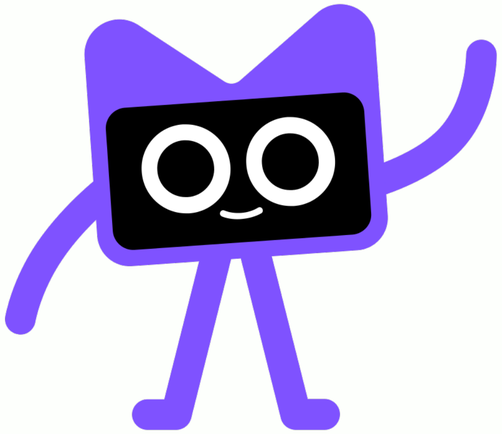
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Persistência

QXD0276 - Desenvolvimento de Software para Dispositivos Móveis



Prof. Bruno Góis Mateus (brunomateus@ufc.br)

Conteúdo

- Introdução
- A camada de dados
- Flow
- Room
- DataStore
- Injeção de dependência
- Firebase
- Resumo

Introdução

Introdução

Por que precisamos de persistência?

- App funciona mesmo offline
- Dados precisam ser guardados (ex: tarefas, usuários, configurações)
- Melhor performance (cache local)
- Economia de rede

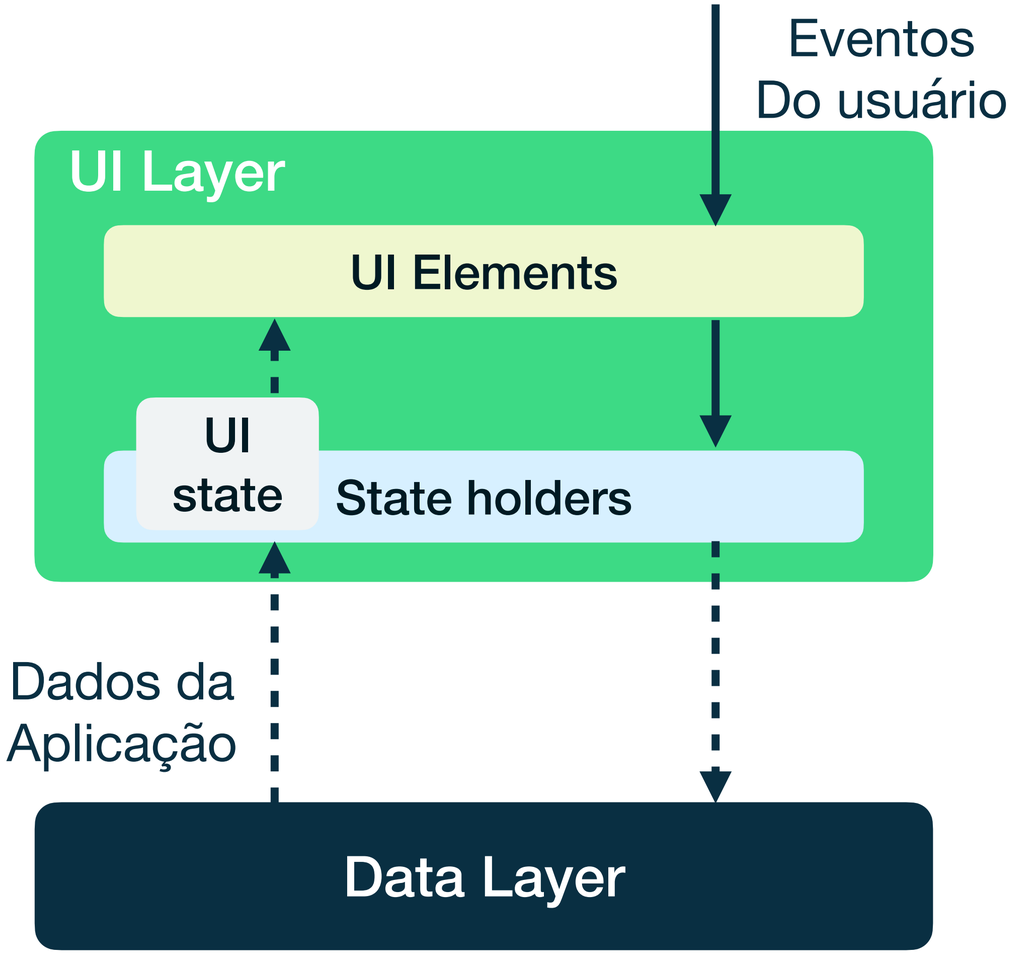
Introdução

Onde persistência é usada na vida real?

- Apps de notas
- Apps de tarefas
- Apps offline-first (WhatsApp, Spotify, YouTube)
- Apps que precisam salvar preferências (tema, idioma, tokens)

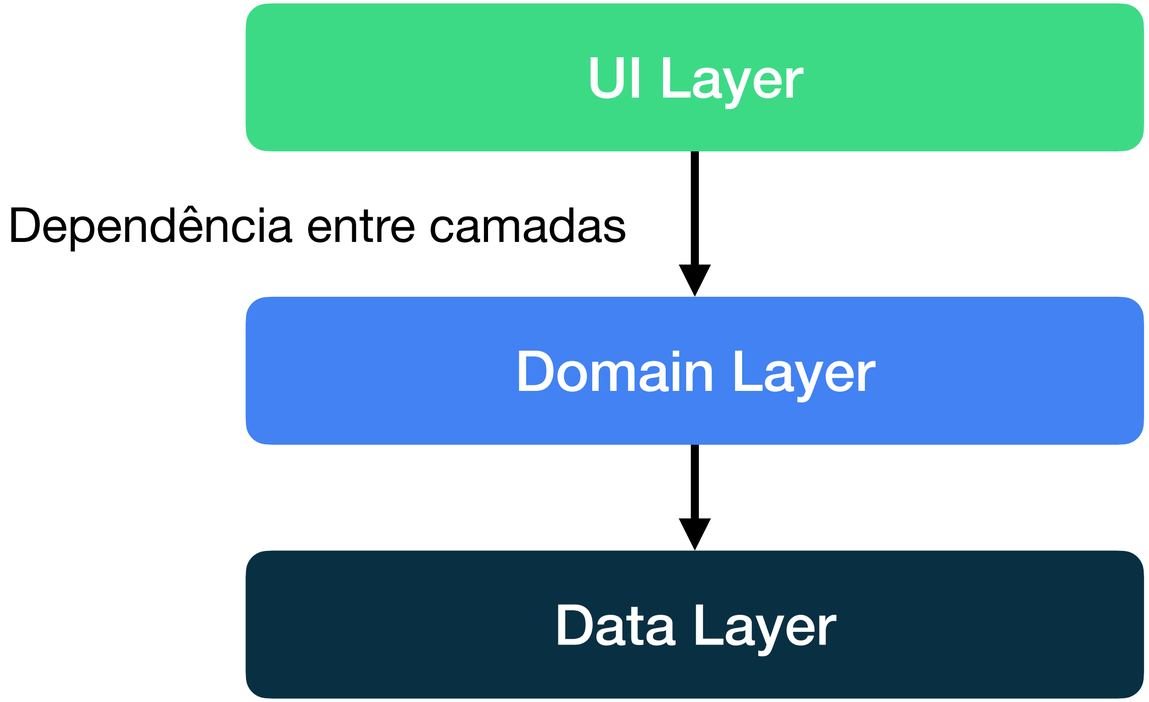
Introdução

Arquitetura Android



A camada de dados

A arquitetura recomendada



A camada de dados

O Essencial da Data Layer (Arquitetura)

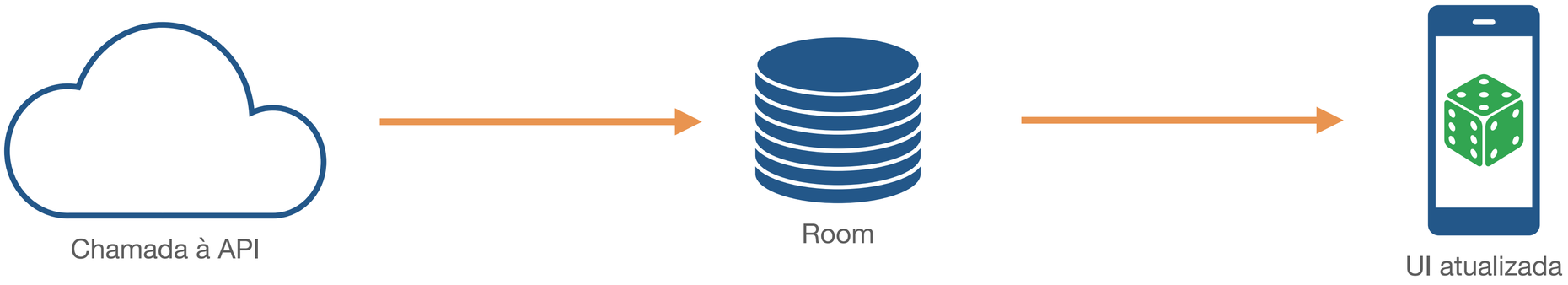
- Fonte Única de Verdade
  - É o princípio mais importante para evitar bugs de inconsistência de dados
- Principais benefícios:
  - Centraliza todas as alterações em um só lugar
  - Protege os dados para que outros tipos de dados não possam alterá-los
  - Torna as alterações nos dados mais rastreáveis e, portanto, erros mais fáceis de detectar

Se a UI exibe dados que vieram direto da rede, mas o banco de dados local diz outra coisa, quem está certo?

A camada de dados

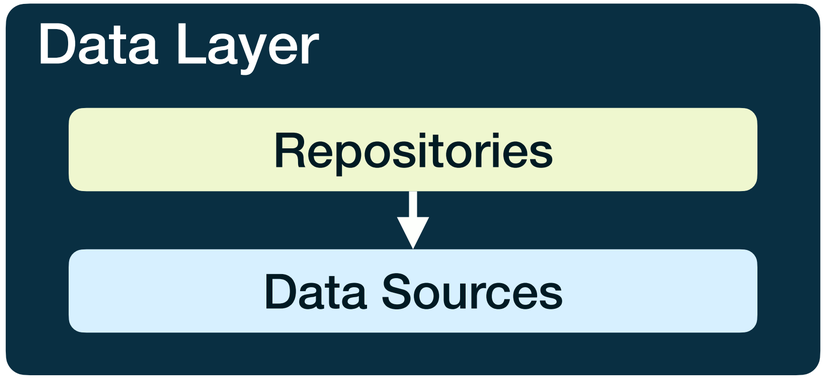
A solução

- Em um aplicativo Offline-First (padrão moderno), o Banco de Dados Local (Room) é a Fonte Única de Verdade.
  - A API (Rede) serve apenas para atualizar o banco de dados
  - A UI observa apenas o banco de dados



A camada de dados

Camada de Dados (Data Layer)



- É composta de vários repositórios
  - Cada repositório define uma fonte única da verdade
  - Um repositório pode estar associado a 0 ou a vários fontes de dados
  - Cada repositório deve gerenciar um tipo de dado

A camada de dados

Repositórios

- É a fronteira entre as regras de negócios/dados e a aplicação
  - Componente arquitetural mais crítico desta camada
- Responsáveis por:
  - Expor os dados para o restante do aplicativo
  - Centralizar as alterações nos dados
  - Resolver conflitos entre múltiplas fontes de dados
  - Manter as regras de negócios

A camada de dados

Repositórios

- Abstração:
  - O ViewModel (e o resto do app) não deve saber se os dados vêm do JSON, do SQLite, do Firebase ou de um arquivo de texto
- Arbitragem:
  - Os dados locais estão velhos? Devo buscar na rede?
  - Ocorreu um erro na rede, devo retornar o que tenho em cache?

A camada de dados

Repositórios

- Centralização:
  - Se você precisar alterar a API de REST para GraphQL, você só altera o Repository
  - Deve ser transparente para o resto da aplicação

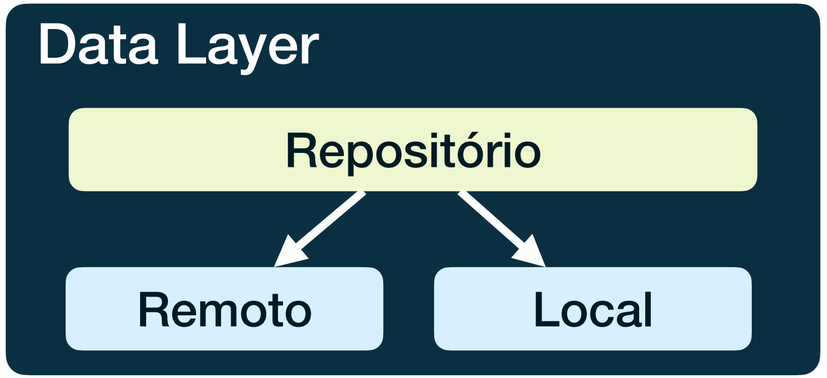
A camada de dados

Fonte de dados (Data Sources)

- Responsável por apenas uma fonte de dados
- Servem de ponte entre a aplicação e o sistema para manipulação de dados
  - Ex: um arquivo, uma fonte dados na rede ou um banco de dados local
- Vantagem:
  - Facilita muito os testes
  - É possível testar o Repository usando um FakeLocalDataSource que é apenas uma lista em memória

A camada de dados

Fonte de dados (Data Sources)



- RemoteDataSource:
  - Encapsula a interação com a API (Retrofit)
  - Lida com endpoints, headers, parsing de JSON
- LocalDataSource:
  - Encapsula a persistência
  - Ex: Room DAO, DataStore

A camada de dados

Threading

- O ViewModel ou a UI não devem saber se precisam rodar em IO ou Default
  - Repositório e os Data Sources são responsáveis por mover a execução para a thread correta
- As bibliotecas usadas na fontes de dados já fornece APIs seguras
  - Ex: Room, Retrofit ou Ktor

A camada de dados

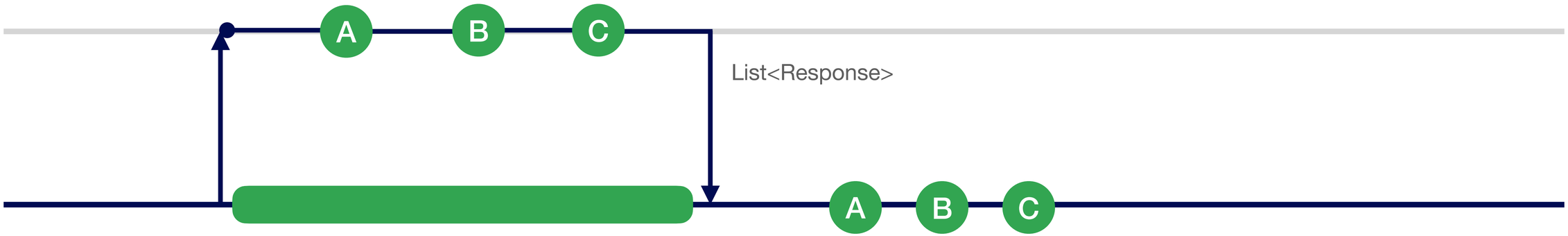
Expondo dados de API

- As classes da camada de dados geralmente expõem:
  - Funções para realizar operações CRUD (Criar, Ler, Atualizar e Deletar)
  - Notificações sobre alterações nos dados ao longo do tempo
- Operações pontuais:
  - A camada de dados deve expor suspend functions
- Notificações sobre alterações nos dados ao longo do tempo:
  - A camada de dados deve expor Flows

Flow

Flow

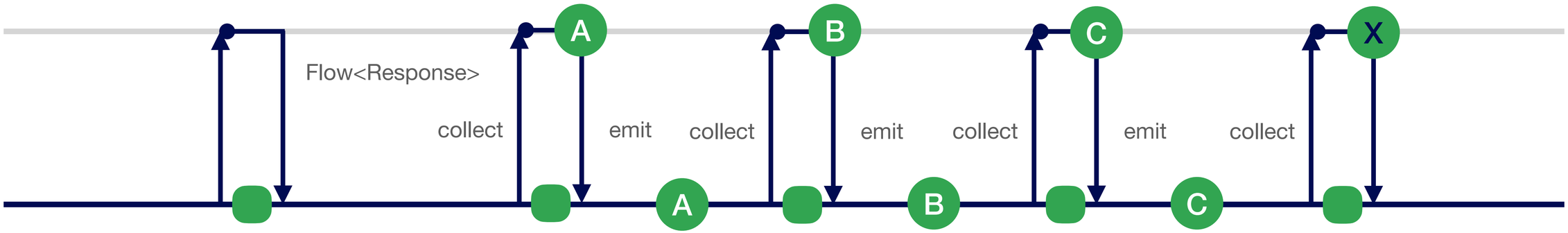
```
suspend fun foo(): List<Response> = buildList {
    add(compute("A"))
    add(compute("B"))
    add(compute("C"))
}
```




```
fun main() = runBlocking {
    val list = foo()
    for (x in list) println(x)
}
```

Flow

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```




```
fun main() = runBlocking {
    val flow = foo()
    flow.collect {x -> println(x)}
}
```

Flow

O que é?

- Uma stream de dados assíncrona e reactiva
  - Construído tendo Coroutines como base
- Emite múltiplos valores ao longo do tempo
  - Ideal para dados que mudam (ex: tabela do Room)
- Suporte ao Kotlin multiplataforma

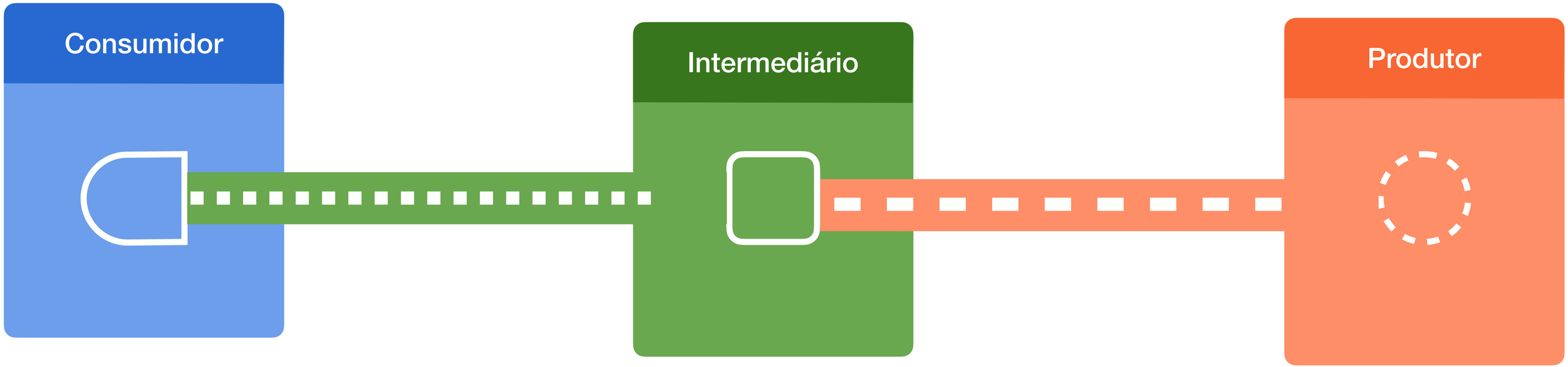
Flow

Flow no Android

Consomem os valores a partir da stream

- Podem modificar os valores emitidos ou a própria stream

Fonte de dados e repositórios são tipicamente produtores de dados



Flow

Flow vs StateFlow

Característica	Flow	StateFlow
Tipo	🥶 Cold stream	🥵 Hot stream
Comportamento	Só executa a query se houver alguém a ouvir (collect). Reinicia se a UI recriar.	Mantém o último valor em memória. Sobrevive à rotação de ecrã.
Uso	Transporte de dados da DB para o ViewModel.	Exposição de estado do ViewModel para a UI.

Produzindo o estado

Conversão para StateFlow

```
class DiceRollViewModel(
    userRepository: UserRepository
) : ViewModel() {

    val userUiState: StateFlow<String> =
        userRepository.userStream.map { user -> user.name }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5000),
            initialValue = ""
        )
}
```

É um **Flow** que é diferente de um **StateFlow**

Produz outro **Flow**

Converte um **Flow** para um **StateFlow**

Room

Room

Introdução

- Historicamente o Android utiliza o banco SQLite para armazenamento local
- Alguns dos principais problemas ao usarmos o SQLite são nativamente:
  - As consultas não são verificadas em tempo de compilação
  - Mudanças no esquema do banco devem ser gerenciadas manualmente
  - É necessário fazer uso de código boilerplate para converter os resultados das consultas em objetos do domínio da app
- É altamente recomendável usar o Room em vez do SQLite

Room

O que é?

- Abstração sobre SQLite nativo
  - ORM, Object Relational Mapping library
- Garante segurança de tipos
- Facilita queries
- Migrações simplificas
- Integra com coroutines
- Integra com Flow

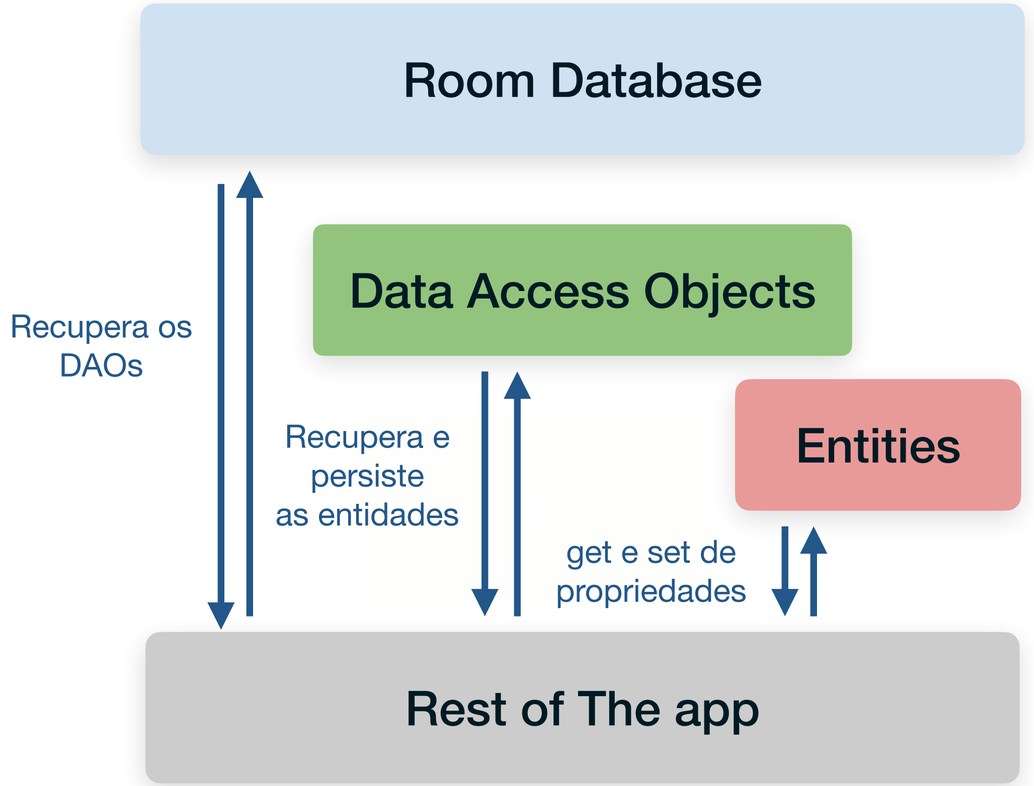
Room

Componentes principais

- A classe Banco de Dados @Database
  - Referência ao bd, serve de ponto principal de acesso ao dados
- Entidades do dados @Entity
  - Representam as tabelas do banco de dados
- DAOs (Data Access Objects) @Dao
  - Prover métodos para que sua app realizer, consultas, inserções, atualização e remoções no banco de dados

Room

- A instância do banco de dados prover instâncias de DAOs
- Os DAOs permitem o acesso aos dados associados as entidades
- As entidades são utilizadas nas consultas



Room

Dependências - Preference DataStore - libs.version.toml

```
[versions]
ksp = "2.0.21-1.0.28"
roomKtx = "2.8.4"
roomRuntime = "2.8.4"
roomCompiler = "2.8.4"

[libraries]
androidx-room-ktx = { group = "androidx.room", name = "room-ktx",
version.ref = "roomKtx" }
androidx-room-runtime = { group = "androidx.room", name = "room-
runtime", version.ref = "roomRuntime" }
androidx-room-compiler = { group = "androidx.room", name = "room-
compiler", version.ref = "roomCompiler" }
```

Room

Dependências - Preference DataStore - libs.version.toml

```
[plugins]
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```

Room

Dependências - Gradle

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.kotlin.compose)
    alias(libs.plugins.ksp)
}

dependencies {
    implementation(libs.androidx.room.ktx)
    implementation(libs.androidx.room.runtime)
    implementation(libs.androidx.datastore.preferences)
    ksp(libs.androidx.room.compiler)
}
```

Room

Criando uma entidade

```
@Entity(tableName = "tasks")
data class TaskEntity(
    @PrimaryKey(autoGenerate = true) val id: Int = 0,
    val title: String,
    val done: Boolean = false
)
```

Room

Criando um DAO

- Quando usar suspend?
  - Quando a operação devolve um único valor
  - Uma operação que não retorna stream
- Quando usar Flow?
  - Quando queremos acompanhar mudanças na tabela
  - Retornar lista atualizada automaticamente

Room

Criando uma DAO

```
@Dao
interface TaskDao {
    @Query("SELECT * FROM tasks ORDER BY id DESC")
    fun getAll(): Flow<List<TaskEntity>>

    @Insert
    suspend fun insert(task: TaskEntity)

    @Update
    suspend fun update(task: TaskEntity)

    @Delete
    suspend fun delete(task: TaskEntity)

    @Query("SELECT * FROM tasks WHERE id = :id")
    suspend fun findById(id: Int): TaskEntity?
}
```

A consulta é re-executada sempre que a tabela é atualizada, independentemente da linha atualizada fazer parte ou não do resultado. Para evitar notificações desnecessária, utilize o operador distinctUntilChanged()

Room

Criando um banco de dados

- A classe deve satisfazer as seguintes condições:
  - Deve ser anotada com @Database
    - Incluir o vetor listando todas as entidades associadas a base de dados
  - Deve ser uma class abstrata que herda de RoomDatabase
  - Para cada classe DAO, a classe deve definir um método abstrato sem parâmetros que retorna uma instância do DAO

Room

Criando um database usando Hilt

```
@Database(entities = [TaskEntity::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun taskDao(): TaskDao
}
```

Room

Criando um database usando Hilt

```
@Module
@InstallIn(SingletonComponent::class)
object DatabaseModule {

    @Provides
    @Singleton
    fun provideDatabase(
        @ApplicationContext context: Context
    ): AppDatabase = Room.databaseBuilder(
        context,
        AppDatabase::class.java,
        "app.db"
    ).build()

    @Provides
    fun provideTaskDao(db: AppDatabase) = db.taskDao()
}
```

Room

Criando um database sem Hilt

```
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null

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

Room

Repository + Room

```
class TaskRepository @Inject constructor(
    private val dao: TaskDao
) {

    val tasks: Flow<List<TaskEntity>> = dao.getAll()

    suspend fun addTask(task: TaskEntity) = dao.insert(task)

    suspend fun updateTask(task: TaskEntity) = dao.update(task)

    suspend fun deleteTask(task: TaskEntity) = dao.delete(task)

    suspend fun findTask(id: Int) = dao.findById(id)
}
```

Room

Criando o viewModel sem Hilt

```
class UserViewModelFactory(private val context: Context) :
ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        val db = AppDatabase.getInstance(context)
        val dao = db.userDao()
        val repo = UserRepository(dao)
        return UserViewModel(repo) as T
    }
}
```

DataStore

DataStore

O Passado e o Futuro

- O Passado: SharedPreferences
  - Durante anos, usámos o SharedPreferences para guardar configurações
  - Problema: A API é síncrona
  - Se o ficheiro de preferências for grande, a leitura/escrita bloqueia a UI Thread
- O Futuro: Jetpack DataStore
  - A solução moderna da Google para armazenar pares Chave-Valor

DataStore

SharedPreferences vs DataStore

SharedPreferences	DataStore
API síncrona	Assíncrono
Baseada em callbacks	Baseado em Coroutines e Flow
Fácil de causar ANR	Seguro
Difícil de integrar com UI reativa	Reativo
API antiga (Java-style)	Compatível com Compose

DataStore

Vantagens

- Assíncrono:
  - Construído totalmente sobre Coroutines (para escrever) e Flow (para ler)
  - Nunca bloqueia a UI
- Seguro:
  - Trata erros de I/O automaticamente e roda em Dispatchers.IO por padrão
- Consistente:
  - Garante atomicidade (a escrita ou acontece totalmente ou não acontece)

DataStore

Preferences DataStore vs Proto DataStore

Preferences DataStore	DataStore
Baseado em chave → valor	Baseado em Protobuf
Não exige schema	Tipado
Mais simples, tipagem não é 100% garantida	Mais seguro
	Mais complexo

DataStore

Dependências - Preference DataStore - libs.version.toml

```
[versions]
datastorePreferences = "1.2.0"

[libraries]
androidx-datastore-preferences = { group = "androidx.datastore",
name = "datastore-preferences", version.ref = "datastorePreferences"
}
```

DataStore

👀 Leitura

```
object PreferencesKeys {
    val IS_DARK_MODE = booleanPreferencesKey("is_dark_mode")
    val USER_AGE = intPreferencesKey("user_age")
}

val darkModeFlow: Flow<Boolean> = context.dataStore.data
    .map { preferences ->
        // Retorna o valor salvo ou 'false' (padrão) se não existir
        preferences[PreferencesKeys.IS_DARK_MODE] ?: false
    }
```

DataStore

✍️ Escrita

```
suspend fun setDarkMode(enabled: Boolean) {
    context.dataStore.edit { preferences ->
        // 'preferences' é mutável aqui dentro
        preferences[PreferencesKeys.IS_DARK_MODE] = enabled
    }
}
```

DataStore

🏗️ Integração na Arquitetura

- Onde o DataStore vive na sua arquitetura MVVM?
  - Data Source:
    - O DataStore deve ser encapsulado
    - Ex: UserPreferencesDataSource
  - Repository:
    - O Repositório expõe os dados limpos para o ViewModel

DataStore

🏗️ Integração na Arquitetura

```
class SettingsRepository(
    private val dataStore: DataStore<Preferences>
) {

    val darkModeFlow: Flow<Boolean> =
        dataStore.data.map { prefs ->
            prefs[DARK_MODE] ?: false
        }

    suspend fun setDarkMode(enabled: Boolean) {
        dataStore.edit {
            it[DARK_MODE] = enabled
        }
    }
}
```

DataStore

🧠 ViewModel

```
class SettingsViewModel(
    private val repository: SettingsRepository
) : ViewModel() {

    val darkMode: StateFlow<Boolean> =
        repository.darkModeFlow
            .stateIn(
                viewModelScope,
                SharingStarted.WhileSubscribed(5000),
                false
            )

    fun toggleDarkMode(value: Boolean) {
        viewModelScope.launch {
            repository.setDarkMode(value)
        }
    }
}
```

DataStore

🎨 Integração com Compose

```
@Composable
fun SettingsScreen(viewModel: SettingsViewModel) {

    val isDarkMode by viewModel.darkMode.collectAsState()

    Switch(
        checked = isDarkMode,
        onCheckedChange = { viewModel.toggleDarkMode(it) }
    )
}
```

DataStore

A Regra do Singleton (Muito Importante!)

- O DataStore mantém uma conexão aberta com o arquivo no disco
  - Jamais crie mais de uma instância do DataStore para o mesmo arquivo
    - Pode corromper os dados
- Solução:
  - Crie a instância como uma extensão de nível superior (Top-Level Extension)
  - Usar Hilt ou Koin para garantir que seja única (Singleton)

DataStore

Criando um Singleton

```
// No nível superior do ficheiro (fora de qualquer classe)
val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(name = "settings")
```

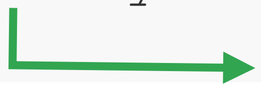
Injeção de dependência

Injeção de dependência

Introdução

- Classes frequentemente precisa referenciar outras classes para funcionar
  - As classes referenciadas são chamada de dependências

```
class UserViewModel {
  private val repository = UserRepository()
}
```



❌ Forte acoplamento

❌ Difícil reutilizar

❌ Difícil testar

❌ Difícil trocar implementação

Injeção de dependência

Introdução

- Existem três maneiras de uma classe obter um objeto de que precisa:
  - A classe constrói a dependência necessária:
    - A classe criar uma instância daquela dependência
  - Obtê-la de algum lugar:
    - Algumas APIs do Android, como os getters de `Context` e `getSystemService()`, funcionam dessa maneira.
  - Recebê-la como parâmetro.
    - As dependências são fornecidas para que a classe seja construída ou são passadas para as funções que delas precisam

Injeção de dependência

Manual

```
class UserViewModel(
    private val repository: UserRepository
) {
    fun loadUsers(): List<String> {
        return repository.getUsers()
    }
}

val repository = UserRepository()
val viewModel = UserViewModel(repository)
```

Criando a dependência

Injeção de dependência

Automatizando

- A injeção manual de dependências também apresenta diversos problemas:
  - Em aplicativos grandes, obter todas as dependências e conectá-las pode exigir uma grande quantidade de código repetitivo
  - Podem existir cenários complexos que se faça necessário manter um grafo de dependência para gerenciar o tempo de vida de cada dependência
- Existem bibliotecas que resolvem esse problema automatizando este processo
  - A biblioteca Hilt que é baseada no Dagger faz isso

Injeção de dependência

Service locator

- É uma alternativa à injeção de dependência
- É um design pattern
  - Diminui o acoplamento entre as classes e suas dependências
- É criada uma classe conhecida como service locator
  - Responsável por criar e armazenar dependências
  - Esta classe fornece as dependências sob demanda

Injeção de dependência

Koin

- É um framework que ajuda você a construir qualquer tipo de aplicação Kotlin e Kotlin Multiplatform
- É desenvolvido pela Kotzilla e colaboradores de código aberto
- Funcionamento básico:
  - Declaração: Define dependências em módulos
  - Resolução: As classes solicitam dependências usando get() ou inject()
  - Comportamento: Atua como um registro central onde se busca serviços

DataStore

Dependências - Preference DataStore - libs.version.toml

```
[versions]
koin = "4.1.1"

[libraries]
io-koin-android = { group = "io.insert-koin", name = "koin-android",
version.ref = "koin"}
io-koin-androidx-compose = { group = "io.insert-koin", name = "koin-
androidx-compose", version.ref = "koin"}
```

DataStore

Dependências - Gradle

```
dependencies {
    implementation(libs.io.koin.android)
    implementation(libs.io.koin.androidx.compose)
}
```

Injeção de dependência

Definindo um módulo Koin

```
val appModule = module {
    single { UserRepository() }
    viewModel { UserViewModel(get()) }
}
```

Indica as dependência da aplicação

Informa ao Koin que queremos apenas uma instância (singleton) da dependência

Injeção de dependência

Inicializando o Koin

```
class MyApp : Application() {
    override fun onCreate() {
        super.onCreate()
        startKoin {
            androidContext(this@MyApp)
            modules(appModule)
        }
    }
}
```

Na classe Application, a função startKoin para injetar o contexto do Android usando a função androidContext

Injeção de dependência

Atualizando o manifest

```
    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher"
        android:supportsRtl="true"
        android:theme="@style/Theme.Todolist"
        android:name=".MyApp">
```

Apontar para classe que herde de Application

Injeção de dependência

Usando o ViewModel em uma @Composable

```
@Composable
fun PostScreen(viewModel: PostViewModel = koinViewModel()) {
    val posts by viewModel.posts. collectAsStateWithLifecycle()
    LazyColumn {
        items(posts) {
            Text(it.title)
        }
    }
}
```

Firebase

Firebase

O que é o Firebase?

- Plataforma Backend-as-a-Service (BaaS) do Google
- Criada para acelerar o desenvolvimento de aplicações
- Remove (ou reduz) a necessidade de backend próprio
- Muito utilizada em apps mobile, especialmente Android
-

Ideia central: o desenvolvedor foca mais no app e menos na infraestrutura

Firebase

Firebase como BaaS

- Oferece:
  - Infraestrutura pronta
  - SDKs para Android, iOS e Web
  - Integração simples com projetos mobile
- Ele cuida de:
  - Autenticação
  - Persistência de dados
  - Sincronização
  - Escalabilidade

Firebase

Principais serviços

🔐 Authentication

🗄️ Firestore (Cloud Firestore)

📦 Realtime Database

☁️ Cloud Functions

📂 Cloud Storage

📊 Analytics

🧪 Crashlytics

Firebase

Quando faz sentido usar Firebase?

- Aplicações pequenas ou médias
- Prototipação rápida
- Times reduzidos
- Apps mobile sem backend dedicado
- Necessidade de tempo real e offline

Firebase

Vantagens

- Setup rápido
- SDKs oficiais bem documentados
- Suporte offline (Firestore)
- Sincronização automática
- Escala sem esforço inicial
- Integração natural com Android

Firebase

Desvantagens

- Forte acoplamento ao fornecedor (vendor lock-in)
- Modelo NoSQL pode ser limitante
- Custos podem crescer com escala
- Regras de segurança exigem atenção
- Menos controle que um backend próprio

Firebase

FireStore - Visão geral

- Banco de dados NoSQL orientado a documentos
- Estrutura baseada em:
  - Collections
  - Documents
- Dados armazenados em formato semelhante a JSON

Firebase

Estrutura do Firestore

```
todos (collection)
└── todoId (document)
    ├── title: "Comprar leite"
    ├── completed: false
    └── updatedAt: timestamp
```

Firebase

FireStore e tempo real

- Firestore permite listeners
- O app recebe atualizações automaticamente
- Ideal para:
  - Chat
  - Listas colaborativas
  - Sincronização entre dispositivos

Firebase

Firestore e suporte offline

- Cache local automático
- Escritas offline
- Sincronização quando a internet retorna

Firebase

Firestore no Android (visão geral)

1. Criar projeto no Firebase Console
2. Adicionar app Android
3. Configurar dependências
4. Inicializar Firestore

Firebase

Dependências - FireStore - libs.version.toml

```
[versions]
firebaseBom = "34.7.0"
googleGmsGoogleServices = "4.4.4"

[libraries]
firebase-bom = { group = "com.google.firebase", name = "firebase-bom",
version.ref = "firebaseBom" }
firebase-firestore = { group = "com.google.firebase", name = "firebase-
firestore" }


[plugins]
google-gms-google-services = { id = "com.google.gms.google-services",
version.ref = "googleGmsGoogleServices" }
```

Firebase

Dependências - Gradle - app/build.gradle.kts

```
plugins {
     alias(libs.plugins.google.gms.google.services)
}

android {
    implementation(platform(libs.firebase.bom))
    implementation(libs.firebase.firestore)

}
```

Firebase

Dependências - Gradle - build.gradle.kts

```
plugins {
     alias(libs.plugins.google.gms.google.services) apply false
}
```

Firebase

Firestore - Lendo dados

```
firestore.collection("todos")
.get()
```

Firebase

Firestore - Inserindo dados

```
firestore.collection(“todos").add(todo)
```

Firebase

Firestore - Salvando dados

```
firestore.collection("todos")
.document(todo.id)
.set(todo)
```

Firebase

Firestore - Salvando dados

```
firestore.collection("todos")
.document(todo.id)
.update(todo)
```

Firebase

Firestore - Removendo dados

```
firestore.collection("todos")
.document(todo.id)
.delete()
```

Firebase

Firestore - Observando dados em tempo real

```
firestore.collection("todos")
.addSnapshotListener { snapshot, _ ->
// dados atualizados
}
```

Firebase

Firestore - Convertendo para Flow

```
fun observeTodos(): Flow<List<Todo>> = callbackFlow {
    val listener = firestore
        .collection("todos")
        .addSnapshotListener { snapshot, error ->
           if (error != null) {
                close(error)
                return@addSnapshotListener
            }
            val todos = snapshot?.documents?.mapNotNull
{ it.toObject(Todo::class.java) } ?: emptyList()
            trySend(todos)
        }
    awaitClose {
        listener.remove()
    }
}
```

Resumo

Resumo

Responsabilidade da data layer

⬜ Centralizar acesso a dados

🔄 Expor dados de forma reativa (Flow)

💾 Combinar local + remoto

🧠 Realizar conversões (DTO ↔ Domain ↔ Entity)

📦 Implementar estratégias de cache

🚫 Desacoplar UI de detalhes técnicos

ViewModel

Componentes essenciais da Data Layer

- 🌐 Remote DataSource (API)
  - Retrofit / Ktor
  - DTOs
  - Serialização
- 💾 Local DataSource (Room)
  - Entities
  - DAO
  - Consultas reativas (Flow)
- 🔄 Repository
  - Une local + remoto
  - Exponibiliza dados de forma reativa
  - Implementa regras de negócio
  - Decide quando usar suspend ou Flow
- 🔀 Mappers
  - Convertem Data → Domain
  - Domain → Data

Referências

- Guide to app architecture
- Documentação oficial do Android - Data Layer
- Documentação oficial do Android - Offline First
- Jetpack Compose: Alternate to Live Data -StateFlow
- Kotlin flows on Android
- Documentação oficial do Android - DataStore
- Prefer Storing Data with Jetpack DataStore

Referências

- DataStore and data migration
- Koin: A Practical Guide to Dependency Injection in Android
- Cloud Firestore

Por hoje é só