**Jetpack Compose**

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

**Deklaratywny UI**

Dla osób, które miały jakąkolwiek styczność z [Flutterem](https://programistabyc.pl/category/flutter/?ref=programistabyc.pl), czy React Native to koncepcja powinna być znana. W deklaratywnym 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?**

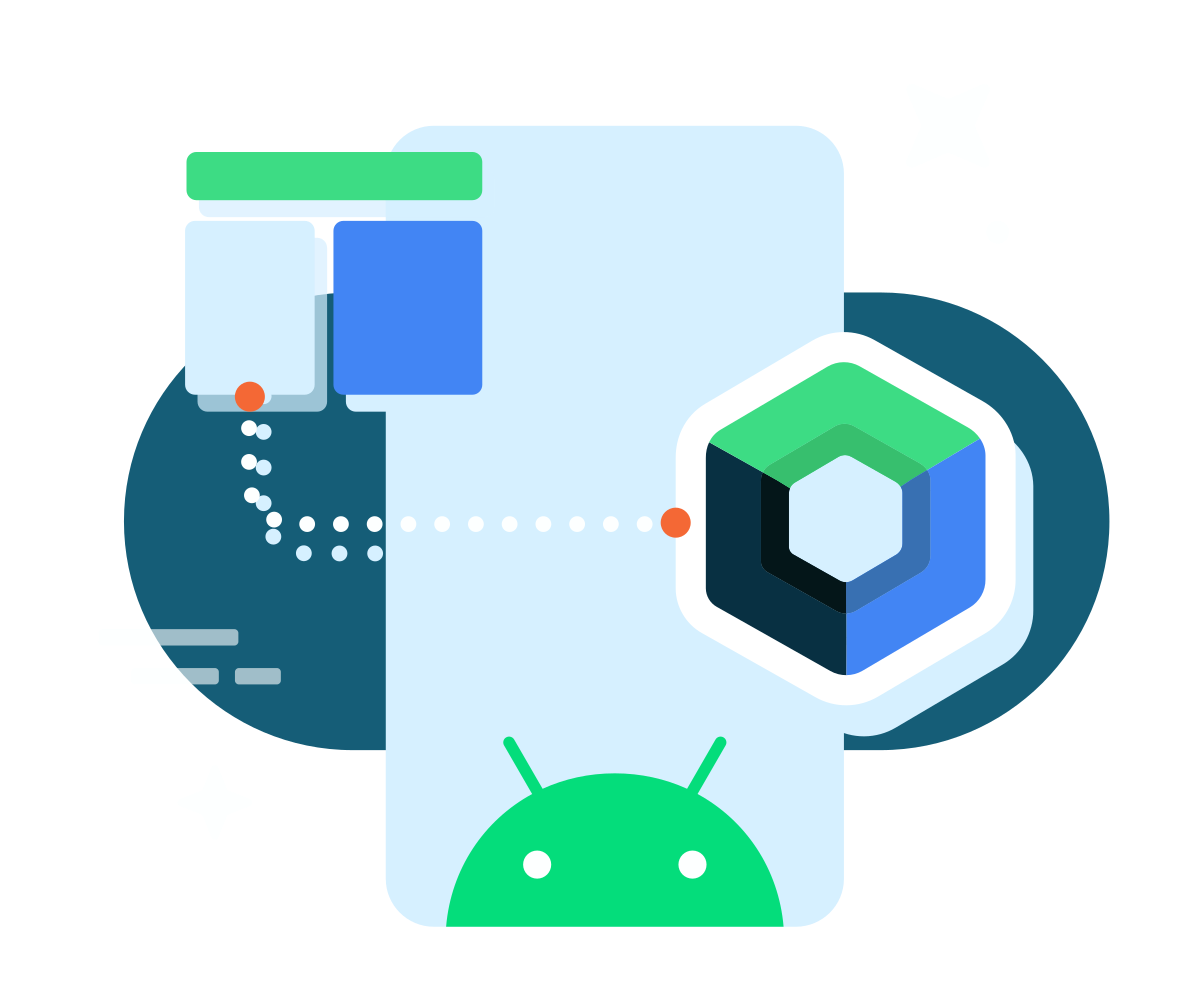
Interfejs użytkownika tworzymy w Compose za pomocą zbioru funkcji oznaczonych annotacją @Composable, które przyjmują model danych i na jego podstawie renderują elementy widoku.

Compose wykorzystuje **Unidirectional Date 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 funkcje 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 zmienią 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](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary?ref=programistabyc.pl#Scaffold(androidx.compose.ui.Modifier,androidx.compose.material.ScaffoldState,kotlin.Function0,kotlin.Function0,kotlin.Function1,kotlin.Function0,androidx.compose.material.FabPosition,kotlin.Boolean,kotlin.Function1,kotlin.Boolean,androidx.compose.ui.graphics.Shape,androidx.compose.ui.unit.Dp,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,kotlin.Function1))

**TopAppBar/BottomAppBar**

Górny/dolny pasek nawigacyjny.

Dokumentacja: [TopAppBar](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary?ref=programistabyc.pl#TopAppBar(kotlin.Function0,androidx.compose.ui.Modifier,kotlin.Function0,kotlin.Function1,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.unit.Dp)) i [BottomAppBar](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary?ref=programistabyc.pl#BottomAppBar(androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Shape,androidx.compose.ui.unit.Dp,androidx.compose.foundation.layout.PaddingValues,kotlin.Function1)).

**Box**

Można go porównać do FrameLayout znanego z XML. To kontener na inne elementy.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary?hl=en&ref=programistabyc.pl#Box(androidx.compose.ui.Modifier,androidx.compose.ui.Alignment,kotlin.Boolean,kotlin.Function1))

**Surface**

Można porównać do CardView znanego z XML.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary?ref=programistabyc.pl#Surface(androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Shape,androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Color,androidx.compose.foundation.BorderStroke,androidx.compose.ui.unit.Dp,kotlin.Function0))

**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](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary?ref=programistabyc.pl#Column(androidx.compose.ui.Modifier,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,kotlin.Function1)), [Row](https://developer.android.com/reference/kotlin/androidx/compose/foundation/layout/package-summary?ref=programistabyc.pl#Row(androidx.compose.ui.Modifier,androidx.compose.foundation.layout.Arrangement.Horizontal,androidx.compose.ui.Alignment.Vertical,kotlin.Function1)), [LazyColumn](https://developer.android.com/reference/kotlin/androidx/compose/foundation/lazy/package-summary?ref=programistabyc.pl#LazyColumn(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Vertical,androidx.compose.ui.Alignment.Horizontal,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,kotlin.Function1)), [LazyRow](https://developer.android.com/reference/kotlin/androidx/compose/foundation/lazy/package-summary?ref=programistabyc.pl#LazyRow(androidx.compose.ui.Modifier,androidx.compose.foundation.lazy.LazyListState,androidx.compose.foundation.layout.PaddingValues,kotlin.Boolean,androidx.compose.foundation.layout.Arrangement.Horizontal,androidx.compose.ui.Alignment.Vertical,androidx.compose.foundation.gestures.FlingBehavior,kotlin.Boolean,kotlin.Function1)).

**Button**

Komponent do przycisków. Jest kilka różnych gotowych wersji, np. OutlinedButton, TextButton.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary?ref=programistabyc.pl#button)

**Text**

Za jego pomocą wyświetlimy tekst.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/material/package-summary?ref=programistabyc.pl#Text(androidx.compose.ui.text.AnnotatedString,androidx.compose.ui.Modifier,androidx.compose.ui.graphics.Color,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.font.FontStyle,androidx.compose.ui.text.font.FontWeight,androidx.compose.ui.text.font.FontFamily,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.style.TextDecoration,androidx.compose.ui.text.style.TextAlign,androidx.compose.ui.unit.TextUnit,androidx.compose.ui.text.style.TextOverflow,kotlin.Boolean,kotlin.Int,kotlin.Int,kotlin.collections.Map,kotlin.Function1,androidx.compose.ui.text.TextStyle))

**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](https://developer.android.com/jetpack/compose/modifiers?ref=programistabyc.pl)

**Podstawowe Modifiery**

**fillMaxSize / fillMaxWidth / fillMaxHeight**

Za ich pomocą definiujemy wielkość danego komponentu.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier?hl=en&ref=programistabyc.pl#(androidx.compose.ui.Modifier).fillMaxSize(kotlin.Float)).

**padding**

Padding umożliwia utworzenie pustych przestrzeni wokół komponentu. Co ważne, w łańcuchu wywołań może być wykorzystany kilkukrotnie. Każdy padding będzie się wtedy odnosić do innych elementów widoku. Pamiętaj, **kolejność Modifierów jest bardzo ważna**.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier?hl=en&ref=programistabyc.pl#(androidx.compose.ui.Modifier).padding(androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp,androidx.compose.ui.unit.Dp))

**clickable**

Composable będzie klikalny i automatycznie uzyskamy ripple effect.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier?hl=en&ref=programistabyc.pl#(androidx.compose.ui.Modifier).clickable(kotlin.Boolean,kotlin.String,androidx.compose.ui.semantics.Role,kotlin.Function0))

**background**

Ustawisz dla Composable background. Jest kilka opcji: Color/Brush, Shape, alpha.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier?hl=en&ref=programistabyc.pl#(androidx.compose.ui.Modifier).background(androidx.compose.ui.graphics.Color,androidx.compose.ui.graphics.Shape))

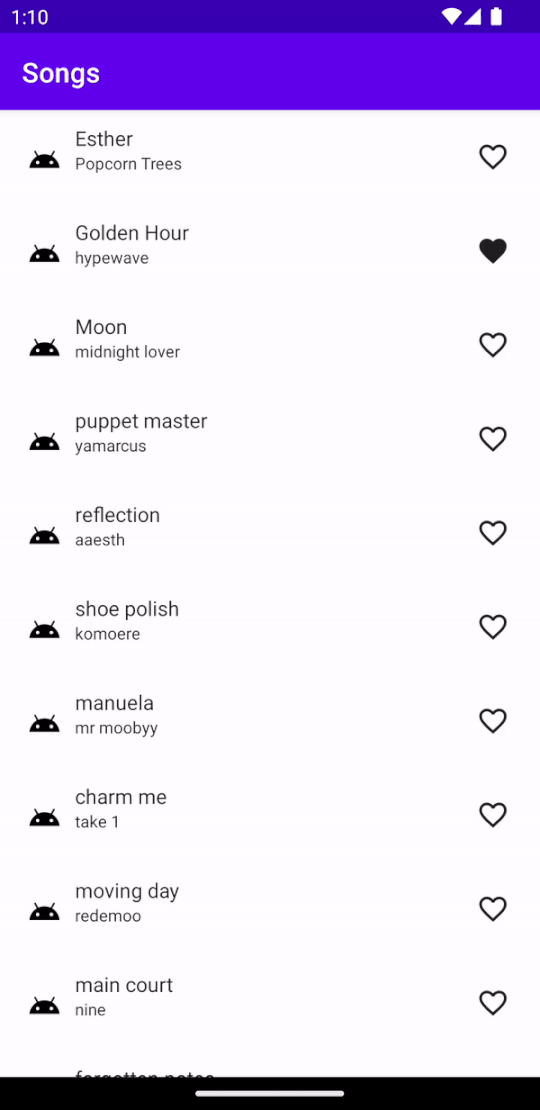
**border**

Composable będzie miał obwódkę z podanym BorderStroke oraz Shape.  
[Dokumentacja](https://developer.android.com/reference/kotlin/androidx/compose/ui/Modifier?hl=en&ref=programistabyc.pl#(androidx.compose.ui.Modifier).border(androidx.compose.foundation.BorderStroke,androidx.compose.ui.graphics.Shape))

**Demonstracyjna aplikacja**

Przeanalizujmy 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:

Demo

**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 {

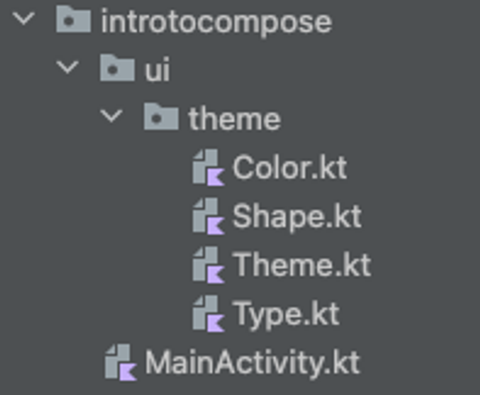
Greeting("Android")

}

}

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.material.Typography

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 przekażemy 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 typustateless, 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