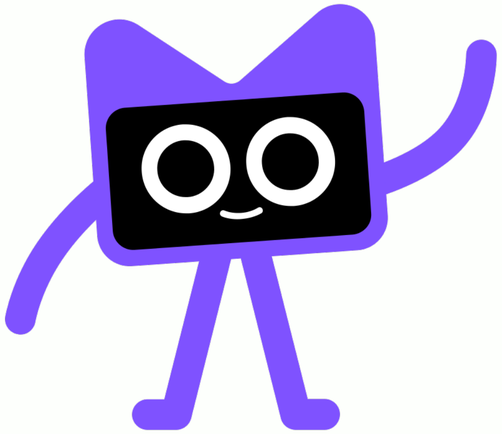
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Navegação

QXD0276 - Desenvolvimento de Software para Dispositivos Móveis



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

Conteúdo

- Introdução
- Navigation 3
- Funcionamento básico
- Gerenciando o estado
- Animações
- Adaptive Layouts
- Layouts Canônicos
- Cenas

Introdução

Introdução

A Evolução da Navegação em Aplicativos Android (2008–2011)

- Desenvolvimento Android começou com Activities como unidade principal
  - Cada tela → uma Activity independente
- Navegação simples
  - Acoplada e com alto custo de ciclo de vida
- Com o crescimento dos apps, surgiu a necessidade de melhor reutilização e melhor gestão de UI

Introdução

A era dos Fragments (2011–2017)

- Por que surgiram?
  - Tablets pediam interfaces mais complexas e flexíveis
  - Uma única Activity passou a hospedar múltiplos Fragments
- Impacto
  - Navegação passou a depender de FragmentManager e back stack manual
  - Reuso

Introdução

A era dos Fragments (2011–2017)

- Mais poder, mas também mais complexidade:
  - Transações manuais
  - Bugs de estado (especialmente em rotação)
  - Fluxos difíceis de manter em apps grandes

Introdução

Navigation Component (2018–2022)

- Mudanças importantes
  - Introdução do NavController, NavHostFragment e grafos de navegação
  - Padronizou:
    - Back stack
    - Deep links
    - Argumentos tipados (Safe Args)
    - Transições entre fragments

Introdução

Navigation Component (2018–2022)

- Benefícios
  - Arquitetura clara para apps baseados em Fragments
  - Redução de boilerplate.
  - Navegação mais previsível e testável
- Limitações:
  - Fortemente acoplado a Fragments
  - Complexo para múltiplos layouts (dual-pane)

Introdução

Navegação no Jetpack Compose (2020 - 2025)

- Nova abordagem
  - UI declarativa → navegação também passou a ser declarativa
  - Tudo feito em código (sem XML)
- Características
  - NavHost recebe destinos composables
  - Transições entre telas como funções
  - Back stack é controlado por NavHostController

Introdução

Navegação no Jetpack Compose (2020 - 2025)

- Limitações que motivaram evolução:
  - Acoplamento excessivo com o NavHost
  - Dificuldade em modelar grafos dinâmicos
  - Integração com Compose limitada para fluxos avançados
    - Controle limitado sobre estado por destino
  - Dificuldade em navegação adaptativa

Introdução

Navigation 3 (2025+)

- Por que surgiu?
  - Para resolver limitações das soluções anteriores e atender:
    - Telas grandes
    - Foldables
    - Múltiplos fluxos simultâneos
    - Navegação baseada em estado

Introdução

Navigation 3 (2025+)

- O que muda?
  - Navegação baseada em screens, não mais em destinos de NavHost
  - Pensada para Compose desde o início
  - Simples, recomponível e previsível

Introdução

Navigation 3 (2025+)

- Benefícios principais
  - Compatível com Compose
  - Navegação realmente state-driven
  - Menos boilerplate, sem dependência de um grafo centralizado
  - Fluxos complexos ficam mais claros
  - Escalável e adaptável
  - Animações avançadas

Navigation 3

Navigation 3

Introdução

- É uma nova biblioteca de navegação projetada para funcionar com o Compose
  - O desenvolvedor tem total controle sobre a backstack
  - Navegar é tão simples quanto adicionar e remover itens de uma lista
- Fornece um sistema de navegação flexível ao prover:
  - Atualização automática da interface por meio de mudanças na backstack
  - Sistema de layout adaptável permite mostrar vários destinos ao mesmo tempo
  - Um mecanismo de comunicação entre layouts por meio de metadados

Navigation 3

Navigation 3 vs Jetpack Navigation

- Oferece uma integração mais simples com o Compose
- Oferece controle total da pilha de retorno
- Permite criar layouts que podem ler mais de um destino da pilha de retorno ao mesmo tempo, permitindo que eles se adaptem a mudanças no tamanho da janela e outras entradas

Navigation 3

Dependências

Nome	O que ele faz	Artefato
Biblioteca de ambiente de execução da Navigation 3	API Core Navigation 3. Inclui NavEntry, EntryProvider e a DSL associada.	androidx.navigation3:navigation3-runtime
Biblioteca de interface da Navigation 3	Fornece classes para mostrar conteúdo, incluindo NavDisplay e Scene.	androidx.navigation3:navigation3-ui
Ciclo de vida do ViewModel para navegação 3	Permite que os ViewModels sejam delimitados a entradas na backstack.	androidx.lifecycle:lifecycle-viewmodel-navigation3

Navigation 3

Dependências

Nome	O que ele faz	Artefato
Layouts adaptáveis do Material 3 para Navigation 3	Fornece layouts adaptáveis (SceneStrategies, Scenes e definições de metadados) para uso com NavDisplay.	androidx.compose.material3.adaptive:adaptive-navigation3
KotlinX Serialization (link em inglês)	Permite que as chaves de navegação sejam serializadas.	Plug-in: org.jetbrains.kotlin.plugin.serialization Biblioteca:

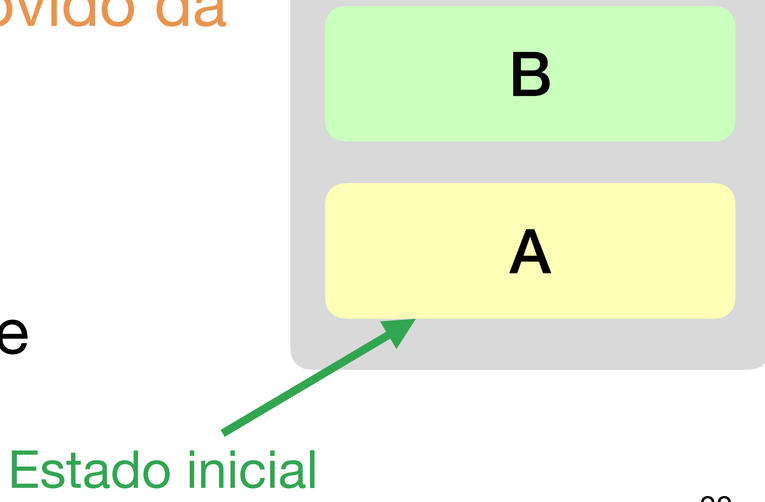
Configura a documentação oficial para configurar sua app

Funcionamento básico

Funcionamento básico

Modelando o estado de navegação

- Uma maneira prática de modelar o estado de navegação é utilizando uma pilha
- Quando o usuário navega para um novo conteúdo, ele é adicionado ao topo da pilha
- Quando ele retorna a partir desse conteúdo, ele é removido da pilha e o conteúdo anterior é exibido
- Chamamos essa pilha de backstack
  - Já ela contém o conteúdo para o qual o usuário pode retornar



Funcionamento básico

≡ Criando a back stack

- No Navigation 3, a pilha de retorno não contém conteúdo propriamente dito
  - Em vez disso, ela contém referências a conteúdo, conhecidas como chaves (Keys)
  - As chaves podem ser de qualquer tipo, mas geralmente são classes de dados simples e serializáveis.

Funcionamento básico

≡ Criando a back stack

- Usar referências em vez de conteúdo oferece os seguintes benefícios:
  - A navegação é simples, bastando adicionar chaves à pilha de retorno
  - Com chaves serializáveis, a pilha de retorno pode ser salva em armazenamento persistente, permitindo que ela sobreviva a alterações de configuração e ao encerramento do processo
  - Importante já que os usuários esperam sair do seu aplicativo, voltar a ele mais tarde e continuar de onde pararam, com o mesmo conteúdo sendo exibido

Um conceito fundamental na API Navigation 3 é que você é o proprietário da pilha de retorno.

Funcionamento básico

≡ Criando a back stack

```
data object ProductList
data class ProductDetail(val id: String)
```

Define as chaves que identificam os conteúdos

```
@Composable
fun MyApp() {

    val backStack = remember { mutableStateListOf<Any>(ProductList) }

    backStack.add(ProductDetail(id = "ABC"))

    backStack.removeLastOrNull()
}
```

Cria a pilha e define o conteúdo inicial

Volta para o conteúdo anterior

Adiciona uma nova chave o que leva o usuário a uma nova “tela"

Funcionamento básico

🔑 De chaves para conteúdo

- O conteúdo é representado no Navigation 3 usando NavEntry
  - Uma classe que contém uma função composable
  - Ela representa um destino
- Um NavEntry também pode conter metadados
  - São um mapa de chaves String para valores Any, fornecendo armazenamento de dados versátil
  - Podem ser lidos por objetos contêineres, como NavDisplay, para ajudá-los a decidir como exibir o conteúdo do NavEntry
  - Ex: Podem ser usados para substituir as animações padrão de um NavEntry específico

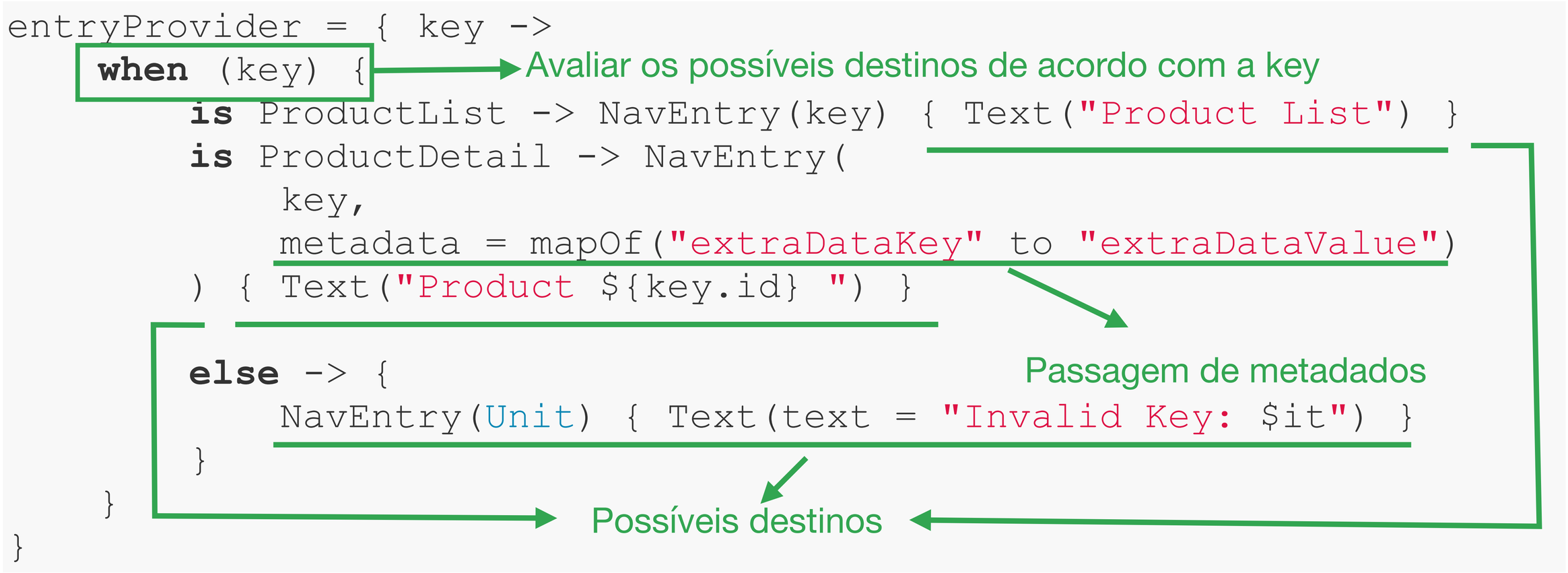
Funcionamento básico

🔑 De chaves para conteúdo

- Para fazer uso do NavEntry é preciso combiná-lo com entryProvider
- Existem duas maneira de criar um Entry Provider
  - Usando a função lambda diretamente
  - Usando a DSL entryProvider

Funcionamento básico

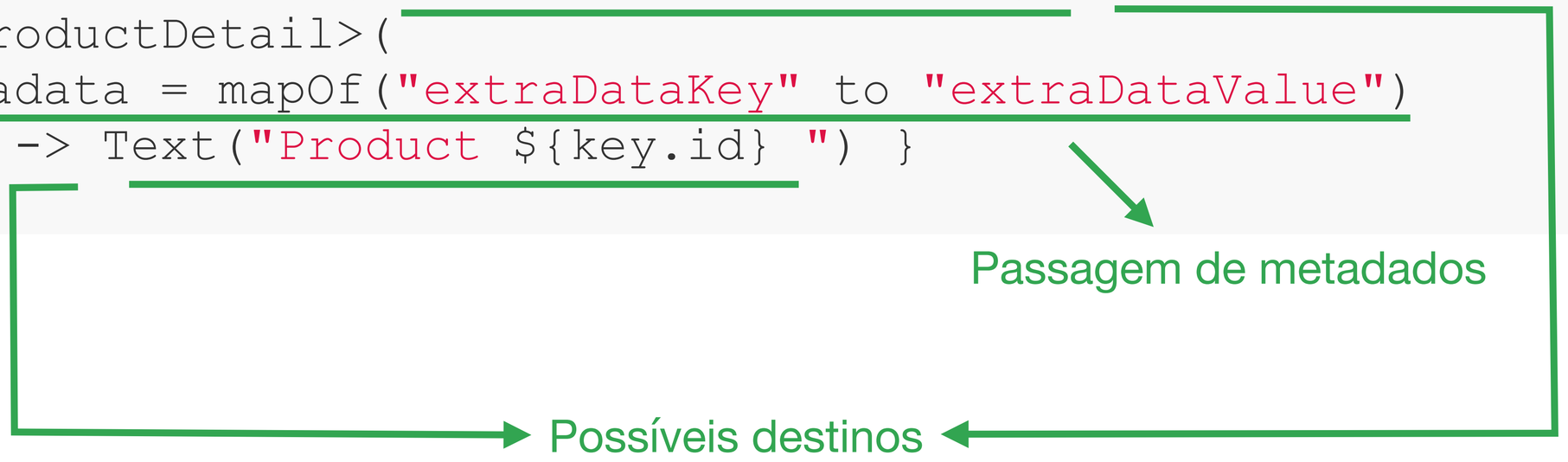
Criando um Entry Provider diretamente



Funcionamento básico

Usando o entryProvider DSL

```
entryProvider = entryProvider {
    entry<ProductList> { Text("Product List") }
    entry<ProductDetail>(
        metadata = mapOf("extraDataKey" to "extraDataValue")
    ) { key -> Text("Product ${key.id} ") }
}
```



Funcionamento básico

≡ Mostrar a back stack

- A back stack representa o estado de navegação do seu aplicativo
- Sempre que a back stack muda, a interface do aplicativo deve refletir o seu novo estado
- No Navigation 3, um NavDisplay observa a back stack e atualiza a interface
- Construa-o com os seguintes parâmetros:
  - Back stack - deve ser do tipo SnapshotStateList<T>, onde T é o tipo das chaves da sua pilha de retorno
  - Um entryProvider para converter as chaves em objetos NavEntry
  - Opcionalmente, forneça uma função lambda para o parâmetro onBack

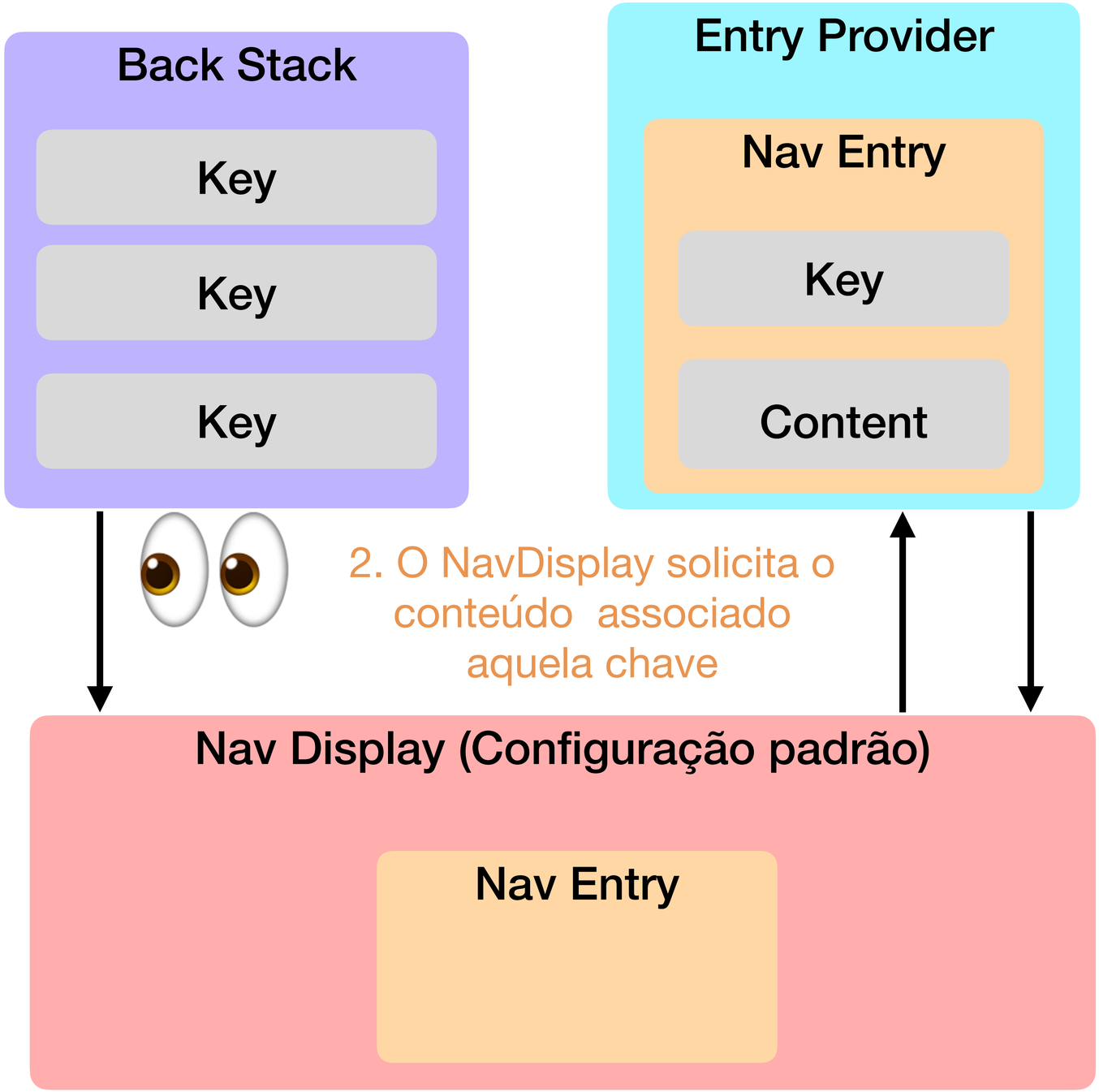
Funcionamento básico

≡ Mostrar a back stack

```
@Composable
fun NavExample() {
    val backStack = remember { mutableStateListOf<Any>(Home) }
    NavDisplay(
        backStack = backStack,
        onBack = { backStack.removeLastOrNull() },
        entryProvider = { key ->
          when (key) {
            is Home -> NavEntry(key) {
               ContentGreen("Welcome to Nav3") {
                  Button(onClick = { backStack.add(Product("123"))}) {
                    Text("Click to navigate")
            }}}
            is Product -> NavEntry(key) { ContentBlue("Product ${key.id} ")}
            else -> NavEntry(Unit) { Text("Unknown route") }}
        }
    )
}
```

5. Eventos de navegação: push e pop

1. Estado inicial



2. O NavDisplay solicita o conteúdo associado aquela chave

3. O entryProvider possui o mapeamento entre chave e conteúdo. Ele envia a NavEntry adequada.

4. O conteúdo é apresentado na tela

Gerenciando o estado

Garantir que o estado de navegação persista durante o ciclo de vida do app, incluindo alterações de configuração e encerramento do processo, é crucial para uma boa experiência do usuário

Gerenciando o estado

rememberNavBackStack

- O Navigation 3 oferece um método conveniente que permite salvar a back stack: rememberNavBackStack
  - Foi criada para que a back stack resista a mudanças de configuração e morte do processo
- Para que a função funcione corretamente é preciso que cada chave atenda as seguinte requisitos:
  - Implemente a interface NavKey
  - Use a anotação @Serializable

Gerenciando o estado

≡ rememberNavBackStack

```
@Serializable
data object Home : NavKey

@Composable
fun NavBackStack() {
    val backStack = rememberNavBackStack(Home)
}
```

Gerenciando o estado

Armazenando no viewModel

- Outra abordagem para gerenciar a back stack é armazená-la em um ViewModel
- Para garantir a persistência mesmo após o encerramento do processo, ao usar um ViewModel ou qualquer outro armazenamento personalizado, você precisa:
  - Garantir que suas chaves sejam serializáveis
  - Lidar com a serialização e desserialização manualmente
    - É preciso salvar manualmente a representação serializada de cada chave e desserializá-la do armazenamento persistente
    - Por exemplo, SharedPreferences, um banco de dados ou um arquivo)

Gerenciando o estado

Definindo o escopo de um View Model

- Por padrão, os ViewModels têm escopo limitado ao ViewModelStoreOwner mais próximo:
  - Geralmente uma Activity ou Fragment.
- No entanto, você pode querer limitar o escopo de um ViewModel a um NavEntry específico
  - Em outras palavra, uma tela ou destino específico na pilha de retorno
  - Isso garante que o estado do ViewModel seja mantido apenas enquanto esse NavEntry específico estiver na backStack

ViewModel

Definindo o escopo de um View Model

- A biblioteca androidx.lifecycle:lifecycle-viewmodel-navigation3 fornece um NavEntryDecorator que facilita isso
  - Fornece um ViewModelStoreOwner para cada NavEntry
- Quando um ViewModel é criado dentro do conteúdo de um NavEntry, ele é automaticamente limitado ao escopo da chave desse NavEntry
- Além de adicionar a dependência ao projeto é necessário passar o entryDecorators do NavDisplay e:
  - Adicionar o método rememberSaveableStateHolderNavEntryDecorator()
  - Adicionar o método rememberViewModelStoreNavEntryDecorator()

View Model

Definindo o escopo de um View Model

```
NavDisplay(
    entryDecorators = listOf(
        rememberSaveableStateHolderNavEntryDecorator(),
        rememberViewModelStoreNavEntryDecorator()
    ),
    backStack = backStack,
    entryProvider = entryProvider { },
)
```

Animações

Animações

- O NavDisplay oferece recursos de animação para criar transições visuais
- É possível personalizar essas animações:
  - Globalmente para o NavDisplay
  - Individualmente para cada NavEntry usando metadados

Animações

Alterando os efeitos padrões

- O NavDisplay usa ContentTransforms para definir como o conteúdo é animado
- É possível substituir as animações padrão por meio parâmetros de transição ao NavDisplay
- transitionSpec:
  - Define a animação a ser aplicada quando o conteúdo é adicionado à pilha (ao navegar para frente).
- popTransitionSpec:
  - Define a animação ser aplicada quando o conteúdo é removido da pilha (ao navegar para trás)
- predictivePopTransitionSpec:
  - Define a animação a ser aplicada quando o conteúdo é removido da pilha usando um gesto preditivo de voltar

Animações

Alterando os efeitos padrões

```
@Composable
fun NavExample() {
    NavDisplay(
        transitionSpec = {
            // Slide in from right when navigating forward
            slideInHorizontally(initialOffsetX = { it }) togetherWith
                slideOutHorizontally(targetOffsetX = { -it })
        },
        popTransitionSpec = {
            // Slide in from left when navigating back
            slideInHorizontally(initialOffsetX = { -it }) togetherWith
                slideOutHorizontally(targetOffsetX = { it })
            },
        predictivePopTransitionSpec = {
            // Slide in from left when navigating back
            slideInHorizontally(initialOffsetX = { -it }) togetherWith
                slideOutHorizontally(targetOffsetX = { it })
     })
}
```

Animações

Definindo a animação individualmente

- É possível definir animações personalizadas para NavEntrys específicos usando seus metadados
- O NavDisplay reconhece chaves de metadados especiais para aplicar transições por entrada:
  - NavDisplay.transitionSpec: Função auxiliar para definir a animação de navegação para frente
  - NavDisplay.popTransitionSpec: Função auxiliar para definir a animação de navegação para trás para um NavEntry específico
  - NavDisplay.predictivePopTransitionSpec: Use esta função auxiliar para definir a animação para gestos preditivos de voltar para um NavEntry específico
- Essas transições de metadados por entrada substituem as transições globais do NavDisplay com o mesmo nome

Animações

Definindo a animação individualmente

```
entry<ScreenC>(
   metadata = NavDisplay.transitionSpec {
       slideInVertically(
          initialOffsetY = { it },
          animationSpec = tween(1000)
       ) togetherWith ExitTransition.KeepUntilTransitionsFinished
  } + NavDisplay.popTransitionSpec {
        EnterTransition.None togetherWith
            slideOutVertically(
               targetOffsetY = { it },
                animationSpec = tween(1000)
            )
   } + NavDisplay.predictivePopTransitionSpec {
        EnterTransition.None togetherWith
            slideOutVertically(
                targetOffsetY = { it },
                animationSpec = tween(1000)
             )
  }
```

Adaptive Layouts

Adaptive Layouts

Por que Adaptive Layouts Importam?

📱 A fragmentação no desenvolvimento para Android

- Smartphones compactos
- Tablets de diversos tamanhos
- Foldables (com display dobrável e múltiplos estados)
- ChromeOS e Desktop windowing
- Multi-window, split-screen e freeform windowing

Permitir que o mesmo app se comporte perfeitamente em qualquer configuração de tela.

Adaptive Layouts

💡 Conceitos-chave

- Responsive layouts
  - Pequenos ajustes na UI para ocupar melhor o espaço.
- Adaptive layouts
  - Mudanças estruturais, como trocar single-pane ↔ two-pane.

Adaptive Layouts

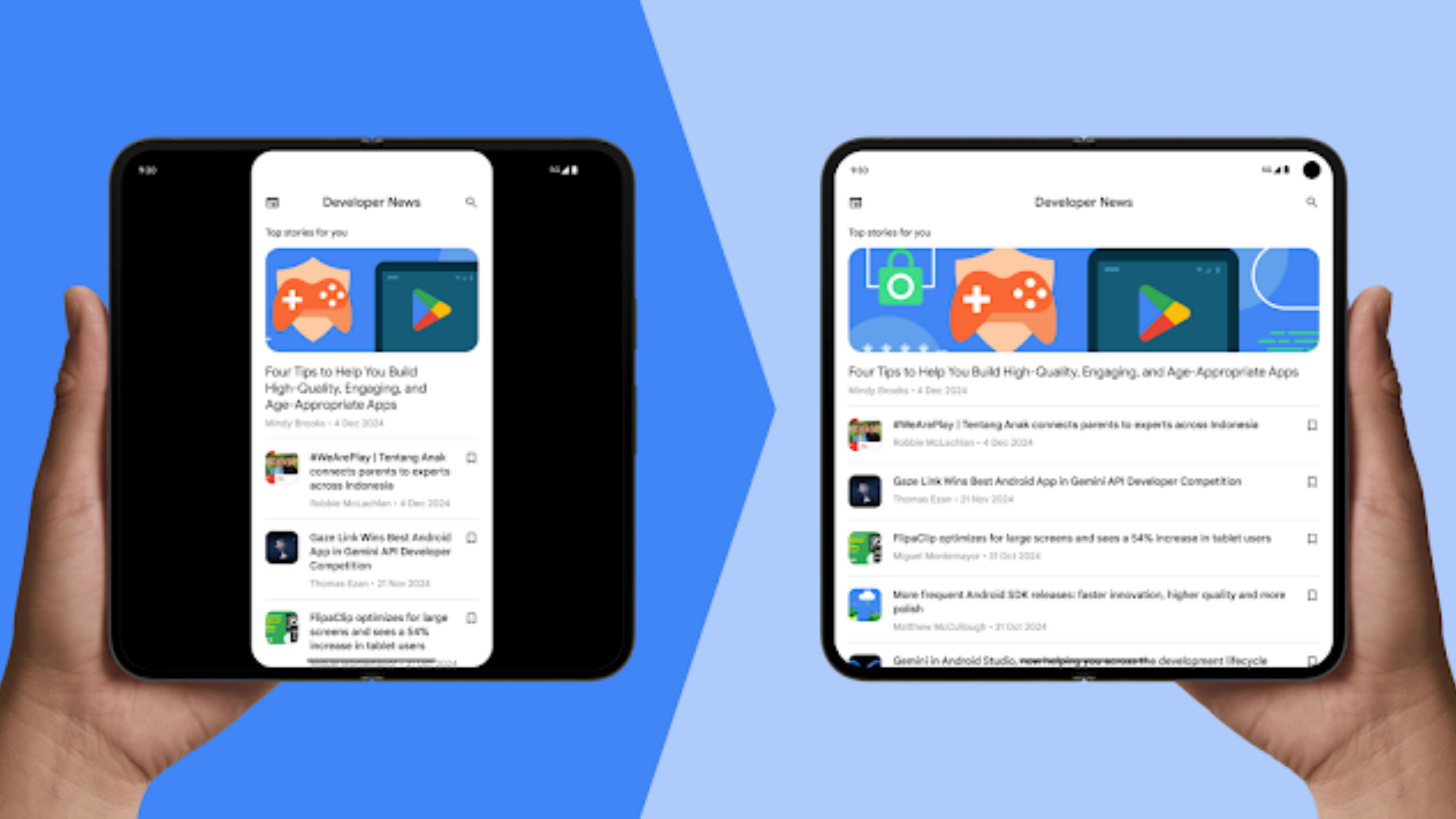
Android 16 (API 36) ignora diversas restrições

- android:screenOrientation="portrait" (ou qualquer valor fixo)
- android:resizeableActivity="false"
- android:minAspectRatio, android:maxAspectRatio
- setRequestedOrientation()
- getRequestedOrientation() para valores fixos

Adaptive Layouts

Android 16 (API 36) ignora diversas restrições

- Consequências
  - resizeableActivity torna-se sempre true
  - apps podem entrar em multi-window sempre
  - letterboxing some → apps passam a rodar full-screen
  - a UI pode mudar de tamanho a qualquer momento


Developer News
Top stories for you
Four Tips to Help You Build High-Quality, Engaging, and Age-Appropriate Apps
#WeAreRiley | Tentang Anak connects parents to experts across Indonesia
Gaze Link Wins Best Android App in Gemini API Developer Competition
FlipaClip optimizes for large screens and sees a 54% increase in tablet users
Developer News
Top stories for you
Four Tips to Help You Build High-Quality, Engaging, and Age-Appropriate Apps
#WeAreRiley | Tentang Anak connects parents to experts across Indonesia
Gaze Link Wins Best Android App in Gemini API Developer Competition
FlipaClip optimizes for large screens and sees a 54% increase in tablet users
More frequent Android SDK releases: faster innovation, higher quality and more polish

Adaptive Layouts

Problemas comuns ao ignorar essa mudança

- UI esticada em orientações inesperadas
- Componentes sobrepostos por falta de responsividade
- Partes do conteúdo fora da tela
- Câmera deformada por assumir uma proporção fixa
- Perda de estado ao redimensionar/rotacionar (activity recreation)

Adaptive Layouts

Suporte a diferentes tamanhos de tlela

- Projete os layouts para serem e adaptáveis
  - Suporte a dispositivos diferentes
  - Suporte a janelas de aplicativos no modo multijanela
- Layouts adaptativos mudam de acordo com o espaço disponível na tela
  - As mudanças variam desde pequenos ajustes de layout até a substituição completa de um layout por outro

Adaptive Layouts

Tipos de composables

- App-level composable
  - O composable raiz da aplicação
  - Ocupa todo o espaço da janela
  - É nele que você decide o layout geral
    - Pane único, dois panes, navigation rail ou drawer etc
  - Ele deve ler o WindowSizeClass e propagar decisões.

Adaptive Layouts

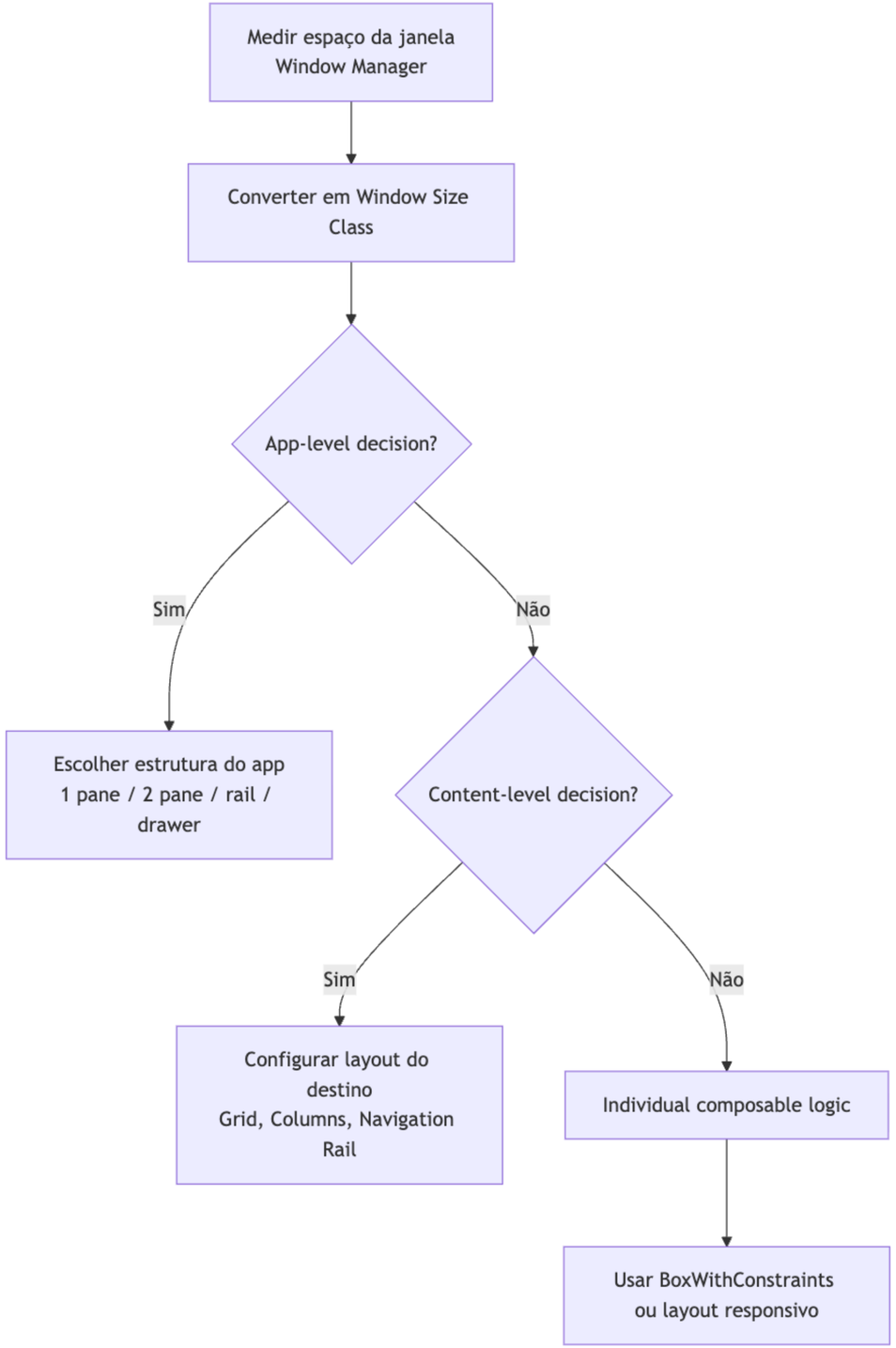
Tipos de composables

- Content-level composables
  - Destinos principais da navegação
  - Eles também devem adaptar seu layout geral (como lista/detalhe).
  - Mas não devem acessar métricas da janela diretamente
  - Devem receber a decisão do app-level

Adaptive Layouts

Tipos de composables

- Individual composables
  - Componentes reutilizáveis (cards, linhas, itens visuais).
  - Podem usar:
    - BoxWithConstraints,
    - Modifier.fillMax…,
    - ou layouts adaptativos locais


Medir espaço da janela
Window Manager
Converter em Window Size Class
App-level decision?
Sim
Não
Escolher estrutura do app
1 pane / 2 pane / rail / drawer
Content-level decision?
Sim
Não
Configurar layout do destino
Grid, Columns, Navigation Rail
Individual composable logic
Usar BoxWithConstraints
ou layout responsivo

Adaptive Layouts

Componentes Reutilizáveis e Adaptativos

- Não devem depender da janela global
- Devem reagir ao espaço que recebem
- Boas escolhas:
  - Modifier.fillMaxSize()
  - BoxWithConstraints (adaptativo)

Adaptive Layouts

Princípios Fundamentais de Adaptive Layouts

- Evite decisões baseadas em hardware físico
- O tamanho físico do dispositivo não importa mais
- O app pode:
  - estar em modo multi-janela,
  - ter janela redimensionada,
  - usar a metade de uma tela dobrável,
  - estar em um monitor de 27 polegadas

Adaptive Layouts

A Métrica Fundamental: Window Size Classes (WSC)

- O Google substituiu as antigas métricas baseadas em densidade (sw600dp, large, etc.) por um sistema que classifica o tamanho da janela do aplicativo em três categorias básicas para largura e altura
- O WSC define breakpoints em dp que categorizam o espaço disponível em:
  - Compact, Medium, Expanded

```
currentWindowAdaptiveInfo().windowSizeClass
```

Adaptive Layouts

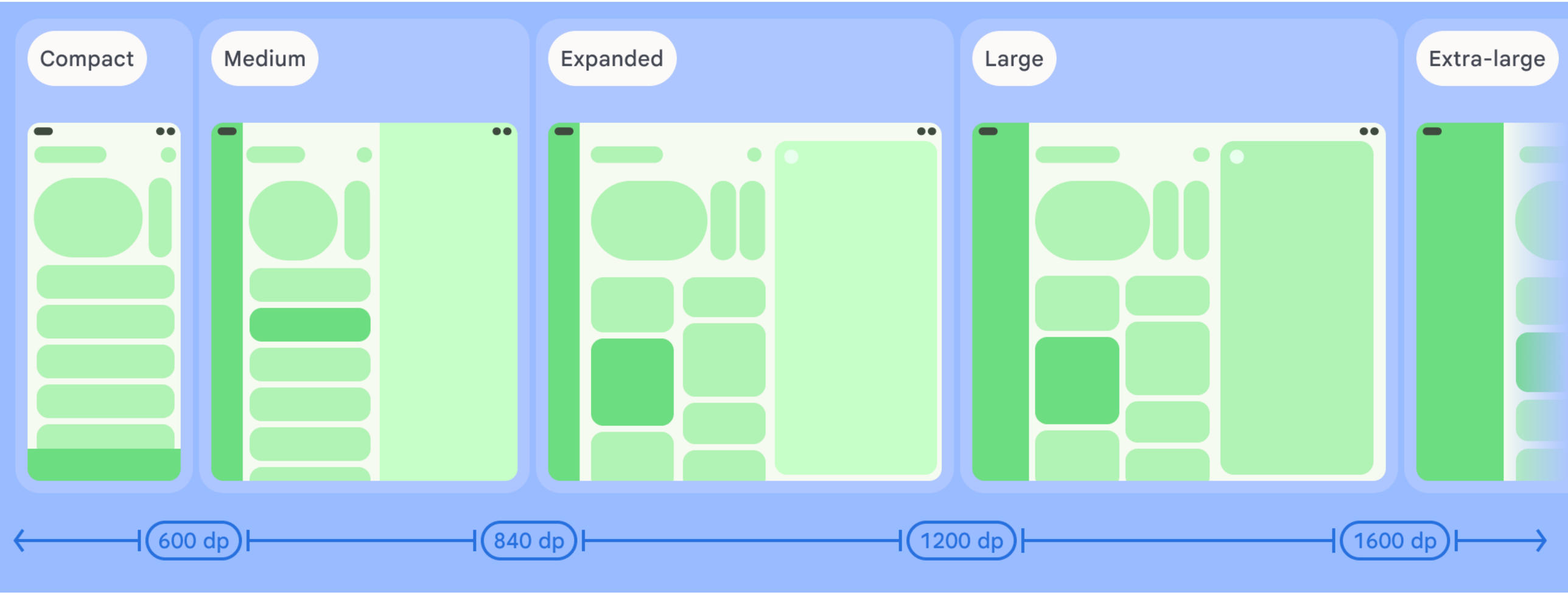
Windows Size Classes

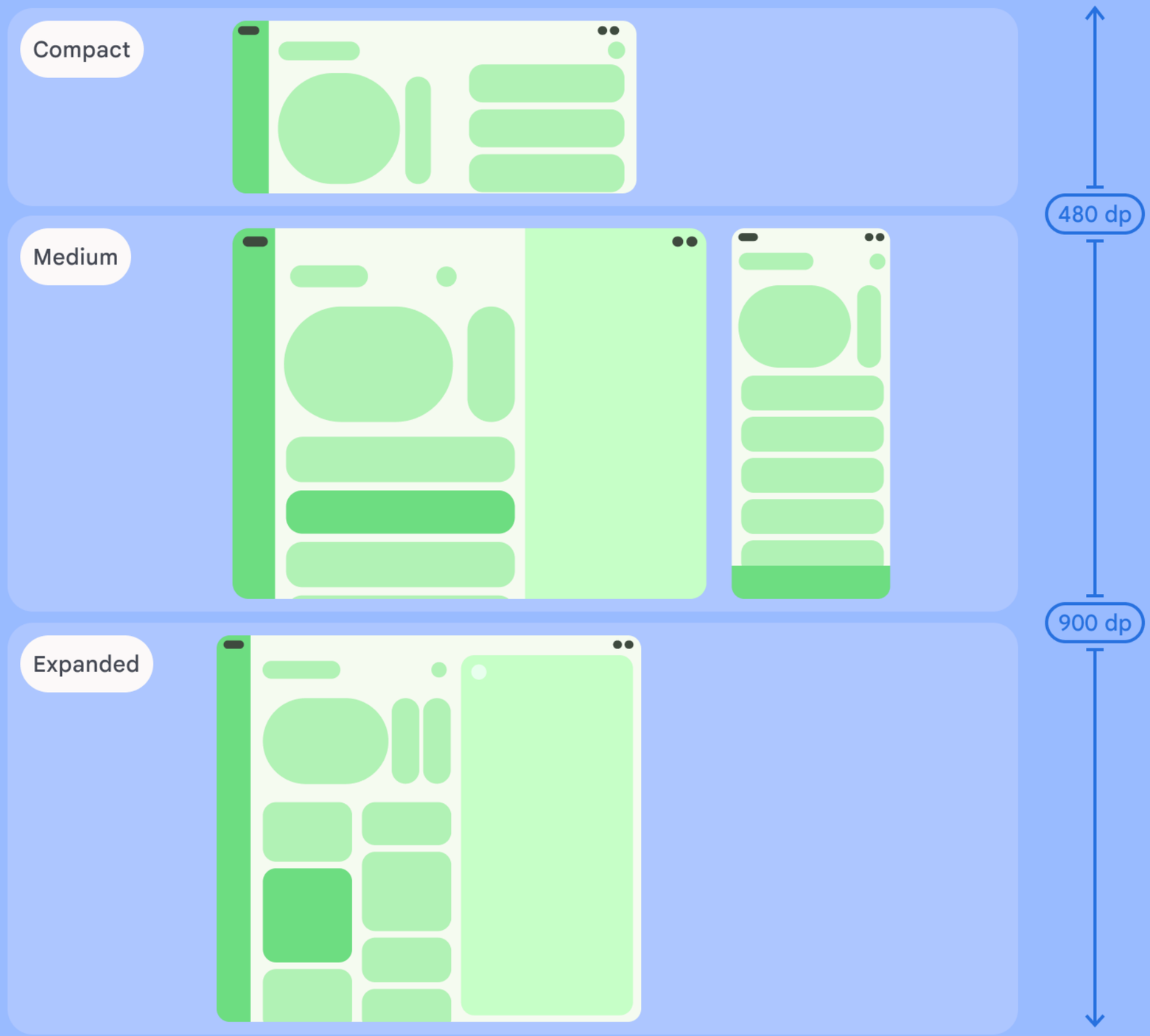
Size class	Breakpoint	Device representation
Compact width	`width < 600dp`	`99.96% of phones in portrait`
Medium width	`600dp ≤ width < 840dp`	`93.73% of tablets in portrait, most large unfolded inner displays in portrait`
Expanded width	`840dp ≤ width < 1200dp`	`97.22% of tablets in landscape, most large unfolded inner displays in landscape are at least expanded width`
Large width	`1200dp ≤ width < 1600dp`	`Large tablet displays`
Extra-large width	`width ≥ 1600dp`	`Desktop displays`

Adaptive Layouts

Windows Size Classes

Size class	Breakpoint	Device representation
Compact height	`height < 480dp`	`99.78% of phones in landscape`
Medium height	`480dp ≤ height < 900dp`	`96.56% of tablets in landscape,` `97.59% of phones in portrait`
Expanded height	`height ≥ 900dp`	`94.25% of tablets in portrait`


Compact
Medium
Expanded
Large
Extra-large
600 dp
840 dp
1200 dp
1600 dp


Compact
Medium
Expanded
480 dp
900 dp

A maioria dos aplicativos consegue criar uma interface de usuário adaptável considerando apenas a classe de tamanho da janela (width)

Adaptive Layouts

Windows Size Classes

```
@Composable
fun MyAppAdaptive() {

    val windowSizeClass =
calculateWindowSizeClass(LocalContext.current as Activity)


    MainLayout(windowSizeClass.widthSizeClass)
}
```

Adaptive Layouts

Windows Size Classes

```
if (widthSizeClass == WindowWidthSizeClass.Expanded) {
    // Layout para tablet: Row com dois painéis
    TwoPaneLayout()
} else {
    // Layout para celular: Column
    SinglePaneLayout()
}
```

Adaptive Layouts

Como Planejar Layouts Adaptativos

- Quais são as partes principais da tela? (lista, detalhes, menu)
- Quais partes fazem sentido aparecer juntas?
- Em telas pequenas, o que deve ser oculto?
- Em telas grandes, o que pode aparecer simultaneamente?
- Como a navegação muda entre single- e multi-pane?

Adaptive Layouts

📦 Estratégia recomendada para iniciantes

- Comece projetando para Compact
- Depois adicione comportamento para Medium
- Por fim suporte para Expanded
- Crie telas modulares
- Use composables reutilizáveis
- Evite lógica de tamanho espalhada
- Centralize a decisão de layout

Adaptive Layouts

Estratégias de Layout Adaptável

- 1. Layout Único com Comportamento Flexível
  - Mesma tela muda comportamento conforme width
  - Ex.: coluna → duas colunas → painel lateral fixo
- 2. Multiformat Layout (layouts específicos)
  - Você troca estruturas inteiras quando os breakpoints mudam

Adaptive Layouts

Estratégias de Layout Adaptável

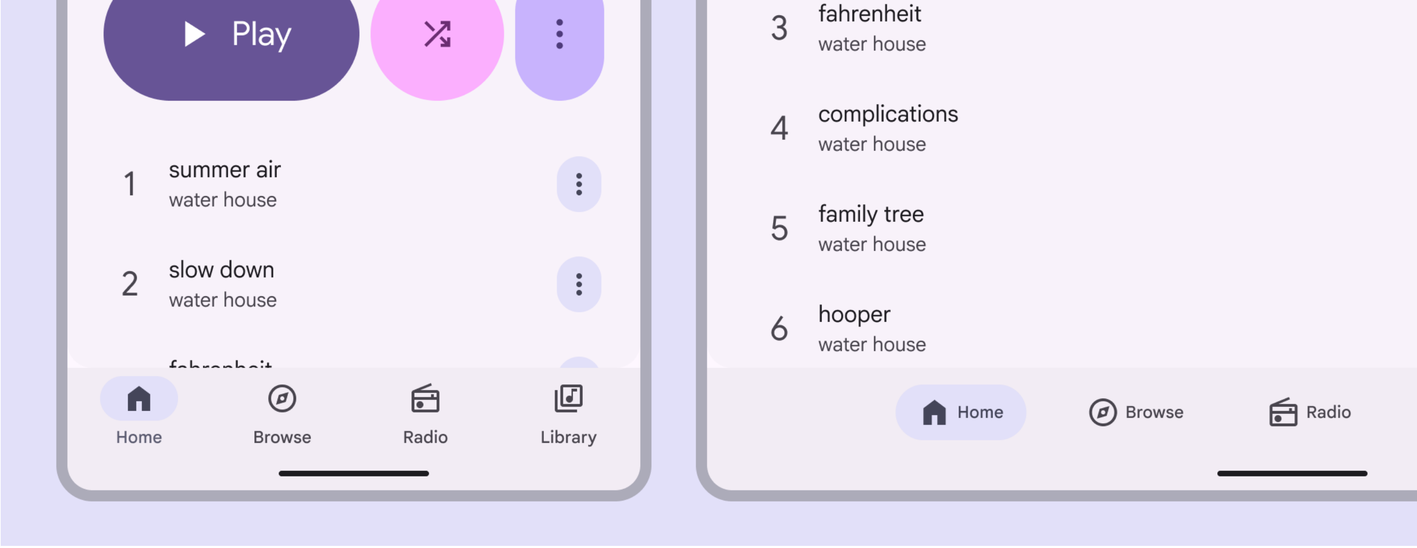
- 3. Componentes responsivos
  - Cards que expandem
  - Grids que mudam o número de colunas
  - Toolbars que se transformam
- 4. Navigation patterns diferentes
  - Bottom bar (compact)
  - Navigation rail (medium)
  - Navigation drawer (expanded)

Layouts canônicos

A navegação deve ser o primeiro elemento a se adaptar, pois define a estrutura de toda a tela

Layouts canônicos

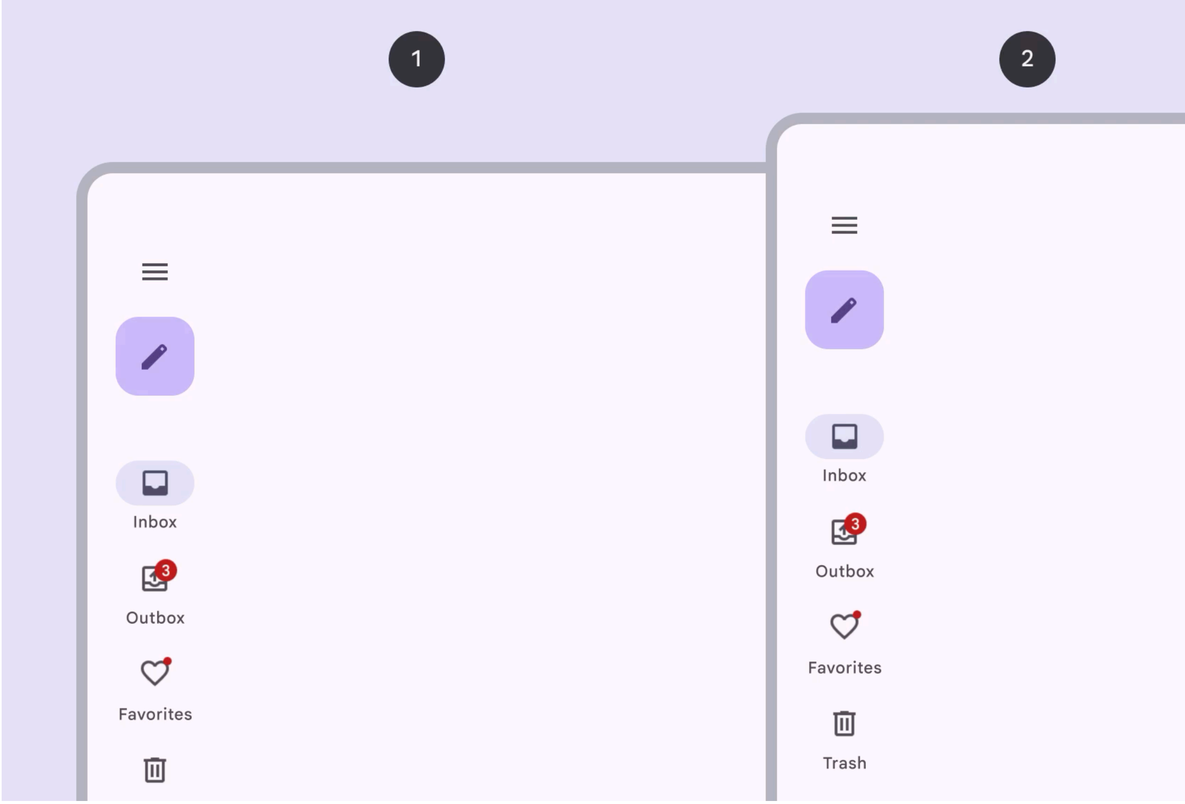
Navegação adaptativa



- Navegação Inferior
  - WSC: Compact (< 600 dp).
  - Uso: Usado em celulares para 3 a 5 destinos principais
  - Motivo: O polegar do usuário alcança facilmente a parte inferior da tela

Layouts canônicos

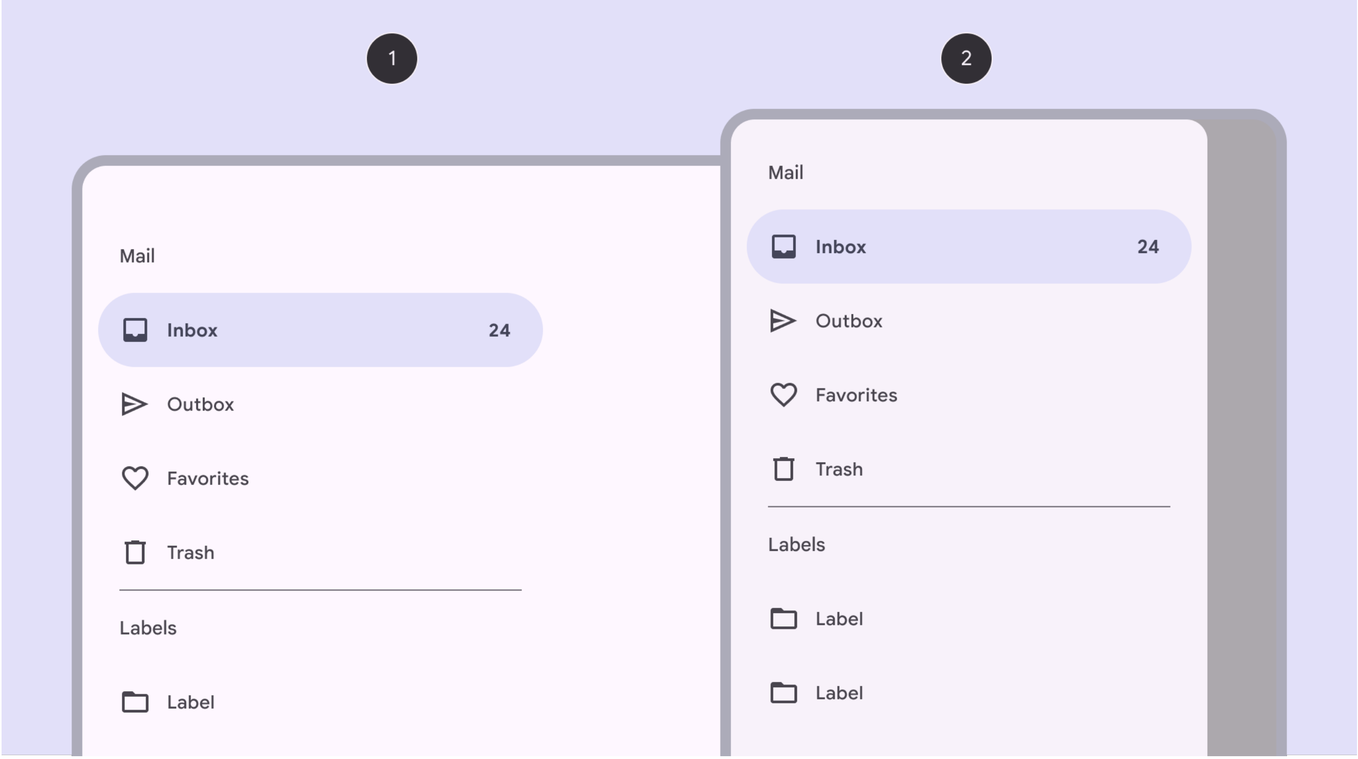
Navegação adaptativa



- Trilho de Navegação (Navigation Rail)
  - WSC: Medium (600 dp+) ou Expanded
  - Uso:
    - Substitui a barra inferior em telas mais largas
    - É uma barra vertical fixa na lateral
- Motivo: Em telas largas, a lateral é mais ergonômica para o toque, e a barra inferior ocuparia espaço horizontal desnecessariamente

Layouts canônicos

Navegação adaptativa



- Menu Lateral Permanente
  Permanent Navigation Drawer
  - WSC: Expanded
  - Uso:
    - Opção para aplicativos com muitos destinos
    - O menu lateral fica sempre visível e não pode ser fechado
  - Motivo: Utiliza o grande espaço horizontal para mostrar a estrutura completa do app sem exigir cliques adicionais.

Layouts canônicos

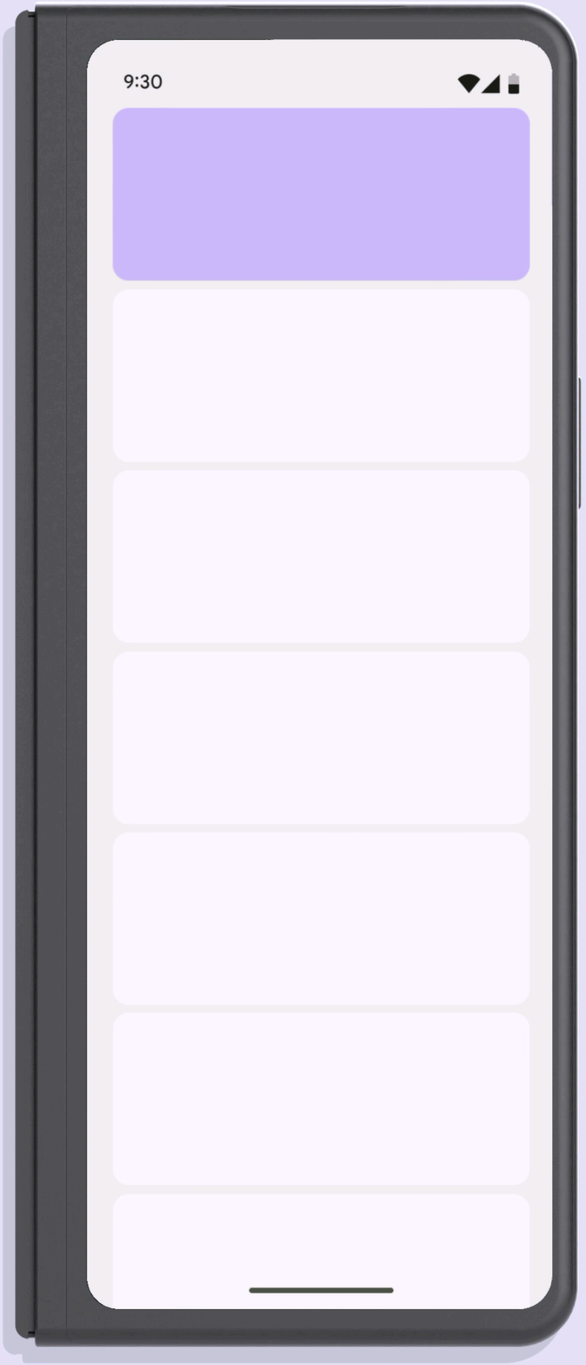
- São modelos de design que fornecem diretrizes para a estrutura visual de um aplicativo em diferentes tamanhos de tela
- O Material 3 definem 3 layouts canônicos:
  - Layout Lista-Detalhe
  - Layout Painel de Suporte
  - Layout de Feed

Adaptive Layouts

List-Detail

- Este é o padrão para gerenciar coleções de itens

1. Para Telas Compactas (WSC < 600 dp)
    - Estrutura: Sequencial. A lista e o detalhe são telas completamente separadas (destinos de navegação diferentes)
    - Comportamento: O usuário clica em um item da lista, e a tela de detalhe substitui a tela de lista
    - Navegação: Usa a pilha de retorno (back stack) padrão
2. Para Telas Médias e Expandidas (WSC > 600 dp)
    - Estrutura: Simultânea (Dois Painéis). A lista e o detalhe são exibidos lado a lado dentro de uma Row
    - Comportamento: O clique em um item da lista apenas atualiza o painel de detalhes à direita, sem alterar a URL de navegação principal


9:30

Adaptive Layouts

List-Detail no Navigation 3

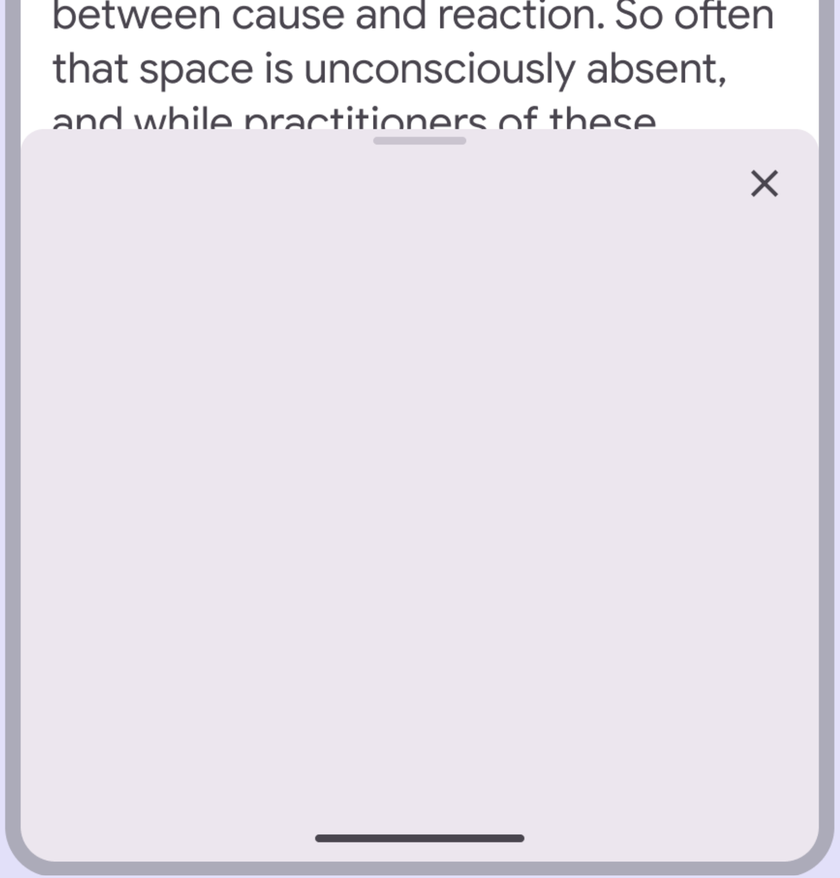
```
val listDetailStrategy = rememberListDetailSceneStrategy<NavKey>()

NavDisplay(
 backStack = backStack,
 sceneStrategy = listDetailStrategy,
 entryProvider = entryProvider {
    entry<ConversationList>(
       metadata = ListDetailSceneStrategy.listPane(
       detailPlaceholder = {
         ContentYellow("Choose a conversation from the list")
       })
     ) { // Conteudo  }
     entry<ConversationDetail>(
         metadata = ListDetailSceneStrategy.detailPane()
     ) { conversation ->   // Conteudo }
     entry<Profile>(
         metadata = ListDetailSceneStrategy.extraPane()) {// Conteudo }
     })
 }}
```

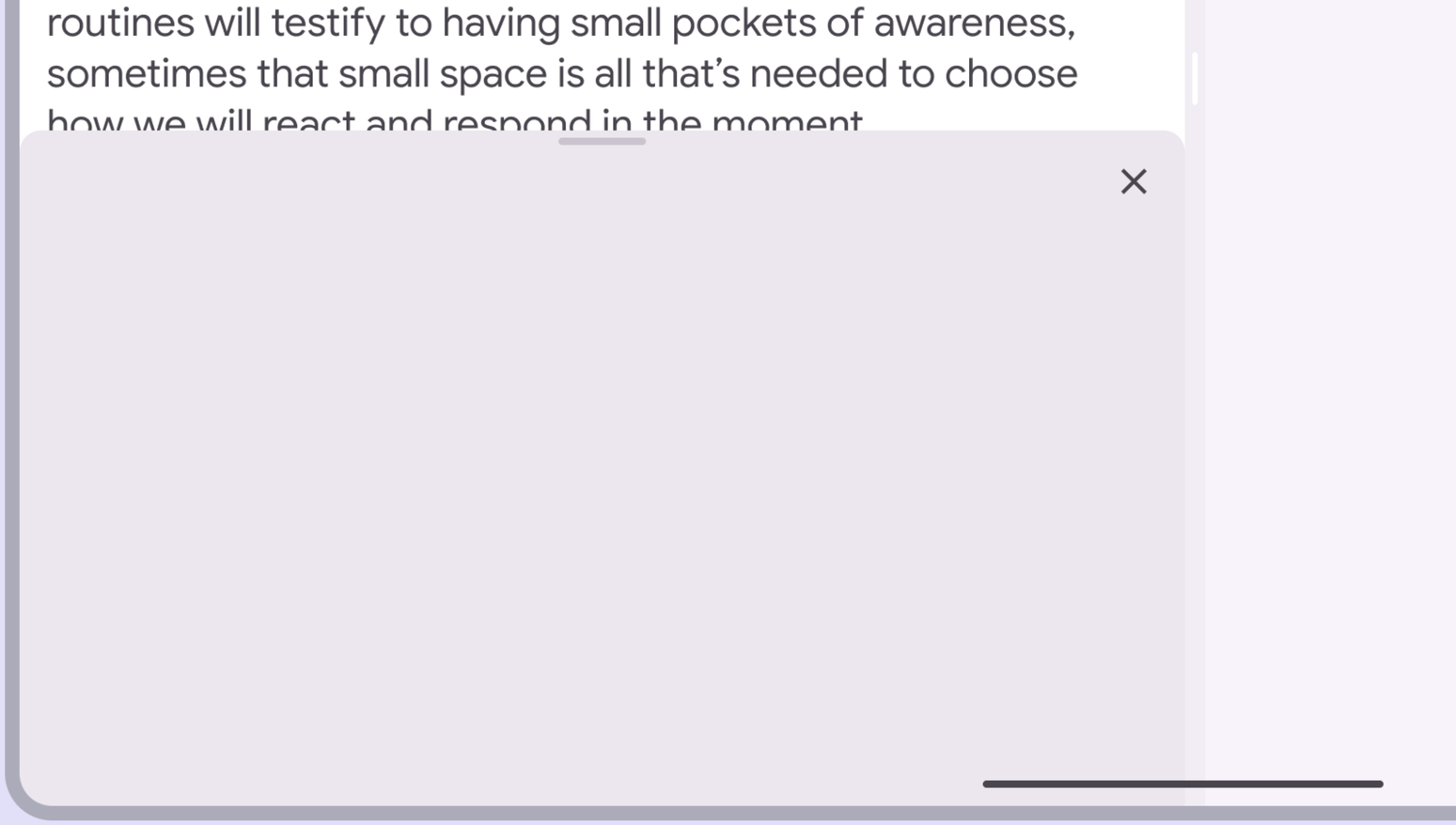
Adaptive Layouts

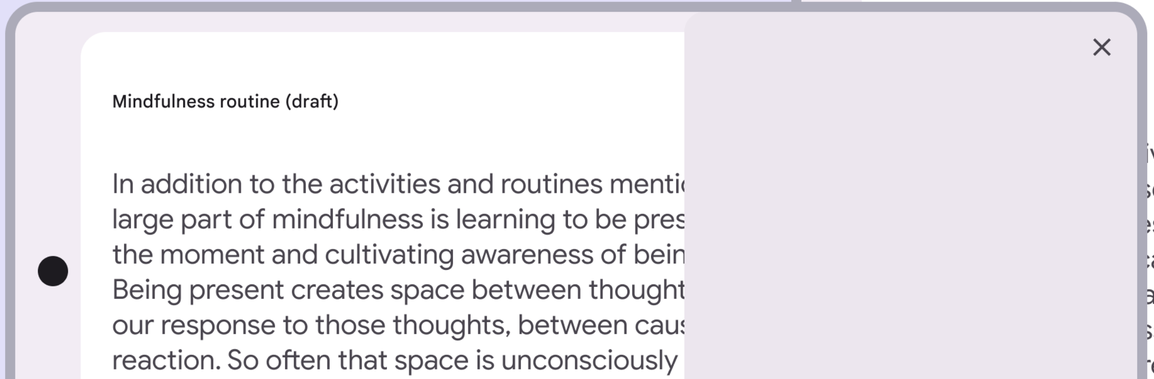
Supporting Panel

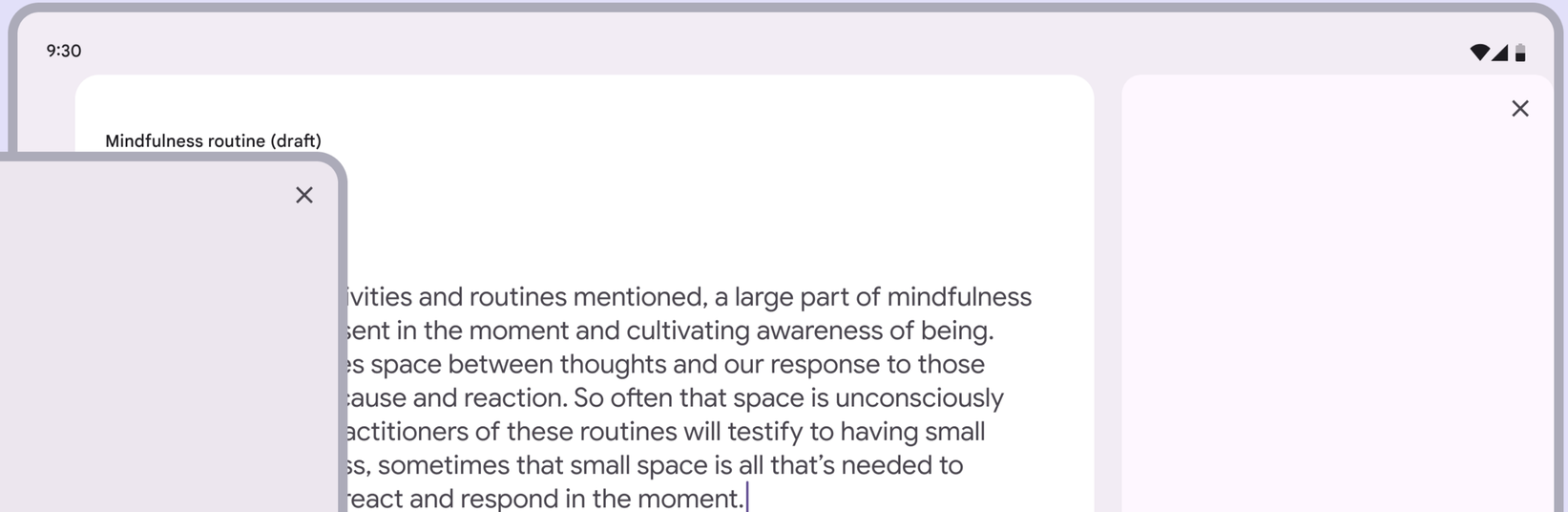
- Usado para complementar o conteúdo principal com informações secundárias ou contexto.

1. Telas Compactas
    - Estrutura: O painel de suporte é geralmente escondido ou colocado em uma aba separada.
2. Telas Expanded
    - Estrutura: Painel Fixo. O painel de suporte (ex: comentários, detalhes de um item, chat) é exibido em uma coluna estreita, geralmente na direita.


between cause and reaction. So often
that space is unconsciously absent,


routines will testify to having small pockets of awareness,
sometimes that small space is all that's needed to choose


Mindfulness routine (draft)
In addition to the activities and routines menti
large part of mindfulness is learning to be pres
the moment and cultivating awareness of bein
Being present creates space between thought
our response to those thoughts, between caus
reaction. So often that space is unconsciously


9:30
Mindfulness routine (draft)
ivities and routines mentioned, a large part of mindfulness
sent in the moment and cultivating awareness of being.
s space between thoughts and our response to those
ause and reaction. So often that space is unconsciously
actitioners of these routines will testify to having small
ss, sometimes that small space is all that's needed to
react and respond in the moment.

Adaptive Layouts

Supporting Panel no Navigation 3

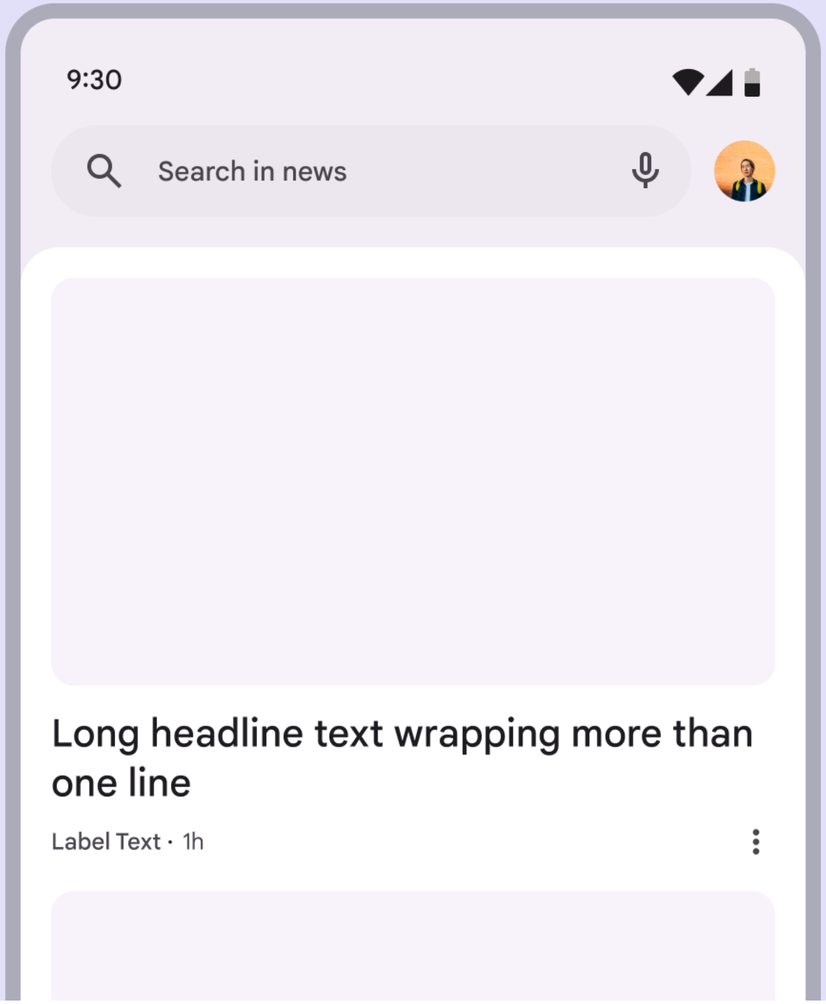
```
val supportingPaneStrategy = rememberSupportingPaneSceneStrategy<NavKey>(
   backNavigationBehavior =
BackNavigationBehavior.PopUntilCurrentDestinationChange)

NavDisplay(
  backStack = backStack,
  sceneStrategy = supportingPaneStrategy,
  entryProvider = entryProvider {
     entry<MainVideo>(
        metadata = SupportingPaneSceneStrategy.mainPane()
     ) { // Conteúdo }
     entry<RelatedVideos>(
        metadata = SupportingPaneSceneStrategy.supportingPane()
     ) { //Conteúdo }
     entry<Profile>(
        metadata = SupportingPaneSceneStrategy.extraPane()
     ) { // Conteúdo}
  })
```

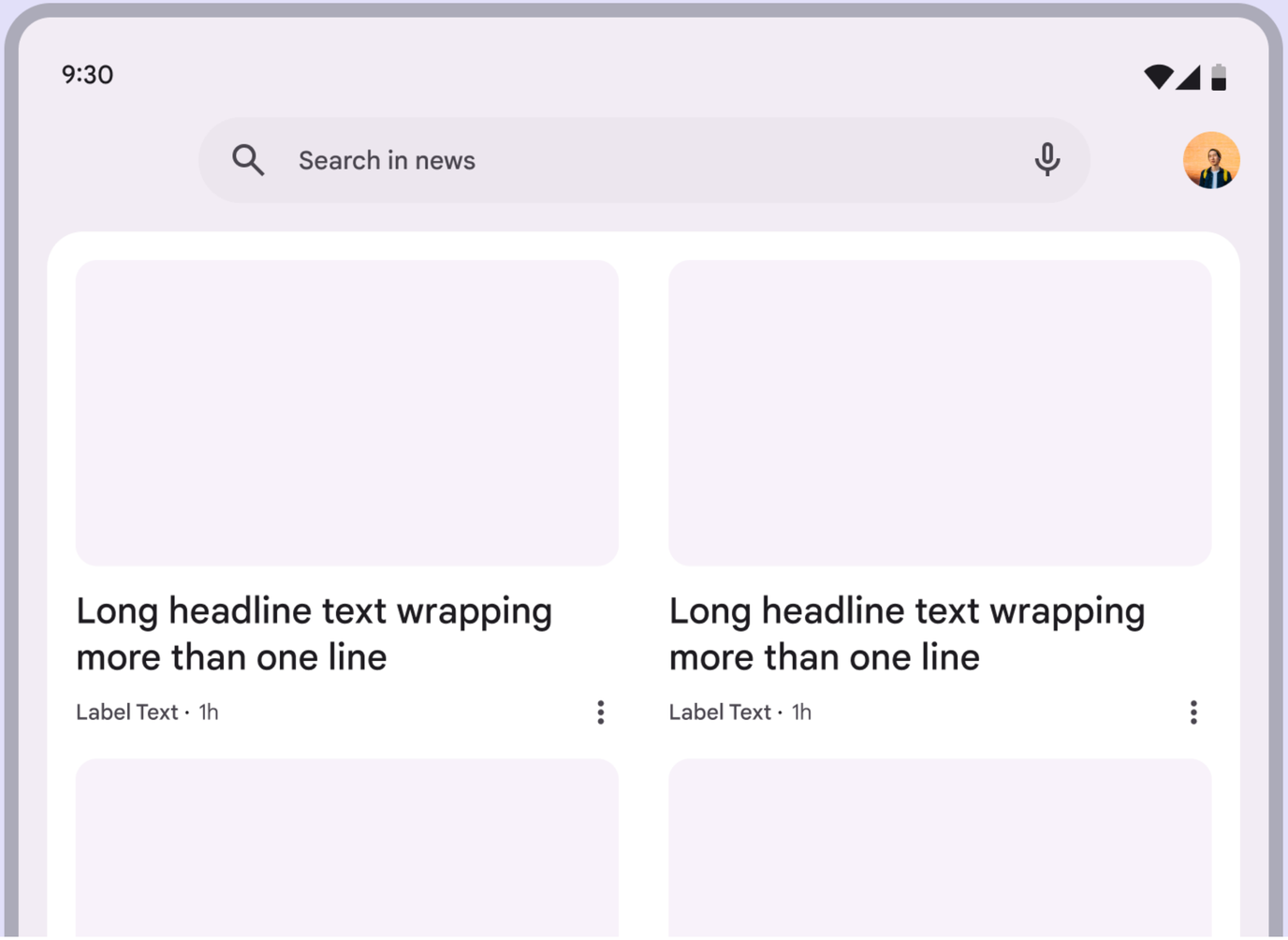
Adaptive Layouts

Feed Layout

- O Layout de Feed é usado para exibir um fluxo contínuo de conteúdo (ex: redes sociais, notícias).

1. Para Telas Compactas (WSC < 600 dp)
    - Estrutura: Coluna Única. O conteúdo é disposto verticalmente em uma única LazyColumn
    - Comportamento: O layout maximiza a legibilidade
2. Para Telas Expandidas (WSC > 840 dp)
    - Estrutura: Colunas Múltiplas. O conteúdo é disposto em um LazyVerticalGrid ou LazyVerticalStaggeredGrid para utilizar a largura extra
    - Vantagem: Evita que os cartões de conteúdo fiquem muito largos, o que prejudicaria a leitura
    - Implementação Prática: Use LazyVerticalGrid com GridCells.Adaptive


9:30
Search in news
Long headline text wrapping more than one line
Label Text · 1h


9:30
Search in news
Long headline text wrapping more than one line
Label Text · 1h
Long headline text wrapping more than one line
Label Text · 1h

Cenas

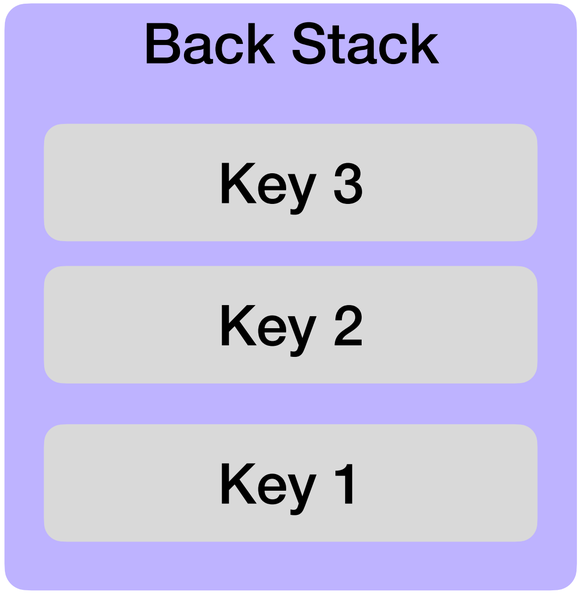
Cenas

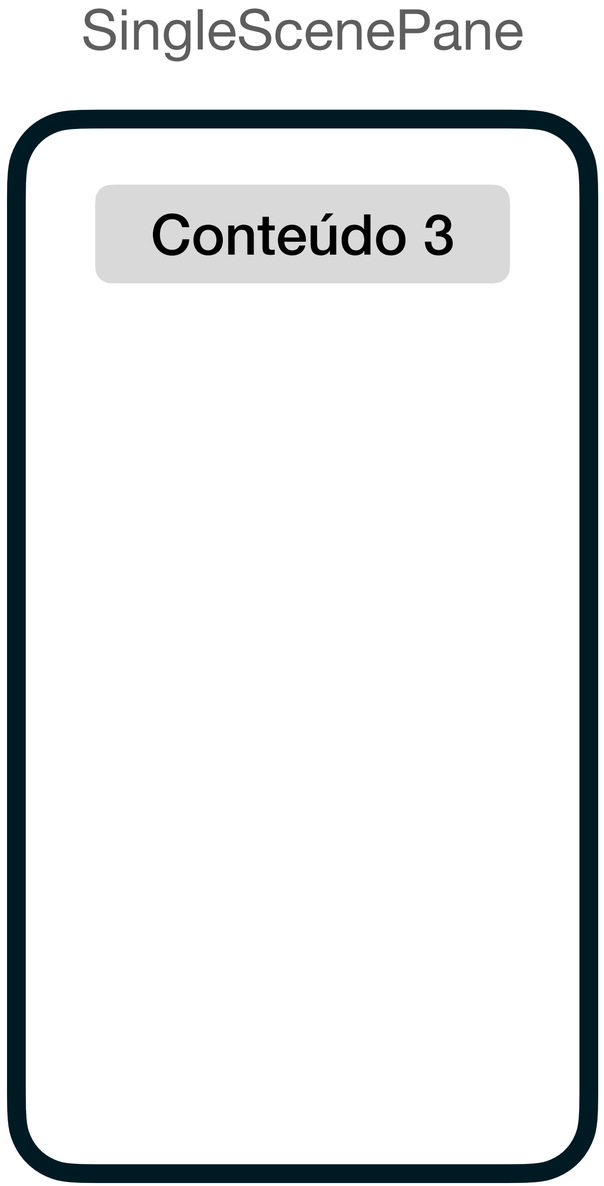
Construido layouts customizados

- Uma Scene é capaz de renderizar uma ou mais instâncias de NavEntry
- Pense em uma Scene como uma seção da sua interface de usuário que pode conter e gerenciar a exibição de conteúdo da back stack

Cenas

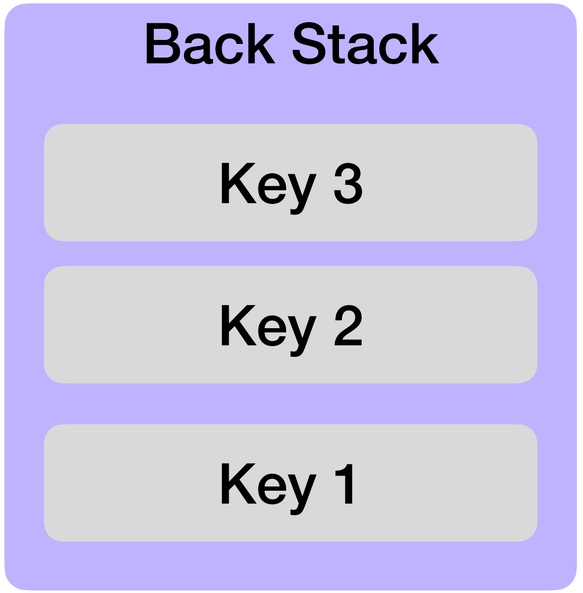
Construido layouts customizados





Cenas

Construido layouts customizados



Cenas

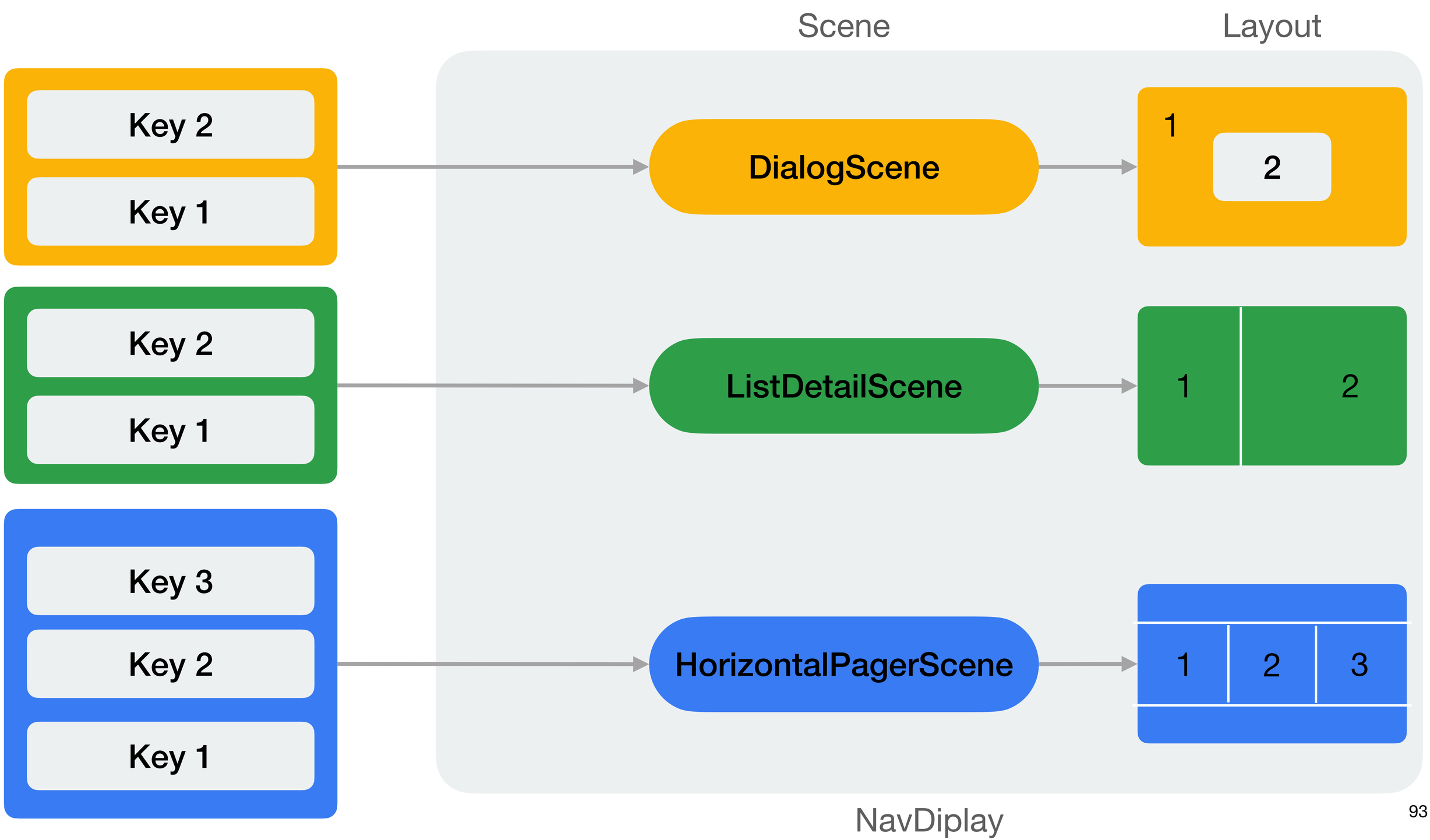
Construido layouts customizados

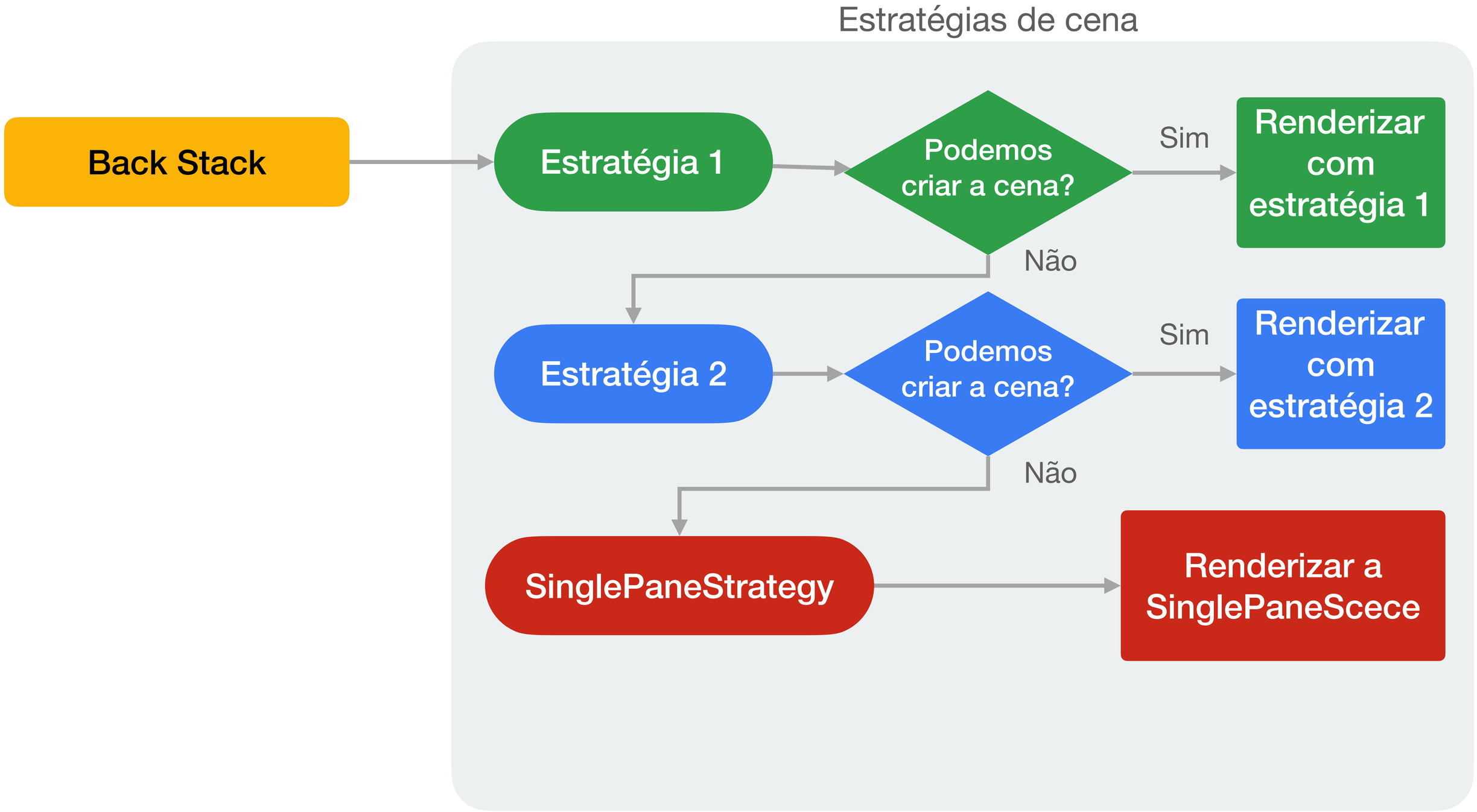
- A interface Scene possui as seguintes propriedades:
- key: Any:
    - O identificador único para esta instância específica de Scene
    - Combinada com a classe da Scene, garante a distinção, principalmente para fins de animação
- entries: List<NavEntry<T>>:
    - A lista de objetos NavEntry que a Scene é responsável por exibir

Cenas

Construido layouts customizados

- previousEntries: List<NavEntry<T>>:
  - Define os NavEntrys que resultariam se uma ação de "voltar" ocorresse a partir da Scene atual.
  - É essencial para calcular o estado preditivo de retorno correto
- conteúdo: @Composable () -> Unit:
  - Esta é a função composable onde você define como a Scene renderiza suas entradas e quaisquer elementos de UI circundantes específicos para essa Scene


Scene
Layout
Key 2
Key 1
DialogScene
1
2
Key 2
Key 1
ListDetailScene
1
2
Key 3
Key 2
Key 1
HorizontalPagerScene
1
2
3
NavDiplay


Estratégias de cena
Back Stack
Estratégia 1
Podemos criar a cena?
Sim
Renderizar com estratégia 1
Não
Estratégia 2
Podemos criar a cena?
Sim
Renderizar com estratégia 2
Não
SinglePaneStrategy
Renderizar a SinglePaneScece

Cenas

Estratégia de cena

- O núcleo de uma SceneStrategy é o seu método calculateScene:

```
@Composable
public fun calculateScene(
    entries: List<NavEntry<T>>,
    onBack: (count: Int) -> Unit,
): Scene<T>?
```

- Deve retornar um `Scene<T>` se conseguir formar uma cena a partir das entradas fornecidas, ou `null` caso contrário

Bibliografia

- Navigation 3
- About adaptive Layouts
- Nav 3 Recipes
- Material Design 3 - Canonical Layouts
- Material Design 3 - Navigation Bar
- Material Design 3 - Navigation Rail
- Material Design 3 - Navigation Drawer

Por hoje é só