POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

ĆWICZENIE LABORATORYJNE NR 4

REST API, Retrofit i Coil

Programowanie Aplikacji Mobilnych (Android / Kotlin / Jetpack Compose)

Projekt: **PokeApp - przeglądarka Pokemonów z PokeAPI**

Wymagania: ukończone lab 1–3 (Room, ViewModel, Navigation, Coroutines)

Spis treści

1. Jak działa komunikacja przez Internet?

Zanim napiszemy pierwszą linię kodu sieciowego, musimy zrozumieć, co tak naprawdę dzieje się pod maską, gdy aplikacja mobilna prosi o dane z Internetu. Ten rozdział wyjaśnia mechanizm krok po kroku - od wpisania adresu URL aż do wyświetlenia listy Pokemonów na ekranie.

1.1. Sekwencja żądania HTTP - krok po kroku

Każde żądanie sieciowe to złożona choreografia wielu systemów. Poniższa tabela pokazuje kolejność zdarzeń - co dokładnie się dzieje zanim zobaczysz dane na ekranie.

#	Etap żądania HTTP (od aplikacji do danych na ekranie)
1	DNS Resolution - aplikacja pyta serwer DNS o adres IP dla nazwy pokeapi.co. Bez tego kroku internet nie wie, gdzie wysłać dane.
2	TCP Handshake - trzy pakiety (SYN → SYN-ACK → ACK) nawiązują połączenie z serwerem. Dopiero teraz mamy 'otwartą linię'.
3	TLS Handshake - wymiana certyfikatów i ustalenie klucza szyfrowania (dla HTTPS). Gwarantuje, że nikt nie podsłucha danych.
4	Wysłanie żądania HTTP - GET /api/v2/pokemon?limit=20 z nagłówkami (Accept, User-Agent). OkHttp buduje i wysyła ten pakiet.

#	Etap żądania HTTP (od aplikacji do danych na ekranie)
5	Serwer przetwarza żądanie - PokeAPI odpytuje swoją bazę danych i generuje odpowiedź JSON.
6	Odpowiedź HTTP dociera - kod statusu 200, nagłówki Cache-Control, body z JSON-em. OkHttp odbiera pakiety.
7	Deserializacja (Gson) - JSON zamieniany jest na obiekty Kotlin (PokemonListDto). Gson mapuje klucze JSON na pola data class.
8	Warstwą Repository - dane trafiają do Room lub bezpośrednio do ViewModel. Room staje się jedynym źródłem prawdy dla UI.
9	Aktualizacja UI - StateFlow emituje nowy stan, Compose automatycznie rerenderuje ekran z nowymi danymi.

🍽 Analogia: Kelner i restauracja

Wyobraź sobie, że jesteś w restauracji:

• APLIKACJA = Ty (klient siedzący przy stoliku) - chcesz coś zjeść.

• API = Kelner - przyjmuje zamówienie, zanosi do kuchni, przynosi danie.

• SERWER/BAZA DANYCH = Kuchnia - tu faktycznie przygotowywane są dane.

• ŻĄDANIE HTTP = 'Poproszę spaghetti bolognese' - sformułowane zamówienie.

• ODPOWIEDŹ HTTP = Talerz z jedzeniem - albo informacja 'nie ma dziś w menu' (kod 404).

• JSON = Format, w jakim danie zostało zapakowane - zawsze ten sam standard, niezależnie od kuchni.

Kelner (API) nie musi wiedzieć, jak się gotuje. Kuchnia (serwer) nie musi wiedzieć, kim jesteś.

To właśnie jest istota REST: oddzielenie klienta od serwera przez standaryzowany interfejs.

1.2. Metody HTTP - kiedy używać której

HTTP definiuje kilka 'czasowników' opisujących, co chcemy zrobić z zasobem. Błędne użycie metody jest jednym z najczęstszych błędów przy integracji z API.

Metoda	Kiedy używać - zasada i przykład
`GET`	Pobieranie danych bez ich modyfikacji. Idempotentna (wielokrotne wywołanie daje ten sam wynik). Przykład: GET /pokemon/25 (pobierz Pikachu).
`POST`	Tworzenie nowego zasobu. Serwer nadaje mu ID. NIE jest idempotentna. Przykład: POST /users (stwórz nowego użytkownika).
`PUT`	Zastąpienie całego zasobu nową wersją. Musisz wysłać wszystkie pola. Przykład: PUT /tasks/5 (zastąp całe zadanie nr 5).
`PATCH`	Aktualizacja tylko wybranych pól zasobu. Wysyłasz tylko to, co chcesz zmienić. Przykład: PATCH /tasks/5 (zmień tylko status).
`DELETE`	Usunięcie zasobu. Idempotentna. Przykład: DELETE /pokemon/25 (usuń Pikachu z listy ulubionych).

1.3. Kody statusu HTTP - co mówią deweloperowi

Każda odpowiedź HTTP zawiera trzyznakowy kod statusu. Pierwsza cyfra informuje o kategorii wyniku. Znajomość tych kodów jest kluczowa przy debugowaniu aplikacji.

Kod / grupa	Znaczenie i reakcja dewelopera
`2xx - Sukces`	Żądanie zakończone pomyślnie. 200 OK: zasób zwrócony. 201 Created: zasób stworzony. 204 No Content: akcja wykonana, brak treści. → Przetwarzaj dane normalnie.

Kod / grupa	Znaczenie i reakcja dewelopera
`4xx - Błąd klienta`	Problem po stronie żądania. 400 Bad Request: niepoprawne dane (sprawdź JSON). 401 Unauthorized: brak autoryzacji. 403 Forbidden: brak uprawnień. 404 Not Found: zasób nie istnieje. → Pokaż komunikat użytkownikowi, NIE próbuj ponownie.
`5xx - Błąd serwera`	Problem po stronie serwera. 500 Internal Server Error: błąd kodu serwera. 503 Service Unavailable: serwer przeciążony. → Pokaż komunikat, rozważ retry z exponential backoff.
`429 - Rate Limit`	Zbyt wiele żądań. Serwer odmawia obsługi. → Zaimplementuj opóźnienie przed ponowną próbą. PokeAPI ma limit 100 req/min.

2. JSON - format wymiany danych

JSON (JavaScript Object Notation) jest dziś dominującym formatem wymiany danych w aplikacjach mobilnych. Jest czytelny dla człowieka, lekki i obsługiwany przez praktycznie każde środowisko programistyczne. Rozumienie JSON jest absolutnie niezbędne do pracy z REST API.

2.1. Typy wartości JSON i ich odpowiedniki w Kotlinie

Typ JSON	Odpowiednik Kotlin / uwagi
`{ ... } (obiekt)`	data class - każdy klucz JSON odpowiada polu klasy. Użyj @SerializedName jeśli nazwy się różnią.
`[ ... ] (tablica)`	List<T> - kolejność elementów jest zachowana. Może być pusta (pusta lista, nie null!).
`"tekst" (string)`	String - zawsze w cudzysłowach w JSON. Może być null w JSON → String? w Kotlinie.
`42, 3.14 (number)`	Int, Long, Double, Float - wybierz odpowiedni zakres. ID często wymagają Long.
`true / false (boolean)`	Boolean - bezpośrednie mapowanie.
`null`	Wymaga typu nullable T? w Kotlinie. Pola nullable MUSZĄ mieć ? - inaczej Gson rzuci wyjątek.

2.2. Fragment JSON z PokeAPI - analiza linia po linii

Odpowiedź GET /api/v2/pokemon/25 (fragment)	Wyjaśnienie
`{`	Początek obiektu JSON - będzie mapowany na data class PokemonDetailDto
`  "id": 25,`	Klucz 'id', wartość integer. → val id: Int
`  "name": "pikachu",`	Klucz 'name', wartość string. → val name: String (zawsze małe litery!)
`  "base_experience": 112,`	Klucz z podkreślnikiem. → @SerializedName("base_experience") val baseExperience: Int
`  "height": 4,`	Wysokość w decymetrach (nie metrach!). W mapperze podzielimy przez 10.0.
`  "weight": 60,`	Waga w hektogramach (nie kilogramach). W mapperze podzielimy przez 10.0.
`  "sprites": {`	Zagnieżdżony obiekt - wymaga osobnej data class SpritesDto.
`    "front_default": "https://..."`	URL obrazka. Może być null dla niektórych Pokemonów. → String?
`  },`	Koniec obiektu zagnieżdżonego.
`  "types": [`	Tablica typów. → List<TypeSlotDto>
`    {"slot": 1,`	Pozycja w hierarchii typów.
`      "type": {"name": "electric"}}`	Zagnieżdżony obiekt z nazwą typu.
`  ],`	Koniec tablicy.
`  "abilities": [...]`	Lista zdolności. Każda zdolność ma flagę 'is_hidden'. Filtrujemy w mapperze.
`}`	Koniec głównego obiektu.

⚠ Najczęstsze pułapki przy mapowaniu JSON → Kotlin

1. BRAK @SerializedName - jeśli pole w JSON ma format snake_case (base_experience), a Kotlin camelCase (baseExperience), musisz dodać @SerializedName("base_experience"). Bez tego Gson nie znajdzie pola i zwróci domyślną wartość (0, null, false).
2. BRAK ZNAKU '?' dla wartości nullable - jeśli w JSON pole może być null (np. sprites.front_default), a w Kotlinie masz String (bez ?), Gson rzuci JsonSyntaxException lub - co gorsza - cicho zignoruje błąd i Coil otrzyma null.
3. ZŁY TYP - ID Pokemona mieści się w Int, ale ID zasobów na wielu API wymagają Long (ponad 2 miliardy wpisów).
4. ZAGNIEŻDŻONE OBIEKTY wymagają osobnych data class. Nie możesz zmapować { "type": { "name": "fire" } } na String.

3. Architektura stosu sieciowego

Aplikacja PokeApp korzysta z kilku bibliotek, które razem tworzą warstwowy stos sieciowy. Każda warstwa ma jedną, dobrze zdefiniowaną odpowiedzialność. Zrozumienie tej architektury ułatwia debugowanie: gdy coś nie działa, wiesz od razu w której warstwie szukać błędu.

3.1. Diagram warstw

Warstwa	Odpowiedzialność / typowe błędy
`ViewModel / Repository`	Logika biznesowa. Decyduje: odczytać z cache czy z sieci? Mapuje DTO na Domain Model. Błędy: brak mappera, złe zarządzanie stanem.
`Retrofit`	Interfejs API. Buduje żądanie HTTP z adnotacji (@GET, @Path). Przekazuje do OkHttp. Błędy: zły baseUrl, brakujące @Path.
`Gson Converter`	Serializacja/deserializacja JSON ↔ Kotlin. Mapuje pola JSON na pola data class. Błędy: JsonSyntaxException, brak @SerializedName.
`OkHttp`	Transport HTTP. Zarządza połączeniami, cache, timeoutami, interceptorami (logowanie). Błędy: timeout, SSL, brak sieci.
`Android Network Stack`	Niskopoziomowy dostęp do sieci. Wymaga uprawnienia INTERNET w Manifeście. Błędy: NetworkOnMainThreadException, brak uprawnień.

Dlaczego tyle bibliotek?, czyli zasada Single Responsibility

Każda biblioteka rozwiązuje jeden konkretny problem i robi to bardzo dobrze. Retrofit skupia się wyłącznie na zamienianiu adnotacji na żądania HTTP - nie zarządza połączeniami.

OkHttp zarządza połączeniami, pulą wątków i cache - ale nie parsuje JSON.

Gson parsuje JSON - ale nie wie nic o sieci.

Dzięki temu możesz zamienić Gson na Moshi bez dotykania kodu OkHttp. Możesz dodać nowy interceptor bez zmiany interfejsu Retrofit.

Gdyby jeden 'super-komponent' robił wszystko, zmiana jednej rzeczy powodowałaby kaskadę modyfikacji w całym kodzie.

To właśnie jest zaleta architektury warstwowej i zasady Single Responsibility.

🔍 Dlaczego suspend fun? Problem NetworkOnMainThreadException

Android ma jeden główny wątek (Main Thread / UI Thread), który odpowiada za rysowanie interfejsu i obsługę dotyku.

Jeśli ten wątek zostanie zablokowany nawet na 16ms - aplikacja 'przycina'. Jeśli zablokuje się na 5 sekund - Android zabija aplikację.

Operacje sieciowe mogą trwać od 100ms do kilku sekund. Dlatego Android od wersji 3.0 dosłownie rzuca wyjątek NetworkOnMainThreadException jeśli spróbujesz wykonać żądanie HTTP na głównym wątku.

Rozwiązanie: 'suspend fun' + Coroutines. Funkcja oznaczona suspend może zostać 'zawieszona' bez blokowania wątku.
Dispatcher.IO uruchamia ją na puli wątków IO (do 64 jednocześnie). Gdy dane wrócą, Dispatcher.Main aktualizuje UI.
Dla programisty wygląda jak kod synchroniczny - bez callbacków, bez AsyncTask, bez boilerplate.

4. Konfiguracja projektu PokeApp

Zaczynamy od nowego projektu Android Studio (Empty Activity, Kotlin, Jetpack Compose, minSdk 26). Projekt nazywamy PokeApp. W tej sekcji dodamy wszystkie niezbędne zależności i uprawnienia.

4.1. Zależności w libs.versions.toml

gradle/libs.versions.toml

```
# gradle/libs.versions.toml
[versions]
retrofit          = "2.11.0"     # Retrofit - główna biblioteka HTTP klienta
okhttp            = "4.12.0"     # OkHttp - warstwa transportu HTTP
coil              = "2.7.0"      # Coil - ładowanie i cachowanie obrazów

[libraries]
# Retrofit + konwerter JSON
retrofit-core     = { module = "com.squareup.retrofit2:retrofit", version.ref = "retrofit" }
retrofit-gson     = { module = "com.squareup.retrofit2:converter-gson", version.ref = "retrofit" }

# OkHttp + logger (TYLKO debug!)
okhttp-core       = { module = "com.squareup.okhttp3:okhttp", version.ref = "okhttp" }
okhttp-logging    = { module = "com.squareup.okhttp3:logging-interceptor", version.ref = "okhttp" }

# Coil - AsyncImage w Compose
coil-compose      = { module = "io.coil-kt:coil-compose", version.ref = "coil" }
```

4.2. Plik build.gradle.kts (app)

app/build.gradle.kts

```
// app/build.gradle.kts
dependencies {
    // Retrofit + Gson
    implementation(libs.retrofit.core)
    implementation(libs.retrofit.gson)

    // OkHttp - wymagany przez Retrofit
    implementation(libs.okhttp.core)

    // Logger TYLKO dla debugowania - NIE trafi do release!
    debugImplementation(libs.okhttp.logging)   // <-- debugImplementation, NIE implementation!

    // Coil dla Jetpack Compose
    implementation(libs.coil.compose)
}
```

⚠ debugImplementation vs implementation - dlaczego to ważne?

Plik okhttp-logging-interceptor loguje PEŁNĄ treść żądań i odpowiedzi do Logcat - włącznie z tokenami autoryzacyjnymi, danymi osobowymi użytkownika i wszelkimi poufnymi informacjami.

debugImplementation = biblioteka zostanie dołączona TYLKO do buildu debug (APK które instalujesz w trakcie developmentu).

implementation = biblioteka trafi do buildu release (APK który publikujesz w Google Play).

Gdybyś użył implementation, twój logger trafiłby do produkcyjnej aplikacji i każdy mógłby obserwować ruch sieciowy twojej aplikacji w Logcat. To poważna luka bezpieczeństwa!

Zasada: logging-interceptor zawsze debugImplementation. Nigdy implementation.

4.3. Uprawnienie INTERNET w AndroidManifest.xml

AndroidManifest.xml

```
<!-- app/src/main/AndroidManifest.xml -->
<manifest xmlns:android="http://schemas.android.com/apk/res/android">

    <!-- Uprawnienie do dostępu do internetu - WYMAGANE dla żądań HTTP -->
    <!-- To jest 'normal permission' - Android nadaje je automatycznie, bez pytania użytkownika -->
    <uses-permission android:name="android.permission.INTERNET" />

    <application ...>
        <!-- Jeśli API używa HTTP (nie HTTPS): -->
        <!-- android:usesCleartextTraffic="true"  ← TYLKO DEV, nigdy produkcja! -->
    </application>
</manifest>
```

🔍 Normal Permission vs Dangerous Permission

Android dzieli uprawnienia na dwie kategorie:

NORMAL PERMISSIONS (np. INTERNET, VIBRATE) - dostęp do danych/funkcji o niskim ryzyku dla prywatności.
System nadaje je automatycznie przy instalacji. Użytkownik nie widzi dialogu z prośbą o zgodę.
Wystarczy zadeklarować w Manifeście i biblioteka może z nich korzystać.

DANGEROUS PERMISSIONS (np. CAMERA, READ_CONTACTS, LOCATION) - dostęp do wrażliwych danych.
Wymagają jawnej zgody użytkownika w czasie działania aplikacji (Runtime Permission).
Musisz użyć ActivityResultContracts.RequestPermission() i obsłużyć odpowiedź.

INTERNET jest normal permission - dlatego nie widzisz dialogu 'Zezwól na dostęp do internetu'.

5. Modele danych - DTO i Domain Model

Jednym z ważniejszych wzorców architektonicznych, który zastosujemy w PokeApp, jest rozdział między modelem sieciowym (DTO) a modelem domenowym. Zrozumienie tego rozdziału chroni aplikację przed kruchością na zmiany w API.

🍽 Analogia: paczka kurierska

Wyobraź sobie, że zamawiasz książkę przez internet.

OPAKOWANIE KURIERSKIE (DTO) = Data Transfer Object
Paczka jest zoptymalizowana pod kątem transportu: posiada kod kreskowy, adres w formacie kuriera, może być zabezpieczona folią bąbelkową lub styropianem. To co jest w środku, nie interesuje kuriera.
DTO to dokładna kopia struktury JSON z serwera - wszystkie pola, dokładnie takie same nazwy.

ZAWARTOŚĆ (Domain Model)
To jest właściwa książka - obiekt, z którym pracujesz: czytasz, zaznaczasz strony, oceniasz.

Domain Model to obiekt zoptymalizowany pod kątem Twojej aplikacji: pola w camelCase, typy Kotlin, przeliczone jednostki (decymetry → metry), nullable zamienione na sensowne domyślne.

Jeśli kurier (API) zmieni format opakowania (zmieni nazwę pola z 'base_exp' na 'base_experience'), wystarczy zmienić DTO i mapper. Reszta aplikacji (Domain Model, UI) nie wie nic o tej zmianie.

5.1. Struktura katalogów projektu

Struktura katalogów PokeApp

```
app/src/main/java/pl/edu/pam/pokeapp/
├── data/
│   ├── remote/
│   │   ├── dto/
│   │   │   ├── PokemonListDto.kt      // Odpowiedź listy: count + results[]
│   │   │   ├── PokemonListItemDto.kt  // Element listy: name + url
│   │   │   └── PokemonDetailDto.kt    // Szczegóły: id, name, height, sprites...
│   │   └── api/
│   │       ├── PokemonApiService.kt   // Interfejs Retrofit (@GET, @Path)
│   │       └── RetrofitClient.kt      // Singleton z OkHttp + Gson
│   ├── local/
│   │   ├── PokemonDatabase.kt         // Room database (z Lab 3 - analogia)
│   │   └── PokemonDao.kt              // DAO dla cache offline
│   └── repository/
│       └── PokemonRepository.kt       // Offline-First: Room + Retrofit
├── domain/
│   ├── model/
│   │   └── Pokemon.kt                 // Domain Model - optymalny dla UI
│   └── mapper/
│       └── PokemonMapper.kt           // DTO → Domain Model
└── ui/
    ├── list/
    │   ├── PokemonListScreen.kt        // LazyVerticalGrid + AsyncImage
    │   └── PokemonListViewModel.kt     // StateFlow<UiState>
    └── detail/
        ├── PokemonDetailScreen.kt
        └── PokemonDetailViewModel.kt
```

5.2. DTO - Data Transfer Object

PokemonDetailDto.kt - dokładna kopia struktury JSON	Wyjaśnienie
`data class PokemonDetailDto(`	data class bo Gson potrzebuje equals/hashCode do cacheowania
`  val id: Int,`	ID Pokemona - zawsze Int, mieści się w zakresie
`  val name: String,`	Nazwa małymi literami - serwer zwraca 'pikachu', nie 'Pikachu'
`  @SerializedName("base_experience")`	@SerializedName mapuje klucz JSON na pole Kotlin
`  val baseExperience: Int?,`	Nullable! Niektóre Pokemony nie mają base_experience w API
`  val height: Int,`	UWAGA: w decymetrach! 4 = 0.4m. Przelicz w mapperze.
`  val weight: Int,`	UWAGA: w hektogramach! 60 = 6.0kg. Przelicz w mapperze.
`  val sprites: SpritesDto,`	Zagnieżdżony obiekt - osobna data class
`  val types: List<TypeSlotDto>,`	Lista typów, np. [Electric] dla Pikachu
`  val abilities: List<AbilitySlotDto>`	Lista zdolności - część jest ukryta (is_hidden = true)
`)`	
`data class SpritesDto(`	Osobna klasa dla zagnieżdżonego obiektu sprites
`  @SerializedName("front_default")`	Klucz JSON to 'front_default' (snake_case)

PokemonDetailDto.kt - dokładna kopia struktury JSON	Wyjaśnienie
`  val frontDefault: String?`	Nullable! Niektóre formy Pokemona nie mają sprite'a
`)`	

5.3. Domain Model; zoptymalizowany dla UI

Pokemon.kt - Domain Model	Wyjaśnienie
`data class Pokemon(`	Czysty model domenowy - bez adnotacji Gson, bez @SerializedName
`  val id: Int,`	ID - identyczne jak w DTO
`  val name: String,`	Nazwa z dużej litery ('Pikachu') - przeliczone w mapperze
`  val heightMeters: Double,`	W METRACH - czytelna jednostka dla UI. Pole nazwa jasna.
`  val weightKg: Double,`	W KILOGRAMACH - przeliczone z hektogramów w mapperze.
`  val imageUrl: String?,`	Nullable - UI obsłuży brak obrazka przez placeholder Coil
`  val types: List<String>,`	Lista nazw typów (String), nie obiektów - łatwe do wyświetlenia
`  val abilities: List<String>,`	Tylko widoczne zdolności - filtrowane w mapperze
`  val baseExperience: Int`	Fallback 0 zamiast nullable - UI nie musi sprawdzać null
`)`	

5.4. Mapper; konwersja DTO → Domain Model

PokemonMapper.kt - extension functions	Wyjaśnienie
`fun PokemonDetailDto.toDomainModel(): Pokemon {`	Extension function - wywołujesz dto.toDomainModel()
`  return Pokemon(`	Tworzymy Domain Model z danych DTO
`    id   = this.id,`	ID kopiujemy bezpośrednio
`    name = this.name.replaceFirstChar {`	Zmień pierwszą literę na dużą: 'pikachu' → 'Pikachu'
`      it.uppercaseChar() },`	replaceFirstChar działa poprawnie z Unicode
`    heightMeters = this.height / 10.0,`	API zwraca decymetry. /10.0 → metry (4 → 0.4m)
`    weightKg     = this.weight / 10.0,`	API zwraca hektogramy. /10.0 → kilogramy (60 → 6.0kg)
`    imageUrl     = this.sprites.frontDefault,`	Null jeśli brak sprite'a - Coil obsłuży placeholder
`    types        = this.types.map {`	Mapujemy listę TypeSlotDto na listę nazw (String)
`      it.type.name.replaceFirstChar {`	Każdy typ z dużej litery: 'electric' → 'Electric'
`        it.uppercaseChar() } },`	
`    abilities    = this.abilities`	Filtrujemy i mapujemy listę zdolności
`      .filter { !it.isHidden }`	Pomijamy ukryte zdolności (is_hidden = true)
`      .map { it.ability.name },`	Tylko nazwa zdolności jako String
`    baseExperience = this.baseExperience ?: 0`	Elvis operator: null → 0 (bezpieczny fallback)
`  )`	
`}`	

6. Retrofit; interfejs API

🍽 Analogia: magiczny asystent

Wyobraź sobie asystenta, któremu możesz dać kartkę z prostymi notatkami:
'Pobierz listę pokemonów - 20 na raz' → GET /api/v2/pokemon?limit=20
'Pobierz pokemona o imieniu pikachu' → GET /api/v2/pokemon/pikachu

Retrofit jest właśnie tym asystentem. Ty piszesz interfejs z adnotacjami (notatki),

a Retrofit w czasie działania aplikacji tworzy prawdziwą implementację (wykonuje żądania). Technicznie robi to przez mechanizm Dynamic Proxy - generuje bytecode implementacji interfejsu w czasie działania, bez konieczności pisania kodu przez programistę.

Nie musisz pisać: 'otwórz połączenie, zbuduj URL, dodaj parametry, obsłuż timeout'. Wystarczą adnotacje - Retrofit zajmie się resztą.

6.1. Adnotacje Retrofit; tabela referencyjna

Adnotacja	Znaczenie i przykład użycia
`@GET("path")`	Żądanie HTTP GET. Ścieżka jest dołączana do baseUrl. Przykład: @GET("pokemon")
`@POST("path")`	Żądanie HTTP POST. Ciało żądania przekazujesz przez @Body.
`@Path("name")`	Wartość dynamiczna w ścieżce. @GET("pokemon/{name}") + @Path("name") val n: String → /pokemon/pikachu
`@Query("key")`	Parametr zapytania (po ?). @Query("limit") val limit: Int → ?limit=20
`@Body`	Ciało żądania (dla POST/PUT). Gson serializuje obiekt do JSON automatycznie.
`@Header("Key")`	Nagłówek HTTP. Przydatny dla tokenów: @Header("Authorization") val token: String
`suspend`	Słowo kluczowe Kotlin - funkcja może być 'zawieszona'. Wymagane dla współpracy z Coroutines.

6.2. Interfejs PokemonApiService

PokemonApiService.kt	Wyjaśnienie
`interface PokemonApiService {`	Interfejs - Retrofit wygeneruje implementację automatycznie (Dynamic Proxy)
`  @GET("pokemon")`	Metoda HTTP GET, ścieżka 'pokemon' → pełny URL: baseUrl + pokemon
`  suspend fun getPokemonList(`	suspend: funkcja może być zawieszona (Coroutines). BEZ suspend = crash!
`    @Query("limit") limit: Int = 20,`	@Query zamienia parametr na ?limit=20 w URL
`    @Query("offset") offset: Int = 0`	@Query offset → ?offset=0 (paginacja od początku)
`  ): PokemonListDto`	Zwraca DTO z listą pokemonów. Gson automatycznie parsuje JSON.
`  @GET("pokemon/{name}")`	{name} to placeholder - wartość wstrzykuje @Path
`  suspend fun getPokemonDetail(`	Ponownie suspend - operacja I/O
`    @Path("name") name: String`	@Path("name") = wartość wstawiana zamiast {name} w URL
`  ): PokemonDetailDto`	Zwraca szczegółowy DTO Pokemona
`}`	

6.3. RetrofitClient; konfiguracja singletona

RetrofitClient.kt	Wyjaśnienie
`object RetrofitClient {`	object = Singleton Kotlin. Jeden egzemplarz na całą aplikację.
`  private const val BASE_URL =`	Stała - URL bazowy. MUSI kończyć się '/' !!
`    "https://pokeapi.co/api/v2/"`	Bez trailing slash → IllegalArgumentException w runtime!
`  private val logger by lazy {`	by lazy = inicjalizacja przy pierwszym użyciu, nie przy starcie
`    HttpLoggingInterceptor().apply {`	HttpLoggingInterceptor z OkHttp - loguje żądania do Logcat

RetrofitClient.kt	Wyjaśnienie
`level = HttpLoggingInterceptor`	Poziom BODY = loguj nagłówki + pełne ciało żądania/odpowiedzi
`.Level.BODY } },`	W BuildConfig.DEBUG blokujemy to w release (patrz niżej)
`private val client by lazy {`	OkHttpClient - zarządza połączeniami, cache, interceptorami
`OkHttpClient.Builder()`	Builder pattern - każda metoda zwraca Builder do dalszej konfiguracji
`.addInterceptor(logger)`	Dodajemy logger (tylko dla DEBUG - tu dla czytelności)
`.connectTimeout(10, TimeUnit.SECONDS)`	Max 10s na nawiązanie połączenia TCP
`.readTimeout(15, TimeUnit.SECONDS)`	Max 15s na odczytanie odpowiedzi serwera
`.build() },`	Budujemy finalny klient
`val api: PokemonApiService by lazy {`	Właściwy interfejs API - inicjalizowany leniwie
`Retrofit.Builder()`	Tworzymy Retrofit przez Builder
`.baseUrl(BASE_URL)`	Podstawowy URL (MUSI być z '/')
`.client(client)`	Przekazujemy skonfigurowany OkHttpClient
`.addConverterFactory(`	Fabryka konwerterów - tu Gson (JSON ↔ Kotlin)
`GsonConverterFactory.create())`	GsonConverterFactory automatycznie (de)serializuje JSON
`.build()`	Buduje Retrofit
`.create(PokemonApiService::class.java) }`	Dynamic Proxy: generuje implementację interfejsu
`}`	Koniec obiektu RetrofitClient

⚠ baseUrl MUSI kończyć się '/' **---- zasada absolutna! ----**

Retrofit konkatenuje baseUrl i ścieżkę z adnotacji @GET przez proste sklejenie stringów.

POPRAWNIE: baseUrl = "https://pokeapi.co/api/v2/" + @GET("pokemon") = /api/v2/pokemon
NIEPOPRAWNIE: baseUrl = "https://pokeapi.co/api/v2" + @GET("pokemon") = /api/v2pokemon (BŁĄD!)

Brak trailing slash powoduje IllegalArgumentException: 'baseUrl must end in /' rzucany przy starcie aplikacji.
Ten błąd jest jednym z **najczęściej popełnianych przez początkujących**!

🔍 Co to jest Interceptor? - wzorzec Chain of Responsibility

Interceptor w OkHttp działa jak seria filtrów przez które przechodzi każde żądanie i odpowiedź.
Każdy interceptor może: modyfikować żądanie przed wysłaniem, modyfikować odpowiedź po otrzymaniu, logować dane, dodawać nagłówki (np. tokeny), obsługiwać odświeżanie tokenów.

Wzorzec: Chain of Responsibility - każdy interceptor wywołuje chain.proceed(request) aby przekazać żądanie dalej. Może zatrzymać chain (np. gdy token wygasł) lub kontynuować (logger tylko obserwuje).

Przykłady użycia: AuthInterceptor (dodaje Bearer token), LoggingInterceptor (debugowanie), CacheInterceptor (własna logika cache), RetryInterceptor (automatyczne ponawianie przy błędach).

7. Obsługa błędów - sealed class Result<T>

Komunikacja sieciowa może zakończyć się na wiele sposobów: sukcesem, błędem sieci, błędem serwera, błędem parsowania, timeout'em... Musimy mieć solidny mechanizm obsługi każdego z tych przypadków.

7.1. Katalog możliwych błędów sieciowych

Typ błędu	Przyczyna i jak się objawia
`Brak sieci (IOException)`	Urządzenie offline. OkHttp rzuca IOException. Sprawdź ConnectivityManager.
`Timeout (SocketTimeoutException)`	Serwer nie odpowiada w czasie readTimeout/connectTimeout.
`HTTP 4xx (HttpException)`	Błąd po stronie klienta. Retrofit rzuca HttpException z kodem błędu.
`HTTP 5xx (HttpException)`	Błąd po stronie serwera. HttpException.code() zwraca 500-599.
`Błąd parsowania (JsonSyntaxException)`	JSON nie pasuje do data class. Brakuje pola, zły typ, brak @SerializedName.
`Błąd SSL (SSLException)`	Problem z certyfikatem HTTPS. Często przy połączeniach z HTTP (nie S).
`Anulowanie Coroutine (CancellationException)`	ViewModel zniszczony przed końcem żądania. NIE łap tego wyjątku!

Dlaczego nie zwykły try-catch w ViewModelu? Bo… 3 osobne problemy

1. BRAK SEMANTYKI - try-catch zwraca Unit lub rzuca. Nie wiadomo czy funkcja zakończyła się sukcesem czy błędem bez czytania kodu wewnątrz. Result<T> w typie zwracanym jasno komunikuje możliwość błędu.
2. BRAK EXHAUSTIVE CHECKING - kompilator Kotlin nie wymusza obsługi wyjątków (w przeciwieństwie do Javy). Możesz zapomnieć obsłużyć IOException. Z sealed class Result, kompilator wymusi obsługę wszystkich stanów.
3. BRAK CZYSTOŚCI - funkcja która 'może rzucić' to niejawna umowa. Result<T> jako typ zwracany to jawna umowa: 'ta funkcja może się nie powieść - sprawdź wynik'.

🍽 Analogia: sygnalizacja świetlna

sealed class Result jest jak sygnalizacja świetlna - zawsze wiesz, który stan jest aktywny:

ZIELONE = Result.Success<T> → masz dane, możesz jechać (renderuj UI z danymi)
CZERWONE = Result.Error → stop, coś poszło nie tak (pokaż komunikat błędu)
ŻÓŁTE = Result.Loading → poczekaj, operacja w toku (pokaż spinner)

Nie ma możliwości, żeby sygnalizacja była jednocześnie zielona i czerwona.
Tak samo sealed class gwarantuje, że wynik jest DOKŁADNIE jednym ze znanych stanów.
when (result) na sealed class wymusza obsługę WSZYSTKICH możliwości - kompilator pilnuje.

7.2. Implementacja sealed class Result<T>

Result.kt	Wyjaśnienie
`sealed class Result<out T> {`	sealed: tylko podklasy w tym pliku. out T: kowariantny (bezpieczny do użycia jako Result<Any>)
`  data class Success<T>(`	data class: automatyczny equals/hashCode/toString
`    val data: T`	Dane zwrócone przez API - w domenowym typie T
`  ) : Result<T>()`	Success dziedziczy po Result<T>
`  data class Error(`	Nothing: typ bez wartości - Error nie zawiera T
`    val message: String,`	Czytelny komunikat błędu dla użytkownika lub logowania
`    val code: Int? = null`	Kod HTTP (400, 404, 500) - null jeśli błąd nie jest HTTP
`  ) : Result<Nothing>()`	Nothing pozwala używać Error zamiast Result<T> dla dowolnego T

Result.kt	Wyjaśnienie
`  object Loading : Result<Nothing>()` `}`	object (nie data class) - Loading nie ma danych, jeden egzemplarz

7.3. Funkcja safeApiCall - centralna obsługa błędów

safeApiCall.kt	Wyjaśnienie
`suspend fun <T> safeApiCall(`	suspend: wymagane bo wywołuje suspend funkcje Retrofit
`  apiCall: suspend () -> T`	Lambda z suspend call - parametr funkcyjny
`): Result<T> {`	Zawsze zwraca Result - nigdy nie rzuca
`  return try {`	try-catch jako fallback bezpieczeństwa
`    Result.Success(apiCall())`	Sukces: wywołaj API i zawiń wynik w Success
`  } catch (e: HttpException) {`	Retrofit rzuca HttpException dla kodów 4xx i 5xx
`    Result.Error(`	Mapujemy na Result.Error z kodem HTTP
`      message = "Błąd serwera: ${e.code()}",`	e.code() = 404, 500 itp.
`      code    = e.code())`	Zapisujemy kod dla bardziej szczegółowej obsługi w VM
`  } catch (e: IOException) {`	IOException = brak sieci, timeout, SSL, DNS
`    Result.Error("Brak połączenia z internetem")`	Przyjazny komunikat dla użytkownika
`  } catch (e: JsonSyntaxException) {`	Gson nie mógł sparsować odpowiedzi - błąd w DTO
`    Result.Error("Błąd parsowania danych")`	Wskazówka: sprawdź DTO i @SerializedName
`  }  // CancellationException celowo NIE jest łapany`	Coroutine musi móc zostać anulowana!
`}`	

7.4. Obsługa Result w ViewModelu i UI

Obsługa Result w ViewModel i Compose

```
// W ViewModelu - po pobraniu danych
viewModelScope.launch {
  _uiState.value = UiState.Loading  // Pokaż spinner
  val result = safeApiCall {         // Wywołaj API bezpiecznie
    RetrofitClient.api.getPokemonDetail(name)
  }
  _uiState.value = when (result) {  // Kompilator wymusi obsługę WSZYSTKICH przypadków!
    is Result.Success -> UiState.Success(result.data.toDomainModel())
    is Result.Error   -> UiState.Error(result.message)
    is Result.Loading -> UiState.Loading  // Normalnie nie trafia tu przez ViewModel
  }
}
// W Compose UI - obsługa każdego stanu
when (val state = uiState.collectAsStateWithLifecycle().value) {
  is UiState.Loading -> CircularProgressIndicator()  // Spinner
  is UiState.Success -> PokemonContent(state.pokemon) // Dane
  is UiState.Error   -> ErrorScreen(                  // Błąd
    message = state.message,
    onRetry = viewModel::reload
  )
}
```

8. Coil - ładowanie obrazów

Pokemon API zwraca URL-e do obrazków sprite'ów. Nie możemy po prostu użyć standardowego Image z Compose - Compose nie obsługuje ładowania obrazów z sieci. Do tego celu służy biblioteka Coil, zoptymalizowana specjalnie dla Androida i Kotlin Coroutines.

? Dlaczego nie można użyć Image(bitmap) z URL?

Komponent Image w Jetpack Compose przyjmuje tylko lokalne zasoby (painterResource, bitmapResource) lub wcześniej załadowane obiekty Bitmap. Nie ma wbudowanego mechanizmu pobierania obrazu z URL.

Naiwne rozwiązanie: pobierz bajty URL ręcznie w LaunchedEffect → BitmapFactory.decodeByteArray → Image.
Problem: brak cache (pobierasz za każdym razem), brak obsługi błędów, brak placeholdera, brak anulowania przy zniszczeniu komponentu, brak dekodowania na tle.

Coil rozwiązuje wszystkie te problemy: trójpoziomowy cache, placeholdery, obsługa błędów, anulowanie przy zniszczeniu kompozycji, wsparcie dla transformacji (roundedCorners, blur itp.).

🔍 Trójpoziomowy cache Coil. Jak to działa i dlaczego tak.

Coil sprawdza trzy poziomy cache w kolejności (od najszybszego do najwolniejszego):

1. MEMORY CACHE (pamięć RAM) - obrazy zdekodowane do Bitmap. Dostęp natychmiastowy (<1ms). Ograniczenie: resetowany przy zamknięciu aplikacji. Wielkość: ~25% dostępnej RAM.

2. DISK CACHE (pamięć masowa) - surowe bajty pliku obrazu zapisane na dysku. Dostęp ~5-50ms. Przetrwa restart aplikacji. Coil używa domyślnie 10% dostępnej przestrzeni dyskowej.

3. SIEĆ - jeśli oba cache są puste, pobierany jest obraz z URL. Dostęp ~100ms-kilka sekund. Pobrane bajty są zapisywane do Disk Cache, zdekodowany Bitmap do Memory Cache.

Praktyczne znaczenie: po pierwszym pobraniu lista 20 Pokemonów ładuje się niemal natychmiastowo. Coil automatycznie unieważnia cache jeśli serwer zwróci nagłówek Cache-Control: no-cache.

8.1. AsyncImage - podstawowe użycie

AsyncImage w Compose	Wyjaśnienie
`AsyncImage(`	Composable z biblioteki coil-compose - ładuje obraz asynchronicznie
`  model = pokemon.imageUrl,`	URL do pobrania. Może być String, Uri, File, Int (drawable). Null = error.
`  contentDescription =`	WYMAGANE dla dostępności! TalkBack przeczyta ten opis niewidomym.
`    "Sprite ${pokemon.name}",`	Opisowy tekst, nie 'obraz' ale co przedstawia
`  placeholder =`	Composable/Drawable wyświetlany PODCZAS ładowania
`    painterResource(R.drawable.ic_pokeball),`	Pokeball jako placeholder - tematycznie pasujący
`  error =`	Composable/Drawable wyświetlany gdy ładowanie się NIE powiodło
`    painterResource(R.drawable.ic_error),`	Ikona błędu - użytkownik wie, że coś poszło nie tak
`  contentScale = ContentScale.Fit,`	Jak skalować obraz: Fit (zachowaj proporcje), Crop (wypełnij), FillBounds
`  modifier = Modifier`	Standardowy modifier Compose - rozmiar, padding, kształt itp.

AsyncImage w Compose	Wyjaśnienie
`.size(96.dp)`	Sprite Pokemona ma małą rozdzielczość - 96dp to dobry rozmiar
`.clip(RoundedCornerShape(8.dp))`	Zaokrąglone rogi dla estetyki - Coil obsługuje transformacje
`)`	

8.2. AsyncImage vs SubcomposeAsyncImage

Composable	Kiedy używać
`AsyncImage`	DOMYŚLNY WYBÓR. Prosty i wydajny. Obsługuje placeholder/error jako Painter. Brak dostępu do stanu ładowania. Używaj dla list (LazyColumn, LazyVerticalGrid) gdzie wydajność jest kluczowa.
`SubcomposeAsyncImage`	Gdy potrzebujesz niestandardowego UI podczas ładowania (np. animacja, skeleton loader) lub po błędzie. Udostępnia state (loading/success/error) jako kompozycję. Wolniejszy - unika w listach.

⚠ contentDescription jest obowiązkowy; kwestia dostępności (Accessibility)

contentDescription = null wyłącza opis dla TalkBack (screen readera dla niewidomych użytkowników).
Jest to dopuszczalne TYLKO dla obrazów czysto dekoracyjnych (separatory, wzory tła).

Sprite Pokemona NIE jest dekoracyjny - przekazuje informację o wyglądzie Pokemona.
Ustaw: contentDescription = "Sprite ${pokemon.name}"

Google wymaga poprawnej obsługi TalkBack dla aplikacji w Google Play (zgodność z WCAG 1.1.1).
Testy automatyczne (Espresso) domyślnie sprawdzają brak contentDescription i zgłaszają błąd.

9. Repository - strategia offline-first

Dobra aplikacja mobilna działa nawet bez dostępu do internetu. Strategia Offline-First oznacza, że Room Database jest jedynym źródłem prawdy dla UI - sieć służy tylko do aktualizacji Room.

? Po co Offline-First? - 3 powody

1. METRO I TUNEL - użytkownicy często korzystają z aplikacji w miejscach bez zasięgu.
   Bez cache aplikacja pokaże pustą listę lub błąd zamiast poprzednich danych.

2. OSZCZĘDNOŚĆ BATERII I DANYCH - każde żądanie sieciowe kosztuje energię i transfer.
   Cache redukuje ilość żądań dla danych które rzadko się zmieniają (Pokemony są stałe!).

3. RATE LIMITING - PokeAPI pozwala na 100 żądań na minutę. Bez cache przy szybkim scrollowaniu łatwo przekroczyć limit i dostać 429 Too Many Requests.

9.1. Strategia Offline-First - sekwencja

#	Przepływ danych w PokemonRepository (offline-first)
1	Natychmiastowo emit() z Room - UI dostaje dane z cache (jeśli istnieją). Użytkownik widzi poprzednie dane od razu, bez czekania na sieć.
2	W tle: safeApiCall do PokeAPI - pobierz świeże dane. Wykonaj na Dispatchers.IO (nie blokuj UI).
3	Jeśli sukces: zapisz do Room (upsert) - Room Flow automatycznie emituje nowe dane do UI.
4	UI aktualizuje się automatycznie - Compose przerysowuje zmienione elementy bez jawnego wywołania.
5	Jeśli błąd sieciowy, a cache NIE pusty - zachowaj stare dane, pokaż dyskretny Snackbar 'Dane mogą być nieaktualne'.

#	Przepływ danych w PokemonRepository (offline-first)
6	Jeśli błąd sieciowy, a cache PUSTY - brak danych, pokaż ErrorScreen z przyciskiem 'Spróbuj ponownie'.

9.2. Implementacja PokemonRepository

PokemonRepository.kt - Offline-First (fragment)

```
class PokemonRepository(                                    // Nie singleton - wstrzykuj przez konstruktor
  private val api: PokemonApiService,                       // Zależność sieciowa (Retrofit)
  private val dao: PokemonDao                               // Zależność lokalna (Room)
) {
  // Flow<List<Pokemon>> - emituje za każdym razem gdy Room się zmieni
  fun getPokemonList(): Flow<List<Pokemon>> = flow {
    // KROK 1: Natychmiastowo emit z Room (może być pusta lista)
    emitAll(dao.getAllPokemon().map {                       // Flow z Room
      it.map { entity -> entity.toDomainModel() }           // Mapuj Entity → Domain
    })
  }.onStart {                                               // Przed pierwszą emisją z Room:
    // KROK 2: Fetch z API w tle
    val result = safeApiCall { api.getPokemonList(limit = 20) }
    if (result is Result.Success) {
      // KROK 3: Zapisz do Room → Room Flow automatycznie wyemituje
      dao.upsertAll(result.data.results.map { it.toEntity() })
    }
    // KROK 4: Błąd obsługujemy w ViewModel przez dodatkowy Flow stanu
  }.flowOn(Dispatchers.IO)                                  // Cały flow na wątku IO
}
```

10. Diagnostyka i debugowanie sieci

Zanim napiszesz pierwszą linię kodu integrującego API, sprawdź API ręcznie za pomocą narzędzi zewnętrznych. Oszczędzi Ci to wielu godzin debugowania błędów, które tkwią w DTO, a nie w kodzie.

10.1. Logi OkHttp w Logcat

Przykładowe logi OkHttp (Logcat, tag: OkHttp)

```
// ✓ UDANE ŻĄDANIE - co zobaczysz w Logcat:
--> GET https://pokeapi.co/api/v2/pokemon?limit=20
--> END GET
<-- 200 OK https://pokeapi.co/api/v2/pokemon?limit=20 (342ms)
Content-Type: application/json; charset=utf-8
{"count":1302,"next":"...offset=20","results":[{"name":"bulbasaur",...}]}
<-- END HTTP (1200-byte body)

// ✗ BŁĄD 404 - zasób nie istnieje:
--> GET https://pokeapi.co/api/v2/pokemon/abcdef
<-- 404 Not Found (89ms)

// ✗ BRAK SIECI - IOException w Logcat:
java.net.UnknownHostException: Unable to resolve host 'pokeapi.co'
  (sprawdź: czy emulator ma internet? Czy wpisałeś poprawny URL?)
```

10.2. Narzędzia diagnostyczne

Narzędzie	Do czego służy i kiedy używać
`Logcat + OkHttp Logger`	Podgląd każdego żądania i odpowiedzi w Android Studio. Używaj zawsze podczas dewelopmentu. Filtruj po tagu 'OkHttp'.
`Insomnia / Postman`	Testuj API PRZED napisaniem DTO! Sprawdź strukturę JSON, kody odpowiedzi, parametry. Oszczędza godziny debugowania.

Narzędzie	Do czego służy i kiedy używać
`Android Studio Network Profiler`	Wizualizacja żądań w czasie. Pokaż waterfall, czas połączenia vs czas odpowiedzi. Przydatny do optymalizacji.
`Database Inspector`	Podgląd Room Database na żywo. Sprawdź, czy dane faktycznie trafiają do cache offline.
`Curl / HTTPie (terminal)`	Szybkie sprawdzenie API z linii poleceń. curl -s 'https://pokeapi.co/api/v2/pokemon/25' | python -m json.tool

Złota zasada: najpierw Insomnia, potem DTO

ZAWSZE sprawdź odpowiedź API zewnętrznym narzędziem przed napisaniem kodu:

1. Otwórz Insomnia lub Postman.
2. Wyślij GET https://pokeapi.co/api/v2/pokemon/25
3. Przejrzyj strukturę JSON - zanotuj wszystkie pola, typy, nullable.
4. Dopiero teraz pisz PokemonDetailDto z odpowiednimi typami i @SerializedName.

Często JSON zawiera pola, które wyglądają inaczej niż można się spodziewać.
Na przykład: height w PokeAPI to decymetry (nie centymetry, nie metry!).
Odkrycie tego w Insomnia zajmuje 30 sekund. Debugowanie w kodzie - kilka godzin.

11. Zadania do wykonania

Poniższe zadania tworzą kompletną aplikację PokeApp krok po kroku. Każde zadanie buduje na poprzednim - nie pomijaj kolejności. Przed przystąpieniem do każdego zadania upewnij się, że poprzednie przeszło weryfikację.

Zadanie 1 (20 pkt) - Konfiguracja projektu i pierwsze żądanie

1.1 Utwórz nowy projekt Android Studio: Empty Activity, Kotlin, Jetpack Compose, minSdk 26, nazwa: PokeApp.
1.2 Dodaj zależności Retrofit, OkHttp, Gson, Coil do libs.versions.toml i build.gradle.kts. Pamiętaj: okhttp-logging jako debugImplementation!
1.3 Dodaj uprawnienie INTERNET do AndroidManifest.xml.
1.4 Zaimplementuj RetrofitClient (object, by lazy, baseUrl z '/', GsonConverterFactory).
1.5 Utwórz PokemonApiService z metodą getPokemonList(limit, offset): PokemonListDto.
1.6 Wywołaj API z MainActivity.onCreate() (tymczasowo, tylko dla weryfikacji) i sprawdź w Logcat czy widzisz logi OkHttp z kodem 200 i JSON z listą Pokemonów.
WERYFIKACJA: Pokaż prowadzącemu logi Logcat z udanym żądaniem i odpowiedzią JSON.

Zadanie 2 (25 pkt) - DTO, Domain Model i obsługa błędów

2.1 Zaimplementuj pełne DTO: PokemonListDto, PokemonListItemDto, PokemonDetailDto ze wszystkimi polami i @SerializedName tam gdzie potrzeba.
2.2 Stwórz Domain Model Pokemon z polami: id, name, heightMeters, weightKg, imageUrl, types (List<String>), abilities (List<String>), baseExperience.
2.3 Zaimplementuj PokemonMapper z funkcją extension toDomainModel() przeliczającą jednostki (decymetry → metry, hektogramy → kilogramy) i filtrującą ukryte zdolności.
2.4 Zaimplementuj sealed class Result<T> z podklasami Success, Error, Loading.
2.5 Napisz suspend fun safeApiCall() z obsługą HttpException, IOException, JsonSyntaxException.
WERYFIKACJA: Testy jednostkowe dla mappera (JUnit 4, bez Androida). Sprawdź przeliczenia jednostek.

Zadanie 3 (35 pkt) - Ekrany UI: Lista i Szczegóły

3.1 Zaimplementuj PokemonListViewModel z StateFlow<UiState> (Loading/Success/Error).
Pobierz listę przez safeApiCall, zmapuj na Domain Model, wyemituj stan.
3.2 Zaimplementuj PokemonListScreen z LazyVerticalGrid (2 kolumny).
Każdy element: AsyncImage (sprite), nazwa, lista typów (Chip lub Text).
Obsłuż stany Loading (CircularProgressIndicator), Error (ErrorScreen z przyciskiem Retry).
3.3 Zaimplementuj ekran ErrorScreen (ikona, komunikat, przycisk 'Spróbuj ponownie').
3.4 Zaimplementuj PokemonDetailScreen z pełnymi danymi Pokemona:
Duży sprite (AsyncImage 200dp), nazwa, typ(y), wzrost, waga, zdolności, base experience.
3.5 Dodaj nawigację (NavHost) między ListScreen a DetailScreen z przekazaniem nazwy/ID.
WERYFIKACJA: Aplikacja działa na emulatorze - lista ładuje się, tap otwiera szczegóły, brak crash.

Zadanie 4 (20 pkt) - Offline-First, paginacja i diagnostyka

4.1 Dodaj Room Database (PokemonEntity, PokemonDao) zgodną z wzorcem z Lab 3.
PokemonDao musi mieć upsert (INSERT OR REPLACE) i Flow<List<PokemonEntity>>.
4.2 Zaimplementuj PokemonRepository z strategią Offline-First (emit z Room + fetch z API).
4.3 Zaimplementuj paginację: przycisk 'Załaduj więcej' lub automatyczne ładowanie przy
dotarciu do końca listy (LazyVerticalGrid z state.firstVisibleItemIndex).
4.4 Przetestuj tryb offline: wyłącz sieć w emulatorze (Settings → Network) i sprawdź,
że aplikacja wyświetla poprzednio załadowane dane z komunikatem 'Tryb offline'.
WERYFIKACJA: Demonstracja trybu offline prowadzącemu. Screenshot Database Inspector z danymi.

12. Kryteria oceniania

12.1. Punktacja zadań

Zadanie	Punkty	Co weryfikuje prowadzący
Zadanie 1: Konfiguracja	20 pkt	Logi OkHttp w Logcat - kod 200, JSON z listą Pokemonów. Zależności w gradle.
Zadanie 2: DTO + Mapper	25 pkt	Testy JUnit dla mappera. Poprawne przeliczenia jednostek. sealed class Result.
Zadanie 3: UI	35 pkt	Działająca lista + szczegóły. AsyncImage z placeholder. Obsługa błędów. Nawigacja.
Zadanie 4: Offline-First	20 pkt	Demo trybu offline. Database Inspector z danymi. Paginacja.
RAZEM	100 pkt	

12.2. Skala ocen

Ocena	Punkty	Wymagania
5.0	90–100 pkt	Wszystkie zadania kompletne. Offline-First działa. Testy mappera. Kod czysty i skomentowany.
4.5	80–89 pkt	Zadania 1–3 kompletne + paginacja lub Room cache. Drobne braki w UI.
4.0	70–79 pkt	Zadania 1–3 kompletne. Brak Offline-First. Obsługa błędów zaimplementowana.
3.5	60–69 pkt	Zadania 1–2 kompletne. Podstawowy UI listy działa. Brak szczegółów lub nawigacji.
3.0	50–59 pkt	Konfiguracja + pierwsze żądanie działa. DTO i mapper obecne, ale mogą mieć błędy.
2.0	0–49 pkt	Projekt nie kompiluje się lub nie pobiera danych z API.

13. Najczęstsze błędy i ich rozwiązania

Poniższa tabela zawiera rzeczywiste błędy, które studenci napotykają podczas realizacji tego ćwiczenia. Gdy coś nie działa, zacznij od sprawdzenia tej listy przed szukaniem w Internecie.

Komunikat błędu / objaw	Przyczyna i rozwiązanie
`CLEARTEXT communication not permitted for URL http://...`	Aplikacja próbuje połączyć się przez HTTP (nie HTTPS). Android 9+ blokuje to domyślnie. Rozwiązanie: użyj HTTPS. Jeśli absolutnie musisz HTTP (dev): dodaj android:usesCleartextTraffic="true" w Manifeście (TYLKO dev!).
`NetworkOnMainThreadException`	Wywołałeś żądanie sieciowe na głównym wątku UI. Rozwiązanie: użyj suspend fun + viewModelScope.launch. Nigdy nie wywołuj API bezpośrednio z onClick bez coroutine.
`JsonSyntaxException: Expected ... but was ...`	Niezgodność między typem JSON a typem Kotlin. Np. JSON zwraca null, ale pole Kotlin nie jest nullable. Rozwiązanie: dodaj ? do typu (String → String?), sprawdź @SerializedName.
`IllegalArgumentException: baseUrl must end in /`	Zapomniałeś o trailing slash w baseUrl. Rozwiązanie: zmień na "https://pokeapi.co/api/v2/" (z '/' na końcu).
`AsyncImage nie wyświetla obrazu (brak błędu)`	imageUrl jest null (pole sprites.front_default w DTO nie jest nullable lub Gson zwrócił null). Rozwiązanie: dodaj ? do pola w DTO. Sprawdź w Logcat czy URL jest poprawny. Dodaj logger interceptor.
`Dane z API nie trafiają do Room (Room pusty)`	Brak wywołania dao.upsertAll() po sukcesie API, lub Room entity ma inne pole @PrimaryKey niż DTO.id. Rozwiązanie: sprawdź mapper Entity. Użyj Database Inspector aby zobaczyć zawartość Room.
`Response 429 Too Many Requests`	Przekroczono limit PokeAPI (100 req/min). Bez cache każde przewinięcie listy wysyła nowe żądanie. Rozwiązanie: zaimplementuj Room cache (zadanie 4). Tymczasowo: dodaj Thread.sleep lub delay między żądaniami.
`Kotlin type mismatch: Int? vs Int`	Pole w DTO jest nullable (Int?) ale próbujesz użyć go tam gdzie wymagany jest Int. Rozwiązanie: użyj operatora Elvis: dto.baseExperience ?: 0 lub sprawdź null przed użyciem.

Instrukcja Laboratoryjna nr 4 - REST API, Retrofit i Coil
Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki
MobileHub
Następne ćwiczenie: Lab 5 - Hilt, Dependency Injection, testy jednostkowe i instrumentacyjne