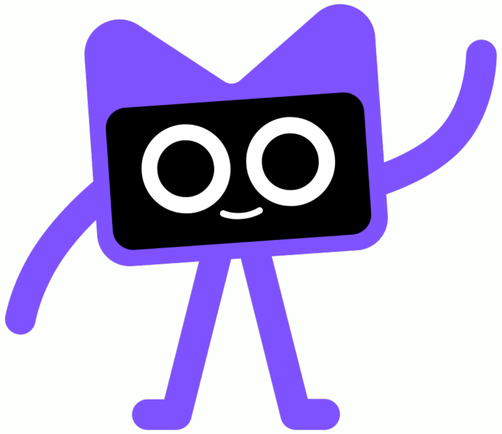
UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

Arquitetura - ViewModel

QXD0276 - Desenvolvimento de Software para Dispositivos Móveis



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

Conteúdo

- Introdução
- Arquitetura MVVM
- View Model
  - Definindo o estado
  - Expondo o estado
  - Produzindo o estado

Introdução

Introdução

Por que se preocupar com arquitetura?

- Problemas comuns:
  - Código confuso e difícil de manter
  - Lógica espalhada entre telas
  - Dificuldade para testar e corrigir bugs

Introdução

Restrições no Android

💥 Componentes Voláteis

- Componentes como Activity e Fragment são de curta duração e voláteis
- O sistema operacional os destrói e recria a qualquer momento, seja por:
  - Ações do Usuário:
    - Rotação de tela, dobrar um dispositivo foldable (mudança de configuração)
  - Restrições de Recurso:
    - O SO mata o processo do aplicativo para liberar memória

Introdução

Restrições no Android

📱 Múltiplos Fatores de Forma (Adaptabilidade)

- Um aplicativo não pode assumir uma orientação fixa (retrato/paisagem) ou um tamanho de tela
- A arquitetura deve suportar layouts adaptáveis (adaptive layouts)
- A UI deve se ajustar a classes de tamanho de janela (window size classes) sem perder o estado

Introdução

Restrições no Android

🚫 Regra: Não Armazene Estado ou Dados na UI

- Se você fizer isso, o progresso do usuário será perdido quando o componente for recriado
- A função da UI é apenas hospedar e exibir a interface

Introdução

A “arquitetura” tradicional

- A "God Activity”
  - Toda a lógica de negócio, a lógica de UI, as chamadas de banco de dados e as chamadas de rede dentro da Activity ou do Fragment
  - Código confuso e difícil de manter
  - Lógica espalhada entre telas
- Problema de Testabilidade
  - O código se torna impossível de testar, pois a lógica está misturada com a UI
  - Dificuldade para testar e corrigir bugs

Introdução

A “arquitetura” tradicional

- Problema do Ciclo de Vida
  - A Activity é destruída e recriada frequentemente (ex: rotação de tela).
  - Se a lógica de negócio estiver lá, ela é perdida ou precisa ser recarregada, causando bugs e desperdício de recursos

Introdução

Princípios Fundamentais de uma Boa Arquitetura Android

- Separação de Preocupações (Separation of Concerns)
  - Divisão em módulos, onde cada módulo tem uma responsabilidade única e bem definida
  - Exemplo: O código que busca dados na rede não deve estar na mesma classe que exibe um botão na tela
  - Isso torna o código mais testável, pois você pode testar a lógica de rede isoladamente

Introdução

Princípios Fundamentais de uma Boa Arquitetura Android

- A UI deve ser uma função dos modelos de dados (Models)
  - Os modelos de dados representam o estado do aplicativo (ex: um objeto Movie ou User)
    - Devem ser persistentes (salvos em banco de dados) para evitar a perda de dados, mesmo quando o OS destrói o processo

Introdução

Princípios Fundamentais de uma Boa Arquitetura Android

- Fonte única da verdade (Single Source of Truth - SSOT)
  - Designe um único "dono" para cada tipo de dado
    - A SSOT é a proprietária desses dados, e apenas ela pode fazer mudanças neles
      - Expõe os dados usando um tipo imutável
- Benefícios:
  - Centraliza todas as mudanças de um tipo específico de dados em um só lugar
  - Protege os dados para que outros tipos não possam fazer adulterações neles
  - Torna as mudanças mais rastreáveis e, assim, os bugs são mais fáceis de detectar

Introdução

Princípios Fundamentais de uma Boa Arquitetura Android

- Fluxo de Dados Unidirecional (Unidirectional Data Flow -UDF)
  - O estado flui em uma única direção
  - Os eventos que modificam o dado fluem na direção oposta
  - Isso torna os bugs muito mais fáceis de rastrear (traceable)
  - O UDF, combinado com SSOT, garante que as mudanças sejam previsíveis

Introdução

Arquitetura recomendada e seus benefícios

✅ Melhor manutenabilidade, qualidade e robustez do aplicativo

✅ Permite que o aplicativo seja escalável

- Mais pessoas podem contribuir para a mesma base de código com o mínimo de conflitos

✅ Facilita a integração

- Novos membros da equipe podem se adaptar rapidamente e se tornar mais eficientes em menos tempo

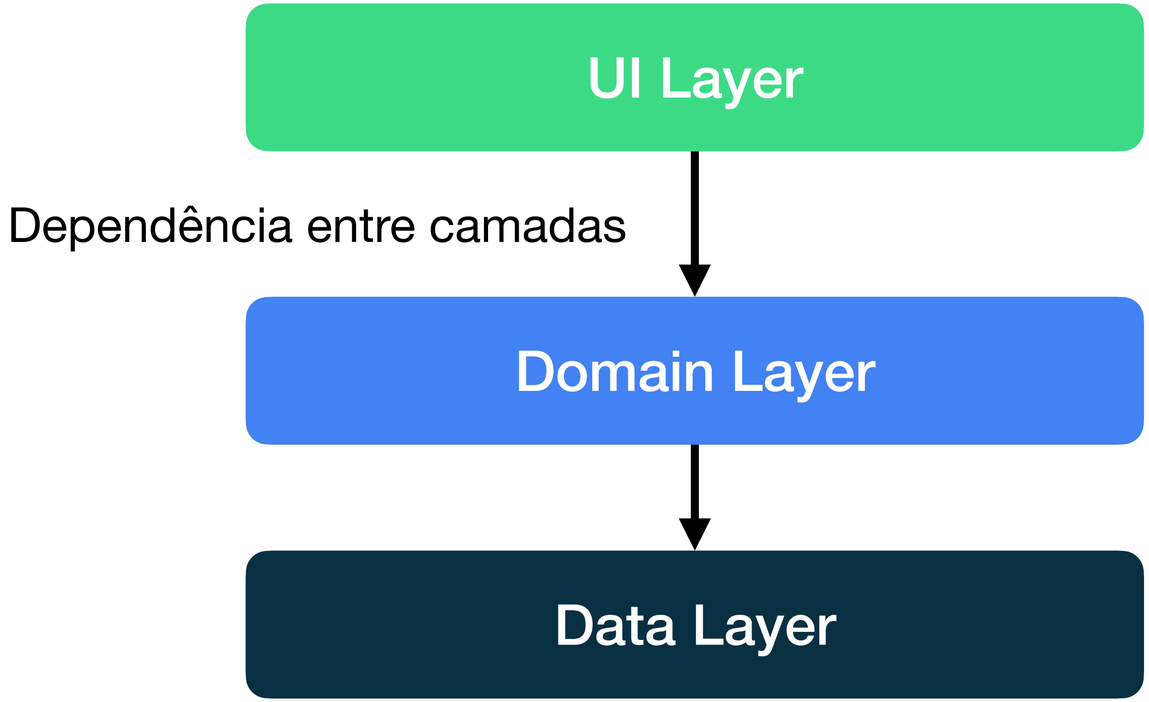
✅ Facilita os testes

- Uma boa arquitetura incentiva tipos mais simples, que geralmente são mais fáceis de testar.

✅ Os bugs podem ser investigados metodicamente com processos bem definidos.

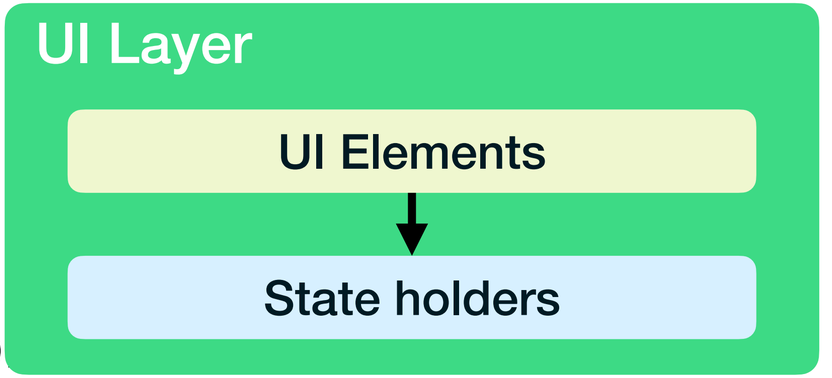
A arquitetura recomendada

A arquitetura recomendada



A arquitetura recomendada

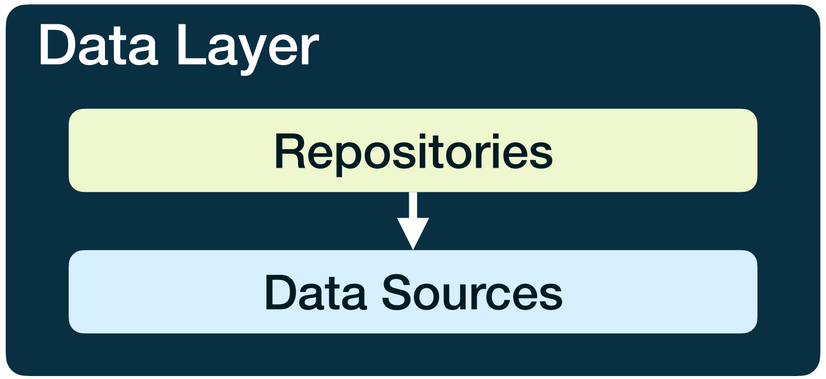
Camada de UI (Interface do Usuário)



- Responsabilidade
  - Exibir os dados na tela e coletar eventos do usuário
- Componentes: Elementos de UI (Composable) e so State Holder (ViewModel)
  - ViewModel: É a ponte entre a View e os dados
    - Mantém o estado da UI
    - Garante que ele sobreviva a mudanças de configuração

A arquitetura recomendada

Camada de Dados (Data Layer)



- Responsabilidade
  - Contém a lógica de negócio
- Componentes:
  - Repositórios:
    - Classes criadas para cada tipo de dado (ex: UserRepository)
    - São responsáveis por expor dados e resolver conflitos entre fontes
- Fontes de Dados:
  - Classes que lidam com uma única fonte de dados (ex: UserDatabase - para dados locais, UserApi - para dados de rede)

A arquitetura recomendada

Camada de Domínio (Domain Layer)

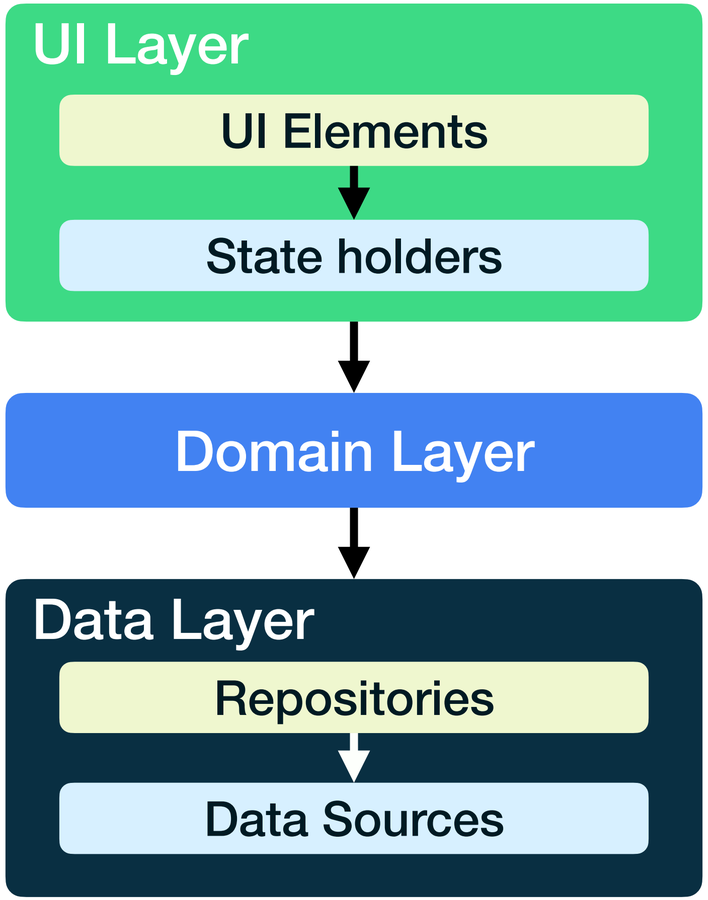
Domain Layer

- Responsabilidade:
  - Encapsular lógica de negócio complexa ou lógica que é reusada por múltiplos ViewModels.
  - Componentes:
    - Casos de Uso (Use Cases ou Interactors)
  - Quando usar:
    - Apenas quando houver complexidade ou necessidade de reusar regras de negócio específicas (ex: LoginUserUseCase, FormatDataUseCase)

A arquitetura recomendada

Arquitetura MVVM

Camada	Papel	Exemplo
Model	Representa os dados e regras de negócio	User, Task, Product
ViewModel	Contém a lógica da tela e o estado	TaskViewModel
View (Compose)	Renderiza o que o usuário vê	TaskListScreen()



A arquitetura recomendada

≡ Arquitetura em camadas

Camada	O que faz	Exemplo
UI	`Mostra dados, envia eventos`	`Composables`
ViewModel	`Gerencia estado e lógica`	`TaskViewModel`
Repository	`Fornece dados de fontes diversas`	`UserRepository`
Data Source	`Implementa acesso a dados`	`API, Room, arquivos`

A arquitetura recomendada

🧩 Principais ferramentas

🧠 ViewModel: mantém o estado da UI entre rotações

🌊 LiveData / StateFlow: fornece dados reativos à UI

🧭 Navigation: gerencia telas e pilha de navegação

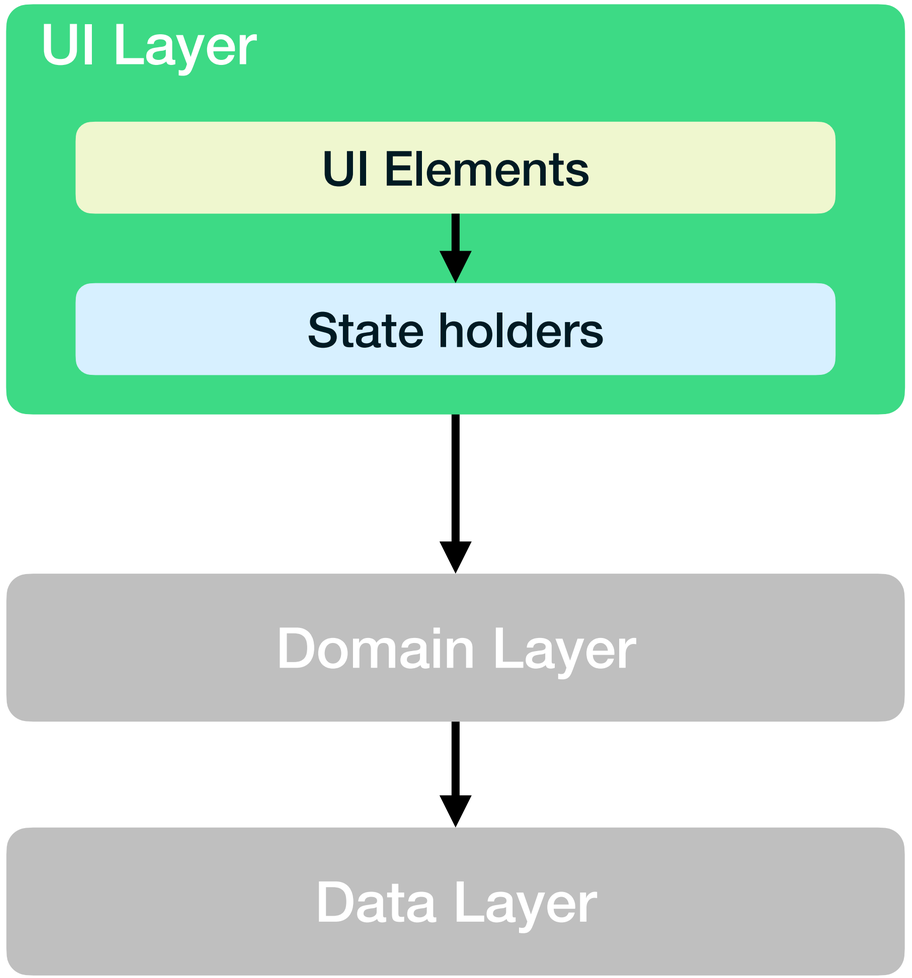
💾 Room: persistência local com SQLite

📡 Repository: abstrai acesso a dados remotos e locais

⚙️ WorkManager: executa tarefas em background

Camada da UI

Camada da UI



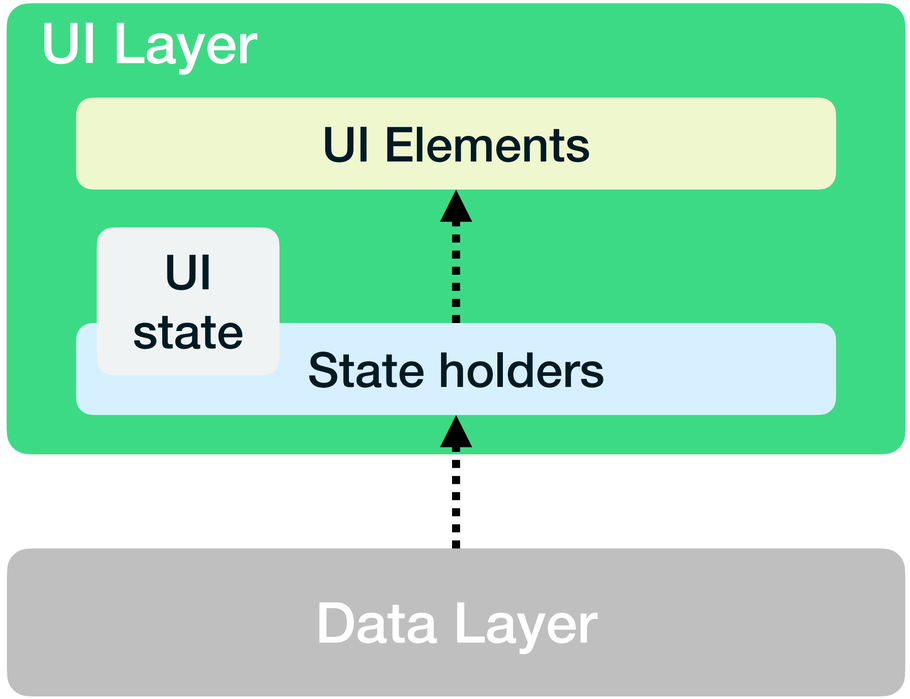
Camada da UI

- Responsabilidade:
  - Exibir os dados da aplicação na tela
  - Servir como o principal ponto de interação do usuário
- Sempre que os dados mudam, a UI deve ser atualizada para refletir essas mudanças
  - Interação do usuário (como pressionar um botão)
  - Entrada externa (como uma resposta da rede)

No entanto, os dados obtidos da camada de dados geralmente estão em um formato diferente do que você precisa exibir.

Ex: você pode precisar apenas de parte dos dados, ou pode precisar mesclar duas fontes de dados para apresentar informações relevantes ao usuário.

A camada de UI é o pipeline que converte os dados para um formato que a interface do usuário pode exibir



Camada da UI

Passos do pipeline

1. Consumir os dados e transformá-los em dados que a interface do usuário pode renderizar
2. Transformar esses dados em elementos da interface
3. Consumir os eventos de entrada do usuário e refletir seus efeitos nos dados da interface do usuário
4. Repita as etapas de 1 a 3 pelo tempo que for necessário

Camada da UI

UI State

Camada da UI

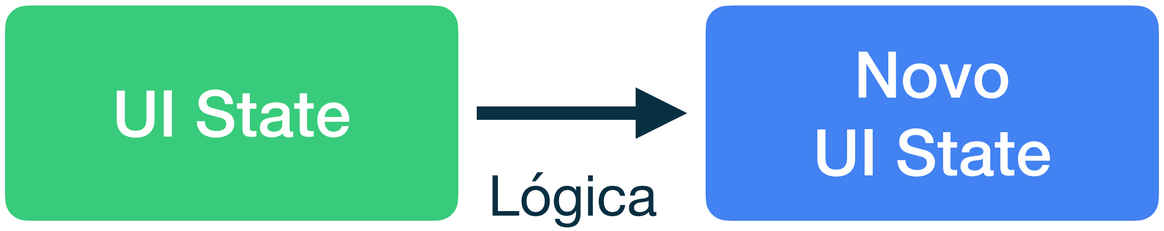
Tipos de estado

Tipo de UI State	O que Descreve	Exemplo
Screen UI State	O conteúdo da tela. Geralmente vem de outras camadas (Dados/ Domínio).	Uma lista de artigos de notícias, a foto do perfil de um usuário.
UI Element State	O estado intrínseco de um elemento da UI.	Um item está expandido ou colapsado, o estado de rolagem de uma lista (LazyListState), se um TextField está em foco.

Camada da UI

UI State

- O estado da interface do usuário não é uma propriedade estática
- Os dados do aplicativo e os eventos do usuário fazem com que o estado da interface mude ao longo do tempo
- A lógica determina os detalhes da mudança
  - Lógica de negócio e Lógica de UI



Camada da UI

Lógica de negócio

Tipo de Lógica	Responsabilidade	Onde Reside (Camada)
Lógica de negócio	O QUE FAZER: Implementação de requisitos de produto (regras de negócio).	Data Layer ou Domain Layer. O State Holder apenas delega essa lógica.
Lógica de UI	COMO EXIBIR: Relacionado à apresentação e ao ciclo de vida da UI.	UI Layer (na função Coposable ou em um State Holder de UI).

Camada da UI

State holders

- Responsabilidade
  - Armazenar o estado para que o aplicativo possa lê-lo
  - Atua como intermediário quando alguma lógica de negócio é necessária
    - Fornece acesso às fontes de dados
  - Produzir um novo estado

Camada da UI

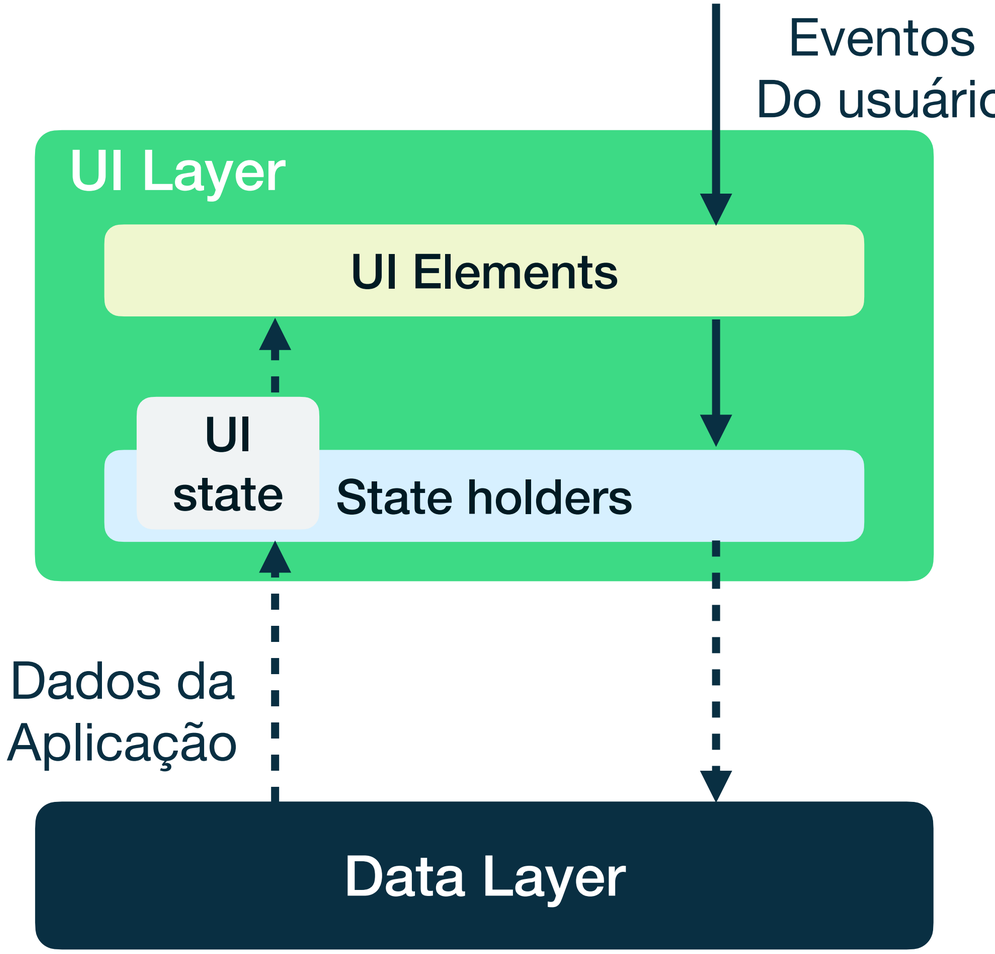
Pipeline de produção de estado

- Estado (State):
  - É o saída da pipeline de produção de estado
  - É a propriedade que sempre existe e descreve o que a UI deve ver
- Evento (Event):
  - É o input da produção de estado
  - É algo transiente, imprevisível (ex: clique, resposta de rede, snackbar timeout).
- Os State Holders processam o evento e produzem um novo estado

Camada da UI

State holders e UDF

- O State holder armazena e expõe o estado a ser consumido pela interface do usuário
- A interface do usuário notifica o ViewModel sobre eventos do usuário
- O State holder lida com as ações do usuário e atualiza o estado
- O estado atualizado é enviado de volta para a interface do usuário para renderização
- O processo acima se repete



Camada da UI

State Holder de lógica de UI

Propriedade	Implicação
Não sobrevive a mudanças de configuração	Tem o mesmo ciclo de vida da UI que o contém (da função Composable)
Reutilizável	Pode ser reutilizado em diferentes partes do aplicativo Ex: um State Holder que gerencia a entrada de dados em um campo de busca.
Contém Context / UI Sources	Pode referenciar fontes de dados com escopo de UI porque compartilha o mesmo ciclo de vida.
Implementação	Classe Simples e criado usando uma função rememberXxxState no Compose.

Camada da UI

State Holder de lógica de negócios - ViewModel

Propriedade	Implicação
Sobrevive à Activity Recreation	Retém o estado e os pipelines de dados através de rotações de tela e mudanças de configuração.
Long-Lived State	Geralmente tem o escopo de um destino de navegação e é mantido em memória (cache) mesmo quando o usuário navega para outra tela.
Business Logic	Acessa o Data Layer (Repositórios) para buscar e modificar dados.
Não é Reutilizável	É único para uma função ou tela específica (ex: AuthorViewModel).

ViewModel

ViewModel

Responsabilidades

Responsabilidade	Detalhamento	Como?
Gerenciar o Estado da UI	Expõe o estado atual da tela (UiState) para que a View o observe e reaja às mudanças.	StateFlow ou Compose State (MutableStateOf).
Coordenar a Lógica	Atua como intermediário para a Lógica de Negócio (chamando o Repositório ou Casos de Uso).	viewModelScope.launch { ... } (Coroutines).
Processar Eventos	Recebe Eventos da View (ex: onLoginClicked) e, como resultado, modifica o estado.	Funções públicas (ex: fun login(user, pass)).
Sobreviver à Recriação	Gerencia a concorrência (CoroutineScope) e o estado, garantindo que as operações assíncronas não sejam interrompidas pelo	ViewModel (nativo do Jetpack).

ViewModel

Sobrevivendo a mudanças de configuração

Veja o ciclo de vida completo

ViewModel

🧠 Benefícios do uso do ViewModel

✅ Persistência de estado entre recriações da tela

✅ Separa responsabilidades (UI ↔ lógica)

✅ Facilidade de testes

✅ Compatível com LiveData, Flow, coroutines e Hilt

✅ Integração direta com Compose (viewModel())

✅ Base para arquiteturas reativas (MVVM)

Definindo o estado

ViewModel

Definindo o UI State (Screen State)

- Deve conter TUDO o que a tela precisa para se renderizar
- Nomenclatura
  - A convenção é utilizar o nome da funcionalidade mais UiState
  - Ex: NewsUiState
- Imutabilidade é lei
  - A interface do usuário concentra-se em uma única função:
    - Ler o estado e atualizar seus elementos de acordo

Nunca deve-se modificar o estado da interface do usuário diretamente na própria interface, a menos que ela seja a única fonte de seus dados

ViewModel

Na prática: definindo o UI State

```
data class NewsUiState(
    val isSignedIn: Boolean = false,
    val isPremium: Boolean = false,
    val newsItems: List<NewsItemUiState> = listOf(),
    val userMessages: List<Message> = listOf()
)

data class NewsItemUiState(
    val title: String,
    val body: String,
    val bookmarked: Boolean = false,
    ...
)
```

ViewModel

Propriedade derivada

```
data class NewsUiState(...)
val NewsUiState.canBookmarkNews: Boolean
    get() = isSignedIn && isPremium // Lógica: Derivada
```

ViewModel

⚠️ Erros comuns

🚫 Lógica de negócio dentro da Activity ou composable

- 🚫 Guardar estado com remember quando deveria estar no ViewModel
- 🚫 Misturar lógica de negócio dentro dos composables

🚫 Criar o ViewModel manualmente com CounterViewModel() (não use new)

🚫 Guardar contexto da Activity no ViewModel (causa memory leak)

🚫 Alterar estados diretamente da UI

Expondo o estado

ViewModel

Expondo o estado

- O estado exposto deve ser observável, portanto, temos algumas opções:
  - Compose State API
    - mutableStateOf e similares
  - StateFlow
  - ⛔ ~~LiveData~~

ViewModel

mutableStateOf

💡 Características

- É parte do runtime do Compose
- Notifica a recomposição automaticamente
- Ideal para pequenos estados de UI

✅ Quando usar

- Quando o estado é usado apenas pelo Compose, isto é:
  - A UI é a única consumidora do dado
  - Não é necessário integração com outras APIs de fluxo reativo (Flow, LiveData, etc)

ViewModel

mutableStateOf

```
class CounterViewModel : ViewModel() {
    var count by mutableStateOf(0)
        private set

}
```

```
@Composable
fun CounterScreen(viewModel: CounterViewModel = viewModel()) {
    Column {
        Text("Contador: ${viewModel.count}")
    }
}
```

ViewModel

🌊 StateFlow: estados em fluxo (reação em cadeia)

💡 Características

- É parte da biblioteca kotlinx.coroutines
- Representa um fluxo de dados reativo que tem sempre um valor atual
- Recomendado para arquiteturas mais robustas (MVVM, MVI).

✅ Quando usar

- O estado precisa ser coletado fora do Compose
- Você quer integrar com coroutines ou outros fluxos (Flow)
- Quando há múltiplas fontes de dados que se combinam reativamente

ViewModel

🌊 StateFlow: estados em fluxo (reação em cadeia)

```
class ContadorViewModel : ViewModel() {
    private val _contador = MutableStateFlow(0)
    val contador: StateFlow<Int> = _contador
}
```

```
@Composable
fun TelaContador(viewModel: ContadorViewModel = viewModel()) {
    val contador by viewModel.contador.collectAsState()
    Text("Contagem: $contador")
}
```

ViewModel

🌊 StateFlow: estados em fluxo (reação em cadeia)

```
class ContadorViewModel : ViewModel() {
    private val  contador = MutableStateFlow(0)
    val contador: StateFlow<Int> = _contador
}
```

```
@Composable
fun TelaContador(viewModel: ContadorViewModel = viewModel()) {
    val contador by viewModel.contador.collectAsState()
    Text("Contagem: $contador")
}
```

ViewModel

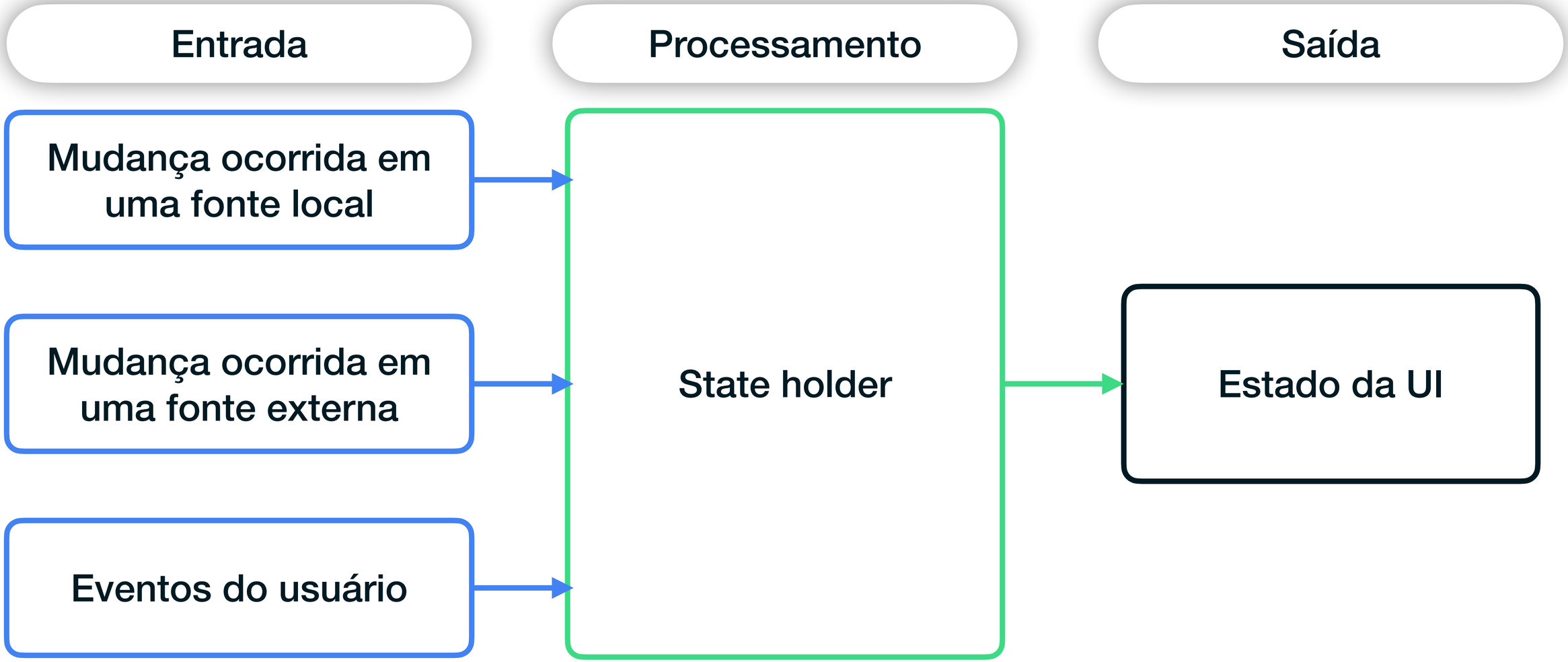
🧰 Boas práticas

- O Composable observa o estado, mas não o modifica diretamente
- Use StateFlow ou mutableStateOf no ViewModel
- Evite armazenar context ou referências a Views
- Use SavedStateHandle se o estado precisar sobreviver à morte do processo

Produzindo o estado

Produzindo o estado

Pipeline de produção



Produzindo o estado

Tipos de entrada

- Os tipos de entrada determinam a maneira de processamento do estado
  - Operações únicas
    - Síncronas
    - Assíncronas
  - Fluxo de dados (Stream API)
    - Fluxos combinados
  - Operações únicas + Fluxo de dados

Produzindo o estado

Operações únicas - síncronas - StateFlow

```
class DiceRollViewModel : ViewModel() {

    private val _uiState = MutableStateFlow(DiceUiState())
    val uiState: StateFlow<DiceUiState> = _uiState.asStateFlow()

    // Called from the UI
    fun rollDice() {
        _uiState.update { currentState ->
            currentState.copy(
            firstDieValue = Random.nextInt(from = 1, until = 7),
            secondDieValue = Random.nextInt(from = 1, until = 7),
            numberOfRolls = currentState.numberOfRolls + 1,
            )
        }
    }
}
```

Produzindo o estado

Operações únicas - síncronas - mutableStateOf

```
class DiceRollViewModel : ViewModel() {
    var uiState by mutableStateOf(DiceUiState())
        private set

    fun rollDice() {
        uiState = uiState.copy(
            firstDieValue = Random.nextInt(from = 1, until = 7),
            secondDieValue = Random.nextInt(from = 1, until = 7),
            numberOfRolls = uiState.numberOfRolls + 1
        )
    }
}
```

Produzindo o estado

Operações únicas - assíncronas - StateFlow

```
class AddEditTaskViewModel(...) : ViewM
    private val _uiState = MutableState
    val uiState: StateFlow<AddEditTaskU

    private fun createNewTask() {
        viewModelScope.launch {
            val newTask = Task(uiState.value.title, uiState.value.description)
            try {
                tasksRepository.saveTask(newTask)
                    _uiState.update { it.copy(isTaskSaved = true)}
            }
            catch(cancelException: CancellationException) {throw cancelException}
            catch(exception: Exception) {
                _uiState.update {it.copy(userMessage = getErrorMessage(exception))}
            }
        }
    }
}
```

É preciso lancar uma Coroutine no viewModelScope.launch { ... }

Cuidado: Coroutines lançadas no viewModelScope rodam até o fim, mesmo que a UI não esteja visível

Produzindo o estado

Operações únicas - assíncronas - mutableStateOf

```
class AddEditTaskViewModel(...) : ViewModel() {
    private val _uiState = MutableStateFlow(AddEditTaskUiState())
    val uiState: StateFlow<AddEditTaskUiState> = _uiState.asStateFlow()

    private fun createNewTask() {
        viewModelScope.launch {
            val newTask = Task(uiState.value.title, uiState.value.description)
            try {
                tasksRepository.saveTask(newTask)
                    _uiState.copy(isTaskSaved = true)
            }
            catch(cancelException: CancellationException) {throw cancelException}
            catch(exception: Exception) {
                _uiState.copy(userMessage = getErrorMessage(exception)
            }
        }
    }
}
```

Produzindo o estado

Trabalhando com Stream API - Flow

```
class DiceRollViewModel(
    userRepository: UserRepository
) : ViewModel() {

    val userUiState: StateFlow<String> =
        userRepository.userStream.map { user -> user.name }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5000),
            initialValue = ""
        )
}
```

É um **Flow** que é diferente de um **StateFlow**

Produz outro **Flow**

Converte um **Flow** para um **StateFlow**

Produzindo o estado

Trabalhando com Stream API e operações únicas

```
class DiceRollViewModel(userRepository: UserRepository) : ViewModel() {
   private val rollState = MutableStateFlow(RollState())

    val uiState: StateFlow<DiceRollUiState> =
        combine(rollState, userRepository.userStream) { roll, user ->
            if (user.name.isEmpty()) { DiceRollUiState.LogUserIn }
            else {
              DiceRollUiState.DiceRoll(user.name, user.numberOfRolls,
              roll.firstDiceValue, roll.secondDiceValue)
            }
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed(5000),
            initialValue = DiceRollUiState.Loading
        )
}
```

escuta ambos os flows e

emite um novo valor sempre que qualquer um deles muda

Produzindo o estado

Convertendo um Flow em um State Flow

💡 O stateIn transforma o Flow resultante em um StateFlow, controlando:

- O escopo (viewModelScope) — quem vai manter o fluxo ativo.
- A política de compartilhamento (SharingStarted) — quando começar e parar de emitir
- O valor inicial (initialValue)

Produzindo o estado

Convertendo um Flow em um State Flow

Política	Fluxo de Dados	Recomendação de Uso
WhileSubscribed(5000)	Quente somente enquanto for observado (com cooldown).	Padrão em ViewModel: Para a maioria dos dados da UI. Otimiza recursos e gerencia bem as transições de tela.
Lazily	Quente a partir do 1º observador e nunca para.	Dados que devem ser coletados durante toda a vida da ViewModel e não precisam iniciar imediatamente.
Eagerly	Quente imediatamente após a criação e nunca para.	Dados que são essenciais e devem ser carregados antes mesmo da UI ser renderizada.

ViewModel

mutableStateOf vs StateFlow

Aspecto	mutableStateOf	StateFlow
Origem	Compose Runtime	Kotlin Coroutines
Uso principal	Estados locais da UI	Estados reativos / em camadas
Reatividade	Automática para Compose	Precisa de coleta (collectAsState())
Visibilidade	Apenas Compose	Pode ser usado em qualquer camada
Complexidade	Simples	Mais robusto
Boas práticas	Use em telas pequenas ou estados isolados	Use em telas com lógica de negócio ou fluxos combinados

Dica prática

👉 Se o estado só existe na tela e é simples, use mutableStateOf.

👉 Se o estado vem do repositório, depende de lógica ou precisa fluir, use StateFlow.

Bibliografia

- 15 Years of Android App Architectures
- Guide to app architecture
- Maintaining ViewModel state the right way in Android
- Jetpack Compose: Alternate to Live Data -StateFlow
- Differences Between collectAsState() and collectAsStateWithLifecycle() in Jetpack Compose

Por hoje é só