Documentação App Minhas Férias

Rafael Leal

21 de janeiro de 2024

Sumário

Parte I

Geral

Capítulo 1

Objetivo

O App foi criado com o propósito de registrar eventos agendados nas férias para organização. Ele pode ser encontrado nesse link: github

Funcionalidades:

1. Registrar eventos com nome, endereço, dia e hora **(Completo)**
2. Registrar companheiros de viagem **(Em andamento)**
3. Registrar controle de gastos e classificar os tipos **(A fazer)**
4. Criar linha do tempo dos eventos **(A fazer)**

Capítulo 2

Arquitetura

Para construção do projeto foram usados os princípios de arquitetura limpa seguindo principalmente as referências [1] e [2].

2.1 SOLID

Também se usam os princípios do acrônimo **SOLID** [3, 4].

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

2.1.1 Single Responsibility Principle

Cada objeto atômico deve ter um único motivo para mudar. Em outras palavras atores diferentes não podem alterar o mesmo objeto para objetivos diferentes. Quando uma classe estiver sendo usada em contextos diferentes e métodos estiverem sujeitos a alterações de stakeholders diferentes então ela deve ser dividida em componentes menores ou ter esses componentes refeitos em outro lugar.

Exemplo: Se uma mesma classe é usada para gerar dados para um componente de view, armazenar informações no banco de dados e tratar inputs dentro de uma lógica de negócio ela tem 3 motivos para mudar. Assim a melhor estratégia seria criar três classes especializadas em cada uma das responsabilidades descritas acima. A mudança em uma delas não vai quebrar a outra tão facilmente.

Benefícios:

- Torna o componente mais reutilizável.
- Simplifica manutenções futuras.

2.1.2 Open-Closed Principle

Cada objeto deve ser fechado para mudança, mas aberto para extensões. Quando classes são modificadas existem uma grande chance de um teste falhar ou algo ter que mudar em várias partes do código que usam essa classe. Assim uma vez feita a classe ela não deve ser mudada diretamente, mas pode ser reaproveitada através de extensão usando a herança ou através de interfaces pela injeção de dependências.

Exemplos:

1. class User(val nome: String, val cpf: String)
   class Admin : User { fun alterarDados() }

   O Código que usa User continua funcionando, mas onde precisa que o usuário seja um Admin as propriedades de usuário continuam disponíveis.

2. class GetUserById @Inject Constructor(userRepository: UserRepository): UseCase { detalhes }

   Se precisarmos alterar a forma de pegarmos o usuário basta alterar a implementação da interface UserRepository sem precisar modificar a classe GetUserById em si.

Benefícios:

- O código fica mais estável.
- Testes continuam passando.
- Código mais limpo e confiável.

2.1.3 Liskov Substitution Principle

Uma classe filha deve poder ser substituída no código sem que algo quebre. Ao implementar a herança devemos nos assegurar que o comportamento da filha é condizente com o que a classe mãe faz.

Exemplo: A classe retângulo possui largura e altura e um método alterar altura. A classe quadrado só tem a propriedade lado, pois são todos iguais e não pode alterar somente a altura. Logo, apesar de todo quadrado ser um retângulo, a classe quadrado não pode estender a classe retângulo.

Benefícios:

- Não são inseridos comportamentos inesperados.

- Testes continuam passando.

2.1.4 Interface Segregation Principle

Cada interface deve conter um mínimo de ações possíveis e relacionadas entre si. Ter vários métodos que nem todos os usuários da interface precisam torna o código mais acoplado gerando compilações desnecessárias e comportamentos inesperados. Podemos mandar um cachorro voar sem querer.

Exemplo:

```
interface mammalsActions{
   falar()
   voar()
}

class Morcego : mammalsActions{
   falar() = detalhes()
   voar() = detalhes()
}

class Cachorro : mammalsActions{
   falar() = detalhes()
   voar() = detalhes()
}
```

Benefícios:

- Cada parte do código fica mais coesa.
- Alterações em parte do código não impacta onde ele não é usado.
- Compilação mais enxuta e rápida.
- Maior reaproveitamento de pequenas ações.

2.1.5 Dependency Inversion Principle

Classes devem depender o mínimo possível de implementações concretas. Devem depender de abstrações e não devem conhecer os detalhes das implementações.

Exemplo: Podemos mudar o local de onde pegamos um dado sem ter que reescrever a classe de repositório

```
class RegisteredEventsRepositoryImpl @Inject constructor(
private val localDataSource: RegisteredEventsLocalDataSource
```

```
) : RegisteredEventsRepository {
   override fun getAllRegisteredEvents(): Flow<List<RegisteredEvent>>
      = localDataSource.getAllRegisteredEvents()
}

interface RegisteredEventsLocalDataSource {
   fun getAllRegisteredEvents(): Flow<List<RegisteredEvent>>
}
```

Benefícios:

- Código fica desacoplado.
- Alterações em métodos não se propagam a todas as classes que o usam.
- Facilidade de criar mocks para testes unitários.
- Núcleo do projeto fica mais estável.

2.2 Padrões de projetos

2.2.1 Adapters

2.3 Módulos

Foram criados os módulos para melhor organizar cada componente de forma mais desacoplada possível facilitando a manutenção e a testabilidade.

Módulos:

- app - Capítulo: 7
- domain - Capítulo: 8
- data-repository - Capítulo: 9
- data-local - Capítulo: 10
- presentation-commom - Capítulo: 11
- presentation-registered-events - Capítulo: 12
- presentation-friends - Capítulo: 13

2.3.1 app

Módulo principal do App que depende de todos os demais.

2.3.2 domain

Esse é o módulo mais estável de todo projeto. Nele é feita a parte de lógica essencial para o funcionamento do aplicativo e portanto não deve sofrer alterações a menos que seja extremamente necessário, seguindo o princípio do **SOLID** Open-Closed Principle 2.1.2 e reforçado pelo Dependency Inversion Principle 2.1.5

2.3.3 data-repository

Esse é o módulo que gerencia o fluxo de dados, onde salvar e de onde pegar os dados.

2.3.4 data-local

Esse é o módulo responsável por armazenar localmente os dados e fazer as devidas conversões de formato de acordo com a maneira como os dados são registrados.

2.3.5 presentation-commom

Esse é o módulo que contem os componentes reutilizáveis em mais de uma camada de apresentação.

2.3.6 presentation-registered-events

Esse é o módulo responsável por gerenciar os eventos registrados através da interação com o usuário.

2.3.7 presentation-friends

Esse é o módulo que registra os companheiros de viagens e pessoas que participam nos eventos.

Capítulo 3

Tarefas em Segundo Plano

Neste capítulo é descrito como as tarefas assíncronas são tratadas no app e suas implicações. É usado como referência o livro base [5] e [6]. Veja também a documentação oficial do Kotlin em Coroutines guide

Este assunto merece destaque uma vez que tarefas muito demoradas podem causar erros e uma experiência ruim para o usuário. Se for executada uma tarefa que demore como um download ou uma consulta complexa no banco de dados na thread principal o app fica parado e dá a sensação ao usuário que não está funcionando mais. Esse é o famoso ANR(Application Not Responding)

3.1 Corrotinas

A corrotina pode ser usada importando as bibliotecas

- org.jetbrains.kotlinx:kotlinx-coroutines-core
- org.jetbrains.kotlinx:kotlinx-coroutines-android
- org.jetbrains.kotlinx:kotlinx-coroutines-test (testes)

As corrotinas tem construtores para diversos propósitos

3.1.1 launch

O launch é um constutor que precisa ser chamado de dentro de um escopo de corrotina. O launch não retorna um valor, mas um objeto Job que representa a corrotina. Ele é ideal para tarefas que ocorrem em segundo plano e você não precisa de um feedback.

Exemplo:

```
scope.launch{
   showLoadingScreen()
}
```

3.1.2 async

O async é também um construtor que precisa ser chamado de dentro de um escopo de corrotina. No entanto ele retorna um objeto Deferred. Ela é interessante de usar quando você precisa de um valor antes de começar a próxima tarefa. O valor pode ser lido usando a função await.

Exemplo:

```
val job = scope.async{
   getData ()
}
val data = job.await()
```

3.1.3 runBlocking

Diferente dos demais, o runBlocking cria uma nova corrotina e bloqueia a thread até que a tarefa tenha sido executada. Ela é muito útil em testes unitários onde precisamos esperar a tarefa ser executada para que possamos verificar os resultados.

Exemplo:

```
@ExperimentalCoroutinesApi
@Test
fun testExecuteSuccess() = runBlocking() {
   val result = useCase.execute(request).first()
   assertEquals(Result.Success(response), result)
}
```

3.2 Escopo

As corrotinas tem um escopo para trabalhar. Esse escopo define como os dados são tratados assim como os erros e o ciclo de vida de uma tarefa em segundo plano. Ao cancelar um escopo você cancela os escopos criados por ele.

1. lifecycleScope:

   Esse é um escopo usado geralmente dentro de Activity e Fragment. Ele é cancelado quando o componente visual associado a ele é destruído. Isso é importante, pois evita que uma tarefa desnecessária continue além de evitar vazamento de memória. Ela pode ser usada importando a biblioteca androidx.lifecycle

   Ela também pode ser usada associada ao ciclo de vida da Activity através dos construtores:

- launchWhenCreated
- launchWhenStarted
- launchWhenResumed

2. viewModelScope:

   Esse é o escopo associado ao ViewModel onde as tarefas mais longas e complexas se concentram. Também é cancelado quando o viewModel associado é destruído.

3. Existem outros escopos, mas não irei entrar em detalhes aqui no momento.

3.3 Despachantes

Os despachantes vão definir em qual thread o trabalho será executado. Exemplos de despachantes são:

1. Na thread principal: Dispatchers.Main

   Geralmente usado para fazer alterações na interface de usuário.

2. Na thread de dados: Dispatchers.IO

   Geralmente usado para transportar dados entre bancos de dados locais e remotos.

3. Na thread de alto processamento: Dispatchers.Default

   Geralmente usado para processamentos pesados e complexos

É possível mudar de thread através do método withContext facilitando o gerenciamento dos despachantes.

Exemplo:

```
suspend fun getData(): List<DataModel> {
   withContext(Dispatchers.IO) { ... }
}
```

3.4 Funções suspensas

São funções marcadas que só podem ser chamadas de dentro de um escopo de corrotina ou de outra função suspensa. Isso auxilia na codificação para não chamar funções que podem travar a tela em lugares sem o devido escopo. Essas funções são precedidas da palavra suspend na definição do corpo.

Exemplo:

```
suspend fun getData(){
...
}
```

3.5 Livedata

3.6 Flow

Flow é uma biblioteca de fluxo de dados assíncronos construído sobre a corrotina do kotlin.

A grande vantagem do Flow é que ele pode emitir múltiplos valores ao longo do tempo ao invés de somente um. Por exemplo, se você está mostrando uma lista de itens do seu banco de dados e você adiciona um item novo, esse item será emitido sem precisar carregar todos os demais. Abiblioteca Room-ktx usa o flow em suas queries. Isso aumenta performance e torna o código mais simples.

3.7 Testes

Capítulo 4

Detalhes de Interface

Aqui detalhamos as escolhas relacionadas a programação de interface de usuário. Existem pelo menos duas maneiras de se enxergar a programação de Views: Paradigma imperativo e declarativo [7] [8]

Veja também: Descomplicando: programação imperativa, declarativa e reativa

4.1 Paradigma Imperativo

4.2 Paradigma Declarativo

4.3 Elevação de Estado de UI

Capítulo 5

Grafos de Navegação

Neste capítulo registramos os diferentes fluxos de navegação entre telas. ainda não conheço referências bibliográficas avançadas sobre Compose e navegação.

Referências no YouTube:

- Stevdza-San Navigation in Jetpack Compose
- Programador de Elite [NAVGRAPH COMPOSE] COMO CRIAR ROTAS DE NAVEGAÇÃO COMPLEXAS EM JETPACK COMPOSE ANDROID
- Philipp Lackner Full Guide to Nested Navigation Graphs in Jetpack Compose

Capítulo 6

Bibliotecas Externas

Para melhorar a performance são usadas algumas bibliotecas externas.

Elas são importadas no arquivo build.gradle do projeto e usadas como referência para cada módulo. Dessa forma não precisamos atualizar as versões em cada módulo manualmente.

6.1 Injeção

Para injeção de dependência usamos

- Hilt.

6.2 Caixas de Diálogos

Para mostrar caixas de diálogo usando o Jetpack Compose foi usada a biblioteca sheets-compose-dialogs.

São usadas as caixas de diálogos para verificar ações com botões de sim e não além de selecionar datas e horas.

- Caixas comuns de diálogo: **sheets-compose-dialogs:core**
- Caixas de diálogo de hora: **sheets-compose-dialogs:clock**
- Caixas de diálogo de data: **sheets-compose-dialogs:calendar**

6.3 Corrotinas

Essas bibliotecas são usadas para gerenciar tarefas em segundo plano.

- org.jetbrains.kotlinx:kotlinx-coroutines-core
- org.jetbrains.kotlinx:kotlinx-coroutines-android
- org.jetbrains.kotlinx:kotlinx-coroutines-test (testes)

Parte II

Módulos

Capítulo 7

app

7.1 Injeções:

Nesse módulo reside a classe MyApplication: Application() necessária para usar a injeção de dependência com Hilt 6.1.

Contem a injeção do UseCase.Configuration usado para definir o Dispatcher do UseCase.

Essa injeção é necessária, pois facilita controlar o contexto de corrotinas nos testes de unidade

7.2 Testes:

Contem os seguintes testes:

- de interface
- migração do banco de dados

Capítulo 8

domain

8.1 Classes seladas

As classes seladas são consideradas nesse módulo para que não possam ser estendidas em outro módulos podendo causar um erro inesperado. Além disso são importantes para que nos, devidos cenários, contenha, informação desejada de acordo com o método invocado.

8.1.1 UseCaseException

São registradas aqui os tipos específicos de exceções para que possamos dar o devido tratamento do erro trazendo um retorno ao usuário mais transparente.

8.1.2 Result

Nessa classe retornamos um sucesso com os dados que foram exigidos podendo ser de qualquer tipo ou um erro com a exceção que a causou.

8.2 UseCase

É uma classe abstrata usada para construir cada caso de uso do app. Nela é tratado o caso de exceção para retornar UseCaseException em caso de haver algum erro evitando assim **ANR**. Ela ocorre dentro do escopo de corrotina de input-output: **Dispatchers.IO** [5], dessa forma nunca teremos o problema de travar a Main Thread mais uma vez evitando **ANR**. O UseCase.Configuration é injetado no módulo app 7.1. Outro motivo para que se considere o Dispatcher dessa forma é que facilita o controle dentro de cada teste de caso de uso uma vez que os testes com trabalho fora da Main Thread precisam esperar que elas terminem para que possa fazer as devidas asserções. Para finalizar devemos implementar o tipo de dado de entrada e de saída através das classes **Request** e **Response**.

O retorno de um caso de sucesso será um objeto do tipo Result.Sucess(obj: T) em caso de sucesso ou um Result.Error(obj: UseCaseException) em caso de falha.

8.2.1 RegisteredEvents

CRUD básico para a tabela RegisteredEvents responsável por armazenar os eventos do usuário.

8.3 Adapters

Esse é um padrão de projeto bem utilizado e citado em [9]

Aqui são criados alguns adaptadores para desacoplar a lógica de transformação de dados baseado no princípios de substituição de Liskov 2.1.3 e de inversão de dependência 2.1.5.

8.3.1 DateTimeAdapter

Usado para converter datas e horas em Long e vice-versa para melhor armazenar no banco de dados, comparar datas e mostrar o formato padronizado ao usuário.

8.4 repositories

São criadas as interfaces usadas pelos UseCases para que se siga o princípio aberto fechado 2.1.2, o princípio de inversão de dependência 2.1.5 e não tenhamos que alterar cada caso de uso na eventualidade de mudarmos a fonte de dados. Essa estratégia ajuda a tornar o módulo Domain mais estável.

8.5 Testes

São realizados testes unitários de cada UseCase e Adapter criados.

Capítulo 9

data-repository

Capítulo 10

data-local

Capítulo 11

presentation-commom

Capítulo 12

presentation-registered-events

Nesse módulo se concentram os componentes de interface relacionados ao registro de eventos.

12.1 Pegar a lista de eventos registrados ao abrir o app

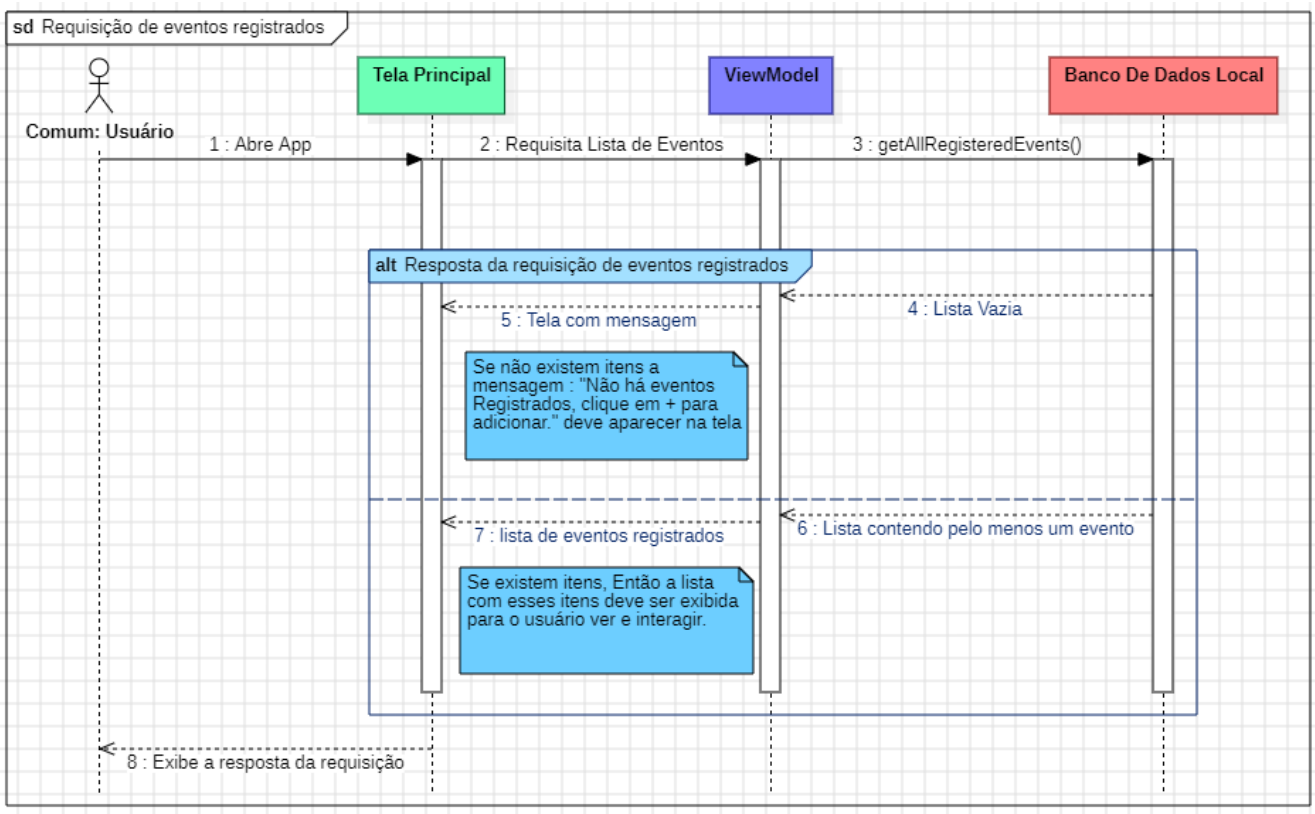
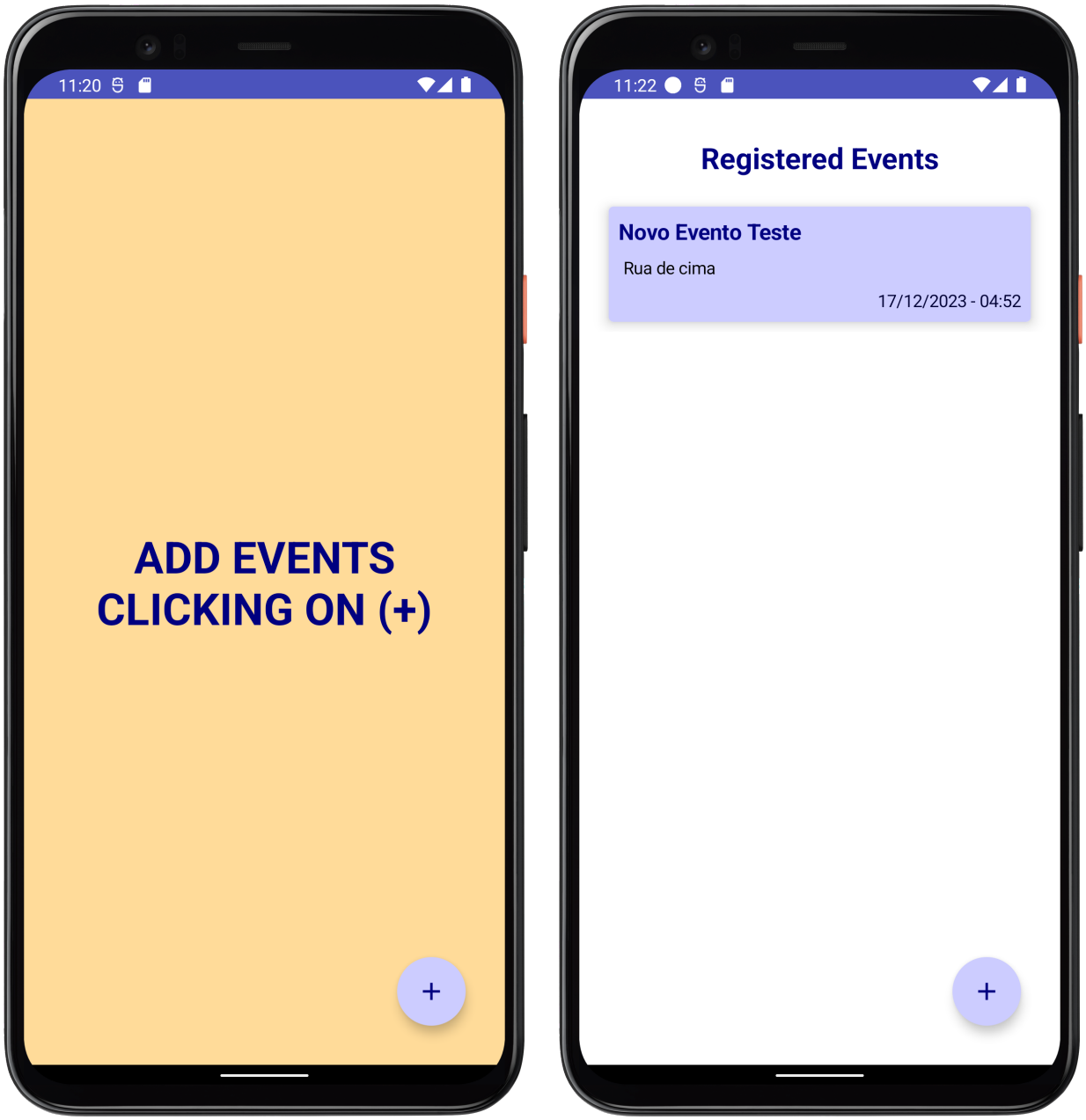


Figura 12.1: Pegar a lista de eventos



(a) Lista vazia

(b) Lista não vazia

Figura 12.2: Tela inicial

12.2 Atualizar item da lista de eventos registrados

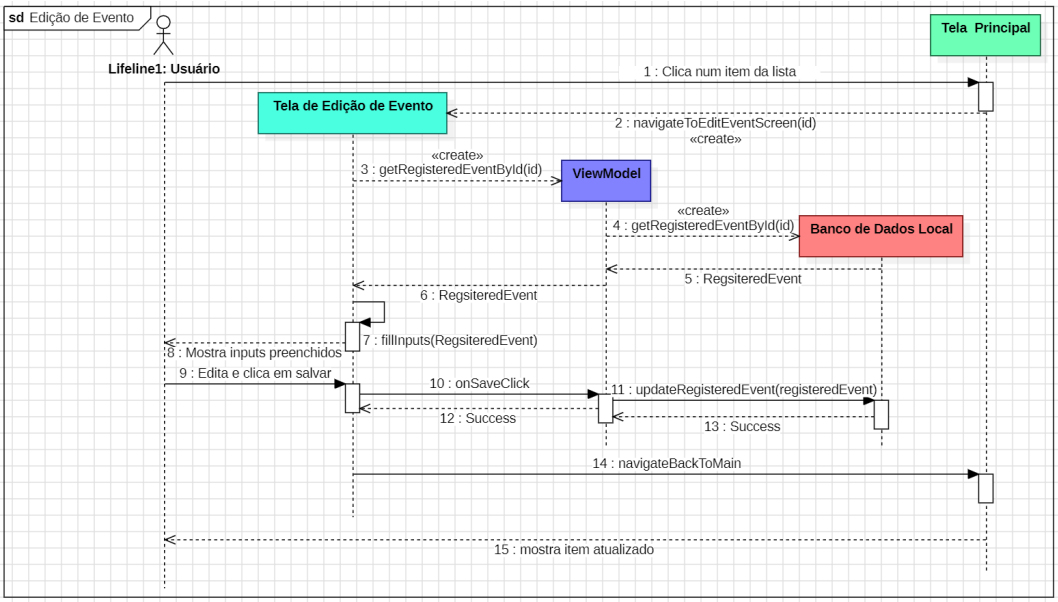
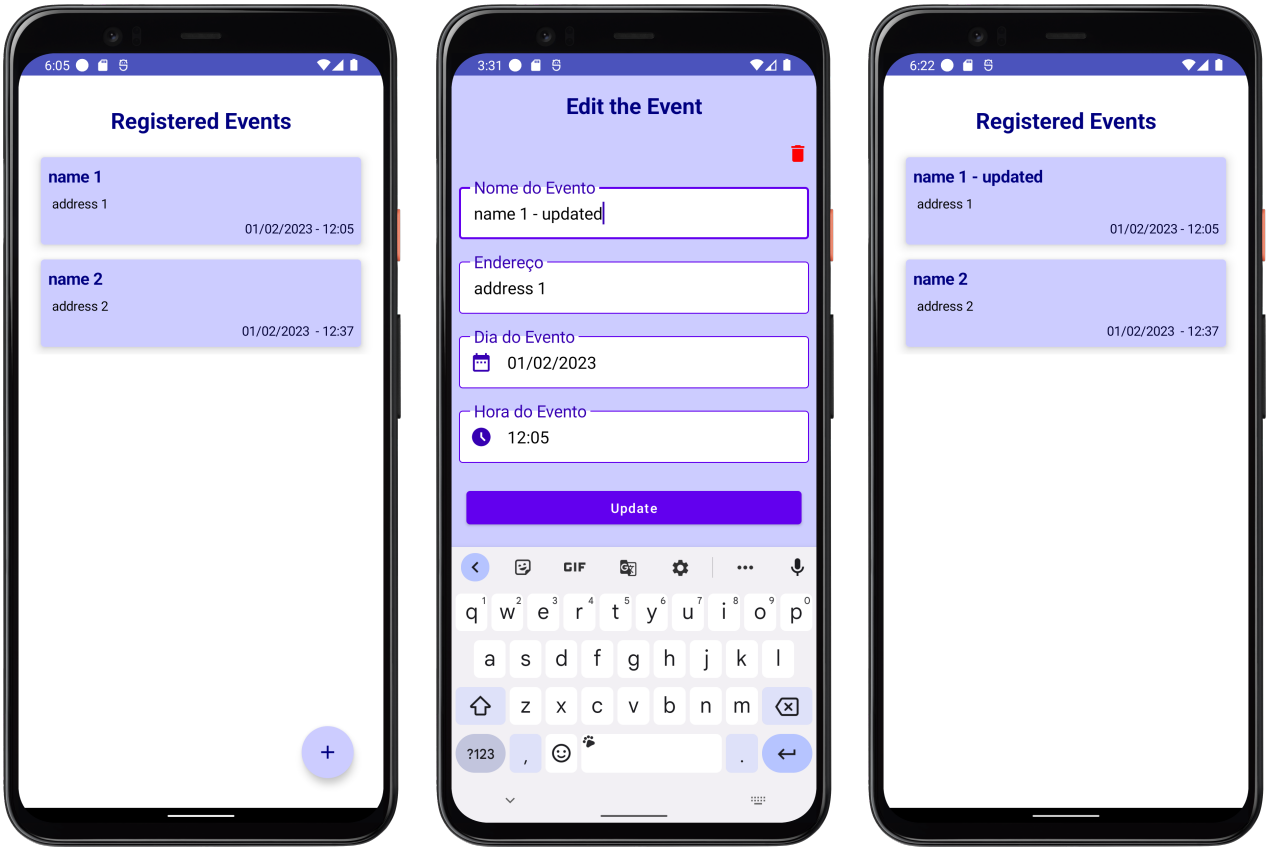


Figura 12.3: Atualizar item da lista de ventos registrados



(a) Lista inicial (b) Tela de edição (c) Lista final

Figura 12.4: Processo de atualização

12.3 Remover item da lista de eventos registrados

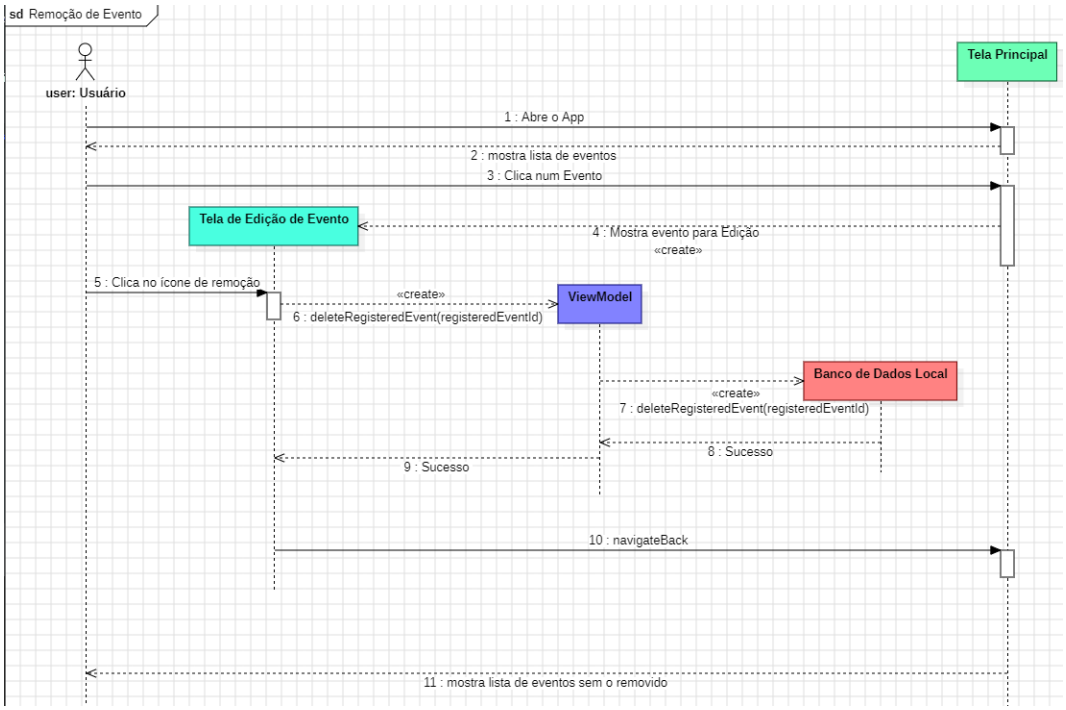
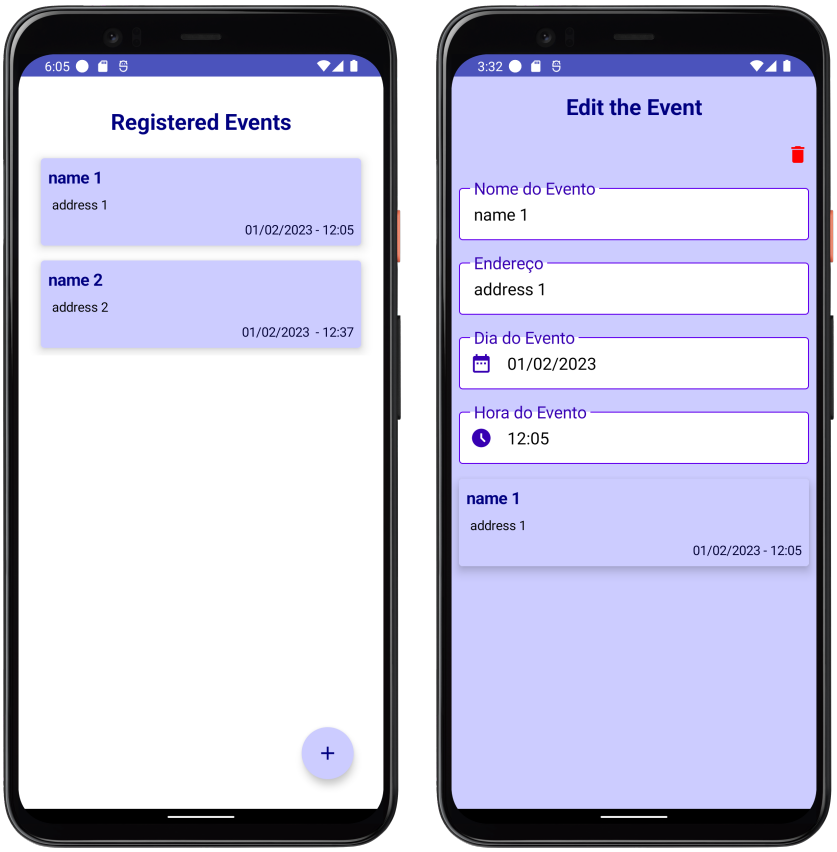
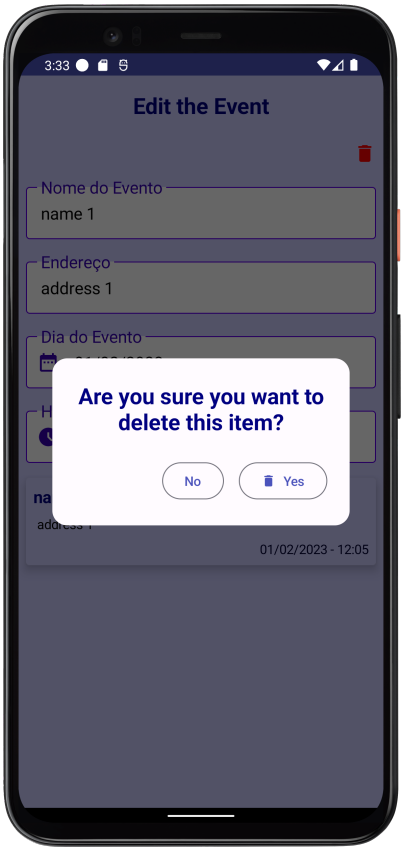


Figura 12.5: Remove item da lista de ventos registrados



(a) Lista inicial

(b) Tela de edição



(c) Diálogo de confirmação

(d) Lista final

Figura 12.6: Processo de remoção de item

Capítulo 13

presentation-friends

Referências Bibliográficas

[1] Alexandru Dumbravan and Ed Price. *Clean Android Architecture: Take a layered approach to writing clean, testable, and decoupled Android applications.* Packt Publishing Ltd, 2022.

[2] Eran Boudjnah. *Clean Architecture for Android: Implement Expert-led Design Patterns to Build Scalable, Maintainable, and Testable Android Apps (English Edition).* BPB Publications, 2022.

[3] C Robert. Clean architecture: A craftsman's guide to software structure and design (robert c. martin series), 2019.

[4] A. Mellor. *Test-Driven Development with Java: Create higher-quality software by writing tests first with SOLID and hexagonal architecture.* Packt Publishing, 2023.

[5] Jomar Tigcal. *Simplifying Android Development with Coroutines and Flows: Learn how to Use Kotlin Coroutines and the Flow API to Handle Data Streams Asynchronously in Your Android App.* Packt Publishing Limited, 2022.

[6] N. Smyth. *Jetpack Compose 1.4 Essentials: Developing Android Apps with Jetpack Compose 1.4, Android Studio, and Kotlin.* Payload Publishing, 2023.

[7] C. Ghita. *Kickstart Modern Android Development with Jetpack and Kotlin: Enhance Your Android Development Skills to Build Reliable Modern Apps.* Packt Publishing, Limited, 2022.

[8] Kodeco Team, D. Buketa, and P. Prasad. *Jetpack Compose by Tutorials (Second Edition): Building Beautiful UI with Jetpack Compose.* Kodeco Incorporated, 2023.

[9] Richard Helm, Ralph E Johnson, Erich Gamma, and John Vlissides. *Design patterns: Elements of reusable object-oriented software.* Braille Jymico Incorporated Quebec, 2000.