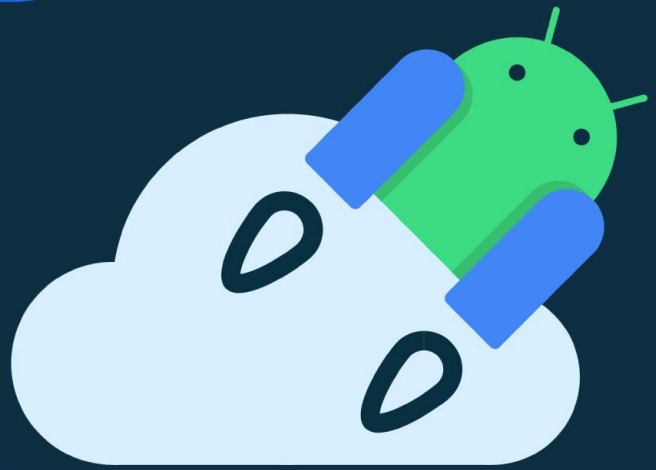


Android for Everyone

Getting Started with Jetpack Compose

Mobile is hard.

Jetpack Compose makes it easier to get started with Android.

I’m Sierra!

Just a senior Android Engineer living in a Jetpack Compose World

What is Jetpack Compose?

Build better apps faster with Jetpack Compose.

Jetpack Compose is Android's modern toolkit for building native UI.

Less Code

Accelerate Development

Intuitive

Powerful

What does “Modern native toolkit” mean?

- Declarative framework

Describe your UI

What does “Modern native toolkit” mean?

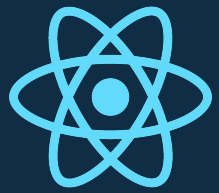
- Declarative framework
- All Kotlin all the time

Can build apps with high levels of complexity and polish

What does “Modern native toolkit” mean?

- Declarative framework
- All Kotlin all the time
- Still in Beta

Just kidding!! It’s officially released as of yesterday!



the land of Declaration

Working with Jetpack Compose

Anatomy of a composable

- @Composable notation
- A pure function that can take in parameters
- A composable can be used multiple times during the app
  - Each time it is used generates a new instance of the composable

Building the Details Screen

The entire screen is a composable

The image is a composable

The lines of text are also composables

Layouts are a breeze

```
@Composable
fun PhotoDetails(
  photo: Photo
) {
 Column {
  PhotoItem(photo = photo)
  Text(text = "Landing date:" +
    photo.rover.landingDate)
  Text(text = "Camera: " +
    photo.camera.fullName)
  Text(text = "Rover:" +
    photo.rover.name)
  }
}
```

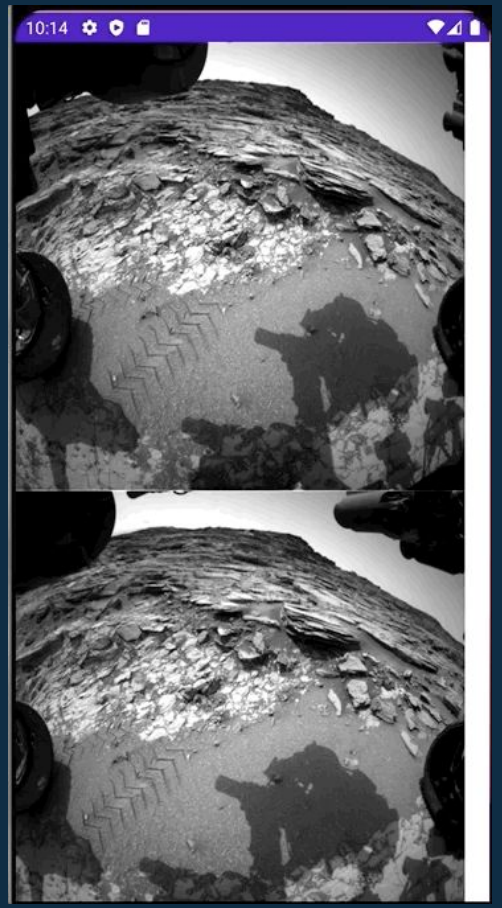
Lists are also breeze

```
Column(modifier = Modifier
 .verticalScroll(rememberScrollState())
) {
    photos.value.forEachIndexed { index,photo ->
        PhotoItem(photo = photo, onClick = {
            photoDetail = index
            showDetails = !showDetails }
        )
    }
}
```

LazyColumn to the rescue!

```
val photos = viewModel
    .photosState
    .collectAsState()

LazyColumn {
    items(photos.value) { photo ->
        PhotoItem(
            photo = photo,
        )
    }
}
```



Okay so how do we start customizing our composables?

Height + width

Background + shape

Modifier

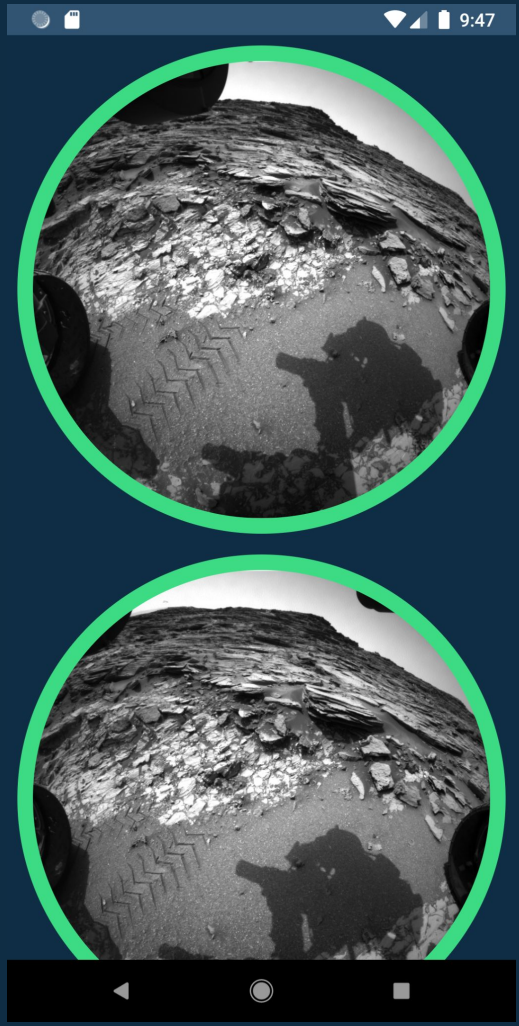
Padding + elevation

Clickable + focusable

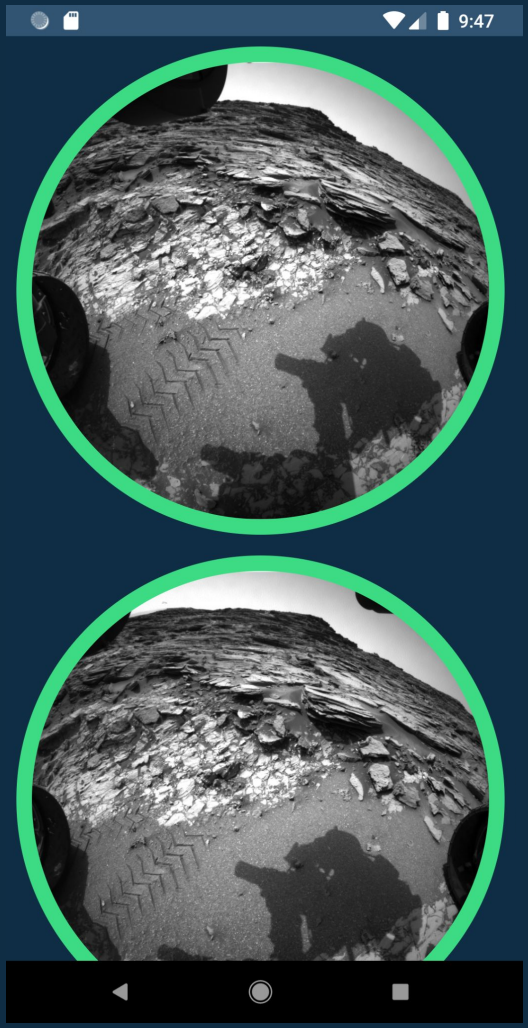
How do we use a modifier?

```
@Composable
fun Image(
    painter: Painter,
    contentDescription: String?,
    modifier: Modifier = Modifier,
    alignment: Alignment = Alignment.Center,
    contentScale: ContentScale = ContentScale.Fit,
    alpha: Float = DefaultAlpha,
    colorFilter: ColorFilter? = null
)
```

Let’s make our app match our presentation



```
LazyColumn(
    modifier = Modifier.fillMaxWidth(),
    horizontalAlignment =
Alignment.CenterHorizontally
) {
    items(photos.value) { photo ->
        Image(
            painter = ourPainter,
            contentDescription =
ourContentDescription,
            modifier = Modifier
                .padding(8.dp)
                .clip(CircleShape)
                .background(color = green)
                .padding(12.dp)
                .clip(CircleShape)
        )
    }
}
```



What about the background?

Theming is easier!

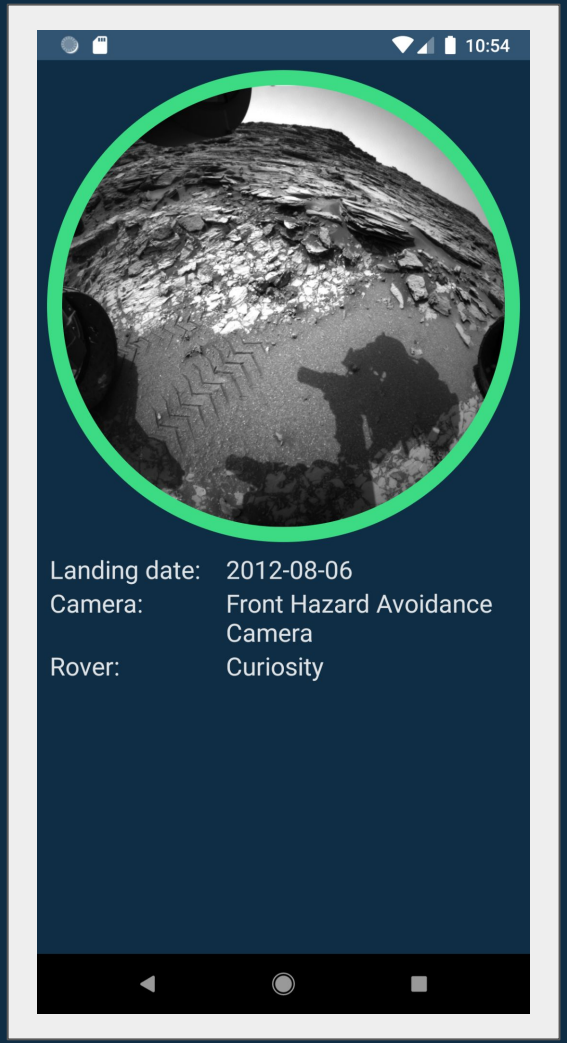
```
setContent {
  MyTheme(
      darkTheme = true
) {
    MyApp { MainScreen() }
  }
}
```

```
@Composable
fun MyTheme(darkTheme: Boolean, content) {
  val colors = if (darkTheme) DarkColorPalette
               else LightColorPalette

  MaterialTheme(
      colors = colors,
      content = content
  )
}
```

What about Constraint Layout?

Let’s revisit our first screen



```
ConstraintLayout {
    val (image, landingLabel, landingText) = createRefs()
    val itemModifier = Modifier

    PhotoItem(
        photo = photo,
        modifier = itemModifier.constrainAs(image) {
            centerHorizontallyTo(parent)
            top.linkTo(parent.top)
        }
    )
    Text(
        text = "Landing date:",
        modifier = itemModifier.constrainAs(landingLabel) {
            start.linkTo(parent.start)
            top.linkTo(image.bottom)
        }.padding(horizontal = 10.dp)
    )
    Text(
        text = photo.rover.landingDate,
        modifier = itemModifier.constrainAs(landingText) {
            start.linkTo(landingLabel.end)
            top.linkTo(image.bottom)
            end.linkTo(parent.end)
            width = Dimension.fillToConstraints
        }.padding(horizontal = 10.dp)
    )
}
```

How do we pull all this together?

State

State

Uni-directional data flow

remember { mutableStateOf() }

State hoisting

```
val photos = viewModel
    .photosState
    .collectAsState()
var showDetails by remember { mutableStateOf(false) }
var photoDetail by remember { mutableStateOf(-1) }

PhotoList(
    photos = photos.value,
    onClick = { index ->
        photoDetail = index
        showDetails = !showDetails
    }
)

if (showDetails)
    PhotoDetails(
        photo = photos.value[photoDetail],
        isShown = { showDetails = false }
    )
```

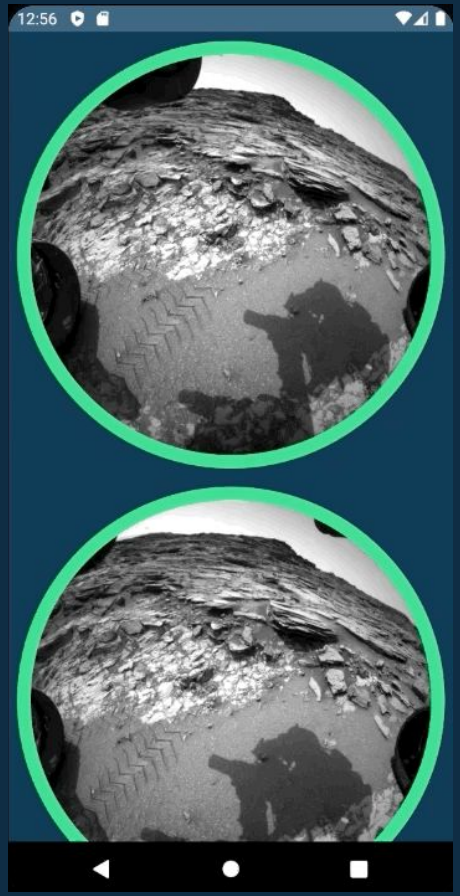
Declare our state variables

Use state in our composables to control UI

Recomposition - what you need to know

- Composable functions can execute in any order
- Composable functions can run in parallel
- Recomposition is smart and optimistic
- Composable functions might run quite frequently

The final product



```
if (showDetails)
    PhotoDetails(
        photo = photos.value[photoDetail],
        isShown = { showDetails = false }
    )
```

What I showed you

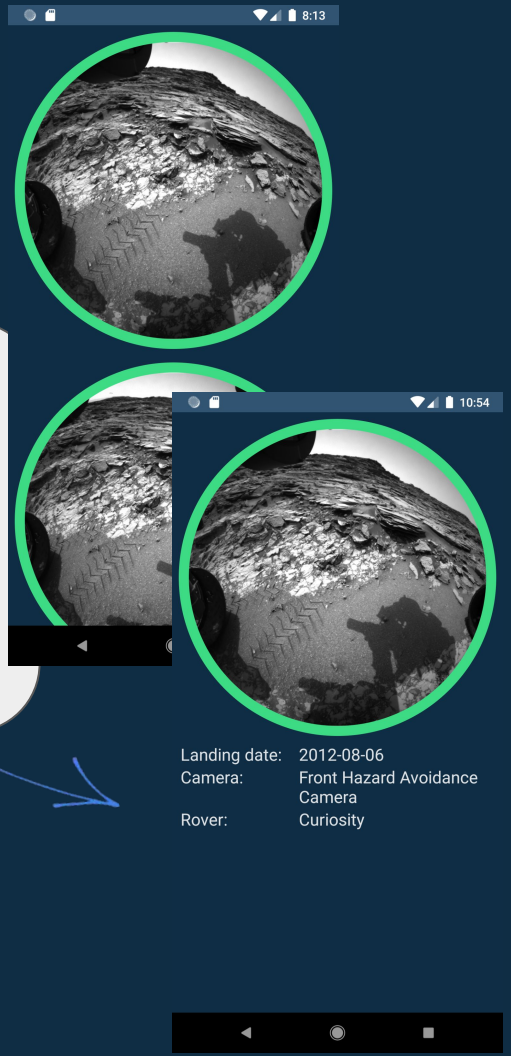
What I actually did

```
AnimatedVisibility(
    visible = showDetails,
    enter = slideInVertically(initialOffsetY = { it }),
    exit = slideOutVertically(targetOffsetY = { it })
) {
    PhotosDetails(
        photo = photos.value[photoDetail],
        isShown = { showDetails = false }
    )
}
```

So what does navigation look like?

First let's define our routes

```
sealed class Screen(val route: String) {
    object ListScreen : Screen("list")
    object DetailsScreen : Screen("details")
}
```

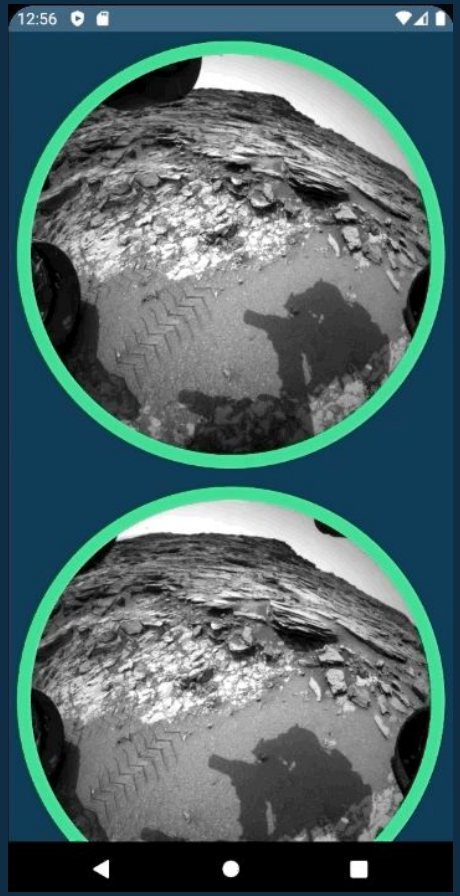


```
setContent {
 val navController = rememberNavController()

 MyTheme(darkTheme = true) {
   MyApp {
       NavHost(
           navController = navController,
           startDestination = Screen.ListScreen.route
       ) {
           composable(Screen.ListScreen.route) {
               PhotoList(mainViewModel, navController)
           }
           composable(Screen.DetailsScreen.route) {
               PhotosDetails(mainViewModel, navController)
           }
       }
     }
   }
}
```

```
onClick = { index ->
    viewModel.setSelectedIndex(index)
    navController.navigate(Screen.DetailsScreen.route) {
        // popUpTo("route") { inclusive = true }
        // launchSingleTop = true
    }
}
```

The final final product



Wrapping up… Why do we love Compose?

- Integration
  1. with your app
  2. with libraries
  3. with MVVM
     (but also other architectures!)

Wrapping up… Why do we love Compose?

- Integration
- All Kotlin all the time!

Compose can be used in a lot of different places!

Wrapping up… Why do we love Compose?

- Integration
- All Kotlin all the time!
- Great time to get started!

Thank you!

Where do you find me?

 @_sierraObryan

sierraobryan.dev

Resources

https://www.droidcon.com/media-detail?video=543570509

https://developer.android.com/courses/pathways/compose

https://developer.android.com/jetpack/compose/mental-model

https://www.raywenderlich.com/books/jetpack-compose-by-tutorials/v1.1/chapters/2-learning-jetpack-compose-fundamentals

These slides and the example app:

https://github.com/sierraobryan/mars-rover-compose

Questions?