Guía de ejercicios - Consumo de API REST (II)

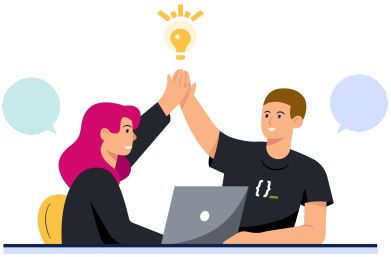
¡Hola! Te damos la bienvenida a esta nueva guía de estudio.

¿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: HTTP request con Retrofit (QueryParams, NetworkResponse)

En esta Actividad Guiada usaremos:

- Retrofit,
- NetworkResponse (opcional),
- MVVM,
- Repository Pattern
- API de GitHub:

¡Empecemos!

Agrega las dependencias Retrofit y `NetworkResponse` a tu proyecto:

```
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-gson:2.9.0"
implementation "com.github.skydoves:network-response:3.0.3"
```

Define su interfaz API de GitHub con Retrofit:

```
interface GithubApiService {
    @GET("users/{username}")
    suspend fun getUser(@Path("username") username: String):
Response<UserResponse>
}
```

Define la data class

```
data class UserResponse(
    val id: Int,
    val login: String
)
```

Define una clase de Repositorio que será responsable de realizar la solicitud de API:

```
class GithubRepository(private val apiService: GithubApiService) {
    private val _apiState = MutableStateFlow<ApiState>(ApiState.Loading)
    val apiState: StateFlow<ApiState> = _apiState

    suspend fun getUser(username: String) {
        _apiState.value = ApiState.Loading
        when (val response = apiService.getUser(username)) {
            is NetworkResponse.Success -> {
                _apiState.value = ApiState.Success(response.body)
            }
            is NetworkResponse.ApiError -> {
                _apiState.value = ApiState.Error("Failed to get user:
${response.body?.message}")
            }
            is NetworkResponse.NetworkError -> {
                _apiState.value = ApiState.Error("Network error
occurred")
            }
            is NetworkResponse.UnknownError -> {
                _apiState.value = ApiState.Error("Unknown error
occurred")
            }
        }
    }
}
```

Define tu clase ViewModel que será responsable de obtener los datos del repositorio y proporcionarlos a la Vista:

```
class GithubViewModel(private val repository: GithubRepository) :
ViewModel() {
    private val _apiState = repository.apiState
    val apiState: StateFlow<ApiState> = _apiState

    fun getUser(username: String) {
        viewModelScope.launch {
            repository.getUser(username)
        }
    }
}
```

En tu Fragment, crea una instancia de ViewModel y observa su `apiState` usando `StateFlow`.
Puedes definirlo así:

```
class GithubFragment : Fragment() {
    private val viewModel: GithubViewModel by viewModels {
        GithubViewModelFactory(GithubRepository(Retrofit.Builder()
            .baseUrl("https://api.github.com/")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
            .create(GithubApiService::class.java)))
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        return inflater.inflate(R.layout.fragment_github, container,
false)
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?)
{
        super.onViewCreated(view, savedInstanceState)

        viewModel.getUser("octocat")

        lifecycleScope.launch {
            viewModel.apiState.collect { apiState ->
                when (apiState) {
                    is ApiState.Loading -> {
                        // Show a loading spinner or other feedback to
the user
                    }
                    is ApiState.Success -> {
                        val user = apiState.data
                        // Update the UI with the user data
                    }
                    is ApiState.Error -> {
                        val message = apiState.message
                        // Handle the error, e.g. display an error
message to the user
                    }
```

```
                }
            }
        }
    }
}
```

¡Eso es todo! En este caso no nos estamos enfocando en cómo mostrar esos datos al usuario, lo importante aquí es recordar que cada vez que quieras hacer un HTTP Request, básicamente, los pasos son siempre los mismos:

- Crear una data class que almacenará la respuesta del request
- Crear una interfaz la cual contendrá la función y la anotación de Retrofit, ejemplo, si estamos tratando de obtener una lista de usuarios "`getUsers()`" entonces la anotación será `GET`
- Crea una clase Repository en la cual se harán algunas operaciones como por ejemplo mapear los datos provenientes desde la API a una clase DTO, especialmente si en algún momento planeas usar base de datos local.

¡Manos a la obra! - HTTP request con Retrofit , manejo de errores

Retrofit puede manejar los errores HTTP usando el mecanismo de manejo de errores integrado. Cuando una respuesta HTTP tiene un código de estado de error (es decir, códigos de estado 4xx o 5xx), Retrofit generará una excepción de tipo HttpException. Puede capturar esta excepción y manejarla apropiadamente.

Creamos la interfaz:

```
interface ApiService {
    @GET("/user")
    suspend fun getUser(): User
}
```

La data class `User` puede ser algo simple como:

```
data class User(val id: Int, val name: String)
```

Creamos la clase Repositorio:

```
class UserRepository(private val apiService: ApiService) {
    suspend fun getUser(): Result<User> {
        return try {
            val user = apiService.getUser()
            Result.Success(user)
        } catch (e: Exception) {
            when (e) {
                is HttpException -> {
                    val errorBody = e.response()?.errorBody()?.string()
                    Result.Error("HTTP error: $errorBody")
                }
                else -> {
                    Result.Error("Network error: ${e.message}")
                }
            }
        }
    }
}
```

Creamos una sealed class que nos permita manejar los distintos estados:

```
sealed class Result<out T> {
    data class Success<T>(val data: T) : Result<T>()
    data class Error(val message: String) : Result<Nothing>()
}
```

Explicación:

En este ejemplo, tenemos una interfaz `ApiService` que define un único método `getUser()`. Este método devuelve un objeto "`User`". También tenemos una clase `UserRepository` que toma una instancia de `ApiService` en su constructor. El método `getUser()` en `UserRepository` llama a `apiService.getUser()` y devuelve un objeto `Result<User>`.

En el bloque try de `getUser()`, intentamos realizar la llamada a la API mediante `apiService.getUser()`.

- Si la llamada a la API es exitosa, devolvemos un objeto `Result.Success` que contiene el objeto Usuario.
- Si la llamada a la API arroja una excepción, la atrapamos en el bloque catch.
- Si la excepción es una instancia de HttpException, extraemos el mensaje de error del cuerpo del error de la respuesta usando `response()?.errorBody()?.string()` y devolvemos un objeto `Result.Error` que contiene el mensaje de error.
- Si la excepción no es una instancia de `HttpException`, asumimos que es un error de red y devolvemos un objeto `Result.Error` con un mensaje de error genérico.

Para probar este código, puedes usar un marco de prueba como JUnit o Mockito. Aquí hay un ejemplo de cómo puede probar la clase `UserRepository`:

```
class UserRepositoryTest {
    private lateinit var apiService: ApiService
    private lateinit var userRepository: UserRepository

    @Before
    fun setUp() {
        apiService = mock(ApiService::class.java)
        userRepository = UserRepository(apiService)
    }

    @Test
    fun `getUser() should return success when API call is successful`()
```

```
= runBlocking {
        val expectedUser = User(1, "John Doe")
        `when`(apiService.getUser()).thenReturn(expectedUser)

        val result = userRepository.getUser()

        assertThat(result).isEqualTo(Result.Success(expectedUser))
    }

    @Test
    fun `getUser() should return error when API call returns HTTP
error`() = runBlocking {
        val errorBody = "Not found"
        val response = Response.error<User>(404,
ResponseBody.create(null, errorBody))
        `when`(apiService.getUser()).thenThrow(HttpException(response))

        val result = userRepository.getUser()

        assertThat(result).isEqualTo(Result.Error("HTTP error:
$errorBody"))
    }

    @Test
    fun `getUser() should return error when API call returns network
error`() = runBlocking {
        val errorMessage = "Network error"

`when`(apiService.getUser()).thenThrow(IOException(errorMessage))

        val result = userRepository.getUser()

        assertThat(result).isEqualTo(Result.Error("HTTP error:
$errorBody"))
```

En este caso hemos usado Unit Test para comprobar el correcto funcionamiento de nuestro código.

Recuerda: manejar errores de esta forma es útil cuando quieres tener más control y también cuando no necesitas ser demasiado específico con el usuario, es decir, no es relevante si por ejemplo al hacer un request recibes un error 401 o 403, el usuario sabrá que ocurrió un error sin importar qué tipo.

Preguntas de proceso

Reflexiona:

- ¿Existió algún ejercicio que se te hizo difícil de comprender?
- ¿Qué opinas del uso de HTTP request con Retrofit hasta ahora?
- ¿Puedes nombrar al menos 2 momentos o situaciones en los cuales el contenido aprendido te sea de utilidad?

Preguntas de cierre

- ¿Qué es un QueryParams?
- ¿Cuál es la ventaja de usar algo como `NetworkResponse`?
- En varios ejercicios y ejemplos hemos usado `GsonConverterFactory`, recuerdas lo que es?, explica con tus palabras

Referencias bibliográficas

- Documentacion NetworResponse: https://github.com/skydoves/retrofit-adapters