Implementación del Consumo de APIs Externas - Fake Store API

Resumen General

La aplicación LevelUp Kotlin implementa un sistema híbrido que consume dos tipos de APIs:

1. **API Backend Propio**: Para gestión de usuarios, productos, categorías y plataformas
2. **Fake Store API Externa**: Para gestión de carritos de compra

Arquitectura de APIs

1. Configuración del Cliente HTTP

```
// ApiClient.kt
object ApiClient {
    private const val BASE_URL = "http://levelup-back-env.eba-277ppcgy.us-
east-1.elasticbeanstalk.com/"

    private val json = Json {
        ignoreUnknownKeys = true
        coerceInputValues = true
    }

    private val loggingInterceptor = HttpLoggingInterceptor().apply {
        level = HttpLoggingInterceptor.Level.BODY
    }

    private val okHttpClient = OkHttpClient.Builder()
        .addInterceptor(loggingInterceptor)
        .connectTimeout(30, TimeUnit.SECONDS)
        .readTimeout(30, TimeUnit.SECONDS)
        .writeTimeout(30, TimeUnit.SECONDS)
        .build()

    private val retrofit = Retrofit.Builder()
        .baseUrl(BASE_URL)
        .client(okHttpClient)

.addConverterFactory(json.asConverterFactory("application/json".toMediaType
()))
        .build()

    val apiService: ApiService = retrofit.create(ApiService::class.java)
}
```

2. Definición de Endpoints

```
// ApiService.kt
interface ApiService {

    // Backend Propio - Usuarios
    @GET("api/v1/users")
    suspend fun getUsers(@Header("Authorization") token: String):
Response<List<UserModel>>

    @POST("api/v1/users")
    suspend fun createUser(
        @Header("Authorization") token: String,
        @Body user: RegisterRequest
    ): Response<UserCreateResponse>

    // API Externa - Fake Store API para Carritos
    @GET("https://fakestoreapi.com/carts")
    suspend fun getAllCarts(): Response<List<Carrito>>

    @GET("https://fakestoreapi.com/carts/{id}")
    suspend fun getCartById(@Path("id") id: Long): Response<Carrito>

    @POST("https://fakestoreapi.com/carts")
    suspend fun createCart(@Body cart: Carrito): Response<Carrito>

    @PUT("https://fakestoreapi.com/carts/{id}")
    suspend fun updateCart(@Path("id") id: Long, @Body cart: Carrito):
Response<Carrito>

    @DELETE("https://fakestoreapi.com/carts/{id}")
    suspend fun deleteCart(@Path("id") id: Long): Response<Unit>
}
```

Modelos de Datos para Fake Store API

Carrito y Productos

```
// Carrito.kt
@Serializable
data class CartProduct(
    val productId: Long,
    val quantity: Int
)

@Serializable
data class Carrito(
    val id: Long? = null,
    val userId: Long,
    val date: String,
    val products: List<CartProduct> = emptyList()
)
```

Implementación del Repository

Gestión de Errores y Llamadas API

```
// LevelUpRepository.kt
class LevelUpRepository(context: Context? = null, private val apiService:
ApiService = ApiClient.apiService) {

    // Helper function para manejar todas las llamadas API
    private suspend fun <T> apiCall(call: suspend () -> Response<T>):
ApiResult<T> {
        return try {
            val response = call()
            if (response.isSuccessful) {
                response.body()?.let {
                    ApiResult.Success(it)
                } ?: ApiResult.Error(Exception("Empty response body"))
            } else {
                val errorBody = response.errorBody()?.string() ?: "Unknown
error"
                ApiResult.Error(Exception("API Error: ${response.code()}
${response.message()} - $errorBody"))
            }
        } catch (e: Exception) {
            ApiResult.Error(e)
        }
    }

    // Métodos para consumir Fake Store API
    suspend fun getAllCarts(): ApiResult<List<Carrito>> {
        val result = apiCall { apiService.getAllCarts() }
        return result
    }

    suspend fun getCartById(id: Long): ApiResult<Carrito> {
        val result = apiCall { apiService.getCartById(id) }
        return result
    }

    suspend fun createCart(cart: Carrito): ApiResult<Carrito> {
        val result = apiCall { apiService.createCart(cart) }
        return result
    }

    suspend fun updateCart(id: Long, cart: Carrito): ApiResult<Carrito> {
        val result = apiCall { apiService.updateCart(id, cart) }
        return result
    }

    suspend fun deleteCart(id: Long): ApiResult<Unit> {
        val result = apiCall { apiService.deleteCart(id) }
```

```
            return result
        }
    }
```

ViewModel para Gestión de Estado

CarritoViewModel - Integración con UI

```
// CarritoViewModel.kt
class CarritoViewModel(private val context: Context) : ViewModel() {

    private val repository = LevelUpRepository(context)

    private val _isLoading = MutableStateFlow(false)
    val isLoading: StateFlow<Boolean> = _isLoading.asStateFlow()

    private val _errorMessage = MutableStateFlow<String?>(null)
    val errorMessage: StateFlow<String?> = _errorMessage.asStateFlow()

    private val _carts = MutableStateFlow<List<Carrito>>(emptyList())
    val carts: StateFlow<List<Carrito>> = _carts.asStateFlow()

    init {
        loadAllCarts()
    }

    fun loadAllCarts() {
        viewModelScope.launch {
            _isLoading.value = true
            _errorMessage.value = null

            when (val result = repository.getAllCarts()) {
                is ApiResult.Success -> {
                    _isLoading.value = false
                    _carts.value = result.data
                }
                is ApiResult.Error -> {
                    _isLoading.value = false
                    _errorMessage.value = result.exception.message
                }
                is ApiResult.Loading -> {
                    _isLoading.value = result.isLoading
                }
            }
        }
    }

    fun createCart(userId: Long, products: List<CartProduct>) {
        viewModelScope.launch {
            _isLoading.value = true
            _errorMessage.value = null
```

```
            val newCart = Carrito(
                userId = userId,
                date = java.time.Instant.now().toString(),
                products = products
            )

            when (val result = repository.createCart(newCart)) {
                is ApiResult.Success -> {
                    _isLoading.value = false
                    playSuccessSound()
                    loadAllCarts() // Refresh list
                }
                is ApiResult.Error -> {
                    _isLoading.value = false
                    _errorMessage.value = result.exception.message
                }
                is ApiResult.Loading -> {}
            }
        }
    }
}
```

Gestión de Resultados API

Sealed Class para Estados

```
// ApiResult.kt
sealed class ApiResult<out T> {
    data class Success<T>(val data: T) : ApiResult<T>()
    data class Error(val exception: Exception) : ApiResult<Nothing>()
    data class Loading(val isLoading: Boolean = true) : ApiResult<Nothing>
()
}
```

Características Implementadas

🔄 Operaciones CRUD Completas

- **GET** `/carts` - Obtener todos los carritos
- **GET** `/carts/{id}` - Obtener carrito específico
- **POST** `/carts` - Crear nuevo carrito
- **PUT** `/carts/{id}` - Actualizar carrito existente
- **DELETE** `/carts/{id}` - Eliminar carrito

🛡️ Manejo de Errores Robusto

```
// Ejemplo de manejo de errores
when (val result = repository.getAllCarts()) {
    is ApiResult.Success -> {
        // Operación exitosa
        _carts.value = result.data
    }
    is ApiResult.Error -> {
        // Error en la operación
        _errorMessage.value = result.exception.message
    }
    is ApiResult.Loading -> {
        // Estado de carga
        _isLoading.value = result.isLoading
    }
}
```

⚡ Estados de Carga y UI Reactiva

- **Loading States**: Indicadores visuales durante operaciones
- **Error Handling**: Mensajes de error específicos
- **Success Feedback**: Sonidos y confirmaciones visuales

🔧 Configuración Avanzada de HTTP

- **Timeouts**: 30 segundos para conexión/lectura/escritura
- **Logging**: Log completo de requests/responses para debugging
- **Serialización**: JSON automático con kotlinx.serialization
- **Manejo de URLs**: Soporte para múltiples endpoints (backend propio + APIs externas)

Ventajas de la Implementación

✅ Separación de Responsabilidades

- **ApiService**: Define endpoints
- **Repository**: Maneja lógica de datos
- **ViewModel**: Gestiona estado de UI
- **Models**: Estructuras de datos tipadas

✅ Manejo Asíncrono

- **Coroutines**: Para operaciones no bloqueantes
- **Flow/StateFlow**: Para datos reactivos
- **ViewModelScope**: Manejo automático del ciclo de vida

✅ Flexibilidad

- **Múltiples APIs**: Backend propio + APIs externas
- **Configuración centralizada**: Un solo cliente HTTP
- **Fácil testing**: Interfaces inyectables

✅ Robustez

- **Manejo de errores**: Try-catch + estados específicos
- **Timeouts**: Evita colgadas de la aplicación
- **Logging**: Debugging efectivo

Ejemplo de Uso en UI

```
// En un Composable
@Composable
fun CarritoScreen(viewModel: CarritoViewModel = hiltViewModel()) {
    val carts by viewModel.carts.collectAsState()
    val isLoading by viewModel.isLoading.collectAsState()
    val errorMessage by viewModel.errorMessage.collectAsState()

    when {
        isLoading -> {
            CircularProgressIndicator()
        }
        errorMessage != null -> {
            Text("Error: $errorMessage", color = Color.Red)
        }
        else -> {
            LazyColumn {
                items(carts) { cart ->
                    CartItem(cart = cart)
                }
            }
        }
    }
}
```

Esta implementación proporciona una base sólida, escalable y mantenible para el consumo de APIs externas, combinando las mejores prácticas de Android con un diseño arquitectónico limpio.