Kliensalkalmazások

Android 5 – Jetpack Compose

2024. 05. 06.

Gazdi László

gazdi.laszlo@aut.bme.hu

Automatizálási és
Alkalmazott
Informatikai Tanszék

Miről volt szó az előző órán?

- Fragmentek
- Navigation Component
- Listakezelés: RecyclerView
- Perzisztens adattárolási lehetőségek
  - Egyszerű kulcs-érték tár: SharedPreferences
  - Adatbázistámogatás, SQLite
  - ORM: Room
  - Filekezelés
- Adattárolás a felhőben
- Content Provider

Tartalom

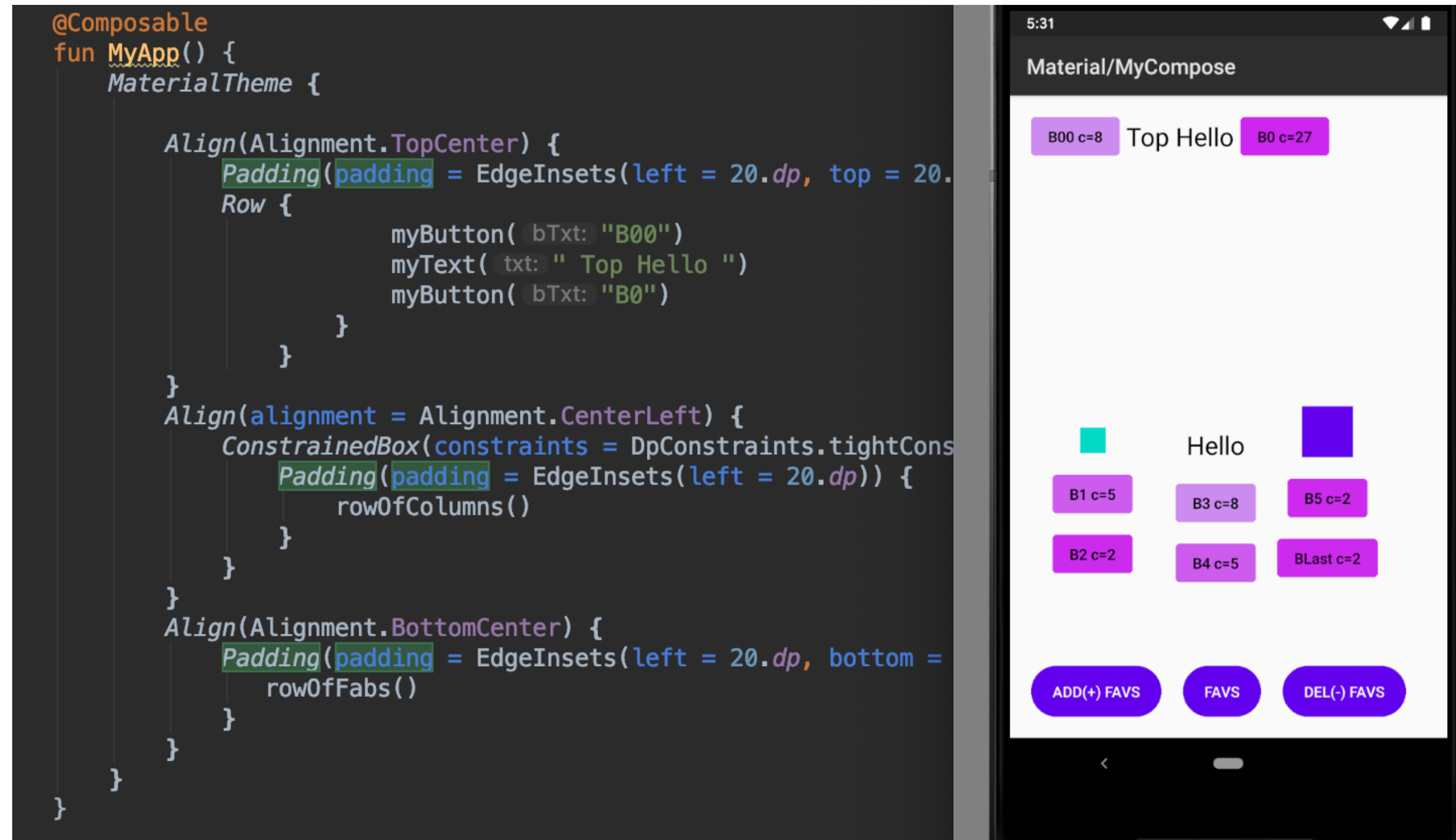
- Compose alapok
- Compose Layout-ok
- Modifier-ek
- Compose alapelvek
- Recomposition
- ViewModel
  - MVVM
  - MVI
- Navigáció
- Dialógusok
- Listák
- Szálkezelés, coroutine-ok
- Flow-k
- View és Compose átjárhatóság

Jetpack Compose

- Modern toolkit UI készítéshez
  - A fejlesztő Kotlin kóddal leírja a UI paramétereit és a Compose motor generálja a felületet
  - Könnyű a UI frissítése az alkalmazás állapota alapján (dated automatically)
  - Erőforrások (pl. képek) ugyanúgy használhatóak
- Csomag része
  - Eszközök
  - Kotlin API-k
- Előnyök
  - Kevesebb kód
  - Nincs szükség XML layout-ra
  - Nincs szükség UI widgetek készítésére
  - A UI elemek kódból készíthetők
  - Könnyebb az újrafelhasználhatóság
  - Kompatibilis a meglévő UI toolkit-el (XML – layout megoldással)

Előkövetelmények

- Legújabb Android Studio
- Letöltési javaslat:
  - JetBrains Toolbox
  - https://www.jetbrains.com/toolbox-app/
- Compose eszközök
- Valós idejű preview
- Új projekt:
  - Empty Compose Activity template



Composable függvények

- Segítségükkel készíthetők UI elemek Kotlin kódból
  - Alakzat és adat függőség megadása
- @Composeble annotáció a függvények elején
- Egymásba ágyazható UI elemek

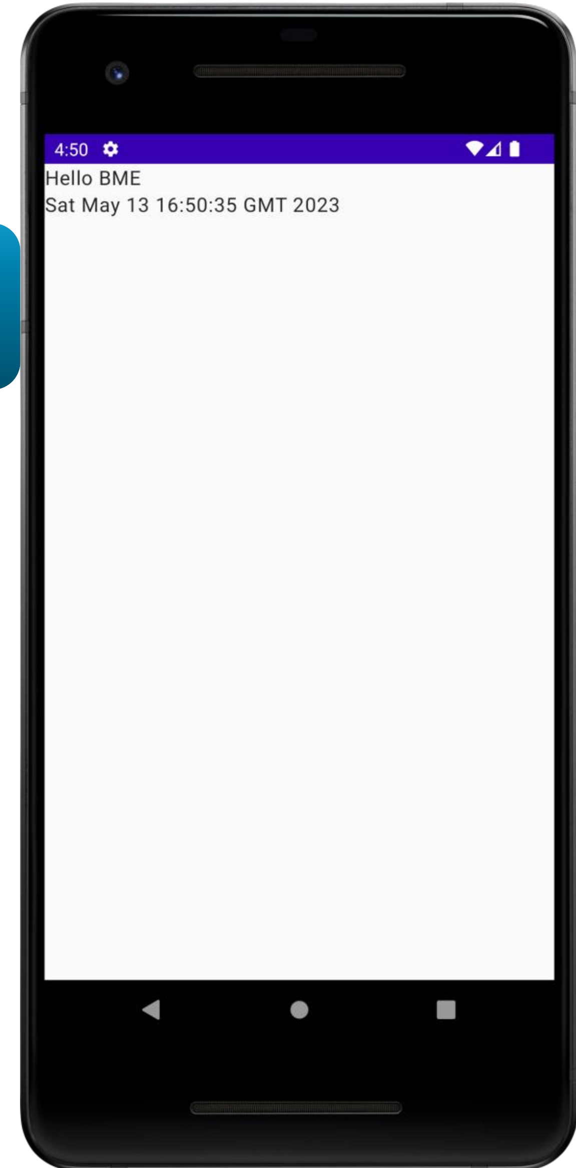
Jetpack Compose – HelloWorld

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

@Composable
fun HelloWorld() {
  Column {
    Text("Hello BME")
    Text(Date(System.currentTimeMillis()).toString())
  }
}
```

ComponentActivity leszármazott

setContentView() helyett

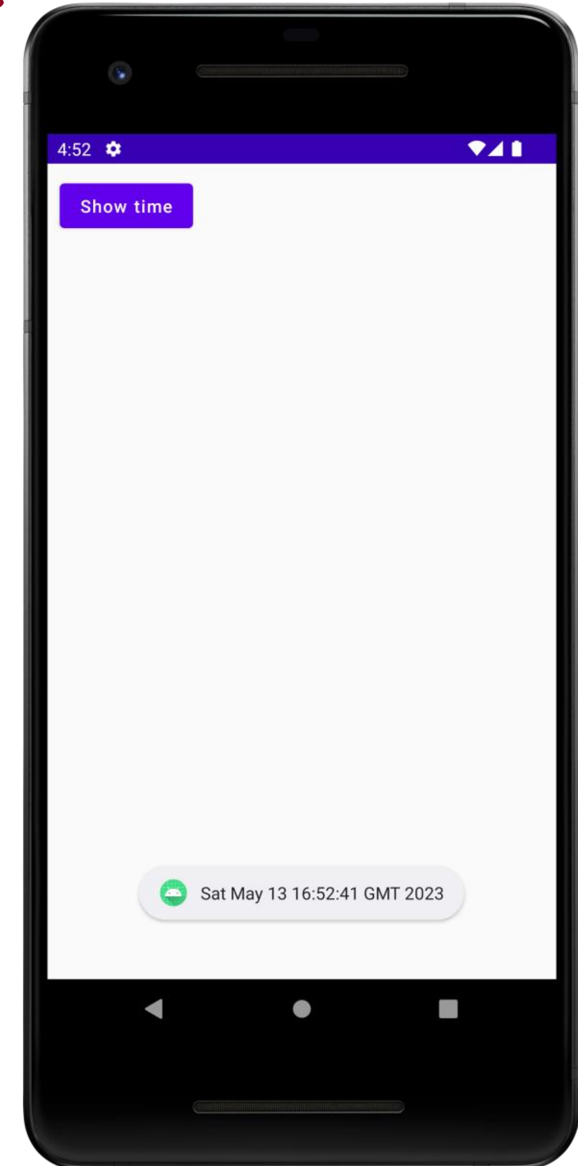
@Composable függvény



Jetpack Compose – Események kezelése

```kotlin
class DemoActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      MaterialTheme {
        ButtonShowTime(this)
      }
    }
  }
}

@Composable
fun ButtonShowTime(context: Context) {
  Button(
    onClick = {
      Toast.makeText(context, Date(System.currentTimeMillis())
        .toString(), Toast.LENGTH_LONG).show()
      },
    modifier = Modifier.padding(Dp(10f))
  ) {
    Text("Show time")

  }
}
```

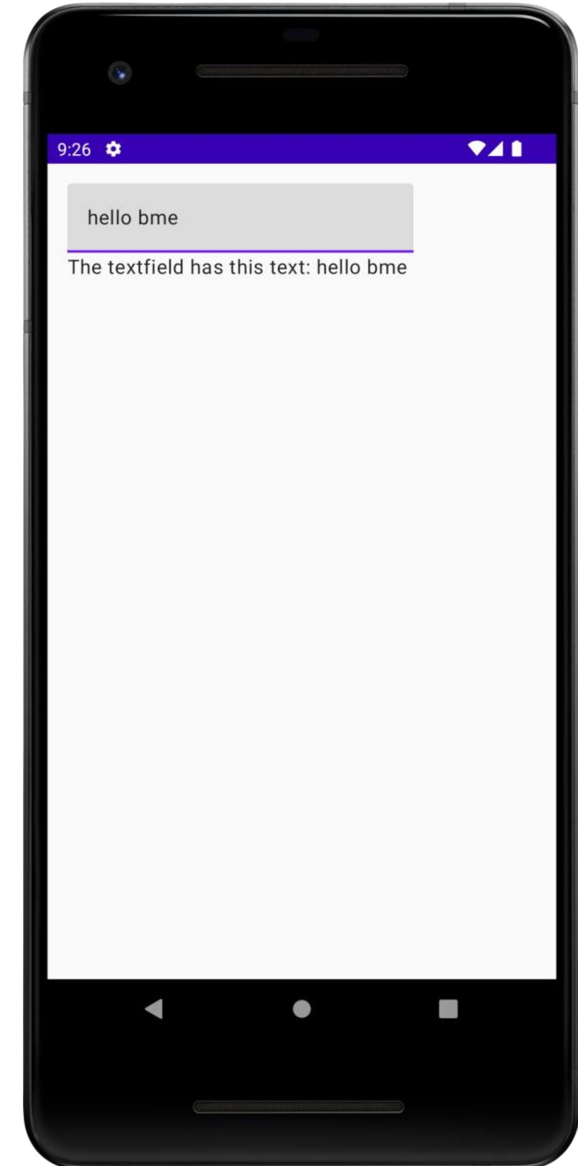
Eseménykezelő



Jetpack Compose – TextField (input)

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

@Composable
fun TextFieldDemo() {
  Column(Modifier.padding(16.dp)) {
    val textState = remember { mutableStateOf(TextFieldValue()) }
    TextField(
      value = textState.value,
      onValueChange = { textState.value = it }
    )
    Text("The textfield has this text: " + textState.value.text)
  }
}
```

Egyszerű állapotkezelés



Jetpack Compose – Fragment példa

```
class ExampleFragment : Fragment() {
  override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View {
    return ComposeView(requireContext()).apply {
      setContent {
        MaterialTheme {
          Text("Hello BME Compose!")
        }
      }
    }
  }
}
```

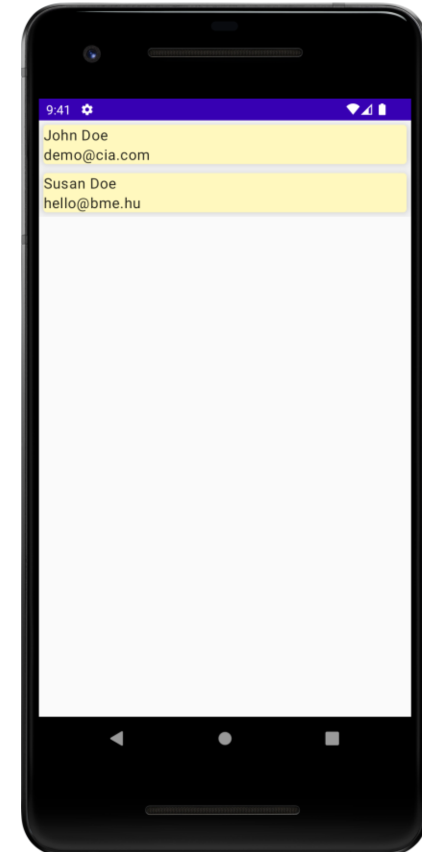
Jetpack Compose – listák kezelése

```kotlin
data class Student(var name: String, var email: String)

@Composable
fun StudentsList(students: List<Student>) {
  LazyColumn() {
    items(students) {
      StudentCardSimple(it.name, it.email)
    }
  }
}
@Composable
fun StudentCardSimple(name: String, email: String) {
  Card(
    modifier = Modifier
      .fillMaxWidth()
      .padding(5.dp),
    elevation = 10.dp,
    backgroundColor = Color(255,248,190)
  ) {
    Column {
      Text(name)
      Text(email)
    }
  }
}
class DemoActivity : ComponentActivity() {
  var students = mutableListOf<Student>(
    Student("John Doe", "demo@cia.com"),
    Student("Susan Doe", "hello@bme.hu")
  )

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

Adat modell

Lista

Lista elem nézet

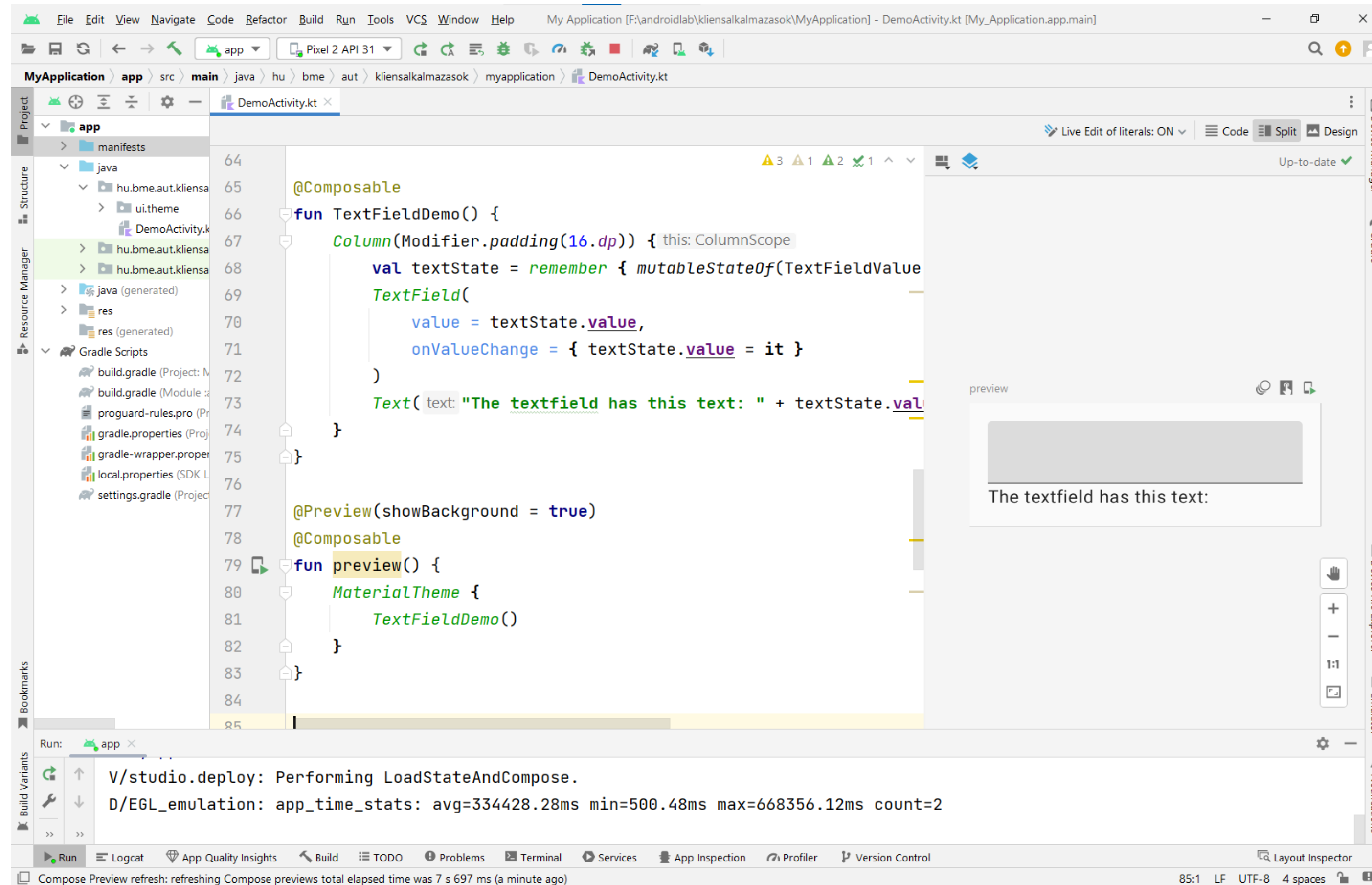


Mi nem igaz a Jetpack Compose-ra?

A. Könnyű a UI frissítése az alkalmazás állapota alapján
B. A UI elemek kódból készíthetők
C. A Composable függvények egymásba ágyazhatók
D. Minden osztályhoz tartozik egy xml layout

Preview: side-by-side

- *@Preview* segítségével a kód írásakor kapunk Android Studioban előnézeti képet
- Több @*Preview* is lehet egy projektben, paraméterei is lehetnek



Compose UI elvek

- A Compose az állapotokat UI elemekké transformálja:
  - Elemek “kompzíciója”
  - Elemek Layout-ja
  - Egyedi rajzolás

1. **Composition**: *What* UI to show. Compose runs composable functions and creates a description of your UI.
2. **Layout**: *Where* to place UI. This phase consists of two steps: measurement and placement. Layout elements measure and place themselves and any child elements in 2D coordinates, for each node in the layout tree.
3. **Drawing**: *How* it renders. UI elements draw into a Canvas, usually a device screen.

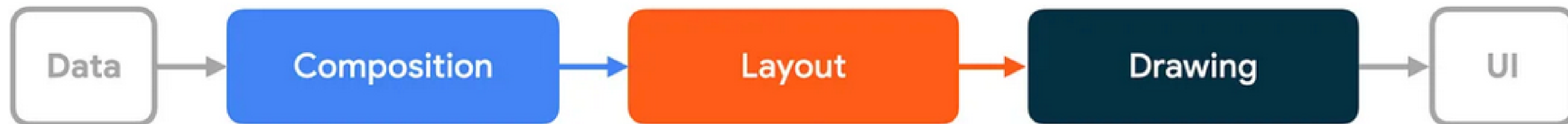


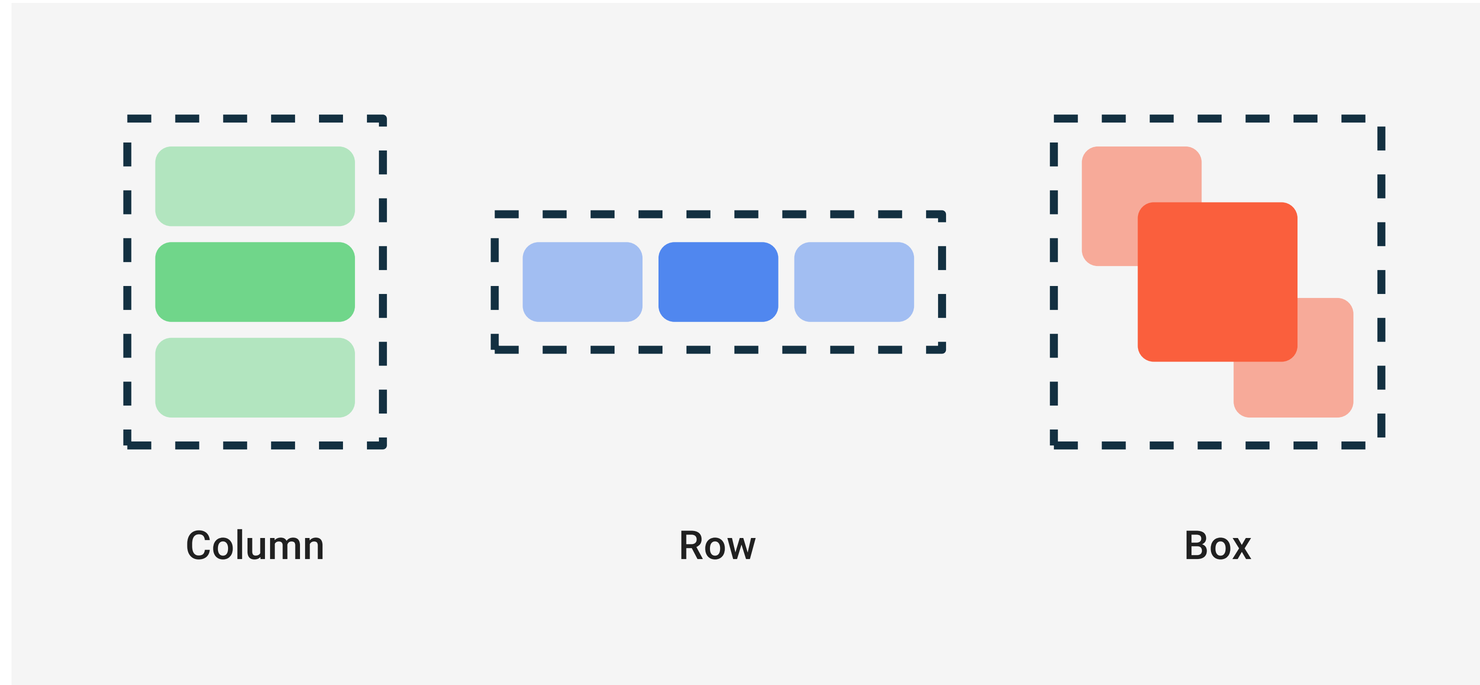
Figure 1. *The three phases in which Compose transforms data into UI.*

Layout-ok

- Egy @*Composable* függvény egy vagy több felhasználói felületi elemet bocsáthat ki
- A rendezésre vonatkozó útmutatás (elrendezések) nélkül előfordulhat, hogy a Compose rosszul rendezi az elemeket
- Alapértelmezés szerint egymásra rakja a szövegelemeket, így olvashatatlanná válnak
- A Compose használatra kész elrendezések gyűjteményét biztosítja, és megkönnyíti az egyéni elrendezések definiálását

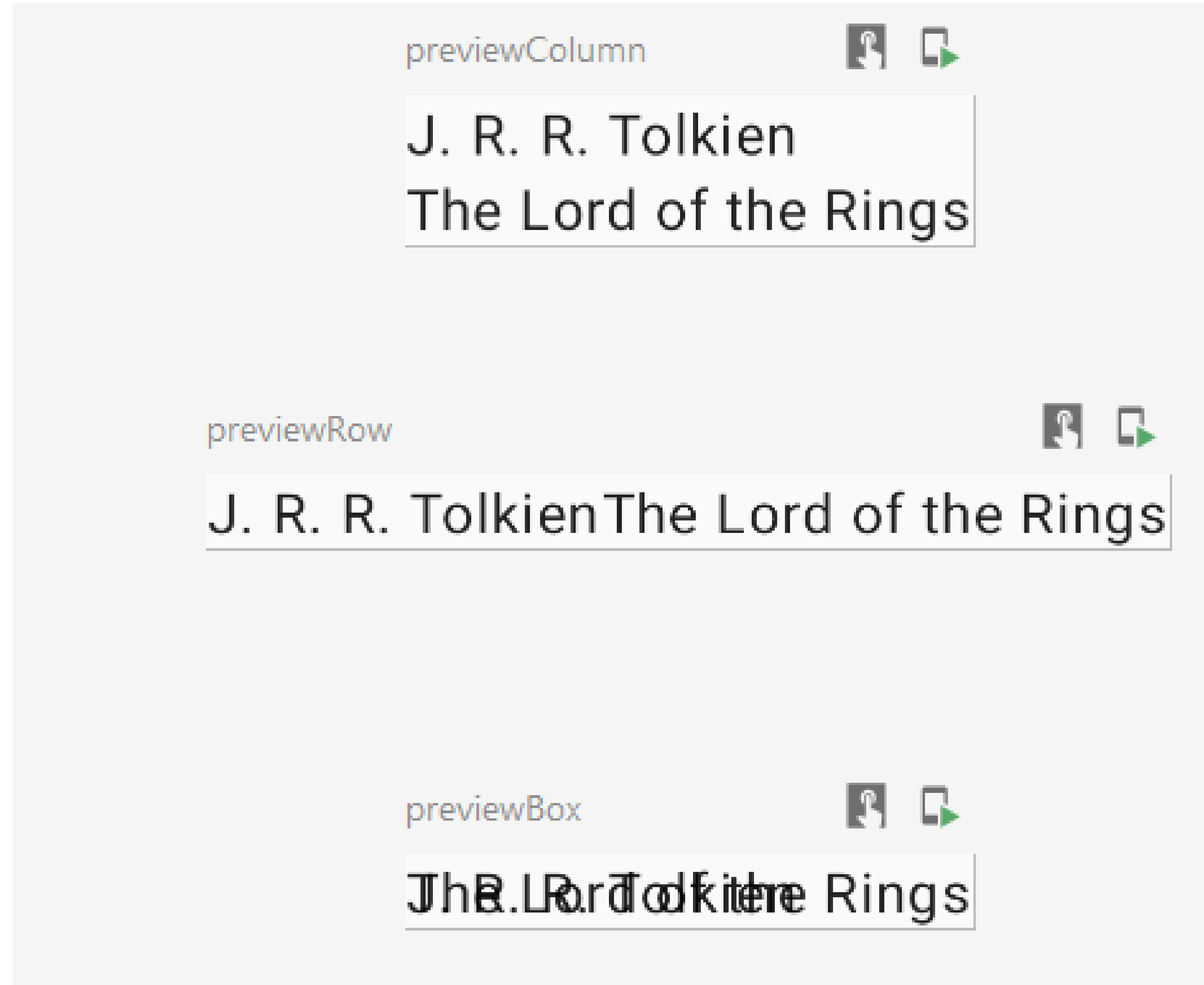
```
@Composable
fun AuthorCard() {
  Text("J. R. R. Tolkien")
  Text("The Lord of the Rings")
}
```

Alap Layout-ok



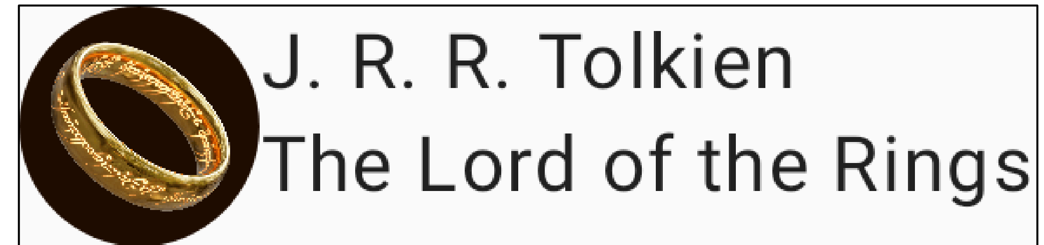
Layout példa

```
@Composable
fun AuthorCardColumn() {
   Column {
      Text("J. R. R. Tolkien")
      Text("The Lord of the Rings")
   }
}

@Composable
fun AuthorCardRow() {
   Row {
      Text("J. R. R. Tolkien")
      Text("The Lord of the Rings")
   }
}

@Composable
fun AuthorCardBox() {
   Box {
      Text("J. R. R. Tolkien")
      Text("The Lord of the Rings")
   }
}
```



Layout példa

```
@Composable
fun AuthorCard() {
  Row {
    Image(
      painter = painterResource(id = R.drawable.tlotr),
      contentDescription = "The Lord of the Rings",
      contentScale = ContentScale.Fit,
      modifier = Modifier
        .size(50.dp)
        .clip(CircleShape)
      )
    Column {
      Text("J. R. R. Tolkien")
      Text("The Lord of the Rings")
    }
  }
}
```



Súlyok Column és Row esetén

```
@Composable
fun ColumnWeightDemo() {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .border(width = 1.dp, color = Color.Blue),
        verticalArrangement = Arrangement.spacedBy(10.dp)
    ) {
        Text(
            text = "J. R. R. Tolkien",
            modifier = Modifier
                .weight(1.0f)
                .fillMaxWidth()
                .background(Color.Cyan)
        )
        Row(modifier = Modifier
            .weight(3.0f)) {
            Text(
                text = "The Lord of the Rings",
                modifier = Modifier
                    .weight(2.0f)
                    .fillMaxHeight()
                    .background(Color.Yellow)
            )
            Text(
                text = "Book",
                modifier = Modifier
                    .weight(1.0f)
                    .fillMaxHeight()
                    .background(Color.Green)
            )
        }
    }
}
```

Alignment és Arrangement

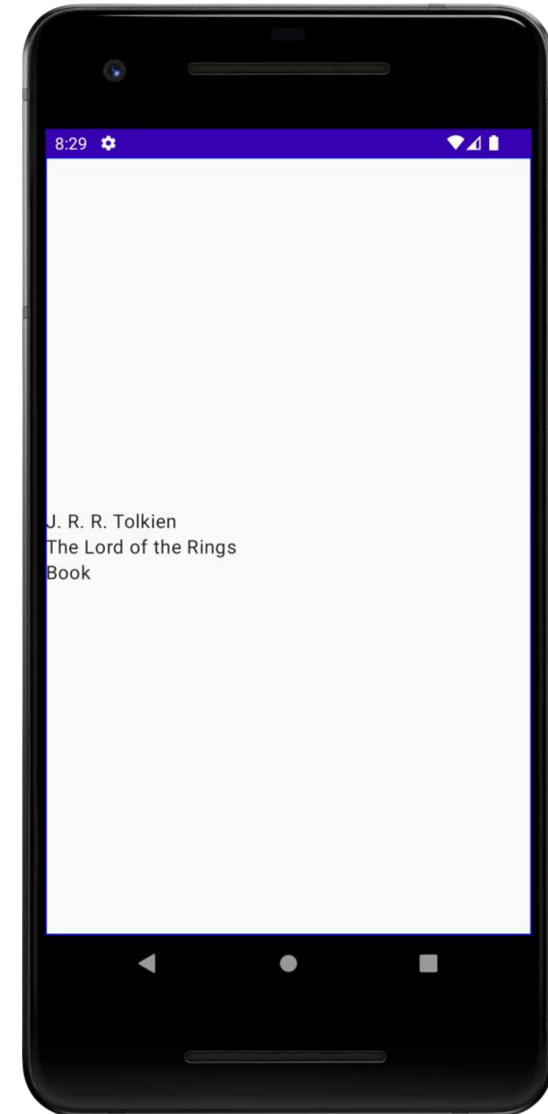
- Egy **Row**-n belül a “gyerek” elemek elhelyezésére használhatók a **horizontalArrangement** és **verticalAlignment** argumentumok
- **Column** esetén a **verticalArrangement** és **horizontalAlignment** használható

```
@Composable
fun AuthorCard() {
    Row(
        modifier = Modifier.padding(0.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.End
    ) {
        Image(
            painter = painterResource(id = R.drawable.tlotr),
            contentDescription = "The Lord of the Rings",
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .size(50.dp)
                .clip(CircleShape)
        )
        Column {
            Text("J. R. R. Tolkien")
            Text("The Lord of the Rings")
        }
    }
}
```

Fő tengely: Arrangement

- Column:
  - Fő tengely a vertikális (függőleges)
  - verticalArrangement arugmentum
- Row:
  - Fő tengely a horizontális (vízszintes)
  - horizontalArrangement argumentum

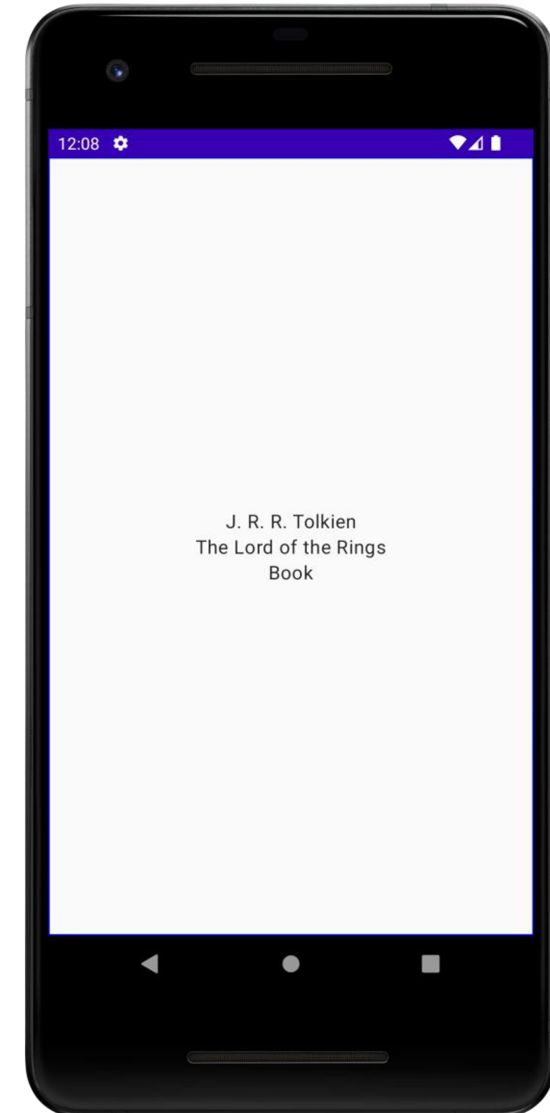
```
@Composable
fun VerticalDemo() {
    Column(
        modifier = Modifier.fillMaxSize()
            .border(width = 1.dp, color = Color.Blue),
        verticalArrangement = Arrangement.Center
    ) {
        Text(text = "J. R. R. Tolkien")
        Text(text = "The Lord of the Rings")
        Text(text = "Book")
    }
}
```



Alignment

- Column:
  - A gyerek vízszintes elhelyezése
- Row:
  - A gyerek függőleges elhelyezése

```
@Composable
fun VerticalDemo() {
    Column(
        modifier = Modifier.fillMaxSize()
            .border(width = 1.dp, color = Color.Blue),
        verticalArrangement = Arrangement.Center
    ) {
        Text(text = "J. R. R. Tolkien")
        Text(text = "The Lord of the Rings")
        Text(text = "Book")
    }
}
```



Box

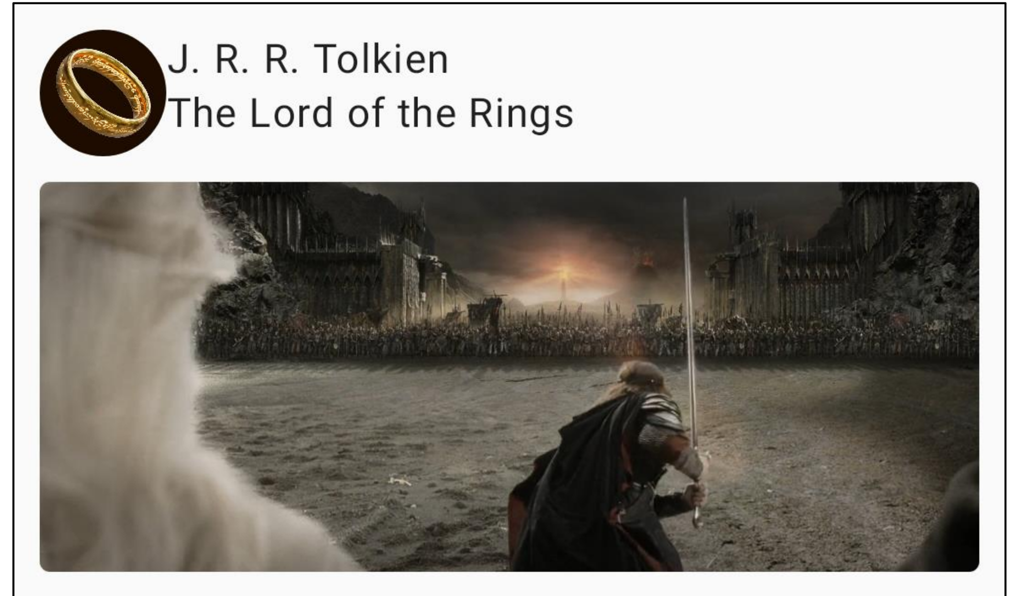
```kotlin
@Composable
fun BoxDemo() {
    Box(
        modifier = Modifier
            .size(400.dp)
            .border(width = 2.dp, color = Color.Magenta)
            .padding(0.dp)
    ) {
        Image(
            painter = painterResource(
                id =
                R.drawable.tlotr
            ),
            contentDescription = "The Lord of the Rings",
            contentScale = ContentScale.Crop,
            modifier = Modifier
                .fillMaxSize()
                .clip(CircleShape)
        )
        Text(
            text = "TopStart",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.TopStart)
        )
        Text(
            text = "TopCenter",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.TopCenter)
        )
        Text(
            text = "TopEnd",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.TopEnd)
        )
        Text(
            text = "CenterStart",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.CenterStart)
        )
        Text(
            text = "CenterEnd",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.CenterEnd)
        )
        Text(
            text = "Center",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.Center)
        )
        Text(
            text = "BottomStart",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.BottomStart)
        )
        Text(
            text = "BottomEnd",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.BottomEnd)
        )
        Text(
            text = "BottomCenter",
            color = Color.Cyan,
            modifier = Modifier.align(Alignment.BottomCenter)
        )
    }
}
```

Modifier-ek (módosítók) bevezetés

- A módosítók a Composable elemek díszítésére vagy kiegészítésére szolgálnak
- A módosítók elengedhetetlenek az elrendezés testreszabásához
- A módosítók hasonló szerepet töltenek be, mint a nézetalapú elrendezések elrendezési paraméterei
- A módosítók típusbiztonságot nyújtanak, és egyértelművé teszik, hogy mi áll rendelkezésre és alkalmazható egy adott elrendezéshez

Modifier példa

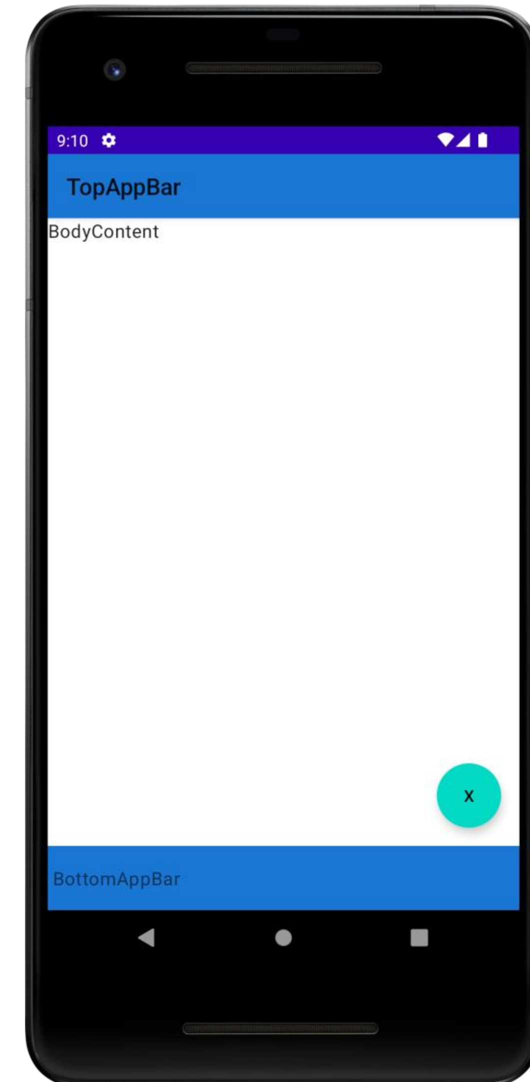
```
@Composable
fun Book() {
    Column(
        modifier = Modifier
            .padding(10.dp)
            .fillMaxWidth()
            .clickable(onClick = {})
    ) {
        AuthorCard()
        Spacer(modifier = Modifier.size(10.dp))
        Card {
            Image(
                painter = painterResource(id = R.drawable.running),
                contentDescription = "Forrest Gump",
                contentScale = ContentScale.FillWidth,
                modifier = Modifier
                    .fillMaxWidth()
            )
        }
    }
}
```



Slot API példa – Scaffold (Material UI alap konténer)

- Üres helyek/konténerek (slotok) a felületen, amik kitöltésre várnak

```
@Composable
fun ScaffoldDemo() {
    val materialBlue700= Color(0xFF1976D2)
    val scaffoldState = rememberScaffoldState(rememberDrawerState(DrawerValue.Open))
    Scaffold(
        scaffoldState = scaffoldState,
        topBar = { TopAppBar(title = {Text("TopAppBar")},backgroundColor = materialBlue700) },
        floatingActionButtonPosition = FabPosition.End,
        floatingActionButton = { FloatingActionButton(onClick = {}){
            Text("X")
        } },
        drawerContent = { Text(text = "drawerContent") },
        content = { Text("BodyContent") },
        bottomBar = { BottomAppBar(backgroundColor = materialBlue700) {
            Text("BottomAppBar") } }
    )
}
```



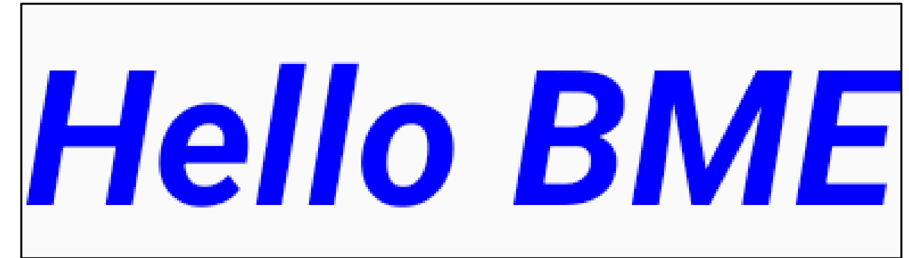
Miért “nem működik” a TextField (input)?

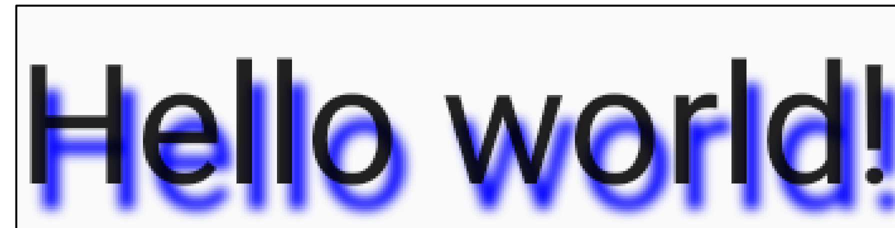
- Compose-ban az olyan elemek mint a TextField nem frissülnek automatikusan
- Egy Composable elem csak akkor frissül, ha az állapot amitől “függ” változik

```
@Composable
fun GameScreen() {
  var text by rememberSaveable { mutableStateOf("") }

  Column() {
    OutlinedTextField(
      value = text,
      onValueChange = {
        text = it
      },
      modifier = Modifier.fillMaxWidth()
    )
    OutlinedButton(onClick = {}) {
      Text(text = "Guess")
    }
    Text(text = "Result: ")
  }
}
```

Text stílus

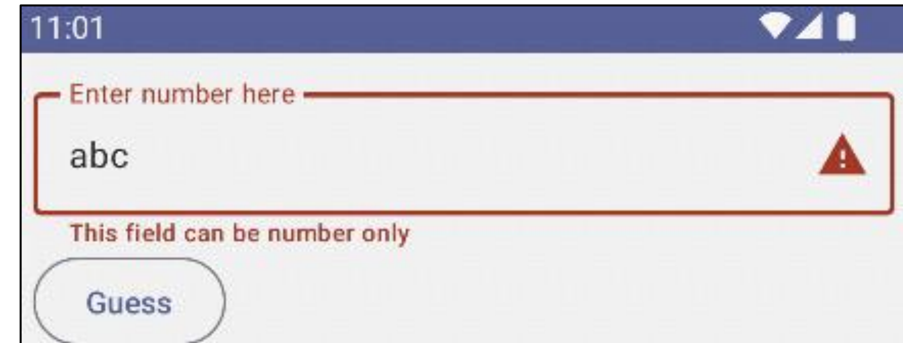
```
Text(
    text = "Hello BME",
    fontSize = 30.sp,
    fontWeight = FontWeight.Bold,
    fontStyle = FontStyle.Italic,
    color = Color.Blue
)

@Composable
fun TextShadow() {
    val offset = Offset(5.0f, 10.0f)
    Text(
        text = "Hello world!",
        style = TextStyle(
            fontSize = 24.sp,
            shadow = Shadow(
                color = Color.Blue, offset = offset, blurRadius = 3f
            )
        )
    )
}
```





Hiba jelzése TextField esetén

```
OutlinedTextField(
  modifier = modifier.fillMaxWidth(),
  value = text,
  isError = inputErrorState,
  onValueChange = {
    text = it
    validate(text)
  },
  label = { Text("Enter number here") },
  singleLine = true,
  keyboardOptions = KeyboardOptions(keyboardType = KeyboardType.Decimal),
  trailingIcon = {
  if (inputErrorState)
    Icon(Icons.Filled.Warning, "error", tint = MaterialTheme.colorScheme.error)
  },
)
```



Material komponensek

- https://foso.github.io/Jetpack-Compose-Playground/

</> Jetpack Compose Playground   Search   Jetpack Compose Playgro... v1.4.3 2.8k 242

Jetpack Compose Playground
Jetpack Compose
- Overview
- Components
- General
- Desktop
- Cookbook
- Compose UI
- Compose Projects
- Contributing
- Resources
- iOS
- Web

🤔 Documentation issue? Report or edit

Jetpack Compose Playground

Introduction

This is a community-driven collection of Jetpack Compose documentation/examples/tutorials and demos.

Show some ❤️ and star the repo to support the project

Star 2.8k | Fork 242 | Watch 51 | Follow @jklingenberg_

MIT | PRs welcome | Compose 1.4.3 | Featured in androidweekly.net Issue #431 | contributors 46 | Tweet

What is Jetpack Compose?

"Jetpack Compose is a modern toolkit for building native Android UI. Jetpack Compose simplifies and accelerates UI development on Android with less code, powerful tools, and intuitive Kotlin APIs. "

Announcing Jetpack Compose 1.0   Link másol...

Table of contents

Introduction
Show some and star the repo to support the project
What is Jetpack Compose?
Composable of the week!! 🎉
New to Compose?
- Animation
- Layouts
- Foundation
- Material
Looking for tutorials/sample code
Contributing
📜 License

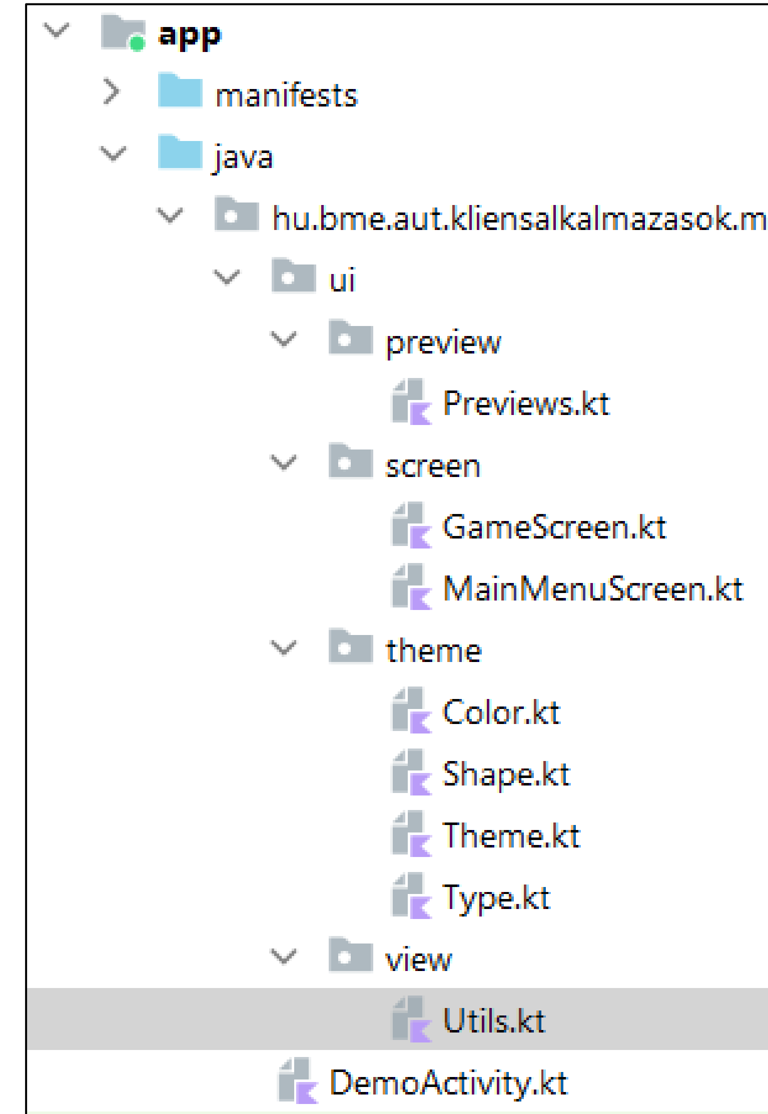
Melyik nem igaz Jetpack Compose esetén?

A. Három fő layout van
B. Row esetén a verticalArrangement argumentum használatos függőleges elhelyezéshez
C. A Row és a Column is súlyozható
D. A Modifier-ek az elrendezés testreszabásához használhatóak

Compose – Alapelvek

Kódszervezés – package-ek

- preview package:
  - Központi @Preview-k
- screen:
  - Fő képernyők
- theme:
  - Design, színek, stílusok, stb.
- view:
  - Segéd nézetek, például dialógusok
  - Újrafelhasználható nézetek
- (navigation):
  - Navigációs konstansok (később)
- root:
  - Activity(k)
- …



Jetpack Compose definíciók

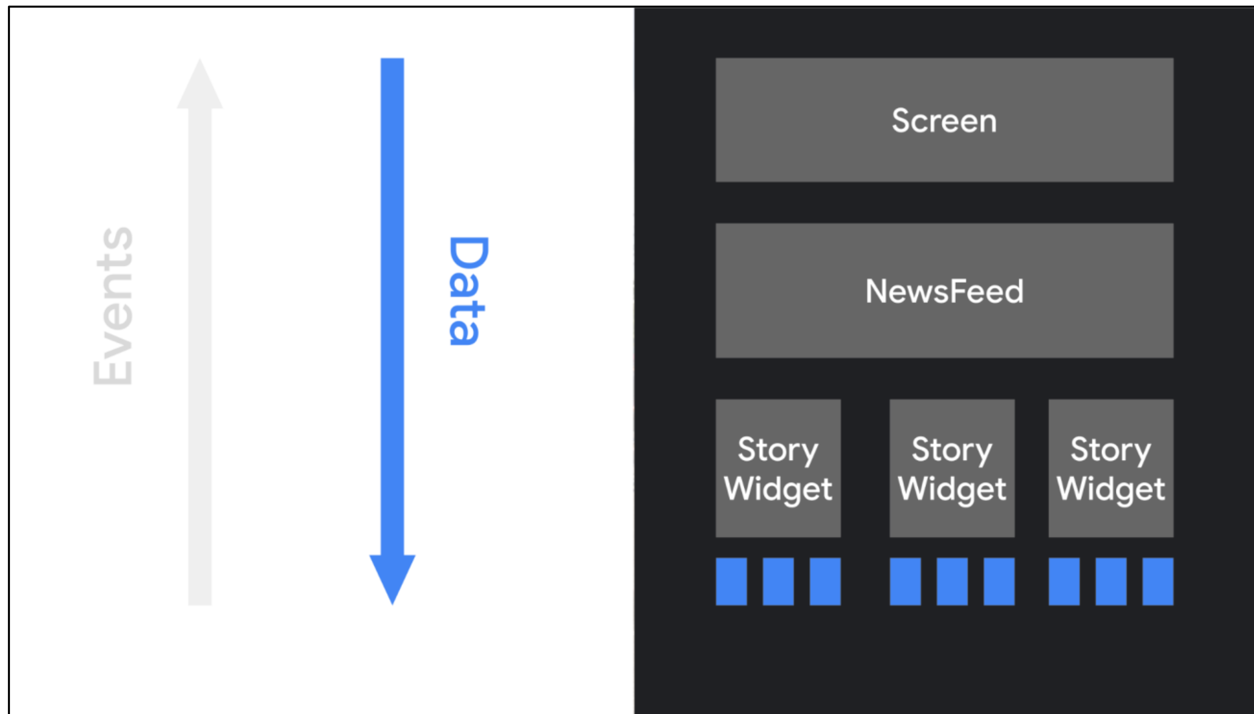
- **Composition**: a Jetpack Compose által készített felhasználói felület leírása
- **Initial Composition**: kompozíció létrehozása a Composable-k első futtatásával
- **Recomposition**: a kompozíciók újbóli futtatása a kompozíció frissítéséhez, amikor az adatok/állapotok megváltoznak

Deklaratív paradigma

- A UI elemek (widgetek) nem érhetők el objektumként
  - Nem érjük el direkt a Text-eket, Button-okat, stb., nincs ID alapú referencia
- A felhasználói felület frissítése úgy történik, hogy ugyanazt az Comosable függvényt hívjuk meg különböző argumentumokkal
- Az állapotot tipikusan a *ViewModel* (később) tárolja és adja a @Composable függvényeknek
- A Composable függvények (@Composable) felelősek az aktuális állapot átalakításáért a felületen (UI renderelés állapottól függően) minden alkalommal amikor egy állapot megváltozik

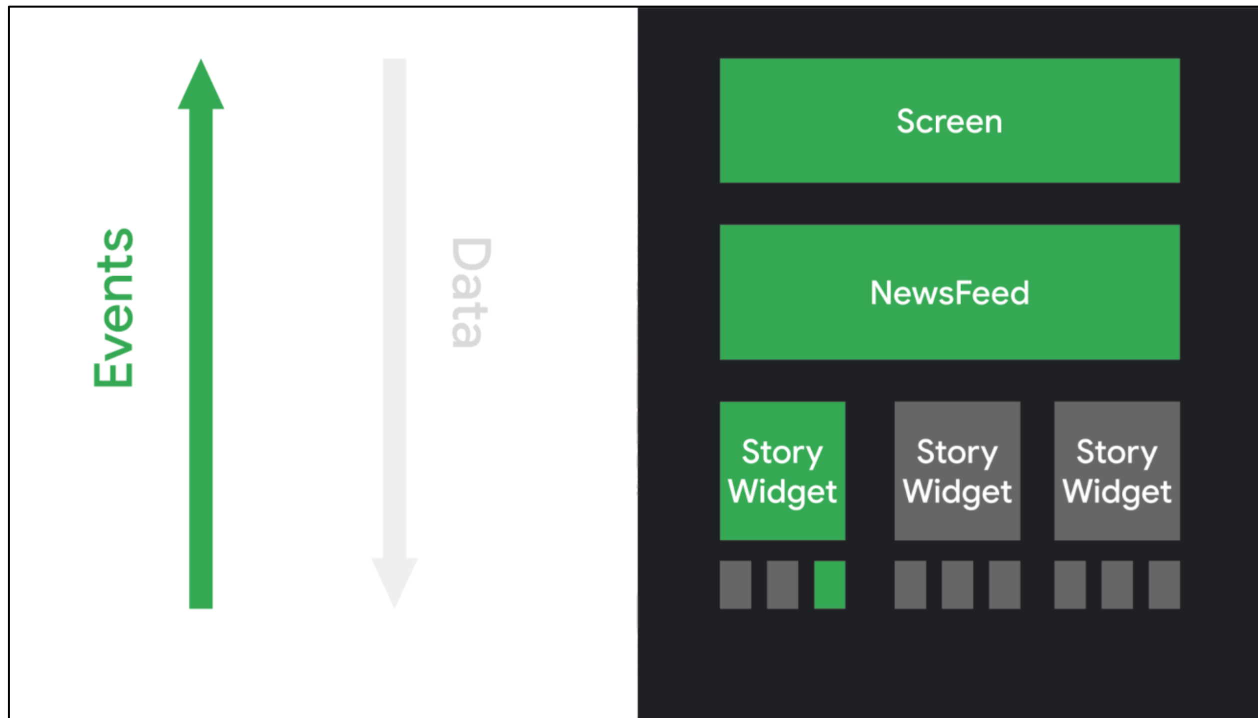
Adatok átadása a widget hierarchiának

- Az alkalmazáslogika adatokat biztosít a legfelső szintű *Composable* függvénynek
- Ez a függvény az adatok alapján írja le a felhasználói felületet más *Composable* elemek meghívásával, és átadja a megfelelő adatokat ezeknek az *Composable*-öknek lefelé a hierarchiában



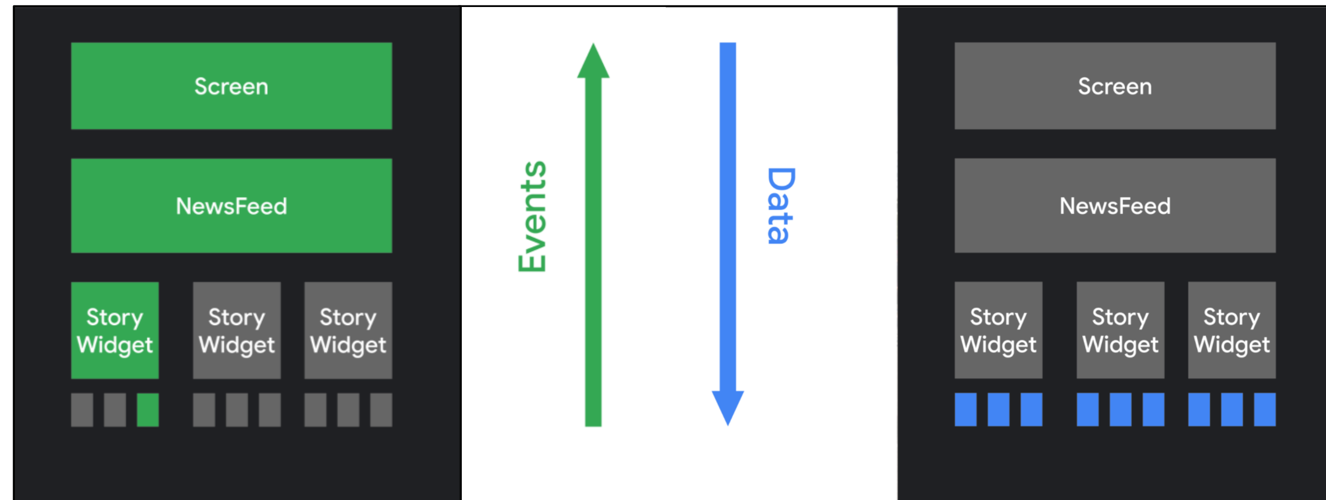
Recomposition UI esemény hatására

- A felhasználó interakcióba lépett egy felhasználói felületi elemmel, ami eseményt váltott ki
- Az alkalmazáslogika válaszol az eseményre
- A Composable függvények szükség esetén automatikusan újra meghívásra kerülnek új paraméterekkel.



Recomposition

- Felhasználói interakció (például onClick)-> esemény kiváltása (event)
- Események értesítik az üzleti logikát (alkalmazás logika)
- Megváltozik egy vagy több állapot **(state change)**
- Állapotváltozás -> minden kapcsolódó Composable újrahívódik (**amelyik függ ettől az állpottól / használja ezt az állapotot**)
    - Ezt hívjuk rekompozíciónak (**Recomposition**)



Dinamikus felhasználói felület

- Composable függvények Kotlinban
- Dinamikus függvények
  - Nem úgy mint a “statikus” XML

```
@Composable
fun WelcomePeople(people: List<String>){
  for (name in people) {
    Text("Hello $name")
  }
}
```

- Szokásos vezérlők használhatók:
  - If-el eldönthető mit mutassunk
  - Ciklusokkal több elem elhelyezhető a felületre
  - Segéd függvények használhatók
  - Kotlin nyelv teljes flexibilitása használható

Recomposition

- A Compose alkalmazásban a Composable widget újra rajzolódik, ha egy módosított értéktől függ
  - Az érték lehet egy szám vagy egy összetett állapot, amely ebben a felületi elemben jelenik meg
- A teljes felhasználói felületi fa újrakomponálása számítási szempontból költséges lehet
  - CPU teljesítmény
  - Akkumulátor
- DE! a Recomposition intelligens és csak azokat rajzolja újra, ahol az állapotot használta az adott Composable elem

Melyik igaz Jetpack Compose esetén?

A. A Recomposition mindig a teljes felületet újrarajzolja

B. Az állapot mindig az Activity-ben tárolódik

C. Egy Composable függvényben ciklus is használható az elemek felületre helyezéséhez

D. A Composable függvény argumentumai futás közben nem változhatnak

Hatékony Recomposition

Hatékony Recomposition

- A rendszer csak azokat a függvényeket vagy lambdákat hívja meg, amelyek esetleg megváltoztak, és kihagyja a többit
- Kihagy minden olyan függvényt vagy lambdát, amelynek nincsenek módosított paraméterei

- A Compose függvények 5 szabálya:
  - 1. Az Composable függvények bármilyen sorrendben végrehajthatók.
  - 2. Az Composable függvények párhuzamosan hajthatók végre.
  - 3. A Recomposition kihagyja a lehető legtöbb Composable függvényt és lambdát.
  - 4. A Recomposition optimista, és leállítható menet közben.
  - 5. Egy Composable függvény nagyon gyakran is futhat/ismétlődhet, olyan gyakran is, mint egy animáció minden képkockája.

Hatékony Recomposition - Az Comosable függvények bármilyen sorrendben végrehajthatók

- Minden Composable függvénynek önállónak kell lennie
- *Nem készíthetünk elő valamit a StartScreen()-ben és használhatjuk a MiddleScreen() vagy EndScreen()-en*

```
@Composable
fun ButtonRow() {
  MyFancyNavigation {
    StartScreen()
    MiddleScreen()
    EndScreen()
  }
}
```

Hatékony Recomposition - A Composable függvények párhuzamosan futtathatók

- Kihasználja a több mag előnyeit
- Alacsonyabb prioritású Composable függvények futtatása, amelyek nincsenek a képernyőn
- Az alkalmazás megfelelő viselkedésének biztosítása érdekében a Composable függvényeknek nem lehetnek mellékhatásai

```kotlin
@Composable
@Deprecated("Example with bug")
fun ListWithBug(myList: List<String>) {
  var items = 0

  Row(horizontalArrangement = Arrangement.SpaceBetween) {
    Column {
      for (item in myList) {
        Text("Item: $item")
        items++
      }
    }
    Text("Count: $items")
  }
}
```

Kerüljük el!
Mellékhatás a Column kirajzolásakor

```kotlin
@Composable
fun ListComposable(myList: List<String>) {
  Row(horizontalArrangement = Arrangement.SpaceBetween) {
    Column {
      for (item in myList) {
        Text("Item: $item")
      }
    }
    Text("Count: ${myList.size}")
  }
}
```

A Recomposition kihagyja a lehető legtöbb Composable függvényt és lambdát

- A Compose mindent megtesz annak érdekében, hogy csak a frissítendő részeket rajzolja újra
- Kihagyhatja egyetlen gomb összeállítható elemének újra futtatását anélkül, hogy végrehajtaná a felhasználói felület fáját felette vagy alatt
- Minden Composable függvény és lambda önmagától újra rajzolódhat

Recomposition példa

```
@Composable
fun Counter(name: String) {
   var clickCount by remember{mutableStateOf(1)}
   Column() {
      Button(onClick = {
         clickCount++
      }) {
         Log.d("TAG_COMPOSE","recompose occured in button")
         Text(text = "Press me $clickCount")
      }
      Log.d("TAG_COMPOSE","recompose occured outside")
      Text(text = "Hello $clickCount!")
   }
}
```

Ha a $clickCount nincs használva a Button-on belül, akkor a **recomposition kihagyhatja** a Button részt

A Recomposition optimista

- Az Recomposition akkor kezdődik, amikor a Compose úgy gondolja, hogy egy összeállítható anyag paraméterei megváltoztak
- • A Recomposition optimista
  - A Compose azzal kalkulál, hogy befejezi az újra rajzolást, mielőtt a paraméterek ismét megváltoznak
  - Ha egy paraméter megváltozik a Recomposition befejezése előtt, előfordulhat, hogy a Compose megszakítja a kirajzolást, és újraindítja az új paraméterrel
  - A Recomposition megszakításakor a Compose elveti a felhasználói felületi fát az újrarajzolásból
- Fontos, hogy az összes Composable függvény és lambda idempontens és mellékhatás mentes legyen
  - *Idempotens*: A művelet szükség szerinti gyakorisággal megismételhető vagy újra próbálkozhat anélkül, hogy nem kívánt hatásokat okozna

Egy Composable függvény nagyon gyakran is futhat

- Előfordulhat, hogy egy Composable függvény a felhasználói felület animációjának minden képkockája esetén lefut
- Ha a függvény költséges műveleteket hajt végre, például az eszköz tárhelyéről való olvasást, a függvény késéseket okozhat a felhasználói felületen

- -> Ha egy Composable függvénynek adatokra van szüksége, paraméterként érdemes átadni neki
- -> Helyezzük át a költséges munkát egy másik szálba, a Compose-on kívülre, és adja át az adatokat a Composable függvényeknek

Mi igaz Recomposition esetén?

A. Az Composable függvények szekvenciálisan hajthatók végre

B. Az Composable függvények bármilyen sorrendben végrehajthatók

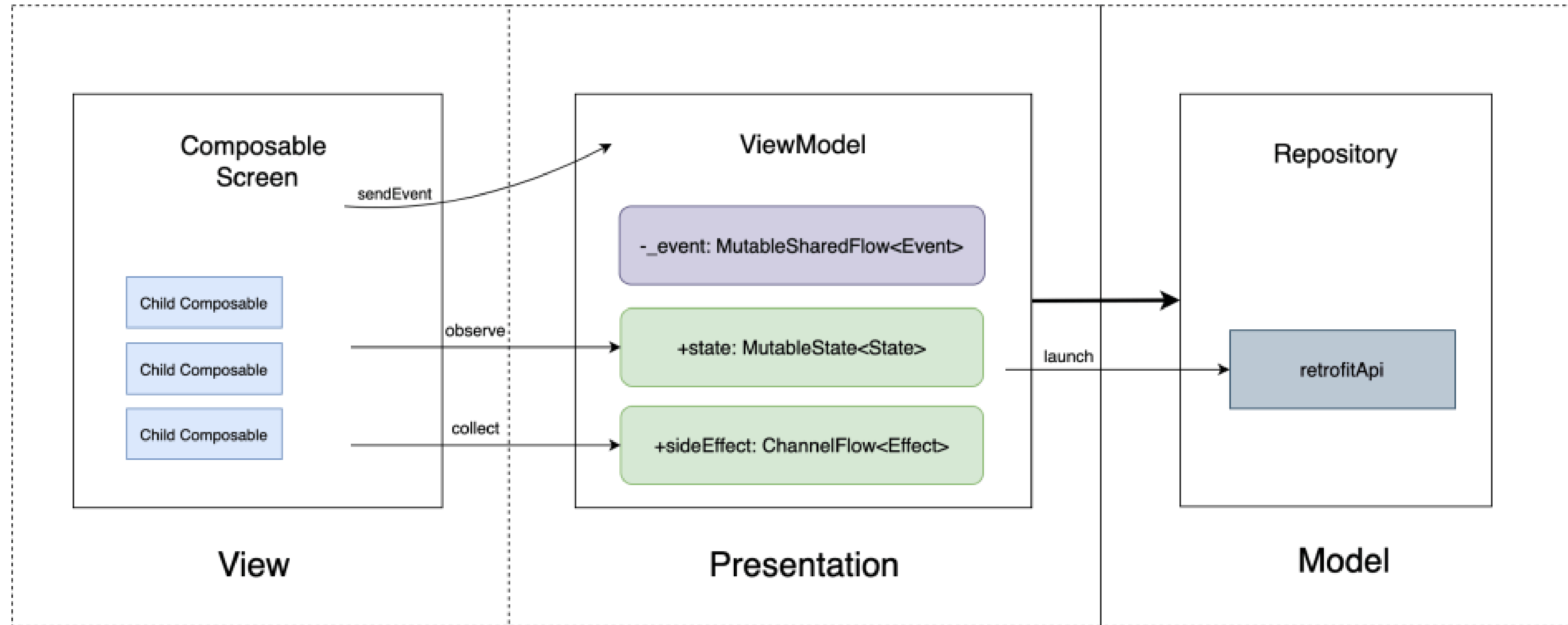
C. A Recomposition minden Composable függvényt és lambdát lefuttat

D. A Recomposition nem állítható le menet közben

ViewModel Compose-ban

Android Alkalmazás Architektúra

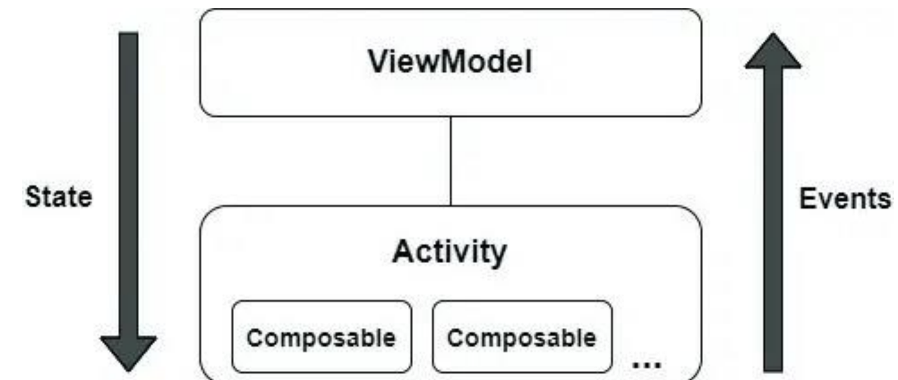
Architektúra minta Compose esetén



https://codingtroops.com/android/compose-architecture-part-1-mvvm-or-mvi-architecture-with-flow/

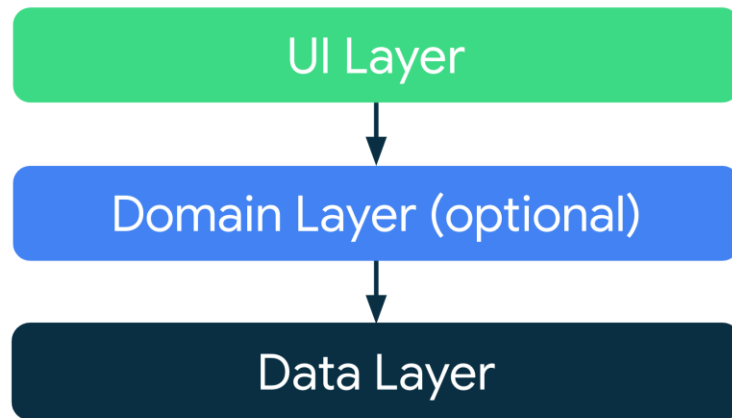
ViewModel alapok

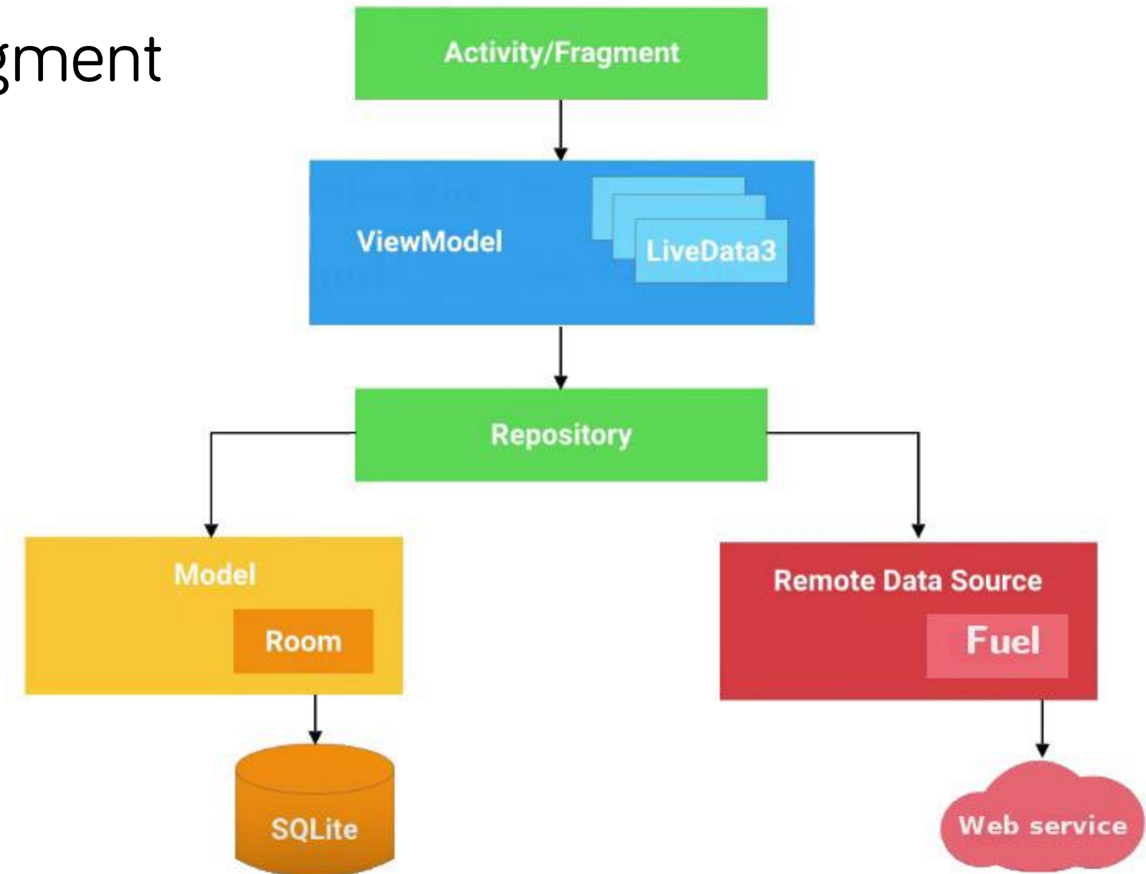
- **Általános elv**: az alkalmazáson belüli különböző felelősségi területek elkülönítése teljesen különálló modulokra ("a felelősségek szétválasztása")
- Ennek a megközelítésnek az egyik kulcsa a ViewModel összetevő
- **A ViewModel olyan adatokat (állapotokat) tárol, amelyeket a felhasználói felület (egy @Composable) megfigyelhet**
- Ez lehetővé teszi, hogy a felhasználói felület reagáljon, ha változás történik a ViewModel-ben tárolt állapotban
- Például egy gomb megnyomásakor (ez egy esemény), a ViewModel-ben egy metódus kerül meghívásra, amely végrehajt valamilyen logikát (pl. lekérdez valamit a DB-ből), és végül frissít egy állapotot, amely Recomposition-t okoz (újrarajzolja a megfelelő részeket a felhasználói felületen)



MVVM Architektúra

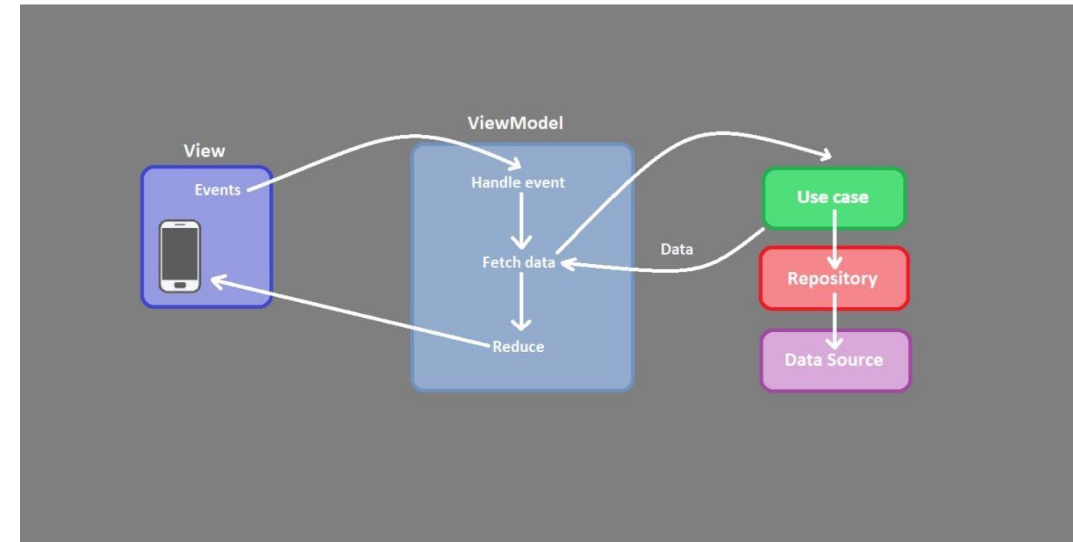
- *Model-View-ViewModel*
  - > ViewModel: Üzleti logika, vezérlés (orchestration)
  - > View: @Composeable vagy Activity/Fragment
  - > Model: entitások/repository





MVI architektúra

- Modell: adatok lekérése és elérhetővé tétele a helyi/távoli forrásból
- View: felhasználói felületi réteg, @Composable függvény
- Intent: cselekedet, esemény végrehajtása (Nem az Android SDK Intent osztálya!)



https://medium.com/@VolodymyrSch/android-simple-mvi-implementation-with-jetpack-compose-5ee5d6fc4908

Állapotok felsorolása (sealed class)

```
sealed class TodoListState {
  object Loading : TodoListState()
  data class Error(val error: Throwable) : TodoListState()
  data class Result(val todoList : List<TodoItem>) : TodoListState()
}
```

MVI – ViewModel példa

```kotlin
class TodoListViewModel() : ViewModel() {
  private val _state =MutableStateFlow<TodoListState>(
    TodoListState.Loading)
  val state = _state.asStateFlow()

  init {
    loadTodos()
  }
```

Flow használat

```kotlin
private fun loadTodos() {
    viewModelScope.launch {
      try {
        _state.value = TodoListState.Loading
        delay(1000)
        //Some todo loading logic
        _state.value = TodoListState.Result(
          todoList = listOf(
            ...
          ),
        )
      } catch (e: Exception) {
        _state.value = TodoListState.Error(e)
      }
    }
  }
}
```

Coroutine

Állapot megfigyelése a UI-on

Állapot „Flow” begyűjtése

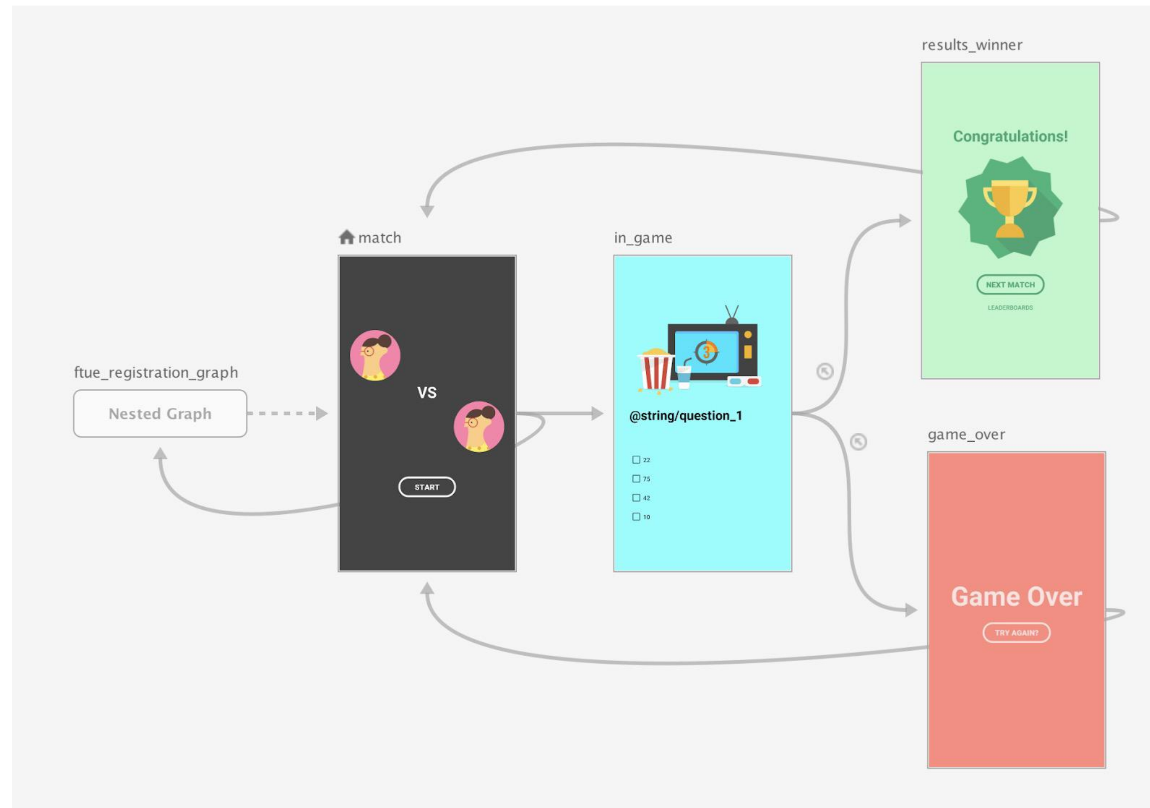
```
@Composable
fun TodoListScreen(viewModel: TodoListViewModel = viewModel()) {
  val state = viewModel.state.collectAsStateWithLifecycle().value
  Box(...) {
    when (state) {
      is TodoListState.Loading -> CircularProgressIndicator(
        color = MaterialTheme.colorScheme.secondaryContainer
      )
      is TodoListState.Error -> Text(text = "Error")
      is TodoListState.Result -> {
        if (state.todoList.isEmpty()) {
          Text(text = "Empty")
        } else {
          //TODO: print list in a LazyColumn
        }
      }
    }
  }
}
```

Állapotfüggő felület megjelenítés

Navigáció Compose esetén

Navigation graph

- Nagyon hasonló az eddig ismert navigációs gráfhoz
- „Screen”-ek között válthatunk egyszerűen



Navigation példa

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      HighLowGameComposeTheme {
        MyAppNavHost(modifier = Modifier.fillMaxSize())
      }
    }
  }
}
```

Navigáció betöltése

```
@Composable
fun MyAppNavHost(
  modifier: Modifier = Modifier,
  navController: NavHostController
rememberNavController(),
  startDestination: String = "mainmenu"
) {
  NavHost(
    modifier = modifier,
    navController = navController,
    startDestination = startDestination
  ) {
    composable("mainmenu") {MainMenuScreen()}
    composable("game") { GameScreen() }
  }
}
```

Kezdő screen

Navigációs vezérlő

Képernyők és azok útvonala „path”

Navigation funkciók

- Paraméterek támogatása

```
NavHost(startDestination = "profile/{userId}") {
  ...
  composable(
    "profile/{userId}",
    arguments = listOf(navArgument("userId") { type = NavType.StringType })
  ) {...}
}
```

- Beágyazott navigációs gráfok
- Deep linkek támogatása
- ... stb.

https://developer.android.com/jetpack/compose/navigation

https://medium.com/@daniel.atitienei/navigate-witharguments-in-jetpack-compose-90846d70bb7f

Navigáció paraméterekkel

```
composable("help/{helptext}",
  arguments = listOf(navArgument("helptext"){ type = NavType.StringType })) {
navBackStackEntry ->

/* Extracting the helptext from the route */
  val text = navBackStackEntry.arguments?.getString("helptext")

/* We check if argument is not null */
  text?.let {
    HelpScreen(
      helpText = text
    )
  }
}
```

Argumentum a „path”-ban

Argumentum típus megadása

A felhasználandó argumentum kiolvasása

Kiolvasott argumentum továbbadása
egy Composable-nek felhasználásra

Opcionális argumentumok

```
composable("help/{helptext}?userId={userId}",
  arguments = listOf(navArgument("helptext"){ type =
NavType.StringType },
  navArgument("userId") {
    defaultValue = 0
    type = NavType.IntType })) { navBackStackEntry ->

  val text = navBackStackEntry.arguments?.getString("helptext")
  val user = navBackStackEntry.arguments?.getInt("userId")
  /* We check if arguments are not null */
  text?.let {
    user?.let {
      HelpScreen(
        helpText = text,
        userId = user
      )
    }
  }
}
```

Opcionális argumentum ? után

Nem kötelező; típusok és nullabilitás itt jelölhető

Opcionális argumentumok használata ugyanúgy mint a path argumentumok esetén

Navigációs argumentumk elérése ViewModel-ben

- savedStateHandle –n keresztül a ViewModel-ben

```
composable("mainmenu") {
  MainMenuScreen(
    onNavigateToGame = {navController.navigate("game?upperBound=20") },
    navController
  )
}

class GameViewModel(savedStateHandle: SavedStateHandle) : ViewModel() {
  var generatedNum by mutableStateOf(0)
  var upperBound by mutableStateOf(3)

  init {
    savedStateHandle.get<Int>("upperBound")?.let {upperBound = it}
    generateNewNum()
  }

  fun generateNewNum() {
    generatedNum = Random(System.currentTimeMillis()).nextInt(upperBound)
  }
}
```

Argumentum beállítása

A saveStateHandle használata amin keresztül az argumentumok elérhetők

Navigációs argumentum kiolvasása és használata

Navigációs konstansok – érdemes külön fileba (path és argumentumok)

```kotlin
const val ROOT_GRAPH_ROUTE = "root"
const val AUTH_GRAPH_ROUTE = "auth"
const val MAIN_GRAPH_ROUTE = "main"

sealed class Screen(val route: String) {
  object Login: Screen(route = "login")
  object Register: Screen(route = "register")
  object Home: Screen(route = "home/{${Args.username}}") {
    fun passUsername(username: String) = "home/$username"
    object Args {
      const val username = "username"
    }
  }
  object Profile: Screen(route = "profile")
  object Settings: Screen(route = "settings")
}
```

Dialógusok Compose-ban

AlertDialog Composable

```
@Composable
fun SimpleAlertDialog(
  show: Boolean,
  onDismiss: () -> Unit,
  onConfirm: () -> Unit
) {
  if (show) {
    AlertDialog(
      onDismissRequest = onDismiss,
      title = { Text(text = "Congratulations!") },
      text = { Text(text = "You have won!") },
      confirmButton = {
        TextButton(onClick = onConfirm){ Text(text = "OK") }
      },
      dismissButton = {
        TextButton(onClick = onDismiss){ Text(text = "Cancel") }
      }
    )
  }
}
```

Megjelenítés állapota és eseménykezelők átvétele argumentumként

Text-en kívül más is lehet

Dialógus használata

```
@Composable
fun GameScreen(
  modifier: Modifier = Modifier,
  gameModel: GameViewModel = viewModel()
) {
  Column(
    modifier = Modifier
      .fillMaxSize()
      .padding(10.dp)) {
…
    var showDialog by rememberSaveable { mutableStateOf(false) }
…
    SimpleAlertDialog(show = showDialog,
      onDismiss = { showDialog = false },
      onConfirm = { showDialog = false }
    )
…
  }
}
```

A dialógus láthatóvá válik ha a showDialog állapota változik

További anyagok

- https://developer.android.com/jetpack/compose/tutorial
- https://developer.android.com/jetpack/compose/state
- https://foso.github.io/Jetpack-Compose-Playground/
- https://joebirch.co/android/exploring-jetpack-compose-card/

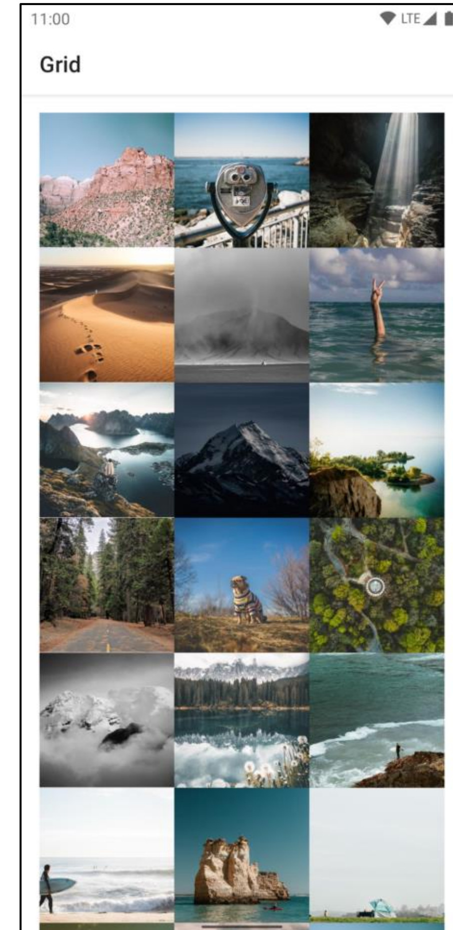
Listák

Lazy loading

- **Csak olyan elemek renderelése amely látható**
  - LazyColumn és LazyRow
  - LazyVerticalGrid, valamint LazyHorizontalGrid
- Fontos!
  - import androidx.compose.foundation.lazy.items manuálisan kell importolni!

Példa – Lazy Loading

```
LazyColumn(
    contentPadding = PaddingValues(horizontal = 16.dp, vertical = 8.dp),
    verticalArrangement = Arrangement.spacedBy(4.dp),
) {
    items(messages) { message ->
        MessageRow(message)
    }
}

LazyVerticalGrid(
    columns = GridCells.Adaptive(minSize = 128.dp)
) {
    items(photos) { photo ->
        PhotoItem(photo)
    }
}
```



Szálkezelés, coroutine-ok

Párhuzamosság vs. konkurencia

- Párhuzamos futás
  - A feladatok egymással egy időben, párhuzamosan futnak
  - Két v több CPU core kell hozzá és ugyanennyi szál, h mindegyik core egy szálat tudjon futtatni egymással egy időben
- Konkurens futás
  - Úgy tűnik, mintha párhuzamosan futnának a feladatok, de valójában egy közös szál halmazon futnak időben felosztva egymás között

Szálak

- Kotlinban a konkurens programozás egyik kelléke

```kotlin
class MyThread: Runnable{
    override fun run() {
        println(Thread.currentThread().getName())
        Thread.sleep(5) //pretend that some heavy calculation happens
    }
}
fun main() {
    Thread(MyThread()).start()
}
```

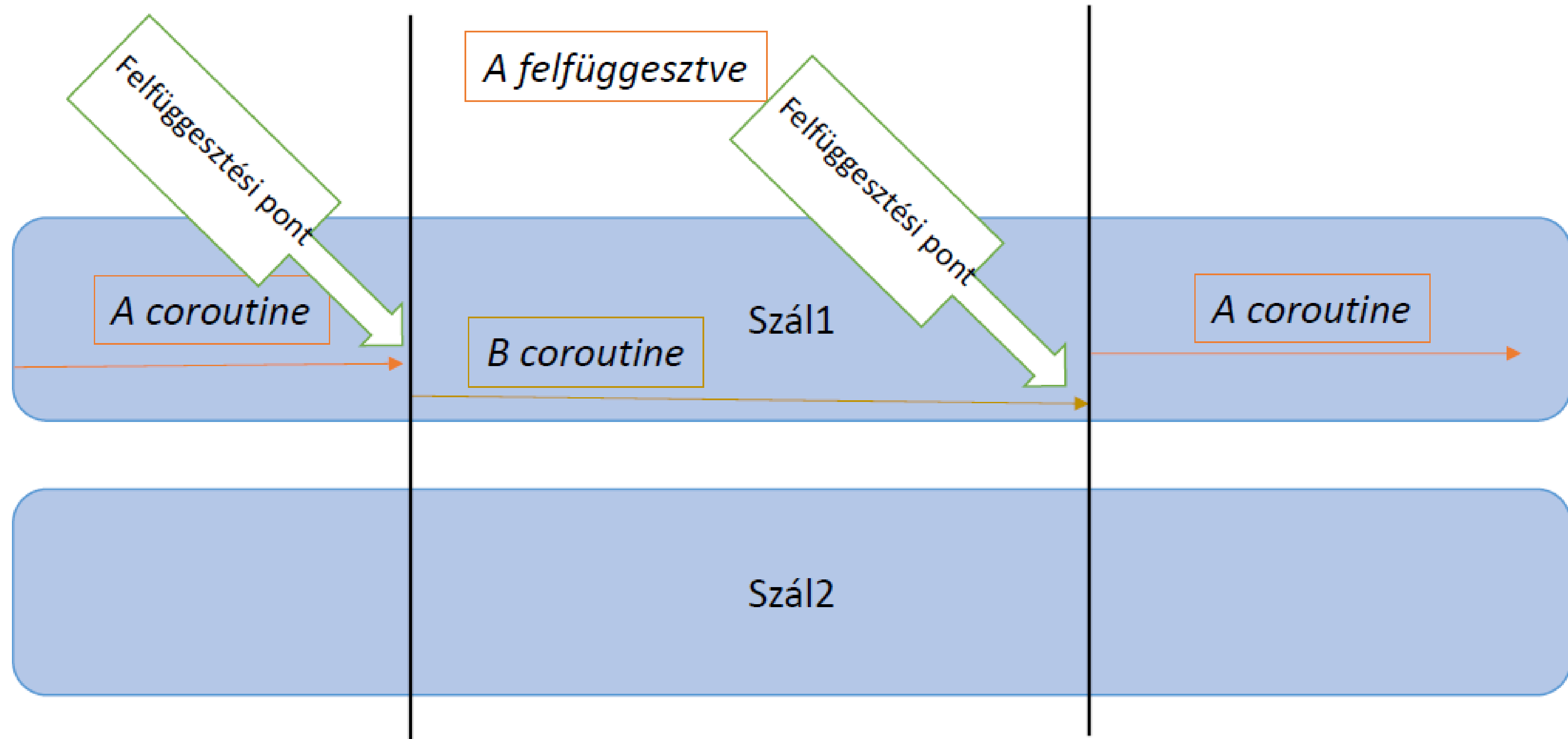
```kotlin
fun main() {
    thread {
        println(Thread.currentThread().getName())
        Thread.sleep(5) //pretend that some heavy calculation happens
    }
}
```

Szálak

- A modern operációs rendszerek támogatják több szál indítását
- Konkurensen futnak
- A szálak azt az illúziót adják, mintha annyi CPU lenne, ahány szál
- Nemcsak annyi szál futhat ahány CPU van, hanem sokat lehet indítani (1 szálnak kb 1 MB memória szükséges JVM-en)
  - Hogyan lehet blokkolni egy szálat?
    - 1. CPU intenzív feladattal (CPU leterhelve)
    - 2. Blokkoló I/O feladattal (nincs kihasználva a CPU, csak várakozik)

Coroutines

- Co + routines = cooperating functions



Coroutines

- A Kotlin a konkurens futtatást implementálta a coroutine-okkal
- Aszinkron, nem blokkoló kód írását teszi lehetővé
- Minden coroutine egy vagy több thread-en fut
- Csak egy parancs futtatható egy adott időben egy thread-en

Coroutine indítás

- viewModelScope egy előre definiált CoroutineScope
- Minden Coroutinnak valamilyen scope hatókörben kell futnia
- A CoroutineScope egy vagy több kapcsolódó coroutine-t kezel
- A launch hozza létre a coroutine-t és elküldi a függvénytörzsének végrehajtását a megfelelő Dispatcher-nek
- Dispatchers.IO jelzi, hogy ezt a coroutine-t az I/O-műveletek számára fenntartott szálon kell végrehajtani (lehet más is, nem kötelező)

```
class LoginViewModel(
  private val loginRepository: LoginRepository
): ViewModel() {

  fun login(username: String, token: String) {
    // Create a new coroutine to move the execution off the UI thread
    viewModelScope.launch(Dispatchers.IO) {
      val jsonBody = "{ username: \"$username\",
        token: \"$token\"}"
      loginRepository.makeLoginRequest(jsonBody)
    }
  }
}
```

Baj, minden a háttérszálon megy, nem lehet UI-t módosítani a Login eredménye után

Coroutine indítás – Megoldás

- withContext(Dispatcher.IO)-val csak adott részt teszünk a háttérszálra

```
class LoginRepository(...) {
  ...
  suspend fun makeLoginRequest(
    jsonBody: String
  ): Result<LoginResponse> {

    // Move the execution of the coroutine to the I/O dispatcher
    return withContext(Dispatchers.IO) {
      // Blocking network request code
    }
  }
}
```

Szál vs Coroutine

- Szál
  - Drága létrehozni
  - Viszonylag sokat létre lehet hozni (erőforrás kérdése)
  - Nem szekvenciális kóddal valósítható meg
- Coroutine
  - Olcsó létrehozni, kevés erőforrást igényel
  - A szálak számának többszörösét lehet létrehozni
  - Szekvenciálisan írható le a programkód

Felfüggesztő függvények

- Suspending (felfüggesztő) függvények
  - Nem blokkolják a hívó szálat
  - El lehet indítani
  - Fel lehet függeszteni (paused)
  - Folytatni lehet felfüggesztés után
  - Amíg fel van függesztve, nem blokkolja a szálat, amin fut
- Gyakran használt dispatcherek, amik meghatározhatják, hogy a felfüggesztő függvények milyen szálon fussanak:
  - Dispatchers.Default – CPU intenzív feladatok
    - Annyi szállal dolgozik a háttérben, ahány CPU core van a gépen
  - Dispatchers.IO – I/O intenzív feladatok

Hasznos olvasnivalók

- Android-os Coroutine dokumentáció:
  - https://developer.android.com/kotlin/coroutines
- Coroutines hivatalos dokumentáció:
  - https://kotlinlang.org/docs/tutorials/coroutines/coroutines-basicjvm.html

Flow

Flow definíció

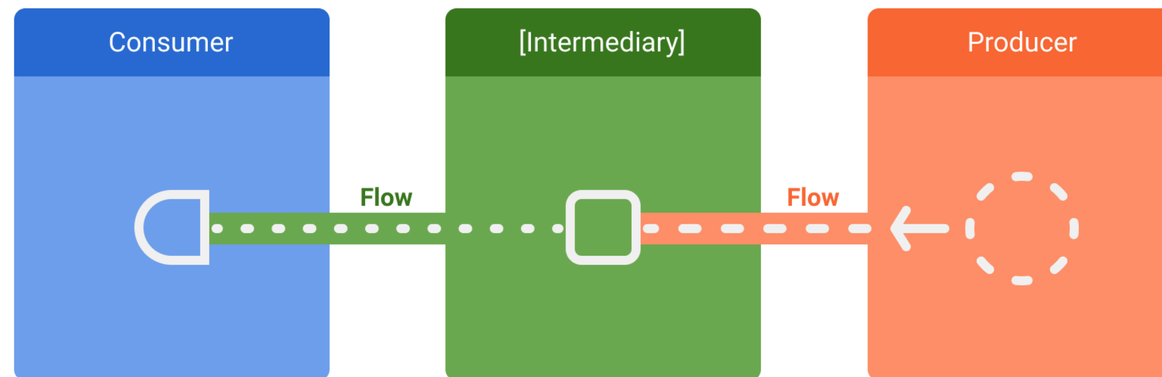
- A coroutinokban a Flow olyan típus, amely egymás után több értéket bocsáthat ki, szemben a suspended függvényekkel, amelyek csak egyetlen értéket adnak vissza.
  - Egy Flow segítségével például élő frissítések fogadhatók egy adatbázisból.
- A Flow-k coroutine-okra épülnek, és több értéket is biztosíthatnak.
- A Flow olyan adatfolyam, amely aszinkron módon számítható ki.
- A kibocsátott (emitted) értékeknek azonos típusúaknak kell lenniük.

Flow definíció

- Például Flow<Int> egy olyan “folyam”, amely egész számokat bocsát ki.
- A Flow nagyon hasonlít egy iterátorhoz, amely értéksorozatot hoz létre, de suspended függvényeket használ az értékek aszinkron előállításához és felhasználásához. Ez például azt jelenti, hogy a folyamat biztonságosan küldhet hálózati kérést a következő érték előállítására anélkül, hogy blokkolná a fő szálat.

A Flow szereplői

- A **producer** létrehozza az adatfolyamhoz hozzáadott adatokat. A coroutinoknak köszönhetően az áramlások aszinkron módon is előállíthatnak adatokat.
- (opcionális) **Intermediater** (közvetítők) módosíthatják a Flow-ban lévő értékeket vagy az egész Flowt.
- A **consumer** felhasználja az adatfolyam (Flow) értékeit.



Flow a gyakorlatban

- A **repository** általában adat előállító **producer** a felhasználói felület számára
- A felhasználói felülettel (UI), mint **consumer** megjeleníti az adatokat
- Más esetekben a felhasználói felület rétege a felhasználói beviteli események előállítója (**producer**), és a hierarchia más rétegei felhasználják (**consume**) azokat. A **producer** és a **consumer** közötti rétegek általában közvetítőként (**intermediates**) működnek, amelyek módosítják az adatfolyamot, hogy azt a következő réteg követelményeihez igazítsák.

Flow – „producer”

```
class NewsRemoteDataSource(
  private val newsApi: NewsApi,
  private val refreshIntervalMs: Long = 5000
) {
  val latestNews: Flow<List<ArticleHeadline>> = flow {
    while(true) {
      val latestNews = newsApi.fetchLatestNews()
      emit(latestNews) // Emits the result of the request to the flow
      delay(refreshIntervalMs) // Suspends the coroutine for some time
    }
  }
}

// Interface that provides a way to make network requests with suspend functions
interface NewsApi {
  suspend fun fetchLatestNews(): List<ArticleHeadline>
}
```

Flow – „consumer”

```kotlin
class LatestNewsViewModel(
  private val newsRepository: NewsRepository
) : ViewModel() {

  init {
    viewModelScope.launch {
      // Trigger the flow and consume its elements using collect
      newsRepository.favoriteLatestNews.collect { favoriteNews ->
        // Update View with the latest favorite news
      }
    }
  }
}
```

https://developer.android.com/kotlin/flow

View motor és Compose átjárhatóság

Compose használata View framework-ből

```
<androidx.compose.ui.platform.ComposeView
android:id="@+id/my_composable"
android:layout_width="wrap_content"
android:layout_height="wrap_content" />
```

```
findViewById<ComposeView>(R.id.my_composable).setContent {
  MaterialTheme {
    Surface {
      Text(text = "Hello!")
    }
  }
}
```

View használata Compose-ból

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <Button
    android:id="@+id/btnDemo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Demo button"/>

  <TextView
    android:id="@+id/tvDemo"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Demo view text"/>
</LinearLayout>
```

```kotlin
@Composable
fun ViewDemo() {
  Column() {
    Text(text = "Hello View Demo from Compose!")
    AndroidView(
      factory = { context ->
        val binding = LayoutDemoBinding.inflate(
          LayoutInflater.from(context))
        binding.btnDemo.setOnClickListener {
          binding.tvDemo.text = Date(
            System.currentTimeMillis()).toString()
        }
        binding.root
      },
      update = { }
    )
  }
}
```

Mi igaz a Jetpack Compose esetén?

A. A navigációhoz FragmentManager-t használunk
B. Egy Coroutine mindig több thread-en fut
C. Egy projektben nem keveredhet a View és a Compose
D. A CoroutineScope egy vagy több kapcsolódó coroutine-t kezel

További anyagok

- Listák és Grid-ek (LazyColumn/Row/Grid):
  - https://developer.android.com/jetpack/compose/lists
- Kotlin coroutines:
  - https://developer.android.com/kotlin/coroutines
- Kotlin flows:
  - https://developer.android.com/kotlin/flow
- Átjárás View és Compose között:
  - https://developer.android.com/jetpack/compose/migrate/interoperability-apis/views-incompose
- Android-os Coroutine dokumentáció:
  - https://developer.android.com/kotlin/coroutines
- Coroutines hivatalos dokumentáció:
  - https://kotlinlang.org/docs/tutorials/coroutines/coroutines-basic-jvm.html

Hogy is volt?

- Mik a Jetpack Compose használatának előnyei?
- Milyen Compose Layout-okat ismer? Jellemezze ezeket!
- Mik a Compose alapelvei?
- Miért hatékony módszer a Recomposition?
- Vázolja fel, milyen architektúrákat ismer Jetpack Compose használata esetén!
- Mire használható a LazyLoading?
- Mik azok Coroutine-ok?
- Mire való a Flow, hogyan használjuk?

Összefoglalás

- Compose alapok
- Compose Layout-ok
- Modifier-ek
- Compose alapelvek
- Recomposition
- ViewModel
  - MVVM
  - MVI
- Navigáció
- Dialógusok
- Listák
- Szálkezelés, coroutine-ok
- Flow-k
- View és Compose átjárhatóság

Köszönöm a figyelmet!