

Jetpack Compose

Jetpack Compose to nowy sposób pisania interfejsów użytkownika w natywnych aplikacjach Androidowych. Piszemy deklaracyjny UI w Kotlinie! Logika biznesowa i interfejs użytkownika w jednym języku programowania.

Deklaracyjny UI

Dla osób, które miały jakąkolwiek styczność z [Flutterem](#), czy React Native to koncepcja powinna być znana. W deklaracyjnym podejściu skupiamy się na tym, **CO** trzeba zrobić, a nie **JAK** coś zrobić.

W przypadku korzystania z widoków zdefiniowanych w plikach XML musieliśmy ręcznie przeszukiwać drzewo widoków poprzez jakąś formę `findViewById` i ręcznie modyfikować stan komponentów.

W Compose tego nie robimy.

Widok jest tworzony w całości, a wszelkie zmiany są wykonywane jak najrzadziej. Przerysowywane jest tylko to, co konieczne, na podstawie przekazanego modelu.

Potencjalnym problemem może być koszt (mocy obliczeniowej, baterii) przerysowania na nowo całego ekranu. Compose inteligentnie wybiera komponenty, które muszą zostać przerysowane, zachowując bez zmian to, co się da.

Jak działa Compose?

Interfejs użytkownika tworzymy w Compose za pomocą zbioru funkcji oznaczonych annotacją `@Composable`, które przyjmują model danych i na jego podstawie renderują elementy widoku.

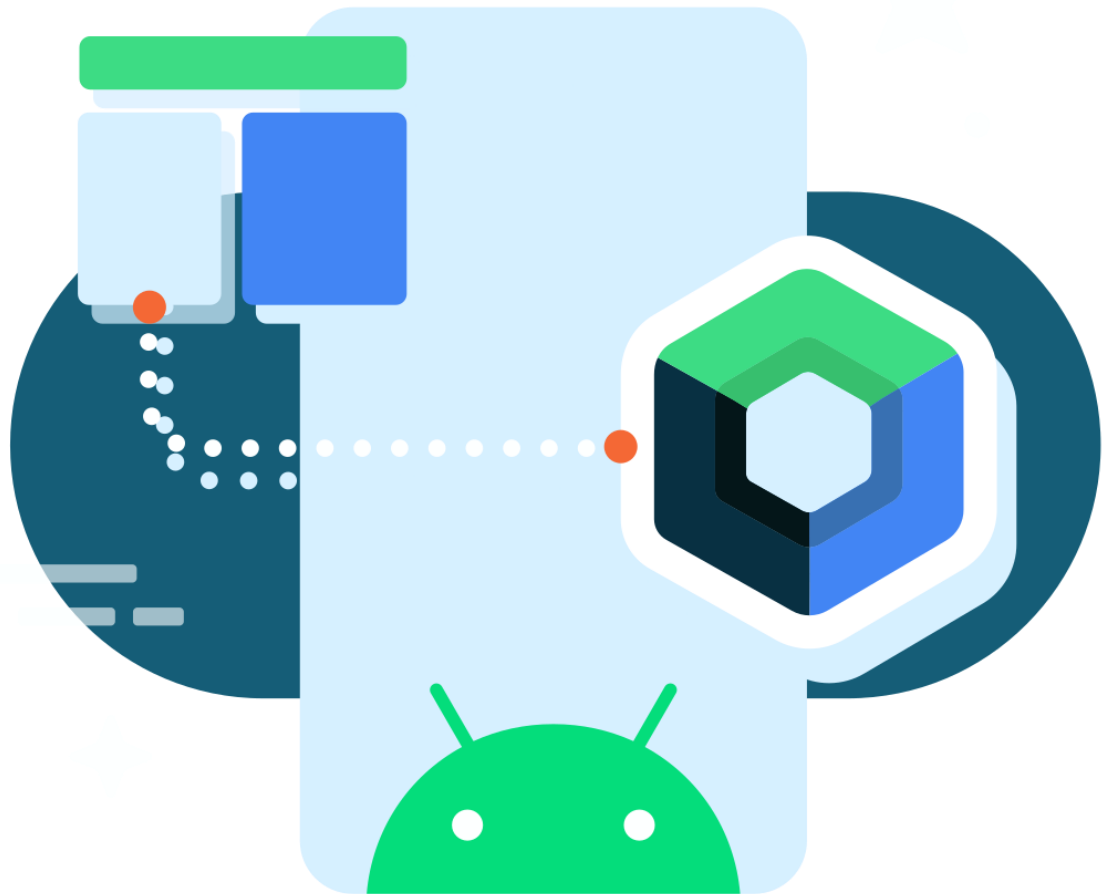
Compose wykorzystuje **Unidirectional Data Flow** (UDF). Oznacza to, że kierunek przepływu danych jest w jedną stronę (od źródeł danych do UI). Kierunek przepływu akcji jest w stronę przeciwną (od UI do źródła danych, gdzie są przygotowane do użycia). Przykładem akcji jest naciśnięcie przycisku przez użytkownika.

W Compose wyróżniamy trzy fazy:

1. **Composition** – jaki UI trzeba pokazać. Compose uruchamia funkcję `Composable` i tworzy opis UI.
2. **Layout** – gdzie umieścić UI. Ta faza składa się z dwóch kroków: pomiaru i umieszczenia. Elementy układu mierzą i umieszczają siebie oraz wszelkie elementy podrzędne we współrzędnych 2D, dla każdego węzła w drzewie.
3. **Draw** – jak narysować UI. Elementy UI rysują się na Canvas.

Ważnym procesem w cyklu życia funkcji `Composable` jest **rekompozycja**. To proces wywołania ponownie funkcji `@Composable`, gdy zmieniają się jej parametry. Cała magia inteligentnego i optymalnego przerysowywania widoków znajduje się właśnie tutaj. Compose

wykrywa, które parametry się zmieniły i uruchamia po raz kolejny tylko te funkcje, które korzystają ze zmienionych parametrów, pomijając te, które się nie zmieniły.



Stan w Jetpack Compose

Kilka ważnych uwag dotyczących funkcji Composable:

- Mogą wykonywać się w dowolnej kolejności.
- Mogą wykonywać się równolegle.
- Rekompozycja pomija najwięcej jak to możliwe.
- Rekompozycja jest optymistyczna i może zostać anulowana.
- Funkcje mogą być uruchamiane bardzo często (nawet dla każdej klatki animacji).

Ważnym aspektem związanym z procesem rekompozycji jest stan, czyli wartość, która zmienia się w czasie. Parametry funkcji Composable są reprezentacją stanu. Rekompozycja następuje w momencie ich zmiany. Żeby Composable został przerysowany, trzeba mu to powiedzieć, wprost przekazując nowe parametry.

Stan możemy zapamiętać w funkcji Composable np. za pomocą funkcji `remember`. Stan w Compose jest reprezentowany przez interfejs `State`, który jest `immutable`. Mamy też mutowalny `MutableState`.

Compose wspiera również inne popularne obserwowalne typy danych, np. `LiveData`, `Flow`, `RxJava2`.

Ostatni aspekt, o którym chcę wspomnieć to funkcje Composable, którą zapamiętują w sobie stan (funkcje **stateful**) oraz funkcje, które tego nie robią (funkcje **stateless**) i po prostu renderują UI na podstawie przekazanych parametrów.

W jaki sposób otrzymać funkcję stateless, która nie trzyma w sobie stanu? Najprościej poprzez wyniesienie tego stanu poziom wyżej — do funkcji, która ją wywołuje. Funkcja stateless ma wtedy parametr związany z przekazaniem stanu oraz lambdę do jej zmiany.

Wiem, że jest tego dużo. Może to się wydać przytłaczające, ale bądź spokojny! Bardzo łatwo zacząć z Compose! Dopiero po jakimś czasie będziesz w stanie zagłębiać się w szczegóły techniczne i aspekty optymalizacji.

Podstawowe komponenty

Scaffold

To gotowy komponent, który ma przygotowane sloty na typowe elementy interfejsu:

- `TopAppBar`
- `BottomAppBar`
- `FloatingActionButton`.

[Dokumentacja](#)

TopAppBar/BottomAppBar

Górny/dolny pasek nawigacyjny.

Dokumentacja: [TopAppBar](#) i [BottomAppBar](#).

Box

Można go porównać do `FrameLayout` znanego z XML. To kontener na inne elementy.

[Dokumentacja](#)

Surface

Można porównać do `CardView` znanego z XML.

[Dokumentacja](#)

Column/Row (LazyColumn/LazyRow)

`Column` i `Row` to komponenty pozwalające na pozycjonowanie elementów widoku wertykalnie i horyzontalnie. Odpowiedniki z przedrostkiem `Lazy` pozwalają na tworzenie list z wieloma elementami, które nie są jednocześnie widoczne na ekranie. Można je porównać do `RecyclerView`.

Dokumentacja: [Column](#), [Row](#), [LazyColumn](#), [LazyRow](#).

Button

Komponent do przycisków. Jest kilka różnych gotowych wersji, np. `OutlinedButton`, `TextButton`.

[Dokumentacja](#)

Text

Za jego pomocą wyświetlimy tekst.

[Dokumentacja](#)

Modifier

Moim zdaniem, to jeden z najciekawszych elementów `Compose`. W dokumentacji jest nawet cała oddzielna strona poświęcona `Modifierom`! `Modifiery` pozwalają modyfikować wygląd i zachowanie `Composabli`.

`Modifier` to po prostu interfejs z przygotowanymi funkcjami umożliwiającymi łączenie kilku `Modifierów` w łańcuch wywołań za pomocą kropki. Mamy do wyboru wiele wbudowanych modyfikatorów, ale można też pisać je samemu.

Musisz zapamiętać jedno — kolejność **`Modifierów` ma ogromne znaczenie!**

[Dokumentacja](#)

Podstawowe Modifiery

`fillMaxSize` / `fillMaxWidth` / `fillMaxHeight`

Za ich pomocą definiujemy wielkość danego komponentu.

[Dokumentacja](#).

`padding`

`Padding` umożliwia utworzenie pustych przestrzeni wokół komponentu. Co ważne, w łańcuchu wywołań może być wykorzystany kilkakrotnie. Każdy `padding` będzie się wtedy odnosić do innych elementów widoku. Pamiętaj, **kolejność `Modifierów` jest bardzo ważna**.

[Dokumentacja](#)

`clickable`

`Composable` będzie klikalny i automatycznie uzyskamy ripple effect.

[Dokumentacja](#)

`background`

Ustawisz dla `Composable` `background`. Jest kilka opcji: `Color/Brush`, `Shape`, `alpha`.

[Dokumentacja](#)

border

`Composable` będzie miał obwódkę z podanym `BorderStroke` oraz `Shape`.



















[Dokumentacja](#)

Demonstracyjna aplikacja

Przeanalizujemy fragment kodu typowej aplikacji.

W naszym przykładzie będzie to prosta aplikacja z jednym ekranem — listą piosenek. Piosenkę można polubić oraz na nią kliknąć. Tak wygląda docelowa aplikacja:

Songs

- | | | |
|---|---------------------------|---|
|  | Esther
Popcorn Trees |  |
|  | Golden Hour
hypewave |  |
|  | Moon
midnight lover |  |
|  | puppet master
yamarcus |  |
|  | reflection
aaesth |  |
|  | shoe polish
komoere |  |
|  | manuela
mr moobyy |  |
|  | charm me
take 1 |  |
|  | moving day
redemoo |  |
|  | main court
nine |  |

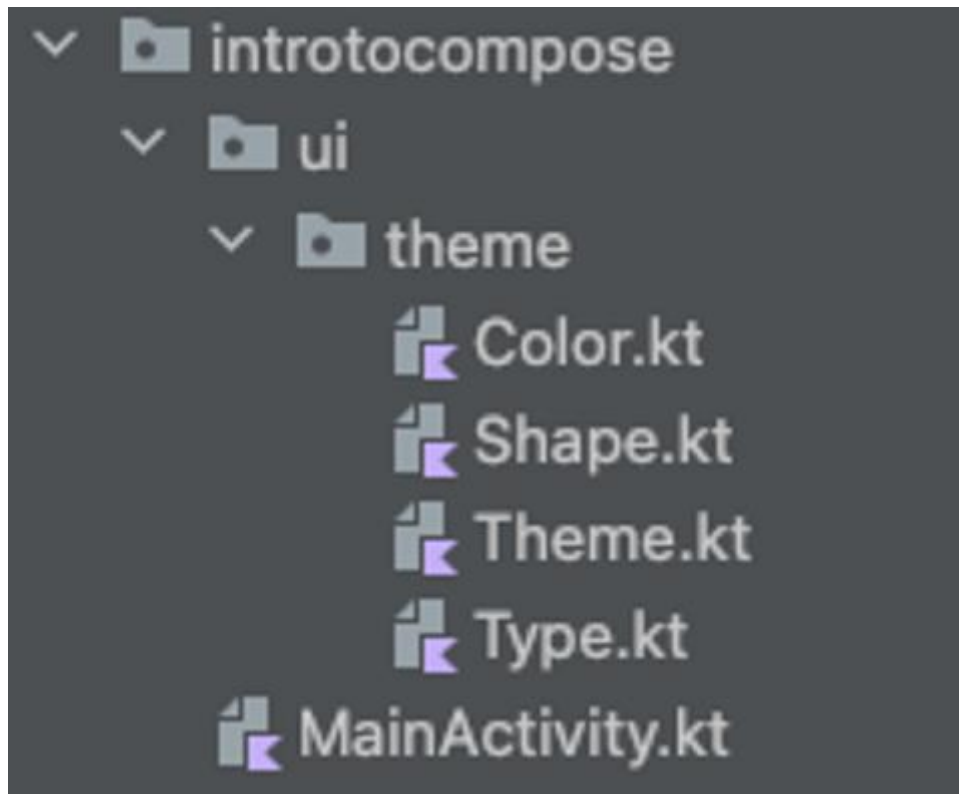
Struktura projektu w Android Studio

Podczas tworzenia projektu wybrałem opcję **Empty Compose Activity**. Tak wygląda MainActivity.kt:

```
class MainActivity : ComponentActivity() {  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContent {  
            IntroToComposeTheme {  
                // A surface container using the 'background' color from  
the theme  
                Surface(modifier = Modifier.fillMaxSize(), color =  
MaterialTheme.colors.background) {  
                    Greeting("Android")  
                }  
            }  
        }  
    }  
}  
  
@Composable  
fun Greeting(name: String) {  
    Text(text = "Hello $name!")  
}  
  
@Preview(showBackground = true)  
@Composable  
fun DefaultPreview() {  
    IntroToComposeTheme {  
        Greeting("Android")  
    }  
}
```

Kotlin

Android Studio wygenerowało strukturę projektu:



W `Color.kt` mamy definicję domyślnych kolorów.

```
import androidx.compose.ui.graphics.Color

val Purple200 = Color(0xFFBB86FC)
val Purple500 = Color(0xFF6200EE)
val Purple700 = Color(0xFF3700B3)
val Teal200 = Color(0xFF03DAC5)
```

Kotlin

W `Shape.kt` znajdziemy domyślnie zdefiniowane wartości kształtów.

```
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.material.Shapes
import androidx.compose.ui.unit.dp

val Shapes = Shapes(
    small = RoundedCornerShape(4.dp),
    medium = RoundedCornerShape(4.dp),
    large = RoundedCornerShape(0.dp)
)
```

Kotlin

W `Type.kt` domyślna typografia.

```
import androidx.compose.materialTypography
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.text.font.FontFamily
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.unit.sp
```



```

val Typography = Typography(
    body1 = TextStyle(
        fontFamily = FontFamily.Default,
        fontWeight = FontWeight.Normal,
        fontSize = 16.sp
    )
)

```

Kotlin

W `Theme.kt` zobaczymy automatycznie wygenerowane wsparcie dla trybu jasnego/ciemnego.

```

import androidx.compose.foundation.isSystemInDarkTheme
import androidx.compose.material.MaterialTheme
import androidx.compose.material.darkColors
import androidx.compose.material.lightColors
import androidx.compose.runtime.Composable

private val DarkColorPalette = darkColors(
    primary = Purple200,
    primaryVariant = Purple700,
    secondary = Teal200
)

private val LightColorPalette = lightColors(
    primary = Purple500,
    primaryVariant = Purple700,
    secondary = Teal200
)

@Composable
fun IntroToComposeTheme(darkTheme: Boolean = isSystemInDarkTheme(),
    content: @Composable () -> Unit) {
    val colors = if (darkTheme) {
        DarkColorPalette
    } else {
        LightColorPalette
    }

    MaterialTheme(
        colors = colors,
        typography = Typography,
        shapes = Shapes,
        content = content
    )
}

```

Kotlin

Zaczynamy

Zacznijmy od najbardziej zewnętrznych komponentów widoku. W tym wypadku skorzystamy ze `Scaffold`, który posiada dodatkowe, interesujące nas, sloty. Skonfigurujemy górny pasek nawigacyjny z nazwą aplikacji oraz obsługę `snackbar`ów.

`TopAppBar` to po prostu `Text`. Możliwość pokazywania `snackbar`ów osiągniemy poprzez przekazanie `snackbarHostState` do `scaffoldState`. Kod poniżej piszę w oddzielnym `Composable` nazwanym `SongsScreen`, który wywołuję w `MainActivity#onCreate`.

```
Scaffold(
    modifier = Modifier.fillMaxSize(),
    scaffoldState = rememberScaffoldState(snackbarHostState =
snackbarHostState),
    topBar = { TopAppBar(title = { Text(text = "Songs") }) },
) {
    // content ...
}
```

Kotlin

W środku Scaffold chcemy umieścić docelową zawartość ekranu.

W tym wypadku jest to po prostu lista piosenek. Osiągniemy to za pomocą LazyColumn i funkcji items, do której prześlemy listę oraz identyfikator każdego elementu. Lambda w środku funkcji items służy zdefiniowaniu, co ma się wydarzyć dla każdego elementu listy songs.

```
LazyColumn {
    items(songs, key = { it.id }) { song ->
        SongItem(
            song = song,
            onSongClick = onSongClick,
            onFavouriteClick = onFavouriteClick,
        )
    }
}
```

Kotlin

Lista piosenek songs jest uzyskana z ViewModelu i przekonwertowana do stanu, który rozumie Compose.

```
val songs by songsViewModel.songs.collectAsStateWithLifecycle()
Kotlin
```

W ramach LazyColumn korzystamy też z funkcji SongItem, która reprezentuje jeden element listy. Najbardziej zewnętrzny Row (elementy ułożone horyzontalnie), w środku Image, kolumna z dwoma napisami oraz animowany przycisk polubienia piosenki.

SongItem jest funkcją typu stateless, czyli **nie posiada w sobie żadnego stanu**. Wszystko jest renderowane na podstawie przekazanych do niej parametrów, w tym dane piosenki oraz lambdy zmieniające stan wybranego elementu.

```
@OptIn(ExperimentalAnimationApi::class)
@Composable
private fun SongItem(
    song: Song,
    onSongClick: (Song) -> Unit,
    onFavouriteClick: (Song) -> Unit,
) {
    Row(
        modifier = Modifier
            .fillMaxWidth()
            .clickable { onSongClick(song) }
            .padding(all = 10.dp)
    ) {
```

```

        Image(
            modifier = Modifier
                .align(CenterVertically)
                .padding(horizontal = 10.dp),
            painter = painterResource(id = R.drawable.ic_android),
            contentDescription = "cover",
        )
        Column(
            modifier = Modifier.weight(1F),
        ) {
            Text(
                text = song.trackName,
                fontSize = 15.sp,
            )
            Text(
                text = song.artist,
                fontSize = 12.sp,
            )
        }
        AnimatedContent(targetState = song.isFavourite) {
            IconButton(
                modifier = Modifier,
                onClick = { onFavouriteClick(song) },
                content = {
                    val icon = if (it) Icons.Filled.Favorite else
Icons.Filled.FavoriteBorder
                    Icon(icon, contentDescription = "")
                },
            )
        }
    }
}

```

Kotlin

Ostatnim elementem jest pokazanie snackbaru w momencie kliknięcia elementu listy. Jest to troszkę podchwytliwe, ponieważ cała obsługa snackbarów w Compose jest zrobiona za pomocą korutyn i funkcja `showSnackbar` jest suspendowalna. Musimy stworzyć nasz własny `coroutineScope`, który zostanie wykorzystany do pokazywania snackbara po kliknięciu elementu.

```

val snackbarHostState = remember { SnackbarHostState() }
val coroutineScope = rememberCoroutineScope()
// ...
onSongClick = {
    coroutineScope.launch {
        snackbarHostState.showSnackbar(
            "You selected: ${it.trackName} by ${it.artist}"
        )
    }
}

```

Kotlin
