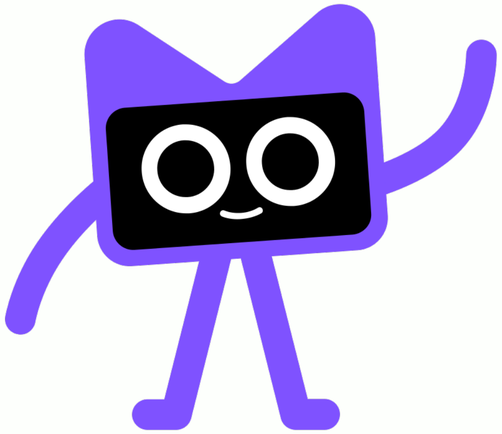
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Gerenciamento de estados e ciclo de vida

QXD0276 - Desenvolvimento de Software para Dispositivos Móveis



Prof. Bruno Góis Mateus (brunomateus@ufc.br)

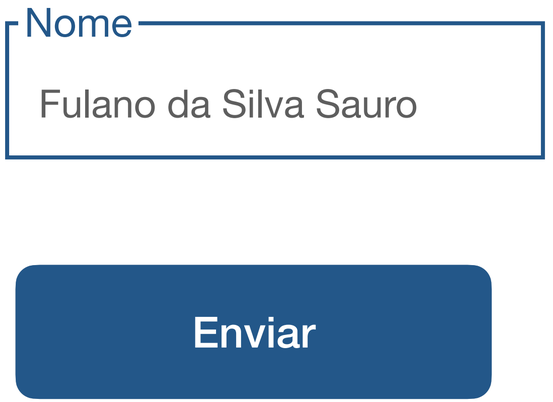
Conteúdo

- Introdução
- Recomposition
- State hoisting
- State hoisting: até onde?
- Salvando dados de UI
- Ciclo de vida de uma activity

Introdução

Introdução

- Em aplicativos reais, a interface muda conforme dados variam:
  - Texto de um campo
  - Itens de uma lista
  - Alternância de botões
- A UI reflete o estado atual dos dados
  - Quando o estado muda, a UI deve mudar junto



Introdução

O que é o “estado”?

- Estado é qualquer informação que pode mudar ao longo do tempo
- Exemplo:
  - Valor digitado em um TextField
  - Item selecionado em uma lista
  - Status de login de um usuário

Introdução

- No Compose, o estado controla como a tela é renderizada
- Estado = dados observáveis
  - Como o Compose é declarativo, a UI é uma função do estado:

    ⚙️ "UI = f(estado)"

  - Se o estado muda, a UI é recomposta *(recomposition)*

Problema: como garantir que o estado se mantenha correto entre recomposições e mudanças no app?

Recomposition

Recomposition

- Uma função composable pode ser reexecutada várias vezes
- Isso ocorre sempre que algum estado observado muda
- Variáveis locais de composables são recriadas a cada recomposição

Recomposition

Problema

```
@Composable
fun BotaoQueConta(modifier: Modifier = Modifier) {
    var nClicks = 0
    Button(onClick = { nClicks++ },
        modifier = modifier) {
        Text(
            text = if (nClicks == 0) "Clique em mim"
            else "Fui clicado $nClicks vezes"
        )
    }
}
```

Recomposition

remember: preservando estado durante recomposições

- remember
  - Usado para armazenar o objeto em memória
  - O valor inicial é armazenado durante o *compositon inicial*
  - O valor é mantido durante *recomposition*
- Ideal para guardar estados locais (ex: texto digitado, contador, etc.)

Recomposition

remember

- Utilizamos a função mutableStateOf para criar um estado mutável observável
- Há três maneiras de utilizá-lo:

```
val mutableState = remember { mutableStateOf(default) }
var value by remember { mutableStateOf(default) }
val (value, setValue) = remember { mutableStateOf(default) }
```

Função lambda que define o valor inicial

Recomposition

```
@Composable
fun BotaoQueConta(modifier: Modifier = Modifier) {
    var nClicks by remember { mutableIntStateOf(0) }
    Button(onClick = { nClicks++ },
        modifier = modifier) {
        Text(
            text = if (nClicks == 0) "Clique em mim"
            else "Fui clicado $nClicks vezes"
        )
    }
}
```

Recomposition

Stateful vs Stateless

- Quando se usa remember cria-se um estado interno, tornando-o *composable stateful*
  - Útil quando outros composable não precisam acessar ou gerenciar o estado
  - Tendem a ser menos reutilizáveis e mais difíceis de testar
- Um *composable stateless*, sem estado, é aquele que não mantém nenhum estado
  - Podemos tornar um *composable stateful, stateless* usando o içamento de estado (*state hoisting*)

Composable stateless são mais reutilizáveis e mais fáceis de testar.

Recomposition

```
@Test
fun whenDrinkButtonClicked_onIncrementIsCalledWithCorrectValue() {
    var incrementedValue = 0f
    val testIncrement = 250f

    composeTestRule.setContent {
        DrinkButton(
            onIncrement = { incrementedValue = it },
            increment = testIncrement,
            icon = R.drawable.half_cup
        )
    }

    composeTestRule.onNodeWithText("+$testIncrement").performClick()
    assertEquals(testIncrement, incrementedValue)
}
```

State hoisting

State hoisting

- É um padrão onde-se move o estado para a função Composable chamadora
  - Torna Composable originalmente dono do estado, stateless
- Em geral o state hoisting envolve a substituição de uma variável de estado por dois parâmetros:
  - value: T: contém valor atual a ser exibido
  - onValueChange: (T) -> Unit: um evento que solicita a alteração do value, onde T é o novo valor proposto

State hoisting

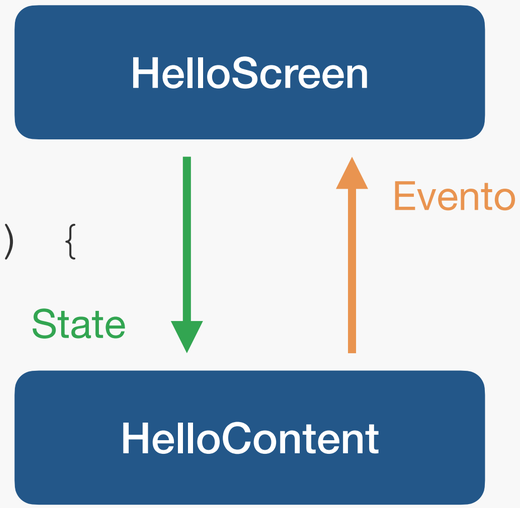
```
@Composable
fun HelloScreen() {
    HelloContent()
}

@Composable
fun HelloContent() {
    var name by rememberSaveable { mutableStateOf("") }
    Column(modifier = Modifier.padding(16.dp)) {
      Text(
        text = "Hello, $name",
        modifier = Modifier.padding(bottom = 8.dp),
        style = MaterialTheme.typography.bodyMedium
      )
      OutlinedTextField(value = name, onValueChange = { name = it }, label =
{ Text("Name") })
    }
}
```

State hoisting

```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }
    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange, label =
{ Text("Name") })
    }
}
```



Você não está limitado a onValueChange. Se eventos mais específicos forem apropriados para sua função Composable, você deve defini-los usando lambdas.

State hoisting

Propriedades importantes

- Single source of truth:
  - Ao mover o estado ao invés de duplicá-lo ficamos com uma única fonte
  - Torna o código menos propenso a erros
- Encapsulamento:
  - Apenas Composables stateful podem modificar o seu estado
- Compatilhável:
  - Se for necessário que o estado seja lido em outros composable, isso é possível com o state hoisting

State hoisting

Propriedades importantes

- Interceptável:
  - É possível ignorar ou modificar os events antes da alteração de estado
- Desacoplado:
  - O estado passado a composable stateless podem ser armazenados em qualquer lugar
  - Por exemplo, podemos mover para um viewModel

State hoisting

Propriedades importantes

- Interceptável:
  - É possível ignorar ou modificar os events antes da alteração de estado
- Desacoplado:
  - O estado passado a composable stateless podem ser armazenados em qualquer lugar
  - Por exemplo, podemos mover para um viewModel

State hoisting

Pontos chaves

- O estado deve ser elevado pelo menos ao nível do ancestral mais próximo comum a todos os Composable que precisam realizar a leitura
- O estado deve ser elevado pelo menos ao nível do Composable em que o estado pode ser alterado (escrita)
- Se dois estados mudarem em resposta aos mesmos eventos, eles devem ser elevados juntos

State hoisting: até onde?

State hoisting: até onde?

Estados da Interface

- Estado da tela
  - É o que você precisa mostrar na tela
  - Ex: Uma classe NewsUiState pode conter as notícias que devem ser mostradas na tela
  - Geralmente vem da camada de domínio e costuma ser gerenciado em um ViewModel
- Estado do elemento
  - Propriedades intrínsecas ao elementos da interface que influenciam a renderização
    - Ex: Um elemento pode ser mostrado ou ocultado
  - No Compose, esse estado é externo ao composable e pode ser controlado pelo chamador

State hoisting: até onde?

Lógica

- Lógica de negócio:
  - Relacionada aos requisitos da aplicação
  - Normalmente está na camada de domínio
  - Ex: favoritar uma notícia. Pode envolver salvar em um banco de dados (camada de domínio)
  - O detentor do estado geralmente delega essa lógica a essas camadas chamando os métodos que elas expõem
- Lógica de interface:
  - Relacionada a como mostrar o estado da interface na tela
  - Ex: rolar uma lista, mostrar/esconder um item, navegar entre telas.

State hoisting: até onde?

Lógica da interface

- Quando a lógica da interface precisa ler ou gravar o estado, é necessário definir o escopo do estado
- Para isso temos algumas opcões:
  - Manter como estado interno de uma função Composable
  - Elevar o estado no nível correto em uma função Composable
  - Criar uma simples classe detentora do estado

State hoisting: até onde?

Não é necessária elevação de estado

```
@Composable
fun ChatBubble(
    message: Message
) {
    var showDetails by rememberSaveable { mutableStateOf(false) }

    Text(
        text = AnnotatedString(message.content),
        modifier = Modifier.clickable {
            showDetails = !showDetails
        }
    )

    if (showDetails) {
        Text(message.timestamp)
    }
}
```

- Só é lido e modificado nesse combinável
- A lógica aplicada é muito simples
- Elevar o estado não traria muitos benefícios

State hoisting: até onde?

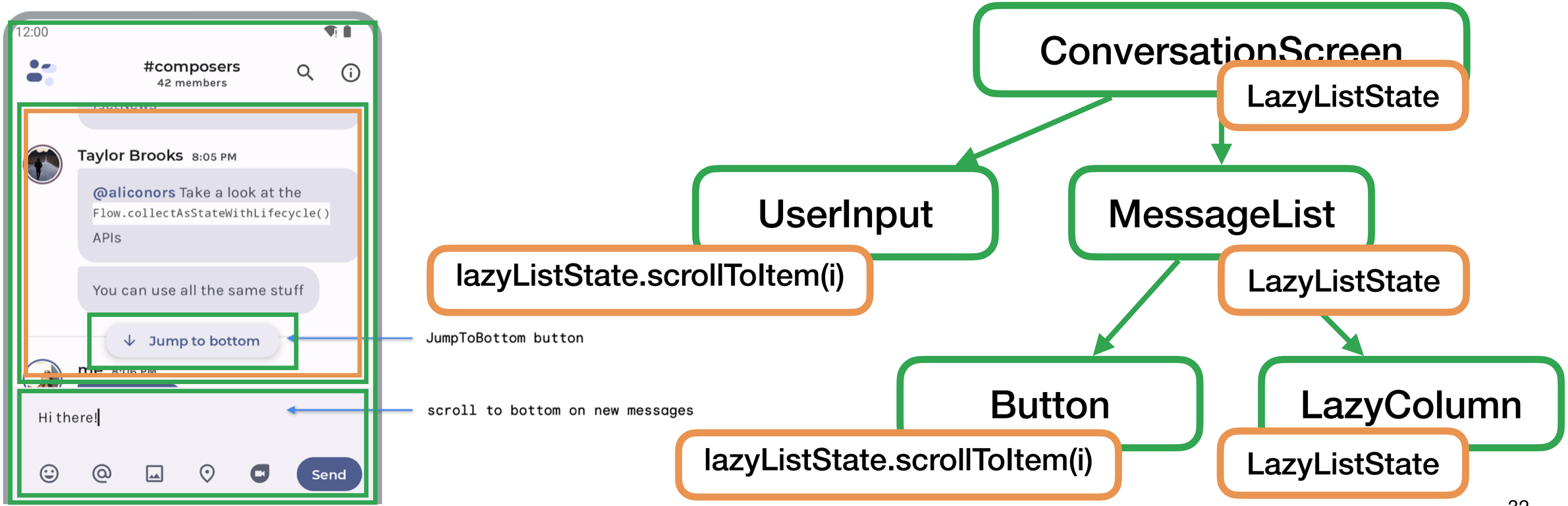
🧰 Quando manter o estado local

- Você pode manter o estado interno ao composable quando:
  - Ele só é lido/modificado dentro dele
  - A lógica é simples
  - Nenhum outro componente precisa acessar esse estado

State hoisting: até onde?

Elevando o estado

- Quando for preciso compartilhar o estado de um elemento com outros Composables ou aplicar alguma lógica em diferentes locais



State hoisting: até onde?

🚀 Quando elevar o estado (state hoisting)

- Eleve o estado quando:
  - Vários composables precisam ler/modificar o mesmo valor
  - Você deseja tornar o componente mais reutilizável e testável
  - A lógica da UI começa a ficar complexa

State hoisting: até onde?

Classe detentora de estado (*Plain state holder class*)

- Quando a lógica de interface é complexa e envolve um ou vários estado de um elemento da interface, é necessário delegar essa reponsabillidade
  - Torna a lógica do Composable mais testável e reduz a complexidade
- O Composable torna-se responsável somente por criar os elementos da interface
- A state holder class controla lógica da interface e o estado dos elemento da UI
- Essas classes simples são criadas e lembradas na composição.
  - Ex: rememberLazyListState()

State hoisting: até onde?

Classe detentora de estado (*Plain state holder class*)

```
@Stable
class LazyListState constructor(
    firstVisibleItemIndex: Int = 0,
    firstVisibleItemScrollOffset: Int = 0
) : ScrollableState {

    private val scrollPosition = LazyListScrollPosition(
        firstVisibleItemIndex, firstVisibleItemScrollOffset
    )

    suspend fun scrollToItem(/*...*/) { /*...*/ }

    override suspend fun scroll() { /*...*/ }

    suspend fun animateScrollToItem() { /*...*/ }
}
```

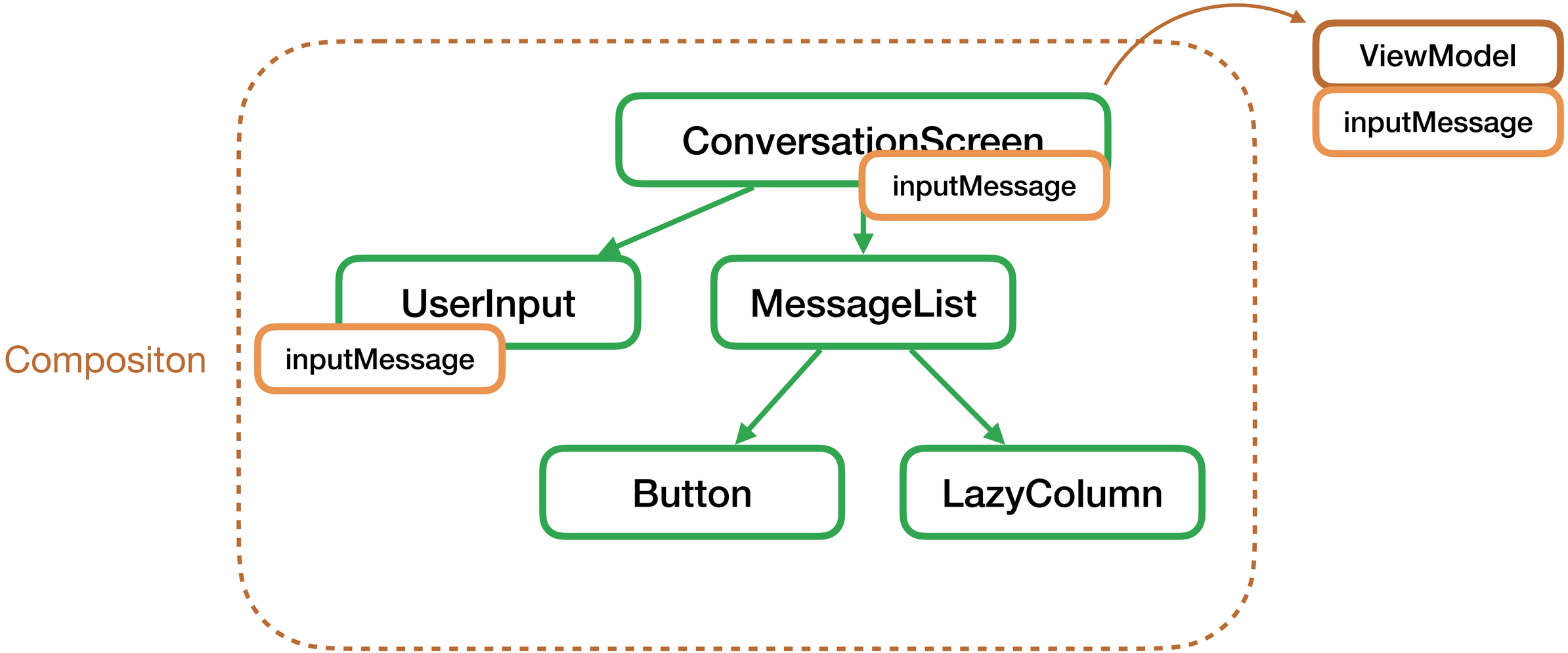
State hoisting: até onde?

Lógica de negócio

- Quando há lógica de negócio envolvida, o estado deve ser movido para o ViewModel
- Ele atua como fonte de verdade (single source of truth) em um ancestral comum mais baixo para o estado da tela.
- Quando você eleva o estado da interface em ViewModel, ele é movido para fora da composição

State hoisting: até onde?

Elevando o estado



State hoisting: até onde?

Boas práticas

✅ Mantenha o estado próximo do local de uso

✅ Eleve apenas quando precisar compartilhar ou aplicar lógica externa

✅ Use ViewModels para lógica de negócio e persistência de estado

✅ Prefira state holders para lógica de UI complexa

✅ Evite hoisting desnecessário — aumenta o acoplamento

Salvando dados de UI

Salvando dados de UI

Por que se preocupar com o estado da UI?

- Qualquer app Android pode perder o estado da UI devido a:

  🔁 Mudanças de configuração (ex: rotação de tela)

  💀 Morte do processo (quando o sistema libera memória)

- Consequências:
  - Campos de texto limpos, listas que “resetam”, navegação perdida

Preservar o estado é essencial para garantir um boa experiência do usuário. A escolha de qual estado persistir vária de acordo com o aplicativo. É recomendado preservar pelo menos o estado relacionado à entrada do usuário e à navegação.

Salvando dados de UI

Que tipo de estado deve ser salvo?

- Salve apenas o estado relevante à experiência do usuário, como:
  - 🧍 Entrada de texto do usuário
  - 📜 Posição de rolagem de uma lista
  - 🗺️ Item selecionado ou tela atual
  - ⚙️ Preferências em andamento
- Não é necessário salvar estados puramente visuais (ex: animações em curso)

Salvando dados de UI

Onde o estado está “hoisted”?

Onde o estado é mantido?	API recomendada	Sobrevive à recriação da Activity?	Sobrevive à morte do processo?
Composables / lógica de UI	`rememberSaveable`	✅ Sim	✅ Sim
ViewModel (lógica de negócio)	`SavedStateHandle`	✅ Sim	✅ Sim (parcial)

Salvando dados de UI

Lógica de UI : remember vs remember saveable

- remember
  - Mantém o estado durante a recomposição.

  ❌ Perde o valor se a Activity for destruída (ex: rotação)
- rememberSaveable
  - Mantém o estado durante recomposição, rotação e morte do processo.

  ✅ Armazena em um Bundle interno (como onSaveInstanceState( ))

Salvando dados de UI

Lógica de UI : remember vs remember saveable

```
@Composable
fun ChatBubble(message: Message) {
    var showDetails by rememberSaveable { mutableStateOf(false) }

    ClickableText(
        text = AnnotatedString(message.content),
        onClick = { showDetails = !showDetails }
    )

    if (showDetails) Text(message.timestamp)
}
```

Salvando dados de UI

Bundle - salvando tipos não primitivos

- É possível armazenar tipos primitivos no Bundle automaticamente
- Existem mecanismos de armazenamento diferentes para tipos não primitivos:
  - @Parcelize
  - APIs do Compose como `listSaver` e `mapSaver`
  - Estender a classe `Saver` do runtime do Compose

Salvando dados de UI

Bundle - salvando tipos não primitivos

```
@Composable
fun rememberLazyListState(): LazyListState {
    return rememberSaveable(saver = LazyListState.Saver) {
        LazyListState()
    }
}
```

Salvando dados de UI

Bundle - Boas práticas

- `rememberSaveable` usa um Bundle que é compartilhado por outras APIs que também gravam nele, como as chamadas `onSaveInstanceState()` em um activity
  - O tamanho do Bundle é limitado
  - Armazenar objetos grandes pode levar a exceções TransactionTooLarge em tempo de execução
  - Particularmente problemático em aplicativos com uma única Activity

Salvando dados de UI

Bundle - Boas práticas

- Boas práticas
  - Evitar armazenar objetos grandes ou listas complexas
  - Salvar apenas IDs, chaves ou valores simples
  - Recarregar os dados complexos do data layer (Room, API, etc.)

Salvando dados de UI

Lógica de negócio

- Um dos principais benefícios de usar um ViewModel em seu aplicativo Android é que ele lida com mudanças de configuração automaticamente
- No entanto, uma instância do ViewModel não sobrevive ao encerramento do processo iniciado pelo sistema
- Para que o estado da interface do usuário sobreviva a isso, use a API SavedStateHandle
    - Ela também utiliza o mecanismo de Bundle

Ciclo de vida de uma activity

Ciclo de vida de uma activity

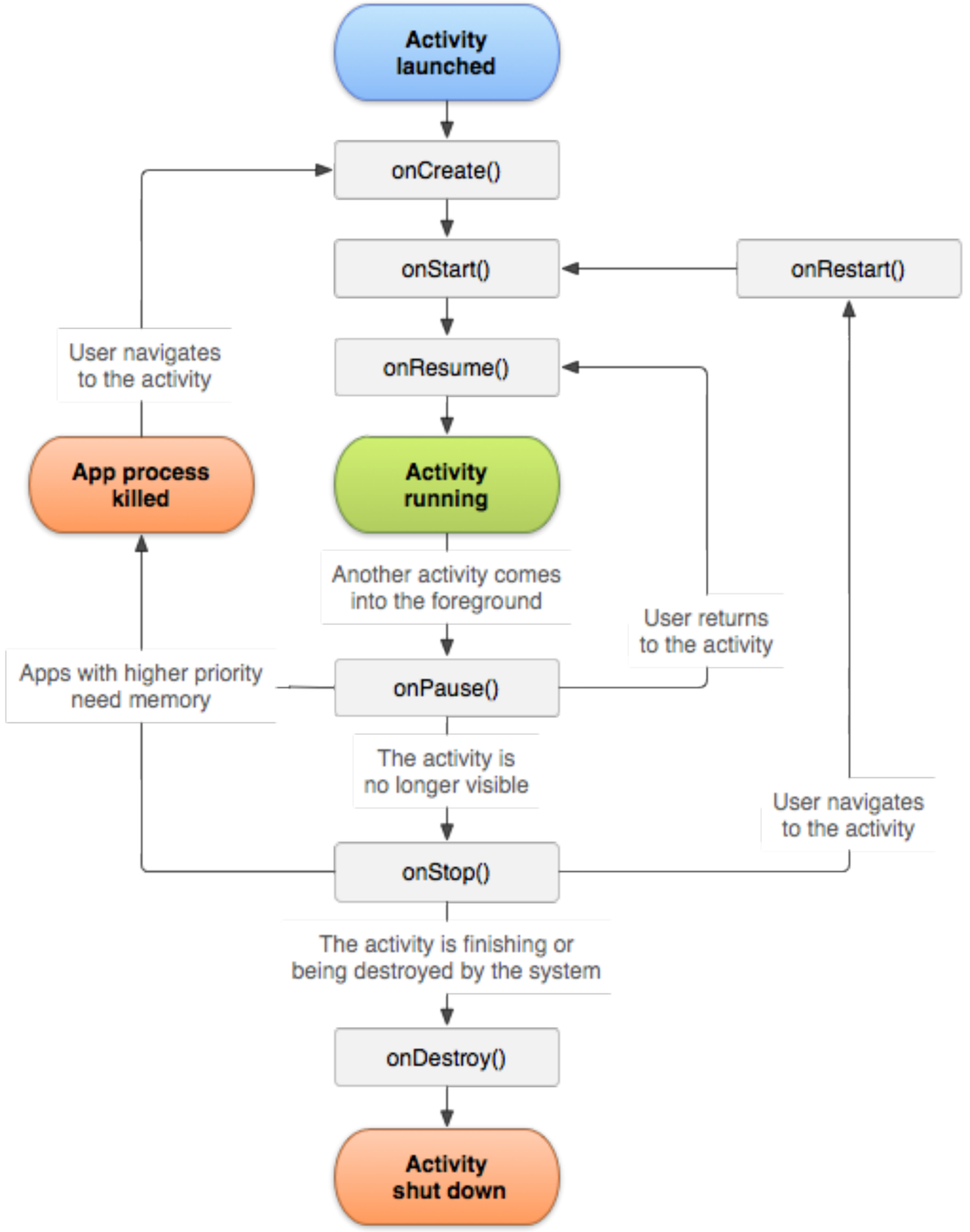
🧩 O que é uma Activity?

- Uma Activity representa uma tela do app
- Cada vez que o usuário abre, fecha ou alterna entre apps, o Android precisa gerenciar recursos e memória
- O sistema controla isso através do ciclo de vida das Activities

Ciclo de vida de uma activity

🔁 Por que o ciclo de vida importa?

- Saber o momento certo para agir evita:
  - Travamentos e consumo excessivo de memória
  - Perda de informações do usuário (ex: texto digitado, rolagem)
  - Bugs quando o Android recria a Activity automaticamente


Activity launched
onCreate()
onStart()
onRestart()
onResume()
User navigates to the activity
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Ciclo de vida de uma activity

⚙️ Os principais métodos do ciclo de vida

Callback	Estado	O que Fazer (Ação Principal)	Quando Ocorre
onCreate()	Created	Inflar o layout, inicializar View Model, vincular dados a listas. Recebe o savedInstanceState (Bundle) para restauração.	A *Activity* é criada pela primeira vez.
onStart()	Started	A *Activity* se torna **visível** ao usuário. Inicializar código que mantém a UI.	Após onCreate() ou após onRestart().
onResume()	Resumed	A Activity está em primeiro plano e interativa. Inicializar componentes que precisam de foco exclusivo (ex: câmera, listeners).	A Activity retorna ao topo.

Ciclo de vida de uma activity

⚙️ Os principais métodos do ciclo de vida

Callback	Estado	O que Fazer (Ação Principal)	Quando Ocorre
onPause()	Paused	Primeira indicação de que o usuário está saindo, mas ainda pode estar visível. Parar operações que não podem continuar (ex: listeners de sensores). NÃO SALVAR DADOS PESADOS AQUI.	O foco é tirado da Activity.
onStop()	Stopped	Liberar ou ajustar recursos intensivos (ex: salvar dados pesados no banco de dados, parar atualizações de localização precisas).	A Activity fica completamente invisível.
onDestroy()	Destroyed	A Activity está prestes a ser destruída. Liberar todos os recursos remanescentes.	A Activity está finalizando ou o sistema está destruindo

Ciclo de vida de uma activity

🧠 Salvando o estado da interface

- A perda de dados é o pior pesadelo do usuário
- Existem dois motivos principais para uma Activity ser destruída:
- Destruição Natural:
  - O usuário pressiona "Voltar" ou o código chama finish()
  - Neste caso, o estado é perdido, e isso é esperado pelo usuário
- Destruição pelo Sistema:
  - Mudança de configuração (rotação de tela) ou falta de memória (memory pressure)
  - Neste caso, o estado precisa ser salvo e restaurado

Ciclo de vida de uma activity

📜 Linha do tempo típica

```
//Ao abrir o aplicativo
onCreate() → onStart() → onResume()

//Ao pressionar Home
onPause() → onStop()

//Ao voltar para o aplicativo
onRestart() → onStart() → onResume()
// Ao fechar o aplicativo
onPause() → onStop() → onDestroy()
```

Ciclo de vida de uma activity

🔄 Mudança de configuração

```
//Ao abrir o aplicativo
onCreate() → onStart() → onResume()

//Ao ocorrer uma mudança de configuração
onPause() → onStop() → onDestroy() → onCreate() → onStart() →
onResume()
```

Ciclo de vida de uma activity

🧠 Salvando o estado da interface

⚡ Quando usar o Bundle

- Ideal para dados simples e de curta duração
  - Exemplo: texto digitado, posição de rolagem, opção selecionada
- Não use para listas grandes ou objetos complexos
  - Prefira ViewModel ou persistência local
  - O ViewModel é preservado quando a Activity é destruída
  - Recomendado para dados que não são apenas de UI

Ciclo de vida de uma activity

🧠 Salvando o estado da interface

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putString("mensagem", "Olá!")
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val mensagem = savedInstanceState?.getString("mensagem")
}
```

- Você pode optar por implementar o método onRestoreInstanceState( ), que é chamado pelo sistema após o método onStart( )

Ciclo de vida de uma activity

Gerenciamento de recursos

- O maior desafio é saber onde iniciar e onde parar os recursos:
  - Recursos exclusivos/interativos:
    - Devem ser iniciados em onResume() e liberados em onPause() (Ex: Câmera, GPS)
    - Garante que o recurso só está ativo quando o usuário está interagindo com a Activity
    - Economizando bateria e liberando o recurso para outros apps rapidamente
  - Recursos de UI (visíveis):
    - Podem ser iniciados em onStart() e liberados em onStop() para melhor suporte ao modo multi-janela, onde a Activity pode estar visível, mas pausada

Boa prática, não colocar lógica complexa diretamente nos callbacks da Activity. Use Componentes Cientes do Ciclo de Vida (Lifecycle-Aware Components) e o padrão ViewModel para manter a lógica de negócio separada e mais limpa

Bibliografia

- State and Jetpack Compose
- Where to hoist state
- Save UI state in Compose
- Jetpack Compose — When should I use derivedStateOf?
- The Hidden Dangers of Jetpack Compose State (And How to Fix Them With Real Examples)
- Activity Lifecycle in Android

Por hoje é só