Jetpack Compose Over Inheritance

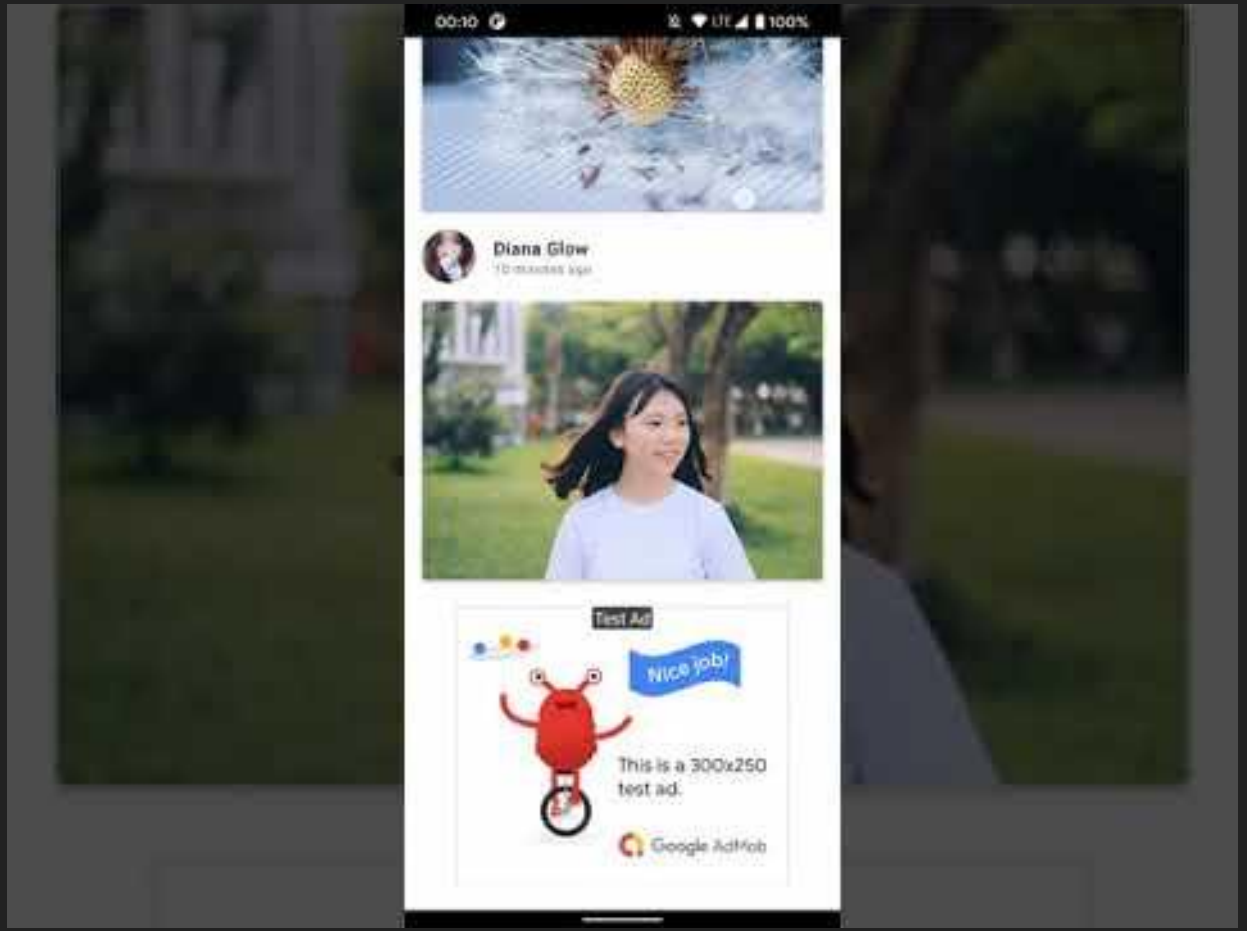
Andrey Kulikov
Google

Anastasia Soboleva
Google

android

Learning by creating an application

Based on 0.1.0-dev14



github.com/andkulikov/compose-photoapp

Agenda

What is Jetpack Compose

Modifiers

Scrollable Layouts

Data Streams

State

Saved instance state

Function's Lifecycle

Custom Layout

Theming

Animations

Preview

Android interop

What is Jetpack Compose

Jetpack Compose

Less Code

Intuitive

Accelerate Development

Powerful

UI widget is a function

```
@Composable
fun Button(
    onClick: () -> Unit,
    ...
    text: @Composable () -> Unit
) {
    // implementation of the Button
}
```

UI widget is a function

```
@Composable
fun Button(
    onClick: () -> Unit,
    ...
    text: @Composable () -> Unit
) {
    // implementation of the Button
}

Button(onClick = {}, shape = CircleShape) {}
```

‘Composable’ annotation

```
@Composable
fun Button(
    onClick: () -> Unit,
    ...
    text: @Composable () -> Unit
) {
    // implementation of the Button
}

Button(onClick = {}, shape = CircleShape) {}
```

Composition

```
@Composable
fun Button(
    onClick: () -> Unit,
    ...
    text: @Composable () -> Unit
) {
    // implementation of the Button
}

Button(onClick = {}, shape = CircleShape) {
    Text("Follow")
}
```

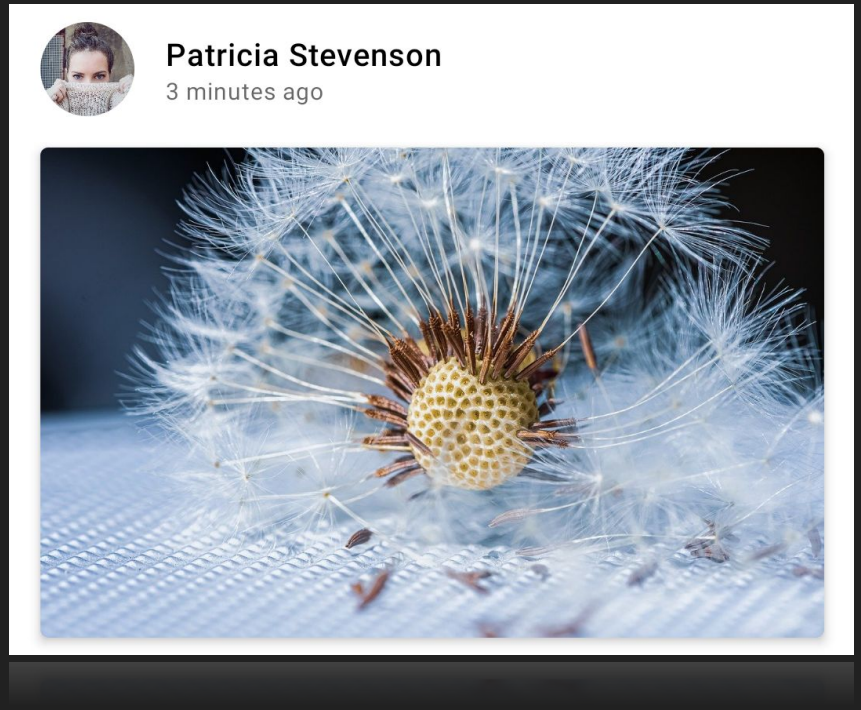
Composition

```
@Composable
fun Button(
    onClick: () -> Unit,
    ...
    text: @Composable () -> Unit
) {
    // implementation of the Button
}

Button(onClick = {}, shape = CircleShape) {
    Text("Follow")
}
```

Entry point to Compose world

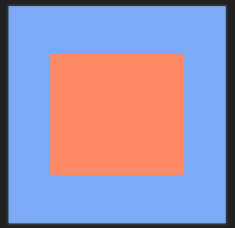
```
class MainActivity : ComponentActivity() {


    override fun onCreate(savedInstanceState: Bundle?) {
        setContent {
            // Compose world
        }
    }
}
```

Photographer Card

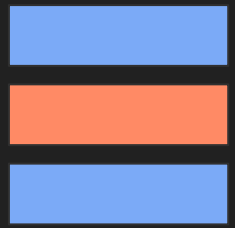


android

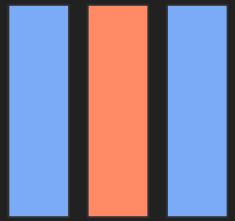
Standard layout components



Stack



Column

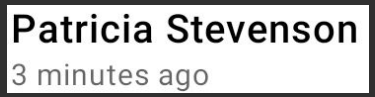


Row

Constraint Layout*

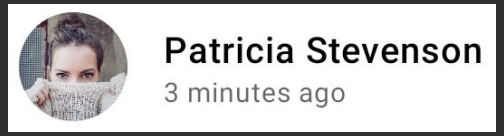
Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    Column {
        Text(photographer.name)
        Text(photographer.lastSeenOnline)
    }
}
```



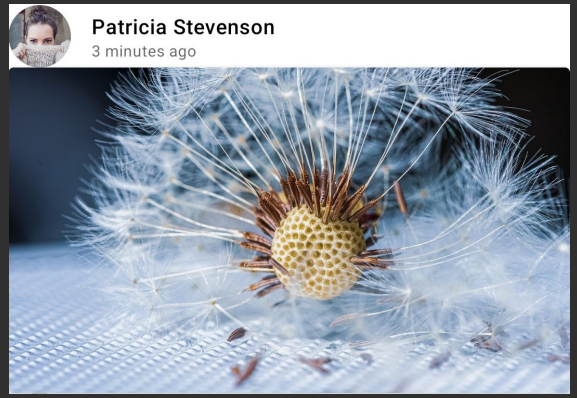
Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    Row(verticalGravity = Alignment.CenterVertically) {
        Image(...)
        Column {
            Text(photographer.name)
            Text(photographer.lastSeenOnline)
        }
    }
}
```



Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    Column {
        Row(verticalGravity = Alignment.CenterVertically) {
            Image(...)
            Column {
                Text(photographer.name)
                Text(photographer.lastSeenOnline)
            }
        }
        Card(elevation = 4.dp) {
            FadeInImage(...)
        }
    }
}
```

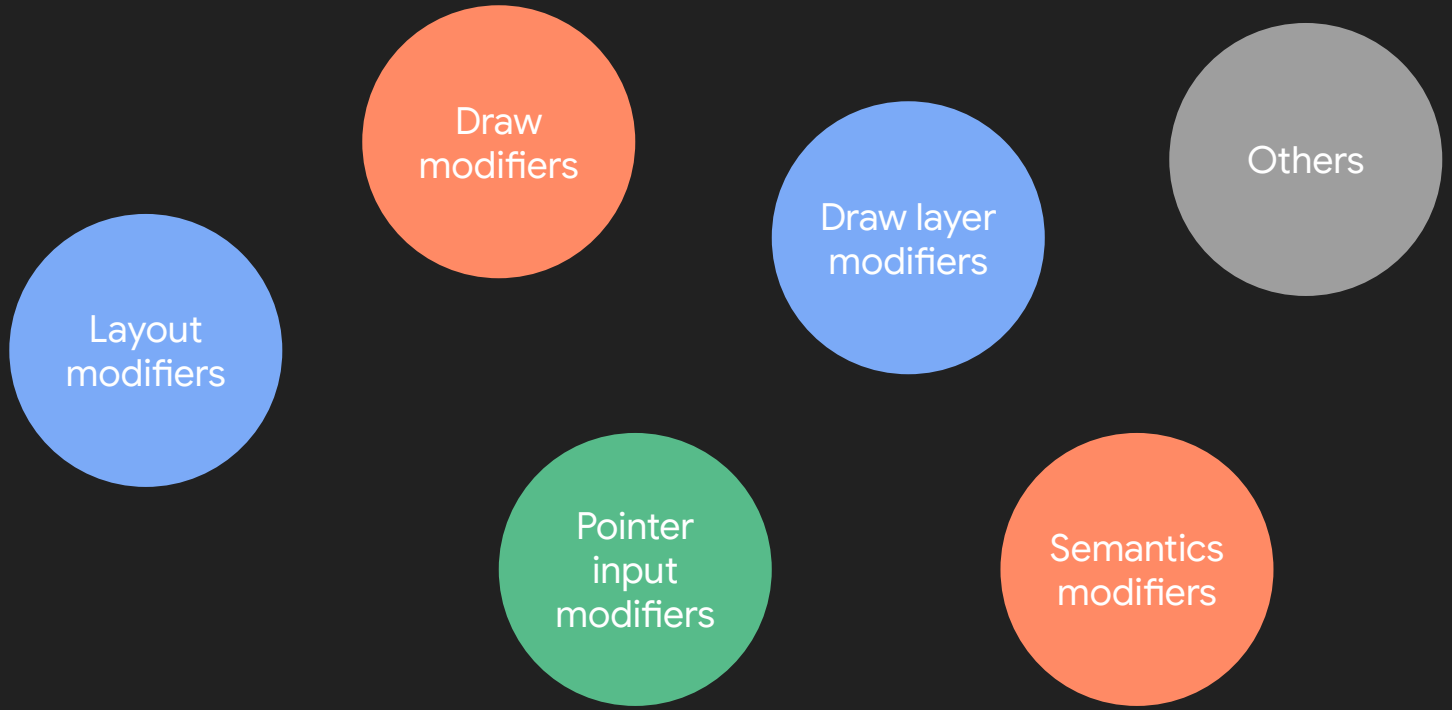


Modifiers

for fine tuning

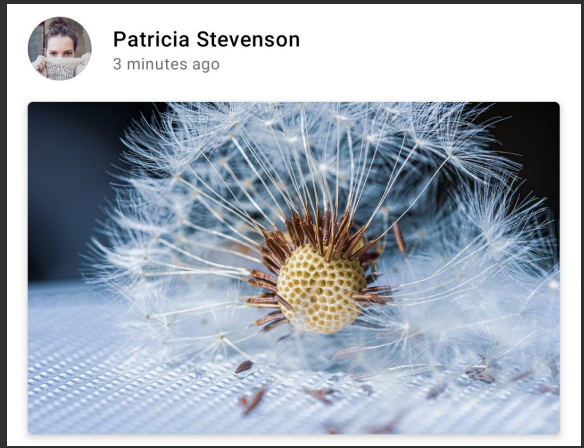
android

Types of modifiers



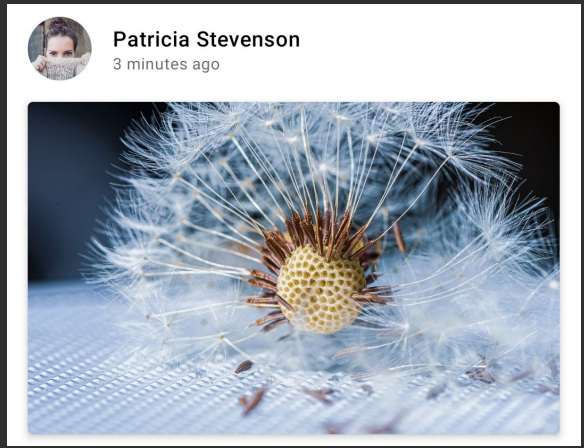
Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        Row(verticalGravity = Alignment.CenterVertically) { …
}
        Spacer(Modifier.size(padding))
        Card(elevation = 4.dp) { … }
    }
}
```



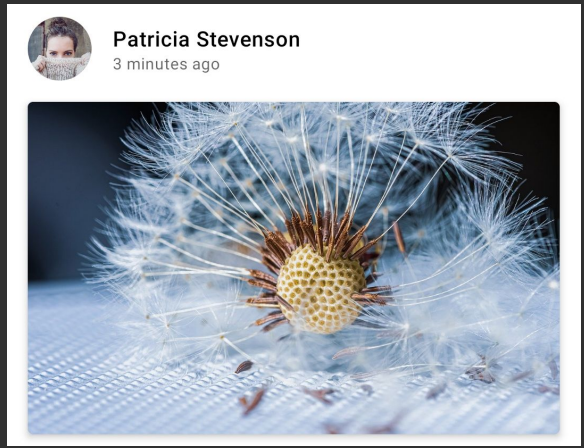
Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        Row(verticalGravity = Alignment.CenterVertically) { …
}
        Spacer(Modifier.size(padding))
        Card(elevation = 4.dp) { … }
    }
}
```



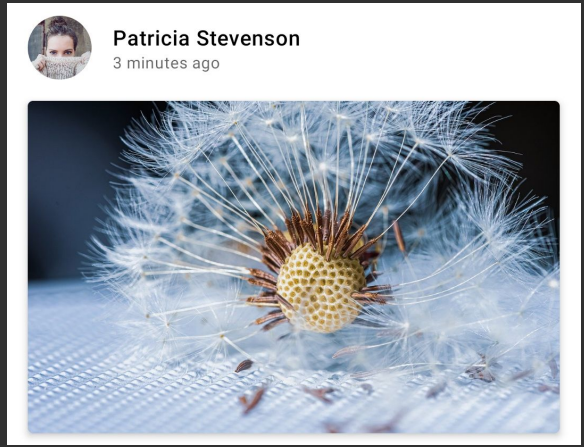
Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        Row(verticalGravity = Alignment.CenterVertically) { …
}
        Spacer(Modifier.size(padding))
        Card(elevation = 4.dp) { … }
    }
}
```



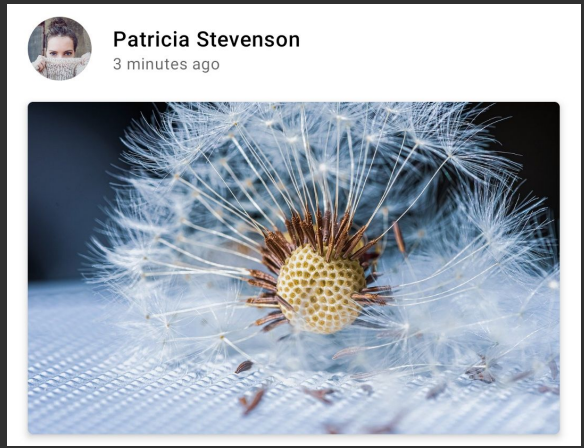
Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        Row(verticalGravity = Alignment.CenterVertically) { …
}
        Spacer(Modifier.size(padding))
        Card(elevation = 4.dp) { … }
    }
}
```



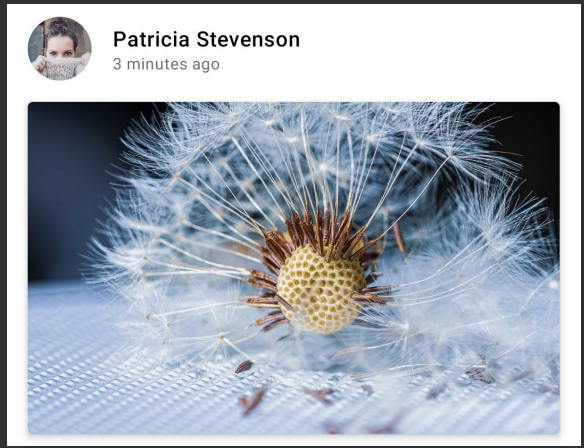
Photographer Card

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        Row(verticalGravity = Alignment.CenterVertically) { …
}
        Spacer(Modifier.size(padding))
        Card(elevation = 4.dp) { … }
    }
}
```



Photographer Card

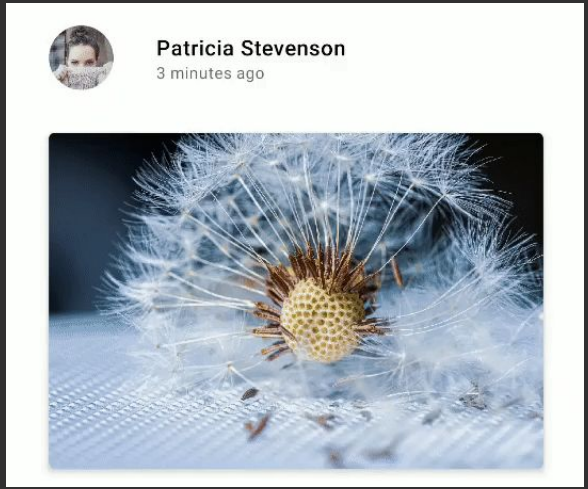
```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        Row(verticalGravity = Alignment.CenterVertically) { …
}
        Spacer(Modifier.size(padding))
        Card(elevation = 4.dp) { … }
    }
}
```



Order of modifiers matters!

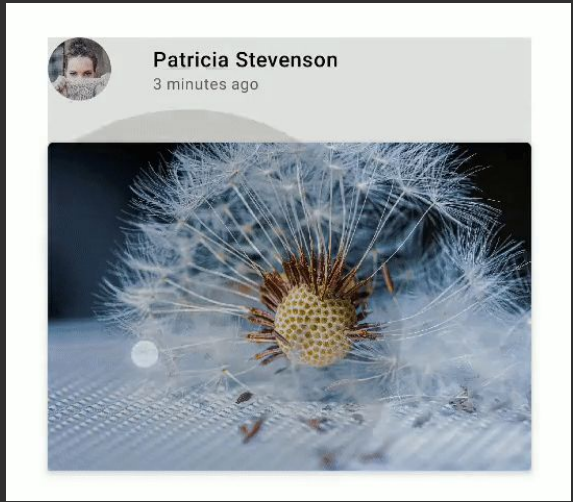
Order of modifiers

```
@Composable
fun PhotographerCard(...) {
    val padding = 16.dp
    Column(
        modifier
            .clickable(onClick = onClick)
            .padding(padding)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```



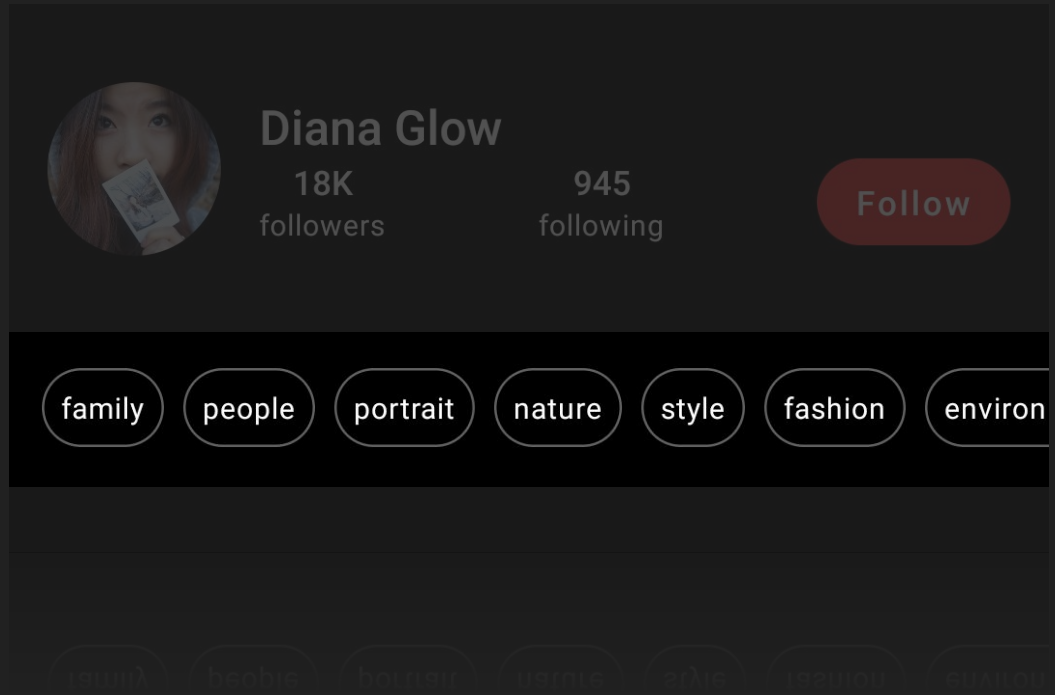
Order of modifiers

```
@Composable
fun PhotographerCard(...) {
    val padding = 16.dp
    Column(
        modifier
            .padding(padding)
            .clickable(onClick = onClick)
            .fillMaxWidth()
    ) {
        // rest of the implementation
    }
}
```

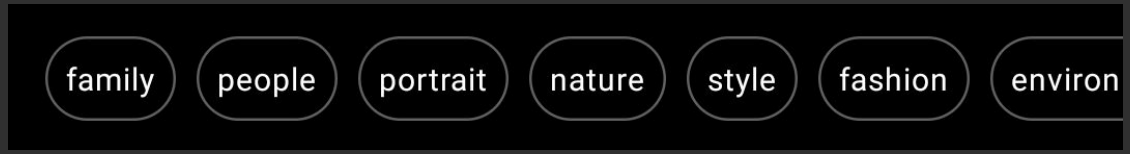


Scrollable Layouts

Tags view

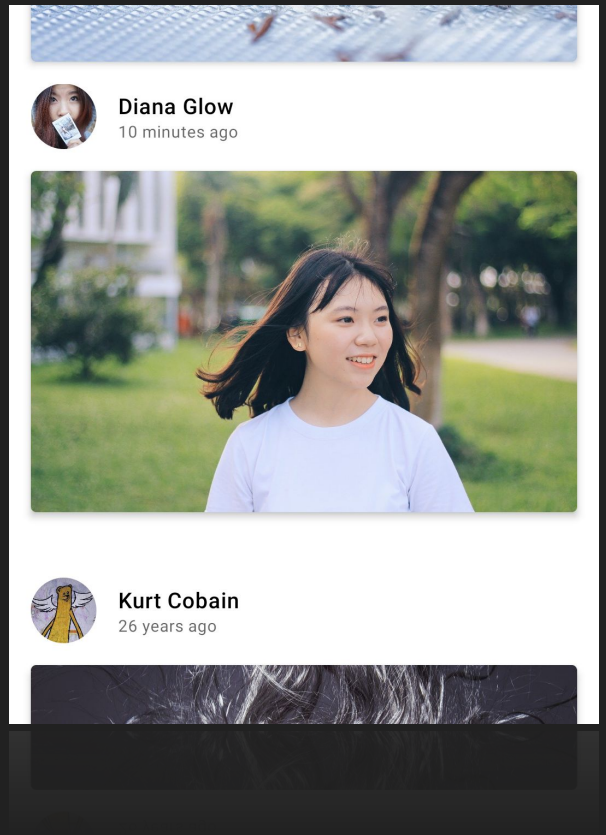

Diana Glow
18K
followers
945
following
Follow
family
people
portrait
nature
style
fashion
environ


```
@Composable
private fun TagsList(tags: List<String>, modifier: Modifier = Modifier) {
    val padding = 8.dp
    Row(Modifier.padding(padding)) {
        tags.forEach {
            Text(...)
            Spacer(Modifier.size(padding))
        }
    }
}
```




```
@Composable
private fun TagsList(tags: List<String>, modifier: Modifier = Modifier) {
    HorizontalScroller(modifier = modifier) {
        val padding = 8.dp
        tags.forEach {
            Text(...)
            Spacer(Modifier.size(padding))
        }
    }
}
```

Feed view



android

Vertical Scroller

```
@Composable
fun Feed(feedItems: List<Photographer>, onSelected: (Photographer) -> Unit) {
    Surface(Modifier.fillMaxSize()) {
        ...
        VerticalScroller {
           feedItems.forEach {
                when (it) {
                    is FeedItem.Header -> FeedHeader()
                    is FeedItem.Ad -> AdBanner()
                    is FeedItem.PhotographerCard -> PhotographerCard(...)
                }
            }
        }
    }
}
```

Lazy Column Items

```
@Composable
fun Feed(feedItems: List<Photographer>, onSelected: (Photographer) -> Unit) {
    Surface(Modifier.fillMaxSize()) {
        ...
        LazyColumnItems(feedItems) { item ->
            when (item) {
                is FeedItem.Header -> FeedHeader()
                is FeedItem.Ad -> AdBanner()
                is FeedItem.PhotographerCard -> PhotographerCard(...)
            }
        }
    }
}
```

Agenda

What is Jetpack Compose

Modifiers

Scrollable Layouts

= вы находитесь здесь =

Data Streams

State

Saved instance state

Function's Lifecycle

Custom Layout

Theming

Animations

Preview

Android interop

Data Streams

Compose Adapters for Data Streams

LiveData

RxJava

Flow

ViewModel with StateFlow

```
class PhotographersViewModel : ViewModel() {
    private val _photographers =
        MutableStateFlow<List<Photographer>>(emptyList())
    val photographers: StateFlow<List<Photographer>> = _photographers
}
```

Flow Adapter

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {

            val viewModel = viewModel<PhotographersViewModel>()

            val photographers by viewModel.photographers.collectAsState()

            Feed(photographers)
            ...
        }

    }
}
```

LiveData and RxJava

```
val liveData: LiveData<String> = ...
val value: String? by liveData.observeAsState()


val observable: Observable<String> = …
val value: String by observable.subscribeAsState(initial = "start")
```

State

Add state

```
@Composable
fun foo() {
    var selectedId : <String?> = null
}
```

Add state

```
@Composable
fun foo() {
    val selectedIdState = state<String?> { null }
}
```

Add state

```
@Composable
fun foo() {
    val selectedIdState: MutableState<String?> = state<String?> { null }
}
```

Add state

```
@Composable
fun foo() {
    val selectedIdState: MutableState<String?> = state<String?> { null }
}

interface MutableState<T> : State<T> {
    override var value: T
}

interface State<T> {
    val value: T
}
```

Add state

```
@Composable
fun foo() {
    val selectedIdState: MutableState<String?> = state<String?> { null }
    var selectedId by state<String?> { null }
}
```

Updating the state

```
setContent {
    val viewModel = viewModel<PhotographersViewModel>()
    val photographers by photographersFlow.collectAsState()
    var selectedId by state<String?> { null }
    Feed(viewModel.photographers, onSelected = {
        selectedId = it.id
    })
}
```

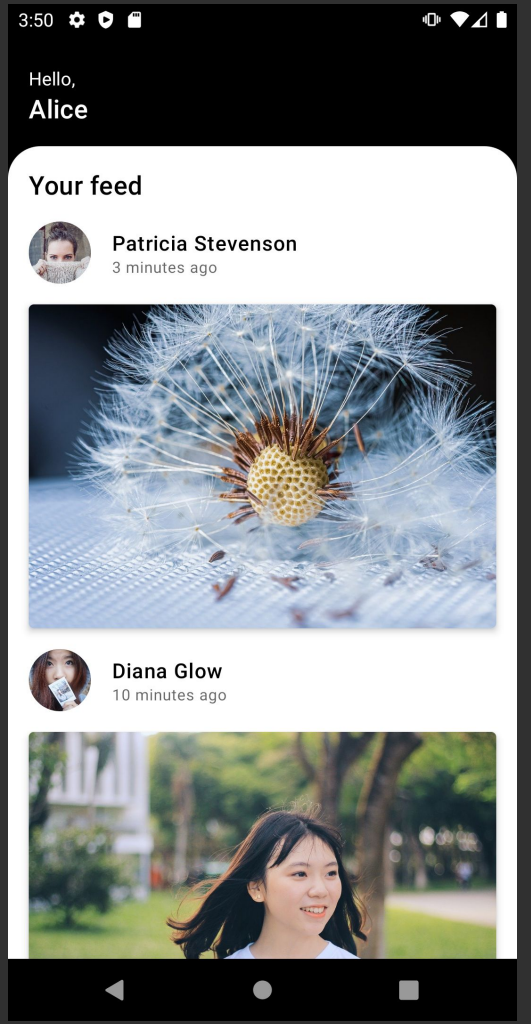
Updating the state

```
@Composable
fun Feed(
    photographersFlow: StateFlow<List<Photographer>>,
    onSelected: (Photographer) -> Unit
) {
    val photographers by photographersFlow.collectAsState()
    LazyColumnItems(photographers) { photographer ->
        PhotographerCard(
            photographer = photographer,
            onClick = { onSelected(photographer) }
        )
    }
}
```

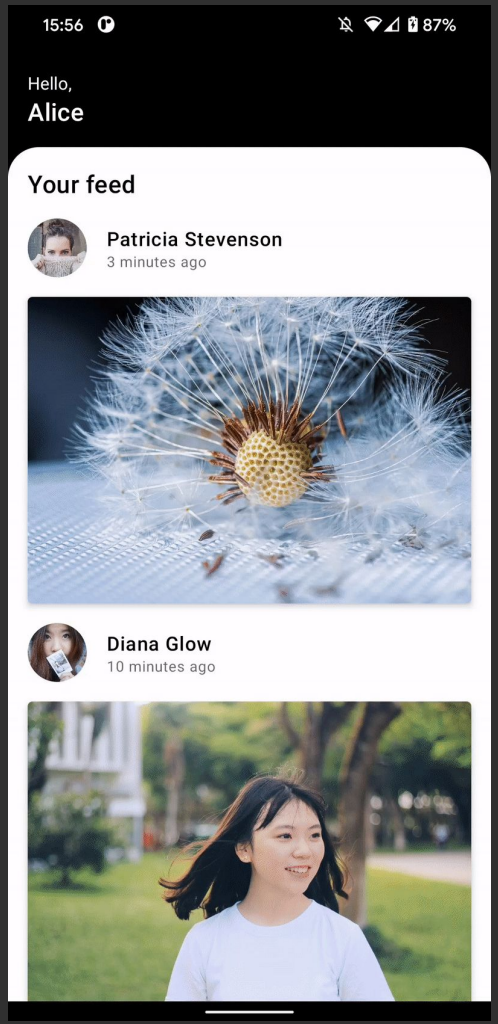
```
@Composable
fun Feed(
   photographersFlow: StateFlow<List<Photographer>>,
   onSelected: (Photographer) -> Unit
) {
    val photographers by photographersFlow.collectAsState()
    LazyColumnItems(photographers) { photographer ->
        PhotographerCard(
           photographer = photographer,
           onClick = { onSelected(photographer) }
        )
    }
}
```

```
@Composable
fun PhotographerCard(
    photographer: Photographer,
    onClick: () -> Unit
) {
    Column(
        Modifier.clickable(onClick = onClick)
    ) {
        ...
    }
}
```

```
var selectedId by state<String?> { null }
Feed(photographers, onSelected = {
    selectedId = it.id
})
```



```
var selectedId by state<String?> { null }
if (selectedId == null) {
    Feed(photographers, onSelected = {
        selectedId = it.id
    })
} else {
    Profile(viewModel.getById(selectedId))
}
```



Saved Instance State

Pre-Compose

```
class MainActivity : ComponentActivity() {
    private var selectedId : String? = null
}
```

```
class MainActivity : ComponentActivity() {
    private val SELECTED_ID_KEY = "selected_id_key"
    private var selectedId : String? = null
}
```

```
class MainActivity : ComponentActivity() {
    private val SELECTED_ID_KEY = "selected_id_key"
    private var selectedId : String? = null

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        outState.putString(SELECTED_ID_KEY, selectedId)
    }
}
```

```
class MainActivity : ComponentActivity() {
    private val SELECTED_ID_KEY = "selected_id_key"
    private var selectedId : String? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        if (savedInstanceState != null) {
            selectedId = savedInstanceState.getString(SELECTED_ID_KEY)
        }
    }

    override fun onSaveInstanceState(outState: Bundle) {
        super.onSaveInstanceState(outState)
        outState.putString(SELECTED_ID_KEY, selectedId)
    }
}
```

Compose approach

```
var selectedId by state<String?> { null }
```

Compose approach

```
var selectedId by ~~state~~<String?> { null }

var selectedId by savedInstanceState<String?> { null }
```

Compose approach

```
var stringState by savedInstanceState<String?> { null }

var intState by savedInstanceState<Int> { 0 }

var floatState by savedInstanceState<Float> { 0.0 }

...
```

Compose approach

```
data class User(val name: String, val age: Int)

var user = savedInstanceState { User("Jim", 30)}
```

Compose approach

```
@Parcelize
data class User(val name: String, val age: Int) : Parcelable

var user = savedInstanceState { User("Jim", 30)}
```

Compose approach

```
data class User(val name: String, val age: Int)

val userSaver = listSaver<User, Any>(
    save = { listOf(it.name, it.age) },
    restore = { User(it[0] as String, it[1] as Int) }
)

var user = savedInstanceState(saver = userSaver) { User("Jim", 30)}
```

Function's Lifecycle

Pre-Compose

```
class MainActivity : ComponentActivity() {

    private var selectedId : String? = null

    override fun onBackPressed() {
        if (selectedId != null) {
            selectedId = null
        } else {
            super.onBackPressed()
        }
    }
}
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        var selectedId : String? = null
        val goToFeedCallback = object : OnBackPressedCallback(false) {
            override fun handleOnBackPressed() {
                selectedId = null
            }
        }
        ...
        onBackPressedDispatcher.addCallback(goToFeedCallback)
    }
}
```

Official solutions for the navigation is not ready yet

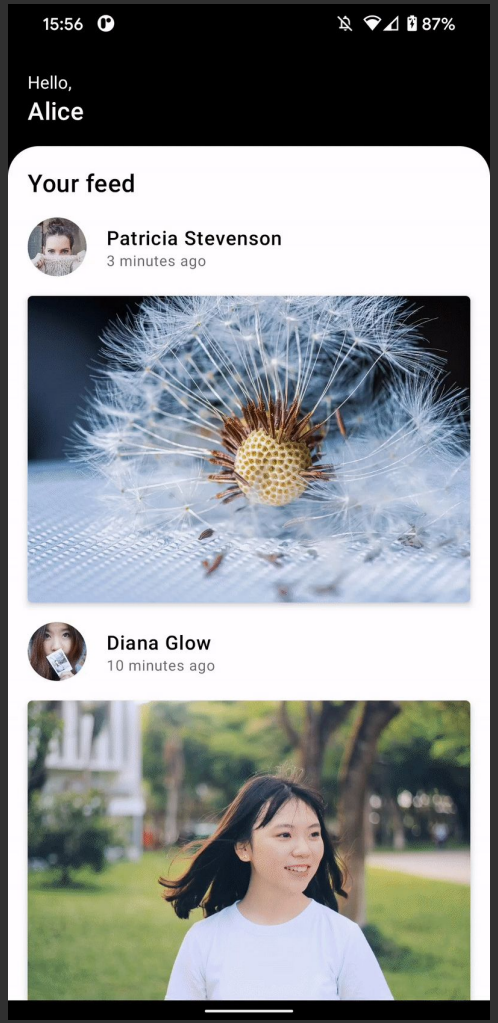
Compose

```
@Composable
fun onBackPressed(dispatcher: OnBackPressedDispatcher, callback: () -> Unit) {
    ...
}
```

```
@Composable
fun onBackPressed(dispatcher: OnBackPressedDispatcher, callback: () -> Unit) {
    onActive {
        val backPressedCallback = object : OnBackPressedCallback(true) {
            override fun handleOnBackPressed() {
                callback.invoke()
            }
        }
        dispatcher.addCallback(backPressedCallback)
    }
}
```

```
@Composable
fun onBackPressed(dispatcher: OnBackPressedDispatcher, callback: () -> Unit) {
    onActive {
        val backPressedCallback = object : OnBackPressedCallback(true) {
            override fun handleOnBackPressed() {
                callback.invoke()
            }
        }
        dispatcher.addCallback(backPressedCallback)
        onDispose {
            backPressedCallback.remove()
        }
    }
}
```

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val viewModel by viewModels<PhotographersViewModel>()
        setContent {
            var selectedId by state<String?> { null }
            if (selectedId == null) {
                Feed(viewModel.photographers, onSelected = {
                    selectedId = it.id
                })
            } else {
                Profile(viewModel.getById(selectedId))
                onBackPressed(onBackPressedDispatcher) {
                    selectedId = null
                }
            }
        }
    }
}
```



Agenda

What is Jetpack Compose

Modifiers

Scrollable Layouts

Data Streams

State

Saved instance state

Function's Lifecycle

= вы находитесь здесь =

Custom Layout

Theming

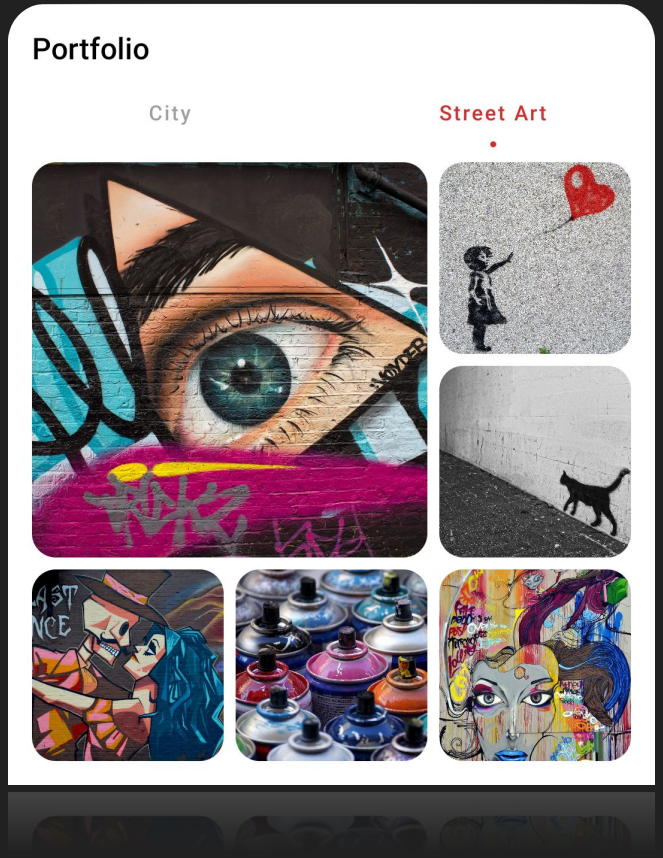
Animations

Preview

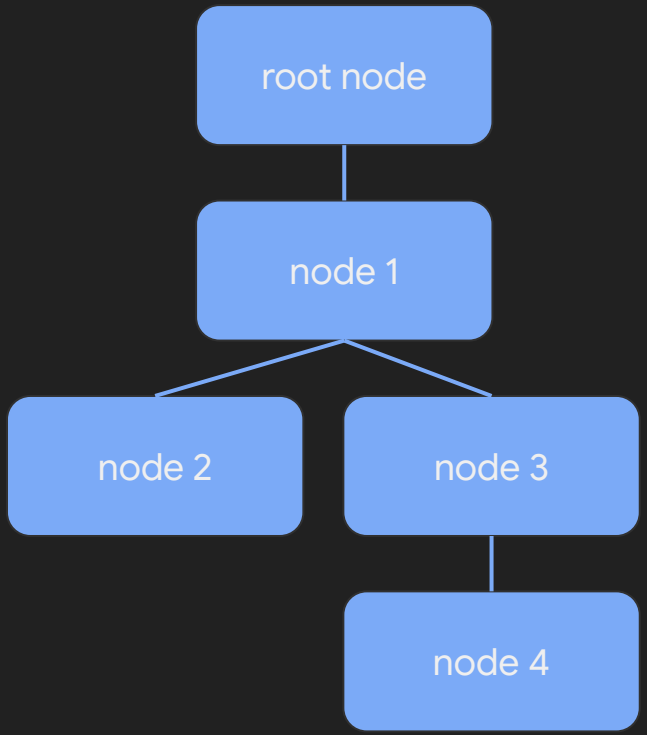
Android interop

Custom Layout

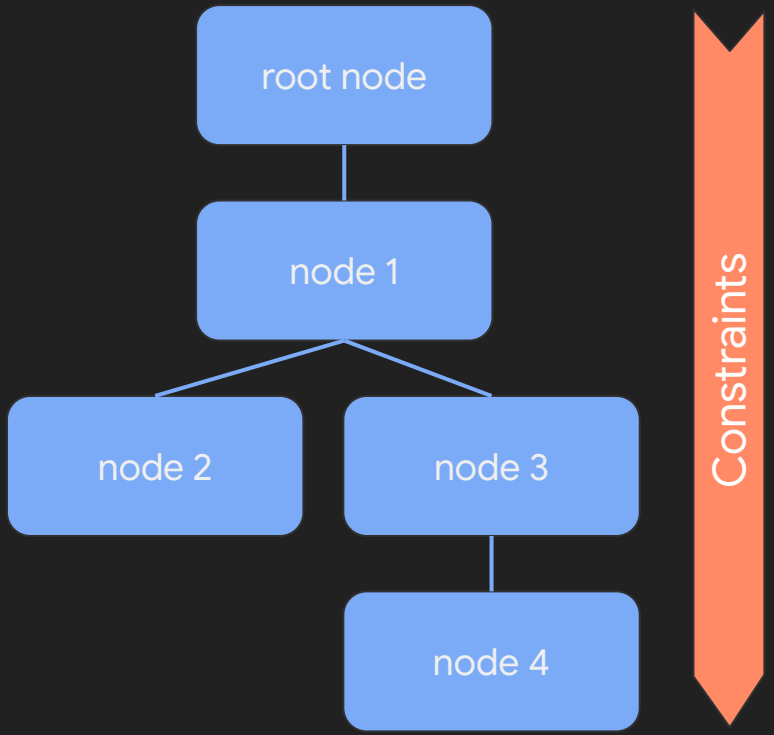
Portfolio screen



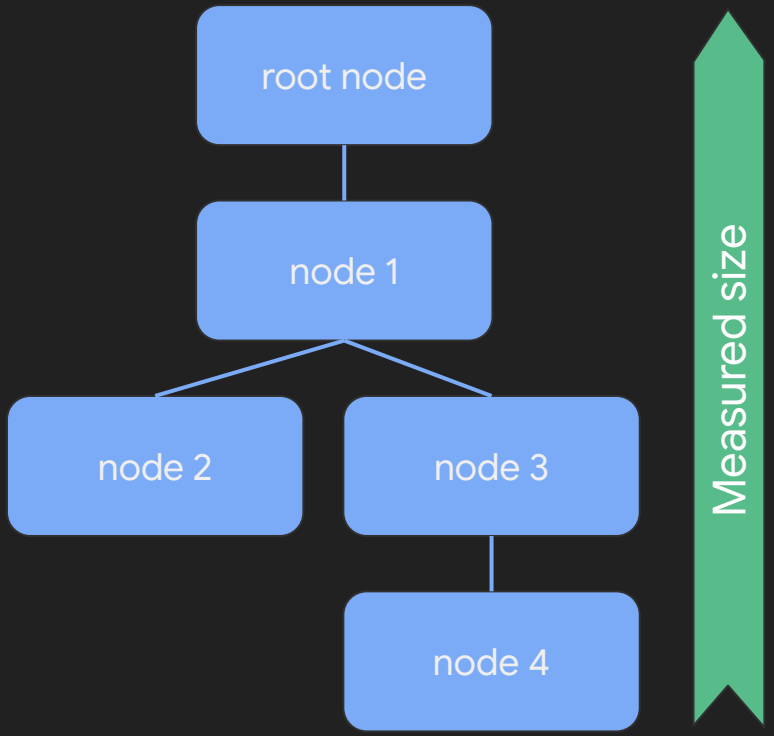
Layout System



Layout System



Layout System



Layout()

```
@Composable
fun Layout(
    children: @Composable () -> Unit,
    modifier: Modifier = Modifier,
    measureBlock: MeasureScope.(List<Measurable>, Constraints) ->
                                                MeasureScope.MeasureResult
) {
    // emits Layout Node
}
```

Layout()

```
Layout(
    children = children,
    modifier = modifier,
) { measurables, constraints ->

    // measure children and define the size

    layout(width, height) {
        // place children
    }
}
```

Layout()

```
Layout(
    children = children,
    modifier = modifier,
) { measurables, constraints ->

    // measure children and define the size

    layout(width, height) {
        // place children
    }
}
```

Layout()

```
Layout(
    children = children,
    modifier = modifier,
) { measurables, constraints ->

    // measure children and define the size

    layout(width, height) {
        // place children
    }
}
```

Photos Grid

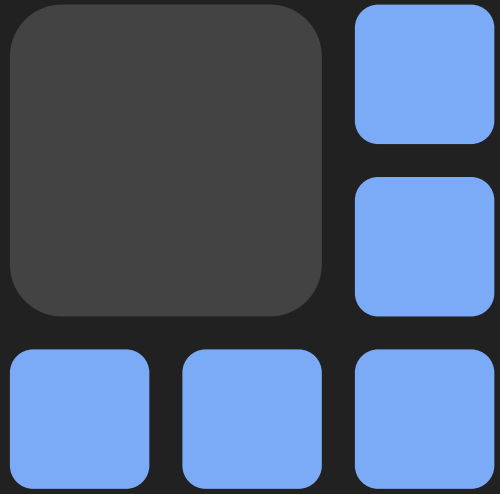
```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...)
}
```

Step 1: add children to layout

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(
        children = {
            require(images.size >= 6) { "Requires 6 photos for the grid." }
            images.subList(0, 6).forEach {
                Image(it, Modifier.aspectRatio(1f).clip(RoundedCornerShape(16.dp)))
            }
        },
        modifier = modifier
    ) { measurables, constraints ->
        ...
    }
}
```

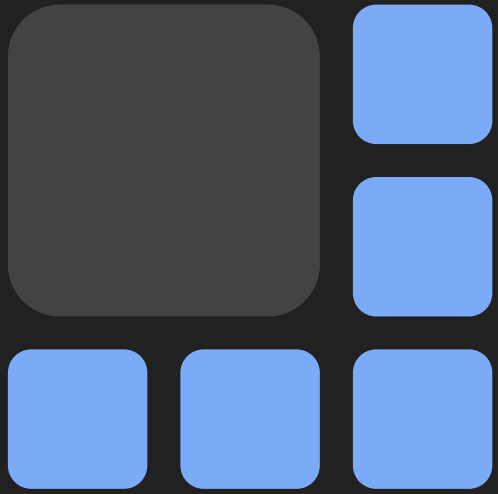
Step 2: measure children

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        val padding = 8.dp.toIntPx()
        val minDimension = min(constraints.maxHeight, constraints.maxWidth)

        val smallImageConstraints = constraints.copy(
            minWidth = (minDimension - padding * 2) / 3,
            maxWidth = (minDimension - padding * 2) / 3
        )
        val placeables = measurables
            .subList(fromIndex = 1, toIndex = measurables.size)
            .map {
                it.measure(smallImageConstraints)
            }
         ...
      }
}
```



Step 2: measure children

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        val padding = 8.dp.toIntPx()
        val minDimension = min(constraints.maxHeight, constraints.maxWidth)

        val smallImageConstraints = constraints.copy(
            minWidth = (minDimension - padding * 2) / 3,
            maxWidth = (minDimension - padding * 2) / 3
        )
        val placeables = measurables
            .subList(fromIndex = 1, toIndex = measurables.size)
            .map {
                it.measure(smallImageConstraints)
            }
        ...
    }
}
```



Step 2: measure children

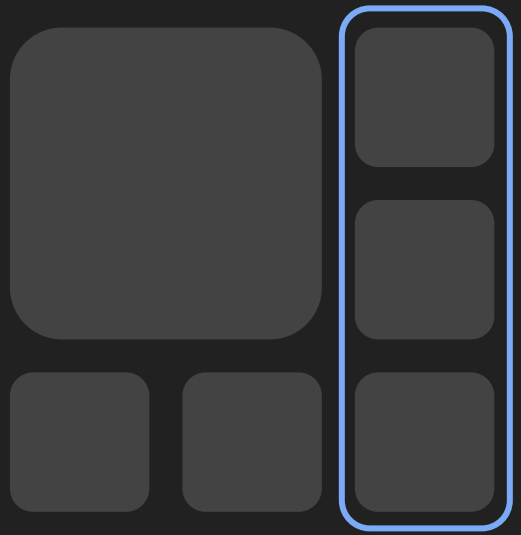
```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        ...

        val bigImageConstraints = constraints.copy(
            minWidth = minDimension - padding - placeables[0].width,
            maxWidth = minDimension - padding - placeables[0].width
        )
        val bigImagePlaceable = measurables.first().measure(bigImageConstraints)

        ...
    }
}
```

Step 2: measure children

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        ...

        val bigImageConstraints = constraints.copy(
            minWidth = minDimension - padding - placeables[0].width,
            maxWidth = minDimension - padding - placeables[0].width
        )
        val bigImagePlaceable = measurables.first().measure(bigImageConstraints)

        ...
    }
}
```

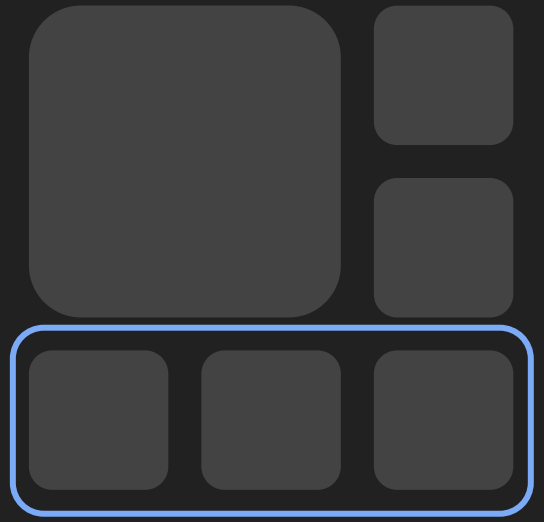
Step 3: define size of layout

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        ...

        // calculate size of the layout
        val height = placeables[0].height * 3 + padding * 2
        val width = placeables[0].width * 3 + padding * 2

        layout(width, height) {
          ...
        }
    }
}
```



Step 3: define size of layout

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        ...

        // calculate size of the layout
        val height = placeables[0].height * 3 + padding * 2
        val width = placeables[0].width * 3 + padding * 2

        layout(width, height) {
          ...
        }
    }
}
```



Step 4: position children

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        ...
        layout(width, height) {
            var positionX = 0.ipx
            var positionY = 0.ipx

            bigImagePlaceable.place(positionX, positionY)

             ...
        }
}}
```

Step 4: position children

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        ...
        layout(width, height) {
            ...

            placeables.forEachIndexed { index, placeable ->
                if (index < 2) { // to the right from the big image
                    placeable.place(bigImagePlaceable.width + padding, positionY)
                    positionY += placeable.height + padding
                } else { // bottom row
                    placeable.place(positionX, positionY)
                    positionX += placeable.width + padding
                }
            }
        }
}}
```

Step 4: position children

```
@Composable
fun PhotosGrid(images: List<Int>, modifier: Modifier = Modifier) {
    Layout(...) { measurables, constraints ->
        ...
        layout(width, height) {
            ...

            placeables.forEachIndexed { index, placeable ->
                if (index < 2) { // to the right from the big image
                    placeable.place(bigImagePlaceable.width + padding, positionY)
                    positionY += placeable.height + padding
                } else { // bottom row
                    placeable.place(positionX, positionY)
                    positionX += placeable.width + padding
                }
            }
        }
}}
```

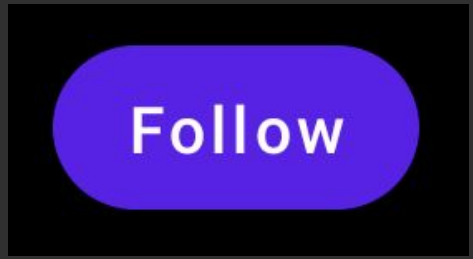
Result

Automatically supports right-to-left direction

Theming

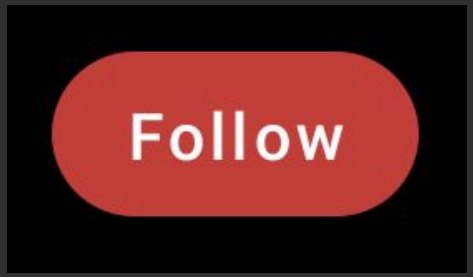
Material Theme

```
class MainActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MaterialTheme {
                ...
            }
        }
    }
}
```



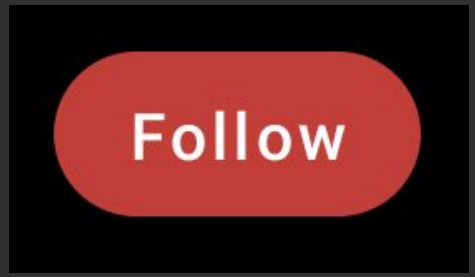
Custom theme

```
@Composable
fun PhotoAppTheme(content: @Composable () -> Unit) {
    val colors = lightColorPalette(
        primary = Color(0xffd32f2f)
    )
    MaterialTheme(colors = colors, content = content)
}
```



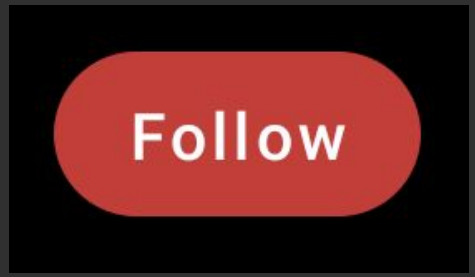
Custom theme

```
@Composable
fun PhotoAppTheme(content: @Composable () -> Unit) {
    val primary = Color(0xffd32f2f)
    val lightColors = lightColorPalette(
        primary = primary
    )
    val darkColors = darkColorPalette(
        primary = primary,
        onPrimary = Color.White
    )
    val colors = if (isSystemInDarkTheme()) darkColors
        else lightColors
    MaterialTheme(colors = colors, content = content)
}
```



Custom theme

```
@Composable
fun PhotoAppTheme(content: @Composable () -> Unit) {
    val primary = Color(0xffd32f2f)
    val lightColors = lightColorPalette(
        primary = primary
    )
    val darkColors = darkColorPalette(
        primary = primary,
        onPrimary = Color.White
    )
    val colors = if (isSystemInDarkTheme()) darkColors
        else lightColors
    MaterialTheme(colors = colors, content = content)
}
```



Other parameters

```
@Composable
fun MaterialTheme(
    colors: ColorPalette = MaterialTheme.colors,
    typography: Typography = MaterialTheme.typography,
    shapes: Shapes = MaterialTheme.shapes,
    content: @Composable () -> Unit
) { ... }
```

Using values in your component

```
@Composable
fun foo() {
    MaterialTheme.colors.primary

    MaterialTheme.typography.h3

    MaterialTheme.shapes.small

}
```

Animations

Animated float

```
@Composable
fun LazyLoadingImage(id: Int, modifier: Modifier = Modifier)   {
   val image = loadImageResource(id = id).resource.resource
   if (image == null) {
       Spacer(modifier = modifier)
   } else {
       Image(image, modifier, contentScale = ContentScale.Crop)
   }
}
```

```
@Composable
fun FadeInImage(id: Int, modifier: Modifier = Modifier) {
    val image = loadImageResource(id = id).resource.resource
    if (image == null) {
        Spacer(modifier = modifier)
    } else {
        val alpha = animatedFloat(0f)
        onCommit(image) {
            alpha.snapTo(0f)
            alpha.animateTo(1f, TweenBuilder<Float>().apply {
                duration = 300
                easing = LinearOutSlowInEasing
            })
        }
        Image(
            image,
            modifier.drawOpacity(alpha.value),
            contentScale = ContentScale.Crop
        )
    }
}
```

```
@Composable
fun FadeInImage(id: Int, modifier: Modifier = Modifier) {
    val image = loadImageResource(id = id).resource.resource
    if (image == null) {
        Spacer(modifier = modifier)
    } else {
        val alpha = animatedFloat(0f)
        onCommit(image) {
            alpha.snapTo(0f)
            alpha.animateTo(1f, TweenBuilder<Float>().apply {
                duration = 300
                easing = LinearOutSlowInEasing
            })
        }
        Image(
            image,
            modifier.drawOpacity(alpha.value),
            contentScale = ContentScale.Crop
        )
    }
}
```

```
@Composable
fun FadeInImage(id: Int, modifier: Modifier = Modifier) {
    val image = loadImageResource(id = id).resource.resource
    if (image == null) {
        Spacer(modifier = modifier)
    } else {
        val alpha = animatedFloat(0f)
        onCommit(image) {
            alpha.snapTo(0f)
            alpha.animateTo(1f, TweenBuilder<Float>().apply {
                duration = 300
                easing = LinearOutSlowInEasing
            })
        }
        Image(
            image,
            modifier.drawOpacity(alpha.value),
            contentScale = ContentScale.Crop
        )
    }
}
```

```
@Composable
fun FadeInImage(id: Int, modifier: Modifier = Modifier) {
    val image = loadImageResource(id = id).resource.resource
    if (image == null) {
        Spacer(modifier = modifier)
    } else {
        val alpha = animatedFloat(0f)
        onCommit(image) {
            alpha.snapTo(0f)
            alpha.animateTo(1f, TweenBuilder<Float>().apply {
                duration = 300
                easing = LinearOutSlowInEasing
            })
        }
        Image(
            image,
            modifier.drawOpacity(alpha.value),
            contentScale = ContentScale.Crop
        )
    }
}
```

```
@Composable
fun FadeInImage(id: Int, modifier: Modifier = Modifier) {
    val image = loadImageResource(id = id).resource.resource
    if (image == null) {
        Spacer(modifier = modifier)
    } else {
        val alpha = animatedFloat(0f)
        onCommit(image) {
            alpha.snapTo(0f)
            alpha.animateTo(1f, TweenBuilder<Float>().apply {
                duration = 300
                easing = LinearOutSlowInEasing
            })
        }
        Image(
            image,
            modifier.drawOpacity(alpha.value),
            contentScale = ContentScale.Crop
        )
    }
}
```

Transition


```
@Composable
private fun TabIndicatorContainer(
    tabPositions: List<TabPosition>,
    selectedIndex: Int,
    indicator: @Composable () -> Unit
) {
    val indicatorOffset = remember { DpPropKey() }
}
```


```
@Composable
private fun TabIndicatorContainer(
    tabPositions: List<TabPosition>,
    selectedIndex: Int,
    indicator: @Composable () -> Unit
) {
    val indicatorOffset = remember { DpPropKey() }
    val transitionDefinition = remember {
        transitionDefinition {
            tabPositions.forEachIndexed { index, position ->
                state(index) {
                    this[indicatorOffset] = (position.left + position.right) / 2
                }
            }
        }
    }
}
```


```
val transitionDefinition = remember(tabPositions) {
    transitionDefinition {
        tabPositions.forEachIndexed { index, position ->
            state(index) {
                this[indicatorOffset] = (position.left + position.right) / 2
            }
        }
        transition {
            indicatorOffset using physics<Dp> {
                dampingRatio = Spring.DampingRatioLowBouncy
                stiffness = Spring.StiffnessLow
            }
        }
    }
}
```

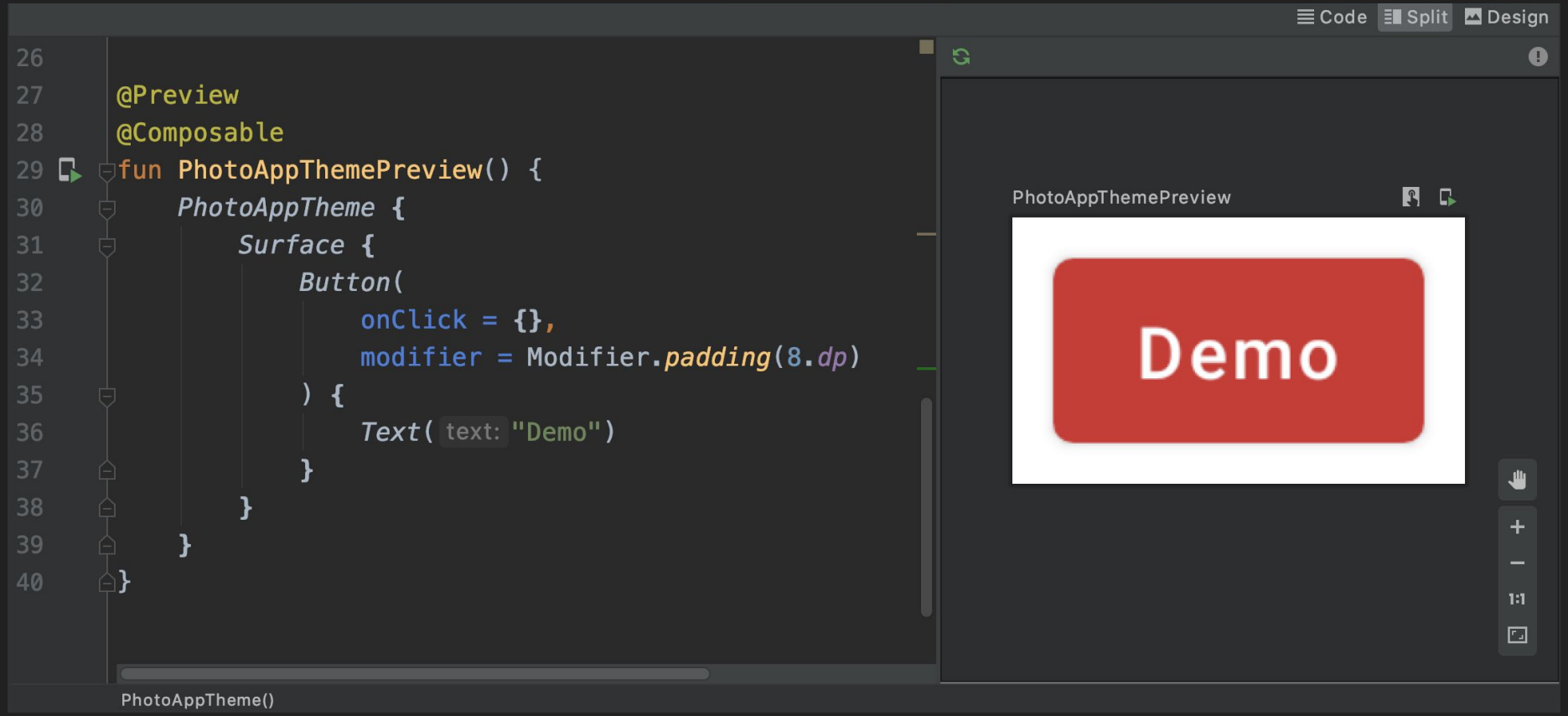

```
val transitionDefinition = remember(tabPositions) {
    transitionDefinition {
        tabPositions.forEachIndexed { index, position ->
            state(index) {
                this[indicatorOffset] = (position.left + position.right) / 2
            }
        }
        transition {
            indicatorOffset using tween<Dp> {
                duration = 300
                easing = FastOutSlowInEasing
            }
        }
    }
}
```

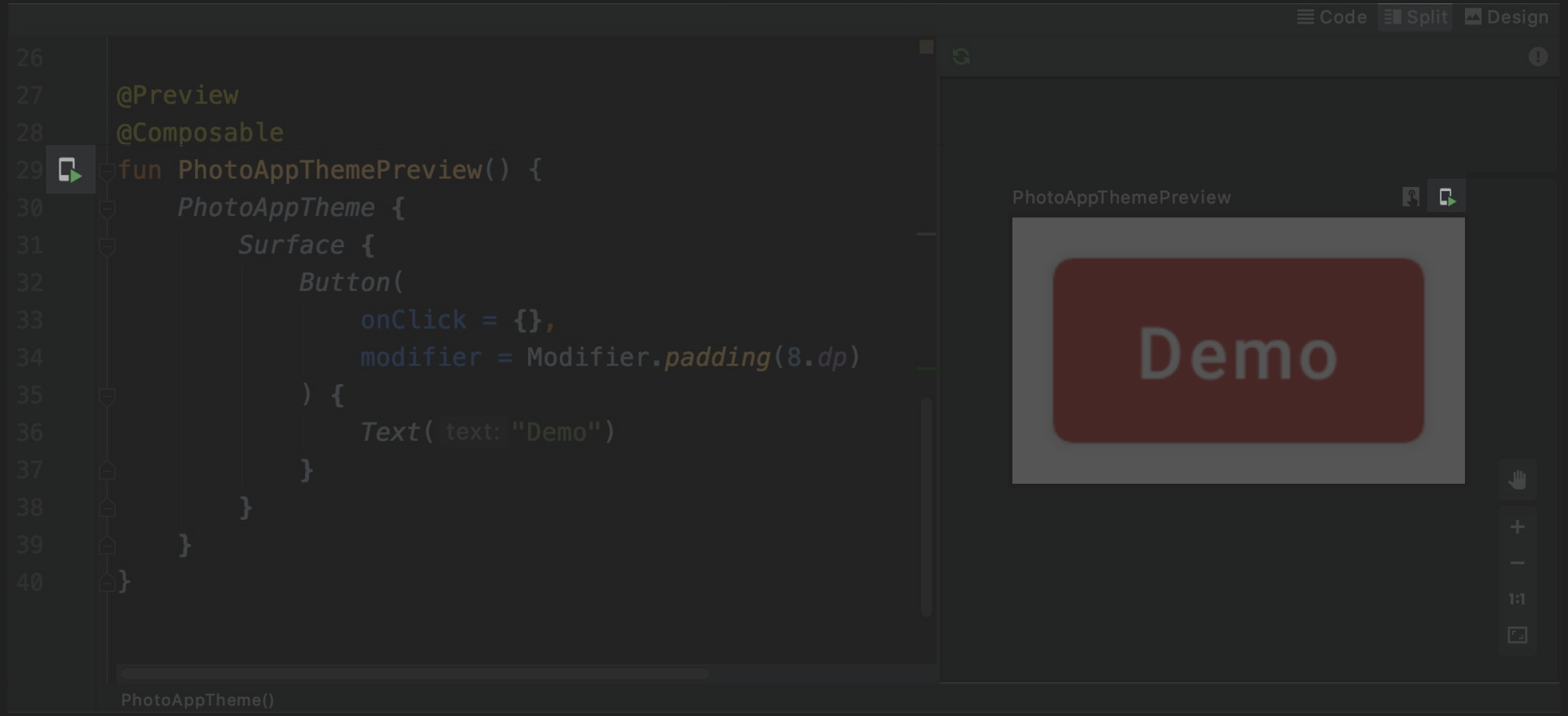

```
@Composable
private fun TabIndicatorContainer(
    tabPositions: List<TabPosition>,
    selectedIndex: Int,
    indicator: @Composable () -> Unit
) {
    ...
    Transition(transitionDefinition, selectedIndex) { state ->
        Box(
            modifier = Modifier
                .fillMaxSize()
                .wrapContentSize(Alignment.BottomStart)
                .offset(x = state[indicatorOffset]),
            children = indicator
        )
    }
}
```


```
@Composable
private fun TabIndicatorContainer(
    tabPositions: List<TabPosition>,
    selectedIndex: Int,
    indicator: @Composable () -> Unit
) {
    ...
    Transition(transitionDefinition, selectedIndex) { state ->
        Box(
            modifier = Modifier
                .fillMaxSize()
                .wrapContentSize(Alignment.BottomStart)
                .offset(x = state[indicatorOffset]),
            children = indicator
        )
    }
}
```

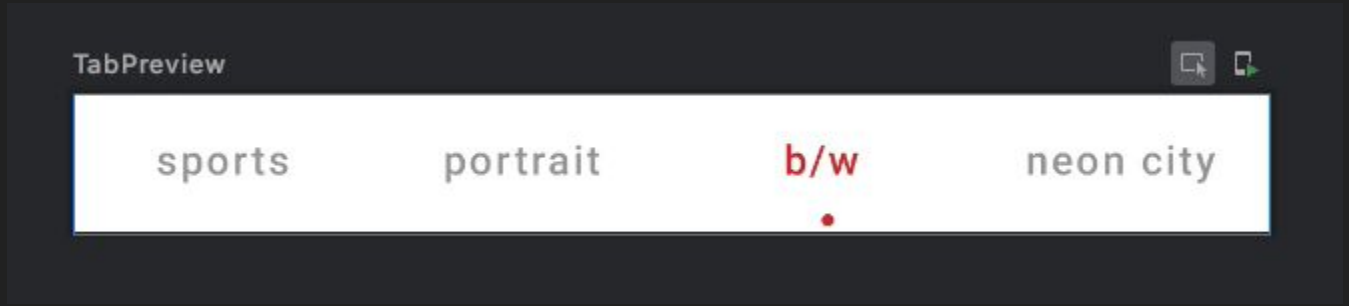

```
@Composable
private fun TabIndicatorContainer(
    tabPositions: List<TabPosition>,
    selectedIndex: Int,
    indicator: @Composable () -> Unit
) {
    ...
    Transition(transitionDefinition, selectedIndex) { state ->
        Box(
            modifier = Modifier
                .fillMaxSize()
                .wrapContentSize(Alignment.BottomStart)
                .offset(x = state[indicatorOffset]),
            children = indicator
        )
    }
}
```

Preview


Code
Split
Design
26
27 @Preview
28 @Composable
29 fun PhotoAppThemePreview() {
30 PhotoAppTheme {
31 Surface {
32 Button(
33 onClick = {},
34 modifier = Modifier.padding(8.dp)
35) {
36 Text(text: "Demo")
37 }
38 }
39 }
40 }
PhotoAppTheme()
PhotoAppThemePreview
Demo
1:1

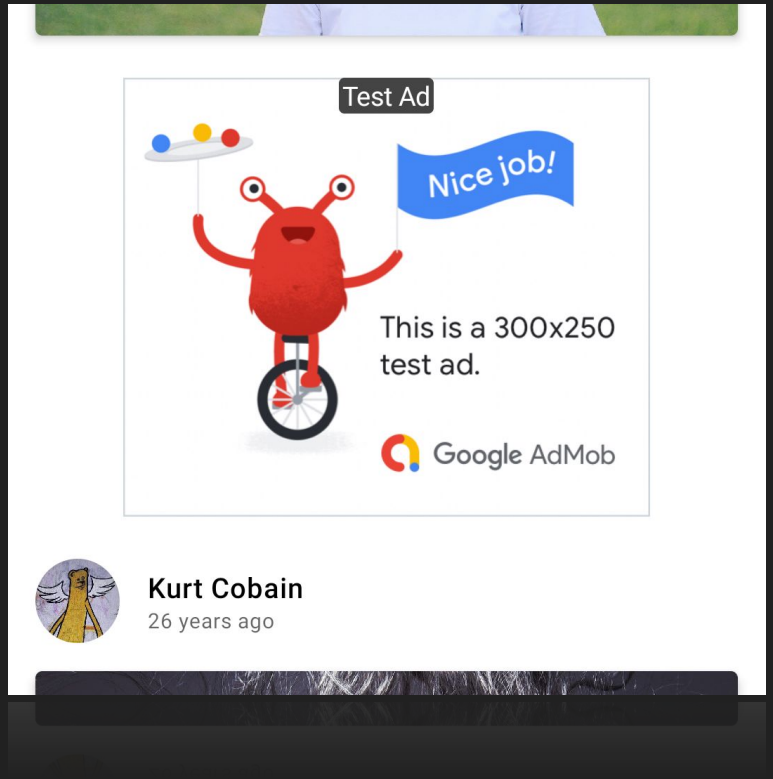

Code
Split
Design
@Preview
@Composable
fun PhotoAppThemePreview() {
PhotoAppTheme {
Surface {
Button(
onClick = {},
modifier = Modifier.padding(8.dp)
) {
Text(text: "Demo")
}
}
}
}
PhotoAppThemePreview
Demo
1:1
PhotoAppTheme()

```
@Preview
@Composable
fun TabPreview() {
    PhotoAppTheme {
        var selectedGroup by state { "b/w" }
        PhotosTab(
            groups = listOf("sports", "portrait", "b/w", "neon city"),
            selectedGroup = selectedGroup,
            onSelected = { selectedGroup = it }
        )
    }
}
```



View interop

Ad Banner


Test Ad
Nice job!
This is a 300x250 test ad.
Google AdMob
Kurt Cobain
26 years ago

Android View in Compose

```
@Composable
fun AndroidView(
    @LayoutRes resId: Int,
    modifier: Modifier = Modifier,
    postInflationCallback: (View) -> Unit = { _ -> }
) { ... }
```

```
@Composable
fun AndroidView(view: View, modifier: Modifier = Modifier) { ... }
```

Ad Banner

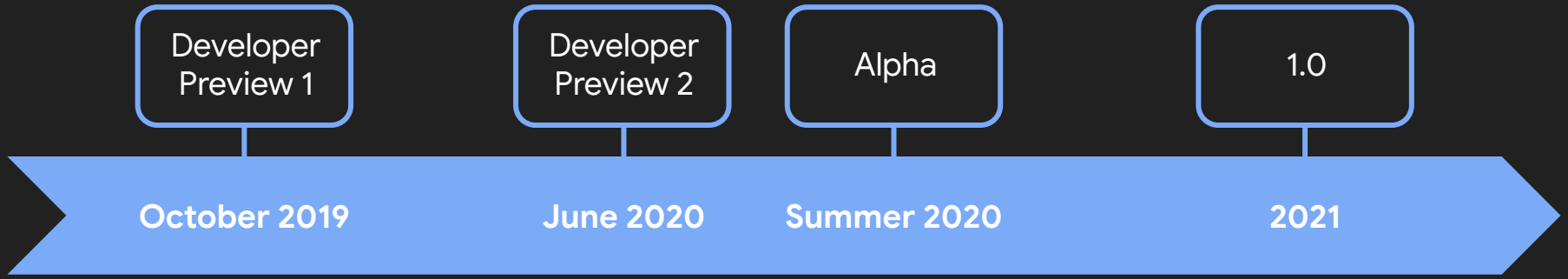
```
@Composable
fun AdBanner() {
    val adView = remember {
        // creates ads view using your advertising library
        createAdView()
    }
    // adds ads view to the Compose
    AndroidView(adView, Modifier.padding(16.dp))
}
```

Compose in Android ViewGroup

```
fun ViewGroup.setContent(
    recomposer: Recomposer,
    content: @Composable () -> Unit
): Composition { ... }
```

Compose it yourself!

Roadmap


Developer Preview 1
Developer Preview 2
Alpha
1.0
October 2019
June 2020
Summer 2020
2021

Links

- PhotoApp source code
  github.com/andkulikov/compose-photoapp
- "Эволюция декларативных UI-фреймворков: От динозавров к Jetpack Compose" by Matvei Malkov
  live.jugru.org/online/100069r10003028
- KotlinLang Slack (#compose channel)
  slack.kotlinlang.org