Państwowa Akademia Nauk Stosowanych w Krnie	Kierunek Informatyka

Instrukcja do ćwiczeń laboratoryjnych nr:	6	Nazwa przedmiotu: Programowanie w języku Kotlin
Temat: Prezentacja danych w formie listy LazyColumn oraz komunikacja z API - Retrofit		Tryb studiów: stacjonarne Czas trwanie ćw. **2x45 min**

Autor materiałów: dr Marcin Skuba

1. Treści programowe:

Programowanie deklaratywne, tworzenie interfejsu UI – Jetpack Compose, Retrofit, API, http, Kotlin

2. Cel zajęć:

Celem zajęć jest zrozumienie zasad komunikacji aplikacji z innymi serwerami poprzez protokół http. Poznanie biblioteki Retrofit oraz jej możliwości komunikacji z zewnętrznym API.

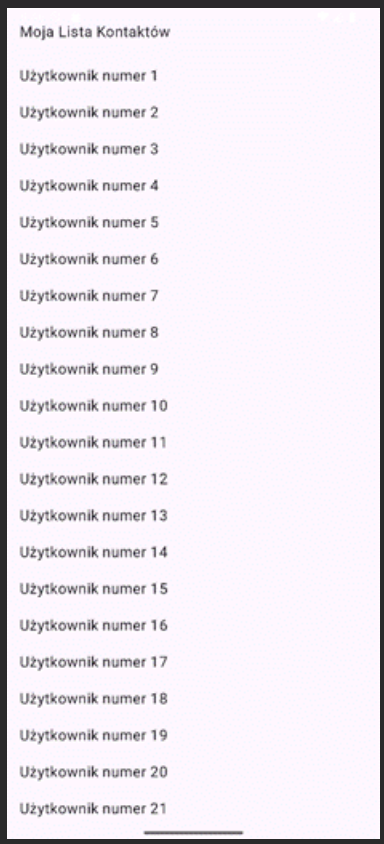
3. Materiały dydaktyczne

I. LazyColumn

Przykład 1. Lista dynamiczna LazyColumn - *prosta lista*

LazyColumn w Jetpack Compose to odpowiednik tradycyjnego RecyclerView ze starszego widoku Androida. Służy do wyświetlania **pionowych list**, które mogą mieć tysiące elementów.

Słowo **Lazy** (leniwa) oznacza, że komponent nie renderuje całej listy na raz. Tworzy na ekranie tylko te elementy, które są aktualnie widoczne dla użytkownika (plus mały zapas). Gdy przewijasz listę, niewidoczne elementy są niszczone, a nowe są tworzone w locie, co zapewnia niesamowitą płynność i oszczędność pamięci.




```
@Composable
fun ProstaListaUserow() {
    // 1. Przygotowanie przykładowych danych (lista 100 stringów)
    val listaImion = List(100) { index -> "Użytkownik numer ${index + 1}" }

    // 2. Główny kontener listy
    LazyColumn(
```

```
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp),
    ) {
        // 3. Nagłówek listy (pojedynczy element)
        item {
            Text(
                text = "Moja Lista Kontaktów",
                modifier = Modifier.padding(bottom = 16.dp)
            )
        }

        // 4. Dynamiczne generowanie elementów na podstawie danych
        items(listaImion) { imie ->
                 Text(
                    text = imie,
                    modifier = Modifier.padding(vertical = 8.dp)
                 )

        }
    }
}
```

Aby **LazyColumn** działało poprawnie, używa się wewnątrz niego specjalnego języka (DSL). Nie możesz tam wrzucić zwykłych komponentów @Composable bezpośrednio – musisz je opakować w dedykowane funkcje, takie jak item lub items.

1. Kontener LazyColumn { ... }

To jest baza. Wszystko, co znajdzie się wewnątrz tych klamer, staje się częścią przewijanej listy. Posiada on parametry takie jak modifier, contentPadding (wewnętrzny margines listy) czy verticalArrangement (odstępy między elementami).

2. Funkcja item { ... }

Służy do dodania **jednego, konkretnego elementu** do listy.

- W przykładzie użyliśmy go do stworzenia nagłówka listy (Moja Lista Kontaktów).
- Możesz go użyć również na samym dole, np. do dodania przycisku "Wczytaj więcej" albo stopki.

3. Funkcja items(kolekcja) { element -> ... }

To serce dynamicznych list. Przyjmuje listę danych (np. List<String>, List<User>) i dla każdego obiektu z tej listy automatycznie generuje osobny wiersz.

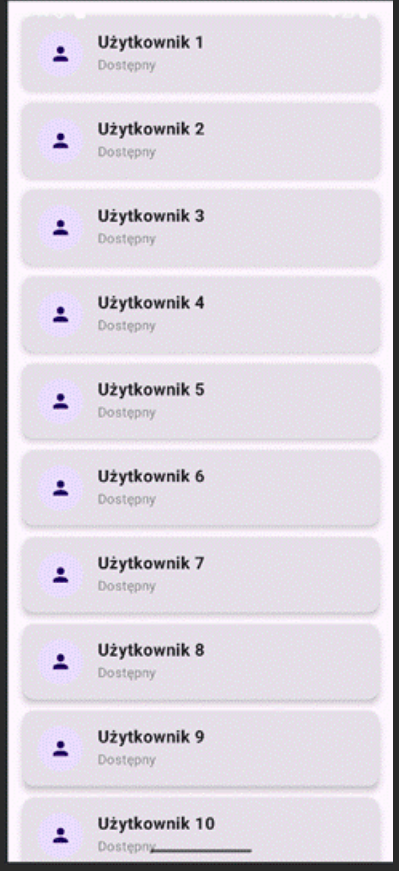
- W naszym przypadku listaImion ma 100 pozycji. Funkcja items przechodzi przez każdą z nich, wyciąga pojedyncze imie i przekazuje je do środka, gdzie definiujemy, jak ten wiersz ma wyglądać (u nas: zwykły Text).

Przydatna wskazówka na przyszłość (dobre praktyki)

Gdy Twoja lista stanie się bardziej skomplikowana (elementy będą usuwane, przesuwane lub zmieniane), warto dodać parametr key wewnątrz funkcji items. Pomaga to Jetpack Compose zrozumieć, który element jest który, bez ponownego przerysowywania całej listy:

```
items(
items = listaImion,
key = { imie -> imie } // Unikalny klucz dla każdego wiersza
) { imie ->
    Text(text = imie)
}
```

Przykład 2. *Rozbudowana lista*



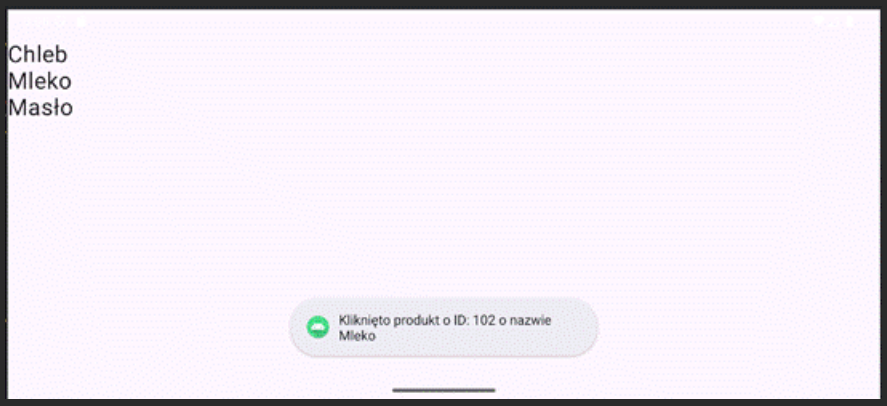

```
// 1. Model danych reprezentujący użytkownika
data class User(val id: Int, val name: String, val status: String)

@Composable
fun RozbudowanaListaUserow() {
    // Generujemy listę 50 unikalnych użytkowników
    val users = List(50) { index ->
        User(id = index, name = "Użytkownik ${index + 1}", status = "Dostępny")
    }

    LazyColumn(
        modifier = Modifier.fillMaxSize(),
        contentPadding = PaddingValues(16.dp), // Margines wokół CAŁEJ listy
        verticalArrangement = Arrangement.spacedBy(12.dp) // Automatyczne odstępy między kartami
    ) {
        items(
            items = users,
            key = { user -> user.id } // Unikalny klucz dla wydajności
        ) { user ->
            // 2. Wykorzystanie komponentu Card jako wiersza
            Card(
                modifier = Modifier.fillMaxWidth(),
                elevation = CardDefaults.cardElevation(defaultElevation = 4.dp) // Efekt cienia
            ) {
                // 3. Układ poziomy wewnątrz karty (Avatar + Teksty)
                Row(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(16.dp),
                    verticalAlignment = Alignment.CenterVertically
                ) {
                    // 4. "Avatar" czyli kółko z ikoną faceta
                    Box(
                        modifier = Modifier
                            .size(48.dp)
                            .background(MaterialTheme.colorScheme.primaryContainer, shape =
CircleShape),
                        contentAlignment = Alignment.Center
```

```
                ) {
                    Icon(
                        imageVector = Icons.Default.Person,
                        contentDescription = "Ikona użytkownika",
                        tint = MaterialTheme.colorScheme.onPrimaryContainer
                    )
                }

                Spacer(modifier = Modifier.width(16.dp)) // Odstęp między avatarem a tekstem

                // 5. Układ pionowy na dwa teksty (Imię + Status)
                Column {
                    Text(
                        text = user.name,
                        fontSize = 18.sp,
                        fontWeight = FontWeight.Bold
                    )
                    Text(
                        text = user.status,
                        fontSize = 14.sp,
                        color = Color.Gray
                    )
                }
            }
        }
    }
}
}
```

Opis nowych elementów, które tu wskoczyły:

- **Card** - Sprawia, że każdy użytkownik ma swoją własną, estetyczną "wizytówkę" z zaokrąglonymi rogami i delikatnym cieniem (elevation = 4.dp), przez co ładnie odkleja się od tła ekranu.
- **Row (Wiersz)**: Układa elementy poziomo, obok siebie. Dzięki temu najpierw mamy avatar, potem odstęp, a na końcu teksty. Parametr verticalAlignment = Alignment.CenterVertically dba o to, aby wszystko w tym wierszu było idealnie wyrównane w pionie (w jednej linii).
- **Box ze** shape = CircleShape: Użyliśmy zwykłego kontenera Box, nadaliśmy mu okrągły kształt, tło i wrzuciliśmy do środka ikonę Icons.Default.Person. W ten prosty sposób stworzyliśmy schludny, zastępczy avatar użytkownika.
- **Column (Kolumna)**: Znajduje się po prawej stronie avatara. Układa dwa teksty (imię oraz status) jeden pod drugim.
- **Arrangement.spacedBy(12.dp)** w LazyColumn: Bardzo wygodny parametr. Zamiast dodawać marginesy do każdej karty z osobna, mówimy liście: *"automatycznie wstawiaj 12 dp odstępu pomiędzy każdy element"*.

Przykład 3. *Pobieranie danych w czasie kliknięcia*




```
data class Produkt(val id: Int, val nazwa: String)

@Composable
fun ListaProduktowZKlikaniem() {
```

```
    val context = LocalContext.current // Potrzebne tylko do wyświetlenia Toastu

    val produkty = listOf(
        Produkt(id = 101, nazwa = "Chleb"),
        Produkt(id = 102, nazwa = "Mleko"),
        Produkt(id = 103, nazwa = "Masło")
    )

    LazyColumn {
        item { Spacer(modifier = Modifier.height(33.dp)) }
        items(
            items = produkty,
            key = { produkt -> produkt.id } // 1. Systemowy klucz dla Compose (ID)
        ) { produkt ->                      // 2. Tutaj mamy bezpośredni dostęp do całego obiektu
            Text(
                fontSize = 24.sp,
                text = produkt.nazwa,
                modifier = Modifier
                    .clickable {
                        // 3. BEZPOŚREDNIE UŻYCIE: Pobieramy ID lub nazwę kliknięgo elementu
                        Toast.makeText(
                            context,
                            "Kliknięto produkt o ID: ${produkt.id} o nazwie ${produkt.nazwa}",
                            Toast.LENGTH_SHORT
                        ).show()
                    }
            )
        }
    }
}
```

Przykład 4. ***Obsługa podwójnego kliknięcia oraz usuwanie elementów listy LazyColumn***

```
data class Produkt(val id: Int, val nazwa: String)

@OptIn(ExperimentalFoundationApi::class)
@Composable
fun ListaJednoObiektowa() {
    val context = LocalContext.current

    // JEDNA LISTA: Łączy w sobie źródło danych i stan Compose
    val produkty = remember {
        mutableStateListOf(
            Produkt(id = 101, nazwa = "Chleb"),
            Produkt(id = 102, nazwa = "Mleko"),
            Produkt(id = 103, nazwa = "Masło")
        )
    }

    LazyColumn {
        items(
            items = produkty,
            key = { produkt -> produkt.id }
        ) { produkt ->
            Text(
                text = produkt.nazwa,
                modifier = Modifier
                    .fillMaxWidth()
                    .combinedClickable(
                        onClick = {
                            Toast.makeText(context, "Kliknięto: ${produkt.nazwa}",
Toast.LENGTH_SHORT).show()
                        },
```

```
                        onDoubleClick = {
                            // Usuwasz tylko raz, z jednej listy!
                            produkty.remove(produkt)

                            Toast.makeText(
                                context,
                                "W liście zostało: ${produkty.size} elementów",
                                Toast.LENGTH_SHORT
                            ).show()
                        }
                    )
                    .padding(16.dp)
            )
        }
    }
}
```

Funkcja mutableStateListOf() tworzy obiekt typu SnapshotStateList. Jest to specjalna odmiana zwykłej listy z Kotlina (MutableList), którą inżynierowie z Google wyposażyli w "uszy".

Działa ona dualnie:

1. **Dla Ciebie (w kodzie):** Zachowuje się jak zwykła lista. Możesz z niej usuwać elementy (.remove()), dodawać nowe (.add()) czy sprawdzać jej rozmiar (.size). Element fizycznie z niej znika.
2. **Dla Jetpack Compose:** Automatycznie wysyła sygnał do ekranu o zmianie struktury danych, wymuszając odświeżenie widoku.

Przykład 5. *Obsługa podwójnego kliknięcia oraz usuwanie na stałe elementów listy.*

Przykład pokazuje jak przekazać listę i funkcje obsługi zdarzenia jako parametr

```
data class Produkt(val id: Int, val nazwa: String)

@Composable
fun EkranZakupowZObrotem() {
    val context = LocalContext.current

    // Używamy rememberSaveable zamiast zwykłego remember.
    // Korzystamy z delegata 'by', co daje nam bezpośredni dostęp do listy.
    var listaProduktow by rememberSaveable {
        mutableStateOf(
            listOf(
                Produkt(id = 101, nazwa = "Chleb"),
                Produkt(id = 102, nazwa = "Mleko"),
                Produkt(id = 103, nazwa = "Masło")
            )
        )
    }

    ListaProduktowKomponent(
        produkty = listaProduktow,
        onUsunProdukt = { produktDoUsuniecia ->
            // Aby Compose zauważył zmianę w mutableStateOf,
            // musimy przypisać NOWĄ listę bez usuniętego elementu.
            listaProduktow = listaProduktow.filter { it.id != produktDoUsuniecia.id }
            Toast.makeText(context, "Usunięto: ${produktDoUsuniecia.nazwa}",
Toast.LENGTH_SHORT).show()
        }
    )
}

@OptIn(ExperimentalFoundationApi::class)
```

```
@Composable
fun ListaProduktowKomponent(
    produkty: List<Produkt>,
    onUsunProdukt: (Produkt) -> Unit
) {
    val context = LocalContext.current
    LazyColumn {
        items(items = produkty, key = { it.id }) { produkt ->
            Text(
                text = produkt.nazwa,
                modifier = Modifier
                    .fillMaxWidth()
                    .combinedClickable(
                        onClick = { Toast.makeText(context, "Kliknięto: ${produkt.nazwa}",
Toast.LENGTH_SHORT).show() },
                        onDoubleClick = { onUsunProdukt(produkt) }
                    )
                    .padding(16.dp)
            )
        }
    }
}
```

Przykład 6. ***Przykład pokazuje jak najprościej wykorzystać obsługę listy z architektury ViewModel***

```
data class Produkt(val id: Int, val nazwa: String)

// 2. ViewModel - prosty i odporny na obroty ekranu
class ProstyZakupyViewModel : ViewModel() {

    // Zapamiętywana lista wewnątrz ViewModelu
    val listaProduktow = mutableStateListOf<Produkt>()

    // Funkcja ładująca dane z bazy/serwera
    fun zaladujDanePoczatkowe(produkty: List<Produkt>) {
        if (listaProduktow.isEmpty()) {
            listaProduktow.addAll(produkty)
        }
    }

    // Intuicyjne usuwanie
    fun usunProdukt(produkt: Produkt) {
        listaProduktow.remove(produkt)
    }
}

// 3. Główny Ekran
@Composable
fun EkranUproszczony(
    produktyZSerwera: List<Produkt>,
    viewModel: ProstyZakupyViewModel = viewModel()
) {
    val context = LocalContext.current

    // NAPRAWA: LaunchedEffect dba o to, aby dane załadowały się TYLKO RAZ przy starcie ekranu.
    // Obrót telefonu nie uruchomi tej funkcji ponownie.
    LaunchedEffect(Unit) {
        viewModel.zaladujDanePoczatkowe(produktyZSerwera)
    }

    ListaProduktowKomponent(
        produkty = viewModel.listaProduktow,
```

```
        onUsunProdukt = { produkt ->
            viewModel.usunProdukt(produkt)
            Toast.makeText(context, "Usunięto: ${produkt.nazwa}", Toast.LENGTH_SHORT).show()
        }
    )
}

// 4. Komponent wyświetlający listę
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun ListaProduktowKomponent(
    produkty: List<Produkt>,
    onUsunProdukt: (Produkt) -> Unit
) {
    val context = LocalContext.current
    LazyColumn {
        items(items = produkty, key = { it.id }) { produkt ->
            Text(
                text = produkt.nazwa,
                modifier = Modifier
                    .fillMaxWidth()
                    .combinedClickable(
                        onClick = {
                            Toast.makeText(context, "Kliknięto: ${produkt.nazwa}",
Toast.LENGTH_SHORT).show()
                        },
                        onDoubleClick = {
                            onUsunProdukt(produkt)
                        }
                    )
                    .padding(16.dp)
            )
        }
    }
}
```

Wywołanie:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        // 1. Przygotowujemy dane startowe (np. symulacja pobrania z bazy danych/sieci)
        val produktyNaStart = listOf(
            Produkt(id = 1, nazwa = "Aparat fotograficzny"),
            Produkt(id = 2, nazwa = "Słuchawki bezprzewodowe"),
            Produkt(id = 3, nazwa = "Smartwatch")
        )
        setContent {
            MaterialTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                  //  EkranNowoczesny(produktyZSerwera = produktyNaStart)
                    EkranUproszczony(produktyNaStart)
                }
            }
        }
    }
}
```

1. Model danych (Produkt)

- To zwykły szablon (klasa danych), który opisuje, jak wygląda pojedynczy element na naszej liście.
- Każdy produkt ma swój unikalny numer identyfikacyjny (id) oraz tekstową nazwę.

2. Centrum Zarządzania Dane i Pamięcią (ProstyZakupyViewModel)

- ViewModel: To specjalna klasa od Google, która działa poza samym ekranem. Jej główną funkcją jest to, że nie znika podczas obracania telefonu. Dane w niej zapisane są w 100% bezpieczne.
- mutableStateListOf<Produkt>(): To inteligentna lista wyposażona w system powiadamiania. Kiedy dodasz lub usuniesz z niej produkt, lista automatycznie "daje znać" interfejsowi Compose, że trzeba odświeżyć widok na ekranie.
- usunProdukt(produkt): Prosta funkcja logiczna. Wywołujesz ją, podajesz obiekt, a lista usuwa go ze swoich zasobów za pomocą jednej komendy.

3. Ekran Główny i Kontrola Cyklu Życia (EkranUproszczony)

- LaunchedEffect(Unit) { ... }: To strażnik porządku. Ponieważ ekrany w Compose potrafią bez przerwy odświeżać swój kod, ten blok dba o to, aby funkcja ładująca dane z Serwera (zaladujDanePoczatkowe) wykonała się dokładnie jeden raz – przy włączeniu ekranu. Dzięki temu obracanie telefonu nie resetuje i nie dubluje listy.
- Przekazywanie akcji (Lambdy): W tym miejscu łączymy wygląd z logiką. Mówimy liście: *"Gdy użytkownik zasygnalizuje usunięcie, uruchom funkcję usunProdukt z naszego ViewModelu"*.

4. Niezależny Wygląd Listy (ListaProduktowKomponent)

- LazyColumn & items: To bardzo wydajny odpowiednik tradycyjnej przewijanej listy. Rysuje na ekranie tylko te elementy, które użytkownik aktualnie widzi (oszczędzając pamięć telefonu).
- key = { it.id }: Informacja dla systemu, który produkt jest który. Dzięki temu, gdy usuniesz np. drugi element, system dokładnie wie, który wiersz wymazać i potrafi ładnie przesunąć pozostałe elementy w górę.
- combinedClickable: Moduł wykrywania gestów użytkownika. Został zaprogramowany tak, aby rozróżniać dwa zachowania:
    - Pojedyncze kliknięcie (onClick): Wyświetla szybki dymek z informacją (Toast).
    - Podwójne kliknięcie (onDoubleClick): Nie usuwa nic samodzielnie, tylko wysyła sygnał w górę (onUsunProdukt(produkt)), prosząc ViewModel o zajęcie się sprawą.

Przykład 7. *Przykład pokazuje jak wykorzystać obsługę listy z użyciem StateFlaw – architektura ViewModel*

Oto kompletny, gotowy do uruchomienia przykład implementacji z użyciem **StateFlow**. Jest to obecnie oficjalnie zalecany przez Google standard budowania aplikacji na Androida.

Struktura ta opiera się na **architekturze reaktywnej**: ViewModel wystawia strumień danych (jak stacja radiowa), a interfejs w Compose "słucha" tego strumienia i natychmiast przerysowuje się, gdy pojawią się nowe wiadomości.

```
data class Produkt(val id: Int, val nazwa: String)

// 2. ViewModel obsługujący StateFlow
class NowoczesnyZakupyViewModel(poczatkowaLista: List<Produkt>) : ViewModel() {

    // Prywatny strumień mutowalny (wewnętrzny)
    private val _uiState = MutableStateFlow<List<Produkt>>(poczatkowaLista)

    // Publiczny strumień tylko do odczytu (zewnętrzny dla Compose)
    val uiState: StateFlow<List<Produkt>> = _uiState.asStateFlow()

    fun usunProdukt(produktDoUsuniecia: Produkt) {
        // Bezpieczna aktualizacja stanu poprzez filtrowanie
        _uiState.update { aktualnaLista ->
            aktualnaLista.filter { it.id != produktDoUsuniecia.id }
        }
    }

    // Fabryka do przekazywania parametrów na start
```

```
    companion object {
        fun Factory(produkty: List<Produkt>): ViewModelProvider.Factory =
            object : ViewModelProvider.Factory {
                @Suppress("UNCHECKED_CAST")
                override fun <T : ViewModel> create(modelClass: Class<T>): T {
                    return NowoczesnyZakupyViewModel(produkty) as T
                }
            }
    }
}

// 3. Główny Ekran (Kontener)
@Composable
fun EkranNowoczesny(produktyZSerwera: List<Produkt>) {
    val context = LocalContext.current

    // Inicjalizacja ViewModelu z fabryką
    val mojViewModel: NowoczesnyZakupyViewModel = viewModel(
        factory = NowoczesnyZakupyViewModel.Factory(produktyZSerwera)
    )

    // Konwersja strumienia StateFlow na stan zrozumiały dla Jetpack Compose
    val listaProduktow by mojViewModel.uiState.collectAsStateWithLifecycle()

    ListaProduktowKomponent(
        produkty = listaProduktow,
        onUsunProdukt = { produkt ->
            mojViewModel.usunProdukt(produkt)
            Toast.makeText(context, "Usunięto: ${produkt.nazwa}", Toast.LENGTH_SHORT).show()
        }
    )
}

// 4. Komponent interfejsu (czysty i niezależny)
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun ListaProduktowKomponent(
    produkty: List<Produkt>,
    onUsunProdukt: (Produkt) -> Unit
) {
    val context = LocalContext.current
    LazyColumn {
        items(items = produkty, key = { it.id }) { produkt ->
            Text(
                text = produkt.nazwa,
                modifier = Modifier
                    .fillMaxWidth()
                    .combinedClickable(
                        onClick = { Toast.makeText(context, "Kliknięto: ${produkt.nazwa}",
Toast.LENGTH_SHORT).show() },
                        onDoubleClick = { onUsunProdukt(produkt) }
                    )
                    .padding(16.dp)
            )
        }
    }
}
```

Wywołanie:

```
        val produktyNaStart = listOf(
            Produkt(id = 1, nazwa = "Aparat fotograficzny"),
            Produkt(id = 2, nazwa = "Słuchawki bezprzewodowe"),
            Produkt(id = 3, nazwa = "Smartwatch")
        )
        setContent {
```

```
            MaterialTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    EkranNowoczesny(produktyZSerwera = produktyNaStart)
                }
            }
        }
    }
}
```

Aby zrozumieć StateFlow, najlepiej wyobrazić sobie **wzorzec projektowy Obserwator (Publikacja-Subskrypcja)**. ViewModel posiada dane i rozgłasza ich zmianę, a interfejs graficzny reaguje na te powiadomienia.

1. Podział na _uiState oraz uiState (Enkapsulacja)

W klasie ViewModel widzisz dwie zmienne o niemal identycznej nazwie:

- _uiState (z podkreśleniem) to **MutableStateFlow**. Jest private, co oznacza, że tylko kod wewnątrz tego konkretnego ViewModelu może modyfikować dane (np. dodawać lub usuwać produkty). To chroni aplikację przed chaosem – żaden zewnętrzny ekran nie zmieni danych "za plecami" ViewModelu.
- uiState (bez podkreślenia) to zwykły **StateFlow** uzyskany przez .asStateFlow(). Jest publiczny, ale pozwala **wyłącznie na odczyt**. Ekran Compose może go podglądać, ale nie może w niego ingerować.

2. Metoda _uiState.update { ... }

W tradycyjnym programowaniu usunąłbyś element z listy poprzez .remove(). W świecie StateFlow listy są **niezmodyfikowalne** (Immutable). Gdy usuwasz produkt, funkcja .update bierze aktualną listę, za pomocą .filter tworzy **zupełnie nową listę** (bez kliknięgo produktu) i "wpycha" tę nową listę do strumienia. StateFlow natychmiast zauważa, że pojawił się zupełnie nowy obiekt listy i alarmuje subskrybentów.

3. collectAsStateWithLifecycle()

To najważniejsza linijka po stronie Compose. StateFlow sam w sobie jest narzędziem czystego Kotlina (nie wie, co to jest Android czy Compose). Funkcja collectAsStateWithLifecycle() to pomost. Mówi ona do Compose: *"Zacznij słuchać tego strumienia. Ilekroć ViewModel wrzuci tam nową listę, przechwyć ją, zamień na stan i odśwież ekran"*.

Dodatkowo człon WithLifecycle dba o **bezpieczeństwo pamięci** – jeśli użytkownik zminimalizuje aplikację i przejdzie do ekranu głównego telefonu, ta funkcja automatycznie przestanie słuchać strumienia, oszczędzając baterię i procesor.

Podsumowanie przepływu danych (Unidirectional Data Flow)

1. Użytkownik klika dwukrotnie w produkt na ekranie.
2. Komponent interfejsu nie usuwa nic sam – odpala callback onUsunProdukt.
3. ViewModel odbiera sygnał i bezpiecznie aktualizuje swój prywatny MutableStateFlow.
4. Publiczny StateFlow emituje nową listę.
5. collectAsStateWithLifecycle() w Compose wyłapuje zmianę.
6. Ekran się przerysowuje, a usunięty element znika.

Dzięki temu dane krążą tylko w jedną stronę, co drastycznie zmniejsza liczbę błędów w aplikacji.

II. Retrofit

Retrofit to biblioteka typu *Type-Safe HTTP client* dla systemów Android i języka Java/Kotlin, rozwijana przez firmę Square. Jej głównym zadaniem jest translacja interfejsu napisanego w Kotlinie na zapytania HTTP za pomocą adnotacji (np. @GET, @POST).

W nowoczesnym programowaniu na platformę Android standardem jest wykonywanie operacji sieciowych w sposób asynchroniczny i nieblokujący głównego wątku (UI Thread). Wykorzystujemy do tego **Kotlin Coroutines** oraz strukturę asynchronicznych strumieni danych **Flow**.

Uprawnienia AndroidManifest i konfiguracja build.gradle.kts

Do poprawnego działania aplikacji wymagane jest zezwolenie na komunikację z siecią Internet oraz dodanie odpowiednich zależności w pliku modułu aplikacji (build.gradle.kts).

```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    // Wtyczka wymagana do kompilacji klas serializowalnych KotlinX
    id("org.jetbrains.kotlin.plugin.serialization") version "2.0.0"
}
```

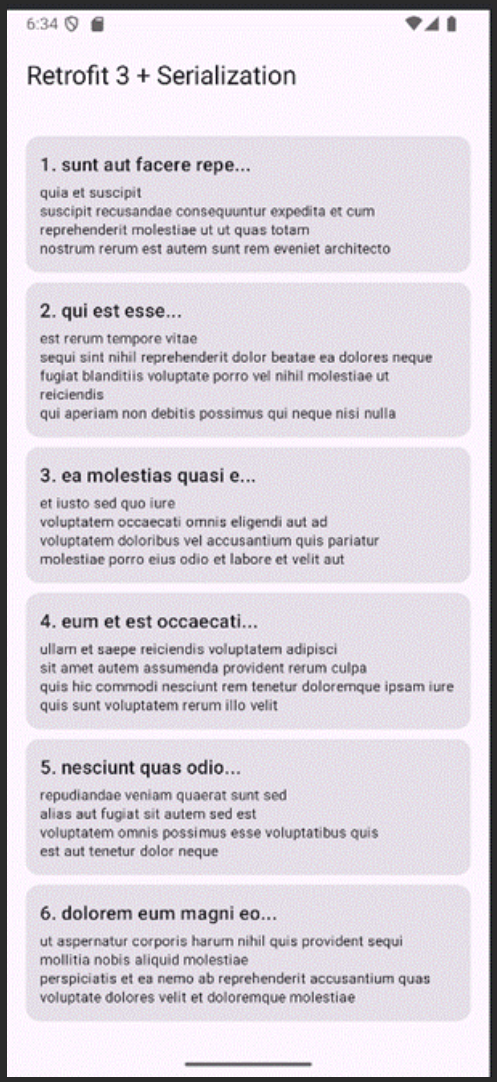
W sekcji dependencies dodaj najnowsze wersje bibliotek Retrofit, OkHttp oraz konwertera JSON:

```
dependencies {
    // Retrofit 3.x core + OkHttp 4.12+ (z pełnym natywnym wsparciem dla Kotlina)
    implementation("com.squareup.retrofit2:retrofit:3.0.0")

    // Nowoczesny i rekomendowany przez Google konwerter JSON
    implementation("org.jetbrains.kotlinx:kotlinx-serialization-json:1.6.3")
    implementation("com.squareup.retrofit2:converter-kotlinx-serialization:3.0.0")
}
```

Dodaj uprawnienie w pliku AndroidManifest.xml (wewnątrz znacznika <manifest>, przed <application>):

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

- Przykład 1.



1. Model Danych (Data Transfer Object – DTO)

```
@Serializable
data class SimplePost(
    @SerialName("userId") val userId: Int,
    @SerialName("id") val id: Int,
    @SerialName("title") val title: String,
    @SerialName("body") val body: String
)
```

Ta część odpowiada za **odzwierciedlenie struktury danych**, jakie zwraca serwer (w formacie JSON), na obiekt zrozumiały dla języka Kotlin.

- **@Serializable**: To najważniejsza adnotacja z biblioteki kotlinx.serialization. Informuje ona kompilator Kotlina podczas budowania aplikacji, że dla tej klasy ma zostać automatycznie wygenerowany kod parsujący (serializer/deserializer). Dzięki temu aplikacja nie musi marnować zasobów procesora na tzw. *refleksję* (przeszukiwanie struktury klasy w czasie działania programu), co było wadą starszych rozwiązań typu Gson.
- **@SerialName("klucz_w_json")**: Wiąże zmienną w Kotlinie z konkretnym kluczem w pliku JSON. Serwer przysyła nam np. "userId": 1. Dzięki tej adnotacji, nawet jeśli zmienimy nazwę zmiennej w Kotlinie na val identyfikatorUzytkownika: Int, biblioteka nadal będzie wiedziała, skąd pobrać wartość.
- **data class**: Klasa danych w Kotlinie automatycznie generuje metody użytkowe (takie jak toString() czy equals()), co ułatwia debugowanie pobranych danych.

2. Interfejs API (Deklaracja kontraktu)

```
interface SimpleApiService {
    @GET("posts")
    suspend fun getPosts(): List<SimplePost>
}
```

W tym miejscu programista nie pisze kodu wykonawczego. Definiuje jedynie **kontrakt** – czyli to, jak mają wyglądać zapytania do serwera. Właściwą implementację (kod, który naprawdę łączy się z Internetem) Retrofit wygeneruje sam.

- **@GET("posts")**: Adnotacja Retrofita wskazująca metodę HTTP (GET – pobieranie danych) oraz względny punkt końcowy (endpoint) adresu URL.
- **suspend**: Słowo kluczowe języka Kotlin (Coroutines). Informuje kompilator, że jest to **funkcja zawieszana**. Może ona trwać długo (oczekiwanie na odpowiedź z serwera), ale potrafi "zawiesić" swoje działanie bez blokowania wątku, na którym została uruchomiona.
- **List<SimplePost>**: Definicja typu zwracanego. Retrofit automatycznie domyśli się, że serwer zwraca tablicę obiektów JSON [...] i sparsuje ją do kolekcji List w Kotlinie.

3. Konfiguracja Klienta HTTP (Wzorzec Singleton)

```
object SimpleRetrofitClient {
    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"

    private val jsonConfig = Json {
        ignoreUnknownKeys = true
    }

    private val contentType = "application/json".toMediaType()

    val api: SimpleApiService by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(jsonConfig.asConverterFactory(contentType))
            .build()
            .create(SimpleApiService::class.java)
```

```
    }
}
```

Ta część odpowiada za powołanie do życia i konfigurację mechanizmu sieciowego. Użycie słowa kluczowego **object** implementuje wzorzec **Singleton** – gwarantuje, że w całej aplikacji będzie istniała tylko jedna, współdzielona instancja klienta HTTP (co oszczędza pamięć RAM urządzenia).

- **BASE_URL**: Adres bazowy serwera. Retrofit połączy go z adnotacją z interfejsu, tworząc pełny adres: https://jsonplaceholder.typicode.com/posts.
- **ignoreUnknownKeys = true**: Kluczowa konfiguracja produkcyjna. Jeśli serwer w przyszłości doda do JSON-a nowe pole, którego nie mamy w naszej klasie **SimplePost**, aplikacja zignoruje je i **nie wykrzyczy błędu (crashu)**.
- **addConverterFactory(...)**: To jest miejsce, w którym "wstrzykujemy" najnowszy konwerter serializacji Kotlina do Retrofita. Informujemy go: *"Do zamiany tekstu na obiekty używaj KotlinX Serialization z nagłówkiem Content-Type: application/json"*.
- **by lazy**: Inicjalizacja leniwa. Obiekt **api** zostanie utworzony dopiero wtedy, gdy po raz pierwszy spróbujemy pobrać dane. Jeśli użytkownik nigdy nie wejdzie na ekran sieciowy, obiekt w ogóle nie zajmie pamięci.

4. Interfejs Użytkownika i Asynchroniczność (Jetpack Compose)

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MinimalRetrofitScreen() {
    var postsList by remember { mutableStateOf<List<SimplePost>>(emptyList()) }
    var isLoading by remember { mutableStateOf(true) }
    var errorMessage by remember { mutableStateOf<String?>(null) }

    // Pobranie danych asynchronicznie przy uruchomieniu ekranu
    LaunchedEffect(Unit) {
        try {
            val result = withContext(Dispatchers.IO) {
                SimpleRetrofitClient.api.getPosts()
            }
            postsList = result
            isLoading = false
        } catch (e: Exception) {
            errorMessage = "Błąd: ${e.localizedMessage}"
            isLoading = false
        }
    }

    Scaffold(
        topBar = { TopAppBar(title = { Text("Retrofit 3 + Serialization") }) }
    ) { paddingValues ->
        Box(
            modifier = Modifier
                .fillMaxSize()
                .padding(paddingValues)
                .padding(16.dp),
            contentAlignment = Alignment.Center
        ) {
            if (isLoading) {
                CircularProgressIndicator()
            } else if (errorMessage != null) {
                Text(text = errorMessage!!, color = MaterialTheme.colorScheme.error)
            } else {
                LazyColumn(modifier = Modifier.fillMaxSize()) {
                    items(postsList) { item ->
                        Card(
                            modifier = Modifier
                                .fillMaxWidth()
                                .padding(vertical = 4.dp)
                        ) {
```

```
                                Column(modifier = Modifier.padding(12.dp)) {
                                    Text(
                                        text = "${item.id}. ${item.title.take(20)}...",
                                        style = MaterialTheme.typography.titleMedium
                                    )
                                    Spacer(modifier = Modifier.height(4.dp))
                                    Text(
                                        text = item.body,
                                        style = MaterialTheme.typography.bodySmall
                                    )
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Ten fragment kodu to **serce interfejsu użytkownika (UI) i logiki asynchronicznej**. W nowoczesnym frameworku Jetpack Compose kod odpowiedzialny za wygląd ekranu ściśle współpracuje z kodem zarządzającym danymi.

1. Adnotacja i definicja stanu (Zmienne reaktywne)

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun MinimalRetrofitScreen() {
    var postsList by remember { mutableStateOf<List<SimplePost>>(emptyList()) }
    var isLoading by remember { mutableStateOf(true) }
    var errorMessage by remember { mutableStateOf<String?>(null) }
```

@OptIn(ExperimentalMaterial3Api::class): Niektóre zaawansowane komponenty wizualne z biblioteki Material 3 (w tym przypadku TopAppBar) są oznaczone przez Google jako eksperymentalne. Ta adnotacja to formalna zgoda programisty na ich użycie – bez niej kompilator zgłosi błąd.

@Composable: Informuje kompilator, że ta funkcja służy do rysowania interfejsu użytkownika. Funkcje kompozycjonalne mogą być wywoływane tylko z poziomu innych funkcji @Composable.

mutableStateOf(...): Tworzy tzw. **Stan (State)**. W Compose interfejs automatycznie nasłuchuje zmian tych zmiennych. Jeśli zmienisz wartość isLoading z true na false, Compose natychmiast przerysuje (zrekomponuje) odpowiednie fragmenty ekranu.

remember { ... }: Smartfony przerysowują ekran wielokrotnie (np. przy obrocie ekranu lub animacji). Bez słowa remember, przy każdym przerysowaniu zmienne resetowałyby się do wartości początkowych (np. lista znowu robiłaby się pusta). remember nakazuje pamięci podręcznej telefonu zachować aktualną wartość zmiennej pomiędzy przerysowaniami.

by: Słowo kluczowe w Kotlinie służące do delegacji właściwości. Dzięki niemu możemy używać zmiennej postsList jak zwykłej listy (np. postsList = nowaLista), zamiast pisać dłuższego .value (postsList.value = nowaLista).

2. Efekt uboczny i współbieżność (Asynchroniczne pobieranie danych)

```
LaunchedEffect(Unit) {
    try {
        val result = withContext(Dispatchers.IO) {
            SimpleRetrofitClient.api.getPosts()
        }
        postsList = result
        isLoading = false
    } catch (e: Exception) {
        errorMessage = "Błąd: ${e.localizedMessage}"
        isLoading = false
    }
}
```

Ta sekcja odpowiada za bezpieczne, jednorazowe pobranie danych w tle.

- **LaunchedEffect(Unit)**: Wykonywanie zapytań sieciowych bezpośrednio w ciele funkcji @Composable jest surowo zabronione, ponieważ sieć byłaby odpytywana przy każdym, najmniejszym przerysowaniu ekranu (pętla nieskończona). **LaunchedEffect** otwiera bezpieczny blok dla **korutyn** (wątków pobocznych), który uruchamia się **dokładnie raz**, gdy ten ekran po raz pierwszy pojawia się na telefonie. Klucz **Unit** (odpowiednik void) oznacza, że ten blok nigdy nie uruchomi się ponownie z powodu zmiany jakichkolwiek parametrów.
- **withContext(Dispatchers.IO)**: Android wymaga, aby operacje wejścia/wyjścia (zapis na dysku, sieć) odbywa się poza głównym wątkiem systemowym. Ta funkcja tymczasowo przenosi wykonanie linijki z Retrofitem na wątek roboczy IO. Gdy Retrofit skończy pobierać dane, sterowanie wraca do głównego wątku, by bezpiecznie zaktualizować interfejs.
- **Modyfikacja stanu w try-catch**:
  - Jeśli operacja **się uda** (try), przypisujemy wynik do postsList, a flagę ładowania isLoading zmieniamy na false.
  - Jeśli nie ma internetu lub serwer padnie (catch), łapiemy wyjątek Exception, zapisujemy tekst błędu do errorMessage i również wyłączamy kółko ładowania (isLoading = false).

3. Szkielet ekranu i kontener (Scaffold & Box)

```
Scaffold(
topBar = { TopAppBar(title = { Text("Retrofit 3 + Serialization") }) }
) { paddingValues ->
    Box(
        modifier = Modifier
            .fillMaxSize()
            .padding(paddingValues)
            .padding(16.dp),
        contentAlignment = Alignment.Center
    ) {
```

- **Scaffold**: To gotowy szablon ekranu zgodny z wytycznymi Google Material Design. Posiada dedykowane sloty (miejsca) na standardowe elementy aplikacji, takie jak górny pasek narzędzi (topBar), dolne menu nawigacyjne czy wysuwany boczny panel.
- **paddingValues**: Scaffold oblicza, ile miejsca na ekranie zajmuje górny pasek (TopAppBar) i przekazuje tę informację do wnętrza w zmiennej paddingValues. Musimy przekazać ten parametr do głównego kontenera, aby treść naszej aplikacji nie "schowała się" pod górnym paskiem.
- **Box**: Najprostszy kontener w Compose. Działa jak stos – układa komponenty jeden na drugim (w osi Z). Używamy go tutaj, aby wyśrodkować elementy.
- **Modifier.fillMaxSize()**: Modyfikator, który rozciąga nasz kontener Box na całą szerokość i wysokość dostępnego ekranu.
- **contentAlignment = Alignment.Center**: Nakazuje kontenerowi Box, aby wszystko, co znajdzie się w jego środku (kółko ładowania, komunikat błędu), było idealnie wyśrodkowane na ekranie telefonu.

4. Drzewo decyzyjne UI (Warunkowe renderowanie widoku)

```
if (isLoading) {
    CircularProgressIndicator()
} else if (errorMessage != null) {
    Text(text = errorMessage!!, color = MaterialTheme.colorScheme.error)
} else {
    LazyColumn(modifier = Modifier.fillMaxSize()) {
        items(postsList) { item ->
            Card(
                modifier = Modifier
```

```
                .fillMaxWidth()
                .padding(vertical = 4.dp)
            ) {
                Column(modifier = Modifier.padding(12.dp)) {
                    Text(text = "${item.id}. ${item.title.take(20)}...", style =
                        MaterialTheme.typography.titleMedium)
                    Spacer(modifier = Modifier.height(4.dp))
                    Text(text = item.body, style = MaterialTheme.typography.bodySmall)
                }
            }
        }
    }
}
```

W Jetpack Compose **nie** ukrywamy widoków za pomocą właściwości typu view.visibility = View.GONE. Po prostu używamy standardowych instrukcji warunkowych języka Kotlin (if-else).

1. **Warunek 1 (if (isLoading))**: Jeśli stan isLoading ma wartość true, Compose rysuje na ekranie tylko kręcące się kółko postępu (CircularProgressIndicator).
2. **Warunek 2 (else if (errorMessage != null))**: Jeśli ładowanie się skończyło, ale pojawił się błąd, zamiast kółka rysowany jest czerwony tekst (Text) z informacją o błędzie. Wykrzykniki errorMessage!! (tzw. *not-null assertion operator*) informują kompilator, że jesteśmy pewni, iż w tej gałęzi kodu ta zmienna nie jest nullem.
3. **Warunek domyślny (else)**: Jeśli dane zostały pobrane poprawnie (isLoading == false oraz errorMessage == null), uruchamia się właściwy widok listy:
    - **LazyColumn**: Wydajna, przewijana lista. Dynamicznie tworzy i niszczy elementy widoczne na ekranie.
    - **items(postsList)**: Pętla, która przechodzi przez każdy element pobrany z internetu i generuje dla niego osobną kartę.
    - **Card / Column**: Komponenty stylizujące – tworzą białą plakietkę z zaokrąglonymi rogami (Card), w której układa się teksty pionowo, jeden pod drugim (Column).
    - **item.title.take(20)**: Funkcja pomocnicza Kotlina, która obcina długi tytuł posta z API do pierwszych 20 znaków i dodaje wielokropek, dbając o estetykę interfejsu.

Przykład 2.

```kotlin
package com.example.jpc_nav2
import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import okhttp3.MediaType.Companion.toMediaType
import retrofit2.Retrofit
import retrofit2.converter.kotlinx.serialization.asConverterFactory
import retrofit2.http.Body
import retrofit2.http.GET
import retrofit2.http.PUT
import retrofit2.http.Path
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.unit.dp
import kotlinx.coroutines.launch

// 1. Model danych
@Serializable
data class Post(
    @SerialName("userId") val uzytkownikId: Int,
    @SerialName("id") val identyfikator: Int,
    @SerialName("title") val tytul: String,
    @SerialName("body") val tresc: String
)

// 2. Interfejs API
interface JsonPlaceholderApi {
    @GET("posts/{id}")
    suspend fun getPostById(@Path("id") id: Int): Post

    @GET("posts")
    suspend fun getAllPosts(): List<Post>

    @PUT("posts/{id}")
    suspend fun updatePost(@Path("id") id: Int, @Body post: Post): Post
}

// 3. Osobny OBIEKT dla Klienta (Singleton)
object ApiClient {
    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"
    private val jsonConfig = Json {
        ignoreUnknownKeys = true
    }
    private val contentType = "application/json".toMediaType()
    // Leniwa (lazy) inicjalizacja interfejsu API
    val api: JsonPlaceholderApi by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(jsonConfig.asConverterFactory(contentType))
            .build()
            .create(JsonPlaceholderApi::class.java)
    }
}

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

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

@Composable
fun JsonPlaceholderScreen() {
    var postIdInput by remember { mutableStateOf("") }
    var postsList by remember { mutableStateOf(listOf<Post>()) }
    var errorMessage by remember { mutableStateOf("") }
    val coroutineScope = rememberCoroutineScope()

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    ) {
        OutlinedTextField(
            value = postIdInput,
            onValueChange = { postIdInput = it },
            label = { Text("Wpisz ID posta (np. 1)") },
            keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
            modifier = Modifier.fillMaxWidth()
        )

        Spacer(modifier = Modifier.height(8.dp))

        // 1. GET - Pojedynczy post
        Button(
            onClick = {
                if (postIdInput.isBlank()) {
                    errorMessage = "Błąd: Wpisz najpierw ID!"
                    return@Button
                }
                coroutineScope.launch {
                    try {
                        errorMessage = ""
                        // Wywołanie z dedykowanego obiektu ApiClient
                        val post = ApiClient.api.getPostById(postIdInput.toInt())
                        postsList = listOf(post)
                    } catch (e: Exception) {
                        errorMessage = "Błąd GET: ${e.localizedMessage}"
                        postsList = emptyList()
                    }
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text("Pobierz jeden")
        }

        // 2. GET - Wszystkie posty
        Button(
            onClick = {
                coroutineScope.launch {
                    try {
                        errorMessage = ""
                        // Wywołanie z dedykowanego obiektu ApiClient
```

```
                    val posts = ApiClient.api.getAllPosts()
                    postsList = posts
                } catch (e: Exception) {
                    errorMessage = "Błąd GET (Wszystkie): ${e.localizedMessage}"
                    postsList = emptyList()
                }
            }
        },
        modifier = Modifier.fillMaxWidth()
    ) {
        Text("Pobierz wszystkie")
    }

    // 3. PUT - Aktualizacja posta
    Button(
        onClick = {
            val id = postIdInput.toIntOrNull() ?: 1
            coroutineScope.launch {
                try {
                    errorMessage = ""
                    val updatedPostData = Post(
                        uzytkownikId = 42,
                        identyfikator = id,
                        tytul = "Tytuł zmieniony przez PUT i własny obiekt klienta",
                        tresc = "Architektura kodu staje się lepsza!"
                    )
                    // Wywołanie z dedykowanego obiektu ApiClient
                    val response = ApiClient.api.updatePost(id, updatedPostData)
                    postsList = listOf(response)
                } catch (e: Exception) {
                    errorMessage = "Błąd PUT: ${e.localizedMessage}"
                    postsList = emptyList()
                }
            }
        },
        modifier = Modifier.fillMaxWidth()
    ) {
        Text("Test metody PUT")
    }

    Spacer(modifier = Modifier.height(16.dp))

    if (errorMessage.isNotEmpty()) {
        Text(
            text = errorMessage,
            color = MaterialTheme.colorScheme.error,
            modifier = Modifier.padding(bottom = 8.dp)
        )
    }

    // Lista kart z wynikami
    LazyColumn(
        modifier = Modifier.fillMaxWidth().weight(1f),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(postsList) { post ->
            PostItemCard(post = post)
        }
    }
  }
}

@Composable
fun PostItemCard(post: Post) {
    Card(
        modifier = Modifier.fillMaxWidth(),
```

```
        elevation = CardDefaults.cardElevation(defaultElevation = 4.dp)
    ) {
        Column(
            modifier = Modifier
                .padding(16.dp)
                .fillMaxWidth()
        ) {
            Text(
                text = "ID: ${post.identyfikator} (User ID: ${post.uzytkownikId})",
                style = MaterialTheme.typography.labelSmall,
                color = MaterialTheme.colorScheme.secondary
            )
            Spacer(modifier = Modifier.height(4.dp))
            Text(
                text = post.tytul,
                style = MaterialTheme.typography.titleMedium,
                color = MaterialTheme.colorScheme.primary
            )
            Spacer(modifier = Modifier.height(8.dp))
            Text(
                text = post.tresc,
                style = MaterialTheme.typography.bodyMedium
            )
        }
    }
}
```

Przykład 3. Wykorzystanie ViewModel

Przeniosłem listy danych, błędy oraz całą logikę sieciową (coroutineScope) do nowo utworzonej klasy JsonPlaceholderViewModel. Wykorzystuje ona bezpieczny viewModelScope, a stan interfejsu jest zarządzany przez mutableStateOf bezpośrednio w ViewModelu.

Pamiętaj, aby w pliku build.gradle.kts upewnić się, że masz dodaną zależność:

implementation("androidx.lifecycle:lifecycle-viewmodel-compose:2.8.7") (lub nowszą).

```
package com.example.jpc_nav2

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.compose.foundation.layout.*
import androidx.compose.foundation.lazy.LazyColumn
import androidx.compose.foundation.lazy.items
import androidx.compose.foundation.text.KeyboardOptions
import androidx.compose.material3.*
import androidx.compose.runtime.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.text.input.KeyboardType
import androidx.compose.ui.unit.dp
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.lifecycle.viewmodel.compose.viewModel
import kotlinx.coroutines.launch
import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable
import kotlinx.serialization.json.Json
import okhttp3.MediaType.Companion.toMediaType
import retrofit2.Retrofit
import retrofit2.converter.kotlinx.serialization.asConverterFactory
import retrofit2.http.Body
import retrofit2.http.GET
```

```
import retrofit2.http.PUT
import retrofit2.http.Path

// 1. Model danych
@Serializable
data class Post(
    @SerialName("userId") val uzytkownikId: Int,
    @SerialName("id") val identyfikator: Int,
    @SerialName("title") val tytul: String,
    @SerialName("body") val tresc: String
)

// 2. Interfejs API
interface JsonPlaceholderApi {
    @GET("posts/{id}")
    suspend fun getPostById(@Path("id") id: Int): Post

    @GET("posts")
    suspend fun getAllPosts(): List<Post>

    @PUT("posts/{id}")
    suspend fun updatePost(@Path("id") id: Int, @Body post: Post): Post
}

// 3. Osobny OBIEKT dla Klienta (Singleton)
object ApiClient {
    private const val BASE_URL = "https://jsonplaceholder.typicode.com/"
    private val jsonConfig = Json {
        ignoreUnknownKeys = true
    }
    private val contentType = "application/json".toMediaType()

    val api: JsonPlaceholderApi by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(jsonConfig.asConverterFactory(contentType))
            .build()
            .create(JsonPlaceholderApi::class.java)
    }
}

// 4. NOWOŚĆ: Klasa ViewModel zarządzająca stanem i logiką sieciową
class JsonPlaceholderViewModel : ViewModel() {

    // Pola stanu komponentów Compose wystawione jako "read-only" dla widoku
    var postsList by mutableStateOf(listOf<Post>())
        private set

    var errorMessage by mutableStateOf("")
        private set

    // GET - Pojedynczy post
    fun getPostById(idText: String) {
        if (idText.isBlank()) {
            errorMessage = "Błąd: Wpisz najpierw ID!"
            return
        }
        viewModelScope.launch {
            try {
                errorMessage = ""
                val post = ApiClient.api.getPostById(idText.toInt())
                postsList = listOf(post)
            } catch (e: Exception) {
                errorMessage = "Błąd GET: ${e.localizedMessage}"
                postsList = emptyList()
            }
```

```
        }
    }

    // GET - Wszystkie posty
    fun getAllPosts() {
        viewModelScope.launch {
            try {
                errorMessage = ""
                val posts = ApiClient.api.getAllPosts()
                postsList = posts
            } catch (e: Exception) {
                errorMessage = "Błąd GET (Wszystkie): ${e.localizedMessage}"
                postsList = emptyList()
            }
        }
    }

    // PUT - Aktualizacja posta
    fun updatePost(idText: String) {
        val id = idText.toIntOrNull() ?: 1
        viewModelScope.launch {
            try {
                errorMessage = ""
                val updatedPostData = Post(
                    uzytkownikId = 42,
                    identyfikator = id,
                    tytul = "Tytuł zmieniony przez PUT i własny obiekt klienta",
                    tresc = "Architektura kodu staje się lepsza!"
                )
                val response = ApiClient.api.updatePost(id, updatedPostData)
                postsList = listOf(response)
            } catch (e: Exception) {
                errorMessage = "Błąd PUT: ${e.localizedMessage}"
                postsList = emptyList()
            }
        }
    }
}

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    JsonPlaceholderScreen()
                }
            }
        }
    }
}

@Composable
fun JsonPlaceholderScreen(viewModel: JsonPlaceholderViewModel = viewModel()) {
    // Stan pola tekstowego zostaje lokalnie w widoku, bo dotyczy tylko wpisywania na tym
ekranie
    var postIdInput by remember { mutableStateOf("") }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    ) {
```

```
        OutlinedTextField(
            value = postIdInput,
            onValueChange = { postIdInput = it },
            label = { Text("Wpisz ID posta (np. 1)") },
            keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Number),
            modifier = Modifier.fillMaxWidth()
        )
        Spacer(modifier = Modifier.height(8.dp))

        // 1. GET - Pojedynczy post
        Button(
            onClick = { viewModel.getPostById(postIdInput) },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text("Pobierz jeden")
        }

        // 2. GET - Wszystkie posty
        Button(
            onClick = { viewModel.getAllPosts() },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text("Pobierz wszystkie")
        }

        // 3. PUT - Aktualizacja posta
        Button(
            onClick = { viewModel.updatePost(postIdInput) },
            modifier = Modifier.fillMaxWidth()
        ) {
            Text("Test metody PUT")
        }

        Spacer(modifier = Modifier.height(16.dp))

        // Odczyt błędu bezpośrednio z ViewModelu
        if (viewModel.errorMessage.isNotEmpty()) {
            Text(
                text = viewModel.errorMessage,
                color = MaterialTheme.colorScheme.error,
                modifier = Modifier.padding(bottom = 8.dp)
            )
        }

        // Lista kart z wynikami powiązana ze stanem z ViewModelu
        LazyColumn(
            modifier = Modifier.fillMaxWidth().weight(1f),
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            items(viewModel.postsList) { post ->
                PostItemCard(post = post)
            }
        }
    }
}

@Composable
fun PostItemCard(post: Post) {
    Card(
        modifier = Modifier.fillMaxWidth(),
        elevation = CardDefaults.cardElevation(defaultElevation = 4.dp)
    ) {
        Column(
            modifier = Modifier
                .padding(16.dp)
                .fillMaxWidth()
```

```
        ) {
            Text(
                text = "ID: ${post.identyfikator} (User ID: ${post.uzytkownikId})",
                style = MaterialTheme.typography.labelSmall,
                color = MaterialTheme.colorScheme.secondary
            )
            Spacer(modifier = Modifier.height(4.dp))
            Text(
                text = post.tytul,
                style = MaterialTheme.typography.titleMedium,
                color = MaterialTheme.colorScheme.primary
            )
            Spacer(modifier = Modifier.height(8.dp))
            Text(
                text = post.tresc,
                style = MaterialTheme.typography.bodyMedium
            )
        }
    }
}
```

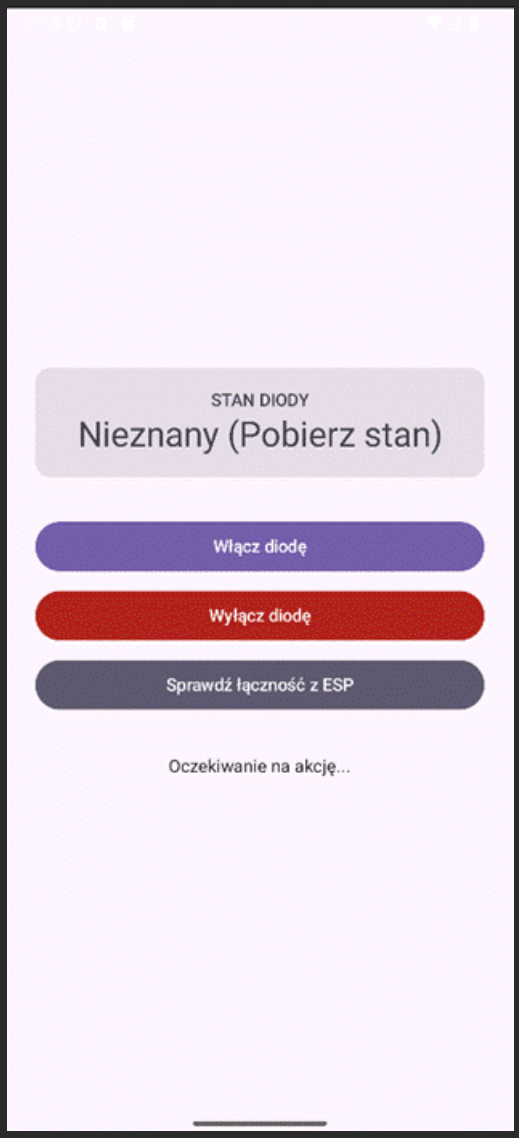
Przykład 4. Aplikacja komunikująca się z układem ESP8266

Program dla płytki:

```
#include <ESP8266WiFi.h>
#include <ESP8266WebServer.h>

const char* ssid = "PANS-KROSNO";
const char* password = "haslo";
const int pinOut = D5;

ESP8266WebServer server(80);

void handleGet() {
  server.send(200, "text/plain", "To jest GET!");
}
void handleGetPar() {

    String state = server.arg("state");

  if (state == "on") {
    digitalWrite(pinOut, HIGH);
    server.send(200, "text/plain", "1"); // sygnaĹ,: wĹ,Ä►czone
  } else if (state == "off") {
    digitalWrite(pinOut, LOW);
    server.send(200, "text/plain", "0"); // sygnaĹ,: wyĹ,Ä►czone
  } else {
    server.send(400, "text/plain", "Nieznany parametr: powinno być: state=on lub state=off");
  }

}
void handlePost() {
  String body = server.arg("state");
  server.send(200, "text/plain", "Odebrano POST: " + body);
}

void setup() {
  Serial.begin(115200);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) delay(500);
```

```
pinMode(pinOut, OUTPUT);
  digitalWrite(pinOut, LOW); // startowo wyĹ‚Ä…czone

  server.on("/get", HTTP_GET, handleGet);
  server.on("/getpar", HTTP_GET, handleGetPar);
  server.on("/post", HTTP_POST, handlePost);

  server.begin();
}

void loop() {
  server.handleClient();
}
```

Program na urządzenie mobilne – Android:



Wymagana zależność:

```
TA LINIA JEST KLUCZOWA - dedykowany konwerter dla String/typów prostych
// implementation("com.squareup.retrofit2:converter-scalars:2.11.0")
```

```
package com.example.jpc_nav2

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
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.unit.dp
import kotlinx.coroutines.launch
import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable
import retrofit2.Retrofit
import retrofit2.converter.scalars.ScalarsConverterFactory
import retrofit2.http.GET
import retrofit2.http.Query

// 1. Model danych z adnotacjami @Serializable i @SerialName
@Serializable
data class EspResponse(
    @SerialName("ledState") val stanDiody: String,
    @SerialName("message") val komunikat: String
)

// 2. Interfejs API (odbiera surowy String, aby nie wywalić programu na kodzie Arduino)
interface EspApi {
    @GET("getpar")
    suspend fun controlLed(@Query("state") state: String): String
}

// 3. Singleton dla Klienta - tutaj mapujemy tekst na nasz serylizowalny model danych
object EspClient {
    private const val BASE_URL = "http://192.168.1.122/"

    private val apiService: EspApi by lazy {
        Retrofit.Builder()
            .baseUrl(BASE_URL)
            .addConverterFactory(ScalarsConverterFactory.create())
            .build()
            .create(EspApi::class.java)
    }

    // Funkcja pośrednicząca, która zwraca ładny obiekt zgodny z Twoją nową strukturą
    suspend fun sendCommand(state: String): EspResponse {
        val rawResponse = apiService.controlLed(state)

        return when (rawResponse) {
            "1" -> EspResponse(stanDiody = "WŁĄCZONA", komunikat = "Dioda została pomyślnie
włączona!")
            "0" -> EspResponse(stanDiody = "WYŁĄCZONA", komunikat = "Dioda została pomyślnie
wyłączona!")
            else -> EspResponse(stanDiody = "NIEZNANY", komunikat = rawResponse) // np.
komunikat o błędzie z bloku 'else' w Arduino
        }
    }
}

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    EspControlScreen()
                }
            }
        }
    }
```

```
}

@Composable
fun EspControlScreen() {
    var ledStateText by remember { mutableStateOf("Nieznany (Pobierz stan)") }
    var statusMessage by remember { mutableStateOf("Oczekiwanie na akcję...") }

    val coroutineScope = rememberCoroutineScope()

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(24.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        // Nagłówek wizualny ze stanem diody
        Card(
            modifier = Modifier.fillMaxWidth().padding(bottom = 32.dp),
            colors = CardDefaults.cardColors(
                containerColor = if (ledStateText == "WŁĄCZONA")
                    MaterialTheme.colorScheme.primaryContainer
                else MaterialTheme.colorScheme.surfaceVariant
            )
        ) {
            Column(
                modifier = Modifier.padding(16.dp).fillMaxWidth(),
                horizontalAlignment = Alignment.CenterHorizontally
            ) {
                Text(text = "STAN DIODY", style = MaterialTheme.typography.labelLarge)
                Text(
                    text = ledStateText,
                    style = MaterialTheme.typography.headlineMedium,
                    color = if (ledStateText == "WŁĄCZONA")
                        MaterialTheme.colorScheme.primary
                    else MaterialTheme.colorScheme.onSurfaceVariant
                )
            }
        }

        // PRZYCISK 1: Włącz diodę (?state=on)
        Button(
            onClick = {
                coroutineScope.launch {
                    try {
                        statusMessage = "Wysyłanie sygnału: ON..."
                        // Wywołanie korzystające z nowego modelu danych EspResponse
                        val response: EspResponse = EspClient.sendCommand("on")

                        ledStateText = response.stanDiody
                        statusMessage = response.komunikat
                    } catch (e: Exception) {
                        statusMessage = "Błąd połączenia: ${e.localizedMessage}"
                    }
                }
            },
            modifier = Modifier.fillMaxWidth().padding(bottom = 8.dp)
        ) {
            Text("Włącz diodę")
        }

        // PRZYCISK 2: Wyłącz diodę (?state=off)
        Button(
            onClick = {
                coroutineScope.launch {
                    try {
```

```
                        statusMessage = "Wysyłanie sygnału: OFF..."
                        // Wywołanie korzystające z nowego modelu danych EspResponse
                        val response: EspResponse = EspClient.sendCommand("off")

                        ledStateText = response.stanDiody
                        statusMessage = response.komunikat
                    } catch (e: Exception) {
                        statusMessage = "Błąd połączenia: ${e.localizedMessage}"
                    }
                }
            },
            colors = ButtonDefaults.buttonColors(containerColor =
MaterialTheme.colorScheme.error),
            modifier = Modifier.fillMaxWidth().padding(bottom = 8.dp)
        ) {
            Text("Wyłącz diodę")
        }

        // PRZYCISK 3: Sprawdź stan
        Button(
            onClick = {
                coroutineScope.launch {
                    try {
                        statusMessage = "Sprawdzanie połączenia..."
                        // Wywołanie korzystające z nowego modelu danych EspResponse
                        val response: EspResponse = EspClient.sendCommand("status")

                        statusMessage = "Serwer ESP odpowiada: ${response.komunikat}"
                    } catch (e: Exception) {
                        statusMessage = "Błąd: Brak odpowiedzi z ESP (${e.localizedMessage})"
                    }
                }
            },
            colors = ButtonDefaults.buttonColors(containerColor =
MaterialTheme.colorScheme.secondary),
            modifier = Modifier.fillMaxWidth().padding(bottom = 32.dp)
        ) {
            Text("Sprawdź łączność z ESP")
        }

        // Log tekstowy na dole ekranu informujący o operacjach
        Text(
            text = statusMessage,
            style = MaterialTheme.typography.bodyMedium,
            color = MaterialTheme.colorScheme.onSurface
        )
    }
}
```