

ANDROID DEVELOPMENT

Class 3: Thinking in Compose

UI Basics: Building Static Screens in Kotlin

Instructor: Mark Joseli

| Today's Goal

Building static screens completely in Kotlin.

- Shift from XML to declarative UI
- Understand the Component Tree
- Create Basic UI Elements



Jetpack compose

- A new toolkit for building native user interfaces on Android.
- Jetpack Compose is a modern toolkit developed by Google for building user interfaces (UI) on Android, based on a declarative model.
- Released to simplify UI development.
- It allows developers to create UI directly with Kotlin code, without the need for XML, simplifying the development process.

Why Jetpack Compose?

- Simplifying UI Development
- With Jetpack Compose, you no longer need to create XML layouts separate from your code. Instead, the entire UI is declared directly in the Kotlin code.
 - This reduces fragmentation between layout XML files and application logic, resulting in a cleaner, more efficient workflow.
- UI code is more concise and readable, eliminating the need to work with long imperative classes and methods to modify interface components.

Why Jetpack Compose?

- Declarative Model
 - Jetpack Compose uses a declarative model, in which you describe what the interface should display based on the current state of the application.
- This means that instead of manually manipulating the interface to reflect changes in state, Compose automatically reacts to data changes and updates the UI accordingly.
- This model improves code clarity, making it easier to manage complex and dynamic UIs.

Why Jetpack Compose?

- Less Boilerplate Code
- Jetpack Compose eliminates much of the repetitive code developers face when working with traditional Views and XML.
- Less code means less chance of errors, as well as easier-to-read and maintain interfaces.

Why Jetpack Compose?

- Better Productivity
- Jetpack Compose offers more fluid integration with Android Studio, including tools such as interactive preview, which allows you to view interface changes in real time without having to compile the project.
- Additionally, Compose supports hot-reload, which allows you to instantly see changes to the UI while the application is running, which speeds up the development cycle.

Why Jetpack Compose?

- Kotlin integration
- Compose is built on Kotlin, which allows developers to leverage the language's modern features, such as high-order functions and lambda expressions, to create more flexible and powerful UIs.
- Leveraging the power of Kotlin also reduces the need for extensive code and increases the expressiveness of the language.

Why Jetpack Compose?

- Reactive State Management
- Compose makes it easy to manage state within user interfaces using reactive APIs like State and remember.
- This makes it much simpler to create UIs that respond to real-time changes, such as displaying updated search results or lists of data.

Why Jetpack Compose?

- Ease of Adaptation and Integration with Existing Projects
- Compatibility with Traditional Views: Although Jetpack Compose is new, it can coexist with Android's traditional Views system. This facilitates gradual adoption into existing projects, allowing developers to migrate parts of the interface to Compose as needed.
- Flexibility: You can start using Compose in smaller parts of your application and gradually convert the entire interface to the new framework.

Why Jetpack Compose?

- Integrated Material Design
- Compose has seamless integration with Material Design, allowing you to quickly create consistent, modern interfaces. Material Design components are directly available and easy to customize.
- Improved Performance
- Compose's architecture is highly optimized for rendering complex interfaces and dynamic UI changes. It performs recompositions only on the parts of the interface that really need to be updated, reducing resource usage and improving the application's overall performance.

Imperative Model

- The imperative model is the traditional method of building UIs on Android, where the developer tells the system how to build the interface and what should be done at each step. This is generally done using XML to define layouts and Java/Kotlin code to modify the behavior of components at runtime.
- Features:
- Defines Step by Step:
- The developer needs to define the interface layout and then control the behavior of each element individually, specifying each action and response to events.
 - Example: When clicking a button, you need to capture the event and manually change the text of a TextView through code.

Imperative Model

- Separation of Logic and Interface:
- Typically, the UI is defined in XML files, while the logic is in Kotlin or Java code. This results in multiple points of interaction (between XML and code) to define and change the interface.
- Explicit State Maintenance:
- The developer is responsible for explicitly managing the state of the UI, which means they need to ensure that the UI updates correctly when the state changes.

Challenges of the Imperative Model

- Complexity:
- For dynamic interfaces, where the state changes frequently, the developer needs to manually ensure that the UI updates correctly with each change, which can lead to bugs and confusing code.
- Maintenance:
- Keeping track of what is being displayed in the UI can become difficult as the complexity of the application increases, as various parts of the code need to be updated.

Modelo Imperativo

```
<!-- Layout XML -->
```

```
<Button
```

```
    android:id="@+id/button"
```

```
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
```

```
    android:text="Click me"/>
```

```
<TextView
```

```
    android:id="@+id/textView"
```

```
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
```

```
    android:text="Hello"/>
```

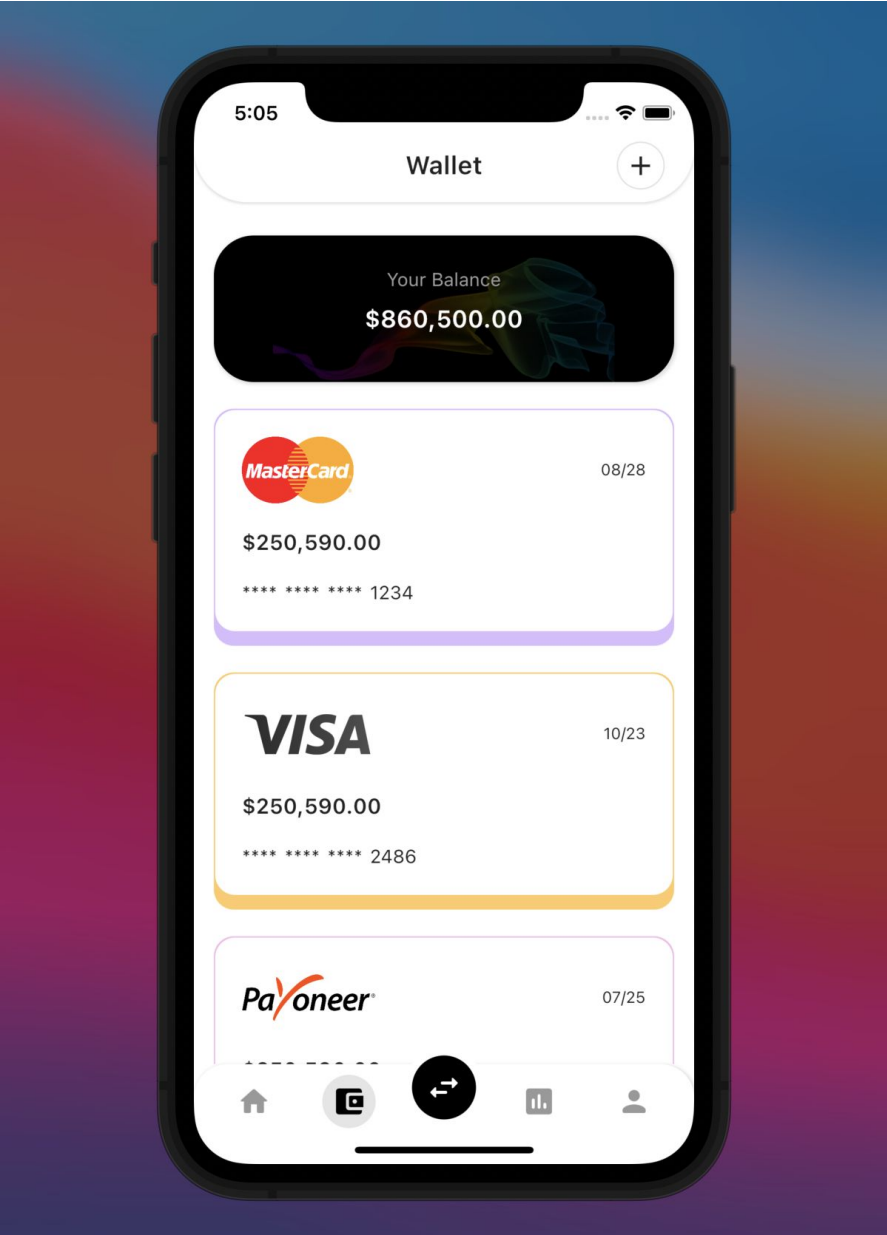
```
val button = findViewById<Button>(R.id.button)
```

```
val textView = findViewById<TextView>(R.id.textView)
```

```
button.setOnClickListener {
```

```
    textView.text = "Button clicked!"
```

```
}
```

Declarative Model

- In the declarative model, like Jetpack Compose, the developer describes what the interface should show based on the current state, and the framework takes care of rendering that interface automatically, without having to explicitly define how each element should change over time.
- Features:
- Describes the UI State:
 - Instead of telling the system "how" to make changes, the developer simply defines "what" the interface should display. If the state changes, Compose takes care of updating the UI to reflect that change.

Declarative Model

- Integration with the State:
 - Jetpack Compose automatically manages the recomposition of parts of the UI when the state associated with it changes. This eliminates the need for manual UI updates.
- Life Cycle Controlled by the Framework:
 - The UI is reactive and, unlike the imperative model, the framework is responsible for reacting to changes in state and ensuring that the interface always reflects these states.

Declarative Model

```
@Composable
fun MyButton() {
    var clicked by remember { mutableStateOf(0) }

    Column {
        Button(onClick = { clicked++ }) {
            Text("Click me")
        }
        Text(text = "Button clicked $clicked times")
    }
}
```

Advantages of the Declarative Model

- **Simplicity:** The developer does not need to manually control each interface change. Instead, just set the UI based on the current state.
- **Readability:** The code is more concise and easier to understand, as the interface is defined as a pure function that reacts to state changes.
- **Fewer Errors:** As the framework takes care of ensuring that the UI is always synchronized with the state, there is less chance of errors when manipulating the interface.

| The Paradigm Shift

Imperative (XML)

- > Manually manipulate the UI hierarchy.
- > Use `findViewById()` to locate views.
- > **Mutate** the state of views directly (e.g., `setText()`).

Declarative (Compose)

- > Describe the UI based on the current state.
- > UI updates **automatically** when state changes.
- > Uses `@Composable` functions.

| The `@Composable` Annotation

Transforms a Kotlin function into a UI component.

```
@Composable fun Greeting(name: String) { Text(text  
= "Hello $name!") }
```

- > Tells compiler: *"This function converts data into UI."*
- > Composable functions can call other composables.



Jetpack Compose Concepts - @Composable

- Functions annotated with @Composable are the fundamental building blocks of Jetpack Compose.
- They describe how the user interface should be rendered.
- Every function that draws something in the UI is marked with @Composable, indicating that this function is part of the Jetpack Compose composition system.
- These are functions that return UI instead of a value.
- They can be combined with each other to build complex interfaces.
- Each @Composable can call other composable functions.

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello, $name!")
}
```

Basic Components: **Text**

Displays read-only text. Equivalent to `TextView`.

```
Text( text = "Welcome Back", color = Color.Blue,  
      fontSize = 24.sp, fontWeight = FontWeight.Bold )
```

Key styles include `color`, `fontSize`, and `fontWeight`.

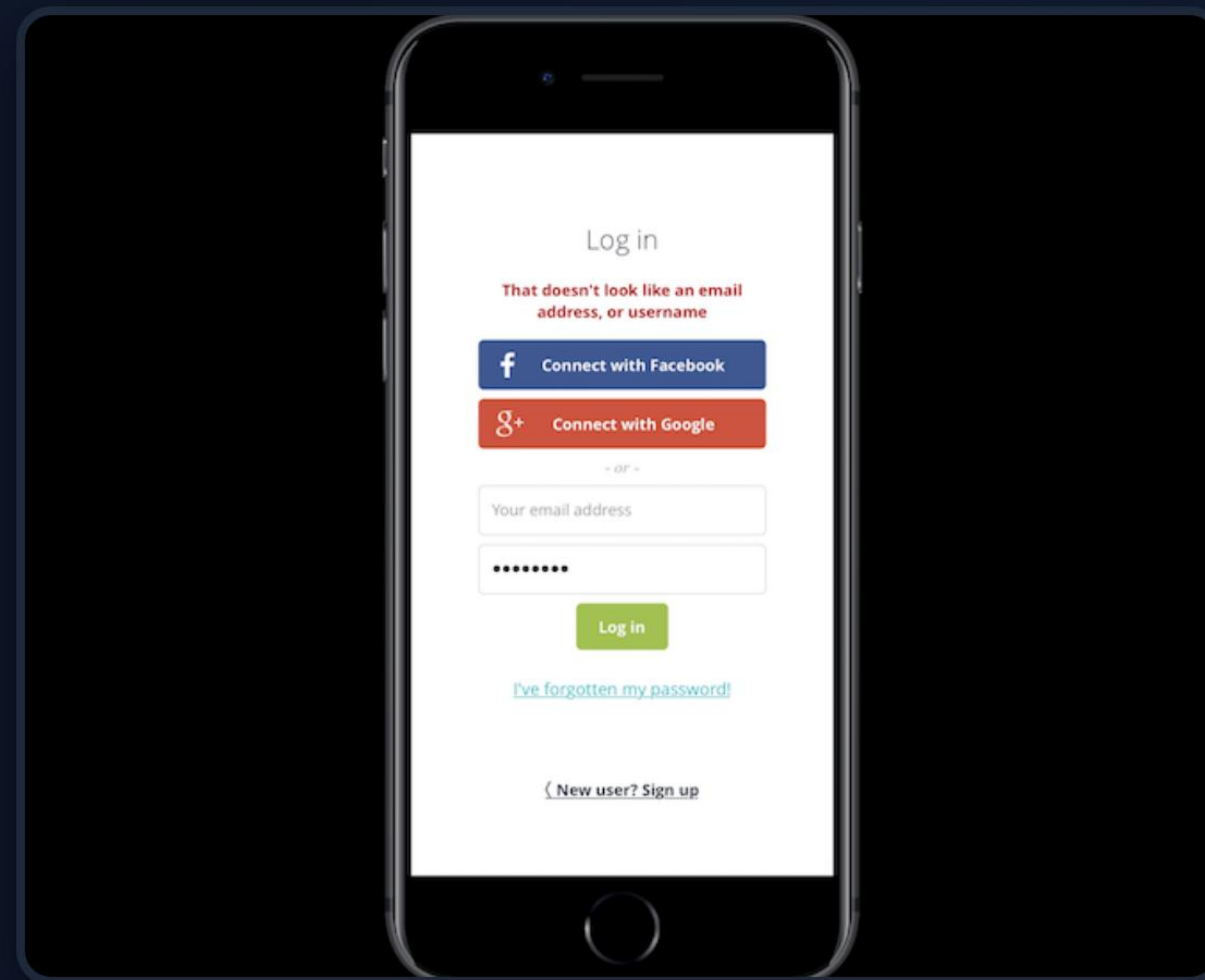


Basic Components: Button

A clickable container. You add content (like text) inside it.

```
Button( onClick = { /* Do something */ } ) {  
  Text("Click Me") }
```

- > **onClick:** Lambda function for logic.
- > **Content:** It's a container! Add icons or text inside.

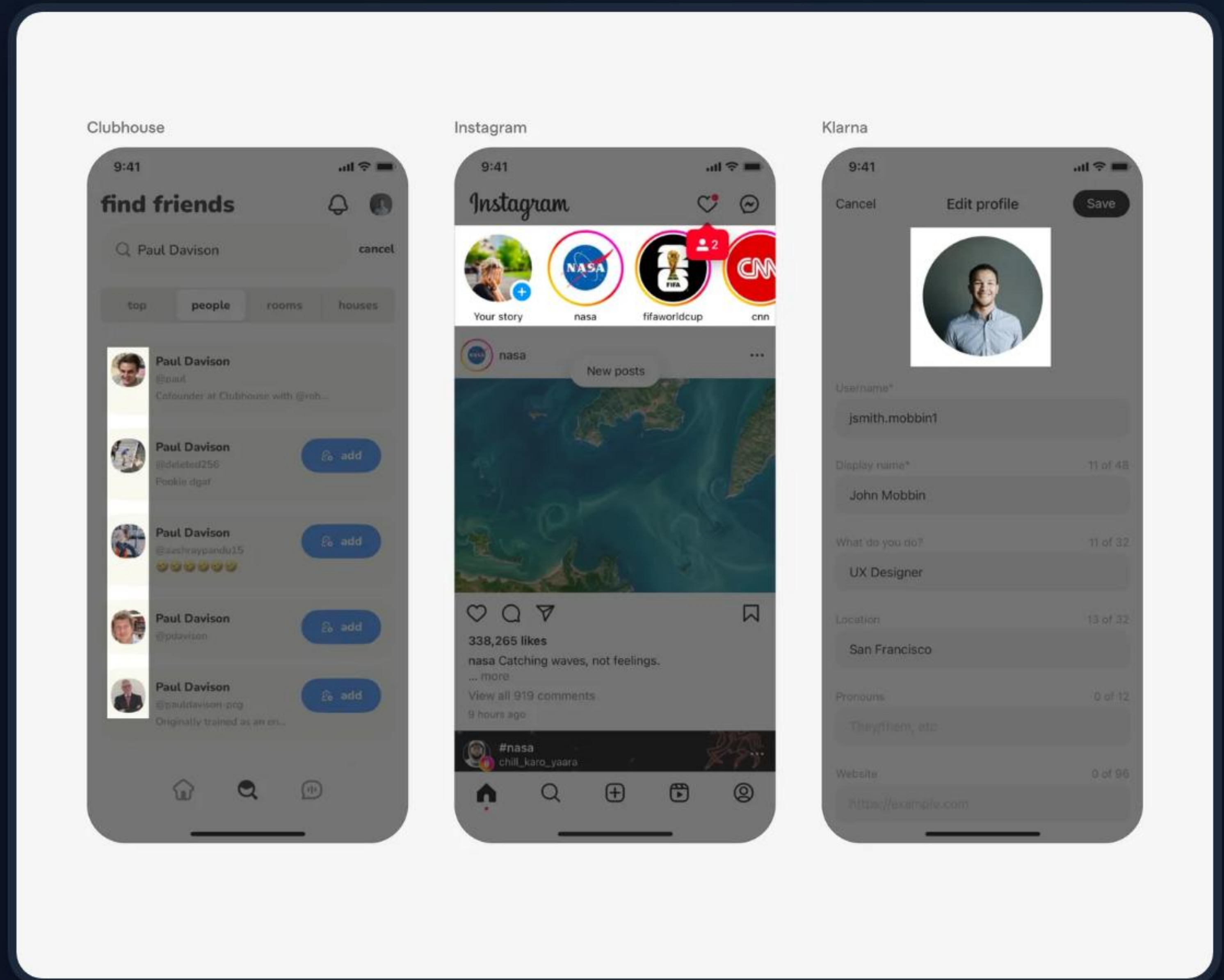


Basic Components: Image

Displays a visual asset (drawable or bitmap).

```
Image( painter = painterResource(id =  
R.drawable.profile), contentDescription = "User  
Avatar" )
```

- > **painter**: The image resource to draw.
- > **contentDescription**: Crucial for accessibility.

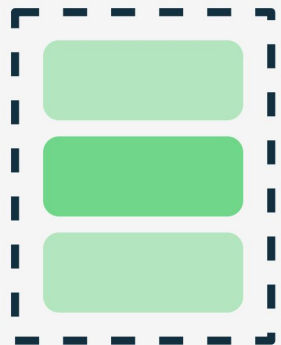


Jetpack Compose Concepts - Layouts

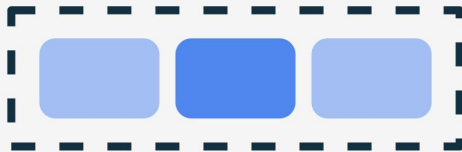
- Layouts are composable functions that organize visual elements on a screen.
- Unlike the Views system, layouts in Compose are more flexible and less verbose.
- Main Layouts:
 - Column: Arranges elements vertically.
 - Row: Arranges elements horizontally.
 - Box: Stacks elements on top of each other.

```
@Composable
fun MyScreen() {
    Column {
        Text(text = "Hello")
        Button(onClick = { /* ação */ }) {
            Text(text = "Click me")
        }
    }
}
```

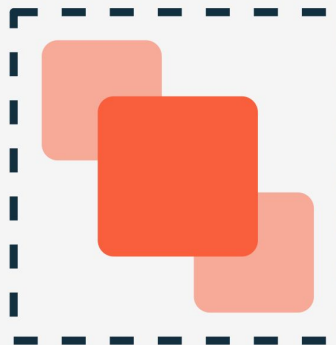
Jetpack Compose Concepts - Layouts



Column



Row



Box

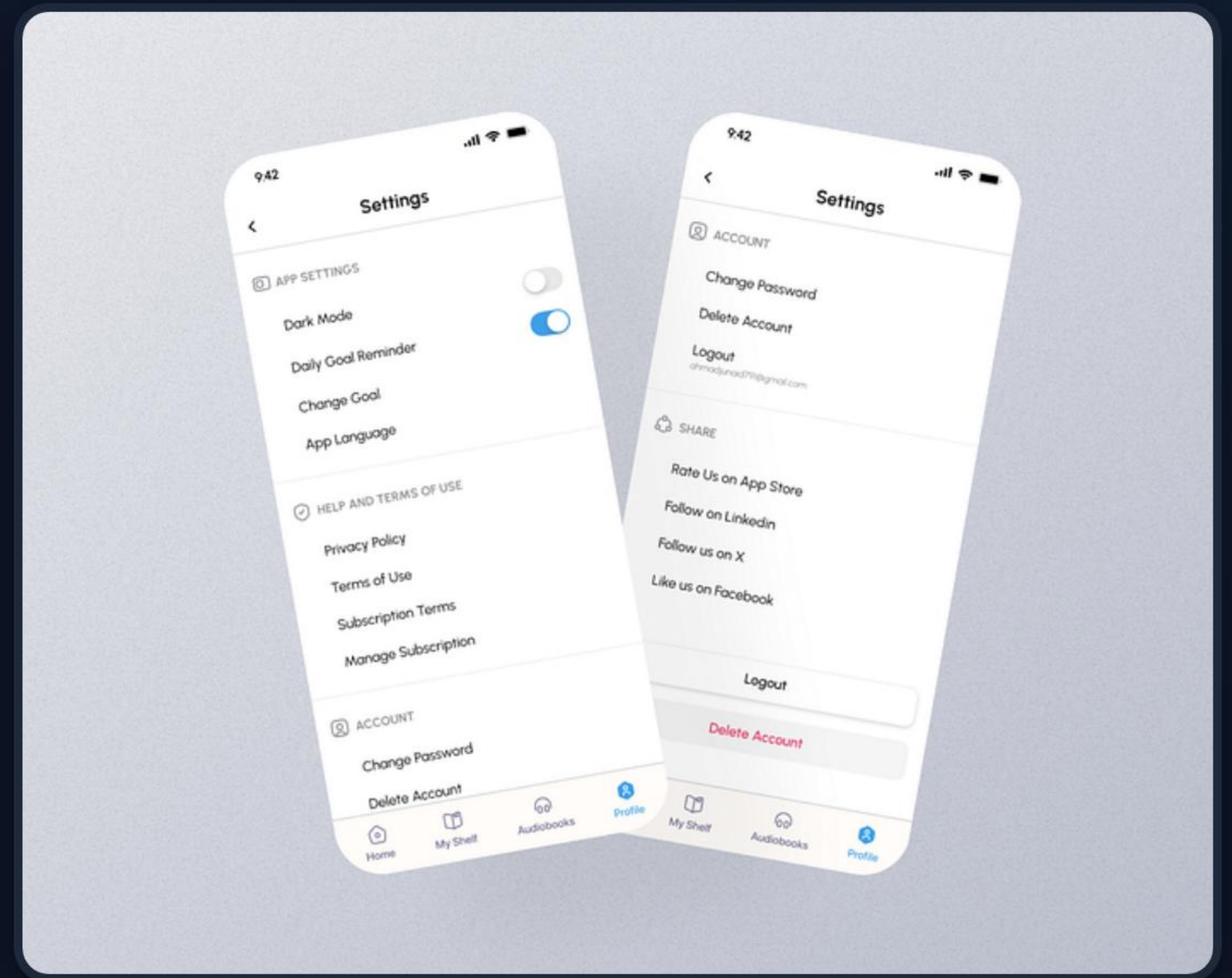
```
@Composable
fun MyScreen() {
    Column {
        Text(text = "Hello")
        Button(onClick = { /* ação */ }) {
            Text(text = "Click me")
        }
    }
}
```


Layouts: Column

Places items **vertically** (Y-axis).

```
Column { Text("Setting 1") Text("Setting 2")  
Button(...) }
```

Equivalent to a vertical **LinearLayout**.

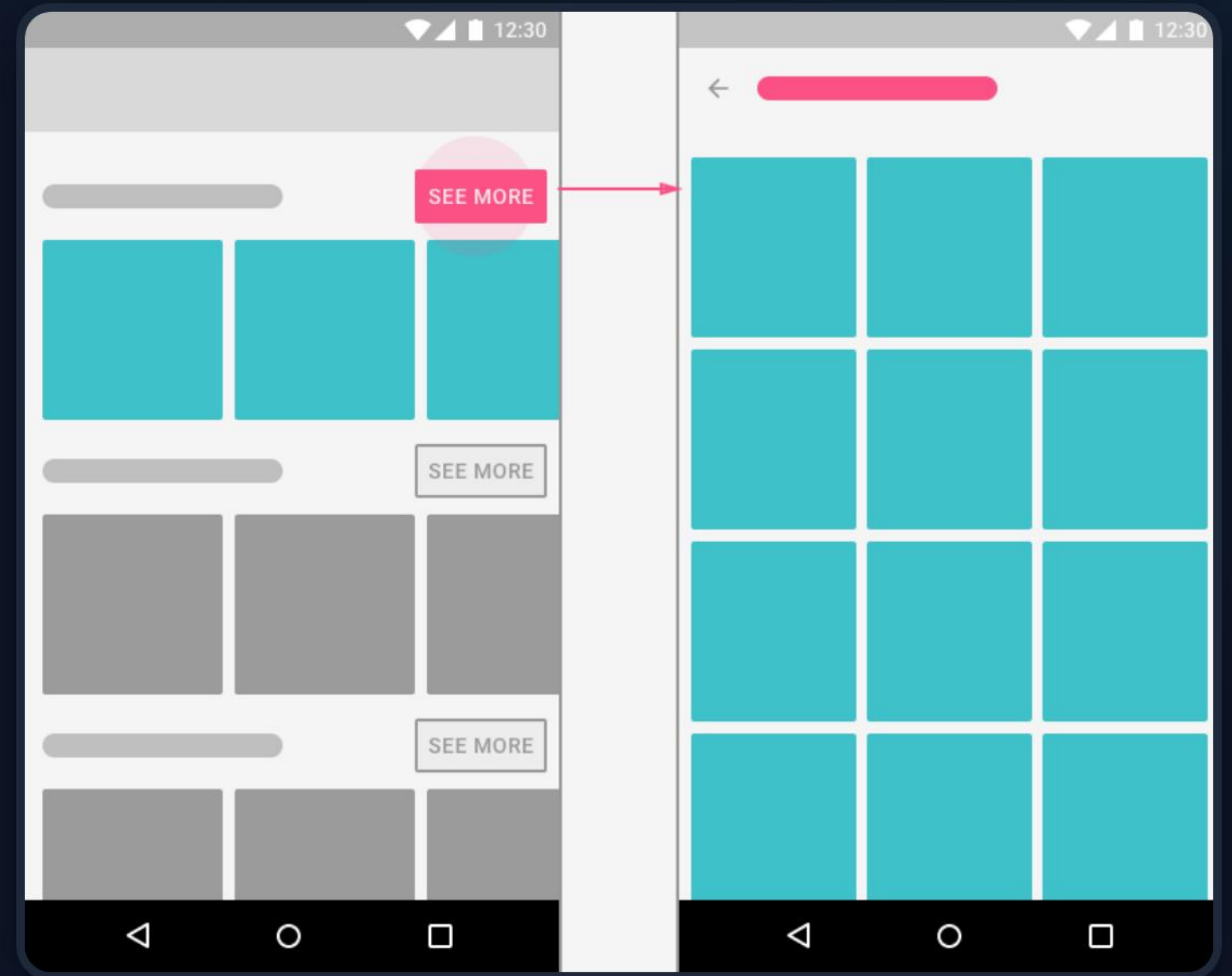


Layouts: Row

Places items **horizontally** (X-axis).

```
Row { Icon(...) Text("Artist Name") }
```

Equivalent to a horizontal **LinearLayout**.



Layouts: **Box**

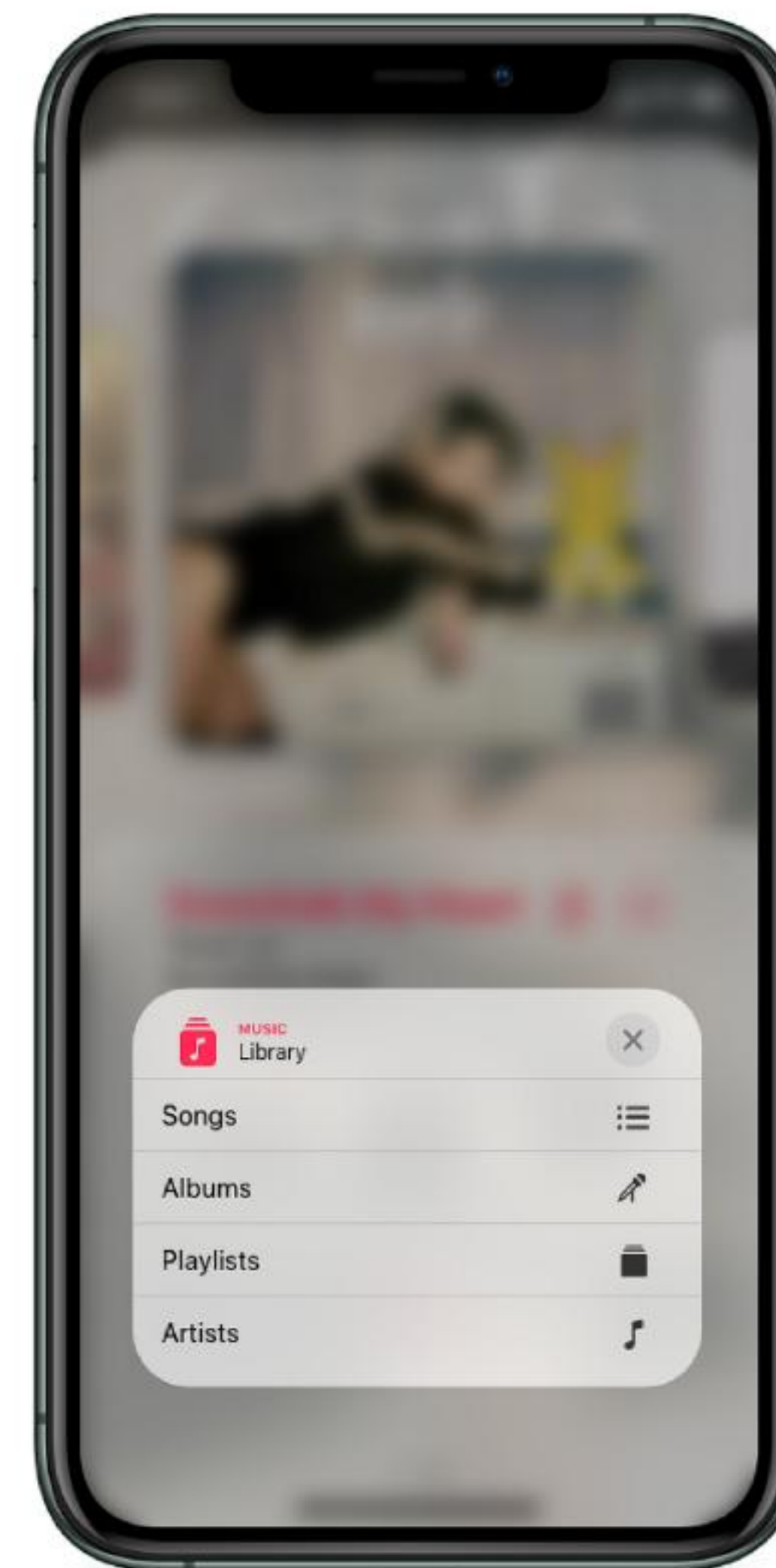
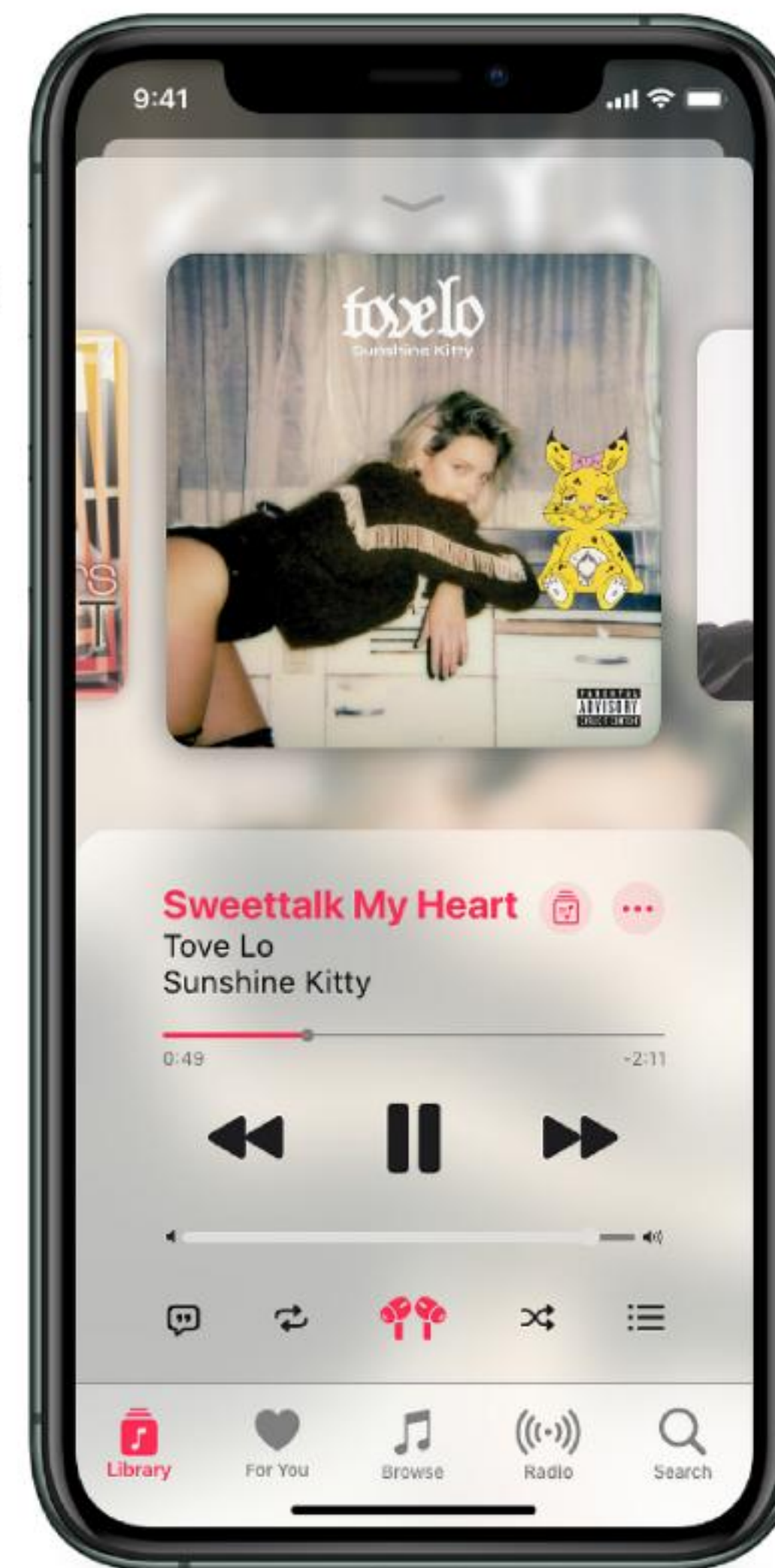
Stacks items on **top** of each other.

```
Box { Image(painter = /* Album Art */) Icon(  
  imageView = Icons.Default.Play, modifier =  
  Modifier.align(Alignment.Center) ) }
```

Great for overlays and backgrounds.

Music
UX CASE STUDY
NOW PLAYING
JAYMIE GILL

Quick Navigation



Jetpack Compose Concepts - Modifiers

- Modifiers are used to change the appearance or behavior of a composable element, such as defining dimensions, positioning, colors, borders, etc.
- Modifiers can be applied to any composable element and are highly chainable.

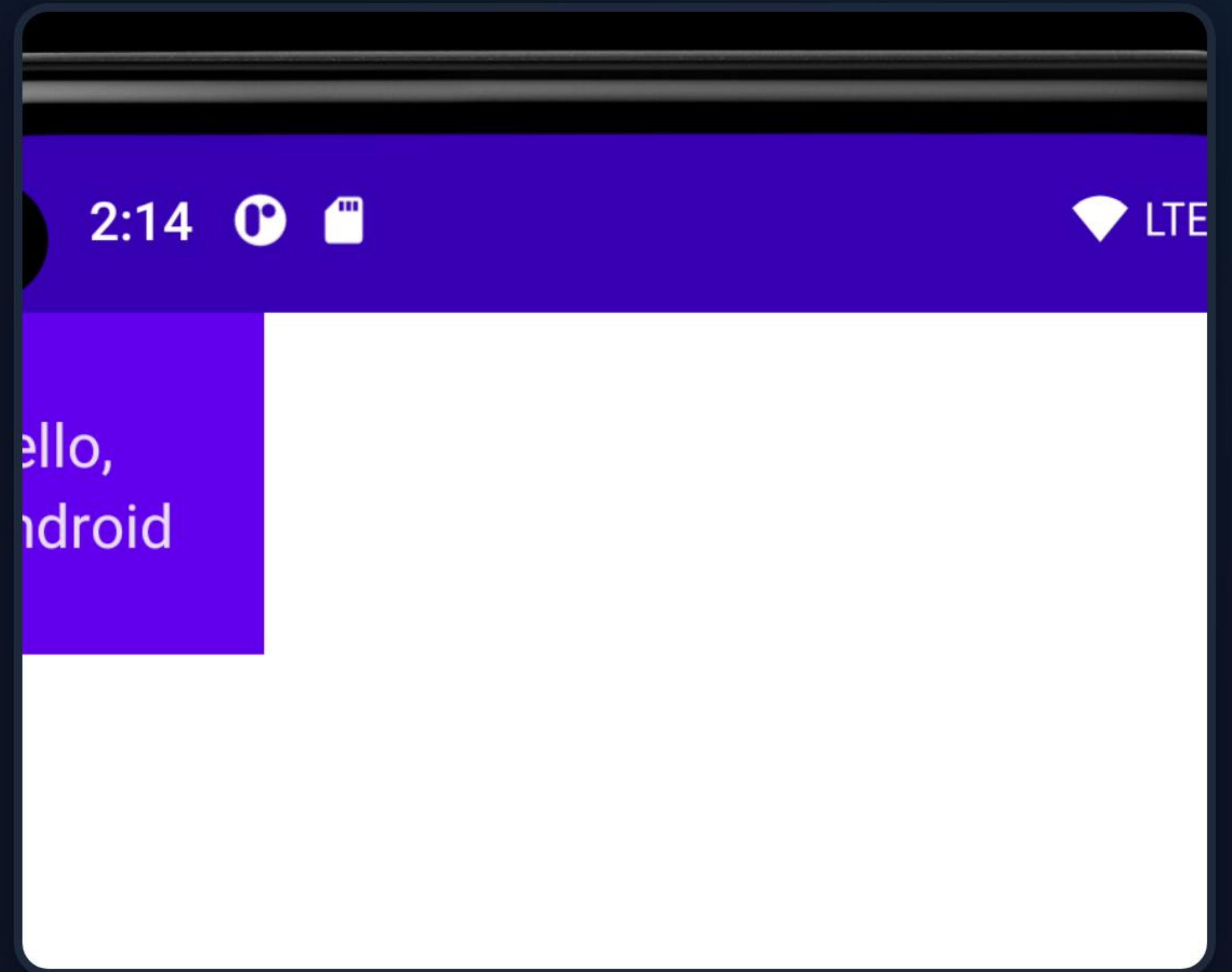
```
@Composable
fun ModifiedButton() {
    Button(
        onClick = { /* action */},
        modifier = Modifier.padding(16.dp).fillMaxWidth()
    ) {
        Text("Click me")
    }
}
```


Modifiers

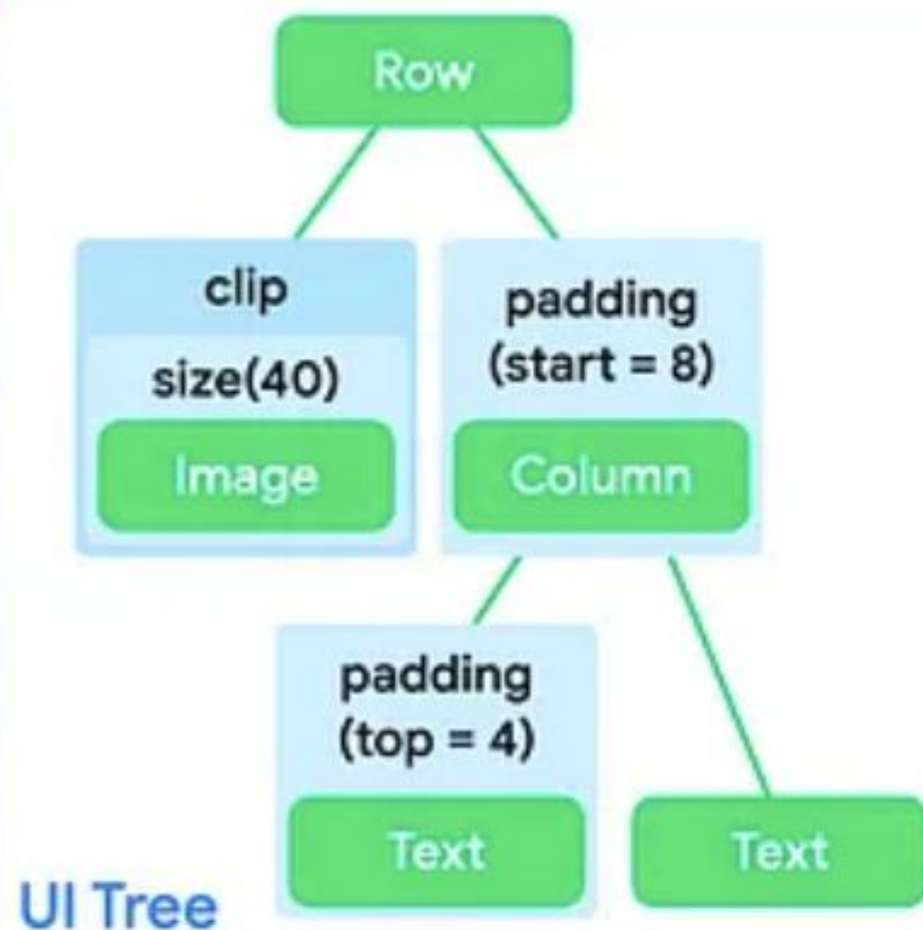
The "Magic Wand" for styling and behavior.

```
Text( text = "Click Me", modifier = Modifier  
      .padding(16.dp) .background(Color.Green) .clickable  
      { } )
```

⚠ **Order matters!** Padding before background ≠ Padding after background.



```
Row {  
  Image(  
    ...,  
    Modifier  
      .clip(CircleShape)  
      .size(40.dp)  
  )  
  Column(  
    Modifier.padding(start = 8.dp)  
  ) {  
    Text(  
      ...,  
      Modifier.padding(top = 4.dp)  
    )  
    Text(...)  
  }  
}
```



Jetpack Compose Concepts - Preview

- Definition: The Preview feature in Android Studio allows you to preview a `@Composable` function without the need to run the application on an emulator or physical device.
- Preview is useful for seeing UI changes in real time as you develop.

```
@Preview
@Composable
fun PreviewGreeting() {
    Greeting(name = "Preview")
}
```

Questions?

Let's open Android Studio and build some screens!

Image Sources



<https://i.ytimg.com/vi/IU0CHh3tf20/sddefault.jpg>

Source: www.youtube.com



<https://www.datocms-assets.com/22695/1751322361-1719248390-typography-for-moblie-apps-1.webp>

Source: www.ramotion.com



<https://d585tldpucybw.cloudfront.net/sfimages/default-source/blogs/2019/08/canva-login.png>

Source: www.telerik.com



<https://bytescale.mobbin.com/FW25bBB/image/mobbin.com/prod/file.webp?enc=1.DfSR4rss.RTMKptUdZpPJvTZR.9bpgGiPOSXNgss4FIYgpiow0Mgq3HVKuL1o0bEBHzQG8qetZu-4mf bPUBtEtIOZZOWrnaevZlcKAZTHjOeVO08H2AOs95F4gqWTH7trj-h9wBRr5CZcPlnj5JnMKED4ACxoP3K3Y7m78IFCz5JeDfeiH3pUlHV4rDYiq8jMBFrYNDc5nqxqrHZHtbN4jCJm17CmkQTkRhO627uqUb4>

Source: mobbin.com



<https://cdn.dribbble.com/userupload/43299987/file/original-acda92c8d1b8e3a50f03a8d5ab7cb045.png?resize=752x&vertical=center>

Source: dribbble.com



https://miro.medium.com/v2/resize:fit:1400/1*uwC0rDxPr1q9-lHkBBLMKQ.png

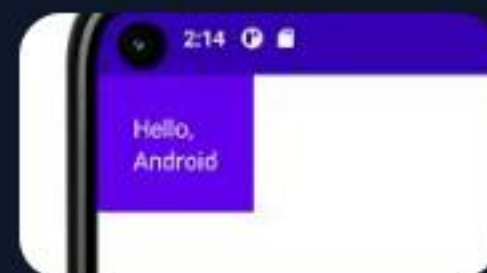
Source: uxdesign.cc

| Image Sources



https://miro.medium.com/v2/resize:fit:2000/1*NPlwHsVxGwoP5fnZnQd-gg.png

Source: blog.prototypr.io



<https://developer.android.com/static/develop/ui/compose/images/modifier-1-modifier.png>

Source: developer.android.com