Systemy mobilne: Android (część 3)

Nawigacja oraz obsługa bazy danych

Dr inż. Szymon Sieciński, Dr hab. inż. Leszek Siwik

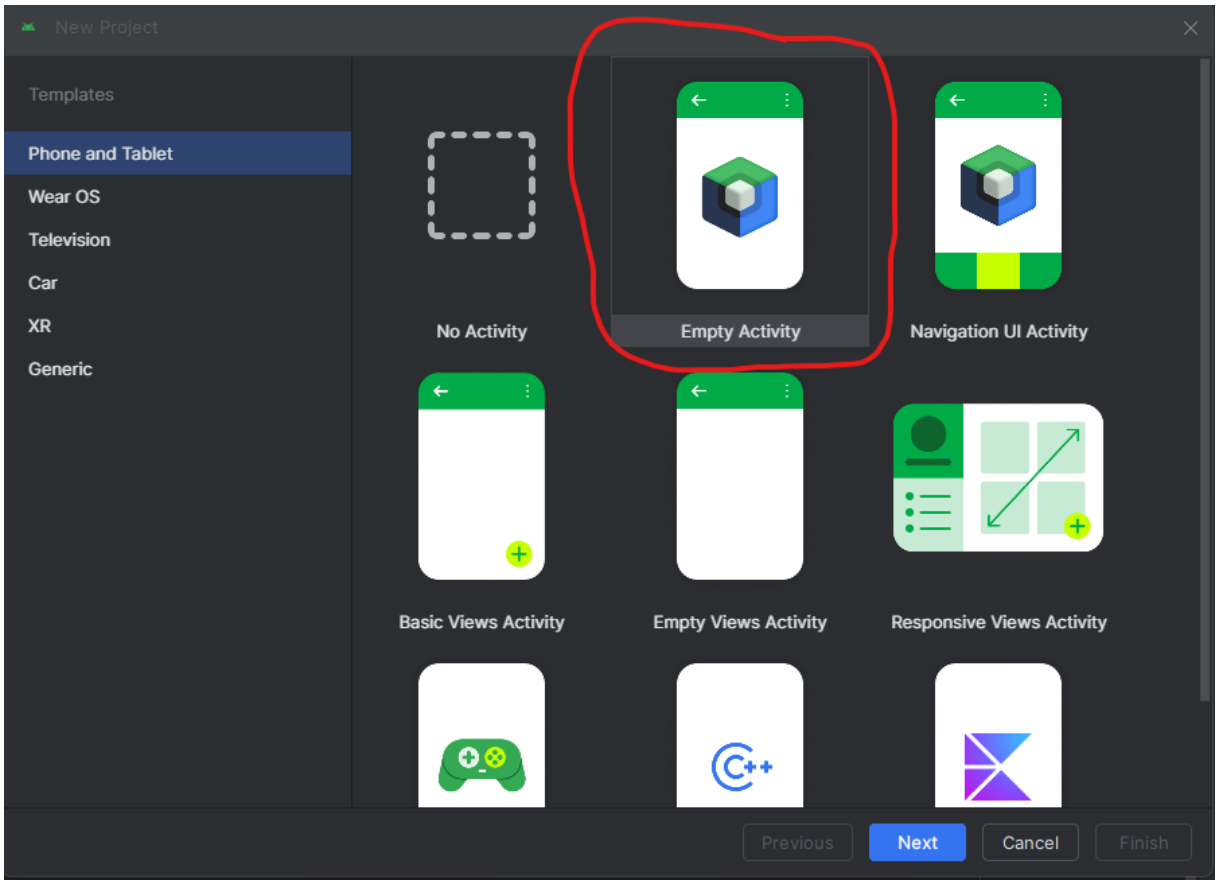
Wydział Informatyki AGH

Kraków, 2.11.2025 r.

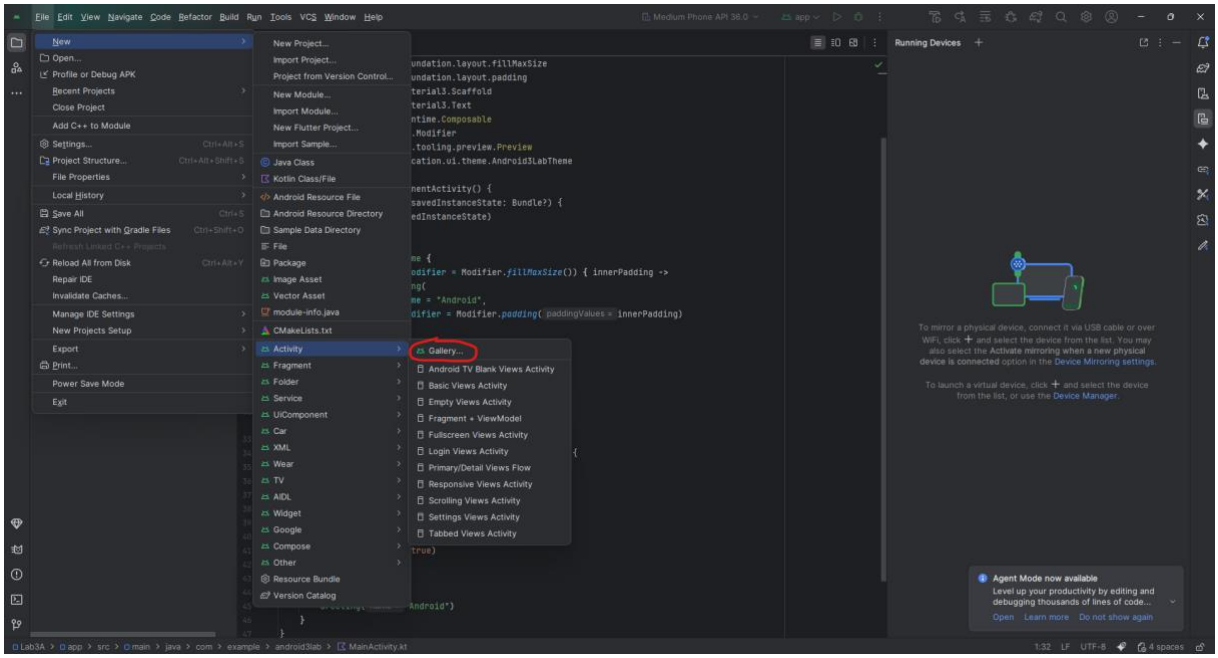
Nawigacja (część A)

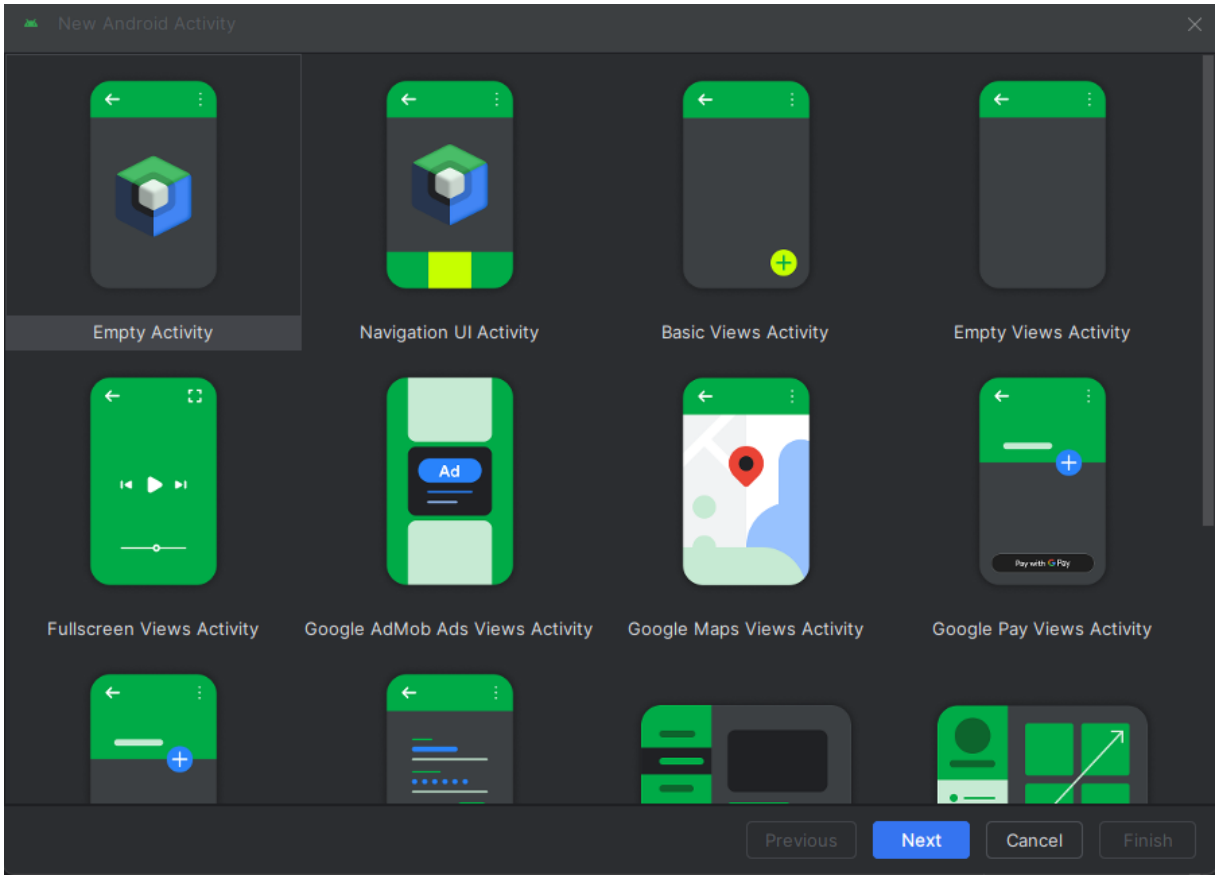
Uruchamianie aktywności

Utwórz nowy pusty projekt (Empty Activity)

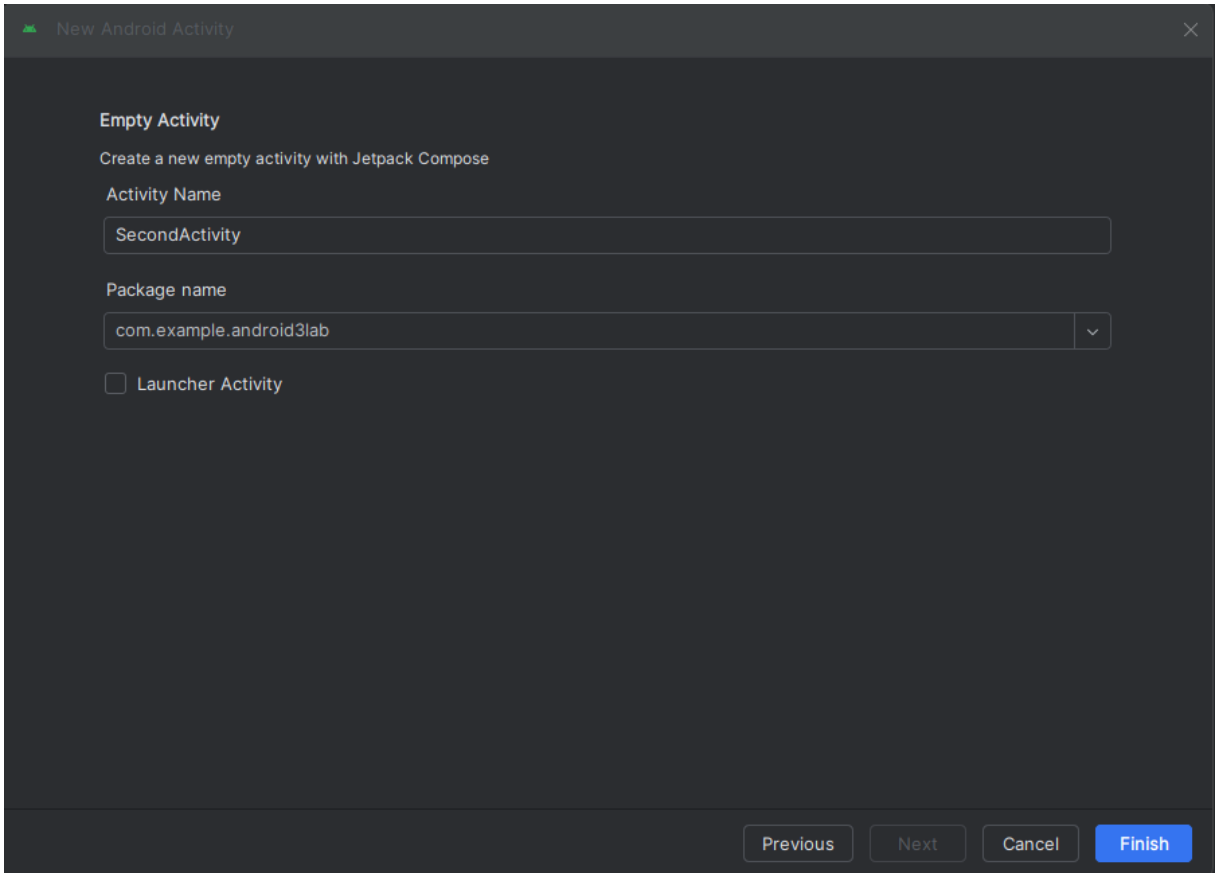


Zaczekaj na wygenerowanie podstawowych plików, a następnie dodaj drugą pustą aktywność przez wybór z galerii:





Nadaj jej nazwę, pozostaw ten sam pakiet i naciśnij „Finish”.



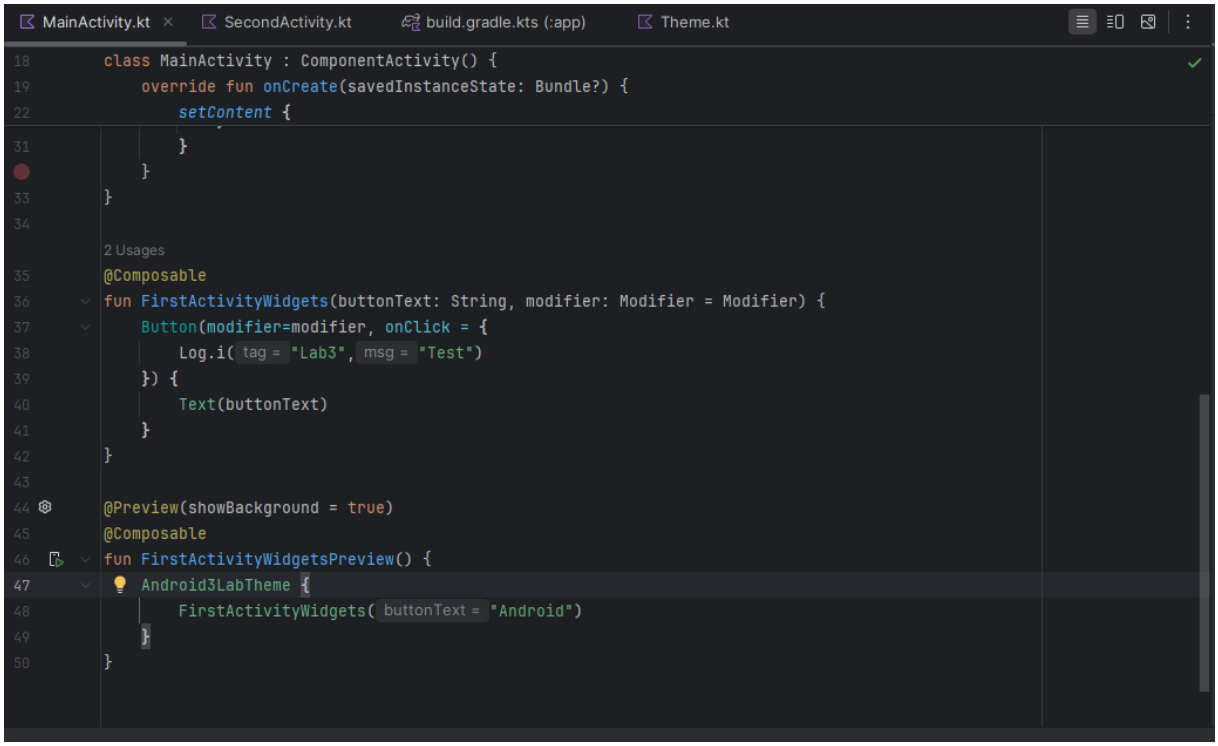
W pierwszej wersji usuń dotychczasową zawartość domyślnej funkcji Greeting i dodaj przycisk jak w pierwszym ćwiczeniu przez funkcję Button:

```
Button(modifier=modifier, onClick = {
    Log.i("Lab3","Test")
}) {
    Text(buttonText)
}
```

Można również zmienić nazwę funkcji Greeting na bardziej adekwatną, np.

```
@Composable
fun FirstActivityWidgets(buttonText: String, modifier:
Modifier = Modifier) {
    Button(modifier=modifier, onClick = {
        Log.i("Lab3","Test")
    }) {
        Text(buttonText)
    }
}
```

Co przedstawia poniższy fragment w Android Studio



Aby uruchomić nową aktywność, należy w funkcji `onClick` (lub wyżej) zdefiniować kontekst poleceniem

```
val context = LocalContext.current
```

Następnie trzeba stworzyć intencję (ang. Intent), najlepiej w funkcji `onClick`

```
val intent = Intent(context, SecondActivity::class.java)
```

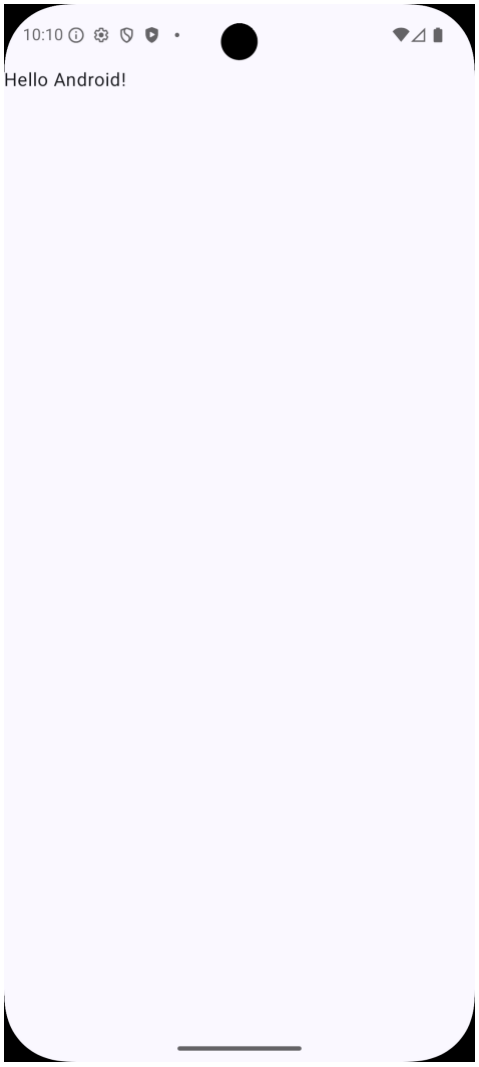
Oraz uruchomić nową aktywność poleceniem

```
context.startActivity(intent)
```

Zapis `Klasa::class.java` oznacza przekazanie instancji klasy „Klasa”

Dzięki temu mechanizmowi możemy uruchamiać inną aktywność przyciskiem z poziomu pierwszej (głównej) aktywności

W drugiej aktywności pozostawiamy domyślny kod i wygląd.



Przekazywanie danych przez intencje

Intencje pozwalają również na przekazywanie danych między aktywnościami. W tym celu stworzymy układ liniowy (Column) i dodamy pole tekstowe (TextField) do przekazywania komunikatu.

W polu tekstowym dodajemy do pola `value` zmienną przechowującą bieżący tekst (przykładowa nazwa `textForIntent`) typu `mutableState<String>` , implementujemy funkcję onValueChange jako

```
onValueChange = {
    textForIntent.value = it
}
```

oraz etykietę sugerująca przeznaczenie.

Odnajdujemy linię odpowiedzialną za tworzenie intencji `Intent(<args>)` i dodajemy metodę `.putExtra("nazwa", textForIntent.value)`, aby przekazać napis do nowej aktywności.

W drugiej aktywności dodajemy do funkcji onCreate linię

```
val forwardedText = intent.getStringExtra("nazwa") ?:
"Android"
```

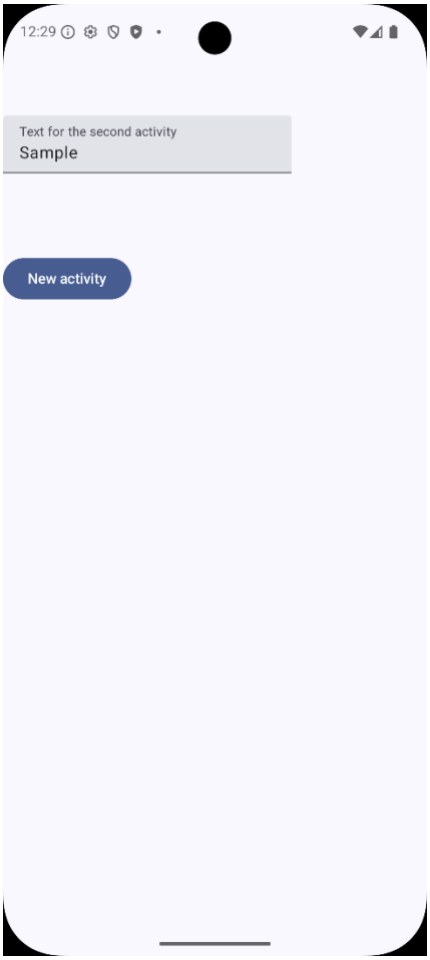
Ta linia pozwala na odczytanie dodatkowych parametrów przekazanych w intencji i pokazanie zawartości pola tekstowego. Zapis `warunek ? then : else` pozwala na szybkie przypisanie wartości zależnej od warunku przed znakiem zapytania (?), aby zachować zgodność typów z pierwotną funkcją do wyświetlania napisu. Można również użyć innych funkcji do tworzenia elementów interfejsu.

Do drugiej aktywności dodajemy układ liniowy (Column) i dodajemy przycisk pozwalający na powrót do pierwszej aktywności (poniżej nazwa MainActivity).

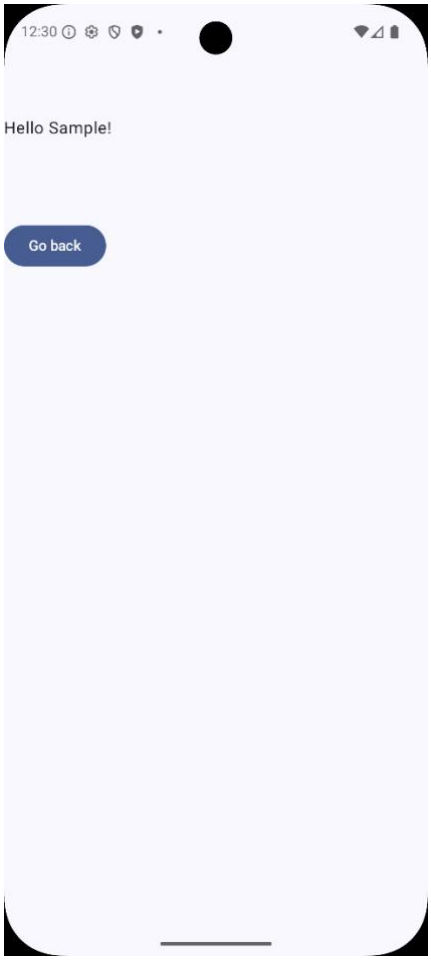
W przycisku dodajemy przejście do aktywności podobnie jak w pierwszej aktywności:

```
Button(modifier=modifier,
    onClick = {
        val context = LocalContext.current
        val intent = Intent(context, MainActivity::class.java)
        context.startActivity(intent)
    },
) {
    Text("Go back")
}
```

Pierwsza aktywność powinna przypominać to:



Druga aktywność powinna przypominać to:

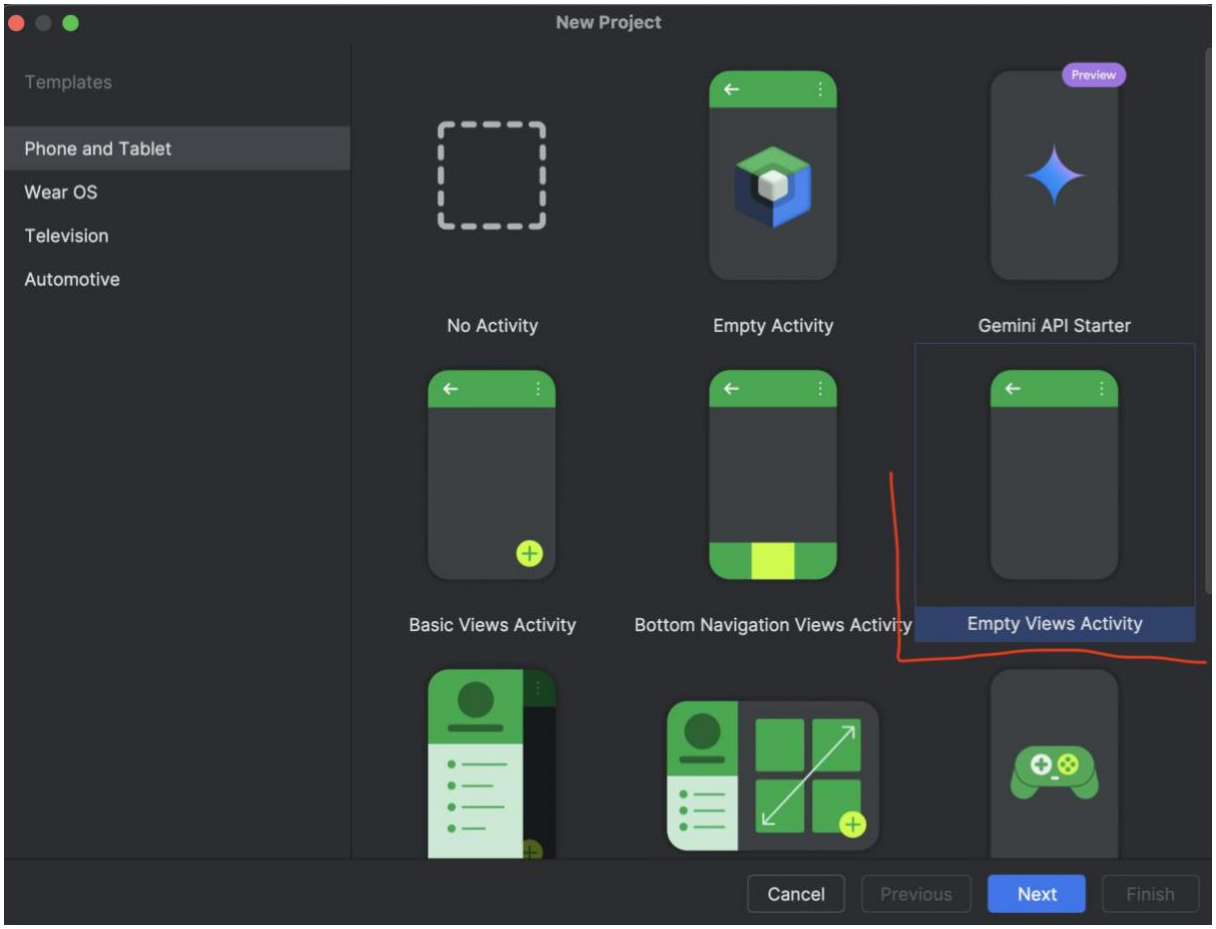


Fragmenty i Navigation Compose

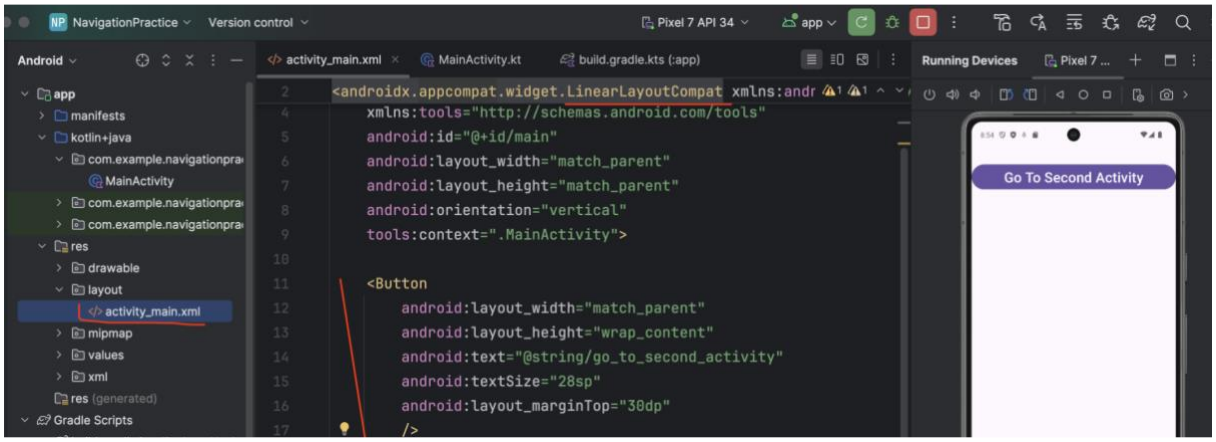
Fragmenty

Fragmenty pozwalają na definiowane różnych elementów interfejsu.

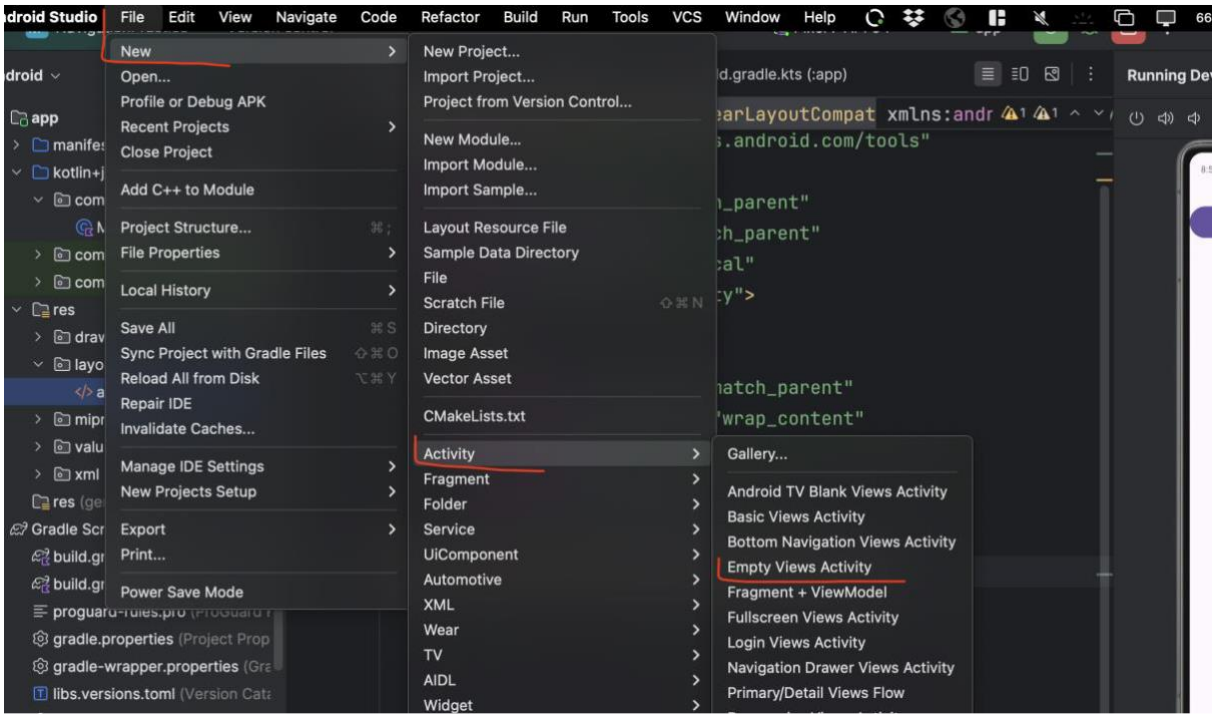
Pierwszym krokiem jest stworzenie pustego projektu z pustymi widokami:



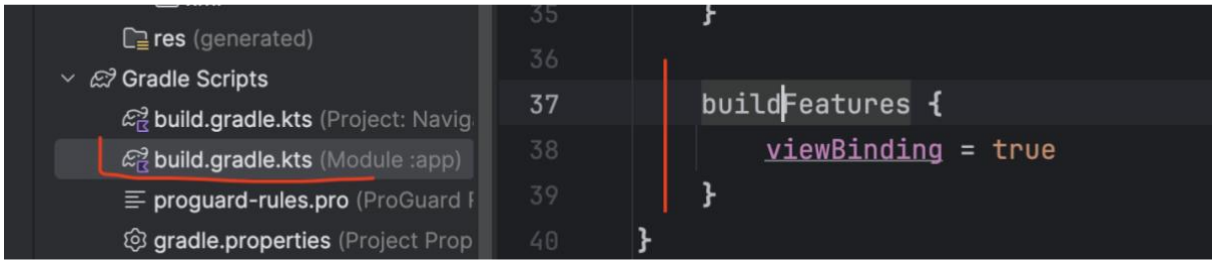
Następnie zmieniamy layout w `activity_main.xml` na LinearLayoutCompat i dodajemy przycisk.



Dodajemy nową aktywność tego samego typu jak pierwszą:



Włączamy viewBinding w pliku `build.gradle.kts`, aby włączyć automatyczne wiązanie elementów widoków z kodem.

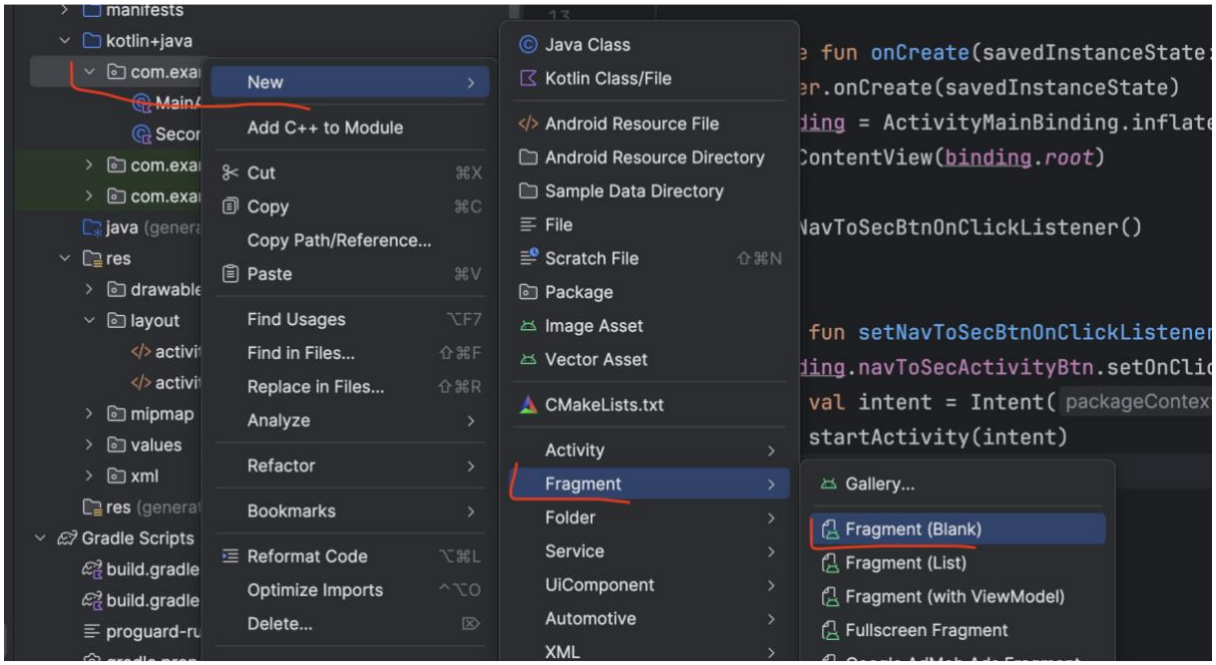


Dodajemy view binding do głównej aktywności:

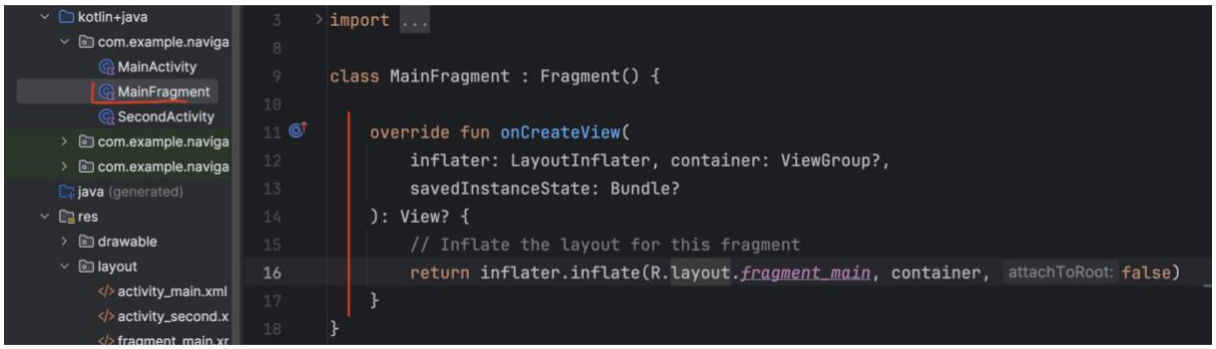
```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

Dodajemy do przycisku `onClickListenter` (funkcję do obsługi kliknięcia) bez zawartości. Zawartość uzupełnimy później.

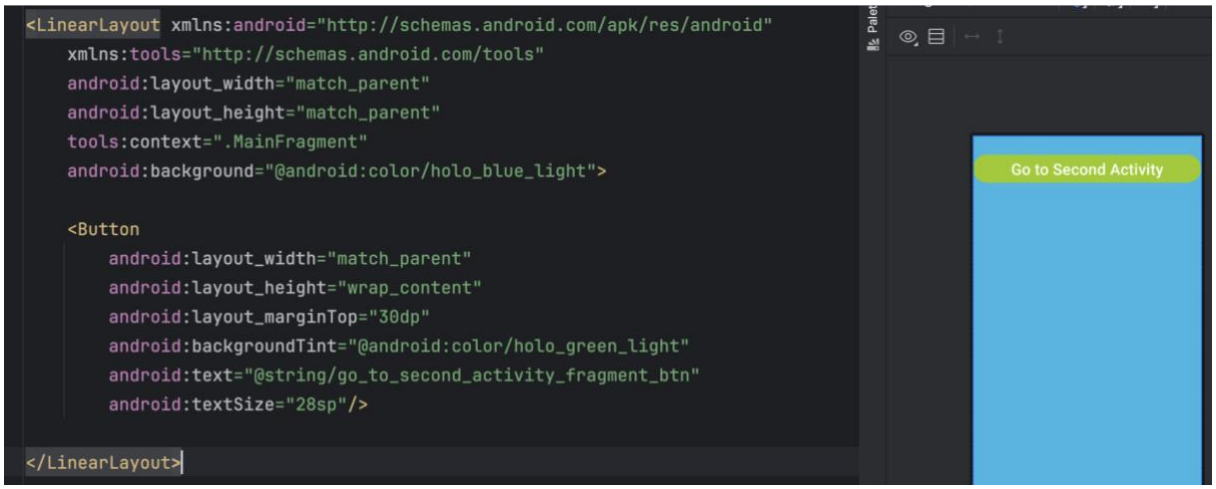
Następnie dodajemy pierwszy fragment i nazywamy go `MainFragment`



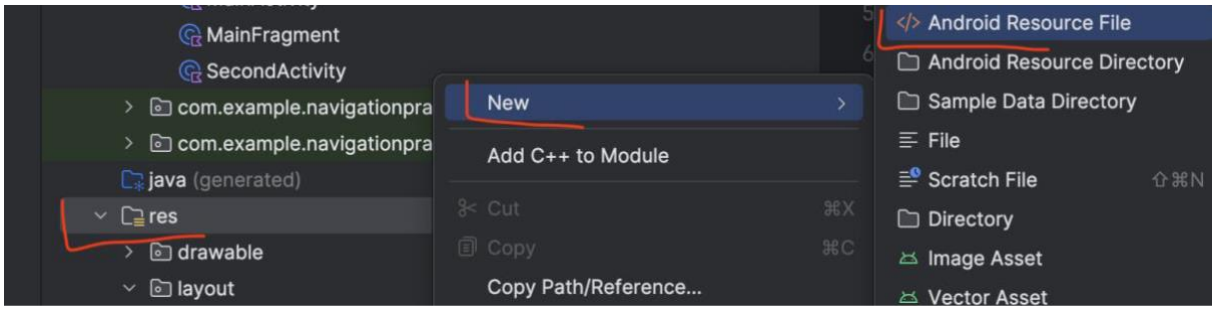
W nowym fragmencie usuwamy wszystkie metody oprócz onCreateView.



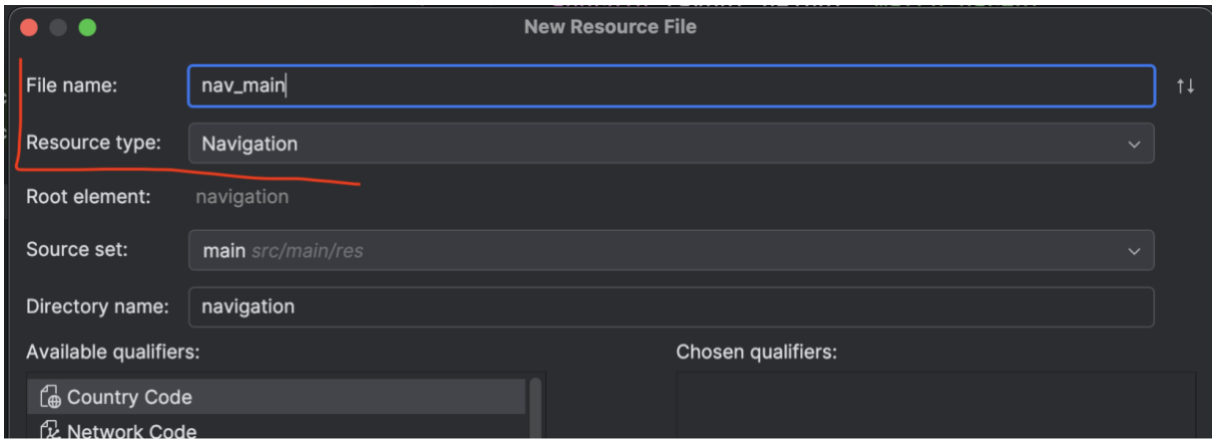
Zmieniamy layout na Linear, dodajemy przycisk i zmieniamy tło.



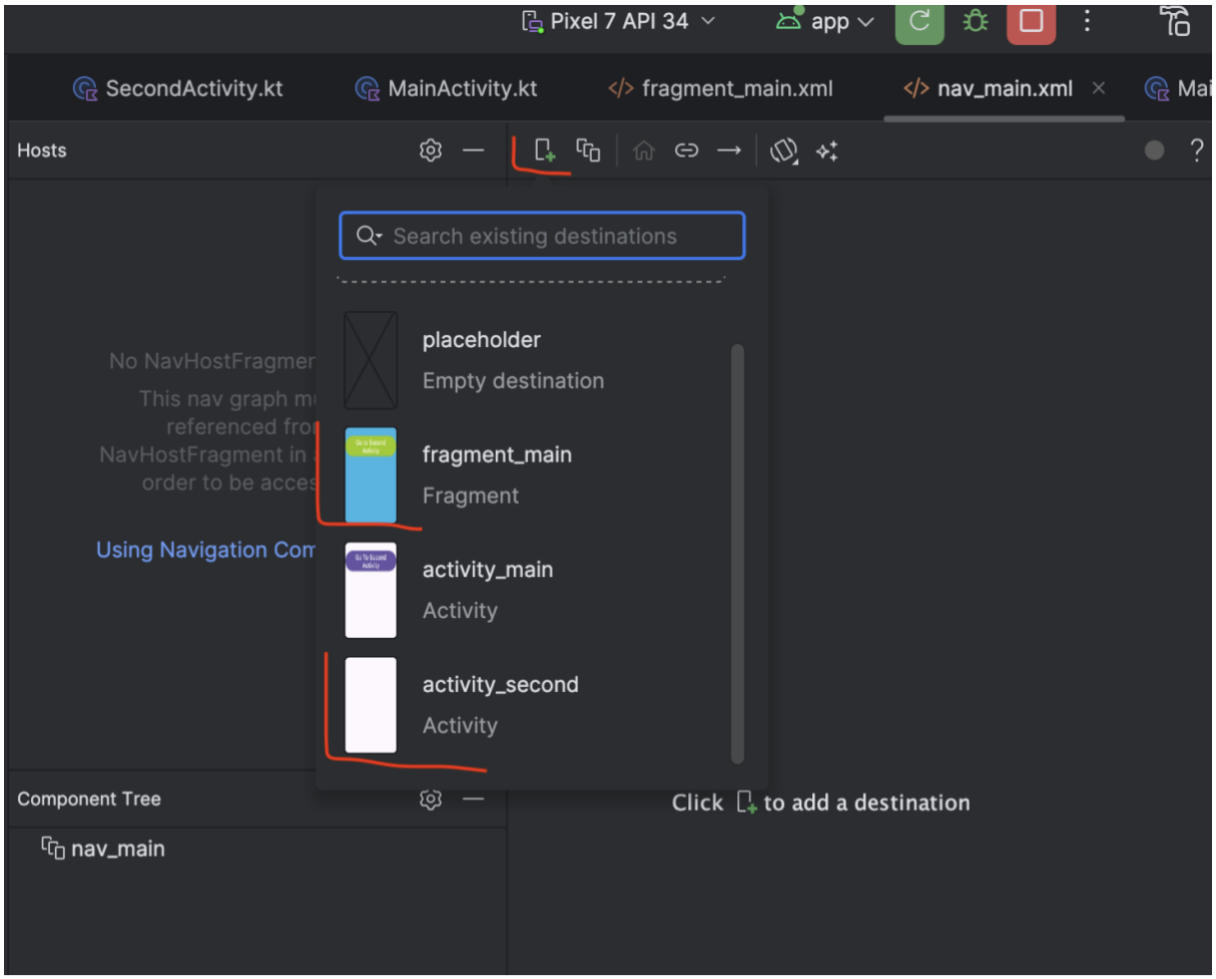
Dodajemy fragment do zasobów projektu przez new → Android Resource File



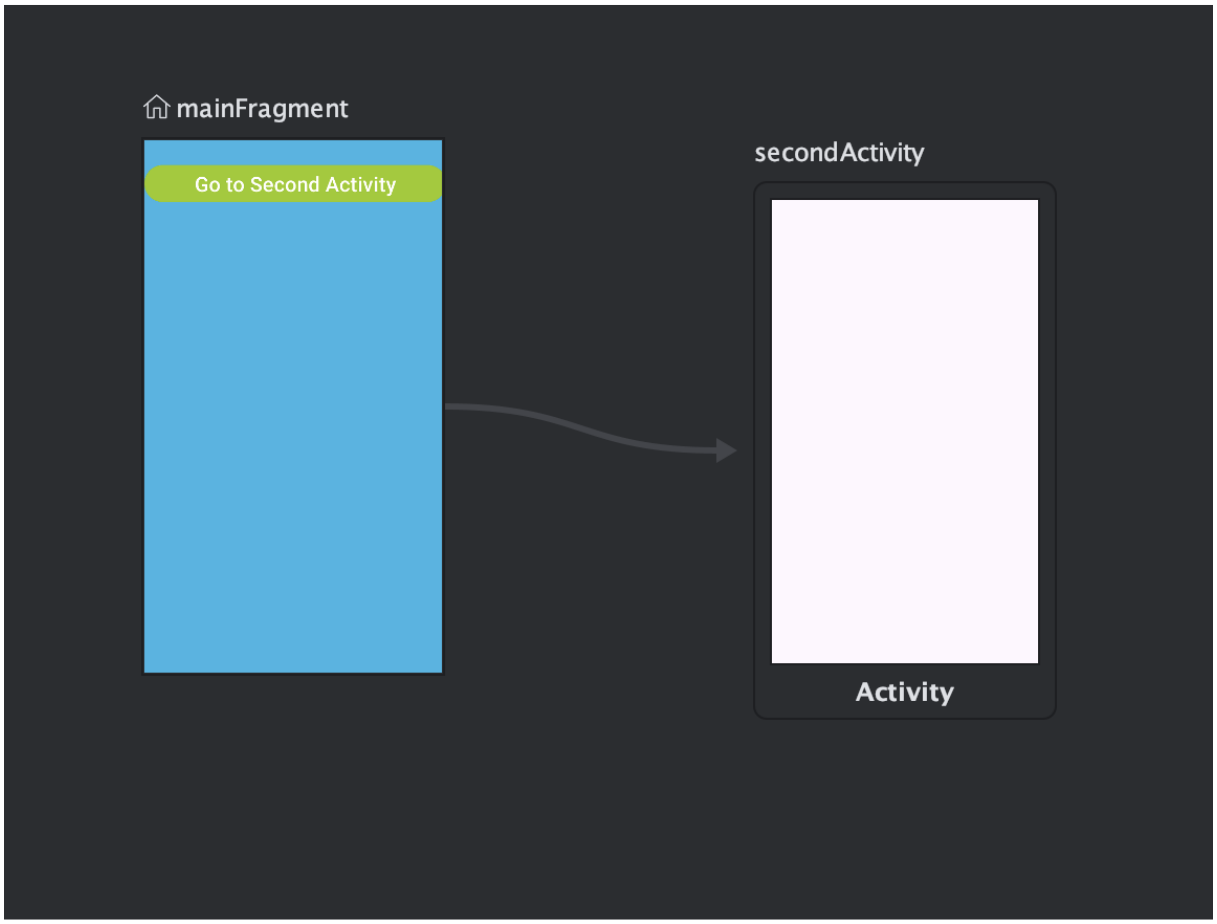
W oknie dialogowym zmieniamy typ zasobu na Navigation i nazwyamy go nav_main.



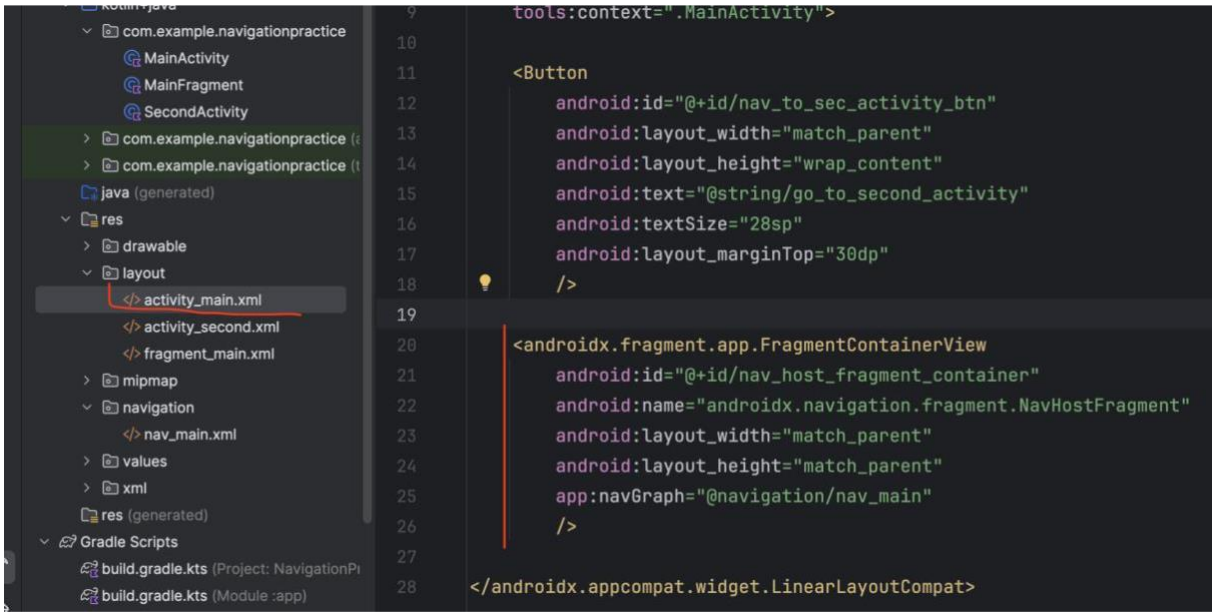
W razie potrzeby importujemy brakujące zasoby i dodajemy nowy fragment do grafu nawigacji:



Dodajemy przejścia między `mainFragment` i drugą aktywnością jak poniżej:



Dodajemy nowy fragment do layoutu głównej (pierwszej) aktywności:



Ostatnią rzeczą jest zaimplementowanie akcji przejścia do nowego fragmentu.

Pierwszym krokiem jest dodanie view binding do klasy głównego fragmentu (MainFragment) jak poprzednio:

```
class MainFragment : Fragment() {
    private lateinit var binding: FragmentMainBinding
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = FragmentMainBinding.inflate(inflater,container, attachToParent: false)
        return binding.root
    }
}
```

Dodajemy id przycisku, np. `nav_to_sec_activity_from_fragment_btn` oraz nadpisujemy metodę `onViewCreated`.

```
class MainFragment : Fragment() {
    private lateinit var binding: FragmentMainBinding
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        binding = FragmentMainBinding.inflate(inflater,container, attachToParent: false)
        return binding.root
    }

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)
        binding.navToSecActivityFromFragmentBtn.setOnClickListener {
            findNavController().navigate(R.id.)
        }
    }
}
```

```
main                                        Int
nav_main                                    Int
mainFragment                                Int
action_mainFragment_to_secondActivity       Int
secondActivity                              Int
```

Następnie definiujemy funkcję przypisaną do `onClickListener` (metoda `setOnClickListener`) w klasie `MainFragment` podobnie jak poniżej (przez uruchamianie aktywności i przekazywanie danych przez intencję):

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    setNavToSecBtnOnClickListener()
}

private fun setNavToSecBtnOnClickListener() {
    binding.navToSecActivityBtn.setOnClickListener {
        val intent = Intent( packageContext: this, SecondActivity::class.java)
        startActivity(intent)
    }
}
```

Teraz możemy przechodzić między fragmentami i aktywnościami.

Navigation Compose

Jetpack Compose jest deklaratywnym frameworkiem tworzenia interfejsu użytkownika aplikacji na platformę Android opartym na języku Kotlin rozwijanym od 2019 roku i publicznie dostępnym od 2021 roku.

Na laboratorium nr 1 już wykorzystywano elementy interfejsu tworzone za pomocą Jetpack Compose. Tym razem skupimy się na nawigacji.

Pierwszym krokiem jest utworzenie pustego projektu jak na początku tego ćwiczenia.

Drugim krokiem jest sprawdzenie, czy włączono Compose w `build.gradle.kts` modułu app. W build features musi być ustawione `compose = true` i powinny być dodane zależności **Compose BOM** oraz `navigation-compose`, `material3`, `activity-compose`.

Następnie definiujemy w klasie aktywności szkielet nawigacji w obiekcie Routes:

```
private object Routes {
    const val A = "screen_a"
    const val B = "screen_b"
}
```

Tworzymy funkcję z adnotacją Composable odpowiedzialną za nawigację:

```
@Composable
fun AppNav() {
    val navController = rememberNavController()
```

```
    NavHost(
        navController = navController,
        startDestination = Routes.screenA
    ) {
        composable(Routes.screenA) { ScreenA(navController) }
        composable(Routes.screenB) { ScreenB(navController) }
    }
}
```

W tej funkcji tworzymy zmienną navController (kontroler nawigacji) oraz wywołujemy funkcję NavHost, w której rejestrujemy ekrany przez polecenie

`composable(Routes.A) { Fun1(navController) }`

Rolą NavHost jest definiowanie grafu nawigacji wewnątrz bloku lambda (funkcji anonimowej), wyświetlanie aktualnego ekranu, określenie ekranu startowego (parametr startDestination), przechodzenie do innych ekranów za pomocą metody `navigate("trasa")`.

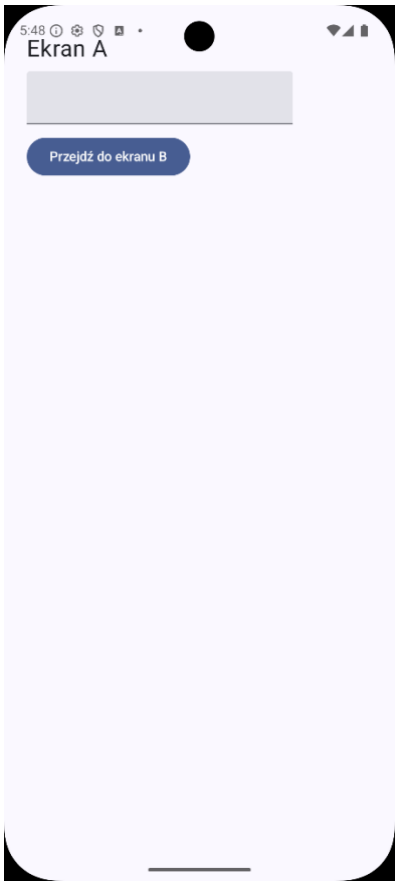
Kolejnym etapem jest stworzenie dwóch funkcji z adnotacją `Composable`, w których tworzymy odpowiednie widoki. Roboczo nazywamy te ekrany ScreenA oraz ScreenB.

Przykładowa funkcja dla ScreenA wygląda następująco:

```
@Composable
fun ScreenA(nav: NavController) {
    val textForScreenB: MutableState<String> = remember { mutableStateOf("") }

    Surface(Modifier.fillMaxSize()) {
        Column(
            modifier = Modifier.fillMaxSize().padding(24.dp),
            horizontalAlignment = Alignment.Start,
            verticalArrangement = Arrangement.Top
        ) {
            Spacer(Modifier.height(8.dp))
            Text("Ekran A", fontSize = 24.sp)
            Spacer(Modifier.height(10.dp))
            TextField(
                value = textForScreenB.value,
                onValueChange = { textForScreenB.value = it }
```

```
            )
            Spacer(Modifier.height(10.dp))
            Button(onClick = { nav.navigate(Routes.screenB) })
{
                Text("Przejdź do ekranu B")
            }
        }
    }
}
```

I daje poniższy ekran:

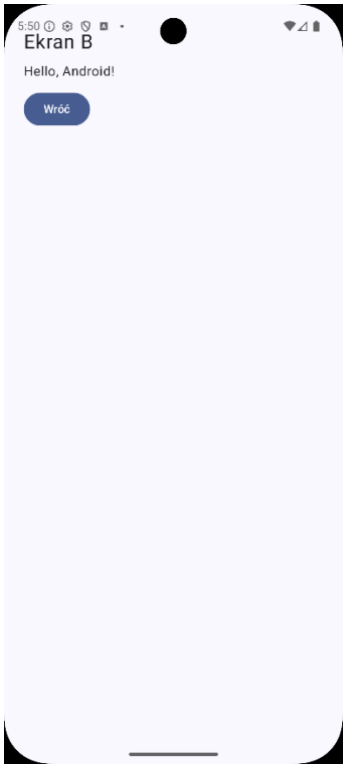


Przykładowa funkcja dla ScreenB wygląda tak:

```
@Composable
fun ScreenB(nav: NavController) {
    Surface(Modifier.fillMaxSize()) {
        Column(
            modifier = Modifier.fillMaxSize().padding(24.dp),
            horizontalAlignment = Alignment.Start,
            verticalArrangement = Arrangement.Top
        ) {
```

```
            Spacer(Modifier.height(8.dp))
            Text("Ekran B", fontSize = 24.sp)
            Spacer(Modifier.height(10.dp))
            Text("Hello, Android!")
            Spacer(Modifier.height(10.dp))
            Button(onClick = { nav.popBackStack() }) {
                Text("Wróć")
            }
        }
    }
}
```

I daje poniższy wynik:



Aby przekazać dane z jednego ekranu do drugiego, należy zmienić nawigację oraz widoki następująco:

1. Dodajemy trasę z argumentem „text” typu String dla ScreenB

```
composable(route = "${Routes.screenB}/{text}",
    arguments = listOf(
        navArgument("text") {
```

```
                    type = NavType.StringType
                    nullable = false
                })
            ) { bstkEntry ->
                val text =
bstkEntry.arguments?.getString("text").orEmpty()
                ScreenB(navController, text=text)
            }
```

Dodajemy w bloku lambda argument, który przechowuje wywołane argumenty i strzałkę. Za strzałką tworzymy zmienną przechwytującą argument do nowego ekranu oraz zmienione wywołanie.

2. Do nagłówka funkcji ScreenB dodajemy parametr „text” typu String i zmieniamy zawartość tekstu do pokazania na "Hello, $text!”:

```
fun ScreenB(nav: NavController, text: String = "Android")
Text("Hello, $text!")
```

Cała funkcja:

```
@Composable
fun ScreenB(nav: NavController, text: String = "Android")
{
    Surface(Modifier.fillMaxSize()) {
        Column(
            modifier =
Modifier.fillMaxSize().padding(24.dp),
            horizontalAlignment = Alignment.Start,
            verticalArrangement = Arrangement.Top
        ) {
            Spacer(Modifier.height(8.dp))
            Text("Ekran B", fontSize = 24.sp)
            Spacer(Modifier.height(10.dp))
```

```
                Text("Hello, $text!")
                Spacer(Modifier.height(10.dp))
                Button(onClick = { nav.popBackStack() }) {
                    Text("Wróć")
                }
            }
        }
    }
```

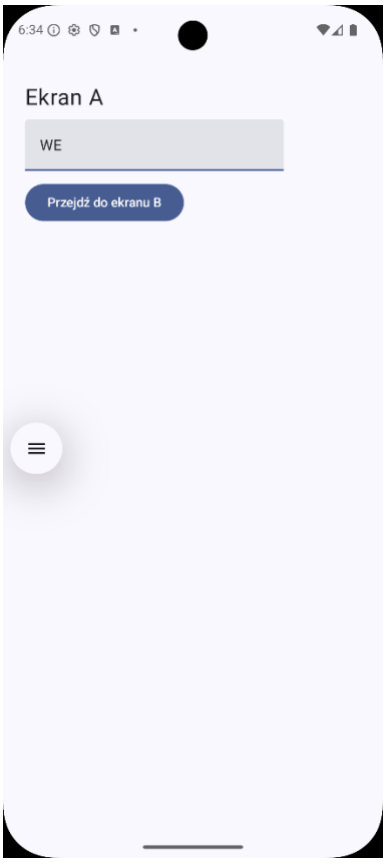
3. W ekranie ScreenA zmieniamy przejście do ekranu ScreenB przez dodanie nawigacji z zakodowanym argumentem text:

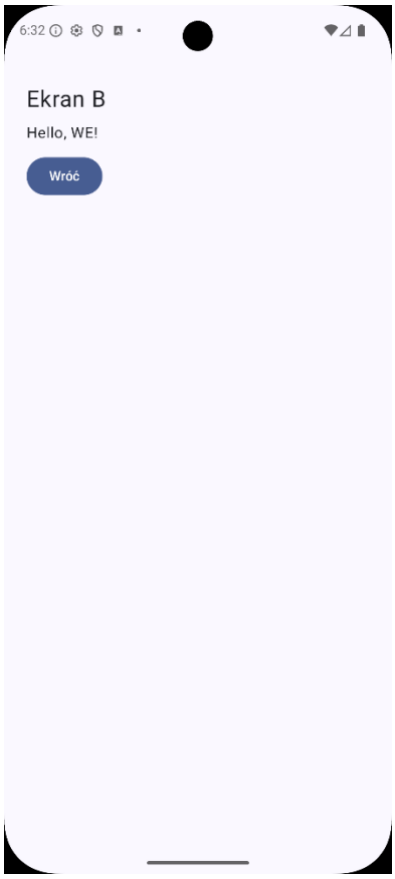
   ```
   val encodedText = Uri.encode(textForScreenB.value.ifBlank { "Android" })
   nav.navigate("${Routes.screenB}/$encodedText")
   ```

   Pierwszym elementem jest zmiana parametru metody `navigate` na nową ścieżkę `{Routes.screenB}/$encodedText`, gdzie `encodedText` jest naszym argumentem. Dla bezpieczeństwa przekazywania danych musimy zakodować nasze dane funkcją `Uri.encode()` z pakietu `android.net.Uri`, aby ścieżka nawigacji była poprawna.

Po wykonaniu poniższych kroków powinniśmy mieć podobny kod funkcji ScreenA do poniższego:

Daje nam to poniższy widok A oraz B:





Zadania do wykonania

1. Stwórz aplikację z dwiema aktywnościami:
    - Pierwsza aktywność jest aktywnością startową.
      W niej musi znaleźć się pole tekstowe z tekstem użytkownika przekazywanym do drugiej aktywności przez mechanizm intencji oraz przycisk pozwalający na przejście do drugiej aktywności.
    - W drugiej aktywności wyświetlany jest tekst przekazany przez pole tekstowe w pierwszej aktywności. Pod tekstem musi znaleźć się przycisk pozwalający na powrót do pierwszej aktywności.
2. Stwórz aplikację z nawigacją opartą o fragmenty. Fragmenty są takie jak dla pierwszego zadania.
3. Stwórz aplikację z nawigacją opartą o Navigation Component:
    - Oba ekrany są zaimplementowane w głównej aktywności.
    - Pierwszy ekran zawiera pole tekstowe z tekstem użytkownika przekazywanym do drugiego za pomocą przycisku.
    - W drugim ekranie wyświetlany jest tekst przekazany przez pole tekstowe z pierwszego ekranu. Pod tekstem musi znaleźć się przycisk pozwalający na powrót do pierwszego ekranu.

We wszystkich zadaniach prześlij kod i zrzuty ekranowe (lub screencast – krótkie video) przedstawiające działanie aplikacji w etapie końcowym.

Obsługa bazy danych

Wiele aplikacji wykorzystuje lokalną bazę danych. Aby włączyć możliwość wykorzystywania bazy danych SQLite, należy wykonać następujące zmiany w plikach konfiguracyjnych projektu:

build.gradle.kts (project):

w obiekcie plugins dopisujemy:

```
id("com.google.devtools.ksp") version "2.0.21-1.0.27" apply false
```

build.gradle.kts (:app):

w obiekcie plugins dopisujemy:

```
id("com.google.devtools.ksp")
```

w obiekcie dependencies dopisujemy:

```
val room_version = "2.8.3"
implementation("androidx.room:room-runtime:$room_version")
ksp("androidx.room:room-compiler:$room_version")
implementation("androidx.room:room-ktx:${room_version}")
implementation("androidx.room:room-rxjava2:${room_version}")
implementation("androidx.room:room-rxjava3:${room_version}")
implementation("androidx.room:room-guava:${room_version}")
implementation("androidx.room:room-paging:$room_version")
testImplementation("androidx.room:room-testing:$room_version")
```

Po dopisaniu zmian synchronizujemy projekt i czekamy na zastosowanie zmian. Od tej pory możemy przejść do kolejnej części – implementacji architektury projektu kalkulatora BMI przechowującego wcześniejsze wyniki:

- Model
    - BmiMeasurement – podstawowa klasa przechowująca dane o obliczeniu BMI: wzrost [m], masa ciała [kg], BMI, kategoria BMI oraz data i godzina zapisana jako Unix Timestamp
    - BmiMeasurementDao – klasa łącząca dane z zapytaniami do bazy danych
    - AppDatabase – klasa singleton odpowiedzialna za połączenie z bazą danych
    - BmiMeasurementRepository – klasa odpowiedzialna za logikę danych
- Kontroler:
    - BmiMeasurementViewModel – klasa obliczająca BMI i przekazująca dane do widoków
- Nawigacja: Dwa widoki:
    - Ekran główny
    - Lista pomiarów
- Widok
    - Ekran główny (MainScreen)
    - Ekran listy pomiarów (ListScreen)

Model danych

Model danych w aplikacji jest implementowany w klasie `BmiMeasurement` oraz `BmiMeasurementDao`. Pierwsza klasa zawiera wyłącznie pola przechowujące dane i zawiera ich podstawowy model, natomiast druga klasa odpowiada za dostęp do danych i wiąże je przez adnotacje z poleceniami SQL (Structured Query Language).

Przykładowa klasa BmiMeasurement:

```
package com.example.bmicalculator

import androidx.room.Entity
import androidx.room.PrimaryKey

@Entity(tableName="BmiMeasurements")
data class BmiMeasurement(

    @PrimaryKey(autoGenerate = true) val id: Long = 0L,
    val weightKg: Double,
    val heightCm: Double,
    val bmi: Double,
    val category: String,
    val timestamp: Long // System.currentTimeMillis()
```

```
)
```

Adnotacja @Entity określa nazwę tabeli, do której będą zapisywane rekordy, natomiast @PrimaryKey oznacza klucz główny.

Klasa BmiMeasurementDao:

```
package com.example.bmicalculator

import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.Query
import androidx.room.Update
import kotlinx.coroutines.flow.Flow

@Dao
interface BmiMeasurementDao {
    @Query("Select * from BmiMeasurements")
    fun getAll(): Flow<List<BmiMeasurement>>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(bmiMeasurement: BmiMeasurement): Long

    @Query("Delete from BmiMeasurements")
    suspend fun clear()

    @Delete
    suspend fun delete(bmiMeasurement: BmiMeasurement)
}
```

Klasa AppDatabase:

```
package com.example.bmicalculator

import android.content.Context
import androidx.room.Room
import androidx.room.RoomDatabase

abstract class AppDatabase : RoomDatabase() {
    abstract fun bmiMeasurementDao(): BmiMeasurementDao
```

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

Uwagi praktyczne:

- `exportSchema=false` może być używanie na start; do migracji w docelowym środowisku należy rozważyć eksport schematu.
- Zmiana schematu → zwiększ `version` i dodaj migracje (np. `AutoMigration`).
- Companion object pozwala na tworzenie statycznych pól i metod

ViewModel

Do obsługi połączenia danych w bazie z widokiem służą ViewModel oraz repozytorium. Repozytorium pozwala na oddzielenie logiki dostępu do danych (DAO) od logiki widoku, co ułatwia testowanie i zmianę źródła danych. Rolą **ViewModel** jest zapewnianie kontekstu aplikacji i połączenie z bazą danych.

Przykładowa klasa repozytorium (`BmiMeasurementRepository`):

```
package com.example.bmicalculator


import kotlinx.coroutines.flow.Flow

class BmiMeasurementRepository(private val dao:
BmiMeasurementDao) {
    fun getAll(): Flow<List<BmiMeasurement>> = dao.getAll()
    suspend fun save(rec: BmiMeasurement) = dao.insert(rec)
    suspend fun clear() = dao.clear()
}
```

Przykładowa klasa ViewModel (`BmiMeasurementViewModel`):

```
package com.example.bmicalculator

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch
import kotlin.math.pow

class BmiMeasurementViewModel(app: Application) :
AndroidViewModel(app) {
    private val repo =
BmiMeasurementRepository(AppDatabase.get(app).bmiMeasurementDa
o())

    val history = repo.getAll()
        .stateIn(viewModelScope, SharingStarted.Lazily,
emptyList())

    /** Oblicza BMI i zapisuje do bazy. Zwraca obliczony wynik
przez callback. */
    fun calculateAndSave(
        weightKg: Double,
        heightCm: Double,
        onSaved: (BmiMeasurement) -> Unit = {}
    ) {
        val hMeters = heightCm / 100.0
        if (hMeters <= 0.0 || weightKg <= 0.0) return

        val bmi = weightKg / hMeters.pow(2.0)
        val cat = bmiCategory(bmi)
        val rec = BmiMeasurement(
            weightKg = weightKg,
            heightCm = heightCm,
            bmi = bmi,
            category = cat,
            timestamp = System.currentTimeMillis()
```

```
        )

        viewModelScope.launch {
            val id = repo.save(rec)
            onSaved(rec.copy(id = id))
        }
    }

    fun clearHistory() = viewModelScope.launch { repo.clear()
}

    private fun bmiCategory(bmi: Double): String = when {
        bmi < 18.5 -> "Underweight"
        bmi < 25.0 -> "Normal"
        bmi < 30.0 -> "Overweight"
        else -> "Obesity"
    }
}
```

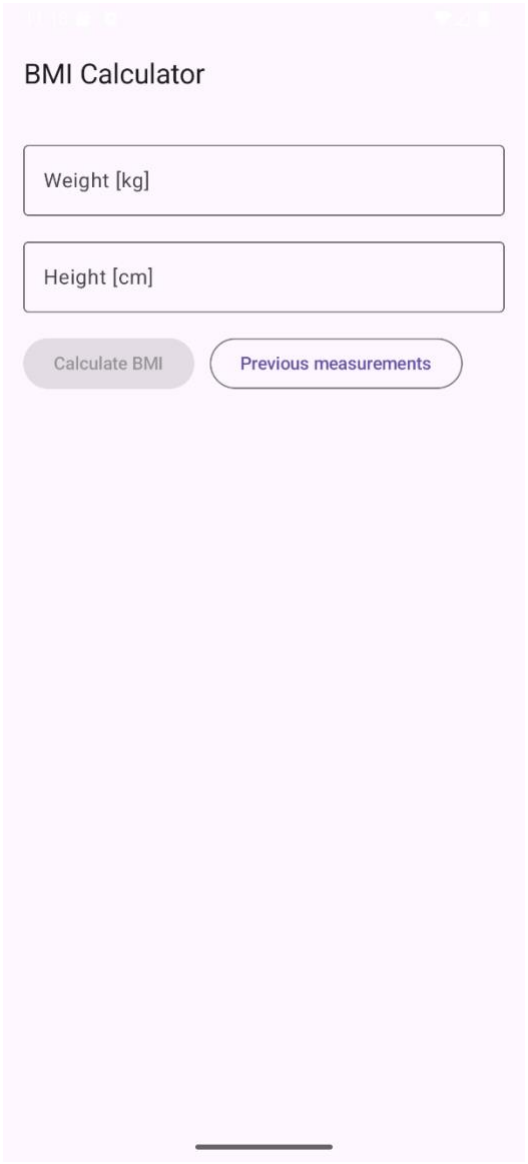
Widok

W aplikacji będą dwa widoki: główny oraz widok listy wcześniejszych pomiarów.

Ekran główny składa się z dwóch pól tekstowych z etykietami odpowiedzialnymi za masę ciała i wzrost, przycisk uruchamiający obliczenie i dodanie do bazy danych oraz przycisk przenoszący do listy poprzednich wyników.

Do wizualizacji bieżącego wyniku można użyć zwykłego tekstu lub użyć grafik jak dla poprzednich laboratoriów.

Przykładowy interfejs wygląda tak:



Ekran listy odpowiada za pokazywanie kolejnych wyników pomiaru oraz możliwość wycofania do poprzedniego widoku oraz czyszczenia listy. Składa się z tytułu, przycisku wycofania, przycisku czyszczenia listy wyników oraz wyświetlania kolejnych wyników z listy wczytywanej z bazy danych.

Lista z danymi zapisanymi w bazie danych jest inicjalizowana przez metodę klasy ViewModel zwracającą wszystkie rekordy z bazy danych. Aby ułatwić wyświetlanie, lista rekordów może zostać zwrócona za pomocą metody `collectAsStateWithLifecycle()`.

Górna część ekranu jest stała, natomiast wcześniej zapisane rekordy są wyświetlane jako kolejne wiersze za pomocą elementu LazyColumn, która wymaga implementacji funkcji anonimowej wyświetlającej rekordy. Robi się to przez użycie funkcji items()

przyjmującej listę rekordów oraz klucz. Funkcja anonimowa na wyjściu pozwala na sformatowanie wyświetlanych wartości.

Przykładowy kod wyświetlania elementów wygląda tak:

```
var list by vm.history().collectAsStateWithLifecycle()
LazyColumn {
          Items(list, key = { it.id }) { rec ->
               ShowRecord(rec)
          }
}
```

Gdzie vm jest obiektem ViewModel, `ShowRecord()` rec jest funkcją pokazującą dane w rekordzie, a `rec` jest obiektem klasy przechowującej wyniki pomiaru (`BmiMeasurement`).

Przykładowy ekran listy wygląda tak:

Klasa głównej aktywności odpowiada za uruchomienie pierwszego ekranu, nawiązanie i utrzymanie połączenia z bazą danych przez ViewModel oraz przekazywanie danych z bazy danych.

Przykładowa klasa głównej aktywności wygląda tak:

```
package com.example.bmicalculator

import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.lifecycle.viewmodel.compose.viewModel
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.compose.rememberNavController
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val nav = rememberNavController()
            val vm: BmiMeasurementViewModel = viewModel()

            NavHost(navController = nav, startDestination =
Routes.MAIN) {
                composable(Routes.MAIN) {
                    MainScreen(
                        vm = vm,
                        onShowHistory = {
nav.navigate(Routes.LIST) }
                    )
                }
                composable(Routes.LIST) {
                    HistoryScreen(
                        vm = vm,
                        onBack = { nav.popBackStack() }
                    )
                }
            }
        }
    }
}
```

W tej implementacji podstawowe polecenia nawigacji są przekazywane przez parametry odpowiednich funkcji ekranów.

Nawigacja

Nawigację w aplikacji można zawrzeć w obiekcie składającym się z dwóch pól typu String: pierwsze pole odpowiada za główny widok, natomiast drugie odpowiada za widok listy wcześniejszych pomiarów. Przykładowa implementacja wygląda tak:

```
object Routes {
    const val MAIN = "main"
    const val HISTORY = "history"
}
```

Zadanie do wykonania

Napisz kalkulator BMI składający się z dwóch widoków, który przechowuje dotychczasowe wyniki w bazie danych:

- Widok główny zawierający następujące elementy:
  - Dwa pola tekstowe dla wzrostu (w metrach lub centymetrach) oraz masy ciała w kilogramach
  - Przycisk „Calculate BMI” lub „Oblicz BMI”
  - Przycisk, który pozwala przejść do widoku historii wyników.
- Widok historii wyników zawierający:
  - Listę wcześniejszych wyników
  - Przycisk pozwalający na powrót do widoku głównego

Prześlij kod i zrzuty ekranowe (lub screencast – krótkie video) przedstawiające działanie aplikacji w końcowym etapie.