicación funcional basada en el API de Rick and Morty.
- Sincronización online/offline con Room.
- Manejo avanzado de estados para diferenciar entre `Loading`, `Error`, y datos cargados.

Relevancia

- Altamente relevante:
    - Ejemplo ideal de manejo de datos online y offline.
    - Estructura de navegación reutilizable para el laboratorio.
    - Clases para manejar estados y eventos listos para ser adaptados.

4. `PlataformasUvg2024.zip`

Estructura

- **`app/src/main/java/com/uvg/ejercicioslabs`**:
  - **`ejercicios`**:
    - **`ktor`**:
      - `safeCall`, `responseToResult`, `NetworkError`, y `HttpClientFactory`.
    - **`room`**:
      - Entidades, DAO, y base de datos configurados.
    - **`navigation`**:
      - Ejercicios avanzados con navegación anidada y paso de parámetros seguros.
  - **`datastore`**:
    - Ejemplos de almacenamiento de preferencias.
  - **`firebase`**:
    - Configuración para integrar Firebase (no relevante para este laboratorio).

Funcionalidad

- Amplio conjunto de ejemplos que cubren:
  - Arquitectura MVVM.
  - Navegación avanzada en Compose.
  - Uso de Ktor y Room con patrones modernos.

Relevancia

- Máxima relevancia:
  - Contiene todas las clases necesarias para el laboratorio y versiones actualizadas.
  - Ejemplos de integración de Room y Ktor.
  - Manejo avanzado de estados y eventos.

Comparación General

Archivo	Recursos Relevantes	Relevancia
`proyecto-app-movil.zip`	Manejo de red (Ktor), Room, estados, y navegación.	Alta
`MelodyMaster.zip`	Poca utilidad, enfocado en multimedia.	Baja

`uvg-rickmorty.zip`	Ejemplo directo de sincronización online/offline, manejo de estados, y uso avanzado de Ktor y Room.	Muy Alta
`PlataformasUvg2024.zip`	Clases reutilizables (`safeCall`, `NetworkError`, `HttpClientFactory`), Room, y navegación avanzada.	Muy Alta

Recomendaciones para el Laboratorio

1. **Priorizar recursos de `uvg-rickmorty.zip` y `PlataformasUvg2024.zip`**:
    a. Copiar configuraciones avanzadas de Ktor, Room, y navegación.
    b. Usar las implementaciones de `safeCall`, `responseToResult`, y `HttpClientFactory`.
2. **Usar `proyecto-app-movil.zip` como referencia secundaria**:
    a. Reutilizar conceptos básicos para MVVM y manejo de estados.
3. **Evitar `MelodyMaster.zip`**, ya que no aporta recursos útiles para este laboratorio.

Con estas recomendaciones, podrás implementar una solución moderna y funcional que cumpla con los requisitos del laboratorio.

Estructura “Inicial” Completa del Proyecto CryptoTracker

La estructura incluye carpetas y archivos organizados según las prácticas modernas de desarrollo en Android Studio, utilizando la última versión de las herramientas y librerías recomendadas (como Firebase, Ktor, Room y Jetpack Compose).

Directorio Base

```
CryptoTracker/
├── app/
│   ├── build.gradle
│   ├── proguard-rules.pro
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/
│   │   │   │   └── com/example/cryptotracker/
│   │   │   │       ├── data/
│   │   │   │       │   ├── network/
│   │   │   │       │   │   ├── ApiService.kt
│   │   │   │       │   │   ├── ApiServiceImpl.kt
│   │   │   │       │   │   ├── HttpClientFactory.kt
│   │   │   │       │   │   ├── NetworkError.kt
│   │   │   │       │   │   ├── responseToResult.kt
│   │   │   │       │   │   ├── safeCall.kt
│   │   │   │       │   ├── database/
│   │   │   │       │   │   ├── AssetEntity.kt
│   │   │   │       │   │   ├── AssetDao.kt
│   │   │   │       │   │   ├── AppDatabase.kt
│   │   │   │       │   │   ├── DatabaseModule.kt
│   │   │   │       ├── domain/
│   │   │   │       │   ├── model/
│   │   │   │       │   │   ├── Asset.kt
│   │   │   │       │   ├── repository/
│   │   │   │       │   │   ├── AssetRepository.kt
│   │   │   │       ├── navigation/
│   │   │   │       │   ├── NavGraph.kt
│   │   │   │       ├── presentation/
│   │   │   │       │   ├── screens/
│   │   │   │       │   │   ├── AssetListScreen.kt
│   │   │   │       │   │   ├── AssetDetailScreen.kt
│   │   │   │       │   ├── viewmodels/
│   │   │   │       │   │   ├── AssetListViewModel.kt
│   │   │   │       │   │   ├── AssetDetailViewModel.kt
│   │   ├── res/
```

```
│   │   │   │   ├── drawable/
│   │   │   │   │   ├── ic_launcher_foreground.xml
│   │   │   │   │   ├── ic_launcher_background.xml
│   │   │   │   ├── values/
│   │   │   │   │   ├── colors.xml
│   │   │   │   │   ├── strings.xml
│   │   │   │   │   ├── themes.xml
│   │   │   └── AndroidManifest.xml
├── build.gradle
├── settings.gradle
└── gradle/
    ├── wrapper/
    │   ├── gradle-wrapper.jar
    │   ├── gradle-wrapper.properties
    └── libs.versions.toml
```

Detalles de Actualización

Ktor

- **Versión Actualizada**: Utiliza Ktor `2.x.x` para aprovechar las mejoras en estabilidad y rendimiento.
- **Clases**:
  - `HttpClientFactory.kt`: Configuración centralizada para el cliente HTTP.
  - `safeCall.kt` y `responseToResult.kt`: Manejo de errores y transformación de respuestas en resultados seguros.

Room

- **Versión Actualizada**: Room `2.x.x` con soporte para coroutines y Flow.
- **Clases**:
  - `AppDatabase.kt`: Configura la base de datos.
  - `AssetDao.kt`: Define operaciones de acceso a datos.
  - `AssetEntity.kt`: Representa la tabla en la base de datos.

Firebase (Opcional)

- Si es necesario, añade Firebase para autenticación o sincronización en tiempo real.
- **Versión Actualizada**: Firebase BoM `32.x.x`: `implementation platform('com.google.firebase:firebase-bom:32.x.x')` `implementation 'com.google.firebase:firebase-auth-ktx'` `implementation 'com.google.firebase:firebase-firestore-ktx'`

- **Configuración**:
  - Agrega `google-services.json` a la carpeta `app/`.
  - Modifica `build.gradle (app)` para incluir: `apply plugin: 'com.google.gms.google-services'`

Jetpack Compose

- **Versión Actualizada**: Compose `1.x.x` con soporte para Material 3.
- **Componentes**:
  - `LazyColumn` para listas dinámicas.
  - `Material3` para diseño moderno.

Navegación

- **Versión Actualizada**: Navigation Compose `2.x.x` para manejar rutas seguras y tipos complejos.

Prácticas Modernas

- **`libs.versions.toml`**:
  - Centraliza las versiones de dependencias: `[libraries]`
    ```
    compose-ui = "androidx.compose.ui:ui:1.x.x"
    ktor-client-core = "io.ktor:ktor-client-core:2.x.x"
    room-runtime = "androidx.room:room-runtime:2.x.x"
    ```

- **Arquitectura MVVM**:
  - `AssetListViewModel.kt` y `AssetDetailViewModel.kt` para manejar la lógica de presentación.
- **Manejo de Estados**:
  - Usa `sealed classes` para representar `Loading`, `Success`, y `Error`.

Esta estructura garantiza una aplicación robusta, actualizada, y preparada para manejar datos online y offline.