Version Catalog, ViewModel, Menu, Composable, Gestures

Version Catalog

https://blog.gradle.org/best-practices-naming-version-catalog-entries#version-catalogs
W pliku libs.version.toml znajdują się wszystkie zależności. Możemy określić wersję bibliotek, grupy zależności oraz aliasy a później używać ich w projekcie.
W dzisiejszej lekcji poznamy ModelView. To dobra okazja na przetestowanie Version Catalog:

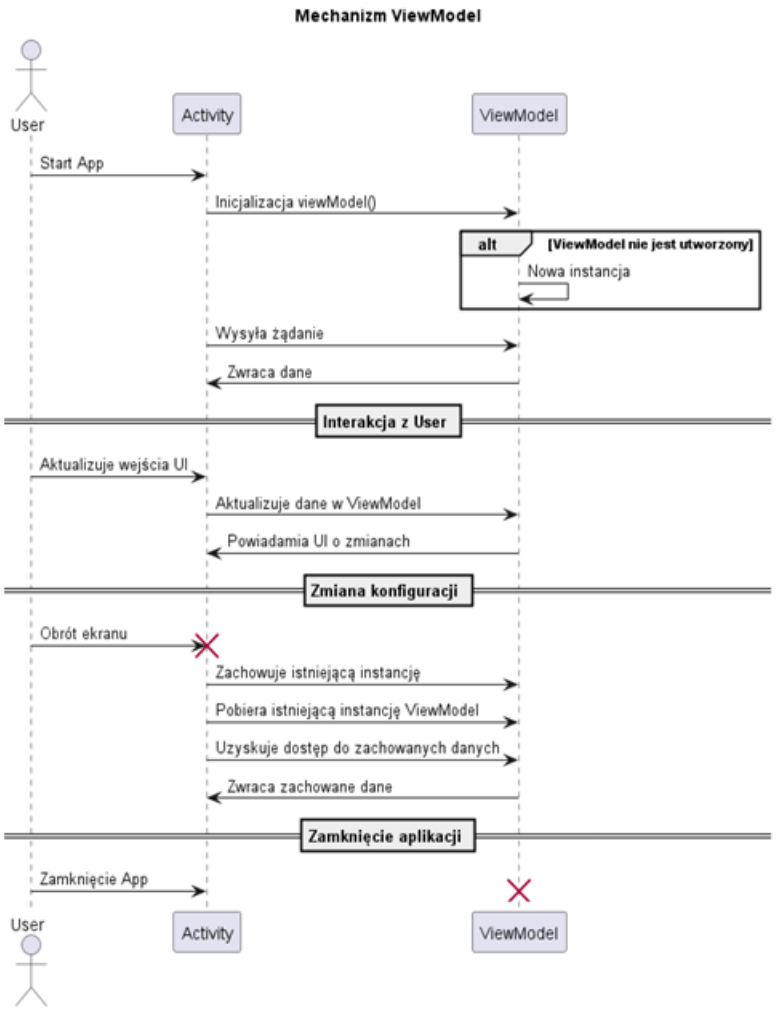
W [versions]:
lifecycle-compose = "2.8.7"
W [libraries]:

androidx-lifecycle-viewmodel-compose = { group = "androidx.lifecycle", name = "lifecycle-viewmodel-compose", version.ref = "lifecycle-compose" }

Należy pamiętać, że gradle zamieni “-„ na „.” W dependencies możemy się odwołać do biblioteki w sposób hierarchiczny:
implementation(libs.androidx.lifecycle.viewmodel.compose)

ViewModel

https://developer.android.com/topic/libraries/architecture/viewmodel



ViewModel jest wykorzystywany do przechowywania i przetwarzania danych, które są potrzebne do wyświetlania na interfejsie użytkownika, ale powinny przetrwać zmiany konfiguracji Activity lub Fragment.

W onCreate() tworzymy instancję konkretnej ViewModel za pomocą viewModel().
viewModel() automatycznie tworzy ViewModel tylko raz i przechowuje tę samą instancję przez cały czas trwania Activity.
Gdy ekran zostanie obrócony, viewModel() zwróci istniejącą instancję ViewModel, zamiast tworzyć nową. Dzięki temu dane przechowywane w ViewModel są zachowane.

ViewModel przechowuje zmienne jako MutableStateFlow. MutableStateFlow pozwala Compose automatycznie reagować na zmiany wartości bez konieczności ręcznego odświeżania widoku.

Np. Gdy użytkownik wprowadza tekst w jednym z pól tekstowych, powinna zostać wywoana funkcja aktualizująca wartość MutableStateFlow.
Jeżeli Compose obserwuje te wartości za pomocą collectAsState(), widok automatycznie odświeży się i wyświetli nowe dane.

ViewModel zapewnia czyste oddzielenie logiki aplikacji od kodu interfejsu użytkownika (UI). W ViewModel przechowujemy stan aplikacji i dane biznesowe, a ekrany Compose tylko wyświetlają i aktualizują te dane.

Przykład:

Tworzymy MyViewModel który będzie obsługiwał 3 zmienne:

```
import androidx.lifecycle.ViewModel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow

class MyViewModel : ViewModel() {
    // Zmienna 1
    private val _place1 = MutableStateFlow("")
    val place1 = _place1.asStateFlow()

    // Zmienna2
    private val _place2 = MutableStateFlow("")
    val place2 = _place2.asStateFlow()

    // Zmienna 3
    private val _place3 = MutableStateFlow("")
    val place3 = _place3.asStateFlow()

    // Funkcje do aktualizacji danych:
    fun updatePlace1(newValue: String) {
        _place1.value = newValue
    }

    fun updatePlace2(newValue: String) {
        _place2.value = newValue
    }

    fun updatePlace3(newValue: String) {
        _place3.value = newValue
    }
}
```

Nazwa	Typ	Dostęp	Rola
_place1	MutableStateFlow	private	Wewnętrzna zmienna pozwalająca na modyfikację stanu przez ViewModel
place1	StateFlow	public	Zmienna tylko do odczytu, udostępniana na zewnątrz ViewModel

StateFlow jest odpowiednikiem strumienia danych (ang. stream), który emituje bieżącą wartość i aktualizuje ją, gdy dane się zmieniają.
collectAsState() to funkcja, która pozwala Compose obserwować StateFlow i automatycznie aktualizować interfejs użytkownika (UI) za każdym razem, gdy zmieni się jego wartość.
Tworzy State – specjalny obiekt zarządzany przez Compose, który powoduje automatyczne odświeżanie UI, gdy StateFlow wyemituje nową wartość.
collectAsState() zwraca State, który jest obiektem opakowującym rzeczywistą wartość. Aby odczytać bieżącą wartość, należy użyć .value.

Gdy chcemy wykorzystać te dane (wyświetlanie i edycja) w funkcji Composable
do wyświetlania używamy place1 z obiektu MyViewModel, do zmiany wartości funkcji do akutalizacji zmiennych z obiektu MyViewModel

```
@Composable
fun PlaceScreen1(viewModel: MyViewModel) {
    val place1 = viewModel.place1.collectAsState().value

    Column(modifier = Modifier.padding(16.dp)) {
        TextField(
            value = place1,
            onValueChange = { viewModel.updatePlace1(it) },
            label = { Text("Place 1") }
        )
        Text(text=place1)
    }
}
```

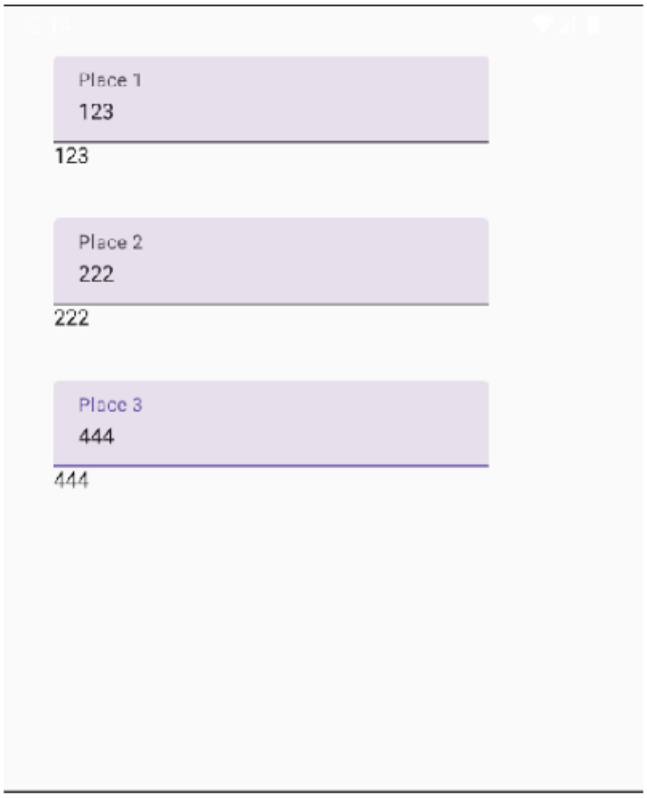
Obiekt MyViewModel tworzymy tylko raz za pomocą viewModel():
Jeśli instancja MyViewModel już istnieje viewModel() zwróci ją bez tworzenia nowego obiektu. Jeśli nie, utworzy nową instancję.

```
import android.os.Bundle
import androidx.activity.ComponentActivity
import androidx.activity.compose.setContent
import androidx.lifecycle.viewmodel.compose.viewModel

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            val viewModel: MyViewModel = viewModel()
            MyApp(viewModel)
        }
```

```
    }
}
```

Funkcja kompozycyjna do rozmieszczenia elementów:

```
@Composable
fun MyApp(viewModel: MyViewModel) {

    Column( modifier = Modifier.fillMaxSize().padding(16.dp)) {
        PlaceScreen1(viewModel)
                        //... kolejne ekrany
    }
}
```



Menu oraz Composable zamiast Fragmentów

Jeżeli chcielibyśmy pozwolić użytkownikowi na wybór ile elementów ma się wyświetlać na ekranie, możemy umożliwić ten wybór za pomocą menu.
Np:
1.DropdownMenu – menu rozwijane na ekranie, służące do prostych wyborów, np. filtrowania czy zmiany ustawień.
2.TopAppBar z Menu – menu w pasku aplikacji (np. trzy kropki) do najważniejszych opcji, takich jak ustawienia czy pomoc.
3.Navigation Drawer – menu boczne, które zapewnia nawigację po różnych sekcjach aplikacji.

Informację o liczbie przechowywanych ekranów możemy zapisać w tym samym ModelView:

```
private val _screenCount = MutableStateFlow(1)
val screenCount = _screenCount.asStateFlow()
//…
fun setScreenCount(count: Int) {
    _screenCount.value = count
}
```

DropdownMenu:

```
import androidx.compose.foundation.layout.*
import androidx.compose.material3.*
import androidx.compose.runtime.*

@Composable
fun MyDropdownMenu(
    screenCount: Int,
    onScreenCountChange: (Int) -> Unit
) {
    var expanded by remember { mutableStateOf(false) }

    Column {
        Text("Wybierz liczbę ekranów:", style = MaterialTheme.typography.titleMedium)
        Box {
            Button(onClick = { expanded = true }) {
                Text("Liczba ekranów: $screenCount")
            }
            DropdownMenu(
                expanded = expanded,
                onDismissRequest = { expanded = false }
            ) {
                (1..3).forEach { count ->

                    DropdownMenuItem(
                        text =  {Text("$count")},
                        onClick = {
                            onScreenCountChange(count)
                            expanded = false
                        }
                    )
                }
            }
        }
    }
}
```

Nasza funkcja kompozycyjna menu będzie potrzebowała 2 argumentów:
screenCount: Int – Przechowuje liczbę aktualnie wybranych ekranów, którą pokazujemy w przycisku. Jest to wartość przekazywana z zewnątrz do komponentu.
W naszym wypadku wartość z ViewModel.
onScreenCountChange: (Int) -> Unit – Funkcja, która zostanie wywołana, gdy użytkownik wybierze nową liczbę ekranów.

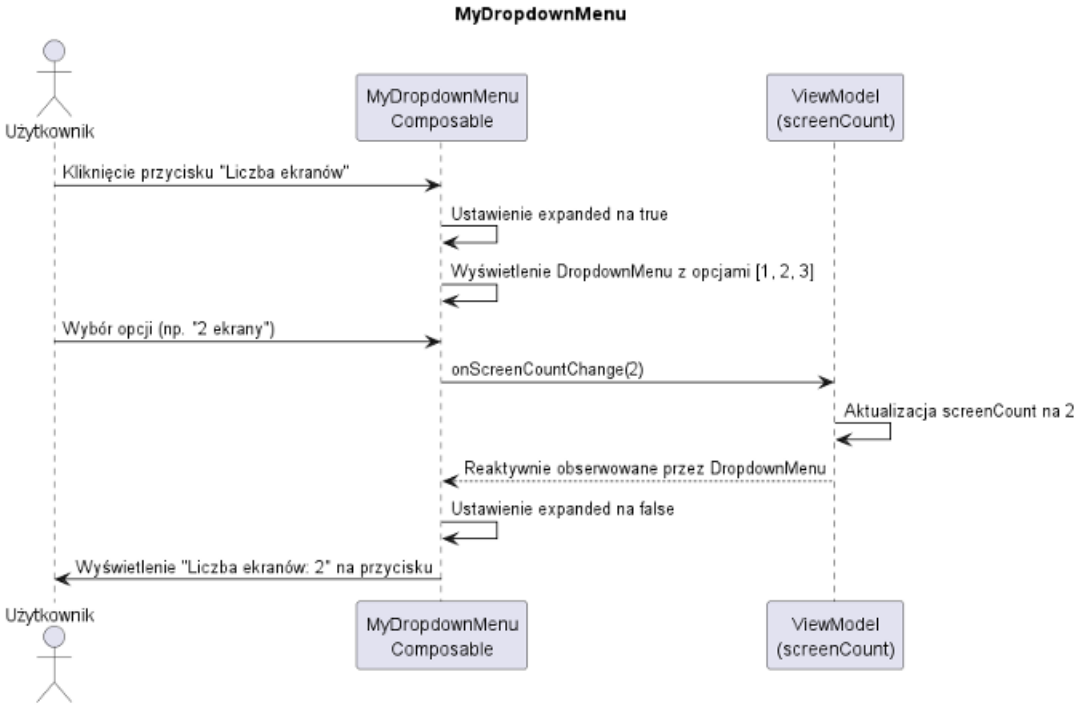
Expanded – przechowuje informację czy menu jest rozwinięte.

Dodajemy kompozycję **DropdownMenu** oraz dynamicznie w pętli forEach 3 **DropDownMenuItem**, które po naciśnięciu będą zmieniały wartości danych w ViewModel.

Funkcję wywołamy tam gdzie resztę ekranów(MyApp):

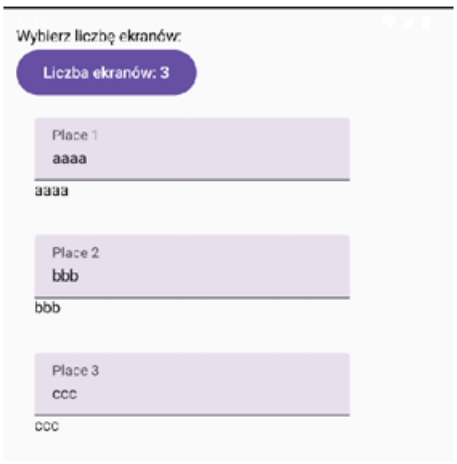
```
val screenCount by viewModel.screenCount.collectAsState()
MyDropdownMenu  (
   screenCount = screenCount,
   onScreenCountChange = { newCount -> viewModel.updateScreenCount(newCount) }
)
```

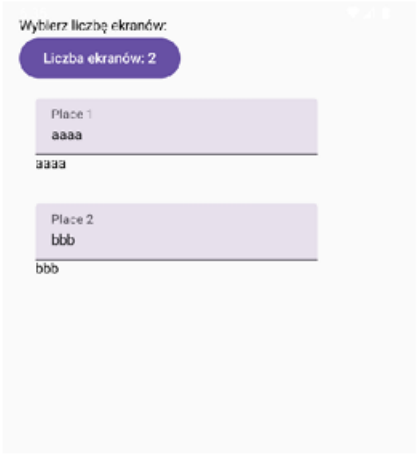
screenCount – pokazuje na bieżąco aktualizowaną wartość liczby ekranów.
{ newCount -> viewModel.setScreenCount(newCount) } - funkcja ustawia wartość przez funkcję z viewModel.



Dodanie prostego wyświetlania wybranej liczby ekranów w MyApp:

```
if (screenCount >= 1) PlaceScreen1(viewModel)
if (screenCount >= 2) PlaceScreen2(viewModel)
if (screenCount >= 3) PlaceScreen3(viewModel)
```





Można zmienić sposób wyświetlania elementów w zależności od orientacji telefonu np. jeżeli orientacja pionowa to wyświetlamy w jednym wierszu :

```
val configuration = LocalConfiguration.current
val isPortrait = configuration.orientation == android.content.res.Configuration.ORIENTATION_PORTRAIT
```

```
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.Row
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.layout.padding
import androidx.compose.runtime.Composable
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue
import androidx.compose.ui.Modifier
import androidx.compose.ui.platform.LocalConfiguration
import androidx.compose.ui.unit.dp

@Composable
fun MyApp(viewModel: MyViewModel) {

  Column( modifier = Modifier.fillMaxSize().padding(16.dp)) {
    // Stan liczby widocznych ekranów
    val screenCount by viewModel.screenCount.collectAsState()

    val configuration = LocalConfiguration.current
    val isPortrait = configuration.orientation == android.content.res.Configuration.ORIENTATION_PORTRAIT


    MyDropdownMenu  (
      screenCount = screenCount,
      onScreenCountChange = { newCount -> viewModel.updateScreenCount(newCount) }
    )

    if (isPortrait) {

        if (screenCount >= 1) PlaceScreen1(viewModel)
        if (screenCount >= 2) PlaceScreen2(viewModel)
        if (screenCount >= 3) PlaceScreen3(viewModel)

    } else {
      Row(modifier = Modifier.fillMaxSize()) {
```

```
    if (screenCount >= 1) { PlaceScreen1(viewModel) }
        if (screenCount >= 2) { PlaceScreen2(viewModel) }
        if (screenCount >= 3) { PlaceScreen3(viewModel) }
      }
    }
  }
}
```

Aby można było przechodzić między ekranami można umieścić composable w Boxie, a następnie użyć wykrywanie gestów (detectHorizontalDragGestures,detectVerticalDragGestures):
https://canopas.com/gestures-in-jetpack-compose-all-you-need-to-know-part-1-9d26570e56bb
W composable, w którym będzie Box:

```
var totalDrag by remember { mutableStateOf(0f) }
```

W .pointerInput dla modfier'a Boxa dodajemy obsługę gestu „swipe":
Obsługa gestu - akumulujemy przesunięcie palca dopóki nie skończymy przesuwać.
Gdy oderwiemy palec sprawdzamy zakumulowaną wartość przesunięcia. Gdy będzie miała odpowiednią wartość zmieniamy stronę:

```
Box(
  modifier = Modifier
    .fillMaxSize()
.pointerInput(Unit) {

  detectVerticalDragGestures(
    onVerticalDrag = { change, dragAmount ->
      change.consume()
      totalDrag += dragAmount
      Log.d("PRZESUNIECIE: ","amount: "+dragAmount+" total: "+totalDrag)

    },
    onDragEnd = {
      if (totalDrag < -50 && currentPage < 2 ) {
        mainViewModel.updateCurrentPage(currentPage + 1)
      } else if (totalDrag > 50 && currentPage > 0 ) {
        mainViewModel.updateCurrentPage(currentPage - 1)
      }
      totalDrag = 0f
    }
  )
},contentAlignment = Alignment.Center
) {
  when (currentPage) {
    0 -> PlaceScreen1(viewModel)
    1 -> PlaceScreen2(viewModel)
    2 -> PlaceScreen3(viewModel)
  }
}
```

Zadanie:

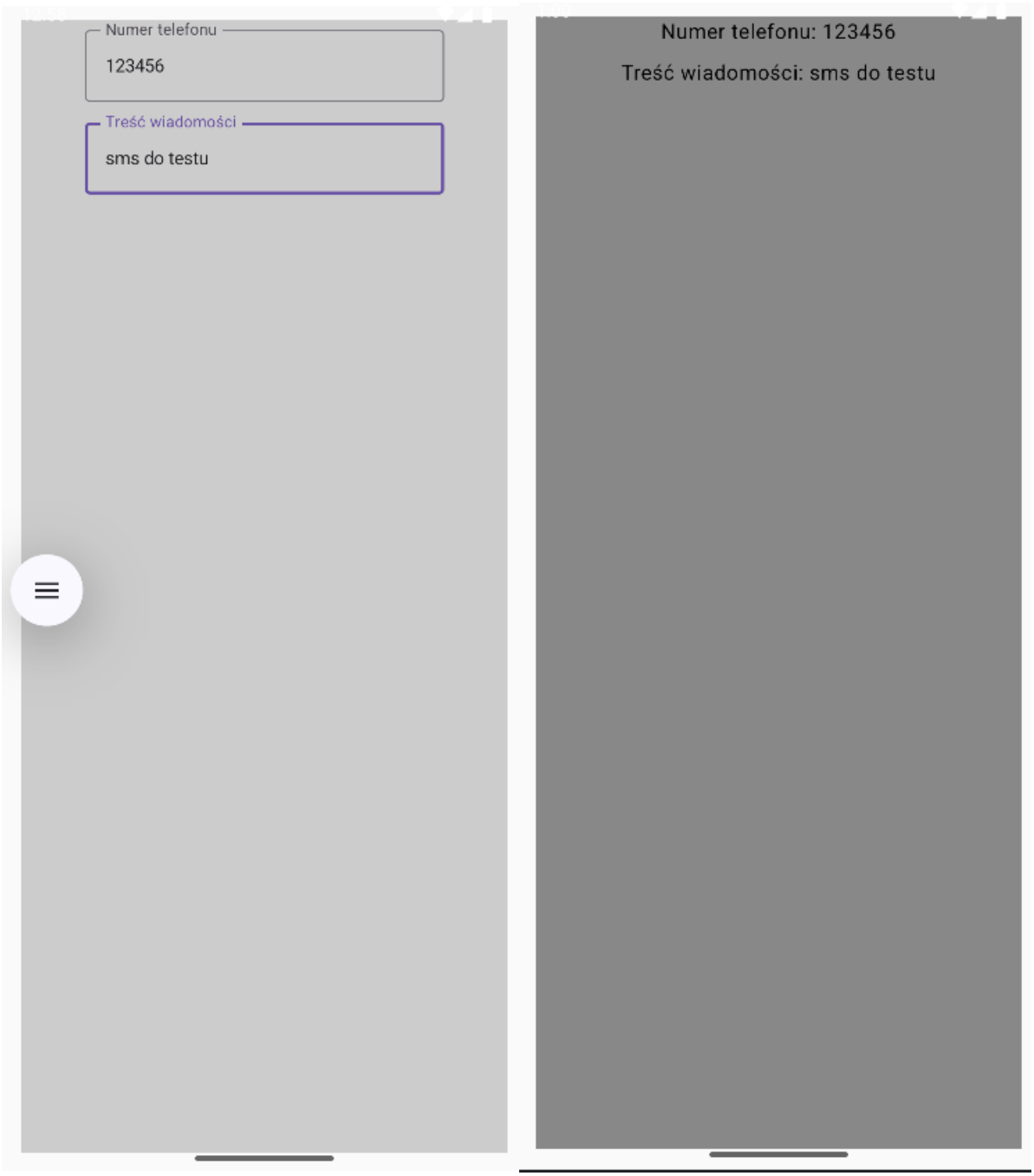
Utwórz trzy ekrany, ekran pierwszy umożliwia wpisanie numeru telefonu oraz treści wiadomości, drugi ekran wyświetla wypisane przez użytkownika dane, 3 ekran zawiera przycisk wyślij.
Na kliknięcie przycisku należy wysłać wiadomość pod wskazany numer (użyj SmsManager'a).

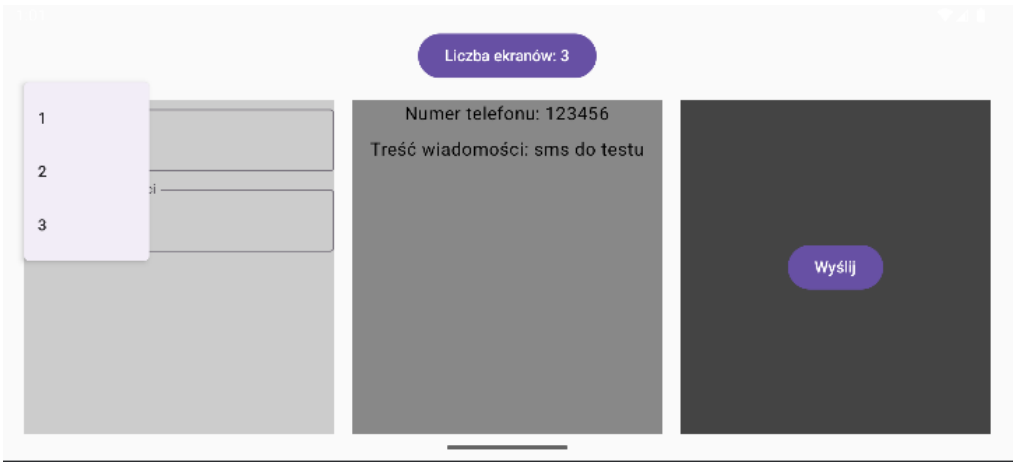
Użyj do przechowywania danych ViewModel.
Użyj gestu przesunięcia do poruszania się pomiędzy ekranami gdy orientacja pionowa telefonu, gdy pozioma użyj menu wyświetlając 1,2 lub 3 ekrany na raz.
Pamiętaj o sprawdzeniu uprawnień przy wysyłaniu sms.
Każdy ekran powinien mieć inny kolor tła.

Uwaga! Aby nasze ekrany mogły podzielić ekran główny w orientacji poziomej na 3 równe części (w jednym wierszu), należy w każdym ekranie użyć modifiera z wagą 1f i przekazać go do kolumny w ekranie.
W wierszu Row kolumny w której są wszystkie ekrany:

```
Row(modifier =  Modifier.fillMaxSize()) {

  //...
     PlaceScreen1(viewModel,modifier = Modifier
        .weight(1f)
        .fillMaxHeight()
        .padding(8.dp))
  //...
```

```
@Composable
fun PlaceScreen1(viewModel: MyViewModel, modifier: Modifier = Modifier) {
   val place1 = viewModel.place1.collectAsState().value

   Column(modifier = modifier.padding(16.dp)) {
      TextField(
         value = place1,
         onValueChange = { viewModel.updatePlace1(it) },
         label = { Text("Place 1") }
      )
      Text(text=place1)
   }
}
```


Numer telefonu
123456
Treść wiadomości
sms do testu
Numer telefonu: 123456
Treść wiadomości: sms do testu


Liczba ekranów: 3
1
2
3
Numer telefonu: 123456
Treść wiadomości: sms do testu
Wyślij