Instituto Tecnológico Superior de Purísima del Rincón

P2_Consumo API

MATERIA:
Desarrollo de aplicaciones para dispositivos móviles

ALUMNOS:
Juan Antonio Pérez Cabrera

DOCENTE:
Jair Emmanuel Ramírez Flores

FECHA: 7 de noviembre de 2025 **CIUDAD:** Purísima del Rincón, Gto.

Introducción

El desarrollo de aplicaciones móviles modernas requiere integrar datos dinámicos y actualizados que enriquezcan la experiencia del usuario. En este proyecto, se implementó el consumo de una API pública utilizando Retrofit en un entorno Android con Kotlin y Jetpack Compose. El objetivo fue construir una interfaz funcional que muestre información de personajes de la saga Harry Potter, obtenida en tiempo real desde un servicio web. A través de la arquitectura MVVM (Model-View-ViewModel) permitiendo separar y facilitar el mantenimiento del código. Este ejercicio fortaleció habilidades técnicas en programación móvil,

Desarrollo

1. Creamos un nuevo proyecto con empty activity
2. Creamos un archivo kotlin “Character”, donde tendremos una data class con los atributos seleccionados de la API de los personajes

```
data class CharacterResponse(
    val results: List<Character>
)

data class Character(
    val id: String,
    val image: String,
    val name: String,
    val house: String,
    val gender: String,
    val actor: String,
    val alive: Boolean
)
```

3. Creamos un archivo tipo “Object” llamado “RetrofitClient”, el cual contiene la URL de la API publica que usaremos para extraer los personajes

```
object RetrofitClient {
    const val BASE_URL = "https://hp-api.onrender.com/api/"
//indicar a que API se va usar
//
// https://hp-api.onrender.com/api/

    //implementacion del servicio
    val apiService: ApiService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL) //url de la api

.addConverterFactory(GsonConverterFactory.create())
//castear json a texto
            .build()
            .create(ApiService::class.java)
    }
}
```

4. Declaramos una clase “CharacterRepository” la cual es una función suspendida que obtiene los personajes de la lista y retorna por medio de retrofit.

```
class CharacterRepository {
    suspend fun getCharacters(): List<Character> {
        return RetrofitClient.apiService.getCharacters()
    }
}
```

5. Creamos una clase Kotlin donde tendremos una clase viwe model,, aquí declaramos una variable como lista vacía y una función que extraerá los personajes del repositorio previamente creado

```
class CharacterViewModel: ViewModel() {
    private val repository = CharacterRepository()

    private val _characters =
MutableStateFlow<List<Character>>(emptyList())
    val character: StateFlow<List<Character>> = _characters

    init {
        fetchCharacter()
    }


    private fun fetchCharacter(){
        viewModelScope.launch {
            try {
                _characters.value = repository.getCharacters()
            }catch (e: Exception){
                _characters.value = emptyList()
            }
        }
    }

}
```

6. Creamos un archivo tipo “interface” donde con la Liberia retrofit obtendremos los caracteres de la api con una función suspendida a una lista de personajes

```
interface ApiService {
    @GET("characters")
    suspend fun getCharacters(): List<Character>

}
```

7. Para la pantalla creamos un archivo kotlin con una función composable que recibirá el parámetro de view model, agregamos el titulo de la pagina y con una LazyColumn mostraremos los ítems.

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun CharacterListScreen(viewModel: CharacterViewModel) {
    val character = viewModel.character.collectAsState()

    Scaffold(
        topBar = {
            TopAppBar(title = { Text("Harry Potter") })
        }
    ) { padding ->
        LazyColumn(
            contentPadding = padding,
            modifier = Modifier.fillMaxSize()
        ) {
            items(character.value) { character ->
                CharacterItem(character)
            }

        }
    }
}
```

8. Dentro del mismo archivo kotlin llamado “CharacterListScreen” creamos otra función composable que recibirá el personaje, esta función tendrá una Card y dentro un Row dentro del cual tendremos la imagen del personaje y sus datos o atributos, los cuales deben coincidir con los declarados en la data class.

```
@Composable
fun CharacterItem(character: Character) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp),
        elevation = CardDefaults.cardElevation(4.dp)
    ) {
        Row(
            modifier = Modifier.padding(8.dp)
        ) {
            Image(
                painter = rememberAsyncImagePainter(model =
character.image),
                contentDescription = character.name,
                modifier = Modifier
                    .size(80.dp)
                    .padding(8.dp)
            )
            Column {
                Text(text = character.name, style =
MaterialTheme.typography.titleLarge)
                Text(text = "Gender: ${character.gender}")
                Text(text = "House: ${character.house}")
                Text(text = "Alive: ${character.alive}")
                Text(text = "Actor: ${character.actor}")
            }

        }
    }
}
```

9. En el MainActivity mandamos a llamar la pantalla de CharacterListScreen

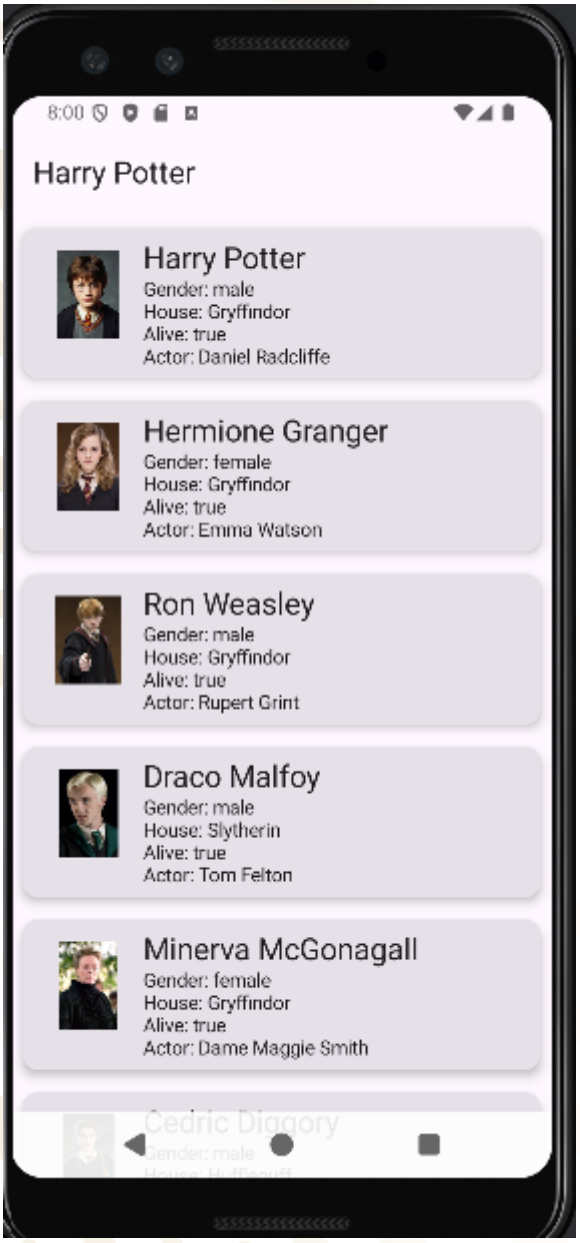
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {

            val characterViewModel: CharacterViewModel =
viewModel()
            CharacterListScreen(viewModel =
characterViewModel)

        }
    }
}
```

Resultados

Vista de la app mostrando los personajes con los atributos especificados en el data class



Conclusiones

La implementación del consumo de una API en una aplicación móvil permitió comprender de manera práctica cómo interactuar con servicios externos, estructurar datos mediante clases modelo y presentar información de forma atractiva usando Jetpack Compose. El uso de Retrofit facilitó la conexión con la API, mientras que la arquitectura MVVM aseguró una gestión ordenada del estado y la lógica de negocio. Esta práctica demostró la importancia de integrar herramientas modernas. Además, refuerza la capacidad de adaptar soluciones a necesidades reales del entorno digital actual, como el consumir información de API's publicas o información alojada en la web.