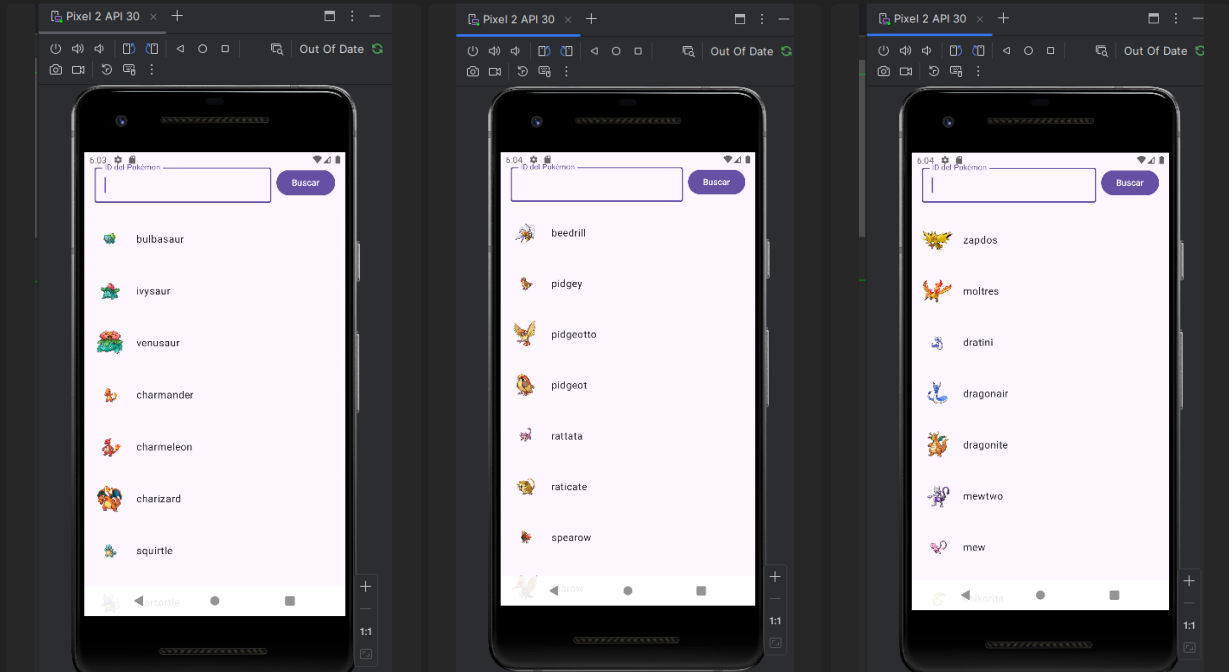


### Tarea 3.- Desarrollo de una Aplicación Móvil Nativa con Consumo de API REST

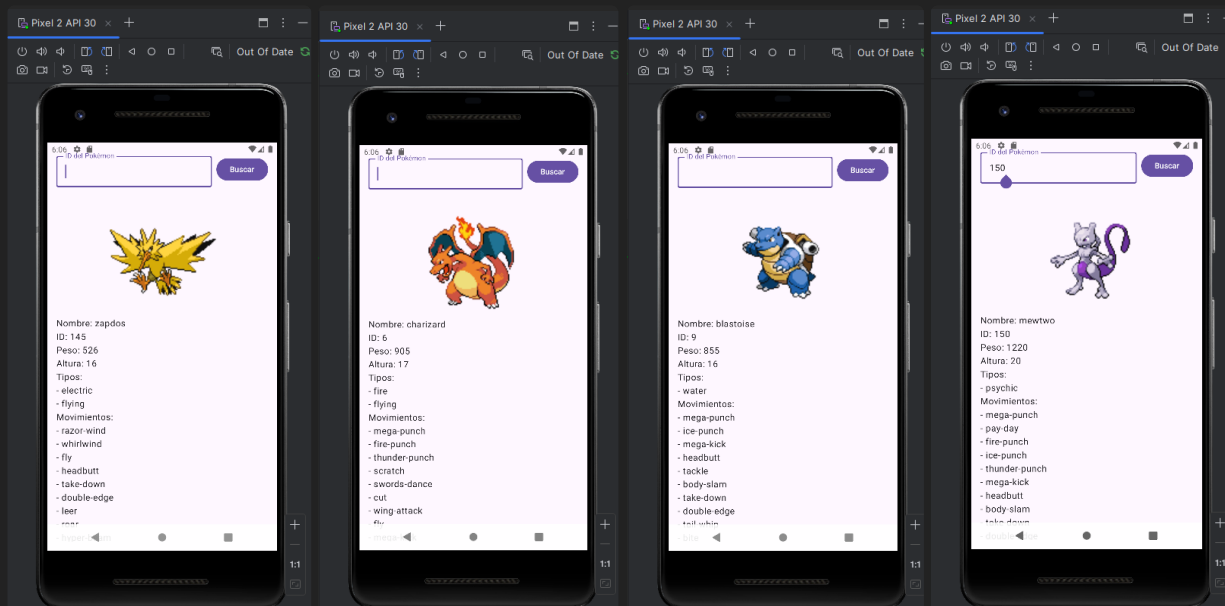
**Lopez Cardoza Victor Josue**

#### Capturas de pantalla de funcionalidad de la aplicación

A continuación, se puede ver el consumo de la PokeApi, donde podremos ver un listado de los primeros 300 pokemon, se muestra su imagen y su nombre.



Si se da clic en alguno de los pokemones se hara una petición GET para obtener los detalles de la seleccion, también es posible introducir un identificador directamente, para obtener los detalles del pokemon deseado.



## Código fuente de la aplicación

### 📌 1. Configuración de la actividad principal

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

Aquí **MainActivity** hereda de **ComponentActivity**, lo que permite que se use **Jetpack Compose** en la UI. Se sobrescribe **onCreate** para definir lo que ocurre cuando la actividad se inicia. Se usa **enableEdgeToEdge()** para optimizar el diseño en dispositivos con pantalla completa.

### 🔗 2. Configuración de Retrofit para consumir la API

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://pokeapi.co/api/v2/")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

val pokeApi = retrofit.create(PokeApi::class.java)
```

Se usa **Retrofit** para interactuar con **PokeAPI**.

- `baseUrl("https://pokeapi.co/api/v2/")` : Define la URL base de la API.

- `.addConverterFactory(GsonConverterFactory.create())`: Agrega un convertidor para transformar los JSON de la API en objetos de Kotlin.
  - `retrofit.create(PokeApi::class.java)`: Crea la interfaz que define las llamadas HTTP (aún no proporcionada, pero se asume que tiene métodos para obtener datos de Pokémon).
- 

### 3. Definiendo la interfaz de usuario con Compose

```
setContent {  
    PokeDexTheme {  
        Surface(  
            modifier = Modifier.fillMaxSize(),  
            color = MaterialTheme.colorScheme.background  
        ) {  
            PokedexApp(pokeApi = pokeApi)  
        }  
    }  
}
```

Aquí comienza la UI de la app.

- `setContent {}`: Define la interfaz con **Jetpack Compose**.
  - `PokeDexTheme {}`: Aplica el tema de la app.
  - `Surface {}`: Actúa como un contenedor de fondo para la UI.
  - `PokedexApp(pokeApi = pokeApi)`: Llama a la función principal que maneja la navegación y la lógica de la app.
- 

### 4. Estructura de la aplicación en PokedexApp

```
val navController = rememberNavController()  
val viewModel: PokemonViewModel = viewModel(factory =  
    PokemonViewModelFactory(pokeApi))  
var pokemonIdInput by remember { mutableStateOf("") }
```

- `rememberNavController()`: Crea un controlador de navegación para manejar las pantallas.
  - `viewModel`: Instancia un **ViewModel** usando una fábrica (`PokemonViewModelFactory`) para obtener datos desde **PokeAPI**.
  - `pokemonIdInput`: Variable de estado que almacena el ID ingresado por el usuario.
- 

### 5. Entrada de usuario para buscar un Pokémon

```
OutlinedTextField(
    value = pokemonIdInput,
    onValueChange = { pokemonIdInput = it },
    label = { Text("ID del Pokémon") },
    modifier = Modifier.weight(1f)
)
```

Aquí se usa `OutlinedTextField` para que el usuario escriba el ID del Pokémon que quiere buscar. Cada cambio en el texto se guarda en `pokemonIdInput`.

```
Button(
    onClick = {
        val pokemonId = pokemonIdInput.toIntOrNull()
        if (pokemonId != null) {
            navController.navigate("pokemonDetail/$pokemonId")
        }
    },
    modifier = Modifier.padding(start = 8.dp)
) {
    Text("Buscar")
}
```

Cuando se presiona el botón:

1. Convierte `pokemonIdInput` en número (`toIntOrNull()` evita errores si el usuario ingresa algo no numérico).
2. Si el número es válido, navega a la pantalla de detalles (`pokemonDetail/$pokemonId`).

## 6. Navegación entre pantallas

```
NavHost(navController = navController, startDestination = "pokemonList") {
```

Se usa `NavHost` para definir la navegación de la app. La pantalla inicial es `"pokemonList"`.

```
composable("pokemonList") {
    PokemonListScreen(navController = navController, viewModel = viewModel)
}
```

- `"pokemonList"` muestra una lista de Pokémon (`PokemonListScreen`).

```
composable(
    "pokemonDetail/{pokemonId}",
    arguments = listOf(navArgument("pokemonId") { type = NavType.IntType })
) { backStackEntry ->
    val pokemonId = backStackEntry.arguments?.getInt("pokemonId") ?: 1
    viewModel.fetchPokemonDetail(pokemonId)
    PokemonDetailScreen(pokemonId = pokemonId, viewModel = viewModel)
}
```

- "pokemonDetail/{pokemonId}" recibe un **argumento dinámico** ( pokemonId ).
- backStackEntry.arguments?.getInt("pokemonId") ?: 1: Obtiene el ID de la URL, si no hay ID usa 1.
- viewModel.fetchPokemonDetail(pokemonId): Llama al ViewModel para obtener los datos del Pokémon seleccionado.
- PokemonDetailScreen(): Muestra los detalles del Pokémon.

## ¿Qué es una interfaz en Retrofit?

En Retrofit, una **interfaz** se usa para definir los **endpoints** (URLs) de la API y cómo interactuar con ellos. Las funciones dentro de la interfaz se transforman en llamadas HTTP cuando Retrofit la implementa.

En este caso, PokeApi expone dos funciones principales:

1. **Obtener una lista de Pokémon** ( getPokemonList ).
2. **Obtener detalles de un Pokémon específico** ( getPokemonDetail ).

### 1. Obtener la lista de Pokémon

```
@GET("pokemon")
suspend fun getPokemonList(
    @Query("limit") limit: Int,
    @Query("offset") offset: Int
): PokemonListResponse
```

◆ @GET("pokemon") → Indica que esta función realizará una **solicitud GET** a la URL <https://pokeapi.co/api/v2/pokemon>.

◆ @Query("limit") limit: Int → Agrega el parámetro **limit**, que define cuántos Pokémon se quieren obtener en una sola petición.

◆ @Query("offset") offset: Int → Especifica desde qué número de Pokémon empezar a listar.

- ◆ `suspend` → Permite que la función sea llamada dentro de una **corrutina** (asíncrona).
- ◆ **Devuelve un `PokemonListResponse`**, que es un objeto que contiene la lista de Pokémon (no está definido aquí, pero es el modelo de datos que representa la respuesta).

### ✓ Ejemplo de uso

Si llamamos a esta función así:

```
pokeApi.getPokemonList(limit = 20, offset = 0)
```

Se traduciría en la siguiente petición HTTP:

```
GET https://pokeapi.co/api/v2/pokemon?limit=20&offset=0
```

**Esta solicitud devolvería una lista con 20 Pokémon, empezando desde el primero.**

## 🔍 2. Obtener detalles de un Pokémon específico

```
@GET("pokemon/{id}")
suspend fun getPokemonDetail(@Path("id") id: Int): Pokemon
```

- ◆ `@GET("pokemon/{id}")` → Realiza una solicitud GET a una URL que contiene un **parámetro dinámico** `{id}`.
- ◆ `@Path("id") id: Int` → Reemplaza `{id}` en la URL con el valor del Pokémon que se quiere consultar.
- ◆ `suspend` → Se ejecuta dentro de una **corrutina**.
- ◆ Devuelve un objeto **`Pokemon`**, que contiene los datos del Pokémon específico (nombre, tipos, estadísticas, etc.).

### ✓ Ejemplo de uso

Si llamamos a:

```
pokeApi.getPokemonDetail(25)
```

Se traduciría en esta petición HTTP:

```
GET https://pokeapi.co/api/v2/pokemon/25
```

## 📖 1. `PokemonListResponse` : Lista de Pokémon

```
data class PokemonListResponse(
    val count: Int,
    val next: String?,
    val previous: String?,
    val results: List<PokemonListItem>
)
```

- ◆ **count** : Número total de Pokémon en la API.
- ◆ **next** : URL de la siguiente página de resultados (puede ser `null` si no hay más).
- ◆ **previous** : URL de la página anterior (puede ser `null`).
- ◆ **results** : Lista de Pokémon obtenidos en la consulta, representados por `PokemonListItem`.

✓ **Ejemplo de respuesta JSON** Si consultamos la API con `limit=2`, obtendremos algo así:

```
{
  "count": 1281,
  "next": "https://pokeapi.co/api/v2/pokemon?offset=2&limit=2",
  "previous": null,
  "results": [
    { "name": "bulbasaur", "url": "https://pokeapi.co/api/v2/pokemon/1/" },
    { "name": "ivysaur", "url": "https://pokeapi.co/api/v2/pokemon/2/" }
  ]
}
```

La conversión a Kotlin con Retrofit se haría automáticamente usando `PokemonListResponse`.

## 🤖 2. `PokemonListItem` : Elementos de la lista de Pokémon

```
data class PokemonListItem(
    val name: String,
    val url: String
)
```

- ◆ **name** : Nombre del Pokémon (ejemplo: "bulbasaur").
- ◆ **url** : URL para obtener más detalles del Pokémon (ejemplo: "https://pokeapi.co/api/v2/pokemon/1/").

## 🔍 3. `Pokemon` : Representa los detalles de un Pokémon

```
data class Pokemon(
    val id: Int,
    val name: String,
    val sprites: Sprites,
    val types: List<Type>,
    val weight: Int,
    val height: Int,
    val moves: List<Move>,
    val stats: List<Stat>
)
```

Esta clase contiene información detallada de un Pokémon.

- ◆ `id` : Número identificador en la Pokédex (ejemplo: 25 para Pikachu).
- ◆ `name` : Nombre del Pokémon ( "pikachu" ).
- ◆ `sprites` : Imagen del Pokémon ( Sprites ).
- ◆ `types` : Lista de tipos (ejemplo: [ "electric" ] ).
- ◆ `weight` : Peso en hectogramos (ejemplo: 60 para Pikachu = 6.0 kg).
- ◆ `height` : Altura en decímetros (ejemplo: 4 para Pikachu = 0.4 m).
- ◆ `moves` : Lista de movimientos ( Move ).
- ◆ `stats` : Estadísticas base ( Stat ).

#### ✓ Ejemplo de respuesta JSON para Pikachu

```
{
  "id": 25,
  "name": "pikachu",
  "sprites": { "front_default": "https://pokeapi.co/media/sprites/pokemon/25.png" },
  "types": [{ "type": { "name": "electric" } }],
  "weight": 60,
  "height": 4,
  "moves": [{ "move": { "name": "thunderbolt" } }],
  "stats": [{ "base_stat": 90, "stat": { "name": "speed" } }]
}
```

Esto se convierte automáticamente en un objeto `Pokemon` .

## 4. Sprites : Imagen del Pokémon

```
data class Sprites(
    val front_default: String
)
```



- ◆ `front_default` : URL de la imagen del Pokémon en la API.

Ejemplo:

```
val pikachuSprite = Sprites("https://pokeapi.co/media/sprites/pokemon/25.png")
```

**Esto permitirá mostrar la imagen en la UI con Jetpack Compose.**

## ⚡ 5. Type : Representa los tipos del Pokémon

```
data class Type(  
    val type: TypeDetail  
)  
  
data class TypeDetail(  
    val name: String  
)
```

- ◆ `type` : Contiene `TypeDetail`.
- ◆ `TypeDetail.name` : Nombre del tipo ( "electric", "fire", "water", etc.).

Ejemplo en código:

```
val pikachuType = Type(TypeDetail("electric"))  
println(pikachuType.type.name) // Output: electric ⚡
```

## 👊 6. Move : Movimientos del Pokémon

```
data class Move(  
    val move: MoveDetail  
)  
  
data class MoveDetail(  
    val name: String  
)
```

- ◆ `move` : Contiene `MoveDetail`.
- ◆ `MoveDetail.name` : Nombre del movimiento ( "thunderbolt", "flamethrower", etc.).

Ejemplo:

```
val pikachuMove = Move(MoveDetail("thunderbolt"))
println(pikachuMove.move.name) // Output: thunderbolt ⚡
```

## 7. Stat : Estadísticas del Pokémon


```
data class Stat(
    val base_stat: Int,
    val stat: StatDetail
)


data class StatDetail(
    val name: String
)
```

- ◆ `base_stat` : Valor numérico de la estadística ( 90 para velocidad).
- ◆ `stat` : Contiene `StatDetail`.
- ◆ `StatDetail.name` : Nombre de la estadística ( "speed", "attack", "defense", etc.).

Ejemplo en código:

```
val pikachuSpeed = Stat(90, StatDetail("speed"))
println("${pikachuSpeed.stat.name}: ${pikachuSpeed.base_stat}") // Output: speed: 90
```

El Composable `PokemonDetailScreen` recibe dos parámetros: un `PokemonViewModel` (que presumiblemente maneja la lógica de negocio y la obtención de datos) y un `pokemonId` (el ID del Pokémon a mostrar). 

Luego, se observan dos estados del `ViewModel` usando `collectAsState()`: `pokemonDetail` y `isLoading`. `pokemonDetail` contiene la información del Pokémon (si ya se ha cargado), y `isLoading` indica si la carga está en progreso. 

La lógica principal se divide en dos partes basadas en `isLoading` y `pokemon`:

1. **Carga en progreso o Pokémon no encontrado:** Si `isLoading` es `true` o `pokemon` es `null`, se muestra un `CircularProgressIndicator` en el centro de la pantalla. Esto indica que la aplicación está esperando la respuesta del servidor o que no se encontró el pokemon.



Kotlin

```

if (isLoading || pokemon == null) {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        CircularProgressIndicator()
    }
}

```

2. **Pokémon cargado:** Si `isLoading` es `false` y `pokemon` no es `null`, se muestra la información del Pokémon. Se usa un `Column` para organizar los elementos verticalmente. Dentro de este `Column`:

- Se muestra la imagen del Pokémon usando `AsyncImage`, cargando la imagen desde la URL `pokemon?.sprites?.front_default`. 🖼️
- Se muestran varios `Text` con información como el nombre, ID, peso, altura, tipos, movimientos y estadísticas del Pokémon. 📄
- Se utilizan bucles `forEach` para iterar sobre las listas de tipos, movimientos y estadísticas, y mostrar cada elemento en un `Text` separado. 🔄

```

else {
    Column(modifier = Modifier.fillMaxSize().padding(16.dp)) {
        AsyncImage(
            model = pokemon?.sprites?.front_default,
            contentDescription = pokemon?.name,
            modifier = Modifier.size(200.dp).align(Alignment.CenterHorizontally)
        )
        Text(text = "Nombre: ${pokemon?.name}")
        // ... otros Text
    }
}

```

Por ejemplo, `pokemon?.types?.forEach { type -> Text(text = "- ${type.type.name}") }` itera sobre la lista de tipos del Pokémon y muestra cada tipo en un `Text` con un guion delante. Esto facilita la lectura de la lista de tipos. 📖

## 🏠 1. **PokemonListScreen** : Pantalla principal de la lista de Pokémon

```

@Composable
fun PokemonListScreen(navController: NavController, viewModel: PokemonViewModel) {
    val pokemonList by viewModel.pokemonList.collectAsState()
    val isLoading by viewModel.isLoading.collectAsState()

    if (isLoading) {
        Column(
            modifier = Modifier.fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            CircularProgressIndicator()
        }
    } else {
        LazyColumn(modifier = Modifier.fillMaxSize()) {
            items(pokemonList) { pokemon ->
                PokemonListItem(pokemon = pokemon) {
                    val pokemonId =
                        pokemon.url.substringAfter("pokemon/").substringBefore("/").toInt()
                    navController.navigate("pokemonDetail/$pokemonId")
                }
            }
        }
    }
}

```

## ¿Qué hace esta función?

La función `PokemonListScreen` es responsable de mostrar la lista de Pokémon en la pantalla.

- ◆ `viewModel.pokemonList.collectAsState()`: Observa el estado de la lista de Pokémon, que proviene del **ViewModel**. Cuando el estado cambia (se cargan más Pokémon, por ejemplo), se vuelve a componer la UI.
- ◆ `viewModel.isLoading.collectAsState()`: Se observa si los datos aún se están cargando. Si es `true`, muestra un **CircularProgressIndicator**.
- ◆ **LazyColumn**: Utiliza `LazyColumn` para renderizar la lista de Pokémon de forma eficiente. A medida que el usuario hace scroll, se van cargando más elementos solo cuando es necesario, optimizando el rendimiento.
- ◆ Si no hay datos de carga (`isLoading = false`), la lista de Pokémon se muestra usando el componente `PokemonListItem`.

## Flujo de navegación

Cuando el usuario hace clic en un ítem de la lista (`pokemon`), se navega a la pantalla de detalles de ese Pokémon utilizando el ID extraído de la URL del Pokémon. Esto es manejado por **NavController**:

```
navController.navigate("pokemonDetail/$pokemonId")
```

## ◆ 2. PokemonListItem : Componente para cada Pokémon en la lista

```
@Composable
fun PokemonListItem(pokemon: PokemonListItem, onClick: () -> Unit) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clickable { onClick() }
            .padding(16.dp),
        verticalAlignment = Alignment.CenterVertically
    ) {
        val pokemonId =
            pokemon.url.substringAfter("pokemon/").substringBefore("/").toInt()
        AsyncImage(
            model = "https://raw.githubusercontent.com/PokeAPI/sprites/master/sprites/pokemon/$pokemonId.png",
            contentDescription = pokemon.name,
            modifier = Modifier.size(50.dp)
        )
        Spacer(modifier = Modifier.size(16.dp))
        Text(text = pokemon.name)
    }
}
```

### 🎨 ¿Qué hace esta función?

`PokemonListItem` es un **componente reutilizable** para mostrar la información de un solo Pokémon dentro de la lista.

- ◆ **Row** : Organiza los elementos (imagen y nombre del Pokémon) horizontalmente.
- ◆ **clickable** : Hace que el elemento sea **cliqueable**, y cuando se hace clic, ejecuta la función `onClick()` , que pasa como parámetro al componente.
- ◆ **AsyncImage** : Usa **Coil** para cargar la imagen del Pokémon de manera asíncrona. La URL de la imagen se genera a partir del ID del Pokémon ( `$pokemonId.png` ). Esto asegura que cada Pokémon tenga su propia imagen cargada desde un recurso externo.
- ◆ **Spacer** : Añade espacio entre la imagen y el nombre del Pokémon para mantener la UI limpia y ordenada.
- ◆ **Text** : Muestra el nombre del Pokémon.

### ⚡ Extracción del ID del Pokémon

El ID del Pokémon se extrae de la URL asociada al Pokémon, que tiene el siguiente formato:

```
val pokemonId = pokemon.url.substringAfter("pokemon/").substringBefore("/").toInt()
```

**Este fragmento de código extrae el número de la Pokédex (por ejemplo, "1" para Bulbasaur) de la URL "https://pokeapi.co/api/v2/pokemon/1/" .**

## Flujo general de la UI

1. **Carga de los Pokémon:** Cuando se muestra la lista de Pokémon, si los datos están siendo cargados, se muestra un **ícono de carga** en el centro de la pantalla con el `CircularProgressIndicator`.
2. **Despliegue de la lista:** Una vez que los datos han sido cargados, se utiliza un `LazyColumn` para mostrar la lista de los Pokémon. Cada elemento es un **componente `PokemonListItem`** con una imagen y el nombre del Pokémon.
3. **Navegación a detalles:** Al hacer clic en un Pokémon, el ID del Pokémon es extraído de su URL, y el **`NavController`** navega a la pantalla de detalles de ese Pokémon.

## 1. `PokemonViewModel` : La vista del modelo de datos

```
class PokemonViewModel(private val pokeApi: PokeApi) : ViewModel() {
    // Estado para la lista de Pokémon
    private val _pokemonList = MutableStateFlow<List<PokemonListItem>>(emptyList())
    val pokemonList: StateFlow<List<PokemonListItem>> = _pokemonList


    // Estado para los detalles de un Pokémon
    private val _pokemonDetail = MutableStateFlow<Pokemon?>(null)
    val pokemonDetail: StateFlow<Pokemon?> = _pokemonDetail

    // Estado para la carga (loading)
    private val _isLoading = MutableStateFlow(false)
    val isLoading: StateFlow<Boolean> = _isLoading

    // Estado para manejar errores
    private val _error = MutableStateFlow<String?>(null)
    val error: StateFlow<String?> = _error

    // Inicialización del ViewModel
    init {
        fetchPokemonList()
    }
}
```

## ¿Qué hace esta clase?

- **PokemonViewModel**: Es una clase que extiende `ViewModel` y sirve como el intermediario entre la **UI** (pantallas) y los datos (información sobre los Pokémon). El `ViewModel` no debería contener lógica de UI, solo debe gestionar el estado y la lógica de los datos.
  -  **Estado del ViewModel:**
    - **\_pokemonList**: Mantiene una lista de objetos `PokemonListItem` (un Pokémon en su forma más básica, como nombre y URL).
    - **\_pokemonDetail**: Mantiene los detalles de un Pokémon, como su nombre, imagen, estadísticas, etc.
    - **\_isLoading**: Representa el estado de carga (si los datos están siendo obtenidos de la API).
    - **\_error**: Guarda un mensaje de error si ocurre algún problema al obtener los datos.
  - **StateFlow**: Se utiliza para exponer datos a la UI. `StateFlow` es una versión de flujo (`Flow`) que se comporta como un **estado observable**. Permite que la UI se suscriba a los cambios de estado.
  - **init { fetchPokemonList() }**: En el bloque `init` (inicialización del `ViewModel`), se llama automáticamente a `fetchPokemonList()`, lo que significa que al iniciar la vista, la lista de Pokémon se obtiene de inmediato.
- 

## 2. **fetchPokemonList : Obtener lista de Pokémon**

```
private fun fetchPokemonList(limit: Int = 300, offset: Int = 0) {
    viewModelScope.launch {
        _isLoading.value = true
        try {
            val response = pokeApi.getPokemonList(limit, offset)
            _pokemonList.value = response.results
            _isLoading.value = false
        } catch (e: Exception) {
            _error.value = e.message
            _isLoading.value = false
        }
    }
}
```

### ¿Qué hace esta función?

- **Objetivo**: Obtener la lista de Pokémon de la API.
- **viewModelScope.launch**: Lanza una **coroutine** en el **viewModelScope**, lo que asegura que la tarea asíncronica se maneje correctamente dentro del ciclo de vida del `ViewModel`.
- **pokeApi.getPokemonList(limit, offset)**: Llama a la función `getPokemonList` del servicio API que hace una petición HTTP para obtener la lista de Pokémon. Los parámetros `limit` y `offset` permiten controlar cuántos elementos se obtienen y desde qué punto.
- **Manejo del estado**:

- `_isLoading.value = true`: Se establece el estado de carga.
  - `_pokemonList.value = response.results`: Una vez que se obtiene la respuesta, se actualiza la lista de Pokémon en el estado.
  - **Manejo de errores**: Si ocurre algún error, se captura y se muestra en el estado `_error`.
- 

### 3. `fetchPokemonDetail`: Obtener detalles de un Pokémon

```
fun fetchPokemonDetail(id: Int) {
    viewModelScope.launch {
        _isLoading.value = true
        try {
            _pokemonDetail.value = pokeApi.getPokemonDetail(id)
            _isLoading.value = false
        } catch (e: Exception) {
            _error.value = e.message
            _isLoading.value = false
        }
    }
}
```

#### ¿Qué hace esta función?

- **Objetivo**: Obtener los detalles de un Pokémon específico.
  - `pokeApi.getPokemonDetail(id)`: Llama a la API para obtener los detalles de un Pokémon usando su ID.
  - **Manejo de estado**:
    - `_isLoading.value = true`: Establece que los datos están siendo cargados.
    - `_pokemonDetail.value = pokeApi.getPokemonDetail(id)`: Una vez que la respuesta se recibe, se actualiza el detalle del Pokémon en el estado.
    - **Manejo de errores**: Si hay un error, se captura y se establece en el estado de error.
- 

#### ¿Cómo se conecta con la UI?

La UI puede observar los **StateFlows** y reaccionar a los cambios de estado:

- La UI está suscrita a `pokemonList` y `pokemonDetail` para actualizar las pantallas cuando los datos cambian.
  - El estado de `isLoading` se usa para mostrar un indicador de carga cuando los datos se están obteniendo.
  - Si ocurre un error, la UI puede mostrar un mensaje de error.
-





## Explicación detallada del código

```
class PokemonViewModelFactory(private val pokeApi: PokeApi) :
    ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(PokemonViewModel::class.java)) {
            return PokemonViewModel(pokeApi) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

### 1. La factoría de ViewModel

- **PokemonViewModelFactory**: Esta clase se encarga de crear instancias del **PokemonViewModel**. La principal razón de su existencia es que el **PokemonViewModel** necesita un parámetro en su constructor (**pokeApi: PokeApi**), lo cual no puede ser hecho automáticamente por el sistema de inyección de dependencias de Android (si no se usa algo como Dagger o Hilt). Así que, al usar esta factoría, podemos proporcionar el objeto **pokeApi** cuando se cree la instancia del **ViewModel**.

### 2. Implementación de **ViewModelProvider.Factory**

- **ViewModelProvider.Factory**: Android usa este patrón para crear **ViewModels**. La factoría se utiliza cuando se necesita proporcionar parámetros adicionales al **ViewModel** en su construcción.
- El método **create** es el núcleo de esta implementación. Este método es llamado cuando Android necesita crear una instancia del **ViewModel**.

### 3. Verificación de tipo y creación del ViewModel

- **modelClass.isAssignableFrom(PokemonViewModel::class.java)**: Verifica si el tipo del **ViewModel** que se está creando es compatible con **PokemonViewModel**. Esto asegura que la factoría solo se utiliza para crear **PokemonViewModel** y no otro tipo de **ViewModel**.
- **return PokemonViewModel(pokeApi) as T**: Si la clase del **ViewModel** es correcta, se crea una instancia de **PokemonViewModel**, pasando el **pokeApi** al constructor. Luego, esa instancia se convierte al tipo **T** y se devuelve.
- **throw IllegalArgumentException("Unknown ViewModel class")**: Si el tipo del **ViewModel** no es compatible con **PokemonViewModel**, se lanza una excepción indicando que el tipo es desconocido. Esto es una medida de seguridad para garantizar que solo se creen los **ViewModel** que se esperan.



## ¿Cómo se usa esta factoría?

En el **MainActivity** o en cualquier otra parte del código donde necesitemos un **PokemonViewModel**, la factoría se utiliza de la siguiente manera:

```
val viewModel: PokemonViewModel = viewModel(factory =
PokemonViewModelFactory(pokeApi))
```

- **viewModel(factory = PokemonViewModelFactory(pokeApi))**: Aquí es donde la factoría entra en acción. Se pasa la factoría `PokemonViewModelFactory` al método `viewModel()`, que crea una instancia del `PokemonViewModel` y proporciona la instancia de `pokeApi`.
- 

## Resumen

- **PokemonViewModelFactory**: Es responsable de crear instancias del `PokemonViewModel` cuando este requiere parámetros adicionales en su constructor (como el `pokeApi`).
- **ViewModelProvider.Factory**: Es una interfaz que permite la creación personalizada de `ViewModels` con parámetros.
- **create**: El método que asegura que el `ViewModel` correcto se cree y se devuelva.

Este patrón es útil cuando necesitamos proporcionar dependencias externas (como una API o un repositorio) al `ViewModel` para que pueda hacer el trabajo adecuado.