# Guía de ejercicios - Consumo de API REST (IV)

 ¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

## ¿En qué consiste esta guía?

La siguiente guía de estudio tiene como objetivo practicar y ejercitar los contenidos que hemos visto en clase.

**¡Vamos con todo!**

## Tabla de contenidos



**¡Comencemos!**

# Actividad guiada: MVVM, Retrofit y StateFlow

En la siguiente Actividad Guiada usaremos MVVM, Retrofit y StateFlow para mostrar cómo se puede hacer un request a la API de GitHub y mostrar los repositorios de un usuario, ya sabemos con trabajar con MVVM y ViewBinding, por lo que no explicaremos esos detalles.

*Empecemos!*

Lo primero que debemos hacer es agregar las siguientes dependencias, si bien `com.squareup.okhttp3:logging-interceptor` no es necesario, es recomendable agregarla ya que es de gran ayuda al momento de debugging.

Agrega la siguiente dependencia a su archivo build.gradle:

```
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
implementation 'com.squareup.okhttp3:logging-interceptor:4.9.0'
```

Define una data class en la cual se almacenarán los datos provenientes desde la API de GitHub, ya que la clase almacena repositorios, la llamaremos `RepositoryResponse`

```
data class RepositoryResponse(
    val name: String,
    val description: String?,
    val html_url: String?,
)
```

Crea una nueva interfaz que defina los puntos de conexión de la API mediante anotaciones Retrofit. Por ejemplo, para obtener una lista de repositorios para un usuario, crea una función como esta:

```
interface GitHubService {
    @GET("/users/{username}/repos")
    suspend fun getRepositories(
        @Path("username") username: String
    ): Response<List<RepositoryResponse>>
}
```

Necesitamos crear una clase o un objeto el cual nos permita llamar al cliente de Retrofit cada vez que lo necesitemos.

```
object RetrofitClient {
    fun getInstance(): Retrofit {
        val interceptor =
HttpLoggingInterceptor().setLevel(HttpLoggingInterceptor.Level.BODY)
        val httpClient =
OkHttpClient.Builder().addInterceptor(interceptor).build()
        return Retrofit.Builder().baseUrl(BASE_URL)
            .addConverterFactory(GsonConverterFactory.create())
            .client(httpClient)
            .build()
    }
}

const val BASE_URL = "https://api.github.com/"
```

Como puedes ver, definimos un Interceptor el cual nos permitirá ver los request en el Logcat, también definimos la url de la API en una constante. Puedes crear un archivo de constantes para almacenar ese valor.

Ya que es posible que cuando hagamos un request a una API responda, por ejemplo, con un código 200, que sería el o los objetos que esperamos, es probable que recibamos otro tipo de respuestas. Como ya vimos anteriormente, debemos ser capaces de manejar esas respuestas, para esto podemos crear una sealed class, la cual nos permitirá manejar esas respuestas con diferentes estados.

```
sealed class ServiceResponse<T> {
    data class Loading<T>(val isLoading: Boolean) : ServiceResponse<T>()
    data class Success<T>(val data: T) : ServiceResponse<T>()
    data class Error<T>(val error: String) : ServiceResponse<T>()
}
```

También debemos crear un Mapper que nos ayude a transformar los objetos provenientes de la API a nuestra clase DTO:

```
fun toRepositoryDto(response: RepositoryResponse): RepositoryDto =
RepositoryDto(
    name = response.name,
    description = response.description ?: "",
    htmlUrl = response.html_url ?: ""
```

```
)

fun repositoryResponseListToRepositoryDtoList(list:
List<RepositoryResponse>): List<RepositoryDto> =
    list.map(::toRepositoryDto)
```

A continuación, crea una nueva clase que manejará las operaciones de datos para tu aplicación. Esta clase debe usar Retrofit para realizar llamadas a la API y devolver los datos al ViewModel. Por ejemplo:

```
class GitHubRepository {
    suspend fun getRepositories(username: String):
Flow<ServiceResponse<List<RepositoryDto>?>> {

        val data:
MutableStateFlow<ServiceResponse<List<RepositoryDto>?>> =
            MutableStateFlow(ServiceResponse.Loading(true))

        val service =
RetrofitClient.getInstance().create(GitHubService::class.java)
        val response = service.getRepositories(username)

        when {
            response.isSuccessful -> {
                response.body()?.let {
                    data.value =

ServiceResponse.Success(repositoryResponseListToRepositoryDtoList(it))
                }
            }

        }
        return flowOf(data.value)
    }
}
```

Crea una nueva clase `ViewModel` que manejará la lógica de presentación de tu aplicación. Esta clase debe usar la clase `Repository` para obtener datos y exponerlos a la Vista usando StateFlow. Por ejemplo:

```
class GitHubViewModel : ViewModel() {
```

```
    private val repository: GitHubRepository = GitHubRepository()
    private val _getRepositories:
MutableStateFlow<ServiceResponse<List<RepositoryDto>?>> =
        MutableStateFlow(ServiceResponse.Loading(true))
    private val repositoryList:
StateFlow<ServiceResponse<List<RepositoryDto>?>> =
        _getRepositories.asStateFlow()

    suspend fun getRepositories(userName: String):
StateFlow<ServiceResponse<List<RepositoryDto>?>> {
        repository.getRepositories(userName).collectLatest { response ->
            when (response) {
                is ServiceResponse.Loading ->
                    _getRepositories.value =
ServiceResponse.Loading(true)
                is ServiceResponse.Success ->
                    _getRepositories.value =
ServiceResponse.Success(response.data)
                is ServiceResponse.Error ->
                    _getRepositories.value =
ServiceResponse.Error(response.error)
            }
        }
        return repositoryList
    }
}
```

Ahora en tu Activity o Fragment mostrarás los datos al usuario. Esta clase debe usar ViewModel para obtener los datos y actualizar la interfaz de usuario, ya que recibiremos una lista de repositorios, podemos mostrarlos en un RecyclerView. Para esto debemos crear un Adapter, y ya que gracias a la sealed class creada anteriormente podemos manejar distintos estados, crearemos un par de funciones las cuales mostraran o ocultaran vistas dependiendo de esos estados, por ejemplo:

```
class MainActivity : AppCompatActivity() {

    private val viewModel: GitHubViewModel by viewModels()
    private lateinit var binding: ActivityMainBinding

    private lateinit var adapter: MainAdapter
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        intiAdapter()
        initSwipeToRefresh()
        fetchData("octocat")
    }

    private fun fetchData(userName: String) {
        lifecycleScope.launchWhenCreated {
            viewModel.getRepositories(userName).collectLatest {
                when (it) {
                    is ServiceResponse.Loading -> loadingState()
                    is ServiceResponse.Success -> submitData(it.data as
List<RepositoryDto>)
                    is ServiceResponse.Error -> errorState()
                }
            }
        }
    }

    private fun initSwipeToRefresh() {
        binding.refresh.setOnRefreshListener {
            fetchData("octocat")
            binding.refresh.isRefreshing = false
        }
    }

    private fun submitData(repositories: List<RepositoryDto>) {
        adapter.list = repositories
        adapter.notifyDataSetChanged()
    }

    private fun intiAdapter() {
        adapter = MainAdapter(emptyList())
        binding.rvRepository.layoutManager = LinearLayoutManager(this)
        binding.rvRepository.adapter = adapter
        binding.pbLoading.visibility = View.GONE
    }

    private fun loadingState() {
        binding.pbLoading.visibility = View.VISIBLE
```

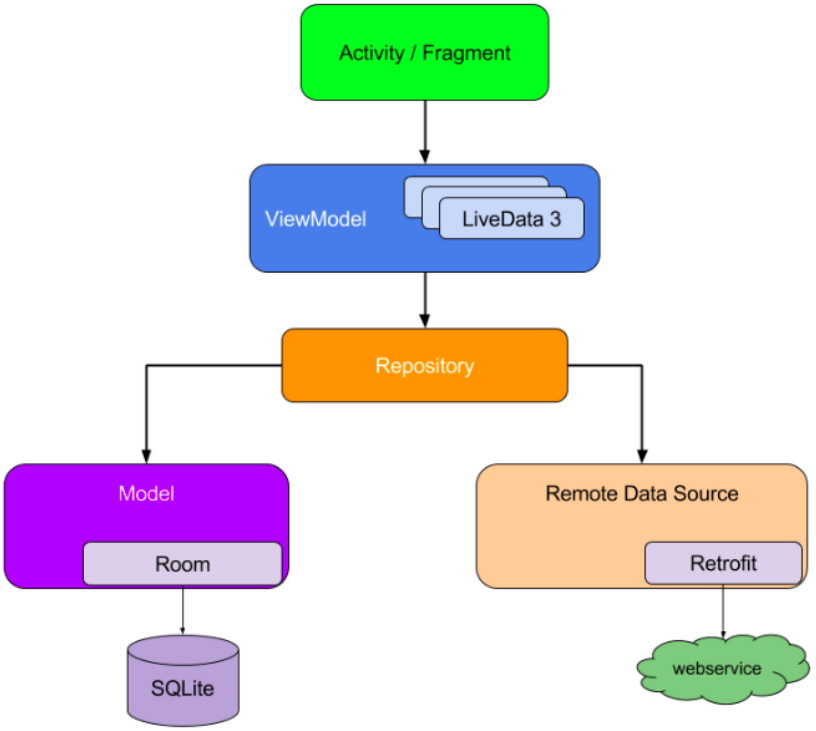
```
        binding.tvError.visibility = View.GONE
        binding.rvRepository.visibility = View.GONE
    }

    private fun errorState() {
        binding.tvError.visibility = View.VISIBLE
        binding.pbLoading.visibility = View.GONE
        binding.rvRepository.visibility = View.GONE
    }
}
```

Y con esto terminamos, no estamos haciendo modificable el nombre del usuario, para que así todos tengan el mismo resultado, para eso estamos usando el usuario "`octocat`".

## Actividad guiada: MVVM, Retrofit, Repository Pattern, Room y StateFlow

En la siguiente actividad guiada usaremos MVVM, Retrofit, Repository Pattern, Room y StateFlow para mostrar cómo se puede hacer un request a la API de GitHub, luego guardar los datos en una base de datos con Room y luego mostrar los datos al usuario, para esto continuaremos con lo desarrollado en la Actividad anterior.

Primero debemos recordar cual es la arquitectura sugerida para Android:




En nuestro caso el lado derecho del diagrama ya lo hemos desarrollado, y ya que estamos usando Repository Pattern, para agregar soporte para Room, solo debemos hacer unas pequeñas modificaciones a la clase Repository.

Lo primero que debemos hacer es crear una data class entity la cual corresponde a la tabla en la que se guardan los datos

```
@Entity(tableName = "repositories")
```

```
data class RepositoryEntity(
    @PrimaryKey(autoGenerate = true)
    val id: Int,
    val name: String,
    val description: String,
    val htmlUrl: String
)
```

Luego necesitamos crear la interfaz DAO la cual almacena las consultas a nuestra tabla "`repositories`"

```
@Dao
interface RepositoryDao {

    @Query("SELECT * FROM repositories")
    fun getRepositories(): Flow<List<RepositoryEntity>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAll(repositories: List<RepositoryEntity>)

    @Query("DELETE FROM repositories")
    fun deleteAll()
}
```

Ahora necesitamos crear una clase que será la base de datos, esta debe extender de Room y ser abstracta.

```
@Database(entities = [RepositoryEntity::class], version = 1)
abstract class RepositoryDatabase : RoomDatabase() {

    abstract fun repositoryDao(): RepositoryDao

    companion object {
        @Volatile
        private var INSTANCE: RepositoryDatabase? = null

        fun getDatabase(context: Context): RepositoryDatabase {
            return INSTANCE ?: synchronized(this) {
                val instance = Room.databaseBuilder(
                    context.applicationContext,
RepositoryDatabase::class.java, "repo_database"
                ).build()
```

```
                INSTANCE = instance
                instance
            }
        }
    }
}
```

Ya que vamos a recibir un respuesta de una API y vamos a guardar esos datos en una base de datos local para luego mostrarlo al usuario, necesitamos actualizar nuestro mapper para permitirle trabajar con la clase entity.

```
fun responseToDto(response: RepositoryResponse): RepositoryDto =
RepositoryDto(
    name = response.name,
    description = response.description ?: "",
    htmlUrl = response.html_url ?: ""
)

fun responseToEntity(response: RepositoryResponse): RepositoryEntity =
RepositoryEntity(
    id = 0,
    name = response.name,
    description = response.description ?: "",
    htmlUrl = response.html_url ?: "",
)

fun entityToDto(entity: RepositoryEntity): RepositoryDto =
RepositoryDto(
    name = entity.name, description = entity.description, htmlUrl =
entity.htmlUrl
)

fun responseListToDtoList(list: List<RepositoryResponse>):
List<RepositoryDto> =
    list.map(::responseToDto)

fun entityListToDtoList(list: List<RepositoryEntity>):
List<RepositoryDto> =
    list.map(::entityToDto)

fun responseListToEntityList(list: List<RepositoryResponse>):
List<RepositoryEntity> =
    list.map(::responseToEntity)
```

Después de estos cambios, solo nos queda actualizar la clase repository y con eso hemos terminado ¡Ahora nuestra app es capaz de obtener datos desde internet y almacenarlos localmente!

## Preguntas de proceso

**Reflexiona:**

- ¿Qué ha sido lo más difícil de entender hasta el momento?
- ¿Hay algún ejercicio que se te haya hecho difícil de entender?
- Finalmente hemos unido todas las partes, MVVM, Repository Pattern, Room, Retrofit y StateFlow, ¿hay alguna de las partes mencionadas anteriormente que sea difícil de entender?
- De las partes mencionadas anteriormente, ¿hay alguna que creas que sea de mayor utilidad?

## Preguntas de cierre

- ¿Por qué es necesario usar un mapper?
- ¿Por que es necesario pasar el "`context`" a la clase Repository
- ¿Qué pasa si quisiera agregar otra tabla, puedo seguir usando la misma base de datos?
- ¿Qué pasa si quisiera agregar otra API, puedo seguir usando la misma clase de Retrofit?

## Referencias bibliográficas


- Android Repository Pattern: https://developer.android.com/codelabs/basic-android-kotlin-training-repository-pattern#0
- Imagen de Android Repository Pattern: https://developer.android.com/static/codelabs/basic-android-kotlin-training-repository-pattern/img/69021c8142d29198_1920.png