# Guia para Desenvolvimento de Aplicação Móvel em Android Studio com Kotlin

Com base no enunciado do projeto e a estrutura da base de dados, este guia vai te ajudar a criar a aplicação móvel. O projeto será organizado em três partes: API, Base de Dados e Aplicação Móvel. Vou fornecer um passo a passo detalhado para a parte da aplicação móvel.

## Passo a Passo de Desenvolvimento

### Passo 1: Configuração Inicial

Criar um novo projeto no Android Studio:

Nome do Projeto: ProjectManagementApp

Linguagem: Kotlin

API mínima: 21 (Android 5.0 Lollipop)

Adicionar dependências ao build.gradle:

### Passo 2: Implementação da Interface de Utilizador

Intro Sliders:

Crie um novo Activity para os sliders introdutórios (IntroActivity).

Use um ViewPager2 para implementar os sliders.

Atividades para Autenticação:

RegisterActivity: Tela para criar uma conta.

LoginActivity: Tela para login.

MainActivity: Tela principal que usará um BottomNavigationView para navegação entre fragmentos.

Fragments para Funcionalidades Principais:

ProfileFragment: Gestão de perfil.

ProjectsFragment: Listagem e gestão de projetos.

TasksFragment: Listagem e gestão de tarefas.

### Passo 3: Implementação da API

Configuração do Retrofit:

Crie uma classe ApiService para definir os endpoints.

Crie um Singleton RetrofitClient para instanciar o Retrofit.

interface ApiService {

@POST("login")

suspend fun login(@Body loginRequest: LoginRequest): LoginResponse

@GET("projects")

suspend fun getProjects(@Header("Authorization") token: String): List<Project>

// Adicione outros endpoints conforme necessário

}

object RetrofitClient {

private const val BASE\_URL = "https://api.example.com/"

private val retrofit by lazy {

Retrofit.Builder()

.baseUrl(BASE\_URL)

.addConverterFactory(GsonConverterFactory.create())

.build()

}

val apiService: ApiService by lazy {

retrofit.create(ApiService::class.java)

}

}

Modelo de Dados e Requisições:

Crie classes de modelo para mapear os dados JSON da API.

data class LoginRequest(val email: String, val password: String)

data class LoginResponse(val token: String)

data class Project(val idProject: Int, val nameProject: String, val startDateP: String, val endDateP: String)

### Passo 4: Implementação do Banco de Dados Local

Configuração do Room:

Crie entidades e DAOs.

kotlin

Copiar código

@Entity

data class User(

@PrimaryKey val idUser: Int,

val email: String,

val photo: String?,

val password: String,

val username: String,

val name: String,

val lastLogin: String?

)

@Dao

interface UserDao {

@Query("SELECT \* FROM User WHERE idUser = :id")

suspend fun getUserById(id: Int): User?

@Insert(onConflict = OnConflictStrategy.REPLACE)

suspend fun insertUser(user: User)

}

@Database(entities = [User::class], version = 1)

abstract class AppDatabase : RoomDatabase() {

abstract fun userDao(): UserDao

companion object {

@Volatile private var instance: AppDatabase? = null

fun getDatabase(context: Context): AppDatabase =

instance ?: synchronized(this) { instance ?: buildDatabase(context).also { instance = it } }

private fun buildDatabase(context: Context) =

Room.databaseBuilder(context, AppDatabase::class.java, "project\_management.db")

.build()

}

}

Repository para Gestão de Dados:

class UserRepository(private val userDao: UserDao, private val apiService: ApiService) {

suspend fun login(email: String, password: String): LoginResponse {

return apiService.login(LoginRequest(email, password))

}

suspend fun getUserById(id: Int): User? {

return userDao.getUserById(id)

}

suspend fun insertUser(user: User) {

userDao.insertUser(user)

}

}

### Passo 5: Implementação dos ViewModels

Exemplo de ViewModel para Login:

class LoginViewModel(private val userRepository: UserRepository) : ViewModel() {

private val \_loginResult = MutableLiveData<LoginResponse>()

val loginResult: LiveData<LoginResponse> = \_loginResult

fun login(email: String, password: String) {

viewModelScope.launch {

try {

val response = userRepository.login(email, password)

\_loginResult.value = response

} catch (e: Exception) {

// Tratar erro

}

}

}

}

Factory para ViewModel:

class LoginViewModelFactory(private val userRepository: UserRepository) : ViewModelProvider.Factory {

override fun <T : ViewModel?> create(modelClass: Class<T>): T {

if (modelClass.isAssignableFrom(LoginViewModel::class.java)) {

@Suppress("UNCHECKED\_CAST")

return LoginViewModel(userRepository) as T

}

throw IllegalArgumentException("Unknown ViewModel class")

}

}

### Passo 6: Implementação da Lógica de Sincronização

Utilização do WorkManager para Sincronização:

class SyncWorker(appContext: Context, workerParams: WorkerParameters) : CoroutineWorker(appContext, workerParams) {

override suspend fun doWork(): Result {

// Lógica de sincronização com a API

return Result.success()

}

}

// Agendar o WorkManager

val syncWorkRequest = OneTimeWorkRequestBuilder<SyncWorker>()

.build()

WorkManager.getInstance(context).enqueue(syncWorkRequest)

Desenvolvimento Contínuo e Testes

Testes: Implemente testes unitários para ViewModels e lógica de dados.

UI/UX: Garanta que a interface seja intuitiva e responsiva.

Segurança: Implemente medidas de segurança, como criptografia de dados sensíveis e uso de HTTPS.

• Intro sliders; ✅

• Criar conta; ✅

• Iniciar sessão; ✅

• Gestão de perfil: Nome, username. email, fotografia, password. ✅

• Pelo menos três tipos de perfil: Administrador, Gestor de Projeto e Utilizador✅

• **Administrador**:

o O administrador deverá criar, editar e remover projetos.

o O administrador poderá criar, alterar e remover os perfis dos utilizadores e gestor de projeto do sistema.

o O administrador associa um gestor de projeto ao projeto criado.

o O administrador pode exportar estatísticas, por utilizador, por projeto ou por tarefa.

• **Gestor de Projeto:**

o O gestor de projeto deverá associar tarefas para cada projeto.

o O gestor de projeto deverá associar utilizadores a projetos e respetivas tarefas. Considere que um utilizador pode ter mais que uma tarefa do mesmo projeto e uma tarefa pode ser realizada por um ou mais utilizadores.

o O gestor de projeto poderá visualizar as tarefas já concluídas e por concluir de determinado projeto.

o O gestor de projeto dá como concluído o projeto e avalia a performance de cada utilizador.

o O gestor de projeto pode exportar estatísticas, por utilizador, por projeto ou por tarefa.

• **Utilizador:**

o O utilizador associado a determinada tarefa em determinado projeto deve:

▪ Registar a data, local, taxa de conclusão e tempo dispensado na realização de determinada tarefa.

▪ Associar observações a tarefas com associação de fotografias, caso seja necessário.

▪ Dar a tarefa como concluída.

▪ Visualizar listagem de tarefas por realizar.

▪ Visualizar histórico de tarefas já concluídas.