Creating a Pokemon Info App with Jetpack Compose

This tutorial will guide you through creating an Android app that fetches Pokemon data using Retrofit2 and displays it with the help of Coil. We'll use an empty activity as the starting point and integrate the necessary libraries.

1. Project Setup:

- **Create a new Android Studio Project:**
  - Select "Empty Activity" as the template.
  - Give your project a name (e.g., "PokeComposerApp").
  - Set the minimum SDK as per your requirements.
- **Add Dependencies to build.gradle (Module: app):**

  Gradle

```
dependencies {
    // ... other dependencies ...

    implementation 'com.squareup.retrofit2:retrofit:2.9.0'
    implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
    implementation 'io.coil-kt:coil-compose:2.1.0'
// ... other dependencies ...

    }
```

The dependencies used in this project are libraries that help us with specific tasks. Retrofit2 handles network requests, allowing us to fetch data from the Pokemon API. GsonConverterFactory converts JSON data from the API into usable objects. Coil efficiently loads and displays images, like the Pokemon sprites. These dependencies save us time and effort by providing pre-built functionality for common tasks, making our app more efficient and easier to develop.

- **Add Internet Permission to AndroidManifest.xml:**

The AndroidManifest.xml file is the primary configuration file for every Android app. It contains essential information about the app, such as its package name, version, permissions, and the components it consists of (activities, services, receivers, etc.). To enable network connectivity, the `android.permission.INTERNET` permission is declared within this file. This permission allows the app to establish network connections and send/receive data over the internet. Without this permission, the app would be unable to fetch data from the Pokemon API, as network access would be restricted.

So our manifest file will have the following code:

```
<uses-permission android:name="android.permission.INTERNET" />
```

2. Define Data Classes:

- Create a new Kotlin file (e.g., PokemonData.kt) in the model package:

```kotlin
data class PokemonData (
    val name: String,
    val sprites: PokemonSprites,
    val types: List<PokemonType>,
    val abilities: List<PokemonAbility>
)

data class PokemonSprites(val front_default: String)
data class PokemonType(val type: PokemonTypeDetails)
data class PokemonTypeDetails(val name: String)
data class PokemonAbility(val ability: PokemonAbilityDetail)
data class PokemonAbilityDetail(val name: String)
```

This step involves creating Kotlin classes that mirror the structure of the JSON data we expect to receive from the Pokemon API. For instance, the `PokemonData` class has properties like `name`, `sprites`, `types`, and `abilities` to match the corresponding fields in the JSON response. When a network request is made using Retrofit2, the JSON response is parsed by GsonConverterFactory. This library maps the JSON data to the defined data classes, automatically populating the fields of each object. This parsed data can then be used to update the UI elements in the app, such as displaying the Pokemon's name, image, types, and abilities.

3. Create Retrofit Service:

- Create a new Kotlin file (e.g., PokemonService.kt) in the network package:

```kotlin
import retrofit2.Call
import retrofit2.http.GET
import retrofit2.http.Path

interface PokemonService {
    @GET("pokemon/{name}")
    fun getPokemon(@Path("name") name: String): Call<PokemonData>
}
```

This step involves creating an interface that defines the network requests we want to make to the Pokemon API. This interface acts as a contract between our app and the API.
In the PokemonService interface, we define a single method getPokemon. This method is annotated with @GET and specifies the endpoint URL with a path parameter /{name}. This parameter will be replaced with the actual Pokemon name when we make the request.
By defining this interface, we're essentially telling Retrofit how to construct the network request: the HTTP method (GET), the URL, and the parameter. Retrofit will then generate the necessary code to make this request when we call the getPokemon method from our activity. This approach makes it easier to work with APIs and simplifies the process of making network requests in our app.

4. Create Pokemon Repository:

- Create a new Kotlin file (e.g., PokemonRepository.kt) in the repository package:

```
import android.util.Log
import com.example.pokecomposerapp.model.PokemonData
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
import java.lang.Exception

class PokemonRepository {
    private val pokemonService = Retrofit.Builder()
        .baseUrl("https://pokeapi.co/api/v2/")
        .addConverterFactory(GsonConverterFactory.create())
        .build()
        .create(PokemonService::class.java)

    suspend fun getPokemon(name: String): PokemonData? {
        return try {
            pokemonService.getPokemon(name).execute().body()
        } catch (e: Exception) {
            Log.e("Pokemon", e.localizedMessage)
            null
        }
    }
}
```

The Pokemon Repository in this example acts as a single source of truth for fetching Pokémon data. It abstracts the underlying data source (the Pokémon API in this case) behind an interface. This allows the ViewModel and other parts of the application to interact with the data without needing to know the specific implementation details of how the data is retrieved. This separation of concerns improves code maintainability, testability, and makes it easier to switch data sources in the future if needed. The concept of the Repository pattern is a common architectural pattern in software development, promoting loose coupling and making the system more flexible and adaptable to change.

5. Create Pokemon ViewModel:

- Create a new Kotlin file (e.g., PokemonViewModel.kt) in the viewmodel package:

```
import android.util.Log
import androidx.lifecycle.ViewModel
import com.example.pokecomposerapp.model.PokemonRepository
import com.example.pokecomposerapp.view.UIState
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.withContext

class PokemonViewModel(private val repository: PokemonRepository) :
ViewModel() {
    private val _uiState = MutableStateFlow(UIState())
    val uiState = _uiState.asStateFlow()

    suspend fun onSearchTextChanged(text: String) {
        withContext(Dispatchers.IO) {
            try {
                val pokemonData = repository.getPokemon(text)
                if (pokemonData != null) {
                    _uiState.value = _uiState.value.copy(pokemonData =
pokemonData)
                    Log.d("Pokemon", "fetched ${pokemonData.name}")
                } else {
                    Log.e("Pokemon", "no data $text")
                }
            } catch (e: Exception) {
                Log.e("Pokemon", e.localizedMessage)
            }
        }
    }
}

data class UIState(val pokemonData: PokemonData? = null)
```

The Pokemon ViewModel acts as an intermediary between the UI (the `PokemonScreen`) and the data layer (the `PokemonRepository`). It's responsible for fetching Pokémon data from the repository, handling any asynchronous operations (like network requests) in a background thread, and updating the UI with the fetched data. By handling the data fetching logic within the ViewModel, the UI remains clean and focused solely on presentation. This also improves testability as the ViewModel's logic can be tested independently of the UI.

6. Create Compose UI Components:

- Create a new Kotlin file (e.g., PokemonScreen.kt) in the view package:

```
@Composable
fun SearchBar(
    value: String,
    onValueChange: (String) -> Unit,
    onSearch: () -> Unit
) {
    Row(Modifier.padding(16.dp)) {
        TextField(
            value = value,
            onValueChange = onValueChange,
            label = { Text(text = "Search Pokemon") }
        )
        OutlinedButton(onClick = onSearch, Modifier.padding(8.dp)) {
            Text(text = "Search")
        }
    }
}

@Composable
fun PokemonCard(pokemonData: PokemonData) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(8.dp)
    ) {
        Column {
            AsyncImage(
                model = pokemonData.sprites.front_default,
                contentDescription = "an image of
${pokemonData.name}",
                modifier = Modifier
                    .size(300.dp)
                    .aspectRatio(1f)
                    .align(alignment = Alignment.CenterHorizontally)
            )
            Text(text = pokemonData.name)
            Text(
                text = "Types: " +
                        "${pokemonData.types.map { it.type.name
}.joinToString(", ")}"
            )
            Text(
                text = "Abilities: " +
                        "${pokemonData.abilities.map { it.ability.name
}.joinToString(", ")}"
            )
        }
    }
}

@Composable
fun PokemonScreen(viewModel: PokemonViewModel) {
    val state = viewModel.uiState.collectAsState()
    var searchText by remember { mutableStateOf("") }
    val scope = rememberCoroutineScope()
    Column() {
        SearchBar(
            value = searchText,
            onValueChange = { searchText = it },
            onSearch = {
                scope.launch {
                    viewModel.onSearchTextChanged(searchText)
                }
            }
        )
        state.value.pokemonData?.let {
            PokemonCard(pokemonData = it)
        }
    }
}
```

In this step, we define the user interface elements of our application using Jetpack Compose.

- **`SearchBar` Composable:**
    - This composable creates a row containing a `TextField` for user input and an `OutlinedButton` for triggering the search.
    - It accepts the current search text (`value`), a function to update the search text (`onValueChange`), and a function to initiate the search (`onSearch`) as parameters.
- **`PokemonCard` Composable:**
    - This composable displays the details of a single Pokémon.
    - It takes a `PokemonData` object as input.
    - It uses `AsyncImage` from the Coil library to load and display the Pokémon's image.
    - It displays the Pokémon's name, types, and abilities using `Text` components.
- **`PokemonScreen` Composable:**
    - This is the main screen of the application.
    - It collects the current UI state from the `PokemonViewModel` using `collectAsState()`.
    - It uses the `SearchBar` composable to allow the user to enter and submit a Pokémon name.
    - It uses `rememberCoroutineScope()` to launch a coroutine for executing the search operation.
    - Conditionally renders the `PokemonCard` if a Pokémon object is available in the UI state.

Key Concepts:

- **State:** The `PokemonScreen` uses `remember` to manage the `searchText` state within the composable.

- **Side Effects:** The search operation is handled as a side effect using `launch` within a `CoroutineScope`.
- **Data Flow:** The `collectAsState()` function observes the `uiState` Flow emitted by the `PokemonViewModel` and provides the latest state to the `PokemonScreen` for rendering.

7. MainActivity:

- In MainActivity.kt:

```
class MainActivity : ComponentActivity() {
    val pokemonViewModel = PokemonViewModel(PokemonRepository())

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            PokeComposerAppTheme {
                // A surface container using the 'background' color
from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    PokemonScreen(viewModel[3] = pokemonViewModel)
                }
            }
        }
    }
}
```

- Purpose: This is the entry point of the application. It's responsible for setting up the Compose UI and providing the necessary context for the application to run.

- ViewModel Injection:

    - `val pokemonViewModel = PokemonViewModel(PokemonRepository())`: This line creates an instance of the `PokemonViewModel`.
        - Note: In a real-world application, you would typically use a dependency injection framework (like Hilt or Koin) to provide instances of the ViewModel and its dependencies. This would improve testability and make the code more modular.
- Setting up Compose UI:

    - `setContent { ... }`: This block defines the Compose UI hierarchy for the

application.
    - `PokeComposerAppTheme`: This is a composable function (you would need to define it in a separate file) that sets up the theme for the application, including colors, typography, and other visual styles.
    - `Surface`: This provides a background color for the UI.
    - `PokemonScreen`: This is the main screen composable that we defined earlier. It's passed the `pokemonViewModel` instance as a parameter.

Key Points:

- `MainActivity` acts as the bridge between the Android Activity lifecycle and the Compose UI.
- It's crucial to properly handle the lifecycle of the `ViewModel` to prevent memory leaks. In this simplified example, we're creating the `ViewModel` within `MainActivity`. However, for more complex applications, using a dependency injection framework is recommended.

8. Run the App:

- Run the app on an emulator or a physical device.
- Enter a Pokémon name in the search bar (e.g., "pikachu") and click "Search."
- The app should display the Pokémon's image, name, types, and abilities.