POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

# ĆWICZENIE LABORATORYJNE - KOTLIN 1

## Konfiguracja środowiska deweloperskiego i pierwsza aplikacja Android

*Programowanie Aplikacji Mobilnych (Android / Kotlin / Jetpack Compose)*

## Spis treści



## 1. Instalacja Android Studio

### 1.1. Pobieranie i instalacja

Android Studio **Panda** jest oficjalnym IDE do tworzenia aplikacji Android. Pobieramy ze strony:

https://developer.android.com/studio

| # | Akcja | Szczegóły |
|---|---|---|
| 1 | **Pobierz instalator** | Wejdź na *developer.android.com/studio* → kliknij "*Download Android Studio*". Akceptujesz umowę licencyjną Google. Rozmiar: ~1.2 GB. |
| 2 | **Windows** | Uruchom .exe jako administrator. Wybierz opcję "*Android Studio*" + "*Android Virtual Device*". Ścieżka domyślna: *C:\Program Files\Android\Android Studio* |
| 3 | **macOS** | Otwórz *.dmg* → przeciągnij do Applications. Pierwsze uruchomienie: kliknij prawym → Open (ominięcie Gatekeeper). Symulator wymaga *Xcode Command Line Tools*. |
| 4 | **Linux (Ubuntu)** | Rozpakowujesz archiwum .tar.gz i uruchamiasz bin/studio.sh. Dodaj do **PATH** lub utwórz desktop shortcut.<br>sudo apt-get install -y libc6:i386 libncurses5:i386 libstdc++6:i386 lib32z1 |

| # | Akcja | Szczegóły |
|---|---|---|
| 5 | **Setup Wizard** | Przy pierwszym uruchomieniu: Standard setup → Next.<br>Pobiera SDK (Android 14,<br>Build Tools, Emulator).<br>Czas: 5-20 minut (zależnie od łącza). |

**Uwaga - problemy z instalacją**

Jeśli emulator nie uruchomi się - sprawdź, czy wirtualizacja sprzętowa jest włączona w BIOS/UEFI (Intel VT-x / AMD-V).
**Windows**: sprawdź "Hyper-V" w funkcjach Windows lub WHPX w ustawieniach AVD Manager. Korporacyjne laptopy mogą blokować VT-x - skontaktuj się z administratorem lub użyj urządzenia fizycznego.
**Linux**: dodaj użytkownika do grupy kvm: ***sudo usermod -aG kvm $USER*** (wymaga wylogowania)

## 1.2. Konfiguracja SDK i narzędzi

Po zainstalowaniu Android Studio otwieramy SDK Manager i weryfikujemy komponenty:

**Ścieżka:** File → Settings → Android SDK (lub Android Studio → Settings na macOS)

| | |
|---|---|
| **SDK Platforms** | Android 14 (API 34) zainstalowany Android 15 (API 35) zainstalowany |
| **SDK Tools** | Android SDK Build-Tools 35.x Android Emulator Android SDK Platform-Tools Intel x86 Emulator Accelerator (HAXM) - Windows/Mac |
| **SDK Command-line Tools** | Zainstaluj: latest version - umożliwia użycie sdkmanager z terminala |
| **Google Play Services** | Opcjonalne dla ćwiczenia, wymagane przy Google Maps / Firebase w projekcie semestralnym |

**Ścieżka SDK**

Zapamiętaj gdzie jest zainstalowany SDK, domyślnie powinno to być:
~/Library/Android/sdk (macOS) lub C:\Users\<user>\AppData\Local\Android\Sdk (Windows).
Możesz zmienić przez: File → Project Structure → SDK Location.
Zmienna ANDROID_HOME w systemie jest wymagana przez narzędzia CLI (np. Flutter, React Native).

# 2. Tworzenie wirtualnego urządzenia (AVD)

AVD *(Android Virtual Device)* to emulator systemu Android. Tworzymy go przez AVD Manager wbudowany w Android Studio.

## 2.1. Konfiguracja AVD Manager

| # | Akcja | Szczegóły i parametry |
|---|---|---|
| 1 | **Otwórz AVD Manager** | Tools → Device Manager (lub ikona telefonu w prawym górnym panelu → Manage Devices). Kliknij + Create Virtual Device. |
| 2 | **Wybierz hardware** | Wybierz **Pixel 9 Pro** z kategorii Phone. Rozmiar: 6.3", gęstość: 480 dpi. Kliknij Next. |
| 3 | **Wybierz system image** | Zakładka **Recommended** → pobierz/wybierz **API Level 35 (Android 15)** lub API Level 34 (Android 14). Wybierz architekturę x86_64 (dla PC) lub arm64 (Apple Silicon Mac). |
| 4 | **Konfiguracja AVD** | AVD Name: Pixel_9_Pro_API35<br>RAM: 2048 MB (zwiększ do 4096 MB jeśli komputer ma ≥ 16 GB RAM)<br>Internal Storage: 2048 MB<br>Enable Device Frame: tak<br>Startup orientation: Portrait |
| 5 | **Uruchom emulator** | Kliknij zielony trójkąt przy AVD. Pierwsze uruchomienie: 1-3 minuty. Patrz na stan w **'Device Manager'**. Emulator musi osiągnąć stan Running. |

## 2.2. Weryfikacja połączenia ADB

W terminalu Android Studio **View → Tool Windows → Terminal**

**Terminal**

```
HelloMobile/
adb devices

# Oczekiwany wynik:
# List of devices attached
# emulator-5554   device
```

Jeśli status to `offline` → zrestartuj emulator.

Jeśli status to `unauthorized` → odblokuj emulator i zaakceptuj klucz RSA.

| Przydatne polecenia ADB do zapamiętania | |
|---|---|
| **adb devices** | **Lista podłączonych urządzeń/emulatorów** |
| `adb kill-server, adb start-server` | Restart serwera ADB (gdy urządzenie „offline”) |
| `adb logcat` | Wyświetl logi systemowe (Ctrl+C aby przerwać) |
| `adb logcat --pid=<PID>` | Logi tylko dla konkretnej aplikacji (po PID) |
| `adb logcat -c` | Wyczyść bufor logów |
| `adb install nazwa_aplikacji.apk` | Zainstaluj plik APK |
| `adb install -r nazwa_aplikacji.apk` | Zainstaluj ponownie (nadpisz istniejącą) |
| `adb uninstall nazwa.pakietu` | Odinstaluj aplikację |

| | |
|---|---|
| `adb shell pm list packages` | Wyświetl listę zainstalowanych pakietów |
| `adb shell` | Otwórz powłokę systemu Android |
| `adb shell <polecenie>` | Wykonaj pojedyncze polecenie w shell |
| `adb shell getprop ro.build.version.release` | Sprawdź wersję Androida |
| `adb shell getprop ro.build.version.sdk` | Sprawdź poziom API |
| `adb shell getprop ro.product.model` | Sprawdź model urządzenia |
| `adb reverse tcp:8080 tcp:8080` | Port forwarding (przydatne przy lokalnym API) |
| `adb forward tcp:8080 tcp:8080` | Przekierowanie portu z urządzenia na komputer |
| `adb pull /ścieżka/plik.txt` | Skopiuj plik z urządzenia na komputer |
| `adb push plik.txt /sdcard/` | Skopiuj plik z komputera na urządzenie |
| `adb shell screencap -p /sdcard/screen.png` | Zrzut ekranu urządzenia |
| `adb shell screenrecord /sdcard/video.mp4` | Nagranie ekranu (max 180 s) |
| `adb reboot` | Restart urządzenia |

# 3. Tworzenie projektu Android

## 3.1. New Project Wizard

| # | Akcja | Szczegóły |
|---|---|---|
| 1 | **Nowy projekt** | File → New → New Project (lub **Ctrl+Shift+N** Windows / **⌘Shift+N** macOS) |
| 2 | **Szablon** | Wybierz kategorię **Phone and Tablet** → szablon **Empty Activity**. To domyślny punkt startowy dla Jetpack Compose. Kliknij Next. |
| 3 | **Konfiguracja** | Name: np. **HelloMobile** |
| | | Package name: pl.edu.prz.hellomobile (zmień na własną domenę odwróconą). |
| | | Save location: do folderu na dysku (unikaj spacji i polskich znaków w ścieżce!). |
| | | Language: Kotlin, minimum SDK: API 26 (Android 8.0) - pokrywa ~94% aktywnych urządzeń. |
| | | Build configuration language: Kotlin DSL (.kts) |
| 4 | **Finish** | Kliknij Finish. Gradle synchronizuje projekt. Pierwsze synchronizowanie: 2-5 minut. Patrz na pasek postępu na dole okna. |

**Konwencja nazewnictwa dla Package name**
Package name to unikalny identyfikator aplikacji w Google Play. Konwencja: odwrócona dome-

na + nazwa app, np. com.przykład.mojaapka. Dla ćwiczenia użyj formatu: pl.put.pam.imienazwisko (bez polskich znaków i spacji). Package name nie może być później zmieniony bez utraty danych użytkownika na Google Play.

## 3.2. Struktura wygenerowanego projektu

Po zakończeniu synchronizacji Gradle widzisz w panelu Project (**View → Tool Windows → Project**) następującą strukturę:

**Struktura projektu** — Kotlin

```
HelloMobile/
├── app/                                  ← moduł aplikacji
│   ├── src/
│   │   ├── main/
│   │   │   ├── java/pl/edu/.../
│   │   │   │   ├── MainActivity.kt       ← główna Activity (ENTRY POINT)
│   │   │   │   └── ui/theme/             ← pliki motywu Material3
│   │   │   ├── res/
│   │   │   │   ├── drawable/             ← grafika wektorowa (ikony)
│   │   │   │   ├── mipmap-*/             ← ikony aplikacji (różne gęstości)
│   │   │   │   ├── values/
│   │   │   │   │   ├── strings.xml       ← teksty (i18n)
│   │   │   │   │   ├── colors.xml        ← definicje kolorów
│   │   │   │   │   └── themes.xml        ← motywy Material3
│   │   │   │   └── xml/                  ← konfiguracja backup/network
│   │   │   └── AndroidManifest.xml       ← OBOWIĄZKOWY manifest aplikacji
│   │   └── test/ + androidTest/          ← testy jednostkowe i instrumentalne
│   ├── build.gradle.kts                  ← zależności modułu
│   └── proguard-rules.pro                ← reguły obfuskacji
├── gradle/
│   └── libs.versions.toml                ← Version Catalog (wersje zależności)
├── build.gradle.kts                      ← root build config
└── settings.gradle.kts                     ← deklaracja modułów
```

Najważniejsze pliki do zrozumienia na tym etapie:

| | |
|---|---|
| **AndroidManifest.xml** | Serce aplikacji. Deklaruje: Activity, uprawnienia (permissions), konfigurację sprzętową, meta-dane. Wymagany przez system Android. |
| **MainActivity.kt** | Punkt wejścia. Klasa dziedziczy po ComponentActivity.<br>Metoda setContent { } definiuje drzewo UI jako funkcje Composable. |
| **build.gradle.kts (app)** | Zależności (libraries), konfiguracja kompilacji: compileSdk, minSdk, versionCode, versionName. Każda biblioteka Jetpack dodawana tutaj. |
| **libs.versions.toml** | Centralne miejsce definicji wersji bibliotek. Zastępuje rozrzucone numery wersji po plikach gradle. Obowiązkowy standard od AGP 8. |
| **ui/theme/Theme.kt** | Konfiguracja motywu Material3: kolory (light/dark), typografia, kształty. Plik generowany automatycznie przez Compose. |

# 4. Analiza i modyfikacja kodu startowego

## 4.1. Analiza MainActivity.kt

Otwórz plik **app/src/main/java/.../MainActivity.kt**. Przeanalizuj każdą linię:

**MainActivity.kt** — **Kotlin**

```
package pl.edu.prz.hellomobile
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge     // fullscreen "edge to edge"
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.material3.Scaffold
import androidx.compose.material3.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.tooling.preview.Preview
import pl.edu.prz.hellomobile.ui.theme.HelloMobileTheme

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
 enableEdgeToEdge()                                                  // (1)
 setContent {                                                        // (2)
            HelloMobileTheme {                                       // (3)
         Scaffold(                                                   // (4)
                modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(                                        //(5)
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
             )
                }
            }
        }
    }
}

@Composable                                                          // (6)
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",                                       // (7)
        modifier = modifier
    )
}

@Preview(showBackground = true)                                      // (8)
@Composable
    fun GreetingPreview() {
        HelloMobileTheme {
            Greeting("Android")
        }
    }
```

| | |
|---|---|
| **(1) enableEdgeToEdge()** | Rozszerza aplikację na całą powierzchnię ekranu, włącznie z paskiem statusu i nawigacji. Standard od Android 15. |
| **(2) setContent { }** | Zastępuje stary setContentView(R.layout.xml). Przyjmuje funkcję Composable jako korzeń drzewa UI. |
| **(3) HelloMobileTheme { }** | Opakowuje UI w Material3 Design System, dostarcza kolory, typografię, kształty. Definicja w ui/theme/. |
| **(4) Scaffold { }** | Podstawowy layout Material3: topBar, bottomBar, floatingActionButton, content. innerPadding uwzględnia status bar i nav bar. |
| **(5) Greeting(...)** | Wywołanie własnej funkcji Composable. Compose buduje drzewo UI z takich wywołań. |
| **(6) @Composable** | Anotacja oznaczająca funkcję jako element UI Compose. Może być wywoływana tylko z innych @Composable lub setContent. |
| **(7) "Hello $name!"** | Kotlin string template - interpolacja zmiennej. Odpowiednik String.format() z Javy, ale czytelniejszy. |
| **(8) @Preview** | Generuje podgląd w Compose Preview (prawy panel Xcode-like). Nie wymaga emulatora. showBackground = true dodaje białe tło. |

## 4.2. Zadanie podstawowe - modyfikacja Greeting

**Zadanie 4.2 - Spersonalizuj ekran powitalny**

Zmodyfikuj funkcję Greeting tak, aby wyświetlała Twoje imię i nazwisko. Dodaj drugi Text z nazwą uczelni. Zmień kolor pierwszego tekstu na zielony: Color(0xFF00C47A). Użyj Compose Preview (zakładka Design w Android Studio) do podglądu bez uruchamiania emulatora. Oczekiwany efekt po modyfikacji - *patrz przykład kodu poniżej.*

**Greeting.kt** — **Kotlin**

```
@Composable fun Greeting(name: String, modifier: Modifier=Modifier) {

    // Zadanie: zmodyfikuj ten kod
    Column(modifier=modifier.padding(16.dp),
        verticalArrangement=Arrangement.spacedBy(8.dp)) {
        Text(text="Cześć, $name!",
            style=MaterialTheme.typography.headlineMedium,
            color=Color(0xFF00C47A) // ← zmień kolor         )
            Text(text="Wydział Elektrotechniki i Informatyki",
                style=MaterialTheme.typography.bodyMedium,
                color=MaterialTheme.colorScheme.onSurfaceVariant)
        }
    }

   // Pamiętaj o importach:
   // import androidx.compose.foundation.layout.Column
   // import androidx.compose.foundation.layout.Arrangement
   // import androidx.compose.foundation.layout.padding
   // import androidx.compose.ui.graphics.Color
    // import androidx.compose.ui.unit.dp
```

# 5. Rozbudowa aplikacji o interaktywny licznik

## 5.1. Niezbędne podstawy Kotlina

Przed pisaniem kodu Compose przypomnijmy niezbędne elementy języka Kotlin:

**Przypomnienie podstaw Kotlina**

```
// 1. VAL vs VAR
val imię = "Anna"       // niezmienne

var licznik = 0         // zmienne
licznik++               // OK: modyfikacja var

// 2. STRING TEMPLATES
val wiadomość = "Masz $licznik wiadomości"
// $zmienna

val info = "Wersja: ${BuildConfig.VERSION_NAME}"
// ${wyrażenie}

// 3. FUNKCJA z domyślnym parametrem
fun pozdrow(imię: String, prefiks: String = "Cześć") =
    "$prefiks, $imię!"

// 4. NULL SAFETY
var tekst: String? = null   // może być null
val dług = tekst?.length    // safe call: null jeśli tekst == null

val dług2 = tekst?.length ?: 0
// Elvis: 0 jeśli null

// 5. LAMBDA
val przycisk: () -> Unit = { println("Kliknięto!") }

// lub w skróconej formie jako ostatni parametr:
Button(
    onClick = { licznik++ }
) {
    Text("Kliknij")
}
```

## 5.2. Implementacja licznika krok po kroku

Utwórz nowy plik Kotlin: kliknij prawym na pakiet w **Project → New → Kotlin Class/File → File → nazwa CounterScreen**.

**CounterScreen.kt** — Kotlin

```
import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp

@Composable
fun CounterScreen() {
  var count by remember { mutableStateOf(0) }
  val message = when {
    count == 0 -> "Zacznij liczyć!"
```

```
    count < 10  -> "Dobry start"
    count < 50  -> "Nie zatrzymuj się!"
    else        -> "Wow, $count - nieźle!"
  }
  Column(
    modifier = Modifier
      .fillMaxSize()
      .padding(24.dp),
    horizontalAlignment = Alignment.CenterHorizontally,
    verticalArrangement = Arrangement.Center
  ) {
    Text(
      text = "$count",
      fontSize = 80.sp,
      fontWeight = FontWeight.Bold,
      color = MaterialTheme.colorScheme.primary
    )
    Spacer(modifier = Modifier.height(32.dp))
    Row(horizontalArrangement = Arrangement.spacedBy(16.dp)) {
      FilledTonalButton(onClick = { if (count > 0) count-- }) {
        Text("-", fontSize = 20.sp)
      }
      OutlinedButton(onClick = { count = 0 }) {
        Text("Reset")
      }
      Button(onClick = { count++ }) {
        Text("+", fontSize = 20.sp)
      }
    }
    Spacer(modifier = Modifier.height(24.dp))

    Text(
      text = message,
      style = MaterialTheme.typography.bodyLarge,
      color = MaterialTheme.colorScheme.onSurfaceVariant
    )
  }
}
@Preview(showBackground = true)
@Composable
fun CounterPreview() {
  HelloMobileTheme {
    CounterScreen()
  }
}
```

## 5.3. Podłączenie CounterScreen do MainActivity

```
setContent {
  HelloMobileTheme {
    Surface(
      modifier = Modifier.fillMaxSize(),
      color = MaterialTheme.colorScheme.background
    ) {
      CounterScreen() // ← zamieniono Greeting na CounterScreen
    }
  }
```

```
}
```

| # | Akcja | Uruchomienie i weryfikacja |
|---|---|---|
| 1 | **Buduj projekt** | Kliknij Build → Make Project (**Ctrl+F9**). Sprawdź panel Build w dolnej belce - brak błędów to wartość 0 errors. |
| 2 | **Uruchom na emulatorze** | Wybierz emulator Pixel 9 Pro w górnym pasku → kliknij Run (**Shift+F10** lub zielony trójkąt). Czas buildu: 30-90 sekund. |
| 3 | **Testuj interakcję** | Kliknij przyciski + i - na emulatorze. Obserwuj jak liczba zmienia się na ekranie w czasie rzeczywistym. Sprawdź komunikat przy 10, 50+. |
| 4 | **Live Edit** | Zmień tekst w jednym z przycisków (np. "Kliknij!" → "+1") i zapisz plik. Live Edit powinno automatycznie zaktualizować emulator **bez pełnego restartu**. |

**Jak działa State w Compose?**

remember { mutableStateOf(0) } tworzy obiekt stanu powiązany z danym miejscem w drzewie Composable. Gdy `count` się zmienia, Compose wie, które części UI zależą od tej wartości i przerysowuje *tylko je* - nie cały ekran. To nazywa się inteligentnym rekomponowaniem (smart recomposition).

Compose 'obserwuje' dostęp do zmiennych state podczas kompozycji i zapamiętuje zależności. Zmiana stanu → rekomponowanie → nowy UI tree.

# 6. Zadania do wykonania

Poniższe zadania są wymagane do zaliczenia ćwiczenia. Wykonaj je w podanej kolejności, gdyż każde kolejne bazuje na poprzednim.

## Zadanie 1 - Konfiguracja środowiska

**Wymagania**

1.1 Zainstaluj Android Studio i pokaż prowadzącemu ekran Welcome Screen z widoczną wersją (About → Android Studio Panda).
1.2 Utwórz i uruchom AVD - emulator Pixel 9 Pro lub Pixel 8 z API ≥ 34. Pokaż działający emulator z ekranem głównym Androida.
1.3 W terminalu Android Studio wykonaj: adb devices - pokaż prowadzącemu wynik z widocznym emulator-xxxx device.
1.4 W SDK Manager (File → Settings → Android SDK) zrób screenshot pokazujący zainstalowane komponenty: SDK Platform 34 lub 35 + Build-Tools.

## Zadanie 2 - Projekt i analiza kodu

**Wymagania**

2.1 Utwórz projekt HelloMobile zgodnie z sekcją 3.1. Package name np.: pl.prz.pam.<twoje_inicjały> (np. pl.put.pam.ajk).

2.2 Zmodyfikuj funkcję Greeting (sekcja 4.2). Wyświetl imię, nazwisko i wydział w trzech osobnych liniach tekstu z różnymi stylami.
2.3 Ustaw kolor tytułu na zgodny z paletą Material3. Użyj MaterialTheme.colorScheme.primary zamiast hardcoded Color().
2.4 Uruchom Compose Preview i zrób screenshot całego IDE z widocznym podglądem w panelu Design.
2.5 Opisz w komentarzu w kodzie co robi każda z anotacji: @Composable i @Preview.

## Zadanie 3 - Licznik interaktywny

### Wymagania

3.1 Zaimplementuj CounterScreen z sekcji 5.2, licznik z trzema przyciskami (+, Reset, -).
3.2 Dodaj walidację: przycisk - nie pozwala zejść poniżej 0 (zaimplementowane w przykładzie). Dodatkowo: przycisk + nie pozwala przekroczyć 99.
3.3 Dodaj wyświetlanie historii: pod licznikiem pokaż ostatnie 5 zmian w formie listy, np. '+1', '-1', 'Reset'. Użyj remember { mutableStateListOf() }.
3.4 Zmień wygląd przycisku + na wyróżniający się (Button z niestandardowymi kolorami; użyj ButtonDefaults.buttonColors()).
3.5 Uruchom na emulatorze i zademonstruj działanie prowadzącemu.

## Zadanie 4 - Rozszerzenie

### Wymagania

4.1 Dodaj TextField (pole tekstowe) nad licznikiem, w którym użytkownik wpisuje imię.
4.2 Powitanie nad licznikiem zmienia się dynamicznie wraz ze wpisywanym imieniem: "Cześć, [imię]! Twój wynik: [count]"
4.3 Obsłuż pustą wartość TextField - gdy pole jest puste, wyświetl "Cześć, nieznajomy!" zamiast "Cześć, !".
4.4 Użyj pamiętanego stanu: var name by remember { mutableStateOf("") }.

**Wskazówka do Zadania 4: TextField:** Poniżej przykład użycia OutlinedTextField w Compose.

**hint do Zadania 4** — Kotlin

```
setContent {
  HelloMobileTheme {
    Surface(
      modifier = Modifier.fillMaxSize(),
      color = MaterialTheme.colorScheme.background
    ) {
      CounterScreen() // ← zamieniono Greeting na CounterScreen
    }
  }
}var name by remember { mutableStateOf("") }

OutlinedTextField(
  value = name,
  onValueChange = { name = it }, // nowa wartość po każdym znaku
  label = { Text("Twoje imię") },
  singleLine = true,
  modifier = Modifier
    .fillMaxWidth()
    .padding(bottom = 16.dp)
)

// Dynamiczny tekst powitania
```

```
val greeting = if (name.isBlank()) {
    "Cześć, nieznajomy!"
} else {
    "Cześć, $name!"
}

Text(
    text = greeting,
    style = MaterialTheme.typography.titleMedium
)
```

# 7. Najczęstsze błędy i ich rozwiązania

| | |
|---|---|
| **Gradle sync failed** | Sprawdź połączenie z Internetem. Kliknij **File → Sync Project with Gradle Files**. Jeśli błąd nadal występuje: **File → Invalidate Caches / Restart → Invalidate and Restart**. |
| **Emulator nie startuje** | Sprawdź: **BIOS → Intel VT-x** lub **AMD-V** włączone. Windows: **Task Manager → Performance → Virtualization: Enabled**. Alternatywa: połącz fizyczny telefon kablem USB i włącz **USB Debugging** w opcjach dewelopera. |
| **Cannot resolve symbol 'Composable'** | Import nie został dodany automatycznie. Naciśnij Alt+Enter na podkreślonej anotacji → **Add import.** Lub dodaj ręcznie: **import androidx.compose.runtime.Composable** |
| **Build error: Unresolved reference** | Sprawdź, czy wszystkie importy są na górze pliku. Sprawdź, czy piszesz w zakresie @Composable (nie poza funkcją). Kliknij **Build → Clean Project**, potem **Rebuild Project**. |
| **Emulator bardzo wolny** | Zwiększ RAM w AVD Manager (edytuj AVD → Show Advanced Settings → RAM: 4096 MB). Windows: upewnij się, że HAXM jest zainstalowany (SDK Manager → SDK Tools). Alternatywa: użyj fizycznego urządzenia Android z USB Debugging. |
| **Live Edit nie działa** | Upewnij się, że aplikacja jest uruchomiona (nie zatrzymana).<br>Live Edit działa tylko dla @Composable - zmiany w logice wymagają pełnego buildu.<br>Sprawdź: **Settings → Editor → Live Edit → Apply Code Changes on the Fly**. |

# 9. Materiały dodatkowe

## Oficjalna dokumentacja

| | |
|---|---|
| **Jetpack Compose** | Oficjalny przewodnik: https://developer.android.com/compose |
| **Codelabs Android** | Interaktywne tutoriale Google: https://developer.android.com/codelabs |
| **Kotlin Koans** | Ćwiczenia z języka Kotlin online: https://kotlinlang.org/docs/koans.html |
| **Material Design 3** | Specyfikacja i komponenty: https://m3.material.io |
| **Android API Reference** | Pełna dokumentacja klas: https://developer.android.com/reference |
| **Compose Layout Basics** | Codelab: https://developer.android.com/codelabs/jetpack-compose-layouts |

## Polecane kursy wideo

Android Basics with Compose (Google) - *oficjalny kurs Google, darmowy, ~50 godzin*

youtube.com/@PhilippLackner - *bardzo aktualny, polecany przez społeczność*

youtube.com/@stevdza-san - *animacje, nawigacja, architecture*

## 10. Sprawozdanie

Sprawozdanie nie jest wymagane dla tego ćwiczenia. **Repozytorium GitHub:** *prześlij link do repozytorium na Moodle.*

| # | Akcja | Instrukcja wysłania repozytorium |
|---|---|---|
| 1 | Utwórz repo | Utwórz prywatne repozytorium na github.com o nazwie pam-lab1-<twoje_inicjały, albo index> (np. pam-lab1-ajk). |
| 2 | Inicjuj Git | W Android Studio: VCS → Enable Version Control Integration → Git.Lub w terminalu: cd /ścieżka/do/projektu && git init && git remote add origin <url> |
| 3 | .gitignore | Android Studio automatycznie generuje .gitignore dla projektu Gradle. Sprawdź czy zawiera: .gradle/, build/, *.apk, .idea/workspace.xml |
| 4 | Pierwszy commit | git add . && git commit -m 'feat: lab1 initial implementation' (Conventional Commits!) |
| 5 | Push i link | git push -u origin main. Dodaj prowadzącego jako collaboratora (Settings → Collaborators). Wklej URL repo na Moodle. |

**Konfiguracja środowiska deweloperskiego i pierwsza aplikacja Android**

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

**MobileHub**

*Następne ćwiczenie: Kotlin 2 - Nawigacja między ekranami, ViewModel i wzorzec MVVM*

KATEDRA INFORMATYKI I AUTOMATYKI

**Programowanie Aplikacji Mobilnych**

Informatyka

**ĆWICZENIE 2**

Nawigacja i ViewModel

# Instrukcja Laboratoryjna Nr 2

*Nawigacja między ekranami, ViewModel i wzorzec MVVM*

Przedmiot: Programowanie Aplikacji Mobilnych

Czas realizacji: 90 minut • Forma: indywidualna • Wymaganie wstępne: ukończone ćwiczenie 1

### Cel ćwiczenia

Po ukończeniu ćwiczenia student będzie umieć:

- ✓ zdefiniować grafy nawigacji w *Jetpack Compose*
- ✓ przekazywać dane między ekranami przez argumenty tras
- ✓ tworzyć ViewModel i zarządzać stanem UI
- ✓ korzystać ze StateFlow i collectAsStateWithLifecycle
- ✓ zbudować wieloekranową aplikację MVVM
- ✓ obsłużyć przycisk Wstecz i back stack
- ✓ dodać zależności przez Version Catalog (libs.toml)

### Wymagania wstępne

Student powinien znać z ćwiczenia 1:

- **@Composable** *i* budowa drzewa *UI Compose,* stan lokalny
- Struktura projektu Android (*Manifest, build.gradle*)
- Uruchamianie aplikacji na emulatorze

### Nowe pojęcia w tym ćwiczeniu

- NavController, NavHost, composable<Route>
- ViewModel, viewModels(); Jetpack Lifecycle
- StateFlow, collectAsStateWithLifecycle()

**MVVM** *(Model–View–ViewModel)*

## Spis treści

# 1. Wzorzec MVVM: teoria i praktyka

Wzorzec MVVM (Model–View–ViewModel) jest oficjalnie rekomendowaną architekturą aplikacji Android przez Google. Rozdziela odpowiedzialności na trzy warstwy, dzięki czemu kod jest testowalny i utrzymywalny.

## 1.1. Trzy warstwy MVVM

**WARSTWA VIEW (ekrany / @Composable)**

- MainActivity, CounterScreen, ListScreen, DetailScreen
- Obserwuje StateFlow z ViewModel przez collectAsStateWithLifecycle()
- NIE zawiera logiki biznesowej - tylko renderuje stan i deleguje zdarzenia
- Przykład: Button(onClick = { viewModel.addTask(text) })

**WARSTWA VIEWMODEL (logika prezentacji)**

- Klasa dziedzicząca po ViewModel() - przeżywa rotację ekranu!
- Zawiera StateFlow<UiState> jako źródło prawdy (Single Source of Truth)
- Wywołuje metody Repository, przetwarza wyniki, emituje nowy stan
- Przykład: fun addTask(text: String) { _tasks.update { it + Task(text) } }

**WARSTWA MODEL (dane i logika domenowa)**

- Repository - abstrahuje źródło danych (Room, API, SharedPreferences)
- Data classes - czyste obiekty danych bez logiki Androida
- Use Cases (opcjonalne) - enkapsulacja reguł biznesowych
- Przykład: data class Task(val id: UUID, val text: String, val done: Boolean)

**Dlaczego ViewModel przeżywa rotację ekranu?**

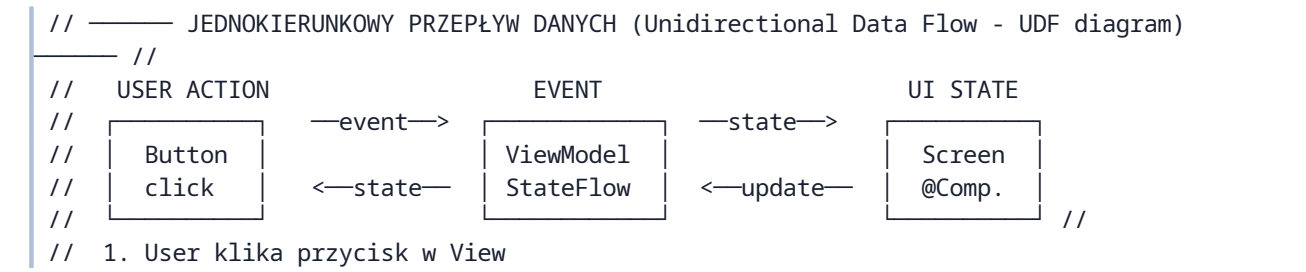
ViewModel jest przechowywany w ViewModelStore, który jest powiązany z Activity, nie z jej instancją. Gdy ekran się obraca, Android niszczy i tworzy na nowo Activity, ale ViewModelStore pozostaje.

ViewModel żyje dopóki Activity nie zostanie faktycznie zakończona (finish() lub Back).

Ważne: nie przechowuj w ViewModel referencji do Context, View ani Activity bo to powoduje memory leaks! Jeśli potrzebujesz Contextu, użyj AndroidViewModel(application).

## 1.2. Przepływ danych w MVVM

**UDF**



```
//  2. View wywołuje metodę ViewModel (event)
//  3. ViewModel aktualizuje StateFlow (state)
//  4. View obserwuje StateFlow i przerysowuje się (recomposition)
//  5. Użytkownik widzi nowy stan UI //
// Zasada: STATE idzie w DÓŁ (ViewModel → View)
//         EVENTY idą w GÓRĘ (View → ViewModel)

// Przykład - sealed class dla stanu UI (zalecany wzorzec)
sealed class TaskUiState {
    object Loading : TaskUiState()
    data class Success(val tasks: List<Task>, val filter: Filter = Filter.ALL) :
TaskUiState()
    data class Error(val message: String) : TaskUiState()
}
```

# 2. Konfiguracja projektu: zależności

Rozpocznij od projektu HelloMobile z Ćwiczenia 1 lub utwórz nowy projekt Empty Activity.

Dodaj wymagane biblioteki Jetpack.

## 2.1. libs.versions.toml: version catalog

**libs.versions.toml** TOML

```
# gradle/libs.versions.toml - dodaj/zweryfikuj te wpisy
[versions]
# Kotlin i Compose (sprawdź aktualne wersje na developer.android.com)
[versions]
kotlin = "2.0.21"
agp = "8.7.3"
compose-bom = "2024.11.00"
lifecycle = "2.8.7"
navigation-compose = "2.8.4"
activity-compose = "1.9.3"
kotlinx-serialization = "1.7.3"

# Compose BOM - zarządza wersjami całego ekosystemu
[libraries.compose-bom]
group = "androidx.compose"
name = "compose-bom"
[libraries.compose-bom.version]
ref = "compose-bom"

[libraries.compose-ui]
group = "androidx.compose.ui"
name = "ui"

[libraries.compose-ui-tooling]
group = "androidx.compose.ui"
name = "ui-tooling"

[libraries.compose-material3]
group = "androidx.compose.material3"
name = "material3"
```

```
[libraries.compose-foundation]
group = "androidx.compose.foundation"
name = "foundation"

[libraries.compose-runtime]
group = "androidx.compose.runtime"
name = "runtime"

# Lifecycle + ViewModel
[libraries.lifecycle-viewmodel-compose]
group = "androidx.lifecycle"
name = "lifecycle-viewmodel-compose"

[libraries.lifecycle-viewmodel-compose.version]
ref = "lifecycle"

[libraries.lifecycle-runtime-compose]
group = "androidx.lifecycle"
name = "lifecycle-runtime-compose"
[libraries.lifecycle-runtime-compose.version]
ref = "lifecycle"

# Navigation Compose
[libraries.navigation-compose]
group = "androidx.navigation"
name = "navigation-compose"
[libraries.navigation-compose.version]
ref = "navigation-compose"

# Activity Compose
[libraries.activity-compose]
group = "androidx.activity"
name = "activity-compose"
[libraries.activity-compose.version]
ref = "activity-compose"

# Kotlin Serialization (type-safe routes)
[libraries.kotlinx-serialization-json]
group = "org.jetbrains.kotlinx"
name = "kotlinx-serialization-json"
[libraries.kotlinx-serialization-json.version]
ref = "kotlinx-serialization"

[plugins.android-application]
id = "com.android.application"
[plugins.android-application.version]
ref = "agp"

[plugins.kotlin-android]
id = "org.jetbrains.kotlin.android"
[plugins.kotlin-android.version]
ref = "kotlin"

[plugins.compose-compiler]
id = "org.jetbrains.kotlin.plugin.compose"
[plugins.compose-compiler.version]
ref = "kotlin"

[plugins.kotlin-serialization]
```

```
id = "org.jetbrains.kotlin.plugin.serialization"
[plugins.kotlin-serialization.version]
ref = "kotlin"
```

## 2.2. build.gradle.kts (app): blok dependencies

**app/build.gradle.kts - sprawdź/dodaj poniższe** **Kotlin DSL**

```
 plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.compose.compiler)
    alias(libs.plugins.kotlin.serialization)
   type-safe routes // ← nowe
 }
 android {
    compileSdk = 35
    defaultConfig {
        minSdk = 26
        targetSdk = 35
        // ...
    }
}
 dependencies {
    // Compose BOM - importuj PRZED innymi artefaktami
Compose
    implementation(platform(libs.compose.bom))
    implementation(libs.compose.ui)
    implementation(libs.compose.material3)
    implementation(libs.compose.foundation)
    implementation(libs.compose.runtime)
    debugImplementation(libs.compose.ui.tooling)
     // Activity + Lifecycle
    implementation(libs.activity.compose)
    implementation(libs.lifecycle.viewmodel.compose)
    implementation(libs.lifecycle.runtime.compose)
  // collectAsStateWithLifecycle
     // Navigation Compose
    implementation(libs.navigation.compose)
     // Kotlin Serialization (type-safe routes)
    implementation(libs.kotlinx.serialization.json)
}
```

| # | Akcja | Synchronizacja Gradle po dodaniu zależności |
|---|---|---|
| 1 | **Sync Now** | Po edycji pliku gradle Android Studio pokazuje baner: "Gradle files have changed" → kliknij **Sync Now**. Czas: 30–120 sekund. |
| 2 | **Weryfikacja** | Otwórz Build → Build Output. Brak błędów BUILD SUCCESSFUL = sukces. Jeśli błąd: najczęstszy powód to literówka w nazwie biblioteki w .toml. |
| 3 | **Problemy** | Jeśli sync nie zakończy się powodzeniem: File → Invalidate Caches → Invalidate and Restart. Po restarcie: File → Sync Project with Gradle Files. |

# 3. Nawigacja Compose: NavController i NavHost

Navigation Compose to oficjalna biblioteka Google do zarządzania nawigacją w aplikacjach Jetpack Compose. Obsługuje back stack, deep links, animacje przejść i type-safe routes.

## 3.1. Kluczowe koncepty nawigacji

| | |
|---|---|
| | Mózg nawigacji. Zarządza back stackiem. Tworzysz przez<br><br>. Nigdy nie przekazuj do ViewModel - to widok! |
| | Composable 'kontener' gdzie wyświetlane są ekrany. Definiujesz w nim graf nawigacji - mapowanie trasa → ekran. |
| | Adnotacja Kotlinx Serialization na klasie trasy. Zastępuje stare stringowe ścieżki ("screen/{id}"). Type-safe: kompilator wykrywa błędy. |
| | Przejście do nowego ekranu. Opcje:<br>(wyczyszcz stos),<br>(nie duplikuj). Przekazujesz obiekt Route. |
| | Powrót do poprzedniego ekranu. Odpowiednik systemowego przycisku Wstecz. Bezpieczniejszy niż . |
| | Przechwytuje systemowy przycisk Wstecz w Compose. Używaj oszczędnie - tylko gdy potrzebujesz niestandardowego zachowania. |

## 3.2. Definiowanie tras: type-safe routes

Utwórz plik Routes.kt w pakiecie aplikacji

**Routes.kt** - definicje tras nawigacji **Kotlin**

```
package pl.edu.prz.taskapp
import kotlinx.serialization.Serializable
// ── TRASY BEZ PARAMETRÓW ──────────────────────────────────────────
@Serializable object HomeRoute              // ekran główny (lista zadań)
@Serializable object AddTaskRoute           // ekran dodawania nowego zadania
@Serializable object SettingsRoute          // ekran ustawień
// ── TRASY Z PARAMETRAMI ───────────────────────────────────────────
@Serializable data class TaskDetailRoute(
    val taskId: String                      // przekazujemy ID jako String (UUID.to-
String())
)
// ── SEALED CLASS dla grupowania tras (opcjonalnie) ────────────────
// Przydatna przy BottomNavigation - mapowanie ikona ↔ trasa
sealed class BottomNavRoute(val label: String) {
    object Home     : BottomNavRoute("Zadania")
    object Settings : BottomNavRoute("Ustawienia") }
// ── PRZYKŁAD użycia w NavHost
composable<HomeRoute>  { HomeScreen(navController) }
```

```
composable<TaskDetailRoute> { backStackEntry ->
    val route = backStackEntry.toRoute<TaskDetailRoute>()
    TaskDetailScreen(taskId = route.taskId)  }
```

## 3.3. NavHost: budowa grafu nawigacji

**AppNavigation.kt** - centralny graf nawigacji **Kotlin**

```
package pl.edu.prz.taskapp
import androidx.compose.runtime.Composable
import androidx.navigation.NavHostController
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
import androidx.navigation.toRoute

object Routes {
  const val HOME = "home"
  const val ADD_TASK = "add_task"
  const val TASK_DETAIL = "task_detail/{taskId}"
    fun taskDetail(taskId: String) = "task_detail/$taskId"
}
  @Composable
  fun AppNavigation(
 val navController = rememberNavController()
     NavHost(
       navController = navController,
       startDestination = Routes.HOME
     ) {
    // ── Ekran główny ──────────────────────────────────────────
  composable(Routes.HOME) {
  HomeScreen(
    onNavigateToAdd = {
      navController.navigate(Routes.ADD_TASK)
  },
    onNavigateToDetail = { id ->
      navController.navigate(
        Routes.taskDetail(id)
      )
    }
  )
  composable(Routes.ADD_TASK) {
    AddTaskScreen(
      onTaskAdded = {
        navController.popBackStack()
      }
    )
 }

composable<HomeRoute> {
             HomeScreen(
                 OnNavigateToAdd = { navController.navigate(AddTaskRoute) },
                 onNavigateToDetail = { taskId ->
                     navController.navigate(TaskDetailRoute(taskId))
                 }
             )
```

```
        }
        // ── Dodawanie zadania ──────────────────────────────────────
        composable<AddTaskRoute> {
            AddTaskScreen(
                onTaskAdded = {                    // Po dodaniu: wróć do Home i wyczyść
AddTask ze stosu
                    navController.navigate(HomeRoute) {
                        popUpTo(HomeRoute) { inclusive = false }
                    }
                },
                onNavigateBack = { navController.navigateUp() }
            )
        }
        // ── Szczegóły zadania - z parametrem ──────────────────────
        composable<TaskDetailRoute> { backStackEntry ->
            val route = backStackEntry.toRoute<TaskDetailRoute>()
            TaskDetailScreen(
                taskId = route.taskId,
                onNavigateBack = { navController.navigateUp() }
            )
        }
        // ── Ustawienia ─────────────────────────────────────────────
        composable<SettingsRoute> {
            SettingsScreen(onNavigateBack = { navController.navigateUp() })
        }
    }
}
```

**NavController tylko w Composable; nie w ViewModel!**

NavController jest elementem warstwy View i **nigdy** nie powinien być przekazywany do ViewModel. ViewModel nie wie nic o nawigacji.

Poprawny wzorzec: ViewModel emituje zdarzenie (np. NavigateToDetail(id)), ekran obserwuje ten event i sam wywołuje navController.navigate().

Możesz użyć SharedFlow<UiEffect> jako kanał zdarzeń jednorazowych (one-shot events), np. nawigacja, snackbar, dialog.

# 4. ViewModel i StateFlow: zarządzanie stanem

## 4.1. Tworzenie ViewModel

**TaskViewModel.kt** Kotlin

```
package pl.edu.prz.taskapp
import androidx.lifecycle.ViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.update
import java.util.UUID
```

```
// ── MODEL - data class ──────────────────────────────────────────────────
 data class Task(
     val id: String = UUID.randomUUID().toString(),
     val text: String,     val isDone: Boolean = false,
     val priority: Priority = Priority.NORMAL,
 )
  enum class Priority { LOW, NORMAL, HIGH }  enum class Filter { ALL, ACTIVE, DONE }

// ── UI STATE - sealed class ─────────────────────────────────────────────
 data class TaskListUiState(
     val tasks: List<Task> = emptyList(),
     val filter: Filter = Filter.ALL,
     val isLoading: Boolean = false,
     val errorMessage: String? = null,
 )

// Computed property - przefiltrowane zadania (nie przechowujemy osobno!)
 val TaskListUiState.filteredTasks: List<Task>
     get() = when (filter) {
         Filter.ALL    -> tasks
         Filter.ACTIVE -> tasks.filter { !it.isDone }
         Filter.DONE   -> tasks.filter { it.isDone }
     }

// ── VIEWMODEL ──────────────────────────────────────────────────────────
 class TaskViewModel : ViewModel() {
     // _uiState - prywatny, mutowalny (tylko ViewModel może modyfikować)
     private val _uiState = MutableStateFlow(TaskListUiState())
     // uiState - publiczny, read-only (View może tylko obserwować)
     val uiState: StateFlow<TaskListUiState> = _uiState.asStateFlow()

     // ── Akcje (eventy z View) ──────────────────────────────────────────
      fun addTask(text: String) {
         if (text.isBlank()) return
           // walidacja
         _uiState.update { current ->
             current.copy(tasks = current.tasks + Task(text = text.trim()))
         }
     }
      fun toggleTask(taskId: String) {
         _uiState.update { current ->
             current.copy(tasks = current.tasks.map { task ->
                 if (task.id == taskId) task.copy(isDone = !task.isDone) else task
             })
         }
     }
      fun deleteTask(taskId: String) {
         _uiState.update { current ->
             current.copy(tasks = current.tasks.filter { it.id != taskId })
         }
     }
      fun setFilter(filter: Filter) {
         _uiState.update { it.copy(filter = filter) }
     }
      fun setPriority(taskId: String, priority: Priority) {
         _uiState.update { current ->
             current.copy(tasks = current.tasks.map { task ->
                 if (task.id == taskId) task.copy(priority = priority) else task
```

```
            })
        }
    }

// Dane przykładowe (do testów - usuń w produkcji)
    init {
        _uiState.update { it.copy(tasks = listOf(
            Task(text = "Skonfigurować Android Studio", isDone = true),
            Task(text = "Napisać pierwszą aplikację Compose"),
            Task(text = "Nauczyć się ViewModel i StateFlow"),
            Task(text = "Zbudować aplikację z nawigacją", priority = Priority.HIGH),
        ))}
    }
}
```

## 4.2. Podłączenie *ViewModel* do *Composable*

**HomeScreen.kt** (fragment) - obserwacja StateFlow **Kotlin**

```
package pl.edu.prz.taskapp
import androidx.compose.runtime.getValue
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.lifecycle.viewmodel.compose.viewModel
 @Composable fun HomeScreen(
    onNavigateToAdd: () -> Unit,
    onNavigateToDetail: (taskId: String) -> Unit,
    // Domyślny ViewModel - tworzony raz na czas życia Activity:
    viewModel: TaskViewModel = viewModel() ) {
    // collectAsStateWithLifecycle - zatrzymuje kolekcję gdy app idzie w tło!
    // Bezpieczniejsze niż collectAsState() - oszczędza baterię.
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
     // Używamy przefiltrowanych zadań (computed property)
    val tasks = uiState.filteredTasks
     Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("Moje Zadania (${tasks.size})") },
                actions = {
                    IconButton(onClick = onNavigateToAdd) {
                        Icon(Icons.Default.Add, contentDescription = "Dodaj")
                    }
                }
            )
        }
    ) { paddingValues ->
        Column(modifier = Modifier.padding(paddingValues)) {
             // FilterChips - zmiana filtra przez ViewModel
            FilterRow(
                currentFilter = uiState.filter,
                onFilterChange = viewModel::setFilter    // method reference
            )
            // Lista zadań
            LazyColumn(modifier = Modifier.fillMaxSize()) {
                items(
                    items = tasks,
                    key = { it.id }    // key = stabilna identyfikacja dla animacji
```

```
                ) { task ->
                    TaskItem(
                        task = task,
                        onToggle = { viewModel.toggleTask(task.id) },
                        onDelete = { viewModel.deleteTask(task.id) },
                        onClick  = { onNavigateToDetail(task.id) }
                    )
                }
            }
        }
    }
}
```

**collectAsStateWithLifecycle vs collectAsState**

**collectAsStateWithLifecycle()** (zalecane) - zatrzymuje kolekcję gdy app przechodzi do stanu STOPPED (np. idzie w tło, ekran się wyłącza). Oszczędza baterię i CPU. Wymaga: lifecycle-runtime-compose.

**collectAsState()** - kolekcja jest aktywna nawet gdy app jest w tle. Prostsze w użyciu, ale nieoptymalne. Używaj tylko w preview lub testach.

Reguła: zawsze używaj collectAsStateWithLifecycle() w produkcyjnym kodzie.

# 5. Budowanie aplikacji Task Manager

Zbudujemy kompletną aplikację zarządzania zadaniami. Nazwa projektu: TaskApp. Składać się będzie ona z 4 ekranów połączonych nawigacją i współdzielącego ViewModel.

## 5.1. Struktura projektu

**Struktura projektu TaskApp**

```
TaskApp/
└── app/src/main/java/pl/edu/prz/taskapp/
    ├── MainActivity.kt          ← setContent { AppNavigation() }
    ├── Routes.kt                ← definicje tras (@Serializable)
    ├── AppNavigation.kt         ← NavHost z grafem nawigacji
    ├── TaskViewModel.kt         ← ViewModel + Task model + UiState
    └── ui/
        ├── HomeScreen.kt        ← lista zadań (LazyColumn)
        ├── AddTaskScreen.kt     ← formularz dodawania
        ├── TaskDetailScreen.kt  ← szczegóły zadania
        └── components/
            ├── TaskItem.kt      ← pojedyncze zadanie (reużywalne)
            └── FilterRow.kt     ← FilterChips do filtrowania
```

## 5.2. MainActivity.kt

**MainActivity.kt** - punkt wejścia, startuje nawigację **Kotlin**

```
package pl.edu.prz.taskapp
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import pl.edu.prz.taskapp.ui.theme.TaskAppTheme
  class MainActivity : ComponentActivity() {
     override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
         enableEdgeToEdge()
         setContent {
             TaskAppTheme {
                 // Cały graf nawigacji - AppNavigation zarządza NavController
                 AppNavigation()
             }
         }
     }
 }
 // WAŻNE: ViewModel jest tworzony przez viewModel() w HomeScreen/AddTaskScreen
 // WAŻNE: Ten sam ViewModel jest współdzielony gdy przekazujesz go w dół
 // Alternatywa dla współdzielenia: hiltViewModel() lub ViewModelProvider z Activity
scope
```

## 5.3. Ekran główny: HomeScreen z LazyColumn

**ui/HomeScreen.kt** - kompletna implementacja **Kotlin**

```
package pl.edu.prz.taskapp.ui
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Add
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.lifecycle.viewmodel.compose.viewModel
import pl.edu.prz.taskapp.*
@OptIn(ExperimentalMaterial3Api::class)
 @Composable fun HomeScreen(
     onNavigateToAdd: () -> Unit,
     onNavigateToDetail: (String) -> Unit,
     viewModel: TaskViewModel = viewModel() ) {
     val uiState by viewModel.uiState.collectAsStateWithLifecycle()
     val tasks = uiState.filteredTasks
      Scaffold(
         topBar = {
             TopAppBar(
                 title = { Text("Zadania") },
                 colors = TopAppBarDefaults.topAppBarColors(
```

```
                    containerColor = MaterialTheme.colorScheme.primaryContainer
                ),
                actions = {
                    Text("${tasks.count { it.isDone }}/${uiState.tasks.size}",
                         style = MaterialTheme.typography.bodyMedium,
                         modifier = Modifier.padding(end = 8.dp))
                    IconButton(onClick = onNavigateToAdd) {
                        Icon(Icons.Default.Add, "Dodaj zadanie")
                    }
                }
            )
        }
    ) { innerPadding ->
        Column(modifier = Modifier.fillMaxSize().padding(innerPadding)) {
            // FilterRow - selektor filtra
            FilterRow(
                currentFilter = uiState.filter,
                onFilterChange = viewModel::setFilter
            )
            // Pusta lista
            if (tasks.isEmpty()) {
                Box(modifier = Modifier.fillMaxSize(), contentAlignment = Align-
ment.Center) {
                    Column(horizontalAlignment = Alignment.CenterHorizontally) {
                        Text("Brak zadań", style = MaterialTheme.typography.titleMe-
dium,
                             color = MaterialTheme.colorScheme.onSurfaceVariant)
                        Spacer(Modifier.height(8.dp))
                        Text(if (uiState.filter == Filter.ALL) "Dotknij + aby dodać
pierwsze zadanie"
                             else "Brak zadań w tej kategorii",
                             style = MaterialTheme.typography.bodySmall,
                             color = MaterialTheme.colorScheme.onSurfaceVariant)
                    }
                }
            } else {
                LazyColumn(
                    modifier = Modifier.fillMaxSize(),
                    contentPadding = PaddingValues(vertical = 8.dp)
                ) {
                    items(items = tasks, key = { it.id }) { task ->
                        TaskItem(
                            task     = task,
                            onToggle = { viewModel.toggleTask(task.id) },
                            onDelete = { viewModel.deleteTask(task.id) },
                            onClick  = { onNavigateToDetail(task.id) }
                        )
                    }
                }
            }
        }
    }
}
```

## 5.4. Komponent TaskItem.kt

**ui/components/TaskItem.kt** - re-używalny komponent zadania **Kotlin**

```
package pl.edu.prz.taskapp.ui.components
import androidx.compose.animation.animateContentSize
import androidx.compose.foundation.clickable
import androidx.compose.foundation.layout.*
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.filled.Delete
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.style.TextDecoration
import androidx.compose.ui.unit.dp
import pl.edu.prz.taskapp.Priority
import pl.edu.prz.taskapp.Task
  @Composable fun TaskItem(
     task: Task,
     onToggle: () -> Unit,
     onDelete: () -> Unit,
     onClick: () -> Unit,
     modifier: Modifier = Modifier ) {
     val priorityColor = when (task.priority) {
         Priority.HIGH   -> MaterialTheme.colorScheme.error
         Priority.NORMAL -> MaterialTheme.colorScheme.primary
         Priority.LOW    -> MaterialTheme.colorScheme.onSurfaceVariant
     }
      Card(
         modifier = modifier
             .fillMaxWidth()
             .padding(horizontal = 16.dp, vertical = 4.dp)
             .animateContentSize()
       // animuje zmiany rozmiaru karty
             .clickable(onClick = onClick),
colors = CardDefaults.cardColors(
containerColor = if (task.isDone)
                 MaterialTheme.colorScheme.surfaceVariant
            else MaterialTheme.colorScheme.surface
         ),
         elevation = CardDefaults.cardElevation(defaultElevation = if (task.isDone) 0.dp
else 2.dp)
     ) {
         Row(
             modifier = Modifier.padding(horizontal = 8.dp, vertical = 12.dp),
             verticalAlignment = Alignment.CenterVertically
         ) {
             // Priorytet - kolorowy pasek
             Box(modifier = Modifier.width(4.dp).height(40.dp)
                 .padding(end = 0.dp), // handled by Spacer
             )
             Checkbox(
                 checked = task.isDone,
                 onCheckedChange = { onToggle() },
                 colors = CheckboxDefaults.colors(
                     checkedColor = MaterialTheme.colorScheme.primary
                )
             )
             Spacer(Modifier.width(8.dp))
             Text(
```

```
                text = task.text,
                modifier = Modifier.weight(1f),
                style = MaterialTheme.typography.bodyLarge.copy(
                    textDecoration = if (task.isDone) TextDecoration.LineThrough else
null,
                    color = if (task.isDone) MaterialTheme.colorScheme.onSurfaceVariant
                            else MaterialTheme.colorScheme.onSurface
                )
            )
            // Ikona priorytetu
            if (task.priority == Priority.HIGH) {
                Text("!", color = priorityColor,
                     style = MaterialTheme.typography.titleMedium,
                     modifier = Modifier.padding(horizontal = 4.dp))
            }
            IconButton(onClick = onDelete) {
                Icon(Icons.Default.Delete, "Usuń",
                     tint = MaterialTheme.colorScheme.onSurfaceVariant)
            }
        }
    }
}
```

## 5.5. Ekran AddTask: formularz z walidacją

**ui/AddTaskScreen.kt**

```
package pl.edu.prz.taskapp.ui
import androidx.compose.foundation.layout.*
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material3.*
import androidx.compose.runtime.* import androidx.compose.ui.Modifier
import androidx.compose.ui.focus.FocusRequester
import androidx.compose.ui.focus.focusRequester
import androidx.compose.ui.unit.dp
import androidx.lifecycle.viewmodel.compose.viewModel
import pl.edu.prz.taskapp.Priority
import pl.edu.prz.taskapp.TaskViewModel
 @OptIn(ExperimentalMaterial3Api::class)
 @Composable fun AddTaskScreen(
    onTaskAdded: () -> Unit,
    onNavigateBack: () -> Unit,
    viewModel: TaskViewModel = viewModel() ) {
    // Stan lokalny - formularz to stan View, nie ViewModel
    var taskText by remember { mutableStateOf("") }
    var selectedPriority by remember { mutableStateOf(Priority.NORMAL) }
    val isValid = taskText.isNotBlank() && taskText.length <= 200
     // Auto-focus na TextField po otwarciu ekranu
    val focusRequester = remember { FocusRequester() }
    LaunchedEffect(Unit) { focusRequester.requestFocus() }
     Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("Nowe zadanie") },
```

**ui/AddTaskScreen.kt**

```
                navigationIcon = {
                    IconButton(onClick = onNavigateBack) {
                       Icon(Icons.AutoMirrored.Filled.ArrowBack, "Wstecz")
                    }
                }
            )
        }
    ) { innerPadding ->
        Column(
            modifier = Modifier.fillMaxSize()
              .padding(innerPadding)
                .padding(horizontal = 24.dp, vertical = 16.dp),
            verticalArrangement = Arrangement.spacedBy(16.dp)
        ) {
            OutlinedTextField(
                value = taskText,
                onValueChange = { if (it.length <= 200) taskText = it },
                label = { Text("Treść zadania") },
                placeholder = { Text("Np. Ukończyć Ćwiczenie 2...") },
                supportingText = {
                    Text("${taskText.length}/200",
                         color = if (taskText.length > 180)
                             MaterialTheme.colorScheme.error
                         else MaterialTheme.colorScheme.onSurfaceVariant)
                },
                isError = taskText.length > 200,
                singleLine = false,
                maxLines = 4,
                modifier = Modifier.fillMaxWidth().focusRequester(focusRequester)
            )
             // Wybór priorytetu - SegmentedButton (Material3)
            Text("Priorytet:", style = MaterialTheme.typography.labelMedium)
            SingleChoiceSegmentedButtonRow(modifier = Modifier.fillMaxWidth()) {
                Priority.entries.forEachIndexed { index, priority ->
                    SegmentedButton(
                        shape = SegmentedButtonDefaults.itemShape(index, Priority.en-
tries.size),
                        onClick = { selectedPriority = priority },
                        selected = selectedPriority == priority,
                        label = { Text(priority.name) }
                    )
                }
            }
             Spacer(Modifier.weight(1f))
             // Przyciski
            Row(modifier = Modifier.fillMaxWidth(),
            horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                OutlinedButton(onClick = onNavigateBack, modifier = Modifier.weight(1f)) {
                    Text("Anuluj")
                }
                Button(
                    onClick = {
                        viewModel.addTask(taskText)
                        onTaskAdded()                    // nawigacja przez callback
                    },
                    enabled = isValid,
                    modifier = Modifier.weight(1f)
                ) {
```

**ui/AddTaskScreen.kt**

```
                    Text("Dodaj zadanie")
                }
            }
        }
    }
}
```

## 5.6. Ekran szczegółów: TaskDetailScreen

**ui/TaskDetailScreen.kt** - szczegóły zadania z parametrem nawigacji **Kotlin**

```
package pl.edu.prz.taskapp.ui
import androidx.compose.foundation.layout.*
import androidx.compose.material.icons.Icons
import androidx.compose.material.icons.automirrored.filled.ArrowBack
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.lifecycle.compose.collectAsStateWithLifecycle
import androidx.lifecycle.viewmodel.compose.viewModel
import pl.edu.prz.taskapp.Priority
import pl.edu.prz.taskapp.TaskViewModel
  @OptIn(ExperimentalMaterial3Api::class) @Composable fun TaskDetailScreen(
    taskId: String,
           // otrzymany z nawigacji
    onNavigateBack: () -> Unit,
    viewModel: TaskViewModel = viewModel() ) {
    val uiState by viewModel.uiState.collectAsStateWithLifecycle()
     // Znajdź zadanie po ID (może być null jeśli usunięto)
    val task = uiState.tasks.find { it.id == taskId }
     // Jeśli zadanie nie istnieje - wróć automatycznie
    LaunchedEffect(task) {
        if (task == null) onNavigateBack()
    }
     Scaffold(
        topBar = {
            TopAppBar(
                title = { Text("Szczegóły zadania") },
                navigationIcon = {
                    IconButton(onClick = onNavigateBack) {
                        Icon(Icons.AutoMirrored.Filled.ArrowBack, "Wstecz")
                    }
                }
            )
        }
    ) { innerPadding ->
        task?.let { t ->
            Column(
                modifier = Modifier.fillMaxSize()
                    .padding(innerPadding)
                    .padding(24.dp),
                verticalArrangement = Arrangement.spacedBy(16.dp)
            ) {
                // Treść zadania
                Card(modifier = Modifier.fillMaxWidth()) {
```

```
                    Column(modifier = Modifier.padding(16.dp)) {
                        Text("Treść", style = MaterialTheme.typography.labelSmall,
                             color = MaterialTheme.colorScheme.onSurfaceVariant)
                        Spacer(Modifier.height(8.dp))
                        Text(t.text, style = MaterialTheme.typography.bodyLarge)
                    }
                }
                 // Status + Priorytet
                Row(horizontalArrangement = Arrangement.spacedBy(8.dp)) {
                    AssistChip(
                        onClick = { viewModel.toggleTask(t.id) },
                        label = { Text(if (t.isDone) "Ukończone ✓" else "W trakcie") },
                        colors = AssistChipDefaults.assistChipColors(
                            containerColor = if (t.isDone)
                                MaterialTheme.colorScheme.primaryContainer
                            else MaterialTheme.colorScheme.surfaceVariant
                        )
                    )
                    AssistChip(
                        onClick = { },
                        label = { Text("Priorytet: ${t.priority.name}") }
                    )
                }
                 // Zmiana priorytetu
                Text("Zmień priorytet:", style = MaterialTheme.typography.labelMedium)
                Priority.entries.forEach { priority ->
                    OutlinedButton(
                        onClick = { viewModel.setPriority(t.id, priority) },
                        modifier = Modifier.fillMaxWidth()
                    ) {
                        Text(priority.name)
                    }
                }
                 Spacer(Modifier.weight(1f))
                 // Usuń zadanie
                Button(
                    onClick = { viewModel.deleteTask(t.id); onNavigateBack() },
                    modifier = Modifier.fillMaxWidth(),
                    colors = ButtonDefaults.buttonColors(
                        containerColor = MaterialTheme.colorScheme.errorContainer,
                        contentColor = MaterialTheme.colorScheme.onErrorContainer
                    )
                ) { Text("Usuń zadanie") }
            }
        }
    }
}
```

## 5.7. FilterRow: FilterChips

**ui/components/FilterRow.kt** Kotlin

```
package pl.edu.prz.taskapp.ui.components
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyRow
import androidx.compose.material3.*
import androidx.compose.runtime.Composable
```

```
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import pl.edu.prz.taskapp.Filter
 @Composable fun FilterRow(
    currentFilter: Filter,
    onFilterChange: (Filter) -> Unit,
    modifier: Modifier = Modifier ) {
    LazyRow(
        modifier = modifier.fillMaxWidth(),
        contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp),
        horizontalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(Filter.entries.size) { index ->
            val filter = Filter.entries[index]
            val label = when (filter) {
                Filter.ALL    -> "Wszystkie"
                Filter.ACTIVE -> "W trakcie"
                Filter.DONE   -> "Ukończone"
            }
            FilterChip(
                selected = currentFilter == filter,
                onClick = { onFilterChange(filter) },
                label = { Text(label) }
            )
        }
    }
}
```

# 6. Zadania do wykonania

Poniższe zadania realizujesz w nowym projekcie TaskApp lub w projekcie HelloMobile z Ćwiczenia 1. Każde zadanie bazuje na poprzednim.

## Zadanie 1. Konfiguracja projektu i nawigacja bazowa (20 pkt)

**Wymagania**

1.1. Utwórz projekt TaskApp lub dodaj zależności do HelloMobile zgodnie z sekcją 2 (libs.toml + build.gradle.kts). Pokaż prowadzącemu Build Output: BUILD SUCCESSFUL.

1.2. Zdefiniuj trasy nawigacji w Routes.kt: HomeRoute, AddTaskRoute, TaskDetailRoute(taskId). Użyj @Serializable.

1.3. Zbuduj AppNavigation.kt z NavHost zawierającym 3 destinations. Na razie ekrany mogą być tymczasowe: Text("Home"), Text("AddTask"), Text("Detail").

1.4. W MainActivity podłącz AppNavigation(). Uruchom na emulatorze i sprawdź czy aplikacja startuje bez crashy.

1.5. Ręcznie wywołaj nawigację: dodaj Button na "Home" który przechodzi do "AddTask". Sprawdź przycisk Wstecz.

## Zadanie 2. ViewModel i lista zadań (30 pkt)

**Wymagania**

2.1. Zaimplementuj TaskViewModel.kt zgodnie z sekcją 4.1: Task, Filter, TaskListUiState, metody: addTask, toggleTask, deleteTask, setFilter.
2.2. Zaimplementuj HomeScreen.kt z LazyColumn wyświetlającym zadania. Każde zadanie jako TaskItem: tekst + Checkbox + przycisk Usuń.
2.3. Podłącz ViewModel do HomeScreen przez viewModel() i obserwuj uiState przez collectAsStateWithLifecycle(). Zmiany przez ViewModel muszą automatycznie aktualizować UI.
2.4. Zaimplementuj FilterRow z FilterChip dla: Wszystkie / W trakcie / Ukończone. Zmiana filtra musi natychmiast filtrować listę.
2.5. Pokaż prowadzącemu: dodaj kilka zadań przez init {} w ViewModel, zaznacz kilka jako done, przełącz filtry.

## Zadanie 3. Ekran dodawania i szczegóły (30 pkt)

**Wymagania**

3.1 Zaimplementuj AddTaskScreen.kt z OutlinedTextField i przyciskiem Dodaj. Walidacja: pole nie może być puste, max 200 znaków. Przycisk Dodaj jest disabled gdy walidacja nie przechodzi.
3.2 Po kliknięciu Dodaj: wywołaj viewModel.addTask(text), wywołaj callback onTaskAdded() (nawigacja wstecz). Nowe zadanie musi pojawić się na liście.
3.3 Zaimplementuj TaskDetailScreen.kt - pokaż szczegóły zadania znalezionego po taskId. Dodaj przycisk toggle (ukończ/cofnij) i usuń.
3.4 Nawigacja: kliknięcie zadania na liście otwiera TaskDetailScreen z właściwym taskId. Wstecz wraca do listy.
3.5 Usuń dane przykładowe z init {} w ViewModel - lista powinna startować pusta.

## Zadanie 4. Rozszerzenia (20 pkt)

**Wymagania**

4.1 Dodaj w AddTaskScreen wybór priorytetu (SegmentedButton lub RadioButton): LOW / NORMAL / HIGH. Priorytet jest przekazywany do viewModel.addTask().
4.2 Na liście zadań HIGH-priority oznacz wyróżnikiem: np. kolorowy pasek z lewej strony karty lub ikona '!' przy tekście.
4.3 Dodaj obsługę pustego stanu (empty state): gdy lista jest pusta, pokaż centralnie tekst z instrukcją i ikoną (ContentUnavailableView lub własny Composable).
4.4 Zaimplementuj Snackbar przy usuwaniu zadania z możliwością cofnięcia (Undo). Wskazówka: użyj SnackbarHostState w Scaffold i zapamiętaj ostatnio usunięte zadanie.

**Wskazówka do Zadania 4.4: Snackbar z Undo**

**Snackbar z Undo** wzorzec

```
// W ViewModel: przechowaj ostatnio usunięte zadanie
 private var lastDeletedTask: Task? = null  fun deleteTask(taskId: String) {
     val task = _uiState.value.tasks.find { it.id == taskId } ?: return
     lastDeletedTask = task                    // zapamiętaj
     _uiState.update { it.copy(tasks = it.tasks.filter { t -> t.id != taskId }) } }
  fun undoDelete() {
     val task = lastDeletedTask ?: return
     _uiState.update { it.copy(tasks = it.tasks + task) }
     lastDeletedTask = null }  // W HomeScreen - SnackbarHostState
 val snackbarHostState = remember { SnackbarHostState() }
 val scope = rememberCoroutineScope()
```

```
  Scaffold(snackbarHost = { SnackbarHost(snackbarHostState) }) { ... }
  // W TaskItem onDelete:
onDelete = {
    viewModel.deleteTask(task.id)
    scope.launch {
        val result = snackbarHostState.showSnackbar(
            message = "Zadanie usunięte",
            actionLabel = "Cofnij",
            duration = SnackbarDuration.Short
        )
        if (result == SnackbarResult.ActionPerformed) {
            viewModel.undoDelete()
        }
    }
}
```

# 7. Kryteria oceniania

Ćwiczenie oceniane jest w skali 0–100 punktów. Maksymalny czas: 90 minut.

| Zadanie | Punkty | Uwagi |
|---|---|---|
| Zad. 1. Konfiguracja i nawigacja bazowa | **20** | Sync Gradle + działające przejścia między ekranami |
| Zad. 2. ViewModel i lista zadań | **30** | LazyColumn + filtry + toggle + delete |
| Zad. 3. AddTask i TaskDetail | **30** | Pełna nawigacja z parametrem taskId |
| Zad. 4. Priorytet + empty state + Undo | **20** | Kompletna implementacja wszystkich 4 podpunktów |
| SUMA | **100** | Minimum do zaliczenia: 50 punktów |

| Punkty | Ocena | Opis |
|---|---|---|
| **90–100** | **5.0** | Wszystkie zadania ukończone, kod czysty, MVVM poprawnie zaimplementowany |
| **80–89** | **4.5** | Zadania 1–4 ukończone z drobnymi brakami lub niepełnym Zadaniem 4 |
| **70–79** | **4.0** | Zadania 1–3 w pełni, Zadanie 4 częściowo (min. 2 z 4 podpunktów) |
| **60–69** | **3.5** | Zadania 1–3 ukończone, Zadanie 4 pominięte |
| **50–59** | **3.0** | Zadania 1–2 + podstawowa nawigacja do AddTask (bez TaskDetail) |
| **0–49** | **2.0** | Niezaliczone. Wymagany termin poprawkowy. |

# 8. Najczęstsze błędy i rozwiązania

| | |
|---|---|
| | Brakuje pluginu w build.gradle.kts. Sprawdź też wpis w libs.versions.toml: plugins → kotlin-serialization. Sync Gradle po każdej zmianie. |
| | Sprawdź, czy @Serializable jest zaimportowane. Upewnij się, że Route klasa/object jest w tym samym pakiecie lub poprawnie importowana w NavHost. |
| | viewModel() tworzy ViewModel skojarzony z najbliższym ViewModelStoreOwner (zazwyczaj Activity). Jeśli ViewModel jest tworzony osobno w kilku ekranach, to różne instancje! Rozwiązanie: przekaż ViewModel jako parametr lub użyj activityViewModel(). |
| | Sprawdź, czy nie używasz mutableListOf() zamiast MutableStateFlow. Mutable collections nie powiadamiają o zmianach! Zawsze używaj .update { } lub copy() z niezmienną listą. |
| | Sprawdź, czy navController.navigateUp() jest poprawnie podłączony. Jeśli używasz BackHandler sprawdź czy enabled=true. Domyślne zachowanie Wstecz jest zarządzane przez NavController automatycznie. |
| | wymaga żeby ID było unikalne i stabilne. Jeśli używasz indeksu jako key, możesz mieć crashe przy usuwaniu elementów. Używaj UUID jako klucz zadania. |

# 9. Porównanie: stan lokalny vs ViewModel

Ważne pytanie: kiedy używać remember { mutableStateOf() } (stan lokalny w Composable), a kiedy ViewModel? Odpowiedź zależy od zasięgu stanu.

| Kryterium | remember { mutableStateOf() } | ViewModel + StateFlow |
|---|---|---|
| Przeżywa rotację | NIE - traci stan przy obróceniu | TAK - przeżywa rotację ekranu |
| Zasięg | Tylko w danym Composable + dzieci | Cały ekran (Activity / Fragment) |
| Kiedy używać | UI state: pole tekstowe, rozwinięty panel | Business state: lista danych, błędy, ładowanie |
| Przykłady | var expanded, var text, var isVisible | lista tasków, user, zalogowany, error message |
| Testowalność | Trudna - zintegrowana z UI | Łatwa - ViewModel to czysta klasa Kotlin |
| Lifecycle | Żyje z Composable | Żyje z ViewModelStoreOwner (Activity) |

# 10. Sprawozdanie i repozytorium

Sprawozdanie nie jest wymagane. Demonstracja kodu prowadzącemu zastępuje pisemne sprawozdanie. **Repozytorium GitHub (wymagane):** ~~*Link do repo na Moodle do końca tygodnia w którym odbywa się ćwiczenie.*~~

| # | Akcja | Checklist przed wysłaniem repozytorium |
|---|---|---|
| 1 | Kod kompiluje się | Build → Make Project → 0 errors. Aplikacja uruchamia się na emulatorze bez crash-a. |
| 2 | Struktura plików | Sprawdź czy wszystkie pliki są w repozytorium: Routes.kt, AppNavigation.kt, TaskViewModel.kt, HomeScreen.kt, AddTaskScreen.kt, TaskDetailScreen.kt |
| 3 | Commit message | Conventional Commits: feat: add navigation and ViewModel; opisz co zrobiłeś. Unikaj: 'fix', 'update', 'changes' bez opisu. |
| 4 | README.md | Dodaj krótki README.md: imię/nazwisko, opis projektu, zrzut ekranu (możesz dodać .png z emulatora). |
| 5 | Brak .apk / build/ | .gitignore powinien wykluczać: build/, .gradle/, *.apk, .idea/workspace.xml. Sprawdź: git status nie pokazuje tych plików. |
| 6 | Link na Moodle | Wklej URL repozytorium (np. https://github.com/twoje-konto/taskapp-lab2) w odpowiednie pole na Moodle. |

*Instrukcja przygotowana dla przedmiotu Programowanie Aplikacji Mobilnych.*
*Wersja 1.5 • Rok akademicki 2025/2026 • Kontynuacja Ćwiczenia 1 (Android Studio + Compose)*
*Następne ćwiczenie: Ćwiczenie 3 - Room Database, persystencja danych i Coroutines IO.*

POLITECHNIKA RZESZOWSKA

KATEDRA INFORMATYKI I AUTOMATYKI

DR INŻ. MATEUSZ POMIANEK

# ĆWICZENIE LABORATORYJNE - KOTLIN 3

## Room Database i Kotlin Coroutines

*Programowanie Aplikacji Mobilnych (Android / Kotlin / Jetpack Compose)*

## Spis treści



## 1. Kotlin Coroutines

W aplikacjach mobilnych konieczne jest przechowywanie danych lokalnie. System Android wykorzystuje bazę SQLite jako podstawowy mechanizm trwałego zapisu danych. Praca bezpośrednio z SQLite wymaga jednak pisania dużej ilości kodu pomocniczego oraz ręcznego mapowania obiektów na rekordy bazy danych. Biblioteka Jetpack Room upraszcza ten proces poprzez wprowadzenie warstwy abstrakcji nad SQLite. Room integruje się z Kotlin Coroutines oraz Flow, co pozwala budować reaktywne aplikacje mobilne zgodne z architekturą MVVM. Kotlin Coroutines to natywny mechanizm asynchroniczności w Kotlinie. Zastępuje Callbacki, RxJava i AsyncTask. Pozwalają pisać asynchroniczny kod sekwencyjnie, bez zagnieżdżonych callbacków.

Asynchroniczne przetwarzanie zadań w środowisku Kotlin opiera się na mechanizmie Coroutines, w języku polskim możemy przyjąć nazwę korutyn () (tak, jest takie słowo!). Umożliwiają one wykonywanie operacji nieblokujących wątek główny interfejsu użytkownika. Poprzez wykorzystanie funkcji zawieszalnych (suspend) oraz maszyn stanów generowanych przez kompilator, możliwe jest tworzenie kodu o strukturze sekwencyjnej, co znacząco upraszcza zarządzanie współbieżnością w porównaniu do tradycyjnych rozwiązań opartych na interfejsach typu Callback czy bibliotece

RxJava. System ten wykorzystuje wyspecjalizowane dyspaczery (`Dispatchers`) do wyboru odpowiedniego kontekstu wykonawczego (np. operacje wejścia/wyjścia lub obliczenia procesora) oraz reaktywne strumienie `Flow` do asynchronicznej emisji danych w czasie.

## 1.1. Dlaczego Coroutines?

Analiza problematyki asynchroniczności w systemie Android wskazuje na krytyczne znaczenie unikania blokowania wątku głównego (UI), co w przeciwnym razie prowadzi do błędów typu Application Not Responding (ANR). Wprowadzenie korutyn jako lekkich, współbieżnych i przerywalnych zadań stanowi rozwiązanie problemów związanych z wcześniejszymi, nadmiarowymi technologiami, takimi jak AsyncTask czy tradycyjne handlery, które były podatne na wycieki pamięci i trudne w testowaniu.

| | |
|---|---|
| **Problem** | Operacje sieciowe i bazodanowe blokują wątek UI. Aplikacja się zawiesza (ANR, Application Not Responding). Wątek główny (UI/Main) powinien wykonywać operacje krótsze niż 16 ms. |
| **Stare podejście** | AsyncTask (deprecated), Handler, Thread to dużo powtarzalnego kodu (boilerplate), trudne w testowaniu, podatne na wycieki pamięci (memory leaks). |
| **Coroutines** | Lekkie współbieżne zadania (możliwe setki tysięcy jednocześnie), nieblokujące, anulowalne (cancellable), łatwe do testowania. Kompilator przekształca kod do postaci maszyny stanów (state machine) – brak kosztów tworzenia wątków systemowych. |
| **suspend fun** | Funkcja oznaczona jako suspend może zostać wstrzymana bez blokowania wątku. Kompilator Kotlin generuje dla niej maszynę stanów (state machine). Może być wywoływana tylko z poziomu innej funkcji suspend lub w obrębie coroutine. |
| **Scope** | Definiuje cykl życia coroutine. Coroutine żyje tak długo jak przypisany jej scope (np. ViewModelScope). Automatyczne anulowanie (auto-cancel) przy zniszczeniu ViewModel czyli brak wycieków pamięci. |
| **Dispatcher** | Określa, na jakim wątku wykonywana jest coroutine:<br>• Dispatchers.Main – wątek UI (np. aktualizacja widoku),<br>• Dispatchers.IO – operacje I/O (sieć, baza danych, pliki),<br>• Dispatchers.Default – operacje CPU (np. sortowanie, parsowanie). |

## 1.2. Przykłady Coroutines

Zastosowanie korutyn w praktyce programistycznej opiera się na wykorzystaniu zakresów związanych z cyklem życia komponentów, takich jak viewModelScope, co gwarantuje automatyczne anulowanie zadań w momencie zniszczenia obiektu. Coroutine uruchomiona w viewModelScope jest powiązana z cyklem życia ViewModel. Mechanizm ten pozwala na sekwencyjne wywoływanie operacji długotrwałych, takich jak pobieranie danych z repozytorium, i bezpośrednie aktualizowanie stanu interfejsu użytkownika w sposób bezpieczny dla stabilności aplikacji.

**Coroutines basics** **Kotlin**

```kotlin
viewModelScope.launch {

    val tasks = repository.getTasks()

    _uiState.value = tasks
}
```

### 1.3. Reaktywny strumień danych Flow

Mechanizm `Flow` definiowany jest jako asynchroniczny strumień danych emitujący wartości w czasie, co umożliwia budowę w pełni reaktywnych interfejsów użytkownika. Dzięki natywnej integracji z biblioteką Room, każda modyfikacja w bazie danych może być automatycznie propagowana do warstwy prezentacji poprzez transformacje takie jak mapowanie, filtrowanie czy łączenie wielu strumieni za pomocą operatora `combine`.Room potrafi zwracać wyniki zapytań w postaci Flow. Room zwraca Flow<List<Task>>, co oznacza że każda zmiana w bazie aktualizuje UI.

## 2. Architektura Room Database

Room to biblioteka Jetpack stanowiąca zaawansowaną warstwę abstrakcji nad systemem SQLite, oferując mechanizm mapowania obiektowo-relacyjnego (ORM) dostosowany do potrzeb platformy Android. Eliminuje konieczność pisania powtarzalnego kodu (boilerplate SQL), weryfikuje zapytania na etapie kompilacji oraz integruje się natywnie z mechanizmami Flow i Coroutines. Architektura Room opiera się na trzech kluczowych komponentach: encjach definiujących strukturę tabel bazy danych, obiektach dostępu do danych (DAO) odpowiedzialnych za deklarację zapytań SQL oraz klasie bazy danych zarządzającej instancją systemu i dostępem do danych.

System Room pełni rolę inteligentnego pośrednika pomiędzy obiektowym modelem danych, a relacyjną strukturą SQLite, automatyzując proces mapowania rekordów na obiekty języka Kotlin. Dzięki ścisłej integracji z kompilatorem, deklaratywne anotacje są przekształcane w zoptymalizowany kod w języku Java, co zapewnia zarówno wysoką wydajność, jak i bezpieczeństwo typów poprzez eliminację błędów składniowych już na etapie kompilacji. W porównaniu do bezpośredniego wykorzystania SQLite, Room oferuje wyższy poziom abstrakcji, zwiększając stabilność, czytelność oraz łatwość utrzymania warstwy trwałości danych, szczególnie w złożonych aplikacjach o charakterze produkcyjnym.

**Podstawy Flow** — Kotlin

```kotlin
// suspend fun zwraca jedną wartość
suspend fun getUser(): User = ...

// Flow emituje wiele wartości w czasie
fun getTasks(): Flow<List<Task>> = ...

// Operatory Flow
val tasksFlow: Flow<List<Task>> = repository.getTasks()

// map – transformacja każdej emitowanej wartości
val sortedTasksFlow: Flow<List<Task>> =
    tasksFlow.map { tasks ->
        tasks.sortedBy { it.text }
    }

// filter (przez map) – filtrowanie elementów wewnątrz emisji
val activeTasksFlow: Flow<List<Task>> =
    tasksFlow.map { tasks ->
        tasks.filter { !it.isDone }
    }

// combine – łączenie dwóch Flow
val filteredTasksFlow: Flow<List<Task>> =
    combine(tasksFlow, filterFlow) { tasks, filter ->
        when (filter) {
```

**Podstawy Flow** **Kotlin**

```
        Filter.ALL    -> tasks
        Filter.ACTIVE -> tasks.filter { !it.isDone }
        Filter.DONE   -> tasks.filter { it.isDone }
      }
    }

// ViewModel – StateFlow z bazy danych
class TaskViewModel(
    private val repository: TaskRepository
) : ViewModel() {

    // stateIn – konwersja Flow → StateFlow
    // zawsze posiada wartość i jest thread-safe
    val tasksState: StateFlow<List<Task>> =
        repository.getAllTasks()
            .stateIn(
                scope = viewModelScope,
                started = SharingStarted.WhileSubscribed(5_000),
                initialValue = emptyList()
            )
}

// Composable (Jetpack Compose)
// Automatyczna aktualizacja UI przy zmianach danych (np. Room)
@Composable
fun TaskScreen(viewModel: TaskViewModel) {
    val tasks by viewModel.tasksState.collectAsStateWithLifecycle()
    // UI wykorzystujący tasks
}
```

## 2.1. Trzy składniki Room

Struktura biblioteki Room opiera się na triadzie komponentów: encjach definiujących schemat tabel, obiektach DAO (Data Access Object) zawierających logikę zapytań oraz klasie bazy danych stanowiącej główny punkt dostępowy. System ten zapewnia bezpieczeństwo typów i automatyczne mapowanie rekordów na obiekty języka Kotlin, a także umożliwia obsługę typów nieobsługiwanych natywnie, jak np. wyliczenia (enums), poprzez mechanizm konwerterów.

**@Entity - tabela bazy danych**

```
@Entity(tableName = "tasks")                    ← anotacja klasy
data class TaskEntity(                          ← Kotlin data class
    @PrimaryKey val id: String,                 ← klucz główny
    @ColumnInfo(name = "task_text") val text: String,
    val isDone: Boolean = false,                ← nazwa kolumny = nazwa pola
    val priority: Int = 1,                      ← enum → Int (Room nie obsługuje enum natywnie)
    val createdAt: Long = System.currentTimeMillis()
)
```

**@Dao - Data Access Object (interfejs)**

```
@Dao                                            ← anotacja interfejsu
```

```kotlin
interface TaskDao {
    @Query("SELECT * FROM tasks ORDER BY createdAt DESC")
    fun getAllTasks(): Flow<List<TaskEntity>>                    ← Flow = reaktywne zapytanie

    @Query("SELECT * FROM tasks WHERE id = :taskId")
    suspend fun getTaskById(taskId: String): TaskEntity?

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertTask(task: TaskEntity)

    @Update  suspend fun updateTask(task: TaskEntity)
    @Delete  suspend fun deleteTask(task: TaskEntity)

    @Query("DELETE FROM tasks WHERE isDone = 1")
    suspend fun deleteCompletedTasks()
}
```

**@Database - abstrakcyjna klasa bazy danych**

```kotlin
@Database(
    entities = [TaskEntity::class],                  ← lista tabel
    version = 1,                                     ← wersja schematu
    exportSchema = true                              ← zapisuje schemat do JSON-a (wymagane!)
)
abstract class AppDatabase : RoomDatabase() {
    abstract fun taskDao(): TaskDao                  ← Room generuje implementację

    companion object {
        @Volatile private var INSTANCE: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase {
            return INSTANCE ?: synchronized(this) {
                Room.databaseBuilder(context, AppDatabase::class.java, "task_database")
                    .build().also { INSTANCE = it }
            }
        }
    }
}
```

**Jak Room przetwarza @Query?**

Kompilator Room (kapt/ksp) czyta anotacje @Query w czasie kompilacji i generuje implementację w Javie. Jeśli SQL zawiera błąd (literówka w nazwie tabeli/kolumny). BUILD FAILED, a nie crash w runtime. To jest główna przewaga Room nad czystym SQLite.

Zapytanie zwracające Flow<List<T>> jest 'żywym' zapytaniem. Room automatycznie ponownie wykonuje je gdy dane w tabeli się zmienią i emituje nowy wynik do wszystkich obserwatorów.

## 2.2. Porównanie SQLite vs Room

| Cecha | SQLite | Room |
|---|---|---|
| Poziom abstrakcji | Niski, bezpośrednie zapytania SQL | Wyższy, ORM nad SQLite |
| Mapowanie obiektów | Ręczne | Automatyczne |
| Weryfikacja zapytań | Runtime | Compile time |
| Integracja z Coroutines | Brak natywnej | Pełna integracja |

| Cecha | SQLite | Room |
|---|---|---|
| Integracja z Flow | Brak | Tak |
| Migracje | Ręczne | Obsługiwane przez Room |

Room w rzeczywistości nadal korzysta z SQLite, jednak znacząco upraszcza dostęp do danych i pracę z bazą poprzez integrację z nowoczesną architekturą aplikacji Android oraz narzędziami takimi jak Coroutines, Flow czy Paging. Analiza porównawcza wskazuje na jego wyraźną przewagę nad surowym SQLite w obszarach mapowania obiektowego, weryfikacji zapytań oraz integracji z frameworkami asynchronicznymi. Podczas gdy SQLite wymaga ręcznej obsługi kursorów i mapowania pól, Room automatyzuje te operacje, redukując ryzyko błędów typu runtime. Kluczową różnicą jest również natywne wsparcie dla mechanizmów Flow i Coroutines, co umożliwia budowę interfejsów użytkownika reagujących na zmiany w bazie danych bez konieczności ręcznego odświeżania zapytań.

# 3. Konfiguracja Room w projekcie

Ten fragment instrukcji skupia się na technicznej konfiguracji projektu przy użyciu plików libs.versions.toml oraz build.gradle.kts. Opisuje proces dodawania zależności dla Room oraz konfigurację pluginu KSP (Kotlin Symbol Processing), który służy jako nowoczesny generator kodu dla bazy danych i automatyzuje generowanie kodu implementacyjnego dla interfejsów DAO. Prawidłowa konfiguracja musi uwzględniać ścisłą zgodność wersji wtyczki KSP z wersją kompilatora Kotlin oraz określenie lokalizacji dla eksportowanych schematów bazy danych.

## 3.1. dodaj Room w libs.versions.toml

Proces konfiguracji środowiska programistycznego rozpoczyna się od deklaratywnego zdefiniowania wersji bibliotek oraz wtyczek w centralnym pliku katalogu wersji. Podejście to zapewnia spójność zależności w całym projekcie i ułatwia zarządzanie aktualizacjami kluczowych komponentów, takich jak room-runtime czy room-compiler.

**libs.versions.toml** TOML

```toml
[versions] # Wersje
room = "2.6.1"
ksp  = "2.0.21-1.0.28"

[libraries] # Biblioteki
room-runtime  = { group = "androidx.room", name = "room-runtime",  version.ref = "room" }
room-ktx      = { group = "androidx.room", name = "room-ktx",      version.ref = "room" }
room-compiler = { group = "androidx.room", name = "room-compiler", version.ref = "room" }
room-testing  = { group = "androidx.room", name = "room-testing",  version.ref = "room" }

[plugins] # Pluginy
ksp = { id = "com.google.devtools.ksp", version.ref = "ksp" }
```

## 3.2. Pluginy KSP i zależności build.gradle.kts

Wykorzystanie procesora symboli KSP (Kotlin Symbol Processing) pozwala na wydajne generowanie kodu DAO w czasie kompilacji, co wymaga precyzyjnej konfiguracji w skryptach Gradle. Kluczowym aspektem jest zachowanie ścisłej zgodności wersji wtyczki KSP z wersją kompilatora języka Kotlin, aby uniknąć błędów synchronizacji projektu.

**build.gradle.kts (fragment)** **Kotlin DSL**

```
plugins {                                    // istniejące pluginy...
   alias(libs.plugins.ksp)                   // KSP dla Room
}
ksp {                                        // Konfiguracja KSP (schemat bazy)
   arg("room.schemaLocation", "$projectDir/schemas")
}
dependencies {                               // Zależności
   implementation(libs.room.runtime)          // Room
   implementation(libs.room.ktx)
   ksp(libs.room.compiler)                   // KSP – generator kodu DAO
   testImplementation(libs.room.testing)     // Testy Room
}
```

**Wersja KSP musi pasować do Kotlin!**

KSP ma format: <kotlin-version>-<ksp-version>. Dla Kotlin 2.0.21 użyj KSP 2.0.21-1.0.28.

Sprawdź aktualne wersje na: https://github.com/google/ksp/releases.

Niezgodność wersji objawia się błędem: *"KSP and Kotlin versions don't match"* podczas Gradle Sync.

# 4. Implementacja warstwy danych

Wykonamy teraz praktyczną implementację warstwy danych zaprojektowanej zgodnie z zasadami Clean Architecture, z wyraźnym podziałem na komponenty odpowiedzialne za lokalne składowanie danych. Struktura została rozbita na cztery pliki w pakiecie data/, co odzwierciedla separację warstw i zapewnia czytelność oraz modularność rozwiązania. W implementacji uwzględniono definicję encji TaskEntity wraz z indeksami optymalizacyjnymi, interfejs DAO wykorzystujący funkcje zawieszalne oraz operacje typu Upsert, a także singleton bazy danych AppDatabase zarządzający dostępem do zasobów.

Kluczowym elementem jest mechanizm mapowania danych pomiędzy encjami bazodanowymi, a modelami domenowymi, który umożliwia pełną izolację warstwy logiki biznesowej od technologii przechowywania danych. Dzięki temu ViewModel pozostaje niezależny od szczegółów implementacyjnych biblioteki Room, natomiast Repository pełni rolę warstwy abstrakcji ukrywającej źródło danych. Takie podejście zwiększa elastyczność systemu, ułatwia jego testowanie oraz pozwala na potencjalną zmianę mechanizmu persystencji, bez wpływu na pozostałe warstwy aplikacji.

**Struktura projektu** **Kotlin**

```
[versions] # Wersje app/src/main/java/pl/edu/prz/taskapp/
├── data/
│   ├── local/
│   │   ├── TaskEntity.kt          ← @Entity - definicja tabeli
│   │   ├── TaskDao.kt             ← @Dao - zapytania SQL
│   │   └── AppDatabase.kt         ← @Database - singleton bazy
│   └── repository/
│       ├── TaskRepository.kt      ← interfejs (abstrakcja)
│       └── TaskRepositoryImpl.kt  ← implementacja z Room
```

**Struktura projektu** **Kotlin**

```
├── domain/
│   └── model/
│       └── Task.kt          ← domain model (używany w ViewModel/UI)
├── ui/ ...                  ← ekrany z Lab 2 (bez zmian)
└── TaskViewModel.kt         ← rozszerzona o Repository
```

## 4.1. Definicja tabeli TaskEntity

Definicja modelu danych w bazie lokalnej odbywa się poprzez tworzenie klas danych opatrzonych adnotacjami określającymi klucze główne, nazwy kolumn oraz indeksy przyspieszające sortowanie. Wykorzystanie domyślnych wartości w schemacie tabeli pozwala na zachowanie spójności danych nawet w przypadku braku pełnych informacji podczas zapisu rekordu.

**data/local//TaskEntity.kt** **Kotlin**

```kotlin
package pl.edu.prz.taskapp.data.local
import androidx.room.ColumnInfo
import androidx.room.Entity
import androidx.room.Index
import androidx.room.PrimaryKey

@Entity(
    tableName = "tasks",
    indices = [
        Index(value = ["created_at"]) // indeks przyspiesza sortowanie
    ]
)
data class TaskEntity(
    @PrimaryKey
    val id: String, // UUID jako String

    @ColumnInfo(name = "task_text")
    val text: String,

    @ColumnInfo(name = "is_done", defaultValue = "0")
    val isDone: Boolean = false,

    @ColumnInfo(name = "priority", defaultValue = "1")
    val priority: Int = 1, // 0=LOW, 1=NORMAL, 2=HIGH

    @ColumnInfo(name = "created_at")
    val createdAt: Long = System.currentTimeMillis(),

    @ColumnInfo(name = "updated_at")
    val updatedAt: Long = System.currentTimeMillis()
)
// TypeConverter - jeśli potrzebujesz enum w bazie
// Room domyślnie nie obsługuje enumów.
```

## 4.2. TaskDao (Data Access Object)

Interfejs Data Access Object (DAO) stanowi fundament komunikacji z bazą danych, pozwalając na precyzyjne definiowanie interakcji z zasobami przy użyciu metod opatrzonych adnotacjami takimi jak `@Insert`, `@Update` czy `@Delete`. Wykorzystanie zapytań SQL w adnotacjach `@Query`

umożliwia realizację zaawansowanych operacji selekcji danych, których poprawność składniowa jest weryfikowana przez system Room już na etapie kompilacji kodu. Implementacja oparta na funkcjach zawieszalnych (`suspend`) oraz asynchronicznych strumieniach `Flow` gwarantuje nieblokujący charakter operacji wejścia/wyjścia oraz pozwala na budowę w pełni reaktywnej warstwy danych, która automatycznie propaguje wszelkie zmiany do obserwujących ją komponentów aplikacji.

**TaskDao.kt** **Kotlin**

```kotlin
@Dao
interface TaskDao {

  @Query("SELECT * FROM tasks ORDER BY createdAt DESC")
  fun getAll(): Flow<List<TaskEntity>>

  @Query("SELECT * FROM tasks WHERE id = :id")
  suspend fun getById(id: String): TaskEntity?

    @Insert
  suspend fun insert(task: TaskEntity)

    @Update
  suspend fun update(task: TaskEntity)

  @Query("DELETE FROM tasks WHERE id = :id")
  suspend fun delete(id: String)
}
```

## 4.3. Singleton bazy danych na przykładzie AppDatabase

Zarządzanie instancją bazy danych realizowane jest poprzez wzorzec projektowy Singleton, co zapobiega nadmiernemu zużyciu zasobów poprzez utrzymywanie tylko jednego połączenia z bazą SQLite. Klasa ta odpowiada za konfigurację budowniczego bazy, definiowanie dostępnych obiektów DAO oraz zarządzanie wersjonowaniem schematu.

**data/local/AppDatabase.kt** **Kotlin**

```kotlin
package pl.edu.prz.taskapp.data.local

import android.content.Context
import androidx.room.*
import androidx.room.migration.Migration
import androidx.sqlite.db.SupportSQLiteDatabase

@Database(
    entities = [TaskEntity::class],
    version = 1,
    exportSchema = true
)
abstract class AppDatabase : RoomDatabase() {

    abstract fun taskDao(): TaskDao

    companion object {
        @Volatile
        private var INSTANCE: AppDatabase? = null
```

**data/local/AppDatabase.kt** Kotlin

```
        fun getInstance(context: Context): AppDatabase =
            INSTANCE ?: synchronized(this) {
                INSTANCE ?: buildDatabase(context).also { INSTANCE = it }
            }

        private fun buildDatabase(context: Context): AppDatabase =
            Room.databaseBuilder(
                context.applicationContext,
                AppDatabase::class.java,
                "task_database.db"
            )
                // .addMigrations(MIGRATION_1_2)
                // .fallbackToDestructiveMigration() // tylko DEV
                .build()

        // Przykład migracji 1 → 2
        val MIGRATION_1_2 = object : Migration(1, 2) {
            override fun migrate(db: SupportSQLiteDatabase) {
                db.execSQL(
                    "ALTER TABLE tasks ADD COLUMN category TEXT NOT NULL DEFAULT 'gene-
ral'"
                )
            }
        }
    }
}
```

## 4.4. Domain model: Task

Zasada separacji warstw w architekturze Clean Architecture wymaga odseparowania modeli bazodanowych od modeli domenowych wykorzystywanych w logice biznesowej. Proces ten realizowany jest za pomocą funkcji rozszerzających (mappers), które dokonują dwukierunkowej konwersji danych, zapewniając niezależność interfejsu użytkownika od fizycznej struktury bazy danych.

**domain/model/Task.kt** (domain model) Kotlin

```
package pl.edu.prz.taskapp.domain.model
import java.util.UUID

enum class Priority(val value: Int) {
    LOW(0),
    NORMAL(1),
    HIGH(2)
}

data class Task(
    val id: String = UUID.randomUUID().toString(),
    val text: String,
    val isDone: Boolean = false,
    val priority: Priority = Priority.NORMAL,
    val createdAt: Long = System.currentTimeMillis()
)
// ── Mapowanie Entity <-> Domain model (w Repository) ──────────────────────

fun TaskEntity.toDomainModel(): Task = Task(
```

**domain/model/Task.kt** (domain model) **Kotlin**

```
    id        = this.id,
    text      = this.text,
    isDone    = this.isDone,
    priority  = Priority.entries.find { it.value == this.priority } ?: Priority.NOR-
MAL,
    createdAt = this.createdAt
)

fun Task.toEntity(): TaskEntity = TaskEntity(
    id        = this.id,
    text      = this.text,
    isDone    = this.isDone,
    priority  = this.priority.value,
    createdAt = this.createdAt,
    updatedAt = System.currentTimeMillis()
)
```

## 4.5. Warstwa Repository: abstrakcja źródła danych

Warstwa repozytorium pełni rolę pośrednika, który enkapsuluje operacje na obiektach DAO i udostępnia warstwom wyższym czyste, domenowe modele danych. Dzięki takiemu podejściu, logika biznesowa pozostaje nieświadoma szczegółów implementacyjnych bazy danych, co ułatwia wymianę źródeł danych lub ich testowanie.

**data/repository/TaskRepository.kt** **Kotlin**

```
class TaskRepository(
    private val dao: TaskDao
) {
    fun getTasks(): Flow<List<Task>> =
        dao.getAll().map { list ->
            list.map { it.toDomain() }
        }
    suspend fun save(task: Task) {
        dao.insert(task.toEntity())
    }
    suspend fun delete(id: String) {
        dao.delete(id)
    }
}
```

## 4.6. Operacja Upsert

Zastosowanie operacji typu `Upsert` pozwala na atomowe wstawienie nowego rekordu lub jego aktualizację (w przypadku konfliktu klucza głównego). Jest to rozwiązanie optymalizujące procesy zapisu, eliminujące potrzebę ręcznego sprawdzania istnienia obiektu przed wykonaniem operacji bazodanowej.

**Kotlin**

```
@Dao
interface TaskDao {
    @Upsert
    suspend fun upsert(task: TaskEntity)
```

### 4.7. Obsługa dużych zbiorów danych przez Paging

Efektywne przetwarzanie obszernych zestawów rekordów realizowane jest poprzez mechanizm stronicowania, który ładuje dane małymi partiami w miarę zapotrzebowania. Wykorzystanie `PagingSource` w zapytaniach DAO minimalizuje zużycie pamięci operacyjnej i poprawia płynność przewijania list w interfejsie graficznym.

**Kotlin**

```kotlin
@Query("SELECT * FROM tasks ORDER BY createdAt DESC")
fun pagingSource(): PagingSource<Int, TaskEntity>
```

## 5. ViewModel z Repository

Implementacja warstwy prezentacji opiera się na wstrzyknięciu instancji `TaskRepository` do konstruktora klasy `TaskViewModel` przy użyciu niestandardowej fabryki `ViewModelFactory`. Kluczowym mechanizmem jest transformacja strumienia danych `tasks: Flow<List<Task>>` pobieranego z repozytorium do reaktywnego stanu `uiState: StateFlow<List<Task>>` za pomocą operatora `stateIn`, co zapewnia bezpieczeństwo wątkowe i optymalne zarządzanie zasobami dzięki parametrowi `SharingStarted.WhileSubscribed(5000)`. Tak skonstruowany model widoku umożliwia komponentom Jetpack Compose bezpośrednią subskrypcję stanu, podczas gdy operacje modyfikujące dane, takie jak `upsertTask(task: Task)` czy `deleteTask(task: Task)`, są delegowane do repozytorium wewnątrz zakresu `viewModelScope`, co gwarantuje pełną separację logiki domenowej od szczegółów implementacyjnych bazy danych Room.

### 5.1. TaskViewModel, wersja z Room

Model widoku zarządza stanem interfejsu użytkownika poprzez agregację strumieni danych z repozytorium w thread-safe obiekt `StateFlow`. Implementacja obejmuje obsługę akcji użytkownika wewnątrz dedykowanych zakresów korutyn oraz mechanizmy filtrowania i wyszukiwania z zastosowaniem opóźnienia (*debounce*) w celu optymalizacji zapytań.

**TaskViewModel.kt** (z Room) **Kotlin**

```kotlin
package pl.edu.prz.taskapp
import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.FlowPreview
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch
import pl.edu.prz.taskapp.data.repository.TaskRepository
import pl.edu.prz.taskapp.domain.model.Task
import pl.edu.prz.taskapp.domain.model.Priority
enum class SortOrder { DATE_DESC, DATE_ASC, PRIORITY_DESC }

data class TaskUiState(
    val tasks: List<Task> = emptyList(),
    val filter: Filter = Filter.ALL,
    val searchQuery: String = "",
    val sortOrder: SortOrder = SortOrder.DATE_DESC,
    val isLoading: Boolean = false,
    val activeCount: Int = 0,
    val totalCount: Int = 0,
```

**TaskViewModel.kt** (z Room) **Kotlin**

```
    val snackbarMessage: String? = null
)
@OptIn(ExperimentalCoroutinesApi::class, FlowPreview::class)
class TaskViewModel(private val repository: TaskRepository) : ViewModel() {
    private val _filter = MutableStateFlow(Filter.ALL)
    private val _searchQuery = MutableStateFlow("")
    private val _sortOrder = MutableStateFlow(SortOrder.DATE_DESC)

    val uiState: StateFlow<TaskUiState> = combine(
        repository.getAllTasks(),
        repository.getActiveTaskCount(),
        _filter,
        _searchQuery.debounce(300),
        _sortOrder
    ) { tasks, activeCount, filter, query, sortOrder ->
        val filtered = tasks
            .filter { task ->
                val matchesFilter = when (filter) {
                    Filter.ALL -> true
                    Filter.ACTIVE -> !task.isDone
                    Filter.DONE -> task.isDone
                }
                matchesFilter && (query.isBlank() || task.text.contains(query, ignore-
Case = true))
            }
            .let { list ->
                when (sortOrder) {
                    SortOrder.DATE_DESC -> list.sortedByDescending { it.createdAt }
                    SortOrder.DATE_ASC -> list.sortedBy { it.createdAt }
                    SortOrder.PRIORITY_DESC -> list.sortedByDescending { it.priority-
.value }
                }
            }
        TaskUiState(
            tasks = filtered,
            filter = filter,
            searchQuery = query,
            sortOrder = sortOrder,
            activeCount = activeCount,
            totalCount = tasks.size
        )
    }.stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5_000),
        initialValue = TaskUiState(isLoading = true)
    )
    // ── Akcje ──────────────────────────────
    fun addTask(text: String, priority: Priority = Priority.NORMAL) {
        if (text.isBlank()) return
        viewModelScope.launch {
            repository.saveTask(Task(text = text.trim(), priority = priority))
        }
    }
    fun toggleTask(taskId: String) = viewModelScope.launch { repository.toggle-
Task(taskId) }
    fun deleteTask(taskId: String) = viewModelScope.launch { repository.delete-
Task(taskId) }
```

**TaskViewModel.kt** (z Room) **Kotlin**

```
    fun deleteCompleted() = viewModelScope.launch { repository.deleteCompletedTasks() }

    fun setFilter(filter: Filter) { _filter.value = filter }
    fun setSearchQuery(query: String) { _searchQuery.value = query }
    fun setSortOrder(order: SortOrder) { _sortOrder.value = order }

    // ── Factory bez Hilt ──────────────────
    companion object {
        fun factory(repository: TaskRepository): ViewModelProvider.Factory =
            object : ViewModelProvider.Factory {
                @Suppress("UNCHECKED_CAST")
                override fun <T : ViewModel> create(modelClass: Class<T>): T =
                    TaskViewModel(repository) as T
            }
    }
}
```

## 5.2. Inicjalizacja w MainActivity

Proces inicjalizacji aplikacji obejmuje ręczne wstrzykiwanie zależności (Manual DI), gdzie w głównej aktywności tworzone są instancje bazy danych, DAO oraz repozytorium. Przekazanie tych komponentów do modelu widoku odbywa się poprzez specjalistyczną fabrykę (`ViewModelFactory`), co jest krokiem poprzedzającym wdrożenie automatycznych systemów wstrzykiwania, takich jak Hilt.

**MainActivity.kt + AppNavigation.kt** (fragment) **Kotlin**

```
// MainActivity.kt
package pl.edu.prz.taskapp

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.activity.enableEdgeToEdge
import androidx.lifecycle.ViewModelProvider
import pl.edu.prz.taskapp.data.local.AppDatabase
import pl.edu.prz.taskapp.data.repository.TaskRepositoryImpl
import pl.edu.prz.taskapp.ui.theme.TaskAppTheme

class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()

        // 1. Inicjalizacja bazy danych
        val database = AppDatabase.getInstance(applicationContext)
        val dao = database.taskDao()
        val repository = TaskRepositoryImpl(dao)

        // 2. Fabryka ViewModel z repository
        val viewModelFactory = TaskViewModel.factory(repository)

        setContent {
            TaskAppTheme {
```

**MainActivity.kt + AppNavigation.kt** (fragment) **Kotlin**

```
                // 3. Przekazanie factory do AppNavigation
                AppNavigation(viewModelFactory = viewModelFactory)
            }
        }
    }
}
```

**Hilt vs ręczne DI**

W tym ćwiczeniu używamy ręcznego wstrzykiwania zależności (manual DI) - przekazujemy Repository przez konstruktor/factory. To prostszy sposób bez dodatkowych bibliotek.

W projektach produkcyjnych zalecane jest Hilt (Dagger) lub Koin. Hilt integruje się z ViewModel przez @HiltViewModel + @Inject constructor(repository: TaskRepository) - eliminuje ViewModelFactory. Hilt będzie tematem Ćwiczenia 5.

# 6. Migracje schematu bazy danych

Ewolucja struktury bazy danych w cyklu życia aplikacji wymaga stosowania mechanizmów migracyjnych, które pozwalają na bezpieczne przekształcenie schematu bez utraty danych użytkownika. Gdy zmieniasz strukturę tabeli (dodajesz kolumnę, zmieniasz typ), musisz napisać migrację. Proces ten obejmuje definicję obiektów `Migration`, zawierających skrypty SQL (np. `ALTER TABLE`), oraz inkrementację wersji bazy danych w anotacji `@Database`. Brak poprawnie zdefiniowanej ścieżki migracji przy zmianie struktury tabel prowadzi do krytycznych błędów i awarii aplikacji podczas startu (rzuca IllegalStateException i aplikacja crashuje przy starcie.).

## 6.1. Kiedy wymagana jest migracja?

Ewolucja aplikacji wymuszająca zmiany w strukturze danych (np. dodanie kolumny czy zmiana typu) wymaga zdefiniowania ścieżek migracyjnych w celu uniknięcia awarii systemu. Proces ten polega na implementacji obiektów `Migration`, które wykonują odpowiednie polecenia SQL ALTER TABLE, zachowując integralność istniejących danych użytkownika.

| | |
|---|---|
| Dodanie kolumny | `ALTER TABLE tasks ADD COLUMN category TEXT NOT NULL DEFAULT 'general'` |
| Zmiana nazwy kolumny | Room nie obsługuje RENAME COLUMN w starej SQLite. Utwórz nową tabelę, skopiuj dane, usuń starą. |
| Usunięcie kolumny | Brak DROP COLUMN (SQLite < 3.35) → utwórz nową tabelę bez kolumny, skopiuj dane, usuń starą |
| Dodanie indeksu | CREATE INDEX idx_tasks_created ON tasks(createdAt) (bez rekonstrukcji tabeli) |
| Nowa tabela | `CREATE TABLE tasks_new (id TEXT PRIMARY KEY NOT NULL, text TEXT NOT NULL, isDone INTEGER NOT NULL, priority INTEGER NOT NULL, createdAt INTEGER NOT NULL)` |

**Migracja Room 1→2** **Kotlin**

```
// 1. Zaktualizuj wersję w AppDatabase
```

**Migracja Room 1→2** **Kotlin**

```kotlin
@Database(entities = [TaskEntity::class], version = 2, exportSchema = true)
abstract class AppDatabase : RoomDatabase() { /* ... */ }

// 2. Dodaj pole do TaskEntity
@ColumnInfo(name = "category", defaultValue = "general")
val category: String = "general"

// 3. Napisz migrację 1 → 2
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL(
            "ALTER TABLE tasks ADD COLUMN category TEXT NOT NULL DEFAULT 'general'"
        )
    }
}

// 4. Zarejestruj migrację przy tworzeniu bazy
Room.databaseBuilder(
    context.applicationContext,
    AppDatabase::class.java,
    "task_database.db"
)
    .addMigrations(MIGRATION_1_2)
    .build()
```

# 7. Testy jednostkowe DAO

Room umożliwia testowanie DAO w izolacji przy użyciu bazy danych in-memory. Testy in-memory są szybkie, nie wymagają emulatora (uruchamiane jako zwykłe unit testy JVM) i nie zostawiają danych na urządzeniu. Takie podejście umożliwia szybkie i izolowane testowanie logiki zapytań SQL bez wpływu na trwałe dane urządzenia. Do testowania asynchronicznych strumieni Flow emitowanych przez DAO stosuje się wyspecjalizowane biblioteki, takie jak Turbine, które pozwalają na sekwencyjną weryfikację emitowanych elementów.

## 7.1. Konfiguracja testu

Weryfikacja poprawności logiki dostępu do danych odbywa się w izolowanym środowisku przy użyciu bazy danych działającej w pamięci RAM (*in-memory*). Takie podejście gwarantuje szybkość testów, brak trwałych skutków ubocznych na urządzeniu oraz umożliwia precyzyjną kontrolę nad stanem początkowym bazy przed wykonaniem asercji.

**test/java/.../data/local/TaskDaoTest.kt** **Kotlin**

```kotlin
package pl.edu.prz.taskapp.data.local

import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import androidx.test.ext.junit.runners.AndroidJUnit4
import app.cash.turbine.test
import kotlinx.coroutines.test.runTest
import org.junit.After
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
```

**test/java/.../data/local/TaskDaoTest.kt** **Kotlin**

```kotlin
import kotlin.test.assertEquals
import kotlin.test.assertNotNull
import kotlin.test.assertNull

@RunWith(AndroidJUnit4::class)
class TaskDaoTest {

    private lateinit var database: AppDatabase
    private lateinit var dao: TaskDao

    @Before
    fun setUp() {
        database = Room.inMemoryDatabaseBuilder(
            ApplicationProvider.getApplicationContext(),
            AppDatabase::class.java
        )
            .allowMainThreadQueries() // tylko w testach
            .build()
        dao = database.taskDao()
    }

    @After
    fun tearDown() {
        database.close()
    }
    // ── Testy ──────────────────────────────
    @Test
    fun insertTask_getById_returnsCorrectTask() = runTest {
        val task = TaskEntity(id = "1", text = "Test task", isDone = false, priority = 1)
        dao.insertTask(task)

        val result = dao.getTaskById("1")
        assertNotNull(result)
        assertEquals("Test task", result.text)
        assertEquals(false, result.isDone)
    }
    @Test
    fun toggleTask_updatesIsDoneStatus() = runTest {
        val task = TaskEntity(id = "2", text = "Toggle me", isDone = false, priority = 1)
        dao.insertTask(task)
        dao.updateTaskStatus(id = "2", isDone = true)

        val result = dao.getTaskById("2")
        assertEquals(true, result?.isDone)
    }
    @Test
    fun deleteTask_removesFromDatabase() = runTest {
        val task = TaskEntity(id = "3", text = "Delete me", isDone = false, priority = 1)
        dao.insertTask(task)
        dao.deleteTaskById("3")

        val result = dao.getTaskById("3")
        assertNull(result)
    }
    @Test
    fun deleteCompleted_removesOnlyDoneTasks() = runTest {
        dao.insertTasks(
            listOf(
```

**test/java/.../data/local/TaskDaoTest.kt** **Kotlin**

```
                TaskEntity(id = "a", text = "Active", isDone = false, priority = 1),
                TaskEntity(id = "b", text = "Done 1", isDone = true, priority = 1),
                TaskEntity(id = "c", text = "Done 2", isDone = true, priority = 1)
            )
        )
        val deletedCount = dao.deleteCompletedTasks()
        assertEquals(2, deletedCount)

        val remaining = dao.getTaskById("a")
        assertNotNull(remaining)
        assertNull(dao.getTaskById("b"))
    }
    @Test
    fun getAllTasks_returnsTasksInDescendingOrder() = runTest {
        dao.insertTask(TaskEntity(id = "x1", text = "First", isDone = false, priority =
1, createdAt = 1000))
        dao.insertTask(TaskEntity(id = "x2", text = "Second", isDone = false, priority =
1, createdAt = 2000))
        dao.getAllTasks().test {
            val tasks = awaitItem()
            assertEquals(2, tasks.size)
            assertEquals("Second", tasks[0].text) // DESC: nowsze pierwsze
            cancelAndIgnoreRemainingEvents()
        }
    }
}
```

**Biblioteka Turbine do testowania Flow**

Turbine (https://github.com/cashapp/turbine) to popularna biblioteka do testowania Flow w Kotlinie. Dodaj zależność: testImplementation("app.cash.turbine:turbine:1.2.0") Bez Turbine możesz użyć: flow.first() (jednorazowe pobranie) lub flow.take(1).toList().

# 8. Zadania do wykonania

Praktyczna część projektu koncentruje się na zastąpieniu stanów tymczasowych trwałą bazą danych Room w istniejącej aplikacji TaskApp. Zadania obejmują pełną implementację cyklu życia danych: od konfiguracji środowiska, przez budowę repozytoriów, aż po integrację paska wyszukiwania z mechanizmem opóźniania (debounce). Kontynuujesz projekt TaskApp z Ćwiczenia 2. Zastępujesz stan in-memory Room Database. Kluczowym celem jest zapewnienie trwałości informacji oraz odporności stanu aplikacji na zmiany konfiguracji urządzenia, takie jak rotacja ekranu. Aplikacja po zamknięciu i ponownym otwarciu powinna zachować wszystkie zadania.

## Zadanie 1. Konfiguracja Room

**Wymagania**

1.1. Dodaj Room + KSP do libs.versions.toml i build.gradle.kts. Pokaż prowadzącemu BUILD SUCCESSFUL po Sync Gradle.

1.2. Utwórz TaskEntity.kt z @Entity(tableName = "tasks") i polami: id (PrimaryKey), text, isDone, priority (Int), createdAt (Long). Minimum 5 pól.

1.3. Utwórz TaskDao.kt z metodami: getAllTasks() (Flow), insertTask(), updateTask() lub updateTaskStatus(), deleteTaskById().
1.4. Utwórz AppDatabase.kt z @Database(entities = [TaskEntity::class], version = 1, exportSchema = **true**). Singleton getInstance().
1.5. Sprawdź w Android Studio: View → Tool Windows → App Inspection → Database Inspector - baza powinna być widoczna po uruchomieniu aplikacji.

## Zadanie 2. Repository i ViewModel

### Wymagania

2.1 Utwórz interfejs TaskRepository z metodami odpowiadającymi operacjom DAO + mapowaniem na domain model Task.
2.2 Zaimplementuj TaskRepositoryImpl(taskDao: TaskDao) : TaskRepository.
Mapuj TaskEntity ↔ Task przez funkcje rozszerzające toDomainModel() i toEntity().
2.3 Zaktualizuj TaskViewModel - wstrzyknij TaskRepository przez konstruktor.
Zastąp MutableStateFlow(lista) przez repository.getAllTasks().stateIn(...).
2.4 Upewnij się że addTask, toggleTask, deleteTask wywołują viewModelScope.launch { repository.xxx() }.
2.5 Zaktualizuj MainActivity - utwórz AppDatabase, TaskRepositoryImpl, ViewModelFactory i przekaż do AppNavigation.

## Zadanie 3. Integracja z UI i trwałość danych

### Wymagania

3.1 Uruchom aplikację, dodaj kilka zadań. Zamknij aplikację (swipe up w recent apps, nie tylko wciśnij Home). Otwórz ponownie - zadania muszą być zachowane.
3.2 Sprawdź Database Inspector: View → Tool Windows → App Inspection → Database Inspector. Kliknij tabelę tasks - pokaż prowadzącemu dane wstawione przez aplikację.
3.3 Wgraj dane testowe przy pierwszym uruchomieniu: jeśli baza jest pusta, wstaw 3 przykładowe zadania. Użyj RoomDatabase.Callback (onCreate) lub sprawdź count w ViewModel init {}.
3.4 Dodaj pole SearchBar (OutlinedTextField lub SearchBar z M3) do HomeScreen. Wpisywanie tekstu filtruje zadania w czasie rzecz. (debounce 300ms przez _searchQuery w ViewModel).
3.5 Obróć emulator (Ctrl+F11 lub Device Manager → Rotate). Zweryfikuj że lista zadań się nie zresetowała, a ViewModel przeżył rotację dzięki Room.

## Zadanie 4. Migracja i testy

### Wymagania

4.1 Dodaj kolumnę category (TEXT NOT NULL DEFAULT 'general') do TaskEntity.
Zaktualizuj version = 2 w @Database. Napisz MIGRATION_1_2 z ALTER TABLE.
4.2 Zarejestruj migrację w AppDatabase.buildDatabase() przez .addMigrations(MIGRATION_1_2).
Uruchom aplikację - dane z poprzedniej sesji muszą być zachowane
(nie wolno używać fallbackToDestructiveMigration!).
4.3 Napisz minimum 3 testy jednostkowe DAO w TaskDaoTest.kt używając Room.inMemoryDatabaseBuilder. Testy: insertTask, deleteTask, deleteCompletedTasks.
4.4 Uruchom testy: kliknij prawym na klasę TaskDaoTest → Run 'TaskDaoTest'.
Pokaż prowadzącemu zielony pasek w test runner.

# 10. Najczęstsze błędy i rozwiązania

| | |
|---|---|
| **Brak getterów / błędna definicja encji** | Upewnij się, że **TaskEntity** jest data class z właściwościami w konstruktorze (np. val, nie pola w stylu Java). Room wymaga publicznych **getterów,** w Kotlin są generowane automatycznie dla właściwości. |

| | |
|---|---|
| **Room nie może zweryfikować schematu** | Problem wynika z różnicy między aktualnym kodem a zapisanym exportSchema. Rozwiązania: usuń katalog schemas/ i przebuduj projekt, lub tymczasowo użyj: fallbackToDestructiveMigration() *(tylko DEV, bo usuwa wszystkie dane!)* |
| **Błędne użycie DAO** (wywołanie na wątku głównym) | Próba uruchomienia DAO bez coroutine lub bez Flow. Sprawdź:<br>◦ czy metoda DAO ma suspend lub zwraca Flow,<br>◦ używaj viewModelScope.launch { ... } dla suspend,<br>◦ lub obserwuj Flow w UI. |
| **Niezgodna wersja KSP** | W **libs.versions.toml** wersja KSP musi być zgodna z wersją Kotlin. W razie problemów: **File → Invalidate Caches → Invalidate and Restart** → potem **Gradle Sync**. |
| **Migracja wykonana, ale brak kolumny** | Najczęściej błąd w SQL w **Migration.migrate().** Sprawdź zapytanie (np. ALTER TABLE). Dodatkowo:<br>◦ testuj na czystej instalacji (usuń dane aplikacji/ przeinstaluj APK),<br>◦ emulator może mieć starą wersję bazy. |
| **Zły typ zwracany w DAO** (brak reaktywności) | Użycie **List<T>** zamiast **Flow<List<T>>**:<br>◦ **List<T>** zwraca jednorazowy wynik (brak aktualizacji),<br>◦ **Flow<List<T>>** automatycznie emituje zmiany.<br>Poprawnie:<br>◦ fun getTasks(): Flow<List<TaskEntity>> (bez suspend) |

# 11. Narzędzie diagnostyczne Database Inspector

Diagnostyka i debugowanie bazy danych w czasie rzeczywistym realizowane są za pomocą zintegrowanego z Android Studio narzędzia, które umożliwia podgląd tabel, edycję rekordów „na żywo" oraz wykonywanie dowolnych zapytań SQL bezpośrednio na działającym procesie aplikacji. Database Inspector to wbudowane narzędzie pozwalające przeglądać, edytować i wykonywać zapytania SQL bez zatrzymywania procesu.

| # | Akcja | Korzystanie z Database Inspector |
|---|---|---|
| 1 | Otwórz inspektor | View → Tool Windows → **App Inspection** → zakładka **Database Inspector**.<br>Aplikacja musi być uruchomiona na emulatorze lub urządzeniu. |
| 2 | Wybierz bazę | Na liście po lewej wybierz task_database. Pojawi się lista tabel: tasks. |
| 3 | Przeglądaj dane | Kliknij tabelę tasks → widok wierszy. Możesz sortować po kliknięciu nagłówka kolumny. Checkbox 'Live updates' automatycznie odświeża widok gdy aplikacja modyfikuje bazę. |
| 4 | SQL Query | Kliknij ikonę New Query Session (lub Open new query tab). Wpisz zapytanie: SELECT * FROM tasks WHERE is_done = 0 ORDER BY priority DESC → Execute. |
| 5 | Edycja na żywo | Kliknij dwukrotnie komórkę - możesz ją edytować bezpośrednio. Naciśnij Enter, a zmiana jest natychmiast zapisana do bazy. Aplikacja odczyta nową wartość przez Flow. |
| 6 | Export bazy | Prawy klik na bazę → Export Database. Pobiera plik .db - możesz go otworzyć w DB Browser for SQLite (dbsqlitebrowser.org) na komputerze. |

# 12. Sprawozdanie i repozytorium

Sprawozdanie nie jest wymagane. Link do repo na Moodle. Nazwa: pam-lab3-<inicjały>. Końcowy etap prac nad projektem obejmuje weryfikację kompletności rozwiązania zgodnie z listą kontrolną, która obejmuje poprawność kompilacji, trwałość danych po zamknięciu aplikacji oraz obecność plików schematu bazy danych w repozytorium. Udokumentowanie wyników testów jednostkowych

oraz przygotowanie instrukcji uruchomienia w pliku README stanowi standard profesjonalnego dostarczenia oprogramowania.

| # | Akcja | Checklist przed wysłaniem |
|---|---|---|
| 1 | Kompilacja | Build → Make Project → 0 errors. Aplikacja startuje i wyświetla listę zadań. |
| 2 | Trwałość danych | Dodaj zadanie → zamknij app (Recent Apps → swipe up) → otwórz ponownie → zadanie widoczne. |
| 3 | Testy przechodzą | Run TaskDaoTest → wszystkie testy zielone. Screenshot testu jako dowód (dołącz do README). |
| 4 | Schemat Room | Sprawdź czy plik app/schemas/pl.edu.../AppDatabase/1.json istnieje i jest commitowany do repo. |
| 5 | README.md | Opis projektu + instrukcja uruchomienia + screenshot aplikacji + screenshot Database Inspector. |
| 6 | Link na Moodle | Wklej URL repo w pole zadania na Moodle. Dodaj prowadzącego jako collaboratora. |

## Instrukcja Kotlin 3: Room Database i Kotlin Coroutines

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

**MobileHub**

*Następne ćwiczenie: Kotlin 4 - REST API, Retrofit i Coil*

POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

# ĆWICZENIE LABORATORYJNE - Kotlin 4

## REST API, Retrofit i Coil

*Programowanie Aplikacji Mobilnych (Android / Kotlin / Jetpack Compose)*

Projekt: **Przeglądarka Pokemonów PokeApp 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. |

| # | **Etap żądania HTTP** (od aplikacji do danych na ekranie) |
|---|---|
| | 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. |
| 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. Co kody statusu HTTP mówią deweloperowi

Każda odpowiedź HTTP zawiera trzy-znakowy 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.<br>**200 OK**: zasób zwrócony.<br>**201 Created**: zasób stworzony.<br>**204 No Content**: akcja wykonana, brak treści. → Przetwarzaj dane normalnie. |
| **4xx - Błąd klienta** | Problem po stronie żądania.<br>**400 Bad Request**: niepoprawne dane (sprawdź JSON).<br>**401 Unauthorized**: brak autoryzacji.<br>**403 Forbidden**: brak uprawnień.<br>**404 Not Found**: zasób nie istnieje. → Pokaż komunikat użytkownikowi, NIE próbuj ponownie. |
| **5xx - Błąd serwera** | Problem po stronie serwera.<br>**500 Internal Server Error:** błąd kodu serwera.<br>**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!) |

| Odpowiedź GET /api/v2/pokemon/25 (fragment) | Wyjaśnienie |
|---|---|
| `"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**

- 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).
- 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.
- ZŁY TYP - ID Pokemona mieści się w Int, ale ID zasobów na wielu API wymagają Long (ponad 2 miliardy wpisów).
- 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. |

| Warstwa | Odpowiedzialność / typowe błędy |
|---|---|
| **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? Otóż gwoli zasady 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_expe-rience")` | @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 |

| PokemonDetailDto.kt (dokładna kopia struktury JSON) | Wyjaśnienie |
|---|---|
| `  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) |
| `  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 |

| PokemonMapper.kt - extension functions | Wyjaśnienie |
|---|---|
| `.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) |

| PokemonApiService.kt | Wyjaśnienie |
|---|---|
| | |
| `@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 |
| `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 |

| RetrofitClient.kt | Wyjaśnienie |
|---|---|
| `.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. |

| Typ błędu | Przyczyna i jak się objawia |
|---|---|
| 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**

- ✓ 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.
- ✓ 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.
- ✓ 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 |

| Result.kt | Wyjaśnienie |
|---|---|
| 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 |
| | |
| 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 i 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 |

| AsyncImage w Compose | Wyjaśnienie |
|---|---|
| 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. |
| .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'. |
| 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. |
| **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. |

| Ocena | Punkty | Wymagania |
|---|---|---|
| 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*

POLITECHNIKA RZESZOWSKA
KATEDRA INFORMATYKI I AUTOMATYKI
DR INŻ. MATEUSZ POMIANEK

# ĆWICZENIE LABORATORYJNE - Kotlin 5

## Hilt, Testy jednostkowe i instrumentacyjne

*Programowanie Aplikacji Mobilnych (Android / Kotlin / Jetpack Compose)*

Projekty: TaskApp (Hilt) + PokeApp (testy)

Wymagania: ukończone Lab 1–4 (Room, ViewModel, Navigation, Retrofit, Coroutines)

## Spis treści



## 1. Czym jest Dependency Injection?

Zanim poznamy Hilt, musimy dokładnie zrozumieć problem, który rozwiązuje. Dependency Injection (DI) to jeden z tych wzorców projektowych, który łatwo zrozumieć po zobaczeniu kodu bez niego, jest po prostu nieczytelny i kruchy.

### 1.1. Świat bez DI - problem twardych zależności

Wyobraź sobie, że piszesz ViewModel, który potrzebuje Repository, które potrzebuje DAO i ApiService. Bez DI każda klasa tworzy swoje zależności samodzielnie:

**Anty-wzorzec: twarde zależności**

```
// BEZ DI - jak NIE należy robić:
class TaskViewModel : ViewModel() {
 // ViewModel sam tworzy Database - hardcoded 'new'
 private val db = Room.databaseBuilder(      // Wymaga Context!
  MyApp.instance,                  // Singleton aplikacji - anty-wzorzec
  TaskDatabase::class.java, "tasks.db"
 ).build()
 private val dao      = db.taskDao()      // Hardcoded zależność
 private val apiService = RetrofitClient.api  // Singleton - nie można zamockować!
 private val repository = TaskRepository(dao, apiService)

 // Problem 1: W teście nie można podmienić prawdziwej bazy na fake
 // Problem 2: Zmiana implementacji bazy = zmiana w ViewModelu
 // Problem 3: ViewModel tworzy Room - narusza Single Responsibility
}
```

**Analogia: budowa domu i podwykonawcy**

Wyobraź sobie budowę domu. Architekt (ViewModel) potrzebuje elektryka, hydraulika i murarza.

TWARDY ZALEŻNOŚĆ (zły sposób): Architekt sam szuka elektryka, dzwoniąc do konkretnego Pana Jana.
Jeśli Pan Jan jest chory, budowa stoi. Jeśli chcemy zmienić elektryka to musimy zmienić cały projekt.

DEPENDENCY INJECTION (dobry sposób): Architekt mówi tylko 'potrzebuję elektryka'.
KIEROWNIK BUDOWY (Hilt) dostarcza odpowiedniego fachowca. Może to być Pan Jan, albo Pani Anna, albo na czas testów - aktor który udaje elektryka (mock). Architekt nie wie i nie musi wiedzieć.

Korzyści: łatwa wymiana implementacji, testowalność, single responsibility, brak cykli zależności.

## 1.2. Trzy sposoby wstrzykiwania

| Typ wstrzykiwania | Opis i kiedy używać |
|---|---|
| Constructor Injection (preferowany) | Zależności przekazywane przez konstruktor. Najbardziej czytelny, obowiązkowy dla klas nie-Android (Repository, UseCase, Mapper). Hilt automatycznie dostarcza argumenty. |
| Field Injection (@Inject lateinit var) | Dla klas tworzone przez Android (Activity, Fragment, ViewModel). Hilt wstrzykuje pola oznaczone @Inject automatycznie po @AndroidEntryPoint. |
| Method Injection (@Inject fun) | Rzadko używany. Wywoływany po constructor injection. Przydatny gdy zależność potrzebna dopiero po częściowej inicjalizacji. |

**Dlaczego właśnie Hilt, a nie ręczne DI lub pure Dagger?**

RECZNE DI: Skaluje się do około 10 klas. Przy 50+ klasach 'fabryki zależności' stają się koszmarem - każda zmiana to lawina editów.

PURE DAGGER 2: Potężny, ale wymaga ręcznego pisania Componentów, Moduleów i Subcomponentów.

Krzywa uczenia się jest bardzo stroma. Tysiące linii boilerplate kodu.

HILT = Dagger 2 + gotowe komponenty dla Android + zero boilerplate komponentów.

Hilt wie o cyklu życia Activity, Fragment, ViewModel, Service - sam tworzy odpowiednie zakresy.

Mniej kodu do napisania, pełna moc Dagger 2 'pod spodem'.

# 2. Hilt: konfiguracja projektu

Hilt wymaga konfiguracji na trzech poziomach: wtyczka Gradle (procesor adnotacji), zależności biblioteki, oraz adnotacja @HiltAndroidApp na klasie Application. Bez któregokolwiek z tych elementów Hilt nie działa.

## 2.1. Wtyczka i zależności

**gradle/libs.versions.toml** — **TOML**

```toml
# gradle/libs.versions.toml
[versions]
hilt        = "2.51.1"   # Hilt - zawsze użyj tej samej wersji dla wszystkich artefaktów!
ksp         = "2.0.21-1.0.25"  # KSP - Kotlin Symbol Processing (szybszy niż kapt)

[libraries]
hilt-android   = { module = "com.google.dagger:hilt-android",        version.ref = "hilt" }
hilt-compiler  = { module = "com.google.dagger:hilt-compiler",       version.ref = "hilt" }
hilt-nav-compose= { module = "androidx.hilt:hilt-navigation-compose",  version = "1.2.0" }
hilt-testing   = { module = "com.google.dagger:hilt-android-testing",  version.ref = "hilt" }

[plugins]
hilt        = { id = "com.google.dagger.hilt.android", version.ref = "hilt" }
ksp         = { id = "com.google.devtools.ksp",       version.ref = "ksp" }
```

**Konfiguracja build.gradle.kts**

```
// build.gradle.kts (project level)
plugins {
  alias(libs.plugins.hilt) apply false  // Deklaruj, nie stosuj na poziomie projektu
  alias(libs.plugins.ksp)  apply false
}

// app/build.gradle.kts
plugins {
  id("com.android.application")
  id("org.jetbrains.kotlin.android")
  alias(libs.plugins.hilt)  // Wtyczka Hilt - musi być przed dependencies!
  alias(libs.plugins.ksp)   // KSP generuje kod w czasie kompilacji
}
```

```
dependencies {
    implementation(libs.hilt.android)       // Runtime Hilt
    ksp(libs.hilt.compiler)                 // Procesor adnotacji - generuje kod DI
    implementation(libs.hilt.nav.compose)   // hiltViewModel() w Compose

    // Dla testów:
    testImplementation(libs.hilt.testing)   // HiltAndroidRule w JVM testach
    androidTestImplementation(libs.hilt.testing) // HiltAndroidRule w Instrumented testach
    kspTest(libs.hilt.compiler)             // Generowanie kodu DI dla testów
    kspAndroidTest(libs.hilt.compiler)      // Generowanie kodu DI dla Android testów
}
```

**Wersja hilt-compiler MUSI być identyczna z hilt-android!**

Hilt używa procesora adnotacji (KSP) do generowania kodu w czasie kompilacji.
Jeśli wersja kompilatora różni się od wersji runtime, Hilt wygeneruje niepoprawny kod.

BŁĄD: hilt-android = 2.51.1 + hilt-compiler = 2.48 → błąd kompilacji, trudny do zdebugowania.
POPRAWNIE: Obie zależności wskazują na version.ref = "hilt" w libs.versions.toml.

Ten sam problem dotyczy par: room-runtime + room-compiler, lifecycle-viewmodel + lifecycle-compiler.
Zawsze użyj jednej zmiennej wersji dla wszystkich artefaktów tej samej biblioteki.

## 2.2. Klasa Application z @HiltAndroidApp

| MyApp.kt - punkt wejścia Hilt | Wyjaśnienie |
|---|---|
| `@HiltAndroidApp` | @HiltAndroidApp uruchamia generator kodu Hilt dla całej aplikacji |
| `class MyApp : Application() {` | Dziedziczy po Application - nadpisuje klasa bazowa Androida |
| `// Hilt automatycznie tworzy` | Nie musisz nic pisac! Hilt generuje komponent App w tle. |
| `// AppComponent pod spodem.` | |
| `}` | |
| `// W AndroidManifest.xml:` | Musisz zarejestrować klase Application w Manifeście! |
| `<application` | |
| `android:name=".MyApp"` | Bez tej linii Android użyje domyślnej klasy Application |
| `...>` | Brak .MyApp = Hilt nie inicjalizuje się = crash przy @Inject |

# 3. Moduły Hilt; jak dostarczać zależności

Hilt musi wiedzieć, jak stworzyć każdą zależność. Jeśli klasa ma konstruktor oznaczony @Inject - Hilt poradzi sobie sam. Ale co jeśli musimy dostarczyć interfejs, lub obiekt tworzony przez builder (jak Retrofit lub Room)? Tu właśnie potrzebujemy Modułów.

## 3.1. Kiedy potrzebny jest @Module?

| Sytuacja | Dlaczego potrzebny moduł? |
|---|---|
| Interfejs (np. PokemonRepository) | Hilt nie wie którą implementację wybrać.<br>@Binds wskazuje konkretną klasę. |
| Zewnętrzna biblioteka (Retrofit, Room, Gson) | Nie możesz dodać @Inject do klasy zewnętrznej.<br>@Provides opis opisuje jak ją stworzyć. |
| Skomplikowana inicjalizacja (Builder pattern) | Tworzenie przez builder (Retrofit.Builder()) wymaga kodu, bo Hilt sam tego nie ogarnie. |
| Klasa tworzona fabryką (OkHttpClient.Builder) | Trzeba skonfigurować timeouty, interceptory,<br>a @Provides to miejsce na tę logikę. |

## 3.2. @Provides - jak dostarczyć zewnętrzne klasy

| NetworkModule.kt: @Provides dla Retrofit i OkHttp | Wyjaśnienie |
|---|---|
| @Module | @Module informuje Hilt, że ta klasa zawiera przepisy na tworzenie obiektów |
| @InstallIn(SingletonComponent::class) | Określa zakres: Singleton = jeden egzemplarz na całą aplikację |
| object NetworkModule { | object (nie class) - funkcje @Provides mogą być statyczne, to szybsze |
| | |
| @Provides | @Provides = ta funkcja tworzy obiekt dla Hilt |
| @Singleton | @Singleton = stwórz raz, reedytuj ten sam egzemplarz wszyscie |
| fun provideOkHttpClient(): OkHttpClient { | Hilt wywoła tę funkcję gdy ktoś poprosi o OkHttpClient |
| return OkHttpClient.Builder() | Builder pattern - nie można tego zrobić przez @Inject |
| .connectTimeout(10, TimeUnit.SECONDS) | Konfiguracja timeoutu |
| .build() | |
| } | |
| | |
| @Provides @Singleton | Dwie adnotacje na jednej linii - poprawna składnia |
| fun provideRetrofit( | Hilt widzi OkHttpClient w parametrze - wstrzykuje automatycznie! |
| client: OkHttpClient // <- wstrzyknij! | Hilt użyje provideOkHttpClient() aby dać ten parametr |
| ): PokemonApiService { | Zwracamy interfejs - Retrofit wygeneruje implementację |
| return Retrofit.Builder() | |
| .baseUrl("https://pokeapi.co/api/v2/") | |
| .client(client) | Przekazujemy wstrzyknięty klient |
| .addConverterFactory(GsonConverterFactory.create()) | |
| .build() | |
| .create(PokemonApiService::class.java) | Dynamic Proxy - implementacja interfejsu |
| } | |
| } | |

### 3.3. @Binds - jak powiązać interfejs z implementacją

| RepositoryModule.kt: @Binds dla interfejsu | Wyjaśnienie |
|---|---|
| @Module | |
| @InstallIn(SingletonComponent::class) | |
| abstract class RepositoryModule { | MUSI być abstract class (nie object) gdy używasz @Binds |
| | |
| @Binds | @Binds jest szybszy od @Provides, bo nie tworzy klasy pośredniej |
| @Singleton | Jeden egzemplarz repository na app |
| abstract fun bindPokemonRepository( | @Binds: Hilt widzi że impl = PokemonRepositoryImpl |
| impl: PokemonRepositoryImpl // <- konkretna klasa | PokemonRepositoryImpl musi mieć @Inject constructor |
| ): PokemonRepository // <- interfejs | Gdy ktoś poprosi o PokemonRepository - dostanie impl |
| } | |

**@Provides vs @Binds – którego i kiedy?**

@BINDS - gdy masz interfejs + klasę z @Inject constructor. Tylko mapowanie interfejs→implementacja. Hilt generuje prostszy, szybszy kod. Wymaga abstract class (nie object).
Przykład: interface PokemonRepository ← @Binds - PokemonRepositoryImpl

@PROVIDES - gdy musisz wykonać kod tworzenia obiektu (Builder pattern, fabryka, external lib). Funkcja może być nieabstrakcyjna, może być w object (statyczna = szybsza).
Przykład: OkHttpClient.Builder() + timeout + interceptory = potrzebujesz kodu ← użij @Provides.

ZASADA: preferuj @Binds gdy się da, użyj @Provides gdy musisz wykonać kod tworzenia.

## 4. Zakresy (Scopes): cykl życia zależności

Zakres (scope) określa, jak długo istnieje wstrzyknięty obiekt. To kluczowe pojęcie: zbędne singletonów prowadzi do wycieków pamięci, a zbyt krótki czas życia do wydajnościowych problemów.

**Analogia: okresy ważności biletów**

Wyobraź sobie różne rodzaje biletów:

@Singleton = Karta stałego klienta - ważna przez cały czas korzystania z usługi (life of app).
Jeden egzemplarz. Wszystkie miejsca dostają tę samą kartę. Przykłady: Database, Retrofit.

@ActivityRetainedScoped = Bilet tygodniowy - przetrwa obrót ekranu (ViewModel jest ActivityRetained).
ViewModel Hilt automatycznie używa tego zakresu.

@ViewModelScoped = Bilet na jedną jazdę w obie strony - żyje dopiero do zniszczenia ViewModela.
Używaj dla UseCase i helperów, które powinny być unikalne dla każdego ViewModela.

@ActivityScoped = Bilet dzienny - ważny tylko w obrębie jednej Activity (nie przetrwa rotacji!).

## 4.1. Tabela zakresów Hilt

| Adnotacja zakresu | Komponent / Czas życia / Zastosowanie |
|---|---|
| @Singleton | SingletonComponent. Żyje przez całą aplikację.<br>Użyj dla: Database, Retrofit, OkHttpClient, Repository. |
| @ActivityRetainedSco-ped | ActivityRetainedComponent. Przetrwa rotację ekranu (tak jak ViewModel).<br>Użyj dla: ViewModel-level helperów. |
| @ViewModelScoped | ViewModelComponent. Żyje przez czas życia konkretnego ViewModela.<br>Użyj dla: UseCase związanych z jednym VM. |
| @ActivityScoped | ActivityComponent. Żyje wraz z Activity - umiera przy rotacji.<br>Użyj dla: helperów UI specyficznych dla Activity. |
| @FragmentScoped | FragmentComponent. Żyje wraz z Fragmentem. Rzadko potrzebny w Compose (brak Fragmentów). |
| (brak adnotacji) | Każde wstrzyknięcie tworzy NOWY egzemplarz.<br>Użyj dla: lekkich, bezstanowych helperów. |

**@Singleton w ViewModelu to wyciek pamięci!**

Jeśli wstrzykniesz @Singleton obiekt do Activity (nie przez ViewModel), ten singleton będzie trzymać referencję do Activity - nawet po jej zniszczeniu.

Przykład: @Singleton class UserPreferences(@Inject val context: Context) - Jeśli context to ActivityContext (nie ApplicationContext), masz wyciek pamięci.

ZASADA: Obiekty @Singleton mogą trzymać jedynie ApplicationContext.
Użyj @ApplicationContext zamiast Context tam gdzie to konieczne.

## 4.2. Hilt i ViewModel: @HiltViewModel

| TaskViewModel.kt z Hilt | Wyjaśnienie |
|---|---|
| @HiltViewModel | @HiltViewModel rejestruje VM w systemie Hilt - Hilt zarządza tworzeniem |
| class TaskViewModel @Inject constructor( | @Inject constructor = tutaj Hilt wstrzykuje zależności |
| private val repository: TaskRepository, | Hilt dostarczy implementację (z @Binds lub @Provides) |
| private val savedStateHandle: SavedStateHandle | SavedStateHandle - Hilt wie jak to dostarczyć automaty-cznie! |
| ): ViewModel() { | |
| val tasks = repository.getAllTasks() | Korzystamy z wstrzykniętego repozytorium |
| } | |
| | |
| // W Compose UI - dostęp do HiltViewModel: | |

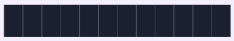
| TaskViewModel.kt z Hilt | Wyjaśnienie |
|---|---|
| @Composable fun TaskScreen( | |
| viewModel: TaskViewModel = hiltViewModel() | hiltViewModel() - z libs.hilt.nav.compose |
| ){ | hiltViewModel() = Hilt tworzy VM lub zwraca istniejący z zakresu |
| val tasks by viewModel.tasks.collectAsStateWithLifecycle() | |
| } | |

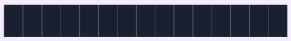
# 5. Filozofia testowania i piramida testów

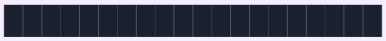
Testy to nie opcja - to gwarancja, że refaktoryzacja nie psuje istniejących funkcjonalności, a nowy kod robi to co powinien. Dobry projekt testowy opiera się na 'piramidzie testów'.

**Piramida testów; metafora budownictwa**

Wyobraź sobie budynek: solidne fundamenty (wiele małych testów) utrzymują ściany (mniej testów integracyjnych), a dach (nieliczne testy E2E) zamyka całość. Odwrotna piramida (dużo E2E, mało unit) = krucha budowla.

 Dach: Testy E2E / UI (5%): wolne, kruche, kosztowne utrzymania

 Ściany: Testy integracyjne (25%): Room + API + ViewModel razem

 Fundamenty: Testy jednostkowe (70%): szybkie, izolowane, tanie

Google rekomenduje: 70% unit - 20% integration - 10% E2E.

## 5.1. Podział testów w projekcie Android

| Typ testu | Lokalizacja / charakterystyka / narzędzia |
|---|---|
| Unit tests (testy jednostkowe) | src/test/ - JVM only, bez Androida. Milisekundy na test.<br>Testuj: mapperów, ViewModel logiki, use caseów, util klas.<br>Narzędzia: JUnit 5, MockK, Turbine, kotlinx-coroutines-test. |
| Integration tests (testy integracyjne) | src/test/ lub src/androidTest/ - Room InMemory + Fake Repository.<br>Testuj: DAO queries, Repository flow, VM + Repository razem.<br>Narzędzia: Room + JUnit 5. |
| Instrumented tests (UI testy) | src/androidTest/ - wymagają emulatora/urządzenia. Sekundy do minut.<br>Testuj: Compose UI interakcje, nawigację, dostępność.<br>Narzędzia: Compose Testing, Espresso, Hilt Testing. |

## 5.2. Zależności testowe; co dodać do build.gradle.kts

**Zależności testowe**

```
// app/build.gradle.kts - sekcja dependencies
dependencies {
 // JUnit 5 - modern framework do testów jednostkowych
 testImplementation("org.junit.jupiter:junit-jupiter:5.10.2")
```

```
    testImplementation("org.junit.jupiter:junit-jupiter-params:5.10.2") // @ParameterizedTest

    // MockK - mocking framework dla Kotlin (zamiast Mockito)
    testImplementation("io.mockk:mockk:1.13.12")

    // Turbine - testowanie Flow (od Cash App)
    testImplementation("app.cash.turbine:turbine:1.1.0")

    // Coroutines test support
    testImplementation("org.jetbrains.kotlinx:kotlinx-coroutines-test:1.8.1")

    // Compose Testing (Instrumented)
    androidTestImplementation("androidx.compose.ui:ui-test-junit4")
    debugImplementation("androidx.compose.ui:ui-test-manifest") // Wymagany manifest dla testów

    // Hilt Testing
    androidTestImplementation(libs.hilt.testing)
    kspAndroidTest(libs.hilt.compiler)
}

// Który JUnit runner? Dla JUnit 5 potrzebny dodatkowy plugin:
tasks.withType<Test> { useJUnitPlatform() } // W bloku android {} lub tasks
```

# 6. Testy jednostkowe - JUnit 5 + MockK

Test jednostkowy (unit test) testuje jedną klasę w izolacji. Wszystkie zależności zastępujemy obiektami pozorowanymi (mock). Dlatego Dependency Injection jest kluczowy dla testowalności!

**Dlaczego MockK zamiast Mockito dla Kotlin?**

Mockito powstał dla Javy i ma kilka problemów z Kotlinem:

1. Klasy final - w Kotlinie wszystkie klasy są domyślnie final. Mockito nie może ich mockować bez hacków.
2. Kotlin coroutines i suspend functions - Mockito nie wspiera natywnie, wymaga obejść.
3. Kotlin-specific API (companion object, extension functions, top-level functions) - Mockito nie obsługuje.

MockK został stworzony dla Kotlin od podstaw:

+ Mockuje klasy final natywnie (przez inline mocking).
+ Natywne wsparcie dla suspend functions (coEvery, coVerify).
+ Mocki companion objectów, object singletonów, extension functions.

## 6.1. Anatomia testu jednostkowego

| PokemonMapperTest.kt - test mappera | Wyjaśnienie |
|---|---|
| class PokemonMapperTest { | Zwykła klasa Kotlin - bez @RunWith, bez Android |
| | |
| @Test | @Test - adnotacja JUnit 5 (import org.junit.jupiter.api.Test) |

| PokemonMapperTest.kt - test mappera | Wyjaśnienie |
|---|---|
| fun `height in decimeters is converted to meters`() { | Nazwa testu jako backtick string - czytelna dokumentacja |
| // GIVEN - przygotuj dane | Sekcja Given: co mamy na wejściu (Arrange) |
| val dto = fakePokemonDetailDto(height = 7) | Tworzymy DTO z wysokością 7 dm (= 0.7m) |
| | |
| // WHEN - wykonaj testowane działanie | Sekcja When: co testujemy (Act) |
| val result = dto.toDomainModel() | Wywołujemy mapper - to jest SUT (System Under Test) |
| | |
| // THEN - zweryfikuj wynik | Sekcja Then: co oczekujemy (Assert) |
| assertThat(result.heightMeters).isEqualTo(0.7) | Google Truth: czytelniejsze niż assertEquals |
| } | |
| | |
| @Test | |
| fun `hidden abilities are filtered out`() { | |
| val dto = fakePokemonDetailDto( | |
| abilities = listOf( | |
| fakeAbility("static", isHidden = false), | Widoczna zdolność - powinna pojawić się w wyniku |
| fakeAbility("lightning-rod", isHidden = true) | Ukryta zdolność - powinna być odfiltrowana |
| ) | |
| ) | |
| val result = dto.toDomainModel() | |
| assertThat(result.abilities).containsExactly("Static") | Tylko 'static' (z wielką literą po capitalize) |
| } | |
| } | |

## 6.2. Testowanie ViewModelu z MockK i Coroutines

| PokemonListViewModelTest.kt | Wyjaśnienie |
|---|---|
| @ExtendWith( | @ExtendWith - JUnit 5, nie @RunWith (JUnit 4) |
| CoroutinesTestExtension::class, // <- niestandardowe | Instaluje TestCoroutineDispatcher jako Main dispatcher |
| MockKExtension::class | MockK extension automatycznie inicjalizuje @MockK pola |
| ) | |
| class PokemonListViewModelTest { | |
| @MockK | @MockK tworzy mock repozytorium - nie prawdziwa implementacja! |
| private lateinit var repository: PokemonRepository | lateinit bo MockK inicjalizuje przez refleksję |
| private lateinit var viewModel: PokemonListViewModel | SUT - System Under Test |
| | |
| @BeforeEach | @BeforeEach - wywoływane PRZED każdym @Test (JUnit 5, nie @Before!) |
| fun setUp() { | |

| PokemonListViewModelTest.kt | Wyjaśnienie |
|---|---|
| `viewModel = PokemonListViewModel(repo-sitory)` | Tworzymy VM przez constructor injection - dlatego DI jest ważne! |
| `}` | |
| | |
| `@Test` | |
| `fun `loading state is emitted first`() = runTest {` | runTest - z kotlinx-coroutines-test. Kontroluje czas coroutine. |
| `// GIVEN` | |
| `coEvery { repository.getPokemonList() } returns` | coEvery = mockK dla suspend fun. 'co' = coroutine |
| `flowOf(listOf(fakePokemon()))` | Możemy zwrócić Flow z daną wartością |
| | |
| `// WHEN + THEN z Turbine` | Turbine: .test() na Flow zbiera emisje |
| `viewModel.uiState.test {` | |
| `assertThat(awaitItem()).isInstan-ceOf(` | awaitItem() = poczekaj na następną emisję |
| `UiState.Loading::class.java)` | Pierwszy emitowany stan: Loading |
| `assertThat(awaitItem()).isInstan-ceOf(` | |
| `UiState.Success::class.java)` | Drugi stan po załadowaniu: Success |
| `cancelAndIgnoreRemainingEvents()` | Anuluj test Flow - oczyszczenie |
| `}` | |
| `}` | |
| `}` | |

## 6.3. Wzorzec Given-When-Then i testy parametryczne

Dobry test składa się z trzech sekcji: Given (co mamy), When (co robimy), Then (co oczekujemy). Ten wzór - znany też jako Arrange-Act-Assert - sprawia, że testy są samodokumentujące.

| ParameterizedTest dla mappera | Wyjaśnienie |
|---|---|
| `@ParameterizedTest` | @ParameterizedTest = jeden test, wiele danych wejściowych |
| `@CsvSource(` | @CsvSource: dane w formacie CSV wprost w adnotacji |
| `"4, 0.4", // 4 dm -> 0.4 m` | Każda linia to jeden przypadek testowy |
| `"10, 1.0", // 10 dm -> 1.0 m` | |
| `"0, 0.0", // 0 dm -> 0.0 m (krawędź)` | |
| `"999, 99.9" // duża wartość` | |
| `)` | |
| `fun `height conversion from decimeters to meters`(` | Parametry testów jako argumenty funkcji |
| `heightDm: Int, expectedM: Double` | JUnit 5 automatycznie wstrzykuje z @CsvSource |
| `){` | |
| `val dto = fakePokemonDetailDto(height = heightDm)` | |
| `val result = dto.toDomainModel()` | |

| ParameterizedTest dla mappera | Wyjaśnienie |
|---|---|
| assertThat(result.heightMeters).isEqualTo(expectedM) | |
| } | |

# 7. Testy integracyjne - Room InMemory

Testy integracyjne sprawdzają, czy wiele komponentów działa ze sobą poprawnie. W kontekcie Room testujemy DAO queries z prawdziwą bazą danych - ale bazującą na pamięci RAM zamiast dysku.

**Dlaczego Room InMemory do testów?**

Testy na prawdziwej bazie SQLite miałyby kilka problemów:

1. STAN między testami - dane z poprzedniego testu widoczne w następnym. Testy muszą być izolowane.
2. SZYBKOŚĆ - operacje dyskowe są powolne. Testy powinny być szybkie.
3. Sprzątanie - po każdym teście trzeba ręcznie usuwać plik bazy.

Room InMemory Database:

+ Każde @BeforeEach tworzy NOWĄ bazę w pamięci. Zerowy stan początkowy.
+ @AfterEach zamknie bazę i pamięć zwalnia automatycznie.
+ Identyczne zachowanie jak SQLite, ale błyskawicznie szybka.
+ Nie wymaga emulatora - można uruchomić w src/test/ (JVM) z robolectric.

## 7.1. Test DAO z Room InMemory

| PokemonDaoTest.kt | Wyjaśnienie |
|---|---|
| `@RunWith(AndroidJUnit4::class)` | AndroidJUnit4 - uruchamia test z prawdziwym Android runtime (na emulatorze) |
| `class PokemonDaoTest {` | Lub użyj @RunWith(RobolectricTestRunner) dla JVM |
| `  private lateinit var database: PokemonDatabase` | Baza testowa - tworzona na nowo przed każdym testem |
| `  private lateinit var dao: PokemonDao` | |
| | |
| `  @Before` | @Before (JUnit 4) lub @BeforeEach (JUnit 5) |
| `  fun setUp() {` | |
| `    database = Room.inMemoryDatabaseBuilder(` | KLUCZOWE: inMemoryDatabaseBuilder zamiast databaseBuilder |
| `      ApplicationProvider.getApplicationContext(),` | Kontekst aplikacji z testów Android |
| `      PokemonDatabase::class.java` | Ta sama klasa Database co w produkcji! |
| `    ).allowMainThreadQueries()  // Tylko w testach!` | Normalne Room blokuje Main thread. W testach to OK. |
| `    .build()` | |
| `    dao = database.pokemonDao()` | DAO dostępny po stworzeniu bazy |

| PokemonDaoTest.kt | Wyjaśnienie |
|---|---|
| `}` | |
| | |
| `@After` | @After = sprzątanie po każdym teście |
| `fun tearDown() { database.close() }` | Zamknij bazę - zwolnij pamięć |
| | |
| `@Test` | |
| `fun `upsert and retrieve pokemon`() = runTest {` | runTest dla suspend DAO queries |
| `// GIVEN` | |
| `val entity = fakePokemonEntity(id = 25, name = "pikachu")` | |
| `// WHEN` | |
| `dao.upsert(entity)` | Zapisz do bazy (w pamięci) |
| `val result = dao.getPokemonById(25).first()` | .first() pobiera pierwszy element z Flow i kończy |
| `// THEN` | |
| `assertThat(result?.name).isEqualTo("pikachu")` | |
| `}` | |
| `}` | |

## 7.2. Fake vs Mock - którego używać?

| Podejście | Kiedy używać i dlaczego |
|---|---|
| **Mock** (MockK / Mockito) | Gdy chcesz ZWERYFIKOWAć interakcje (czy było wywołane? z jakimi argumentami?). Szybki w tworzeniu. Ryzyko: testy sprawdzają implementację, nie zachowanie - będą się psuć przy refaktoryzacji. |
| **Fake** (ręczna implementacja) | Gdy chcesz przetestować ZACHOWANIE z lekką implementacją. FakeRepository trzyma dane w pamięci. Testy są mniej kruche - nie zależą od kolejności wywołań. Przykład: FakePokemonRepository z MutableList. |
| **Room InMemory** | Gdy testujesz DAO i SQL queries. Prawdziwa logika bazodanowa, żerowa konfiguracja. Jedyne sensowne podejście dla testów bazy danych. |
| **Spy** (MockK spyk) | Gdy chcesz częściowo mockować prawdziwy obiekt. Rzadko potrzebny, często sygnalizuje problem z architekturą. |

# 8. Testy instrumentowane: Compose UI Testing

Testy instrumentowane uruchamiają się na prawdziwym urządzeniu lub emulatorze i mogą testować zachowanie UI. Jetpack Compose dostarcza dedykowaną bibliotekę do testowania komponentów i ekranów.

## 8.1. Semantyka: jak Compose Testing widzi UI

Compose Testing nie operuje na View hierarchii (jak Espresso), lecz na drzewie semantycznym. Każdy Composable może ujawniać właściwości semantyczne: etykietę, rolę, stan (zaznaczony/niezaznaczony), wartość.

**Drzewo semantyczne; czego nie widzi test**

Compose Testing działa na DRZEWIE SEMANTYCZNYM, nie wizualnym layoutcie.
Drzewo semantyczne to 'opis dla narzędzi dostępności' - TalkBack i testy widzą to samo.

Konsekwencje dla testów:
+ Szukasz elementów przez contentDescription, text, role - nie przez ID widoków (jak w XML/Espresso).
+ onNodeWithText("Pikachu") znajdzie dowolny Text composable z tym tekstem.
+ onNodeWithContentDescription("sprite") znajdzie AsyncImage z tym contentDescription.

WAZNE: Jeśli composable nie ma ustawionej semantyki (Modifier.semantics), test go nie znajdzie!
Dlatego contentDescription jest tak ważny - nie tylko dla TalkBack, ale i dla testów.

## 8.2. Anatomia testu Compose

| PokemonListScreenTest.kt | Wyjaśnienie |
|---|---|
| @HiltAndroidTest | @HiltAndroidTest = Hilt obsługuje DI w tym teście instrumentowanym |
| class PokemonListScreenTest { | |
| | |
| @get:Rule(order = 0) | Order 0: HiltRule MUSI być pierwsza (inicjalizuje Hilt przed regułą Compose) |
| val hiltRule = HiltAndroidRule(this) | HiltAndroidRule inicjalizuje Hilt dla każdego testu |
| | |
| @get:Rule(order = 1) | Order 1 - po inicjalizacji Hilt |
| val composeTestRule = createAndroidComposeRule<MainActivity>() | createAndroidComposeRule startuje Activity z Hilt |
| | |
| @Before | |
| fun setUp() { hiltRule.inject() } | hiltRule.inject() - wstrzykuje @Inject pola w klasie testowej |
| | |
| @Test | |
| fun `pokemon list shows items after loading`() { | |
| composeTestRule.apply { | |
| // Poczekaj aż Loading zniknie (maks 5 sekund) | |
| waitUntilDoesNotExist( | waitUntil - poczekaj na warunek asynchronicznie |
| hasTestTag("loading_indicator"), | TestTag - pewniejsze niż szukanie po tekście |
| timeoutMillis = 5_000 | |
| ) | |
| // Sprawdź, czy lista zawiera min. jeden element | |

| PokemonListScreenTest.kt | Wyjaśnienie |
|---|---|
| onNodeWithTag("pokemon_list") | Znajdź element po testTag |
| .assertIsDisplayed() | Zweryfikuj że jest widoczny |
| onAllNodesWithTag("pokemon_card") | onAllNodes = wiele elementów |
| .assertCountAtLeast(1) | Co najmniej 1 karta Pokemon |
| } | |
| } | |
| } | |

## 8.3. Selektory i akcje - API Compose Testing

| Selector / Akcja | Opis i przykład |
|---|---|
| onNodeWithText("text") | Znajdź node z danym tekstem. Domyślnie: substring match. Dodaj substring=false dla dokładnego. |
| onNodeWithContentDescription("opis") | Znajdź po contentDescription. Niezbędne dla obrazów i ikon. |
| onNodeWithTag("tag") | Znajdź po Modifier.testTag(). Najbardziej stabilny selektor - nie zmienia się przy tłumaczeniach. |
| onAllNodesWithTag("tag") | Zwraca SemanticsNodeInteractionCollection. Użyj [index] lub assertCountAtLeast(). |
| .performClick() | Akcja: kliknięcie na element. |
| .performTextInput("text") | Akcja: wpisanie tekstu w TextField. |
| .performScrollTo() | Akcja: przewinięcie do elementu (w LazyColumn). |
| .assertIsDisplayed() | Asercja: element widoczny na ekranie. |
| .assertTextEquals("text") | Asercja: tekst elementu jest dokładnie 'text'. |
| .assertIsEnabled() / Disabled() | Asercja: czy przycisk jest aktywny/nieaktywny. |

## 8.4 TestTag - klucz do stabilnych testów

| Dodawanie testTagów w Compose | Wyjaśnienie |
|---|---|
| // W komponencie produkcyjnym: | |
| LazyVerticalGrid( | |
| modifier = Modifier.testTag("pokemon_list") | testTag identyfikuje element dla testów |
| ) { | |
| items(pokemons) { pokemon -> | |
| PokemonCard( | |
| pokemon = pokemon, | |
| modifier = Modifier.testTag("pokemon_card") | Każda karta ma ten sam tag - użyj onAllNodes |
| ) | |
| } | |
| } | |

| Dodawanie testTagów w Compose | Wyjaśnienie |
|---|---|
| | |
| CircularProgressIndicator( | |
| modifier = Modifier.testTag("loading_indicator") | Tag na spinnerze - czekamy aż zniknie |
| ) | |

**testTag nie powinien trafić do release APK w produkcji**

Modifier.testTag() nie zwalnia pamięci, ale dodaje semantykę. Dostępną potencjalnie póxniej dla reverse engineeringu. W produkcie korporacyjnym rozważ użycie semanticsRole lub warunkowego testTag:

modifier = if (BuildConfig.DEBUG) Modifier.testTag("pokemon_list") else Modifier

W projektach akademickich i większości komercyjnych jest to akceptowalne w kodzie produkcyjnym.

Google własne Sample Apps (Jetpack Compose Samples) używa testTag bez warunkowania.

# 9. Hilt w testach: podmienianie zależności

Kluczową zaletą Hilt jest możliwość podmieniania prawdziwych implementacji na testowe (fake/stub) bez zmiany kodu produkcyjnego. Hilt oferuje dwa mechanizmy: @TestInstallIn i @Uninstall-Modules.

## 9.1. Podmienianie modułów w testach

| FakeRepositoryModule.kt - moduł testowy | Wyjaśnienie |
|---|---|
| @TestInstallIn( | @TestInstallIn zamienia moduł produkcyjny na testowy |
| components = [SingletonComponent::class], | W którym komponencie? Singleton - tak jak oryginał |
| replaces = [RepositoryModule::class] | Który moduł produkcyjny zastępujemy? |
| ) | |
| @Module | |
| abstract class FakeRepositoryModule { | |
| | |
| @Binds @Singleton | |
| abstract fun bindRepository( | Zamiast PokemonRepositoryImpl, Hilt wstrzyknie FakeRepository |
| fake: FakePokemonRepository | FakePokemonRepository musi mieć @Inject constructor |
| ): PokemonRepository | Interfejs jest ten sam - ViewModel nie wie o zamianie! |
| } | |
| | |
| // FakePokemonRepository.kt: | |
| class FakePokemonRepository @Inject constructor( | |
| // Brak sieci! Lista w pamięci jako 'baza' | |
| ) : PokemonRepository { | Implementuje ten sam interfejs |
| private val pokemons = MutableList<Pokemon>( | |
| mutableListOf(fakePokemon(25, "pikachu")) | Predefiniowane dane testowe |

| FakeRepositoryModule.kt - moduł testowy | Wyjaśnienie |
|---|---|
| ) | |
| override fun getPokemonList() = | |
| flowOf(pokemons.toList()) | Zwraca Flow z danymi testowymi |
| } | |

**Jak Hilt scala moduły w testach?**

W normalnej aplikacji: Hilt ładuje NetworkModule + RepositoryModule - prawdziwy Retrofit + Room.

W teście z @TestInstallIn:

1. Hilt widzi że FakeRepositoryModule ma replaces = [RepositoryModule::class].
2. RepositoryModule jest IGNOROWANY - jego @Provides/@Binds nie są rejestrowane.
3. FakeRepositoryModule jest załadowany zamiast niego.
4. Reszta modułów (NetworkModule) nadal działa - tylko podmieniony moduł jest zastąpiony.

@UninstallModules (alternatywa, per-test): Adnotacja na klasie testowej. Pozwala podmienić moduł tylko dla jednego testu. @TestInstallIn = globalnie dla wszystkich testów w tym sourceset.

# 10. Testowanie Coroutines: TestDispatcher i Turbine

Coroutines są asynchroniczne - test musi umieć kontrolować czas ich wykonania. Bez tego testy będą niestabilne (flaky) - czasem zaliczone, czasem nie, bez zmiany kodu.

## 10.1. Problem z asynchronicznością w testach

**TestDispatcher** - kontrola czasu coroutine

```
// Bez TestDispatcher - test BŁĘDY:
class BrokenViewModelTest {
 @Test
 fun `data loads correctly`() {
  viewModel.loadData()       // Uruchamia coroutine NA PRAWDZIWYM IO dispatcher
  // Coroutine jest ASYNCHRONICZNA - wynik nie jest jeszcze gotowy!
  assertThat(viewModel.uiState.value)
   .isInstanceOf(UiState.Success::class.java) // FAIL: nadal Loading
 }
}

// Rozwiązanie: StandardTestDispatcher + advanceUntilIdle()
class FixedViewModelTest {
 private val testDispatcher = StandardTestDispatcher() // Kontrolowany scheduler

 @BeforeEach fun setUp() {
  Dispatchers.setMain(testDispatcher) // Podmień Main dispatcher na testowy
 }

 @AfterEach fun tearDown() {
  Dispatchers.resetMain()          // Przywróć po teście
 }

 @Test
 fun `data loads correctly`() = runTest(testDispatcher) {
```

```
    viewModel.loadData()        // Coroutine ZAPLANOWANA, ale nie uruchomiona
    advanceUntilIdle()          // Uruchom WSZYSTKIE oczekujące coroutines
    assertThat(viewModel.uiState.value)
      .isInstanceOf(UiState.Success::class.java)  // OK: coroutine zakończona
  }
}
```

## 10.2. Turbine: testowanie Flow

Testowanie Flow bez Turbine wymaga boilerplate kodu: uruchom kolektor w tle, zbieraj do listy, anuluj, porównaj. Turbine upraszcza to do kilku czytelnych wywołań.

| Turbine - asercje na Flow | Wyjaśnienie |
|---|---|
| viewModel.uiState.test { | .test { } - Turbine blokuje i zbiera emisje Flow |
| // awaitItem() czeka na następną emisję (suspend - poczeka!) | |
| val first = awaitItem() | Pierwsza emisja: oczekujemy Loading |
| assertThat(first).isInstanceOf(UiState.Loading::class.java) | |
| | |
| val second = awaitItem() | Druga emisja: po załadowaniu danych |
| assertThat(second).isInstanceOf(UiState.Success::class.java) | |
| assertThat((second as UiState.Success).data) | Cast do Success, sprawdź dane |
| .hasSize(20) | 20 Pokemonów na pierwszej stronie |
| | |
| // Anuluj, nie czekamy na kolejne emisje (Flow może być nieskończony) | |
| cancelAndIgnoreRemainingEvents() | Bezpieczne zakończenie testu Flow |
| } | |
| | |
| // Alternatywnie dla błędów: | |
| viewModel.uiState.test { | |
| awaitItem() // Loading | |
| val errorState = awaitItem() | |
| assertThat(errorState).isInstanceOf(UiState.Error::class.java) | |
| assertThat((errorState as UiState.Error).message) | |
| .contains("Brak połączenia") | |
| cancelAndIgnoreRemainingEvents() | |
| } | |

**Test Doubles - słownik terminów**

**Dummy**: obiekt przekazany ale niewywoływany (wypełniacz parametru).

**Stub**: zwraca predefiniowane wartości, nie weryfikuje wywołań. Prostszy niż Mock.

**Fake**: lekka implementacja (np. in-memory repository). Zachowuje logikę ale bez zewnętrznych zależno-

ści.

**Mock**: weryfikuje interakcje (czy było wywołane? ile razy? z jakimi args?). Może też stubbować.

**Spy**: opakowuje prawdziwy obiekt, pozwala obserwować wywołania i częściowo mockować.

# 11. Strategia testowania, czyli co testować w projekcie

Wiedza o technikach to jedno - ważniejsza jest odpowiedź na pytanie: które części kodu WARTO testować? Nie wszystko wymaga testów. Koncentruj się tam, gdzie błędy są kosztowne lub trudne do zauważenia.

## 11.1. Priorytety testowania

| Co testować (wysoki priorytet) | Co można pominąć lub testować ręcznie |
|---|---|
| Mappery (DTO → Domain) - przeliczenia jednostek, nullable, capitalize, filtrowanie | Proste gettery/settery - brak logiki = brak testów |
| ViewModel logika - kolejność stanów (Loading→Success→Error), paginacja, retry | Activity / Fragment boilerplate - Hiltów@AndroidEntryPoint |
| DAO queries - upsert, delete, flow emissions, migracje Room | Compose composableów bez logiki - czyste UI które tylko wyświetla dane |
| safeApiCall - obsługa każdego typu wyjątku (IOException, HttpException, Gson) | Konfiguracja Hilt modułów - kompilator sprawdza poprawność |
| Użytkowe funkcje biznesowe - walidacja formularzy, obliczenia, parsowanie dat | Trzecie party biblioteki - nie testujesz Retrofit, Coil, Room |

## 11.2. Sekwencja pisania testów TDD

| # | Test Driven Development - Red, Green, Refactor |
|---|---|
| 1 | RED: Napisz test, który FAILUJE. Opisz co chcesz osiągnąć (Given-When-Then). Uruchom - test musi być czerwony. |
| 2 | GREEN: Napisz MINIMALNY kod produkcyjny, który sprawi, że test przechodzi. Nie optymalizuj. Nie przewiduj. Napisz dosłownie minimum. |
| 3 | REFACTOR: Ulepsz kod produkcyjny (i test!) bez zmiany zachowania. Testy są siatką bezpieczeństwa - jeśli po refaktoryzacji są zielone, refaktoryzacja była bezpieczna. |
| 4 | POWTARZAJ: Każda nowa funkcjonalność = nowy test = nowy cykl Red-Green-Refactor. |

# 12. Zadania do wykonania

Zadania podzielone są na dwie części: integracja Hilt do istniejących projektów (TaskApp + PokeApp) oraz pisanie testów weryfikujących logikę biznesową i interfejs użytkownika.

**Zadanie 1 (25 pkt) - Migracja TaskApp do Hilt**

1.1. Dodaj wtyczkę Hilt i KSP do build.gradle.kts. Dodaj zależności (hilt-android, hilt-compiler, hilt-navigation-compose).

1.2. Stwórz klasę TaskApp z adnotacją @HiltAndroidApp i zarejestruj ją w AndroidManifest.xml.

1.3. Napisz DatabaseModule (@Module, @InstallIn(SingletonComponent::class)) dostarczający TaskDatabase i TaskDao przez @Provides @Singleton.

1.4. Zmień TaskRepository na klasę z @Inject constructor. Dostosuj TaskViewModel do @HiltViewModel @Inject constructor.

1.5. Dodaj @AndroidEntryPoint do MainActivity. Zastąp viewModel() na hiltViewModel() w TaskListScreen.

WERYFIKACJA: Aplikacja kompiluje się i działa identycznie jak przed migracją. Brak singletonów ręcznych.

### Zadanie 2 (20 pkt) - Testy jednostkowe mappera i safeApiCall

2.1. Napisz min. 5 testów jednostkowych dla PokemonMapper (JUnit 5, bez Androida, bez MockK):

a) przeliczenie decymetrów na metry,
b) przeliczenie hektogramów na kilogramy,
c) capitalize nazwy,
d) filtrowanie ukrytych zdolności,
e) fallback dla null baseExperience.

2.2. Napisz testy @ParameterizedTest dla przeliczeń jednostek z co najmniej 4 przypadkami.

2.3. Napisz min. 3 testy dla safeApiCall() używając MockK (coEvery):

a) sukces (Result.Success),
b) HttpException → Result.Error z kodem,
c) IOException → Result.Error.

WERYFIKACJA: ./gradlew test kończy się bez błędów. Pokrycie kodu mappera ≥ 80% (sprawdzone w Android Studio).

### Zadanie 3 (30 pkt) - Testy integracyjne i ViewModelu

3.1. Napisz min. 3 testy DAO (Room InMemory, AndroidJUnit4 lub Robolectric):

a) upsert + pobranie, b) usunięcie + weryfikacja braku, c) Flow emituje aktualizację po upsert.

3.2. Stwórz FakePokemonRepository implementujący PokemonRepository (dane w MutableList, Flow z SharedFlow).

3.3. Napisz min. 4 testy PokemonListViewModel (runTest, Turbine, MockK):

a) kolejność stanów Loading→Success,
b) błąd sieci → UiState.Error z komunikatem,
c) retry po błędzie → ponowna próba (verify coVerify),
d) paginacja - loadMore zwraca więcej elementów.

WERYFIKACJA: ./gradlew test kończy się bez błędów. Wszystkie 10+ testów zielone.

### Zadanie 4 (25 pkt) - Testy instrumentowane Compose UI

4.1. Dodaj testTag do kluczowych komponentów: pokemon_list, pokemon_card, loading_indicator, error_message, retry_button.

4.2. Skonfiguruj FakeRepositoryModule (@TestInstallIn) podmieniający PokemonRepository na FakePokemonRepository.

4.3. Napisz min. 3 testy instrumentowane (createAndroidComposeRule, @HiltAndroidTest):

a) lista wyświetla karty po załadowaniu,
b) kliknięcie karty otwiera ekran szczegółów (nawigacja),
c) FakeRepository zwracający błąd → widoczny ErrorScreen z przyciskiem 'Spróbuj ponownie'.

4.4. Uruchom testy na emulatorze: ./gradlew connectedAndroidTest. Wszystkie zielone.

WERYFIKACJA: Demonstracja uruchomienia testów na emulatorze prowadzącemu. Raport HTML z wynikami.

# 13. Kryteria oceniania

## 13.1. Punktacja zadań

| Zadanie | Punkty | Co weryfikuje prowadzący |
|---|---|---|
| **Zad. 1**: Hilt w TaskApp | 25 pkt | Kompilacja bez błędów. Brak ręcznych singletonów. @HiltViewModel + hiltViewModel(). DatabaseModule poprawny. |
| **Zad. 2**: Testy mappera + safeApiCall | 20 pkt | ./gradlew test zielony. Min. 5 testów JUnit 5. @ParameterizedTest. Testy safeApiCall z MockK (coEvery). |
| **Zad. 3**: Testy integracyjne + ViewModel | 30 pkt | Room InMemory DAO test. FakeRepository. Turbine dla Flow. Testy VM z verify/coVerify. |
| **Zad. 4**: Testy Compose UI | 25 pkt | FakeRepositoryModule @TestInstallIn. Testy instrumentowane na emulatorze. Raport HTML. |
| RAZEM | 100 pkt | |

## 13.2. Skala ocen

| Ocena | Punkty | Wymagania |
|---|---|---|
| 5.0 | 90–100 pkt | Wszystkie 4 zadania. Min. 15 testów. Testy instrumentowane na emulatorze. Pokrycie ≥80%. |
| 4.5 | 80–89 pkt | Zadania 1–3 kompletne. Testy instrumentowane - min. 1 test UI przechodzi na emulatorze. |
| 4.0 | 70–79 pkt | Zadania 1–3. Hilt działa. Min. 8 testów jednostkowych i integracyjnych. |
| 3.5 | 60–69 pkt | Zadanie 1 kompletne + zadanie 2 (min. 5 testów jednostkowych). Brak testów VM lub DAO. |
| 3.0 | 50–59 pkt | Hilt skonfigurowany (@HiltAndroidApp, @HiltViewModel). Min. 3 testy jednostkowe mappera. |
| 2.0 | 0–49 pkt | Projekt nie kompiluje się z Hilt, lub brak jakichkolwiek testów. |

# 14. Najczęstsze błędy i ich rozwiązania

Hilt i testy mają swoje specyficzne pułapki. Poniższa tabela zbiera błędy, które studenci napotykają najczęściej. Sprawdź ją zanim zaczniesz szukać w Internecie.

| Komunikat błędu / objaw | Przyczyna i rozwiązanie |
|---|---|
| [Hilt] ... is not an @AndroidEntryPoint | Klasa Activity/Fragment/ViewModel nie ma adnotacji. Dodaj @AndroidEntryPoint do MainActivity i @HiltViewModel do ViewModela. |
| [Hilt] ... cannot be provided without @Inject or @Provides | Hilt nie wie jak stworzyć tej klasy. Dodaj @Inject constructor() do klasy LUB napisz @Provides w Module LUB @Binds dla interfejsu. |
| Expected single matching node but found 0 | Compose Testing nie znalazł elementu. Sprawdź: testTag pasuje? Element na ekranie? Dodaj Modifier.testTag(). Użyj printToLog() do debug. |
| CancellationException w teście ViewModela | Nie użyto runTest lub brakuje advanceUntilIdle(). Uruchom logikę VM w runTest { } i wywołaj advanceUntilIdle() przed asercją. |

| Komunikat błędu / objaw | Przyczyna i rozwiązanie |
|---|---|
| MockKException: no answer found for ... | Nie skonfigurowałeś coEvery dla wywołanej metody. Dodaj coEvery { repo-.method() } returns ... PRZED wywołaniem kodu testowanego. |
| Cannot access database on the main thread | Brakuje allowMainThreadQueries() w Room InMemory dla testów. LUB: Nie używasz runTest dla suspend queries. |
| @TestInstallIn not replacing module | FakeModule musi być w src/androidTest/ (dla testów instrumentowanych) LUB src/test/ (dla JVM testów). Zły katalog = moduł nie jest widoczny. |
| Hilt rule must be first in @get:Rule order | HiltAndroidRule musi mieć order=0, ComposeTestRule order=1. Hilt inicja-lizuje się przed innymi regułami. |
| ViewModel is not initialized (Hilt) | Brakuje @HiltAndroidApp na klasie Application LUB klasa Application nie jest zarejestrowana w Manifeście android:name=".MyApp". |

**Instrukcja Laboratoryjna Nr 5: Hilt, Testy jednostkowe i instrumentacyjne**

Programowanie Aplikacji Mobilnych | Katedra Informatyki i Automatyki

**MobileHub**

*Następne ćwiczenie: Lab 6 - WorkManager, powiadomienia Push i DataStore Preferences*