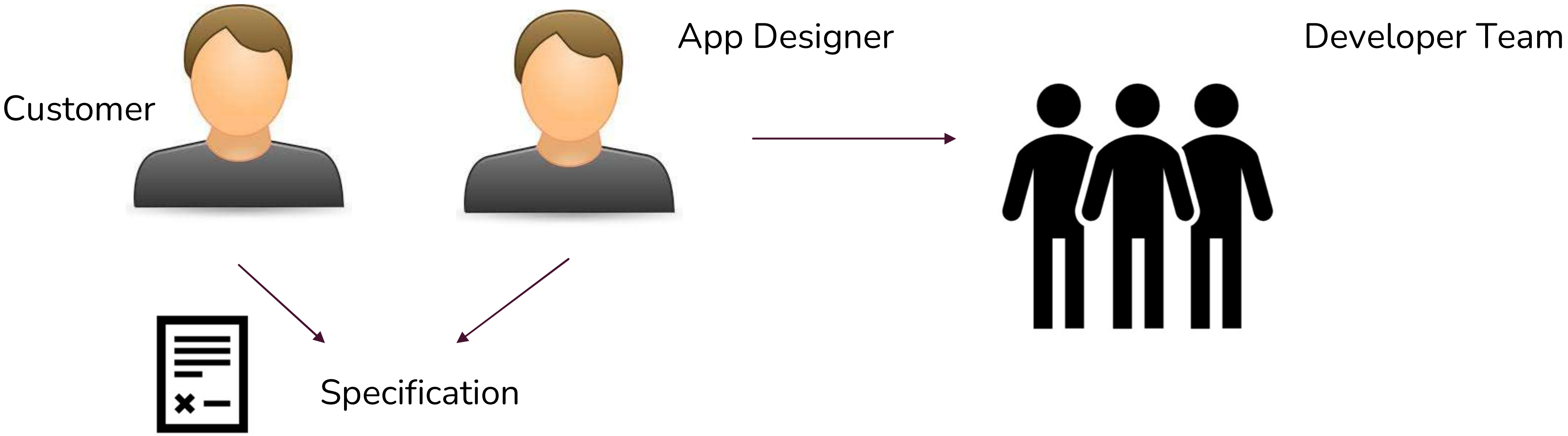
PRESENTATION LAYER

SELECTED RESOURCES

- https://developer.android.com/develop/ui/compose/
- https://developer.android.com/courses/jetpack-compose/course
- https://github.com/android/compose-samples/tree/main/Jetchat
- https://developer.android.com/topic/architecture/recommendations

PRELUDE: APP SPECIFICATION

- When developing an app, the first step is to understand **what** the user expects to see and **how** they will interact with app.
- The design of UI (user interface) isn’t then just aesthetics: it's logic, communication, (and psychology).
- The process starts with a specification of the requirements, which is agreed between a customer and the designer



Customer

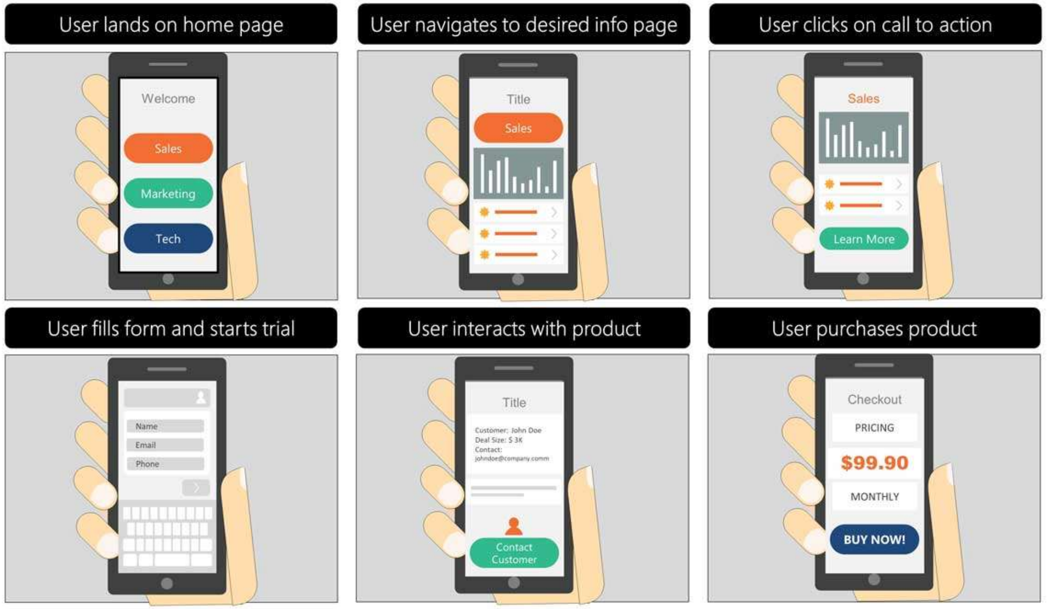
App Designer

Developer Team

Specification

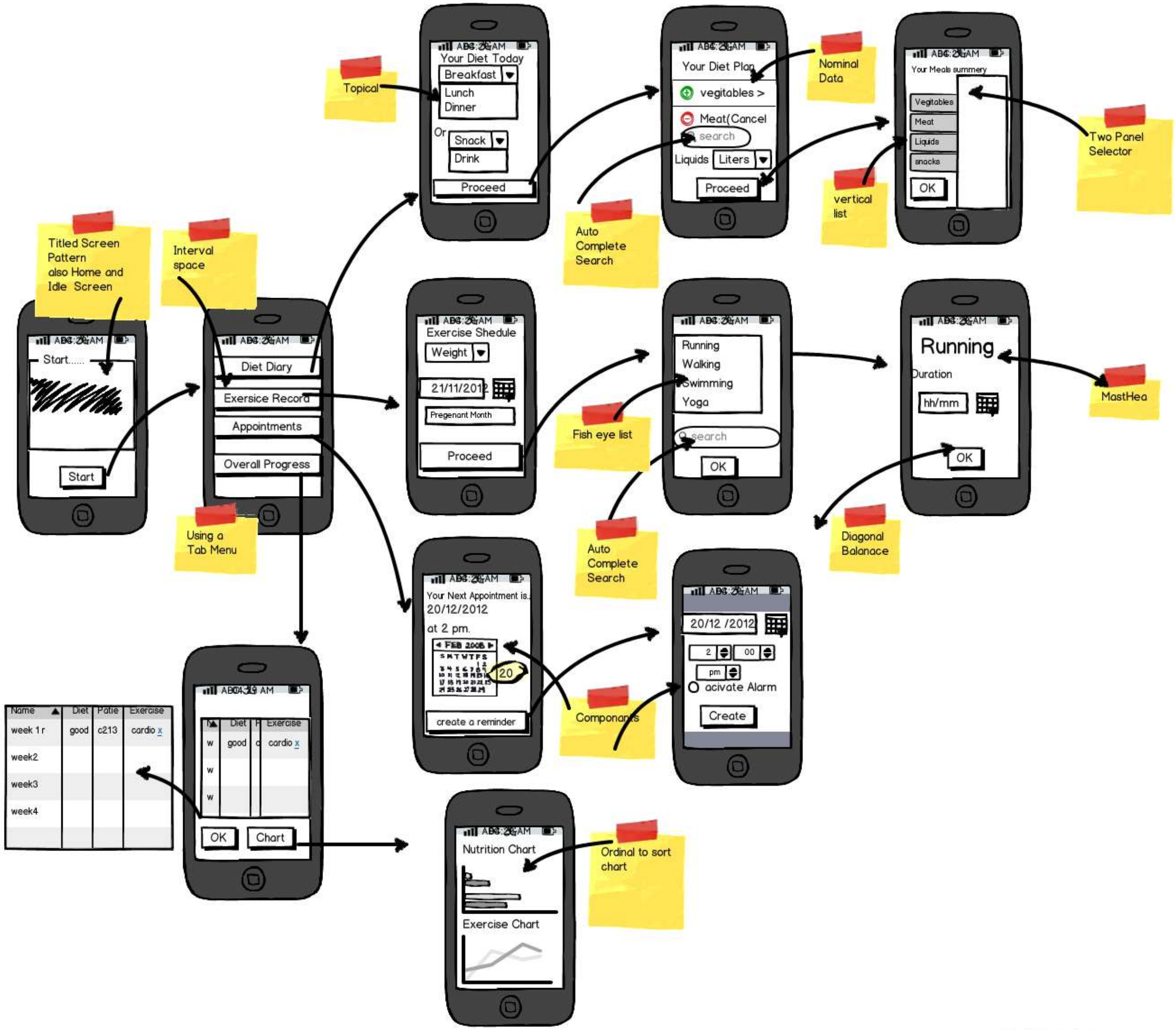
PREDULE: STORYBOARD

- A storyboard is a visual representation of basic screen flow that shows key user interface components and basic user interaction to achieve a user goal.
- It useful for brainstorming on requirements.



slidemodel.com

EXAMPLE

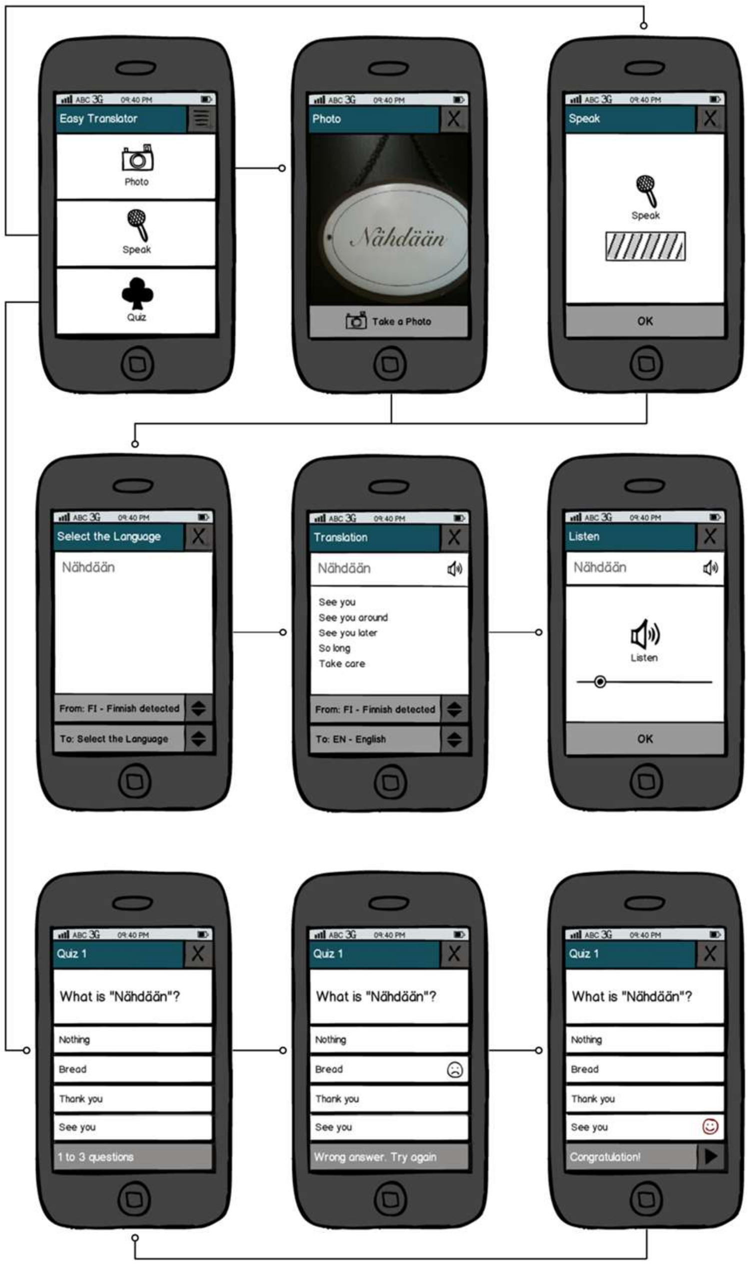


Name	Diet	Patie	Exercise
week 1 r	good	c213	cardio x
week2			
week3			
week4			

N	Diet	P	Exercise
w	good	c	cardio x
w			
w			

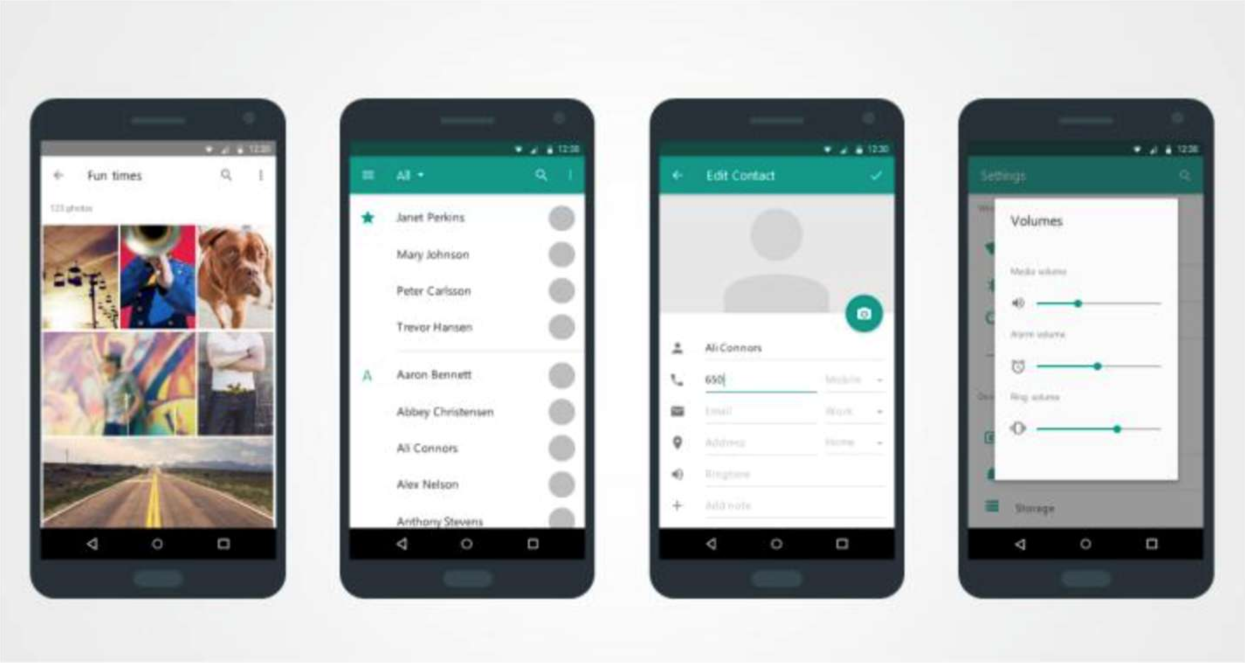
created with Balsamiq Mockups -

ANOTHER EXAMPLE



PRELUDE: MOCKUP

- A storyboard can be ‘executed’ using a **mockup** , which is a high-fidelity static design for representing the final app



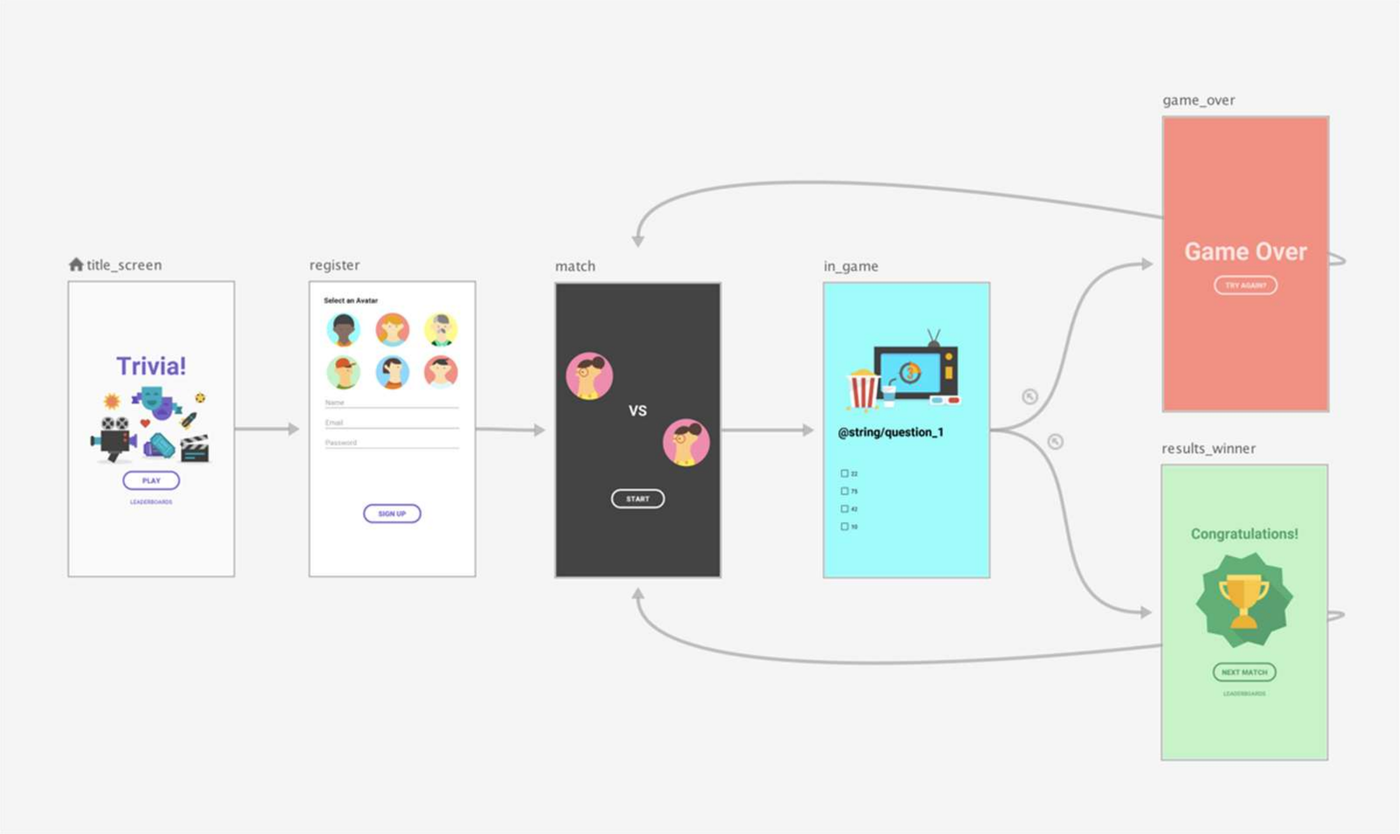
A USEFUL TOOL: FIGMA

- Figma is a collaborative, cloud-based interface design tool used for creating websites, apps, and other digital products.

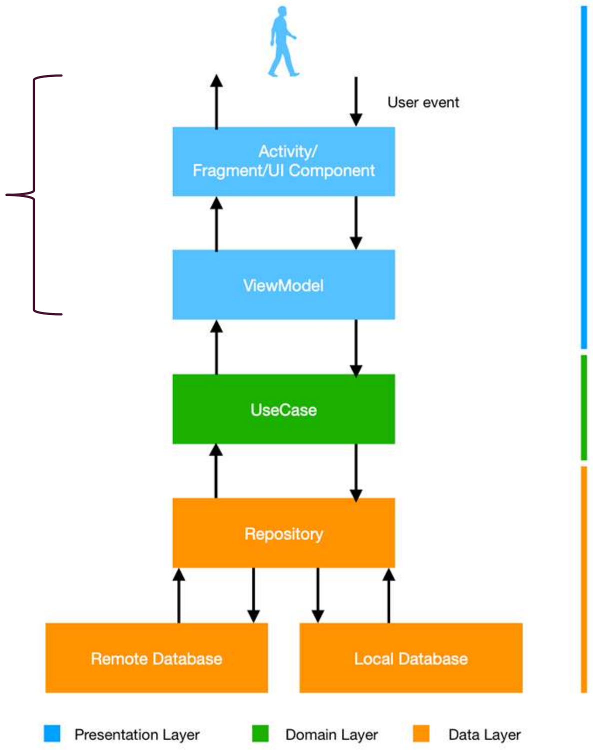
https://www.figma.com/design/IlsGy55CL4NAMG3A5NMyu0/Prototyping-in-Figma?node-id=0-1&p=f&t=AxVRgBp16Trmngpv-0

NAVIGATION GRAPH IN ADROID

- In the Android framework, the storyboard is connected to the concept of **navigation graph**
- The navigation graph is a graph where nodes are screens and edges actions that allows to reach a screen



CLEAN ARCHITECTURE



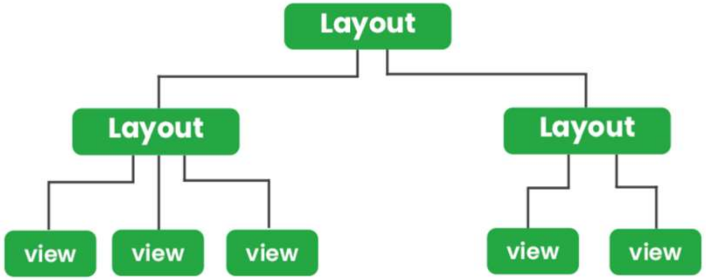
MAIN ADVANTGES

- Easy Modification
- Easy test
- Easy to reuse

THE ANDROID VIEW SYSTEM

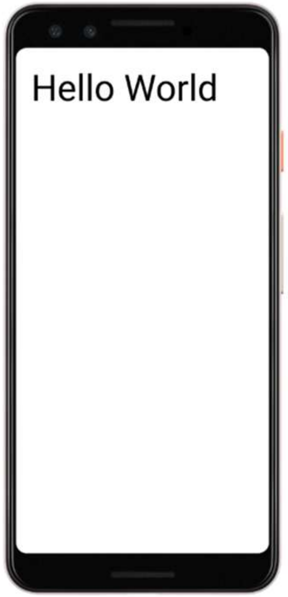
- The area of the screen that can draw UI elements is considered as a **tree of views**
- **View** = rectangular shape on the screen that 'knows how to draw itself' wrt to the containing view
- View can be populated using

1. UI toolkits in the Android framework are based on:
   - Imperative language (the 'old' way) using XML files describing UI building blocks, like HTML
   - Declarative language using composable functions in androidx.
2. Draw directly via Canvas abstractions (2D graphics or openGL ES)

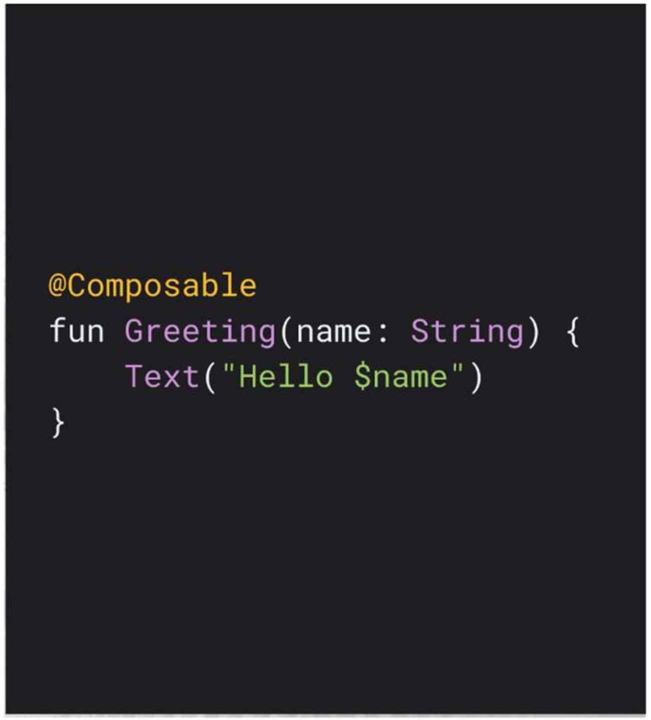


JETPACK’S COMPOSABLE FUNCTIONS

- A composable function takes data as input and emits UI elements



Hello World



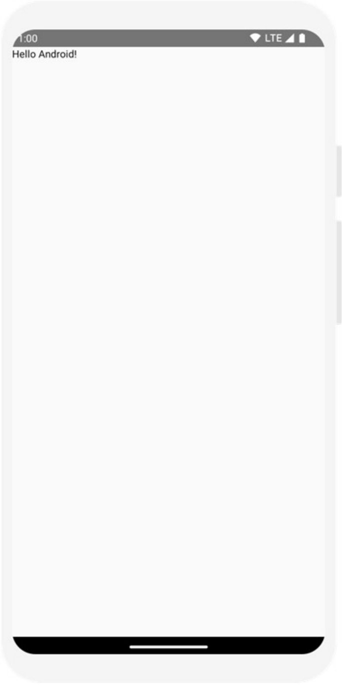
```kotlin
@Composable
fun Greeting(name: String) {
    Text("Hello $name")
}
```

COMPOSE. FUNDAMENTALS

- Jetpack Compose is built around composable functions.
- UI is described programmatically by composable functions describing how UI should look
- A composable function has the annotation @Composable on top of the function name.

```
class MainActivity : ComponentActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContent {
         MessageCard("Android")
      }
   }
}

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

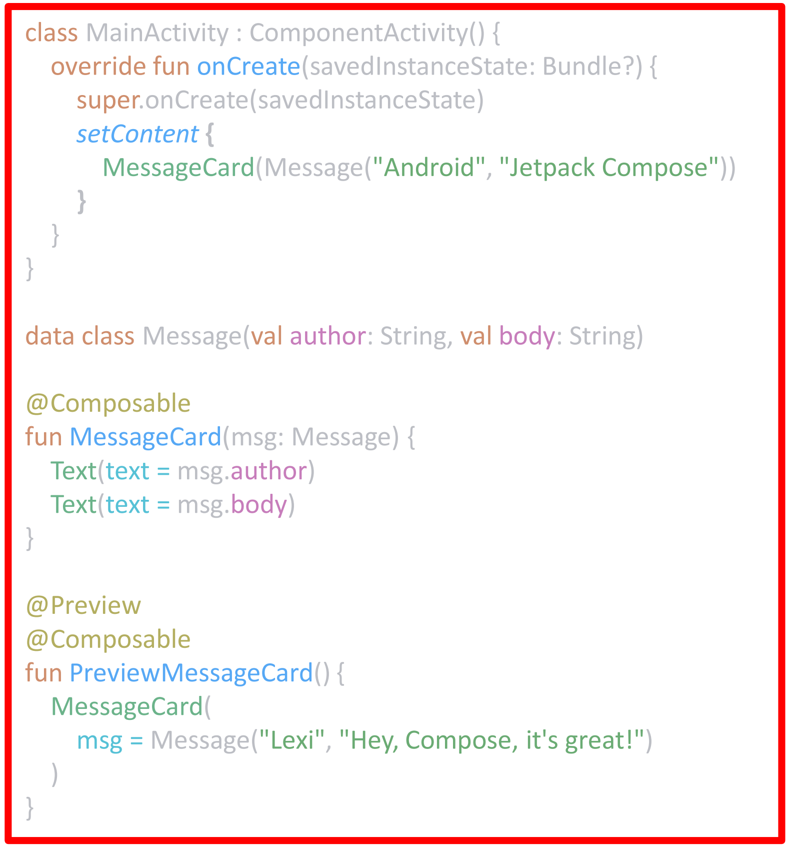
@Preview annotation
shows the function in AS



Hello Android!

COMPOSABLE FUNCTIONS

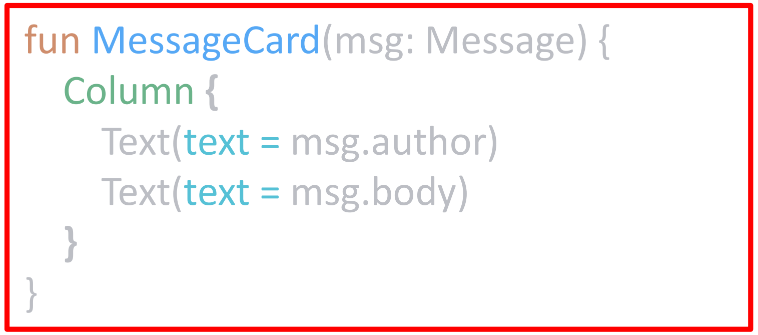
- Composable functions can be composed, meaning that they can combine other functions in their body by calling the functions
- In this way, UI is a hierarchy of composable function
- For example, a screen might be composed of a Column containing Text, Image, and Button composables.
- The leave of this hierarchy are basic UI elements (such as Text, Button, etc.)



```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            MessageCard(Message("Android", "Jetpack Compose"))
        }
    }
}

data class Message(val author: String, val body: String)

@Composable
fun MessageCard(msg: Message) {
    Text(text = msg.author)
    Text(text = msg.body)
}

@Preview
@Composable
fun PreviewMessageCard() {
    MessageCard(
        msg = Message("Lexi", "Hey, Compose, it's great!")
    )
}
```

LAYOUT

- Some function are containers of others
- The Column function allows to arrange elements vertically.
- The Row function arranges items horizontally and Box to stacks elements

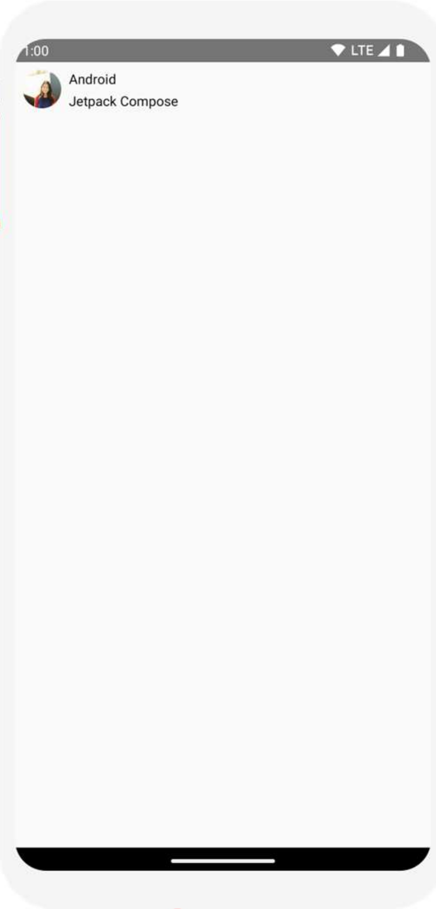


```
fun MessageCard(msg: Message) {
  Column {
    Text(text = msg.author)
    Text(text = msg.body)
  }
}
```

MODIFIERS

- To decorate or configure a composable, Compose uses **modifiers**.
- They allow to change the composable's size, layout, appearance or add high-level interactions, such as making an element clickable.

```
@Composable
fun MessageCard(msg: Message) {
  // Add padding around our message
  Row(modifier = Modifier.padding(all = 8.dp)) {
    Image(
      painter = painterResource(R.drawable.profile_picture),
      contentDescription = "Contact profile picture",
      modifier = Modifier
        // Set image size to 40 dp
        .size(40.dp)
        // Clip image to be shaped as a circle
        .clip(CircleShape)
    )

    // Add a horizontal space between the image and the column
    Spacer(modifier = Modifier.width(8.dp))

    Column {
      Text(text = msg.author)
      // Add a vertical space between the author and message texts
      Spacer(modifier = Modifier.height(4.dp))
      Text(text = msg.body)
    }
  }
}
```

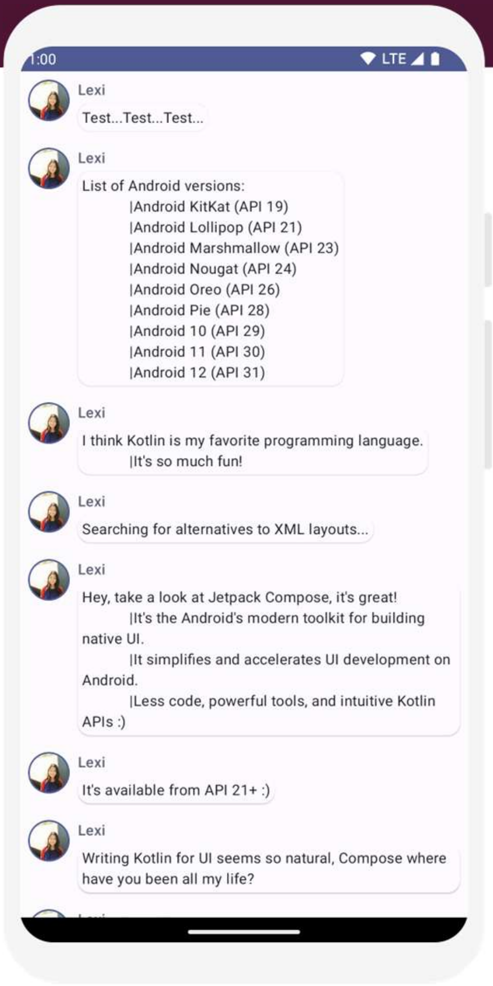


Android

Jetpack Compose

EXAMPLE: LIST

```
import androidx.compose.foundation.lazy.LazyColumn

import androidx.compose.foundation.lazy.items

   @Composable
   fun Conversation(messages: List<Message>) {
      LazyColumn {
         items(messages) { message ->
            MessageCard(message)
         }
      }
   }

   @Preview
   @Composable
   fun PreviewConversation() {
      ComposeTutorialTheme {
         Conversation(SampleData.conversationSample)
      }
   }
```



Lexi

Test...Test...Test...

Lexi

List of Android versions:
        |Android KitKat (API 19)
        |Android Lollipop (API 21)
        |Android Marshmallow (API 23)
        |Android Nougat (API 24)
        |Android Oreo (API 26)
        |Android Pie (API 28)
        |Android 10 (API 29)
        |Android 11 (API 30)
        |Android 12 (API 31)

Lexi

I think Kotlin is my favorite programming language.
        |It's so much fun!

Lexi

Searching for alternatives to XML layouts...

Lexi

Hey, take a look at Jetpack Compose, it's great!
        |It's the Android's modern toolkit for building native UI.
        |It simplifies and accelerates UI development on Android.
        |Less code, powerful tools, and intuitive Kotlin APIs :)

Lexi

It's available from API 21+ :)

Lexi

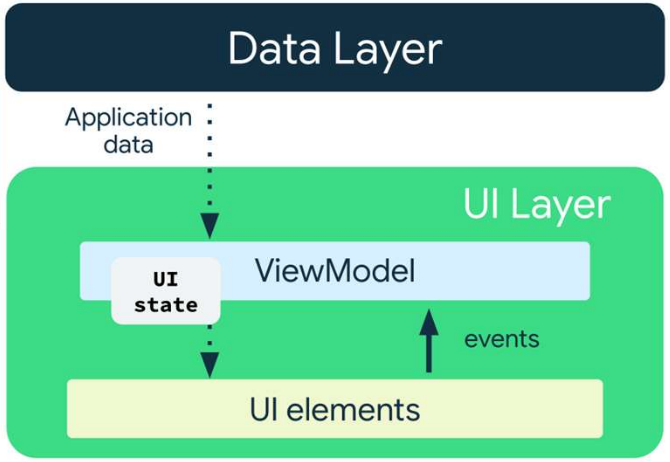
Writing Kotlin for UI seems so natural, Compose where have you been all my life?

STYLES AND MATERIAL DESIGN

- Compose is built to support Material Design principles. Many of its UI elements implement Material Design out of the box.
- Material Design is built around three pillars: Color, Typography, and Shape.
- Material design style is applied using a Theme
- Scaffold provides the high-level layout structure for a typical Material Design screen.
- For example, it gives "slots" for:
  - topBar: A bar at the top (e.g. TopAppBar).
  - bottomBar: A navigation bar at the bottom.
  - floatingActionButton: The floating action button.
  - drawerContent: The drop-down side menu.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MyApplicationTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )

            }
        }
    }
}
```

VIEW AND VIEWMODEL



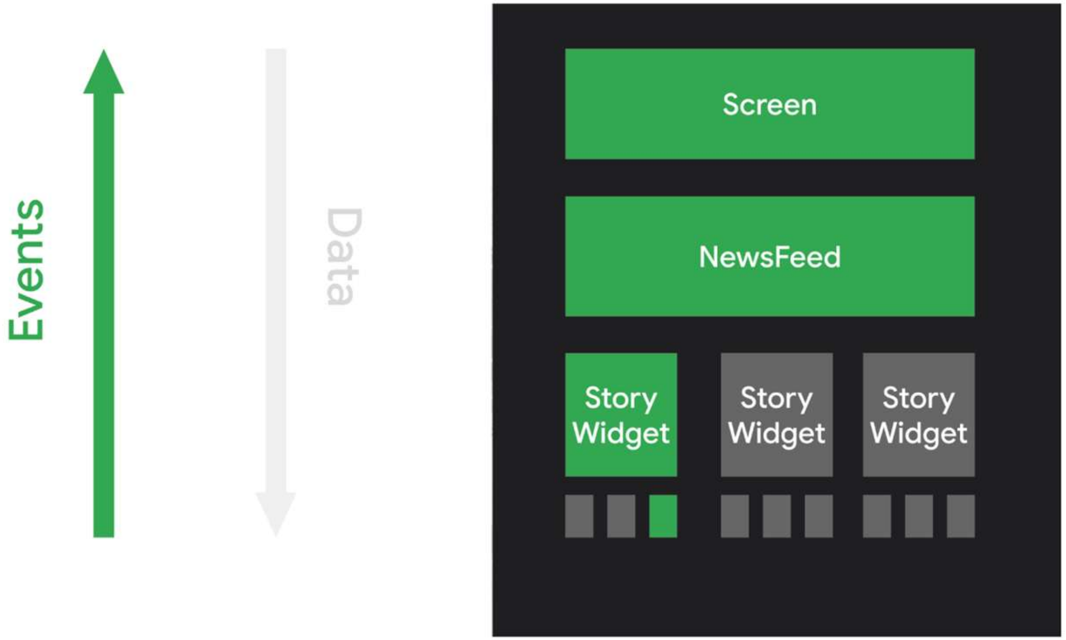
- Recommended design principle:

1. **The UI elements should be stateless**
2. **Dependency Injection: provide dependencies (necessary objects) as argument, instead of creating them directly within it.**
3. **ViewModel holds only the state of the UI elements**

UDF is obtained using two key data types
MutableStateFlow<T> → container of values that can change (state=has a value, flow=emits notification)
StateFlow<T> → Read only version of a mutable (not mutable)

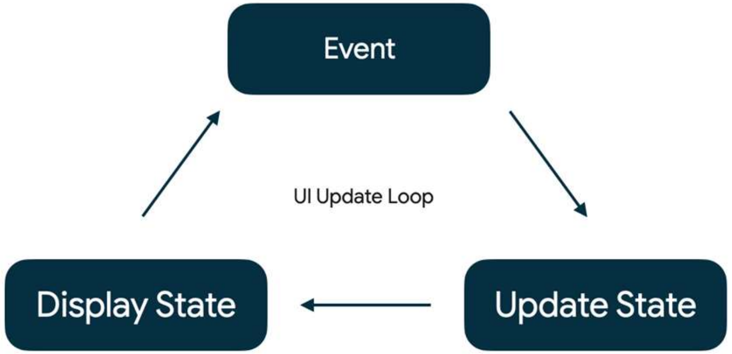
COMPOSABLE FUNCTIONS

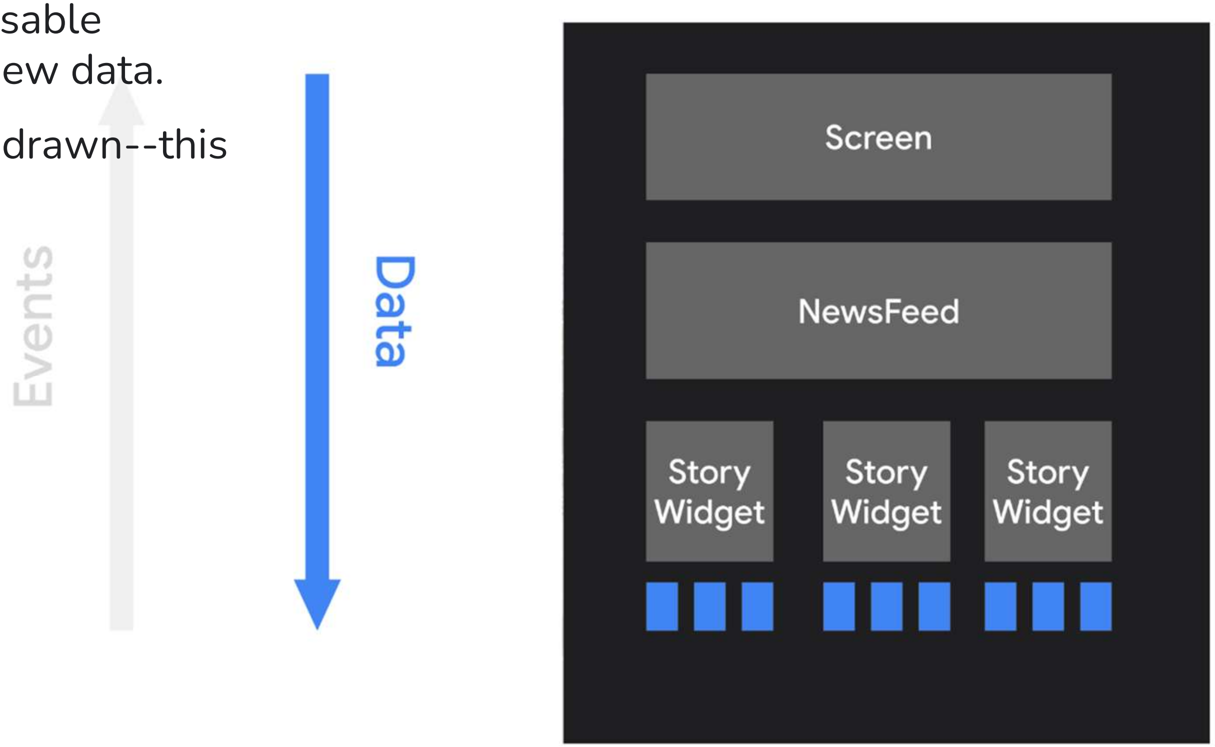
- When the user interacts with the UI, UI raises events such as onClick.
- Those events should notify the app logic, which can then change the app's state.
- Note: In this figure the data is on top

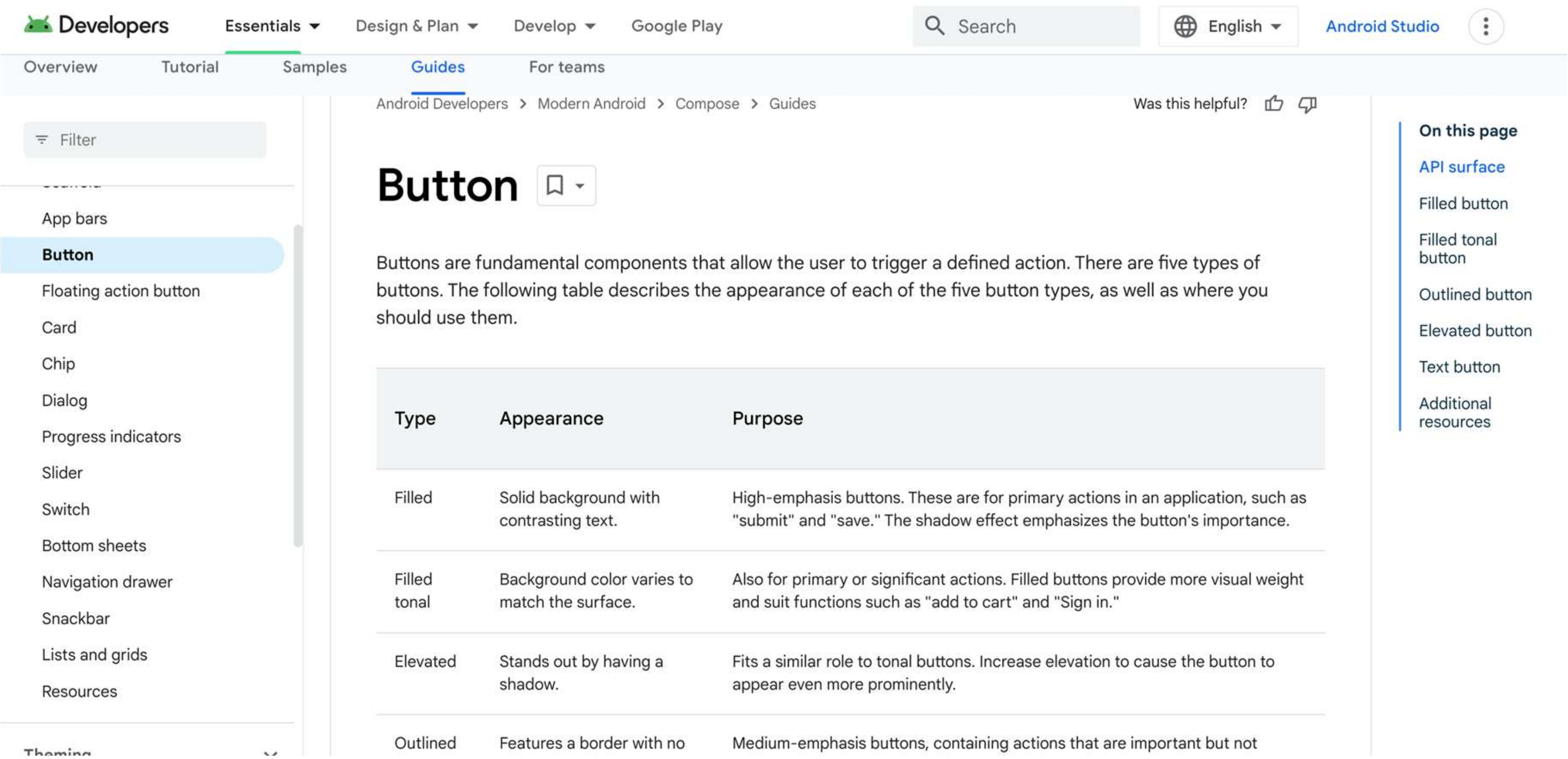


UDF

- Data flow in one direction (in this figure
- When the state changes, the composable functions are called again with the new data.
- This causes the UI elements to be redrawn--this process is called ***recomposition***.



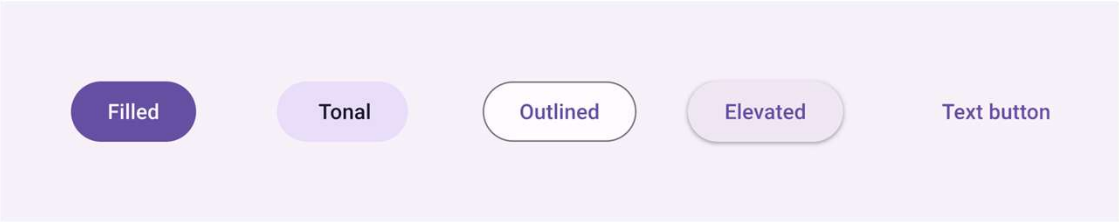




Button

Buttons are fundamental components that allow the user to trigger a defined action. There are five types of buttons. The following table describes the appearance of each of the five button types, as well as where you should use them.

Type	Appearance	Purpose
Filled	Solid background with contrasting text.	High-emphasis buttons. These are for primary actions in an application, such as "submit" and "save." The shadow effect emphasizes the button's importance.
Filled tonal	Background color varies to match the surface.	Also for primary or significant actions. Filled buttons provide more visual weight and suit functions such as "add to cart" and "Sign in."
Elevated	Stands out by having a shadow.	Fits a similar role to tonal buttons. Increase elevation to cause the button to appear even more prominently.
Outlined	Features a border with no	Medium-emphasis buttons, containing actions that are important but not

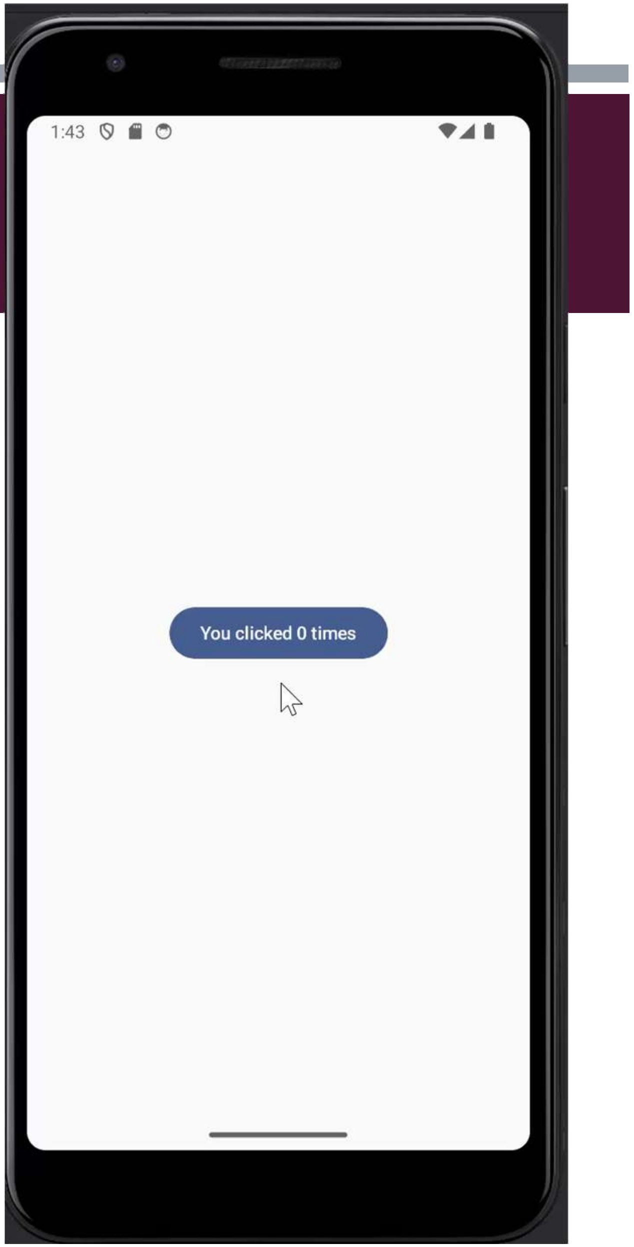


CODE GENERATION VIA PROMPT

- The code produced by the agent, depends on how one details the requests, paraphs with some example, so do drive the reply
- This implies to know the architectural principal of the language
- For example, a prompt like “*write code for a stateful button*” will not generate any viewmodel

CODING

- Write code with a stateful button



BASIC EXAMPLE (STATEFUL BUTTON)

- 1 ViewModel has a private and editable status: _count: MutableStateFlow.
- 2. ViewModel exposes a public and unmodifiable state to the outside world: count: StateFlow.
- 3. UI (CounterScreen) uses .collectAsState() To read and listen to count.
- 4. The user clicks the Button.
- 5. The event counterViewModel.incrementCount() is called .
- 6. ViewModel uses .update { ... } To securely change your private state _count.
- 7. The change of_count is automatically issued by count.
- 8. collectAsState() in the UI receives the new value and forces a recomposition.
- 9. UI updates to show the new number.