

**UNIVERSIDADE FEDERAL DA PARAÍBA  
CENTRO DE INFORMÁTICA  
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**COMPONENTES DE ARQUITETURA ANDROID E MICRO FRONTENDS:  
DESCRIÇÃO, IMPLEMENTAÇÃO E COMPONENTIZAÇÃO DE APLICATIVOS**

**EDNALDO MARTINS DA SILVA**

**João Pessoa  
2020**

EDNALDO MARTINS DA SILVA

**COMPONENTES DE ARQUITETURA ANDROID E  
MICRO FRONTENDS: DESCRIÇÃO, IMPLEMENTAÇÃO  
E COMPONENTIZAÇÃO DE APLICATIVOS**

Monografia apresentada ao curso de graduação em Ciência da Computação, do Centro de Informática da Universidade Federal da Paraíba, como requisito para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Raoni Kulesza

**João Pessoa**

**2020**

## RESUMO

O sistema operacional Android, foi lançado em outubro de 2008, e tornou-se posteriormente sistema mais difundido entre os smartphones, todavia, os desenvolvedores de aplicativos para o Android, ao longo de anos, não encontraram um padrão de arquitetura ideal para a plataforma, e que seja fortemente recomendado para o desenvolvimentos das aplicações. Padrões de arquitetura como *Model-View-Presenter* (MVP), ou *Model-View-ViewModel* (MVVM), sempre foram questionáveis, simplesmente por dificultarem a manutenção, ou geração de testes unitários para o aplicativo. Nesta pesquisa serão apresentados alguns componentes de arquitetura Android que foram anunciados na Google I/O 2017, os quais, são capazes de tornar a arquitetura dos aplicativos componentizada. Para demonstrar a capacidade desses componentes, será desenvolvida um aplicativo, e além disso, serão utilizadas técnicas de *Micro Frontends*, com o objetivo de construir uma aplicação android, componentizado, testável, e melhor de manter.

**Palavras-chaves:** Android. Componentes. Arquitetura. Aplicativo. *Micro Frontends*.

## ABSTRACT

The Android operating system, esse released in October 2008, and later became the most widespread system among smartphones, however, the android application developers, over the years, have not been able to use an ideal architecture patterns for the platform, and they are strongly recommended for application development. Android architecture patterns like Model-View-Presenter (MVP), or Model-View-ViewModel (MVVM), have always been questionable, simply because they make it difficult the maintenance or generate unit tests for the application. This research will present some Android architecture components that were announced at Google I / O 2017, which are capable of making application architecture componentized. To demonstrate the capability of these components, an application will be developed, besides that, Micro Frontends techniques will be used, in order to build an android application, componentized, testable, and maintainable.

**Key-words:** Android. Components. architecture. App. Micro Frontends.

## LISTA DE ILUSTRAÇÕES

Figura 1: Estados do ciclo de vida da atividade e retornos de chamada	19
Figura 2: Diagrama dos módulos dos componentes de arquitetura	24
Figura 3: Vida útil da ViewModel durante execução de uma Activity	26
Figura 4: Implementação independente usando Micro frontends	32
Figura 5: Arquitetura do aplicativo Libflix	39
Figura 6: Representação da camada Presentation	40
Figura 7: Representação da camada Domain	41
Figura 8: Representação da camada Data	42
Figura 9: Splash screen	44
Figura 10: Código - Lista do tipo LiveData recebe os dados e notifica controlador	46
Figura 11: Requisição e carregamento das informações dos filmes com sucesso	46
Figura 12: Requisição e retorno de erro após a tentativa de comunicação com o servidor	47
Figura 13: Erro e sucesso após duas tentativas de comunicação com o servidor	48
Figura 14: Código - Atualizando página da lista de filmes	49
Figura 15: Código - Verificando se o filme está na base de dados local	49
Figura 16: Código - Objeto do tipo LiveData recebe os dados e notifica observador	50
Figura 17: Selecionando filme e requisitando dados na base de dados remota	50
Figura 18: Selecionando filme e abrindo o filme a partir da base de dados local	51
Figura 19: Código - Verificar se é preciso deletar ou salvar filme na base de dados local	53
Figura 20: Marcando um filme como favorito	53
Figura 21: Marcando filme como favorito, e fechando aplicativo posteriormente	54
Figura 22: Buscando por filmes na base de dados remota	56
Figura 23: Código - Realizando busca por filmes na API	57
Figura 24: Realizando uma busca na base de dados local	57
Figura 25: Código - Realizando busca e aplicando nova lista na lista de apresentação	58
Figura 26: Componentização da Activity	59
Figura 27: fragment_film_list.xml usado pelos Fragments que exibem a lista de filmes	60
Figura 28: activty_main.xml Activity principal contém um toolbar.xml em seu front-end	60
Figura 29: Código - Ferramenta de testes e componentes testados	61

Figura 30: Código - Funções chamadas antes de iniciar e ao finalizar os testes.	61
Figura 31: Código - Teste do insert e get	62
Figura 32: Código - Teste do clear	62
Figura 33: Código - Teste do update	62

## **LISTA DE ABREVIATURAS E SIGLAS**

API	Application Programming Interface
OS	Operating System
DAO	Data Access Object
JSON	JavaScript Object Notation
MVC	Model-View-Controller
MVP	Model-View-Presenter
MVVM	Model-View-ViewModel
RF	Requisitos Funcionais
RNF	Requisitos Não Funcionais
TMDB	The Movie Database
URI	Uniform Resource Identifiers
VM	Virtual Machine
XML	Extensible Markup Language

## SUMÁRIO

<b>1 INTRODUÇÃO</b>	<b>14</b>
1.1 DEFINIÇÃO DO PROBLEMA	14
1.2 OBJETIVOS	15
1.2.1 Objetivo Geral	15
1.2.2 Objetivo Específico	15
1.3 MOTIVAÇÃO	15
1.4 GRUPO ALVO	16
1.5 ESTRUTURA DO TRABALHO	16
<b>2 CONCEITOS GERAIS</b>	<b>17</b>
2.1 ARQUITETURA ANDROID	17
2.2 COMPONENTES	18
2.2.1 Activity	18
2.2.1.1 Ciclo de vida	19
2.2.1.2 Comunicação	21
2.2.2 Services	21
2.2.3 Broadcast Receivers	22
2.2.4 Content Providers	22
2.3 COMPONENTES DE ARQUITETURA ANDROID	23
2.3.1 Visão Geral	23
2.3.1.1 ViewModel	25
2.3.1.2 LiveData	27
2.3.1.3 Room	28
2.3.2 Discussão	29
2.3.2.1 Vantagens	29
2.3.2.2 Desvantagens	30
2.3.2.3 Melhorias	30
2.4 MICRO FRONTENDS	31
2.4.1 Atualização do front-end	32
2.4.2 Independência entre os front-ends	32
2.4.3 Implantação do front-end	33
2.4.4 Autonomia das equipes	33
<b>3 O APLICATIVO LIBFLIX</b>	<b>34</b>
3.1 VISÃO GERAL	34
3.2 LISTA DE REQUISITOS	34
3.2.1 Requisitos Funcionais	34
3.2.1.1 [RF01] Solicitar Lista de Destaques	34
3.2.1.2 [RF02] Apresentação da Lista de Filmes da Base de Dados Remota	35

3.2.1.3 [RF03] Apresentação da Lista de Filmes da Base de Dados Local	35
3.2.1.4 [RF04] Apresentação do Cartões do Filme	35
3.2.1.5 [RF05] Mudança de Página	35
3.2.1.6 [RF06] Pesquisa na Base de Dados Remota	35
3.2.1.7 [RF07] Pesquisa no Banco de Dados Local	35
3.2.1.8 [RF08] Chamar Tela de Detalhes do Filme	36
3.2.1.9 [RF09] Adicionar Filmes aos Favoritos	36
3.2.1.10 [RF10] Remover Filmes dos Favoritos	36
3.2.1.11 [RF11] Abrir Homepage do Filme	36
3.2.2 Requisitos Não Funcionais	36
3.2.2.1 [RNF01] Retornar Lista Vazia	36
3.2.2.2 [RNF02] Apresentação dos Botões de Mudança de Página	37
3.2.2.3 [RNF03] Buscar Filme nas Bases de Dados	37
3.2.2.4 [RNF04] Exibir Detalhes do Filme	37
3.2.2.5 [RNF05] Conexão Com Internet	37
3.2.2.6 [RNF06] Tentativas de Requisição	37
3.2.2.7 [RNF07] Comunicação com a API do TMDB	38
3.2.2.8 [RNF08] Status do Carregamento da Lista	38
3.2.2.9 [RNF09] Compatibilidade do Aplicativo	38
3.3 ARQUITETURA DO APLICATIVO	38
3.3.1 As camadas da arquitetura	38
3.3.2 Presentation	40
3.3.3 Domain	41
3.3.4 Data	42
<b>4 APPLICABILIDADE DOS COMPONENTES DE ARQUITETURA</b>	<b>44</b>
4.1 OS COMPONENTES GRÁFICOS E A INTERFACE DE USUÁRIO	44
4.1.1 Splash Screen	44
4.1.2 Ação dos Componentes Arquiteturais durante a requisição da lista de filmes	45
4.1.2.1 Carregando a lista de destaque	46
4.1.2.2 Problemas com o carregamento dos filmes	47
4.1.2.3 Nova tentativa de requisição ou atualização dos dados	48
4.1.3 Ação dos Componentes Arquiteturais ao selecionar um filme	49
4.1.3.1 Abrindo o Filme a partir da base de dados remota	50
4.1.3.2 Abrindo o Filme a partir da base de dados local	51
4.1.4 Ação dos Componentes Arquiteturais ao salvar um filme localmente	52
4.1.4.1 Salvando normalmente	53
4.1.4.2 Salvando em situações inesperadas	54
4.1.5 Ação dos Componentes Arquiteturais ao buscar filmes	55
4.1.5.1 Buscar filme na base de dados remota	56
4.1.5.2 Buscar filme na base de dados local	57

4.2 APLICANDO MICRO FRONTENDS	58
4.3 TESTE DA APLICAÇÃO	60
<b>5 CONCLUSÃO E TRABALHOS FUTUROS</b>	<b>64</b>
<b>REFERÊNCIAS</b>	<b>65</b>

## 1 INTRODUÇÃO

Desde o momento em que os dispositivos móveis começaram a serem lançados, o acesso a esses aparelhos cresceu de forma rápida, principalmente quando se trata de smartphones com o sistema operacional Android. O sistema operacional Android, lançado em outubro de 2008 [1], tem evoluído a cada ano e vem sendo moldado para melhor atender aos usuários e desenvolvedores. Novos componentes para construção dos aplicativos estão sendo testados e oficializados, capazes de facilitar o desenvolvimento de aplicativos e torná-los robusto [2].

Os novos componentes apresentados são recomendados idealmente em futuros aplicativos desenvolvidos, permitindo que uma arquitetura planejada seja criada levando em conta a capacidade desses componentes e a forma como se comportam [2].

### 1.1 DEFINIÇÃO DO PROBLEMA

Fato é que a maior parte das arquiteturas usadas para desenvolver aplicações móveis para dispositivos Android não vêm tendo os melhores resultados e experiência, e acabam apresentando dificuldades para, testar e modular a aplicação. A arquitetura MVC (*Model-View-Controller*) e suas derivações, não foram criadas pensando em uma solução para construir aplicações Android. Esses padrões arquiteturais acabaram por tornar-se os modelos mais utilizados para desenvolver aplicativos Android, e isso tem sido um problema na hora de desenvolver, corrigir erros, e manter esses aplicativos atualizados [5].

Para evitar e reduzir a quantidade de problemas advindos da arquitetura do aplicativo, é sugerido seguir um novo modelo de arquitetura sugerido pelo *Android Developers*, que recomenda fortemente usar os novos componentes de arquitetura lançados, portanto, irá facilitar a criação desses aplicativos, capazes de torná-los reativo e responsivos na sua apresentação, funcional, e seguro em relação ao armazenamento dos dados [2].

## 1.2 OBJETIVOS

### 1.2.1 Objetivo Geral

O aspecto geral deste trabalho é apresentar um modelo de arquitetura idealizado para criação de aplicações Android, baseado em componentes, a qual, possui em sua estrutura os novos componentes de arquitetura Android.

### 1.2.2 Objetivo Específico

Em relação aos aspectos mais específicos deste trabalho, é objetivado:

- Apresentar e descrever os novos componentes de arquitetura Android recomendados pelo *Android Developers*.
  - ViewModel.
  - Lifecycle
  - LiveData.
  - Room.
- Aplicar os conceitos descritos na elaboração de um aplicativo baseado na nova arquitetura recomendada.
  - Identificar e especificar os requisitos funcionais e não funcionais.
  - Apresentar o projeto de arquitetura de alto nível.
- Criar um aplicativo com base no modelo de arquitetura elaborado.
  - Apresentar resultados da interação entre os componentes arquiteturais e interface do usuário.
  - Utilizar técnicas de *Micro Frontends* para aprimorar a componentização da arquitetura elaborada.

## 1.3 MOTIVAÇÃO

Um fator importante para esta pesquisa, é usar um novo modelo de arquitetura para aplicativos Android, que possa prover soluções para problemas existentes em outras arquiteturas que não foram bem adaptadas, e que ainda são utilizadas atualmente, devido a inexistência de modelos conhecidos e difundidos na comunidade de desenvolvedores Android.

## 1.4 GRUPO ALVO

Essa pesquisa é destinada aos desenvolvedores de aplicações Android, principalmente aqueles que estão ingressando nesse mundo, ou buscando novas tecnologias. A pesquisa descreve os benefícios em usar a nova arquitetura recomendada e seus componentes arquiteturais, como também apresenta os resultados obtidos.

## 1.5 ESTRUTURA DO TRABALHO

Este trabalho está dividido em 4 partes. A primeira parte é a sessão atual, o qual é especificado o escopo do trabalho de maneira introdutória, para ter conhecimento do há de ser apresentado neste trabalho. Na seção seguinte, os principais conceitos sobre a arquitetura Android, e componentes arquiteturais serão descritos, além de uma breve explanação sobre *Micro Frontends*. Na seção 3, uma aplicativo será elaborado e especificado, baseada na arquitetura de componentes Android. A seção 4 apresenta os conceitos da seção 2 de forma prática na aplicação descrita na seção 3. Nela o aplicativo construído é demonstrado usando os componentes arquiteturais.

## 2 CONCEITOS GERAIS

Esta seção descreve as características de uma aplicação Android, como essa aplicação é estruturada dentro do próprio sistema Android, e descrever como as partes da arquitetura compõem o sistema e as aplicações.

### 2.1 ARQUITETURA ANDROID

A plataforma Android é um sistema Linux multiusuário, onde os diversos componentes e serviços da plataforma são desenvolvidos em C e C++, e um conjunto de outros recursos estão disponibilizados por APIs programadas utilizando a linguagem de programação Java. A plataforma está dividida em quatro camadas principais: Kernel do Linux, Bibliotecas nativas, Estrutura da Java API, e Aplicativos do sistema [3].

Como foi dito, a plataforma Android é um sistema Linux multiusuário. Cada aplicativo é considerado um usuário diferente, onde cada um desses aplicativos são executados dentro de uma máquina virtual (ou simplesmente VM) exclusiva, de modo que cada um destes recebam um código individual de usuário do Linux. O Android inicia o processo quando se faz necessário executar algum componente do aplicativo, encerrando-o de forma automática quando o sistema precisa abrir espaço na memória para outros aplicativos, ou quando é dispensado pelo usuário do dispositivo [4].

De forma individual o aplicativo é ativado dentro da sandbox de segurança. A sandbox é usada para executar aplicativos não testados vindos de fontes desconhecidas ou não confiáveis, ou seja, de fora da loja de aplicativo android da google (Play Store). Durante essa execução, o acesso a recursos do dispositivo são limitados, logo a segurança torna-se um pouco mais reforçada contra ameaças que possam danificar o dispositivo [10].

Entretanto é possível que dois aplicativos compartilhem o mesmo código de usuário Linux, sendo assim, um pode ter permissão para acessar arquivos de um outro, podendo compartilhar também a mesma VM. A plataforma Android implementa o princípio de privilégio mínimo. Neste caso, o sistema operacional (OS) Android dá permissão aos aplicativos, apenas para acessar os componentes necessários para o funcionamento correto da aplicação. O aplicativo não pode acessar partes do sistema o qual não lhe é permitido [4].

## 2.2 COMPONENTES

De modo geral as aplicações Android têm uma estrutura complexa e baseada em componentes. Levando em conta que as aplicações desenvolvidas para computadores geralmente são executados como um único processo monolítico, podemos enxergar de maneira clara (principalmente no uso prático), que aplicações para dispositivos Android possuem diversos componentes. Dentre estes temos, *activities*, *fragments*, *services*, *content providers*, e *broadcast receivers* [2].

A alternância de aplicativos é comum no sistema Android [2], sendo assim, é visível a componentização dos aplicativos no uso prático do sistema. Um exemplo simples de entender seria, um aplicativo de rede social genérico qualquer, que dispõe a opção de realizar uma postagem com foto em sua rede social, e essa função do aplicativo aciona o aplicativo de câmera padrão do sistema, e após retornar do aplicativo de câmera o usuário tem disponível a foto tirada, podendo postá-la em sua rede social (levando em conta que o usuário tirou uma foto).

Os componentes são como os blocos de construção para o aplicativo Android, onde cada componente é um mecanismo de interação entre o aplicativo e o usuário, ou o sistema [4]. Como dito antes, há quatro tipos de componentes: *Activity* (e *Fragment*), Services, Content Providers, e Broadcast receivers; e cada um deles têm uma finalidade diferente.

### 2.2.1 Activity

Uma *Activity* é um mecanismo de entrada criado para interagir com o usuário [4]. De forma simples e direta uma *Activity* é uma tela única com interface de usuário para apresentação de informações e (ou) entrada de dados.

As *Activities* são armazenadas em uma pilha de *Activities*, e geralmente a *Activity* em execução é a que está no topo desta pilha, pois pode ocorrer de uma ou mais *Activities* estarem em execução, e visíveis na tela. Quando uma *Activity* é iniciada, ela é alocada no topo da pilha de *Activities*, e a *Activity* que antes estava no topo da pilha, passará a ficar abaixo desta nova *Activity*. Ainda é possível que mais de uma *Activity* esteja visível na tela, mas apenas a que está em execução permanece no topo [11].

### 2.2.1.1 Ciclo de vida

Para compreender melhor essa pilha de *Activities*, deve-se compreender que uma *Activity* possui estados, e como esses estados definem a forma como a aplicação está se comportando no sistema. A imagem abaixo mostra o ciclo e estado das *Activities*. Os retângulos com as laterais curvadas representam os estados principais de uma *Activity*, e as setas com nomes entre os estados, representam os métodos de chamados para a transição de estados:

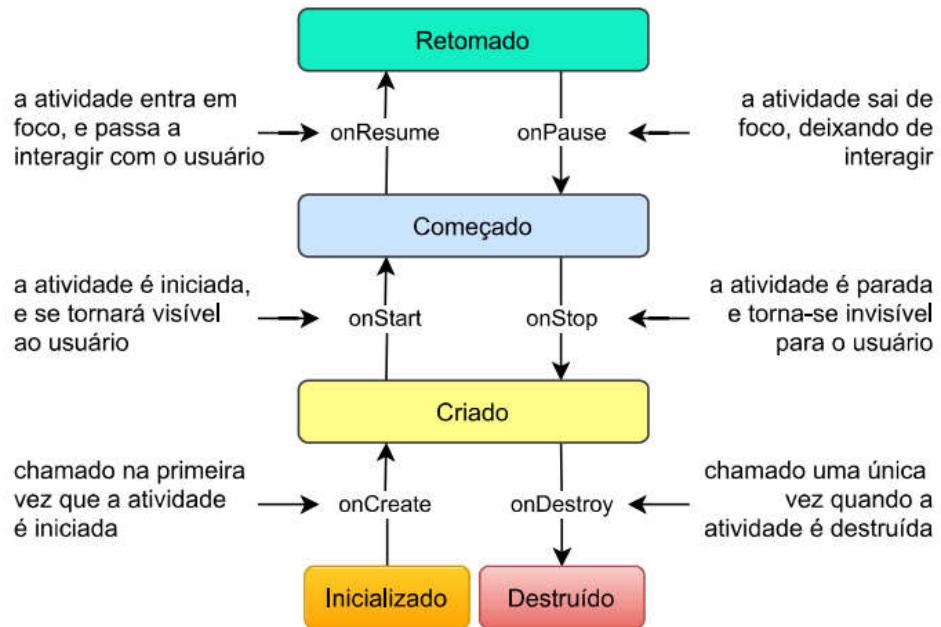


Figura 1: Estados do ciclo de vida da atividade e retornos de chamada<sup>1</sup>

Ao inicializar uma *Activity*, o método `onCreate` é chamado para que ela possa ser criada, e entrar no estado criado, logo, nesse estado a *Activity* está criada. Nesse estado a *Activity* ainda não está visível e nem em foco, portanto não pode interagir com o usuário. Para a *Activity* se tornar visível o método `onStart` é executado, e em seguida o método `onResume` coloca-a em foco, permitindo que o usuário interaja com ela. Após iniciada, ela é colocada no topo da pilha. Após colocada no topo da pilha, a *Activity* entra em execução e permanece em

<sup>1</sup> Disponível em: [https://video.udacity-data.com/topher/2018/November/5be286d0\\_14-1803sc-a-share-dialog-and-onpause-onresume-border/l4-1803sc-a-share-dialog-and-onpause-onresume-border.png](https://video.udacity-data.com/topher/2018/November/5be286d0_14-1803sc-a-share-dialog-and-onpause-onresume-border/l4-1803sc-a-share-dialog-and-onpause-onresume-border.png). Acesso em janeiro de 2020.

foco até que uma nova forma de interação seja chamada. Também é possível que uma *Activity* transparente ou menor que o tamanho real da tela do dispositivo esteja em execução, mesmo que outras *Activities* estejam também visíveis na tela. Uma *Activity* é considerada visível caso ela esteja visível para o usuário na tela do dispositivo, ou esteja sobreposta por outra *Activity* ou forma de interação, e não esteja no topo da pilha. Nesse modo, a *Activity* está criada, mas pode ser parada, tornando-se um processo alvo a ser destruído a qualquer instante pelo sistema, devido a falta de memória. Se a *Activity* for oculta ou parada pelo método `onStop`, ela também pode ser destruída por ter encerrado as suas tarefas [11].

Um *Fragment* possui o comportamento similar a uma *Activity* para o usuário, mas de forma modularizada, pois um *Fragment* pode dividir a tela com outros *Fragments*, como também o mesmo espaço da tela de forma alternada durante uma execução. Além disso um *Fragment* fica contido em uma *Activity* e tem seu ciclo de vida dependente dessa *Activity*, todavia o contrário não é verdade. Vale ressaltar que o ciclo de vida de um *Fragment*, não depende de outros *Fragments*, mesmo que estes pertençam a mesma *Activity* e dividam o mesmo espaço nela [12].

O sistema Android faz o gerenciamento do ciclo de vida dos controladores de interface de usuário (*Activity* ou *Fragment*), e portanto pode decidir quando destruí-los ou recriá-los. Ao destruir ou recriar, todos os dados referentes a interface de usuário serão perdidos [7]. É possível usar o método `onSaveInstanceState()` para salvar uma pequena quantidade de dados, mas esse método não deve ser utilizado para guarda uma grande quantidade de informações. Outros pontos importantes sobre os controladores que devem ser citados são [7]:

- frequentemente precisam realizar solicitações assíncronas.
- gerenciam muitas solicitações e garantem que essas solicitações sejam limpas pelo sistema após a destruição para evitar possíveis vazamentos de memória.
- precisam que o gerenciamento receba bastante manutenção.
- desperdiçam recurso quando é recriado, pois pode ser necessário fazer novas solicitações já realizadas anteriormente.
- não devem conter lógicas de negócios.
- não devem ser responsáveis por carregar ou salvar dados, tanto localmente, quanto remotamente.

A separação de conceitos é o mais importante princípio a seguir aqui. Um erro comum cometido é escrever todo o código ou parte do código, que deveria estar na lógica de negócios, dentro da *Activity*. Classes baseadas em interface de usuário devem conter apenas interações entre o usuário e o aplicativo, como entrada e saída de dados. Fazendo isso têm-se menos problemas com a aplicação e seu ciclo de vida [2], pois o OS Android pode remover uma *Activity* ou *Fragment* da memória principal sempre que necessário, devido a novas interações ou por falta de memória.

#### 2.2.1.2 Comunicação

A comunicação entre componentes e aplicações Android acontecem por intermédio do *Intent*. O *Intent* é um objeto de mensagem, ou simplesmente mensagens que possibilitam a comunicação entre componentes do aplicativo, onde um componente do aplicativo pode solicitar a funcionalidade de outro componente do Android, ou do próprio aplicativo. A comunicação pode ser usada para iniciar uma atividade (*Activity*), um serviço (*Service*), ou fornecer uma transmissão (*Broadcast*) [16].

Existem dois tipos de *Intent*, a explícita, e a implícita. A *Intent* explícita não indica de forma clara qual componente do sistema deve ser chamado, e geralmente é utilizado para iniciar um componente do próprio aplicativo em execução. De outro modo, a *Intent* implícita não deixa especificado para o sistema qual componente deve ser solicitado, permitindo que qualquer componente de outra aplicação receba a comunicação [16].<sup>2</sup>

Geralmente o *Intent* possui dados para a ação. Esses dados podem ser direcionados para diversas ações distintas, e podem conter um número de telefone que será usado para realizar uma chamada telefônica, ou até mesmo uma mensagem que será enviada para um e-mail. Para realizar essas ações faz necessário o uso de Identificadores de Recursos Uniformes (URI), que referencia os dados a serem aproveitados e o seu tipo. Declarar o tipo desses dados é fundamental para realizar essa comunicação, pois um número de telefone não deve ser enviado para uma *Activity* que é responsável apenas por abrir imagens [17].

#### 2.2.2 Services

*Services* é um componente para lidar com tarefas que precisam ser executadas de forma silenciosa. Um *Service* é muito utilizado para realizar processos remotos ou operações

---

<sup>2</sup> Declarar *intent filter* no AndroidManifest.xml do aplicativo.

de longa duração, como uma transferência de arquivo, de modo que esta execução não se comporta de forma tão visível para o usuário do dispositivo [4]. Um *Service* faz parte do todo de um aplicativo, portanto não é um processo separado de uma aplicação, e sua execução é realizada no mesmo processo do aplicativo ao qual pertence [13].

Os *Services* estão divididos em dois tipos [4], os que realizam tarefas iniciadas pelo usuário de forma indireta, e os que operam em segundo plano de forma invisível, sem que o usuário saiba. Um exemplo para a primeira situação seria, um aplicativo de gerenciamento de arquivo, que o usuário pode decidir transferir arquivos de uma pasta para outra, e optar por deixar que esta tarefa seja executada de forma oculta, mas permaneça em primeiro plano no sistema apenas de forma visível como ponto de notificação, enquanto ele navega por outras *Activities*. Para a segunda situação, um aplicativo para backup de fotos ou arquivos, que periodicamente realiza suas tarefas quando necessário, em segundo plano e sem que o usuário perceba.

### 2.2.3 Broadcast Receivers

O *Broadcast Receiver* é um componente do sistema, e que transmite eventos ao aplicativo que não estejam em execução no momento. A função desse componente é fazer com que o aplicativo não precise permanecer em execução até o momento em que ele seja necessário. Um exemplo seria uma notificação avisando que a bateria do dispositivo está com pouca carga [4].

### 2.2.4 Content Providers

Os *Content Providers* ou provedores de conteúdo, encapsulam os dados e definem mecanismos para acessar sistemas de arquivos, permitindo que aplicativos possam armazenar os dados e compartilhá-los com outras aplicações [14]. A partir desse compartilhamento é possível consultar ou modificar os dados, caso o provedor permita. O próprio OS Android oferece um *content provider* para gerenciar os dados dos contatos do usuário. Deste modo qualquer aplicativo com permissão para acessar os contatos pode consultar esses dados [4].

## 2.3 COMPONENTES DE ARQUITETURA ANDROID

Nesta seção, veremos como é recomendado a criação de aplicativos Android, e sua estrutura usando a *Android Architecture Components* (componentes de arquitetura Android). Essa arquitetura serve para diferentes tipos de aplicações. Vale ressaltar que essa arquitetura é recomendada, mas não torna necessariamente as outras arquiteturas utilizadas para criar aplicativos android obsoletas ou defasadas, nem tão pouco ruins.

### 2.3.1 Visão Geral

A princípio os componentes de arquitetura Android utiliza-se de uma variedade de bibliotecas, que juntas tornam o aplicativo robusto, testável e de fácil manutenção [15]. Para melhor atender esses requisitos, são recomendados o uso de pelo menos 4 bibliotecas que vão dar estrutura para o funcionamento do aplicativo, que são: *LifeCycle*, *LiveData*, *ViewModel* e *Room*.

Antes de tudo é preciso saber o que cada uma dessas bibliotecas podem trazer como benefício, e como elas devem ser estruturadas para a criação do aplicativo. As principais características desses componentes são [2]:

- ***LifeCycle***: Gerencia o ciclo de vida do aplicativo, auxiliando os componentes *Activity* e *Fragment* a manterem as suas configurações salvas, sendo assim, evitando perda de dados, e tornando o carregamento dessas informações mais fáceis [15].
- ***LiveData***: Deve ser usado para criar objetos que podem notificar quando eles são alterados, ou quando há alterações feitas no banco de dados [15]. Esse componente trabalha em conjunto com o *LifeCycle*, reconhecendo o ciclo de vida da aplicação [8].
- ***ViewModel***: É responsável por armazenar os dados presentes na interface de usuário, que não são descartados quando um aplicativo é rotacionado, ou para guardar dados que não estejam relacionados às configurações da *Activity* e *Fragment* [15].

- **Room:** Guarda os dados das aplicações em execução em cache no armazenamento do dispositivo [9]. Essa ferramenta faz um mapeamento de objetos SQLite e recomenda-se ser utilizado para retornar um *LiveData* [15].

Como já foi dito, esses componentes são importantes para o correto funcionamento do aplicativo, e durante a execução da aplicação eles interagem entre eles apenas com módulos de nível próximo.

A figura 2 ilustra o funcionamento dos componentes e divisão dos módulos na *Android Architecture Components*, e como cada um desses módulos trocam informações quando a aplicação está em execução. Após construído, cada componente depende apenas do módulo de que está logo abaixo de si [2].

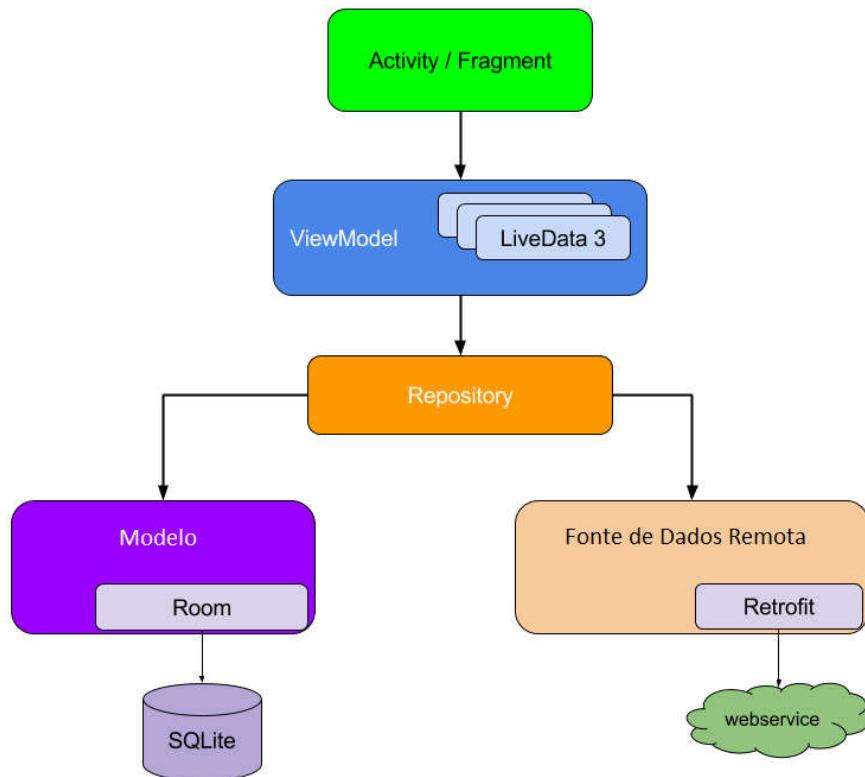


Figura 2: Diagrama dos módulos dos componentes de arquitetura<sup>3</sup>

---

<sup>3</sup> Disponível em: <https://developer.android.com/topic/libraries/architecture/images/final-architecture.png>. Acesso em janeiro de 2020.

Seguindo essa arquitetura, as *Activities* e *Fragments* que fazem parte de um único módulo, dependem apenas do módulo logo abaixo *ViewModel*. Além disso o módulo *Repository* pode estar ligado a outros dois modelos de dados, um modelo dados armazenado localmente, e outro remotamente [2]. A persistência de dados local é importante para que o aplicativo continue funcionando mesmo que o usuário esteja off-line.

Essa estrutura torna a aplicação mais consistente e melhor de se utilizar, pois mesmo que o aplicativo não esteja em segundo plano, ou que o aplicativo já tenha sido destruído da memória principal pelo sistema por falta de espaço, ele será reaberto mais rapidamente, restaurando as informações do usuário à *Activity*, primeiramente trazendo esses dados da persistência local, e posteriormente atualizando as informações através dos dados armazenados remotamente, caso esses dados estejam desatualizados [2].

### 2.3.1.1 ViewModel

A classe *ViewModel* foi criada para encapsular e gerenciar dados referente à interface de usuário, ou seja, armazenar e atualizar dados presentes nas *Activities* e *Fragments*. Isso permite que a *ViewModel* consiga manter os dados intactos mesmo após alterações na configuração do dispositivo feitas pelo sistema Android. O ciclo de vida de um Objeto do tipo *ViewModel* existe na memória até que uma ciclo de vida de uma *Activity* ou *Fragment* seja finalizado [7].

Como já foi dito antes, o ciclo de vida de um *Fragment* não é dependente de outros *Fragments*, e eles podem compartilhar um espaço na mesma *Activity* [12]. Todavia um *Fragment* não tem conhecimento de outro *Fragment*, e portanto o *ViewModel* pode surgir como uma solução para esse problema, pois, dois *Fragments* distintos podem ter um mesmo objeto do tipo *ViewModel* no seu escopo. Sendo assim, dois ou mais *Fragments* podem compartilhar o mesmo *ViewModel* a fim de gerar uma comunicação entre eles [7]. A *ViewModel* está logo abaixo do módulo que contém a *Activity* e *Fragment*, sendo assim, ela não pode ter conhecimento desses componentes [2].

Um objeto do tipo *ViewModel* fornece dados para uma *Activity* ou *Fragment*. Esse tipo de objeto contém lógica de negócios para manipulação de dados, logo ele deve se comunicar com componentes existentes no *model*, para que possa carregar e modificar dados através de encaminhamentos solicitados pelo usuário do dispositivo [2].

Os dados do objeto do tipo *ViewModel* são encapsulados de maneira automática quando acontecem alterações nas configurações do sistema, com o objetivo de que eles sejam rapidamente recuperados quando forem solicitados novamente. Um bom exemplo sempre utilizado é, manter esses dados a salvo mesmo após uma rotação de tela [7]. A figura 3 demonstra como isso ocorre.

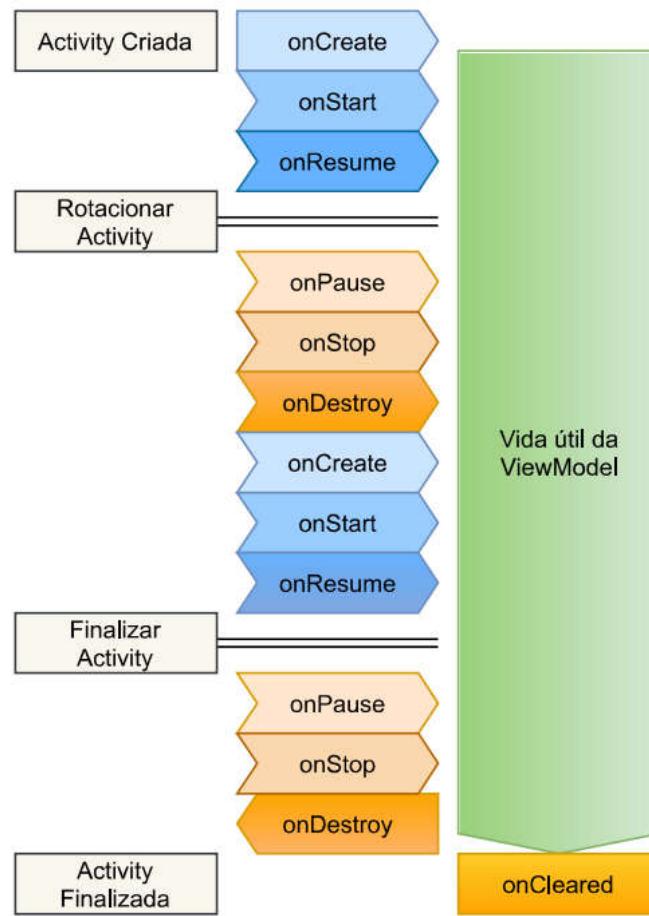


Figura 3: Vida útil da *ViewModel* durante execução de uma *Activity*<sup>4</sup>

A *ViewModel* trabalha em conjunto com o *LiveData* e o *Room*. O *Room* notifica o *LiveData* quando há alterações no banco de dados, enquanto o *LiveData* é responsável por atualizar os dados apresentados na interface de usuário [7].

---

<sup>4</sup> Disponível em: <https://developer.android.com/images/topic/libraries/architecture/viewmodel-lifecycle.png>. Acesso em janeiro de 2020..

### 2.3.1.2 LiveData

A classe *LiveData* é usado para armazenar dados observáveis [8]. O *LiveData* respeita o ciclo de vida dos componentes do aplicativo, como *Activities*, *Fragments* e *Services*. Esta classe pode ser usada por outros componentes para monitorar alterações nos objetos [2]. O *LiveData* atualiza apenas observadores de componentes de aplicativos que estão ativos, onde o ciclo de vida desses componentes estão no estado “começado” ou “retomado” [8].

O *LiveData* permite que os Observadores da *Activity* ou *Fragment* façam parte das atualizações, e sempre que um objeto *LiveData* é alterado, a interface de usuário é atualizada automaticamente. Existem várias vantagens em usar o *LiveData*, como [8]:

- Garantir que a interface de usuário esteja sempre atualizada, pois o *LiveData* notifica os objetos *Observer* quando eles passam por mudanças no seu ciclo de vida. Quando essa mudança ocorre os dados mais recentes são recebidos pelo componente de interface de usuário. Um *Observer* é considerado ativo quando seu estado é “começado” ou “retomado”, e inativo quando seu estado é “parado”.
- Não permitir vazamento de memória, porque o objeto *Observer* é limpo da memória quando o ciclo de vida do componente de interface de usuário é destruído, devido ao fato dele está atrelado ao objeto *LifeCycle*.
- Não ocorrer erros por partes de *Activities* e *Fragments* devido a inatividade, portanto, o objeto *LiveData* não irá notificar esses componentes de interfaces de usuário. O *LiveData* remove de forma automática o *Observer* quando a *Activity* ou *Fragment* está inativo.
- Evitar manipulação manual do ciclo de vida dos componentes da interface de usuário, pois o próprio *LiveData* é responsável por esse gerenciamento.
- Trabalhar em conjunto com *ViewModel*, neste modo, um componente de interface de usuário, terá os seus dados atualizados sempre que for recriado (por motivos de alterações nas configurações do sistema). Quando o componente de interface de usuário é recriado, ele recebe a mesma instância do objeto *ViewModel*.

Ainda é possível usar o padrão de projeto Singleton no objeto do tipo *LiveData* para que qualquer *Observer* que necessite do *LiveData*, possa acessá-lo sempre que for preciso,

conectando o objeto *LiveData* apenas uma vez ao serviço do sistema, oferecendo mais consistência de dados [8].

### 2.3.1.3 Room

Seguindo o modelo da *Android Architecture Component* modelado na figura 2, O repositório pode ser dividido em outros dois ou mais módulos, o *Model* (para armazenamento de dados local, o qual a biblioteca *Room* é a recomendado a ser usada), e o *Remote Data Source* (para recuperação de dados em armazenamento remoto), e ambos os módulos são usados para recuperar e atualizar os dados que são visualizados no componente de interface de usuário [2].

Os dados do componente de interface de usuário são recuperados imediatamente quando disponíveis localmente pelo *Model*. Todavia, como foi visto no tópico 2.2.1, um componente de interface de usuário pode ser destruído a qualquer momento, e quando isso ocorre, o aplicativo precisa buscar os dados remotamente através do módulo *Remote Data Source*. Não é correto simplesmente manter esses dados solicitados remotamente em cache, pois isso pode ocasionar problemas relacionados a inconsistência de dados, portanto, a aplicação pode recarregar o componente de interface de usuário com dados salvos em cache, e esses dados já estarem desatualizados. Isso poderia fazer com que o aplicativo apresentasse duas versões diferentes dos dados em uma mesma execução [2].

A melhor maneira para conter esses problemas é usando um modelo persistente, e a biblioteca *Room* é uma recomendação a se seguir. Com essa biblioteca os aplicativos podem gerenciar melhor uma maior quantidade de dados persistentes localmente, onde o armazenamento em cache de dados mais importantes é muito utilizado [9]. Além disso, essa biblioteca, em cada compilação, valida cada consulta em relação ao seu esquema de dados, sendo assim, os erros de consultas SQL, são erros em tempo de compilação, evitando assim erros em tempo de execução [2]. Qualquer tipo de alteração realizada pelo usuário são sincronizadas com o armazenamento remoto quando o dispositivo estiver online novamente [9].

### 2.3.2 Discussão

Os componentes de arquitetura Android foram construídos para tornar os aplicativos criados pelos desenvolvedores robustos, além de criar uma estrutura recomendada a ser seguida pelos desenvolvedores, de forma padronizada [2].

#### 2.3.2.1 Vantagens

Individualmente cada um desses componentes podem trazer suas vantagens, mas recomenda-se que elas trabalhem em conjunto, para produzir um melhor resultado. Unificadas podem trazer inúmeras vantagens para a aplicação, como gerenciamento do ciclo de vida dos componentes de interface de usuário, prevenção de vazamento de memória, persistência de dados local com atualização de informações em tempo real [2].

Além do que já foi mencionado antes, seguindo essas recomendações, é possível alcançar uma maior capacidade de testes [2], dificuldade comum encontrada na arquitetura *Model-View-Presenter* (MVP) [5]. Um outro problema existente no MVP que pode ser sanado usando componentes de arquitetura Android é a relação estreita entre a *View* e ao *Presenter*, onde um tem referência do outro de forma exclusiva<sup>5</sup>, e segundo a recomendação devemos evitar referências a *View* (no caso de aplicações Android, a *Activity/Fragment*, mesmo que não seja uma *View* de fato), podendo nesse caso, resolver esse problema utilizando um único *ViewModel* para guardar e compartilhar dados entre um ou vários controladores de interface [2][7], permitindo apenas que a *Activity* ou *Fragment* tenha referência da *ViewModel*, e não o contrário. Uma outra vantagem em usar componentes de arquitetura Android é ter melhor modificabilidade [2], problema característico da arquitetura *Model-View-ViewModel* (MVVM) [5].

Ambas arquiteturas, MVP e MVVM, são bastante difundidas e utilizadas para criação de aplicativos Android, devido ao fato de serem adaptações feitas a partir do *Model-View-Controller* (MVC) [18]. Além de tudo isso, usando o componente *LiveData* é possível tornar as apresentações dos dados reativo, devido a sua capacidade de observar modificações em objetos e notificá-las, para que alterações possam ser feitas na apresentação quando for o caso.

---

<sup>5</sup> relação 1..1

### 2.3.2.2 Desvantagens

Apesar desse pacote de componentes trazer diversos benefícios, vale ressaltar que construir uma aplicação seguindo essa recomendação não é tão simples, pois é necessário compreender a função de cada um desses novos componente, e como eles interagem durante o funcionamento, tornando a estruturação do código muito diferente do que tem sido mais utilizado atualmente.

Levando em conta o estado atual das estruturas dos códigos, é algo comum ver *Activities* contendo lógicas de negócios, quando deveria apenas controlar a interface de usuário, e isso tornou-se ainda pior com o crescimento das aplicações, que acabaram por gerar *Activities* gigantescas, portanto, um maior número de interações com o usuário em uma mesma interface [19], logo, uma classe controladora de interface contendo bastante código. O componente *Fragment* foi criado afim de solucionar esse problema, no entanto, erros comuns, como tornar o *Fragment* responsável por gerenciar parte da lógica de negócios ainda são cometidos [19][22].

É um desafio seguir essas recomendações, e manter a aplicação atualizada com todos os recursos do pacote de componentes, pois frequentemente novos componentes estão sendo lançados para melhorar a criação, e utilização dos aplicativos[20].

### 2.3.2.3 Melhorias

Uma das funções da *ViewModel* é compartilhar dados entre *Fragments*, e isso é muito útil para compartilhar uma informação comum entre *Fragments*, principalmente quando esses *Fragments* são criados em momentos distintos durante o uso da aplicação, logo, o *Fragment* 1 pode não conhecer o momento exato de passar uma informação para o *Fragment* 2. Outro problema que também pode ocorrer, é em situações em que um *Fragment* 1, que contém um dado compartilhado com o *Fragment* 2, possa ser destruído, e o *Fragment* 2 permaneça vivo contendo a referência do dado que está no *Fragment* 1. Nessa situação a *ViewModel* pode compartilhar essa informação de forma segura e consistente. Todavia usar *ViewModel* para solucionar esses problemas, pode tornar a *ViewModel* uma classe grande, contendo informações de vários *Fragments*, portanto, contendo também variáveis e método que são exclusivos de um único *Fragment*.

Uma solução para o problema citado é separar a *ViewModel* em outras classes *ViewModel*, caso isso seja possível para a aplicação. Um bom exemplo para essa separação, seria uma aplicação compartilhando uma informação armazenada dentro do Banco de Dados local (pode ser um Banco de Dados criado através da biblioteca *Room*), onde as *ViewModels* podem acessar o mesmo Banco de Dados, simplesmente implementando o padrão de projeto Singleton para a criação dele. Assim a classe *ViewModel* não se tornará grande e complexa, mantendo sua função, guardar os dados, e conter lógicas de negócios para apenas um controlador de interface de usuário.

## 2.4 MICRO FRONTENDS

O conceito de *Micro Frontends* não é uma novidade, sabendo-se que a abordagem é baseada em sistemas independentes, como por exemplo o *Front Integration dor Verticalized System* [23]. Todavia, o termo *Micro Frontends* surgiu no *ThoughtWorks Technology Radar* em 2016 [23][24].

As aplicações atuais costumam usar a ideia de aplicativos de página única, a qual, cada página possui sua estrutura individual [23]. Desenvolver um *front-end*, onde há várias equipes trabalhando não é fácil, e pode se tornar ainda mais difícil se tratando de aplicações grandes. A tendência atual é que grandes *front-ends* sejam divididos em pequenos blocos menores e gerenciáveis [24].

*Micro Frontends* tem como principal objetivo, tornar a criação de aplicações robustas, organizadas e separadas por equipes, o que abrange o conceito de microsserviços. Cada equipe é responsável pela criação de componentes de interface distintos, trazendo ainda mais benefícios, como poder separar as equipes por setores nas quais elas estão mais preparadas e qualificadas. O objetivo é construir interfaces como um conjunto de vários componentes de interface; tornando possível a separação de tarefas em grupos independentes, e obtendo um melhor gerenciamento das aplicações [23]. A figura 4, logo abaixo exemplifica de maneira genérica esse fluxo.

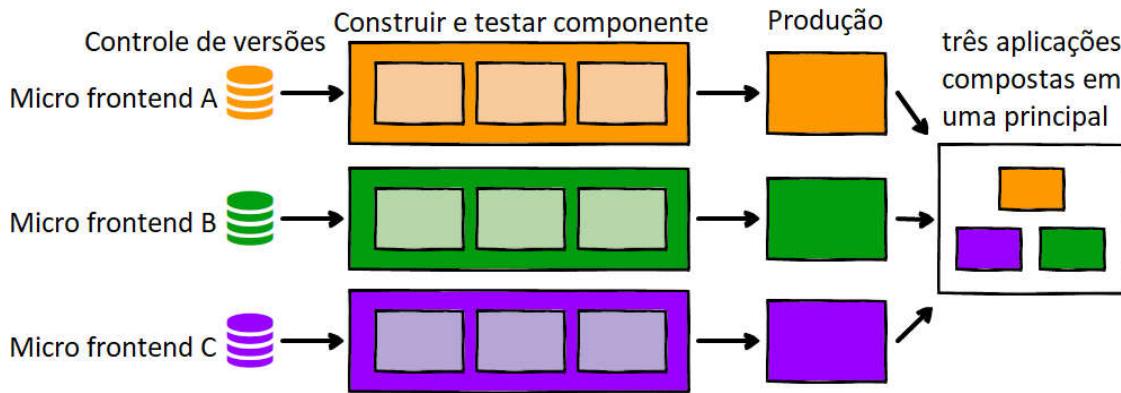


Figura 4: Implementação independente usando *Micro frontends*<sup>6</sup>

A figura demonstra como as equipes se dividem e criam os seus componentes individualmente. As equipes criam, implementam e testam os seus *front-end*, chegam a um produto final, e posteriormente lançam as suas versões dentro da aplicação final.

Usar *Micro Frontends* tornará a arquitetura da aplicação escalável, mantendo a aplicação estável e funcional mesmo durante a manutenção de sua estrutura, o que é muito benéfico, visto que as aplicações atuais estão em constantes crescimento e atualização [24]. A seguir os principais benefícios serão citados ao aplicar as técnicas de *Micro Frontends* na arquitetura da aplicação.

#### 2.4.1 Atualização do *front-end*

Sempre com a aparição de novos recursos, é viável que apenas o componente que deve ser atualizado com esse novo recurso, seja reescrito e atualizado. Com a técnica de atualização incremental, as equipes podem implementar um novo produto sempre que necessário, e lançá-lo para o usuário final quando estiver pronto, incrementando o produto final.

#### 2.4.2 Independência entre os *front-ends*

Organizar a estrutura do código de cada componente de interface de forma separada, desacoplando componentes que não devam ter conhecimento um do outro. Seguindo essa técnica, dificilmente erros acidentais de acoplamento serão cometidos, o que pode tornar o código da aplicação mais organizada, e de leitura mais fácil.

---

<sup>6</sup> disponível em <<https://martinfowler.com/articles/micro-frontends/deployment.png>>. Acesso em 5 Mar. 2020.

#### **2.4.3 Implantação do *front-end***

Cada equipe individualmente está trabalhando na criação do seu próprio componente de interface, e de maneira independente, estas equipes podem lançar novos recursos, que outrora não estavam disponíveis na aplicação. Quando a desacoplação de componentes foi feita corretamente, a tendência é que a implementação de novas funções se torne menos trabalhosa.

#### **2.4.4 Autonomia das equipes**

As equipes têm propriedades para criar novos recursos que trarão benefícios aos usuários através de funções que poderão ser implantadas posteriormente. Para isso as equipes precisam trabalhar de forma vertical na arquitetura, de maneira que uma equipe não seja responsável exatamente por uma camada da arquitetura como ocorre em alguns casos, mas que cada equipe seja responsável por um produto.

### 3 O APLICATIVO LIBFLIX

Para que a arquitetura recomendada possa ser testada, foi desenvolvido o aplicativo Libflix, com o intuito de aplicar os conceitos vistos, de modo a apresentar o comportamento dos componentes, durante a execução da aplicação, como também descrever de modo prático como a aplicação funciona com a utilização desses componentes.

#### 3.1 VISÃO GERAL

O Libflix é um aplicativo para apresentar dados sobre filmes, compartilhando informações como nome, data de lançamento ou duração do filme. Os dados dos filmes são coletados a partir do *The Movie Database* (TMDB), um banco de dados de filmes criado<sup>7</sup> e mantido com a ajuda da comunidade.

Nessa aplicação é possível visualizar a lista de filmes disponibilizada pela base de dados remota, na qual essa lista, por opção do desenvolvedor, é a lista de destaque do TMDB. Apesar disso, é possível que o usuário faça uma busca por filmes, caso ele queira, permitindo que mais filmes dessa base de dados remota possam ser encontrados e apresentados através da lista.

O aplicativo também disponibiliza de uma base de dados local, assim como a arquitetura recomendada exemplifica, deste modo, é possível que o usuário possa salvar filmes na memória do dispositivo, e que estes filmes também sejam apresentados em uma lista.

#### 3.2 LISTA DE REQUISITOS

O aplicativo possui algumas funções que serão descritas nesta seção. Essas funções serão separadas por Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF).

##### 3.2.1 Requisitos Funcionais

###### 3.2.1.1 [RF01] Solicitar Lista de Destaques

O aplicativo deve realizar a requisição da lista de filmes de destaque da base de dados remota, e apresentar a lista de filmes após obter com sucesso, as informações recebidas no

---

<sup>7</sup> O TMDB foi criado em 2008

JSON (Notação de Objetos JavaScript). A lista possui um tamanho determinado pela própria fonte de dados remota.

### 3.2.1.2 [RF02] Apresentação da Lista de Filmes da Base de Dados Remota

A lista responsável por apresentar os filmes da base de dados remota deve conter tamanho que foi definido pelo JSON, e apresentar uma lista de cartões, cada um contendo informações individuais dos filmes recuperados.

### 3.2.1.3 [RF03] Apresentação da Lista de Filmes da Base de Dados Local

A lista responsável por apresentar os filmes da base de dados local deve conter tamanho 10 (definido pelo desenvolvedor), e apresentar uma lista de cartões, cada um contendo informações individuais dos filmes recuperados.

### 3.2.1.4 [RF04] Apresentação do Cartões do Filme

O cartão do filme deve conter detalhes, como pôster do filme, nome do filme, data de lançamento do filme, e popularidade do filme na comunidade do TMDB. O nome do Filme deve ter maior ênfase, possuindo um tamanho de letra maior, e cor mais clara que as demais informações. Além disso o nome do filme deve ser limitado a duas linhas.

### 3.2.1.5 [RF05] Mudança de Página

O aplicativo deve apresentar uma barra de controle de páginas que permita que o usuário possa avançar, voltar, ir para a última página, ou ir para a primeira página, caso seja possível.

### 3.2.1.6 [RF06] Pesquisa na Base de Dados Remota

A aplicação deve realizar uma pesquisa por filmes na base de dados remota através da palavra ou letra digitada pelo usuário na barra de pesquisa. A palavra deve ser usada para realizar uma requisição à API, que pode retornar uma lista de filmes baseado na palavra digitada.

### 3.2.1.7 [RF07] Pesquisa no Banco de Dados Local

A aplicação deve realizar a pesquisa no banco de dados local, usando a palavra ou letra digitada pelo usuário na barra de pesquisa, para encontrar filmes que contenham essa

palavra ou letra no seu nome, e adicionar esses filmes numa lista de filmes filtrados na pesquisa. Ao fim da busca em todo o banco de dados local essa lista de filmes filtrados devem ser usados para atualizar a lista de apresentação.

#### 3.2.1.8 [RF08] Chamar Tela de Detalhes do Filme

Quando o usuário selecionar um filme da lista de filmes, a aplicação deve repassar o código de identificação do filme para a tela de detalhes que se encarregará de recuperar as informações do filme, do banco de dados remoto ou local.

#### 3.2.1.9 [RF09] Adicionar Filmes aos Favoritos

Na tela de detalhes do filme, quando o usuário tocar na estrela de favoritos, o filme deve ser adicionado ao banco de dados local, caso ele ainda não exista nele, ou seja, ainda não é um filme favorito do usuário.

#### 3.2.1.10 [RF10] Remover Filmes dos Favoritos

Na tela de detalhes do filme, quando o usuário tocar na estrela de favoritos, o filme deve ser removido do banco de dados local, caso ele já exista nele, ou seja, é um filme favorito do usuário.

#### 3.2.1.11 [RF11] Abrir Homepage do Filme

Na tela de detalhes do filme, quando o usuário tocar nas informações da homepage do filme, a aplicação deve chamar uma atividade externa para abrir uma página web. O sistema é encarregado de apresentar ao usuários os aplicativos disponíveis capazes de realizar essa ação.

### 3.2.2 Requisitos Não Funcionais

#### 3.2.2.1 [RNF01] Retornar Lista Vazia

Caso não seja possível recuperar a lista de filmes da base de dados remota, a lista responsável por apresentar os dados deve receber uma lista vazia, não permitindo que valores nulos sejam aplicados a lista de apresentação.

### 3.2.2.2 [RNF02] Apresentação dos Botões de Mudança de Página

Na barra de controle de páginas, deve ser apresentado ao usuário apenas os botões de mudança de página de acordo com a página atual e o total de páginas da lista de apresentação. Os botões de voltar ou ir para a primeira página devem ser desabilitados caso o usuário esteja na primeira página. Os botões de avançar ou ir para a última página devem ser desabilitados caso o usuário esteja na última página.

### 3.2.2.3 [RNF03] Buscar Filme nas Bases de Dados

Quando o usuário selecionar o filme da lista, deve ser realizado uma busca no banco de dados local, caso o filme exista no banco de dados dos filmes favoritos, então ao chamar a tela de detalhes do filme, a atividade atual deve repassá-la tanto o código identificador (ID) do filme, quanto a informação de que o filme existe na base de dados local.

### 3.2.2.4 [RNF04] Exibir Detalhes do Filme

Após o usuário selecionar o filme que deseja abrir para ver mais detalhes, a tela que exibe apresentará os dados do filme após receber a notificação de que o filme foi recuperado. Caso o filme exista na base de dados local, o filme deve ser carregado dela, mesmo que tenha sido aberto pela tela que apresenta a lista de filmes da base de dados remota.

### 3.2.2.5 [RNF05] Conexão Com Internet

O aplicativo deve ter acesso a conexão com internet para funcionamento correto e completo das funções disponíveis. Sem conexão, só será possível acessar os filmes armazenados no banco de dados local.

### 3.2.2.6 [RNF06] Tentativas de Requisição

Ao tentar realizar uma requisição para recuperar os dados de um filme na base de dados remota, é possível que ocorra algum erro durante a tentativa por diversos fatores. Caso ocorra algum erro durante a primeira tentativa, a aplicação deve tentar solicitar novamente os dados do filme até no máximo 5 vezes.

### 3.2.2.7 [RNF07] Comunicação com a API do TMDB

O aplicativo deve comunicar-se com a API do TMDB para buscar a lista de filmes, e poder apresentá-las ao usuário.

### 3.2.2.8 [RNF08] Status do Carregamento da Lista

A tela de que apresenta a lista de filmes deve apresentar uma imagem que corresponda com o status atual do carregamento da lista. Quando a lista estiver sendo buscada na base de dados remota, uma imagem de download da lista deve ser exibida até o término do processo. Quando houver erro no carregamento da lista, uma imagem de erro no download da lista deve ser exibida.

### 3.2.2.9 [RNF09] Compatibilidade do Aplicativo

O aplicativo deve estar disponível na plataforma Android. O nível de API recomendada é a partir da 28 (Android 9.0 - *Pie*), e nível mínimo da API é a 26 (Android 8.0 - *Oreo*).

## 3.3 ARQUITETURA DO APLICATIVO

A arquitetura do aplicativo Libflix apresentada na figura 4 mais abaixo, é similar ao diagrama da figura 2 que encontra-se na seção 2.3.1, a qual apresenta os módulos com os componentes de arquitetura. A arquitetura tem três camadas, que são *Presentation*, *Domain* e *Data*, e cada uma delas tem suas distintas responsabilidades.

### 3.3.1 As camadas da arquitetura

A *Presentation* possui os controladores de interface de usuário, como também adaptadores. Os controladores de interface de usuários, as *Activities* e *Fragments*, realizam chamadas ao *Domain* a partir de ações do usuário com a tela, e também podem enviar e receber dados dos adaptadores, que são responsáveis em preparar os dados que serão apresentados na tela. O *Domain* é composto pela *ViewModel*, por entidades de domínio, e por modelos de abstração. Cada *ViewModel* é responsável em guardar os dados de um controlador de interface individualmente. A *ViewModel* tem a referência ao *LifeCycle* dos seus respectivos controladores de interface, podendo realizar ações ao ocorrer mudanças no estado do ciclo de

vida desses controladores. Os dados relacionados a filmes, que ficam guardados na *ViewModel*, são do tipo *LiveData*. Esses dados podem ser requisitados ou guardados através do módulo *Data*. No *Data* está contido a base de dados, e os objetos de acesso a dados (DAO). A base de dados está dividido em base de dados local e remota, podendo solicitar filmes e listas de filmes da API, como também solicitar e guardar os dados dos filmes no armazenamento do dispositivo.

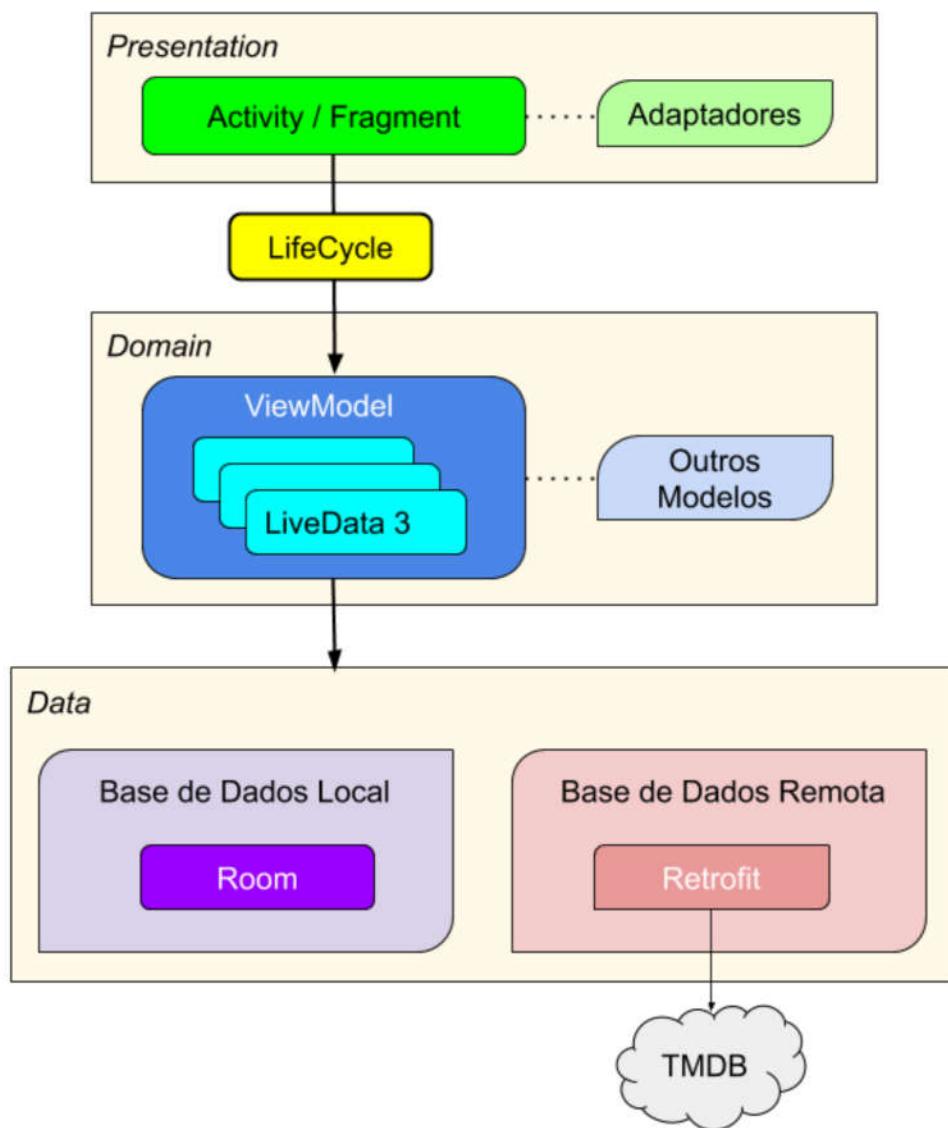
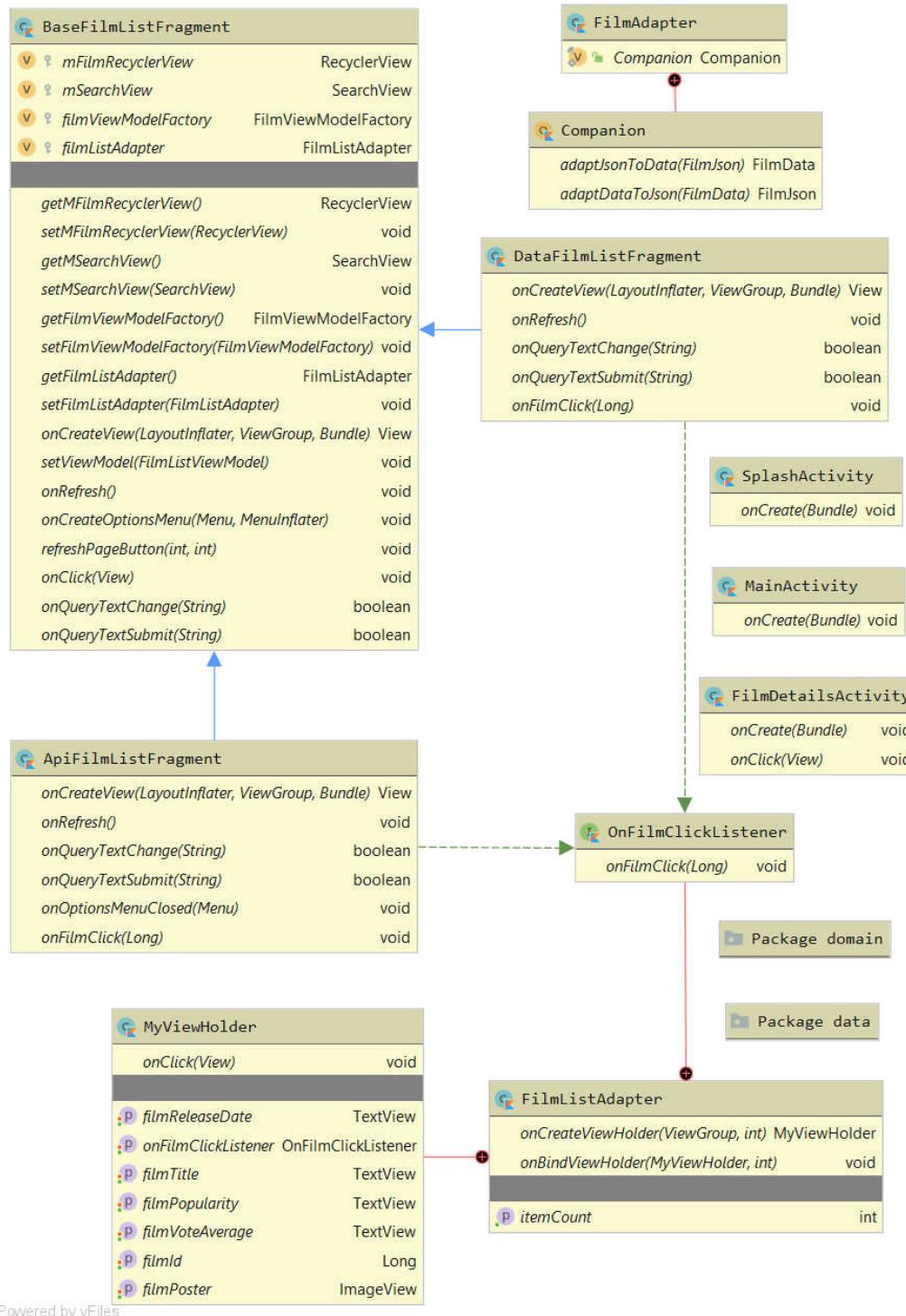


Figura 5: Arquitetura do aplicativo Libflix

Para compreender melhor como cada uma das camadas funcionam, as Figuras 5, 6 e 7 apresentam os modelos do diagrama de classe de cada uma delas.

### 3.3.2 Presentation



Powered by yFiles

Figura 6: Representação da camada *Presentation*

Na camada *Presentation*, o controlador **BaseFilmListFragment** do tipo *Fragment*, é herdado por **ApiFilmListFrgament** e **DataFilmListFrgament**, e ambos implementam o método de *click* dos cartões da lista de filmes, da interface **OnFilmClickListener**, existente na classe **FilmListAdapter**.

A classe **FilmAdapter** possui dois método dentro do *companionObject* para converter os tipos dos dados do filme. Alguns dados recebidos pelo JSON são do tipo que não podem ser salvos pela biblioteca *Room*, e portanto, precisa preparar esses dados para serem adaptados ao formato que o banco de dados local aceita.

### 3.3.3 Domain

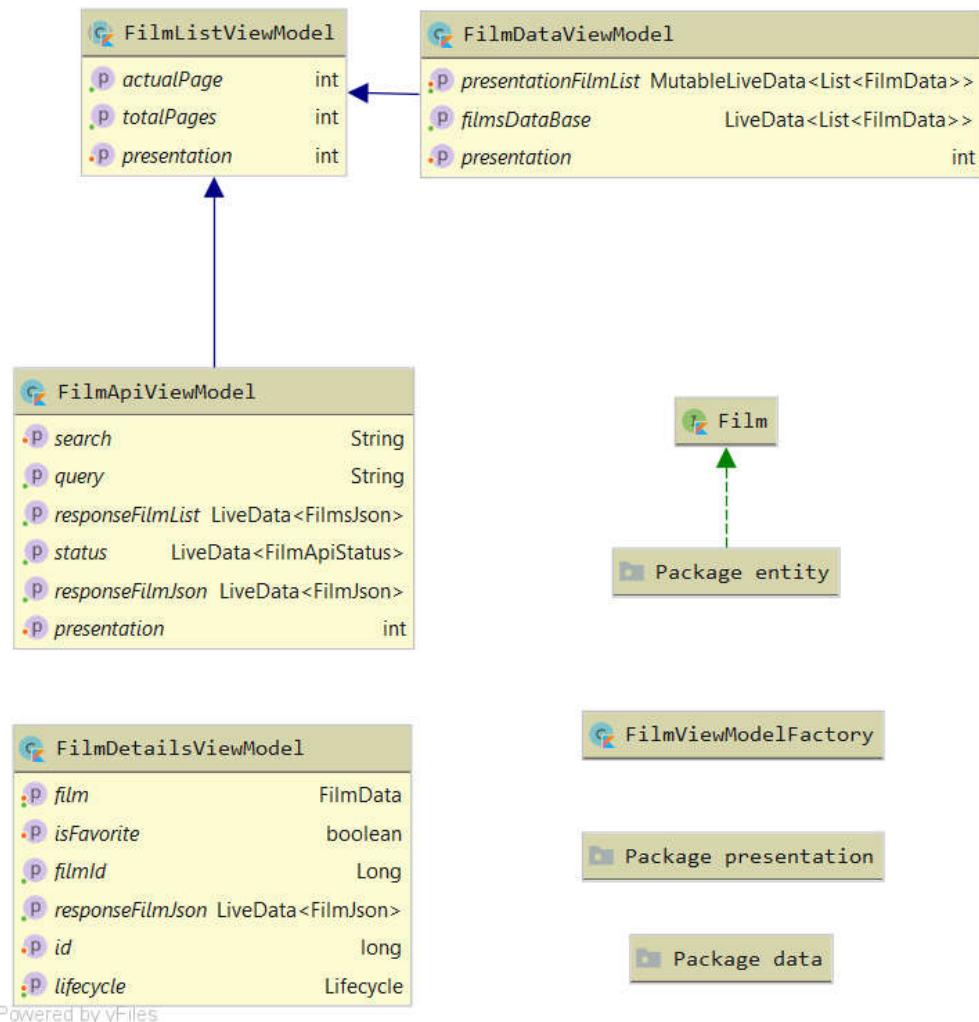


Figura 7: Representação da camada *Domain*

Na camada *Domain*, a classe **FilmListViewModel** é do tipo *ViewModel*, e é herdada pelas classes **FilmApiViewModel** e **FilmDataViewModel**. Estas classes guardam os dados da lista de filmes que podem ser manipulados pelos seus respectivos controladores de interface, e podem acessar a base de dados local e remota. Além disso, o *Domain* contém duas entidades que moldam características de filmes, de modo que, a entidade que é usada para pegar os dados da API pode conter dados nulos, enquanto a outra, não deve conter dados nulos, pois esta segunda, deve conter dados que podem ser salvos ou apresentados na interface de usuário.

### 3.3.4 Data

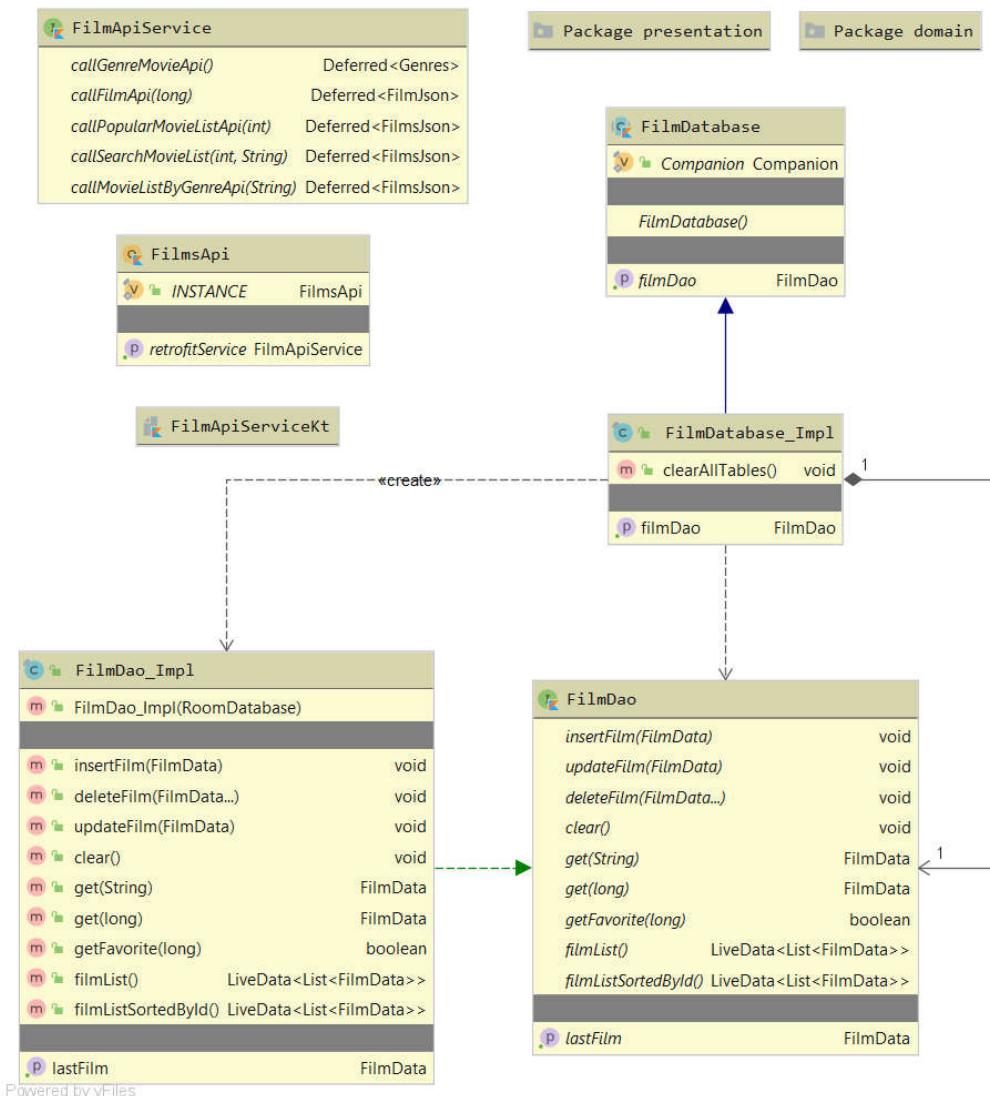


Figura 8: Representação da camada *Data*

A camada *Data* é responsável pelo gerenciamento dos dados, e nela temos a base de dados local e a remota. A base de dados remota (**Film ApiService**) utiliza a biblioteca Retrofit para poder recuperar os dados, auxiliada pelo conversor Moshi. Ambas foram criadas pela Square, e podem operar juntas para recuperar os dados da API, e converter de forma serializada os dados do JSON. A base de dados local (**Film Database**) utiliza a biblioteca *Room* para salvar, recuperar e apagar dados na memória do dispositivo. A base de dados local foi implementada usando o padrão *Singleton*, contendo o **Film Dao** para executar as ações necessárias para a manipulação desses dados.

## 4 APLICABILIDADE DOS COMPONENTES DE ARQUITETURA

Nesta seção, o aplicativo será apresentado, com os resultados obtidos ao construir o aplicativo usando a arquitetura recomendada e seus componentes. Os resultados serão apresentados com a demonstração das telas do aplicativo. O aplicativo está disponível no meu perfil do GitHub<sup>8</sup>.

### 4.1 OS COMPONENTES GRÁFICOS E A INTERFACE DE USUÁRIO

Nesta seção serão apresentados todas as telas do aplicativo. Em cada tópico das telas, serão descritos os componentes de arquitetura existentes, como também a forma como cada um desses componente se comportam durante a execução de suas tarefas.

#### 4.1.1 Splash Screen

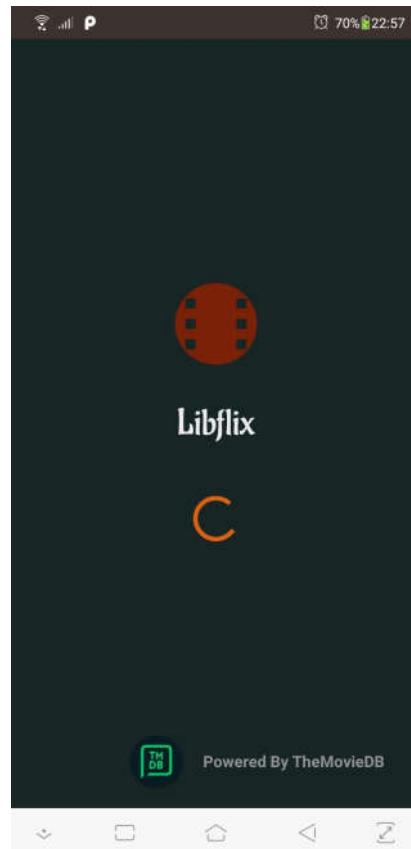


Figura 9: Splash screen

---

<sup>8</sup> repositório do github: <<https://github.com/ednaldomartins/ArchitectureComponentApp>>

A primeira tela que é apresentada, na figura 5, é a splash screen da aplicação, ou tela de abertura. Nesta tela é apresentado o ícone e o nome do aplicativo. Na parte de baixo da tela é mostrado que o The Movie Database é quem alimenta a aplicação (fornece as informações dos filmes). Nessa tela existe apenas componentes de criação de interface. Nesta tela é visto apenas alguns componentes de interfaces comumente utilizados para criação de interface.

#### 4.1.2 Ação dos Componentes Arquiteturais durante a requisição da lista de filmes

A tela que apresenta os dados capturados da API, é composta por uma *Activity* principal, que infla 2 Fragments, um deles é o visível na tela. Esse *Fragment* possui uma *ViewModel* particular, que é responsável por fazer a requisição das informações dos filmes e guardá-las em uma lista do tipo *LiveData*. Após a requisição, o objeto (lista) do tipo *LiveData* notifica os observadores que a lista foi alterada. Os observadores que estão dentro do controlador de interface (o *Fragment*), recebem a notificação, e nesse instante, os dados recebidos são adaptados e posteriormente aplicados a lista de apresentação. Esse tipo de comportamento é semelhante em todas as requisições de lista do servidor.

```
private var _requestFilmList = MutableLiveData<FilmsJson>()
val responseFilmList: LiveData<FilmsJson> get() = _requestFilmList
```

(a) Código - Lista do tipo *LiveData*

```
private suspend fun setRequestResult(callDeferred: Deferred<FilmsJson>) {
    try {
        _status.value = FilmApiStatus.LOADING
        val resultList = callDeferred.await()
        _requestFilmList.value = resultList
        _actualPage = resultList.page
        _totalPages = resultList.totalPages
        _status.value = FilmApiStatus.DONE
    } catch (e: Exception) {
        _status.value = FilmApiStatus.ERROR
    }
}
```

(b) Código - Requisitando lista de filmes

```

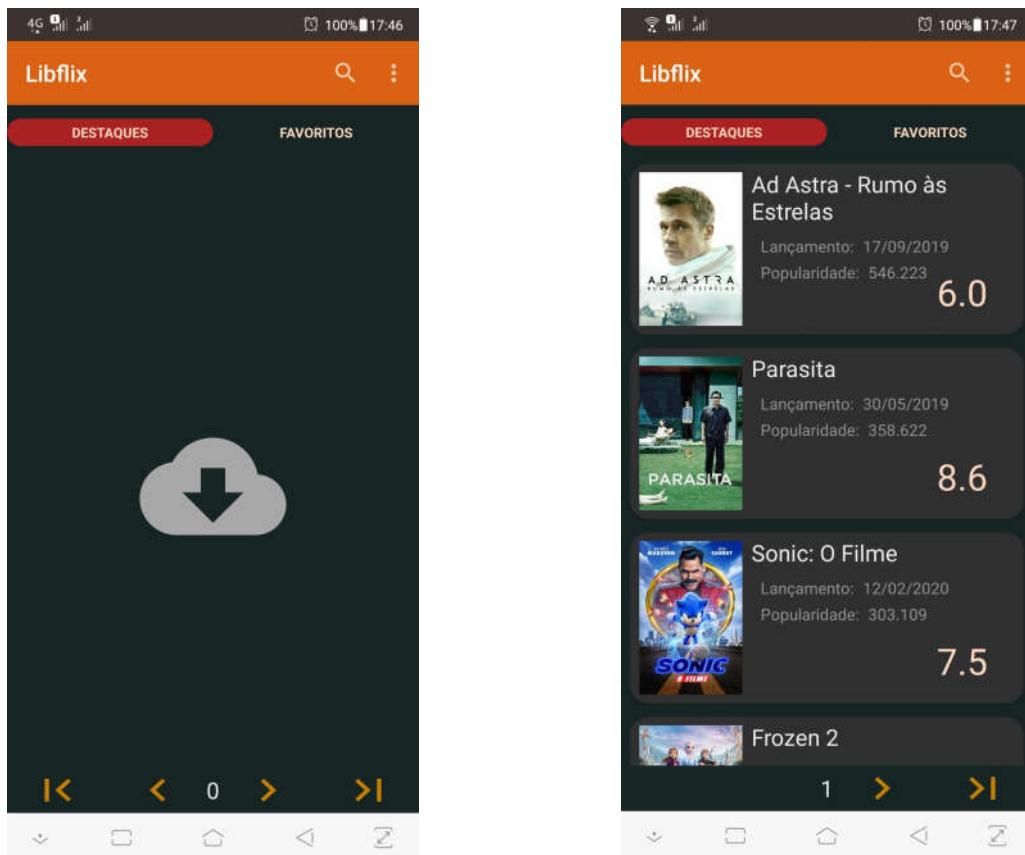
filmViewModel.responseFilmList.observe(owner: this, Observer { it: FilmsJson!
    refreshPageButton(it.page, it.totalPages)
    it.movies?.let { list ->
        filmListAdapter = FilmListAdapter(context = activity, filmListJson = list)
        mFilmRecyclerView.adapter = filmListAdapter
    }
})
}

```

(c) Código - Observador a espera da notificação

Figura 10: Código - Lista do tipo LiveData recebe os dados e notifica controlador

#### 4.1.2.1 Carregando a lista de destaques



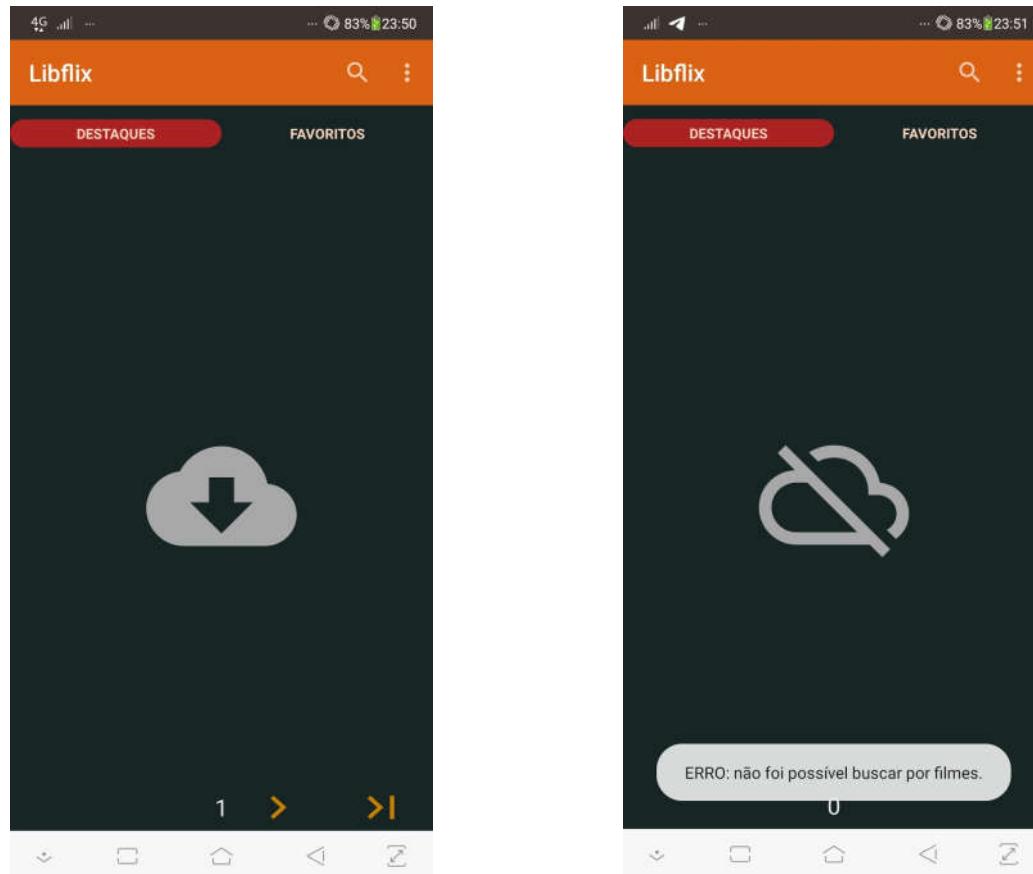
(a) Durante o carregamento das informações.

(b) Após carregar todas as informações.

Figura 11: Requisição e carregamento das informações dos filmes com sucesso

Nas duas imagens acima é demonstrado como a tela se apresenta durante o carregamento da lista de filmes da base de dados remota. Na imagem (a) o usuário está na tela de destaques, e as informações dos filmes estão sendo requisitadas a base de dados remota, e após o término do recebimento dessas informações, vê-se na imagem (b), uma lista de filmes carregada sendo exibida na tela.

#### 4.1.2.2 Problemas com o carregamento dos filmes



(a) Durante o carregamento das informações.

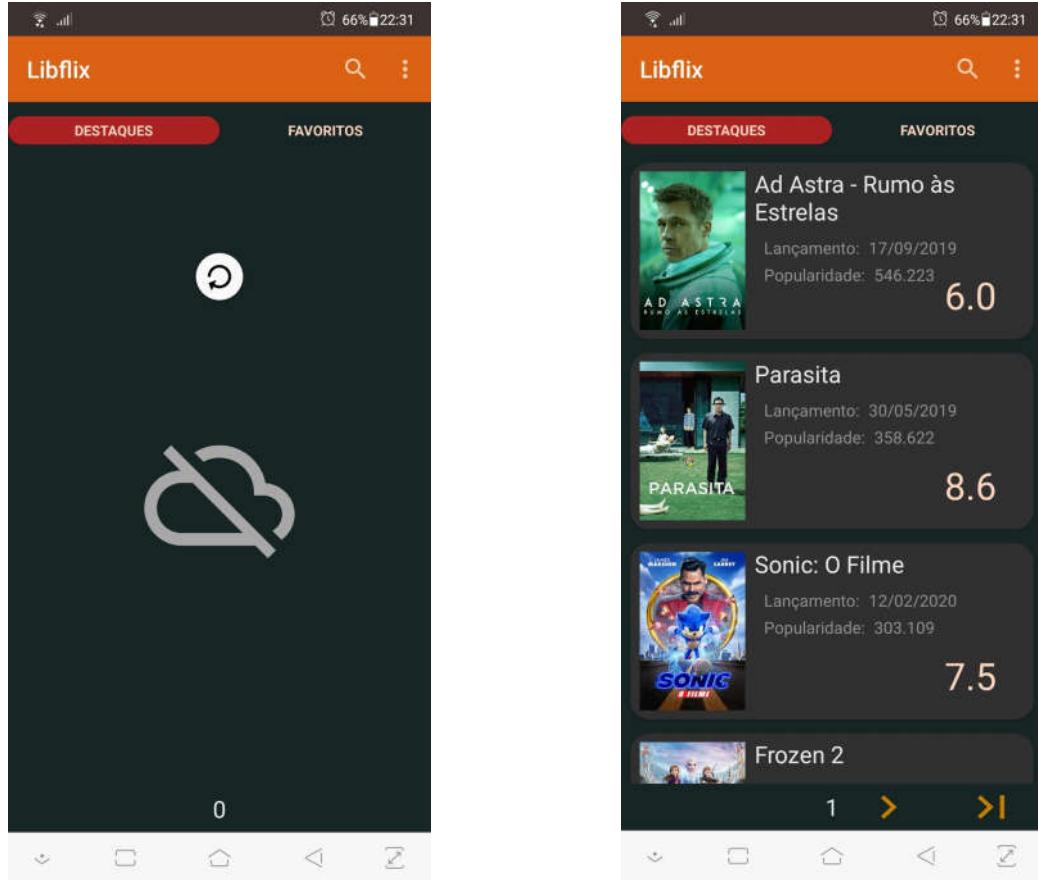
(b) Erro após tentativa da requisição.

Figura 12: Requisição e retorno de erro após a tentativa de comunicação com o servidor

Nas imagens acima, é visto novamente uma tentativa de requisição a base de dados remota na imagem (a), todavia, a resposta obtida na imagem (b) é diferente, devido a falha durante requisição das informações. Essa falha pode ocorrer por diversos fatores, como: problema com a conexão de internet, ou até problemas externos relacionados ao servidor.

Vale destacar que esse erro não ocorre por recebimento de dados nulos, pois os dados são adaptados quando recebidos, e nem por receber uma lista vazia durante uma busca por filmes, pois quando a busca por filmes não retorna nenhum filme, a lista de destaque é recuperada.

#### 4.1.2.3 Nova tentativa de requisição ou atualização dos dados



(a) Erro após tentativa de requisição.

(b) Nova requisição.

Figura 13: Erro e sucesso após duas tentativas de comunicação com o servidor

Na imagem (a) temos a tela de erro após uma tentativa de comunicação com o serviço, e também uma ação de atualização de página. Quando há um erro na página de destaque e o usuário atualiza a página, é realizado uma nova requisição da última página que foi recebida com sucesso. Após a segunda tentativa, o resultado é o obtido na imagem (b).

```
override fun onRefresh() {
    filmViewModel.setPresentation()
    super.onRefresh()
}
```

(a) Código - Atualizando a página

```

    override fun setPresentation (page: Int) {
        val newPage = super.validatePage(page)
        if (_query == "")
            requestPopularFilmList(newPage)
        else
            requestSearchedFilmList(newPage)
    }

```

(b) Código - Realizando requisição

Figura 14: Código - Atualizando página da lista de filmes

#### 4.1.3 Ação dos Componentes Arquiteturais ao selecionar um filme

Ao selecionar um filme da lista recebida pela base de dados remota, uma nova *Activity* é solicitada para apresentar as informações do filme. De antemão, essa nova *Activity* recebe o ID do filme, e também é informada se o filme selecionado está armazenado localmente no dispositivo. O componente responsável em guardar os dados desses filmes no dispositivo é a biblioteca *Room*. Ao selecionar o filme, uma busca é feita na base de dados local, e caso o filme seja encontrado, é preferível que o filme seja aberto localmente, para evitar consumo de dados de internet, e também para que o filme possa ser aberto mesmo que o usuário não esteja conectado no momento da seleção. Após a nova *Activity* entrar em execução, ela deve se comunicar com o servidor e requisitar as informações do filme caso o esse não esteja armazenado localmente.

```

intent.putExtra( name: "filmId", filmId)
val isFavorite: Boolean = filmViewModel.isFavoriteFilm(filmId!!)
intent.putExtra( name: "favorite", isFavorite)

```

Figura 15: Código - Verificando se o filme está na base de dados local

A *Activity* que exibe os detalhes do filme tem um *ViewModel* particular, é capaz de requisitar ao servidor, a informação de um único filme apenas com o ID, e salvar essas informações dentro de um objeto<sup>9</sup> do tipo *LiveData*. Aqui ocorre a mesma coisa que acontece com a lista, o objeto do tipo *LiveData* notifica o observador, que nesse caso, está dentro da *Activity* que exibe os detalhes do filme. Essa notificação também ocorre caso as informações sejam recuperadas localmente, pois ambos os métodos salvam as informações no mesmo objeto. Após recuperar as informações, os dados são adaptados e aplicados na tela de detalhes.

---

<sup>9</sup> Objeto Film existente no projeto, e contém apenas variáveis relacionadas a filme.

```

val getCallDeferred = FilmsApi.retrofitService.callFilmApi(filmId)
try {
    val requestResult = getCallDeferred.await()
    requestFilm.value = requestResult
}

```

(a) Código - Requisitando Filme

```

viewModel.responseFilmJson.observe(owner: this, Observer { it: FilmsJson.FilmJson!
    viewModel.setFilm(FilmAdapter.adaptJsonToData(it))
    submitDetails(viewModel.film!!)
})

```

(b) Código - Observador a espera da notificação

Figura 16: Código - Objeto do tipo *LiveData* recebe os dados e notifica observador

#### 4.1.3.1 Abrindo o Filme a partir da base de dados remota

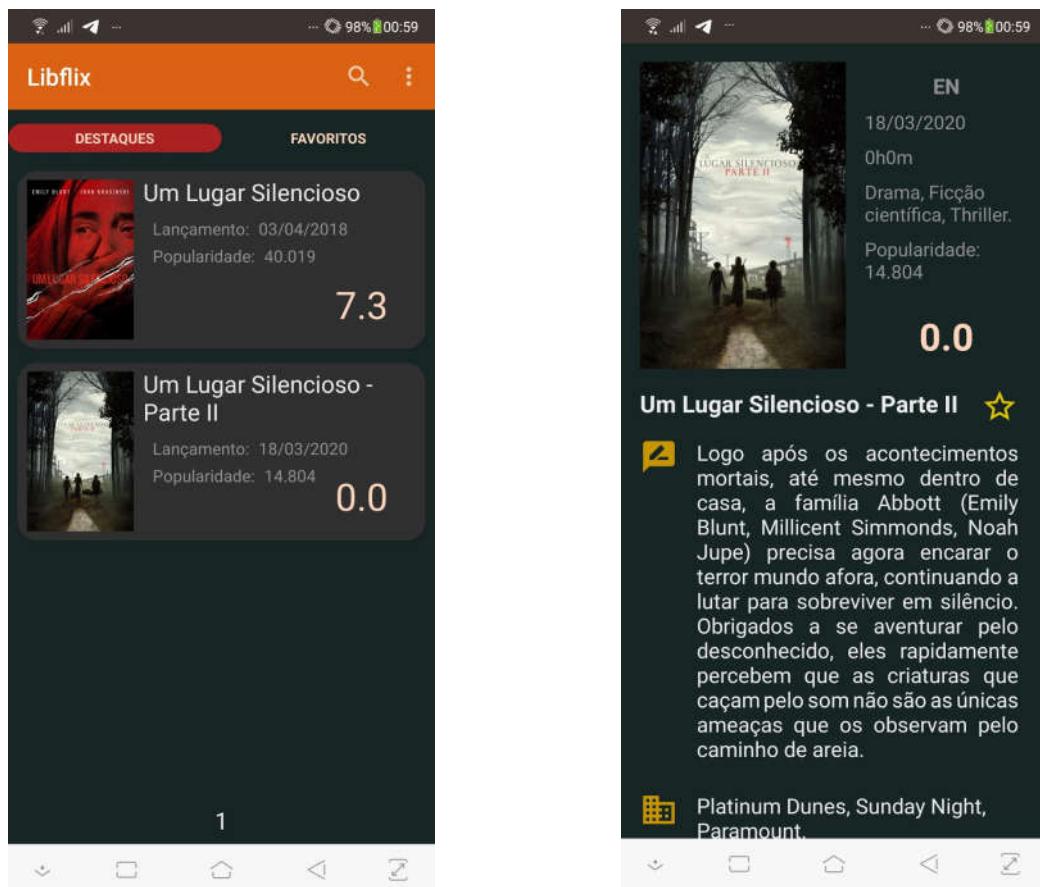
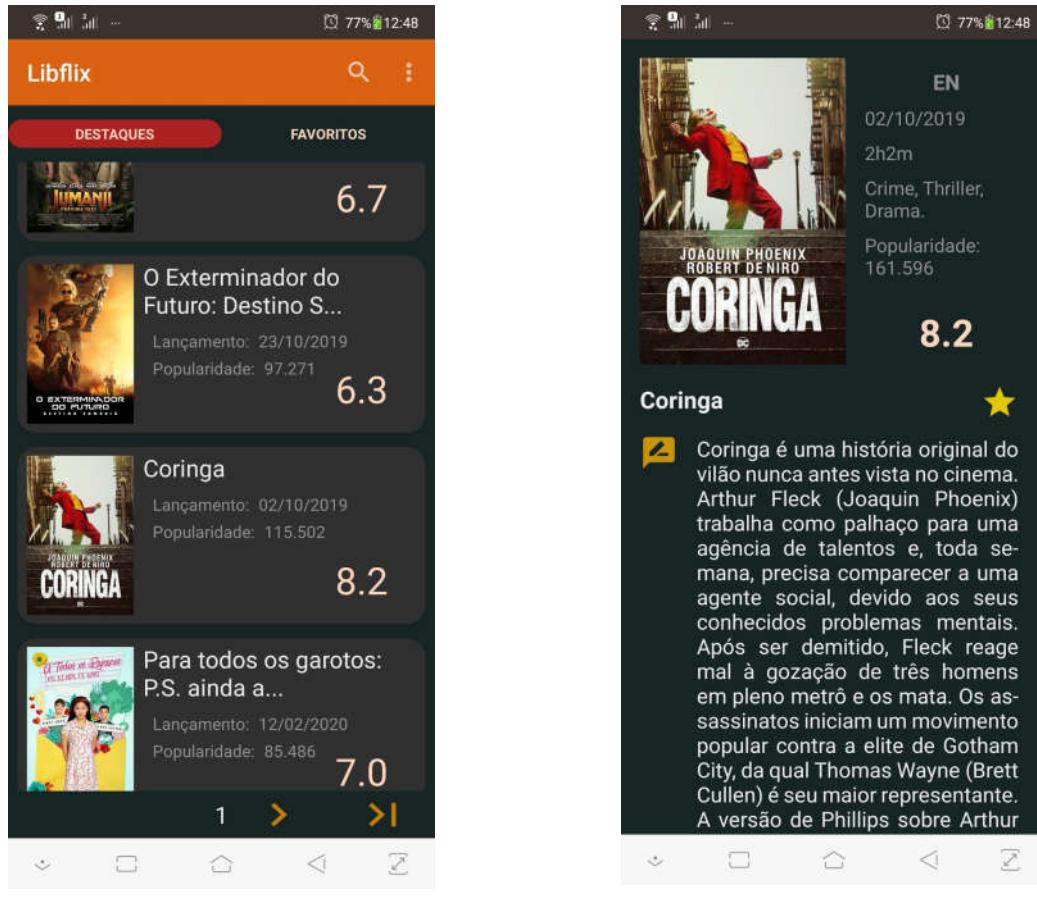


Figura 17: Selecionando filme e requisitando dados na base de dados remota

Na imagem (a), após o usuário tocar no cartão do filme que deseja abrir, uma nova tela será aberta, e assim que as informações forem recuperadas, elas serão exibidas, como

representada na imagem (b). Nesse caso, as informações do filme foram requisitadas e obtidas a partir da base de dados remota, pois o filme não está disponível localmente.

#### 4.1.3.2 Abrindo o Filme a partir da base de dados local



(a) Selecionando um filme

(b) abrindo informações do filme.

Figura 18: Selecionando filme e abrindo o filme a partir da base de dados local

Na imagem (a), a situação é similar a anterior, onde o usuário toca no cartão do filme, de modo que a nova tela é aberta na imagem (b), mas os dados são recuperados a partir da base de dados local. Essa ação acontece tanto na aba “DESTAQUES”, quanto na aba “FAVORITOS”, desde que o filme esteja armazenado na memória do dispositivo. Na imagem (b) é possível ver que a estrela de favorito está preenchida, sendo assim, é um filme favorito do usuário.

#### 4.1.4 Ação dos Componentes Arquiteturais ao salvar um filme localmente

Ao tocar na estrela de favoritos na *Activity* de detalhes, nenhuma ação de na base de dados é realizada, para evitar ações que podem ser desnecessárias ou repetidas. Uma ação desnecessária seria, um caso em que o usuário toca na estrela para marcar o filme com favorito, e depois toca novamente para desmarcá-la, pois isso resultaria em um adição e remoção do mesmo filme na base de dados local.

A *Activity* de detalhes, sempre no início do seu ciclo de vida, envia a sua *LifeCycle* para a sua *ViewModel* particular, e por fim, essa *ViewModel* passa a conhecer o ciclo de vida da *Activity* detalhes. A *ViewModel* precisa ser um observador do ciclo de vida, para que ele possa ser adicionado como observador do *LifeCycle* da *Activity*. A partir desse ponto, a *ViewModel* pode realizar eventos em cada estado da *Activity*. Então, para evitar desperdício de processamento, sempre que a *Activity* entra no estado parado (*onStop*), a *ViewModel* é notificada, e solicita a ação de adicionar ou remover o filme no banco de dados dos filmes favoritos, o qual a biblioteca *Room* é a responsável em realizar essa ação.

```
class FilmDetailsViewModel (private val databaseDao: FilmDao, app: Application) :  
    AndroidViewModel(app),  
    LifecycleObserver
```

(a) Código - *ViewModel* é um Observador do ciclo de vida da *Activity*

```
private var _lifecycle: Lifecycle? = null  
val lifecycle: Lifecycle? get() = _lifecycle  
fun setLifecycle(lf: Lifecycle) {  
    _lifecycle = lf  
    _lifecycle?.addObserver(observer: this)  
}
```

(b) Código - método *get* e *set* para o *LifeCycle* da *Activity*

```
viewModel.setLifecycle(this.lifecycle)
```

(c) Código - *Activity* “seta” o seu ciclo de vida na variável da *ViewModel*

```

@OnLifecycleEvent(Lifecycle.Event.ON_STOP)
fun onResultDetailsActivity () {
    _film?.let { it: FilmData ->
        val inDatabase = this.isFavoriteFilm(it.id)
        if ( _isFavorite!! && !inDatabase ) {
            insertFilm(it)
        }
        else if ( !_isFavorite!! && inDatabase) {
            val deleteFilm = this.getFilm(it.id)
            deleteFilm(deleteFilm)
        }
    }
}

```

(d) Código - Evento realizado quando a *Activity* passa pelo estado de parada

Figura 19: Código - Verificar se é preciso deletar ou salvar filme na base de dados local

## 4.1.4.1 Salvando normalmente

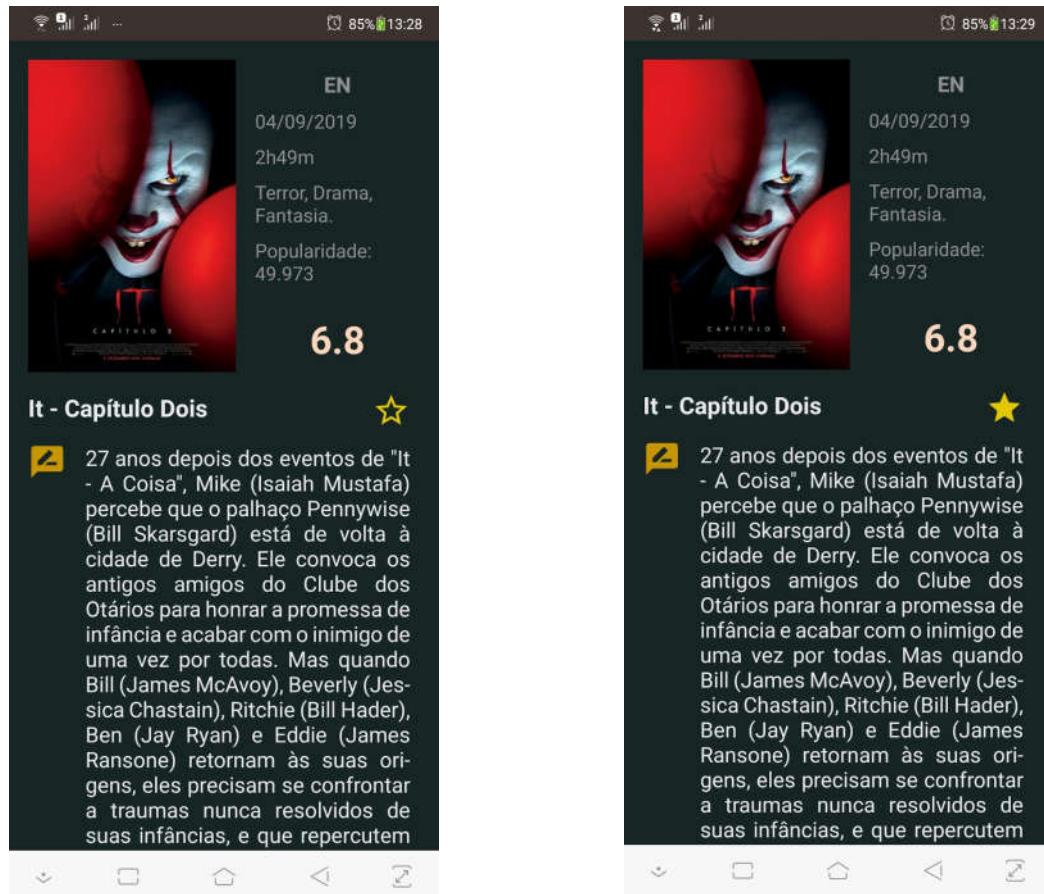


Figura 20: Marcando um filme como favorito

Após carregar as informações do filme na tela, é mostrado na imagem (a) que o filme recuperado não é favorito, e portanto, foi feito uma requisição ao servidor. Posteriormente o filme foi marcado como favorito, mas ainda não adicionado ao banco de dados de filmes favoritos. A ação de salvar as informações do filme no armazenamento do dispositivo só será realizada após o fechamento da tela de detalhes (nesse exemplo).

#### 4.1.4.2 Salvando em situações inesperadas

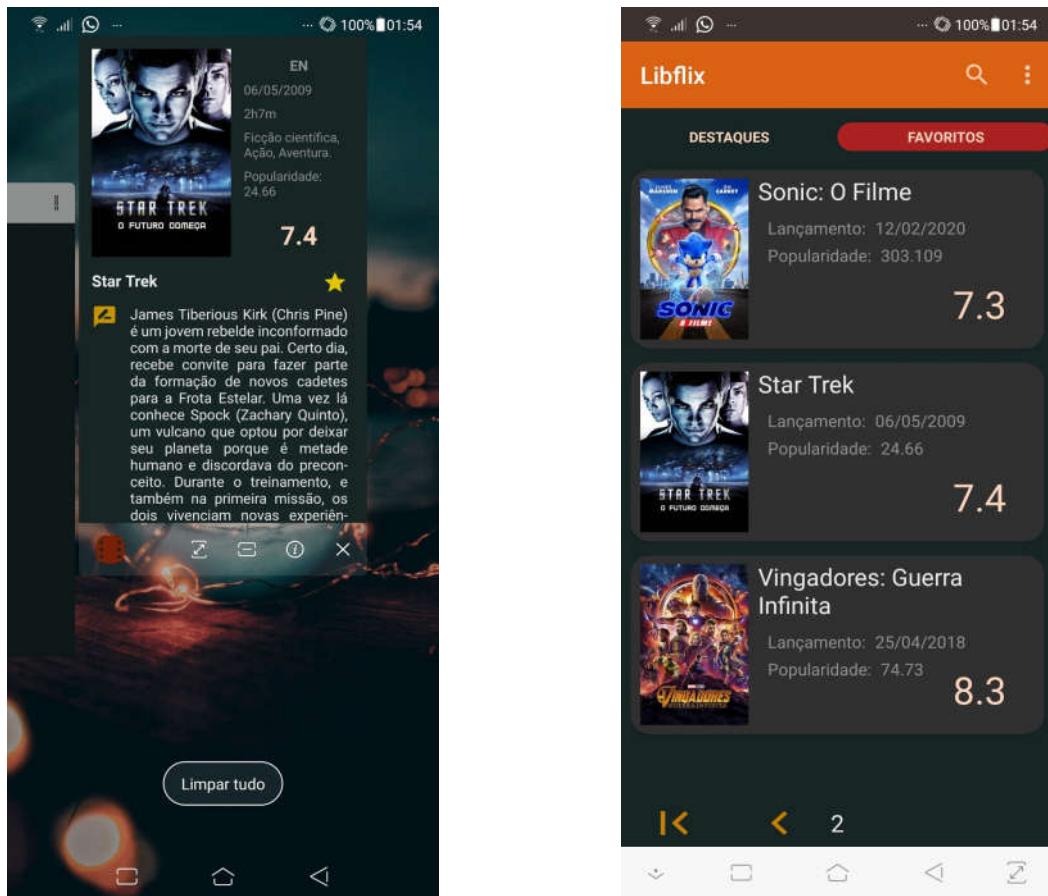


Figura 21: Marcando filme como favorito, e fechando aplicativo posteriormente

Durante essa ação, o filme foi marcado favorito normalmente, mas nesse caso a aplicação foi fechada antes de o usuário voltar para a tela que apresenta a lista de filmes, como visto na imagem (a). Em situações onde o armazenamento dessas informações, seriam salvos apenas quando o usuário saísse da tela que exibe os detalhes do filme (através do método *onActivityResult*), o usuário voltaria posteriormente ao aplicativo o filme não estaria

no banco de dados de filmes favoritos. Nessa situação, o filme foi armazenado com sucesso como mostra a imagem (b).

Essa ação pode ser muito útil quando a tela do aplicativo é colocada em segundo plano devido a interrupções. Um exemplo que pode ser usado é, quando a aplicação é colocada em segundo plano devido o recebimento de uma chamada telefônica. Pode ocorrer de o usuário não abrir novamente a aplicação de imediato, todavia nesse caso, os dados do filme já estão armazenados em segurança.

#### **4.1.5 Ação dos Componentes Arquiteturais ao buscar filmes**

A busca de filmes nos dois *Fragments* ocorre de maneira um pouco diferente, mas o comportamento dos componentes é similar. O *Fragment* que controla a interface que exibe os filmes da base de dados remota, busca os filmes apenas quando o usuário submete a palavra, enquanto o que controla a interface que exibe os filmes da base de dados local, busca os filmes a cada nova letra digitada. No *Fragment* que controla a exibição da lista da base de dado remota, o restante das ações ocorre de maneira similar ao que foi descrito na seção 4.1.2.1. Para o *Fragment* que controla a exibição da lista da base de dados local, o filme é buscado através da biblioteca *Room*, realizando a busca pelo nome do filme, recuperando os dados e aplicando-os na lista de apresentação do tipo *LiveData*, guardada na *ViewModel*. Em seguida, o observador presente no *Fragment* é notificado, realizando as ações necessárias para exibir a nova lista.

#### 4.1.5.1 Buscar filme na base de dados remota

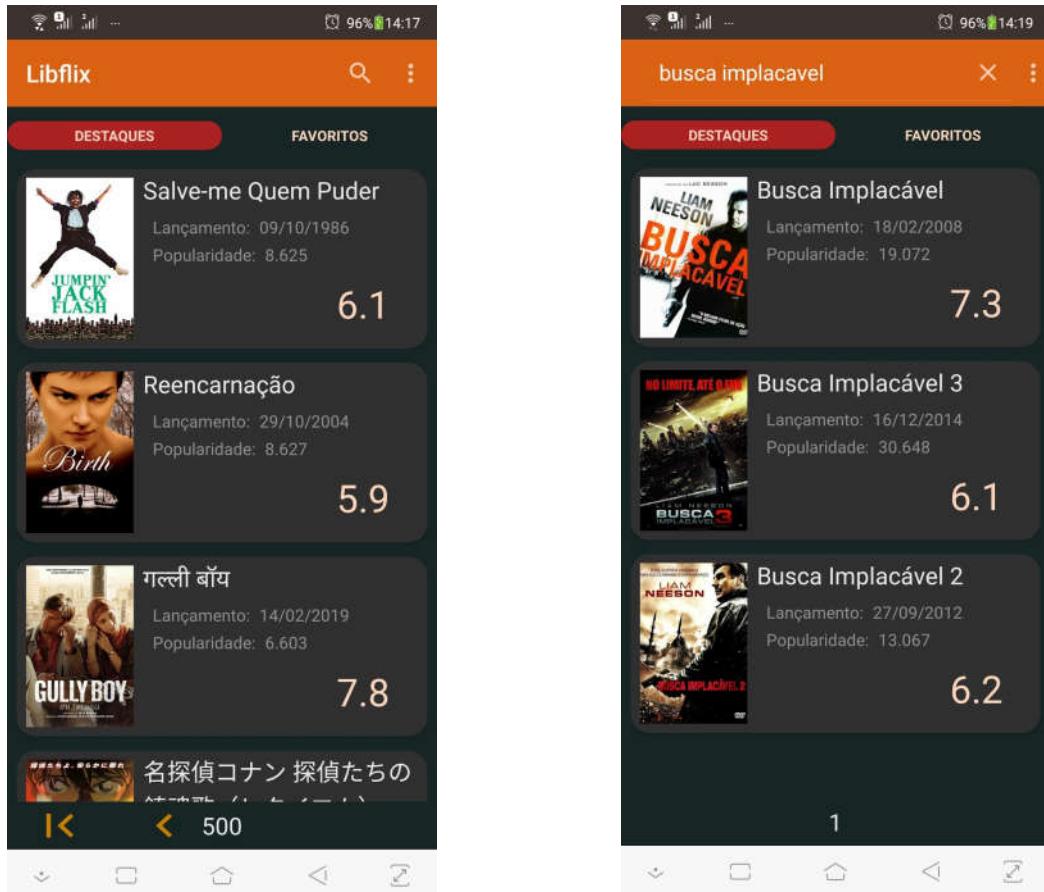


Figura 22: Buscando por filmes na base de dados remota

O usuário pode não encontrar o filme que deseja ao procurar na página de destaques. Além disso, seria demorado e cansativo ir de página em página a procura do filme que deseja encontrar. Para isso ele pode usar a função de busca na barra superior, ao lado do nome da aplicação, como mostra a figura (a). Após digitar e submeter o que deseja buscar, o servidor retorna com as resposta, como pode-se ver na imagem(b).

```
override fun onQueryTextSubmit(query: String?): Boolean {
    filmViewModel.setSearch(query!!)
    filmViewModel.setPresentation(1)
```

(a) Código - Submetendo busca

```

private fun requestSearchedFilmList (page: Int = actualPage) {
    uiCoroutineScope.launch { this: CoroutineScope {
        val getCallDeferred = FilmsApi.retrofitService.callSearchMovieList(page, query)
        setRequestResult(getCallDeferred)
    }
}

```

(b) Código - Realizando busca na API por *query* submetida

Figura 23: Código - Realizando busca por filmes na API

#### 4.1.5.2 Buscar filme na base de dados local

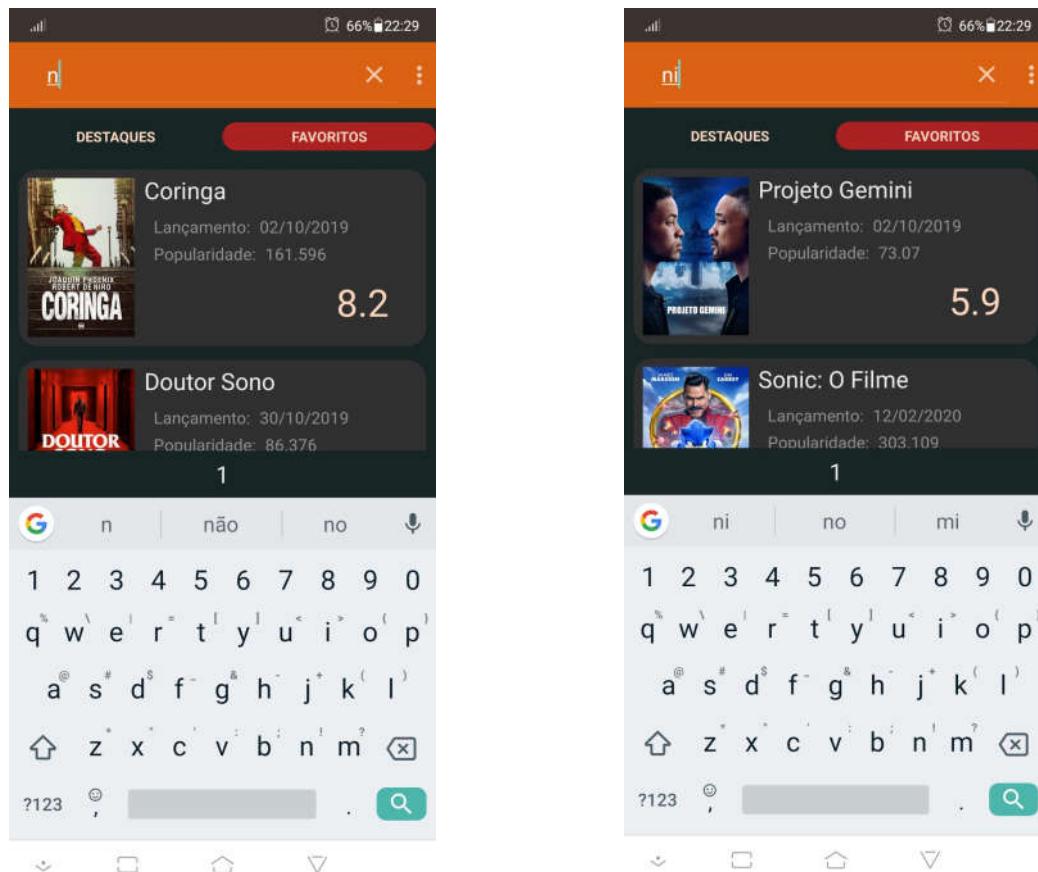


Figura 24: Realizando uma busca na base de dados local

Para buscar filmes localmente, é preciso apenas tocar na lupa na barra superior e começar a digitar. A cada nova letra digitada, a lista que apresenta os filmes favoritos é atualizada, não sendo necessário submeter a palavra.

```
override fun onQueryTextChange(newText: String?): Boolean {
    if (newText != "") {
        filmViewModel.searchFilmDatabase(newText!!)
    }
}
```

(a) Código - Realizando busca sempre que texto da busca for alterado

```
fun searchFilmDatabase (query: String) {
    uiCoroutineScope.launch { this: CoroutineScope
        filmsDatabase?.value?.let { it: List<FilmData>
            val newList: MutableList<FilmData>? = mutableListOf()
            for (i in 0 until it.size) {
                if ( it[i].title.toLowerCase().contains(query.toLowerCase()) )
                    newList?.add(it[i])
            }
            mementoPresentationFilmList?.value = newList
            setPresentation(page = 1)
        }
    }
}
```

(b) Código - Buscando por filmes na base de dados local

Figura 25: Código - Realizando busca e aplicando nova lista na lista de apresentação

## 4.2 APLICANDO MICRO FRONTENDS

Visto os benefícios que *Micro Frontends* deve trazer ao ser introduzido seus conceitos na arquitetura da aplicação, o *front-end* da *Activity* que exibe as listas de filmes, foi desenvolvido usando técnicas para reaproveitar componentes, tornando essa *Activity*, um conjunto de outros componentes menores, que podem ser melhor gerenciados.

A separação de componentes dividiu a *Activity* que exibe a lista de filmes em 5 partes, que podem ser separadas em grupos. A figura abaixo ilustra a *Activity* que exibe a lista de filmes da base de dados remota, e como cada componente de interface está separado, a qual, cada caixa colorida possui uma cor diferente, representando componentes de interface individualmente. A caixa amarela representa o todo da *Activity*, que deve gerenciar, implantando ou removendo outros componentes. A caixa azul próxima ao topo, gerencia o menu, contendo a busca e outras opções de menu que podem ser implementadas futuramente. A caixa na cor roxa, que contém uma caixa menor na cor verde, é responsável por controlar e apresentar a lista de filmes, a qual deve replicar vários componentes da caixa verde, que devem ser exibidas na vertical; enquanto a verde que está contida nela, exibe o cartão do filme com informações básicas. E por fim, a caixa na cor vermelha, no rodapé da *Activity*, é o componente de interface que gerencia as páginas de filmes que são apresentadas ao usuário.



Figura 26: Componentização da *Activity*

Ao usar esses conceitos, tornou-se possível a reutilização desses componentes em outros *front-ends* do sistema, que neste caso, além de serem utilizados para exibir a lista de filmes recuperadas no base de dados remota, também foram reutilizados para exibir a lista de filmes da base de dados local. O *Fragment* que exibe a lista tem a *RecyclerView* como um *front-end*, além do rodapé, que é inserido no *Fragment* através do *include*. O *adapter\_film\_list.xml* é quem dá forma aos cartões que são exibidos na *RecyclerView*.

```

<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/film_list_recycle_view"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:scrollbars="vertical"
        app:layout_constraintBottom_toTopOf="@+id/activity_main_toolbar"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <include
        android:id="@+id/activity_main_toolbar"
        layout="@layout/page_footer"

```

Figura 27: fragment\_film\_list.xml usado pelos *Fragments* que exibem a lista de filmes

A *Activity* principal é apenas responsável por controlar o menu e inflar os *Fragments* que exibem as listas. Nesse caso, os *Fragments* contidos na *Activity* principal controlam as caixas na cor roxa, verde e vermelha, enquanto a *Activity*, controla a caixa amarela e azul. Vale salientar que a *Activity* e os *Fragments*, estão acoplando os componentes de interface, ou pequenos *front-ends*, usando *ConstraintLayout*, para tornar a tela responsiva e adaptável em diversos dispositivos.

```

    tools:context=".presentation.activity.MainActivity">

    <include
        android:id="@+id/activity_main_toolbar"
        layout="@layout/toolbar"

```

Figura 28: activty\_main.xml *Activity* principal contém um toolbar.xml em seu *front-end*

#### 4.3 TESTE DA APLICAÇÃO

Para realização de testes dos componentes, foi utilizado a ferramenta de Testes da Robolectric<sup>10</sup> em sua versão 4.0.2. Para fins de estudo, apenas o componente *Room* foi testado, usando as classes **FilmDao** e **FilmDatabase** que compõem esse componente.

---

<sup>10</sup> link para mais informações disponível em: <http://robolectric.org/>

```

@Config(manifest = "src/main/AndroidManifest.xml", sdk = [26])
@RunWith(RobolectricTestRunner::class)
class FilmDatabaseTest {

    private lateinit var filmDao: FilmDao
    private lateinit var db: FilmDatabase

```

Figura 29: Código - Ferramenta de testes e componentes testados

Ainda na parte do código, as notações `@Before`, `@After` e `@Test` do framework JUnit foram usados para organizar a ordem de execução dos testes. Para realizar os testes, o princípio foi definido o que deve ser realizado antes e depois deles, através das notações `@Before` e `@After`. Antes do testes o banco de dados deve ser criado, e ao final dos testes ele deve ser fechado.

```

@Before
fun createDb() {
    val context = InstrumentationRegistry.getInstrumentation().targetContext
    db = Room.inMemoryDatabaseBuilder(context, FilmDatabase::class.java)
        .allowMainThreadQueries()
        .build()
    filmDao = db.filmDao
}

```

(a) Código - notação para executar escopo da função antes dos testes

```

@After
@Throws(IOException::class)
fun closeDb() {
    db.close()
}

```

(b) Código - notação para executar trecho de código ao final dos testes

Figura 30: Código - Funções chamadas antes de iniciar e ao finalizar os testes.

Para os testar o componente *Room*, foram criados três testes; uma função para inserir e recuperar o filme, outra para inserir um filme e limpar a base de dados em seguida, e por fim uma função para inserir e atualizar os dados de um filme do database.



```

@Test
@Throws(Exception::class)
fun insertAndGetFilm() {
    val film = FilmData(id = 200L)
    filmDao.insertFilm(film)
    val getFilm = filmDao.get(200L)
    assertEquals(getFilm.id, film.id)
}

```

Figura 31: Código - teste do *insert* e *get*

A função **insertAndGetFilm()** da figura 32, insere um filme com ID igual a 200 através do *insert*, em seguida o filme com ID igual a 200 é recuperado, e posteriormente verificado que as ações foram realizadas com sucesso.



```

@Test
@Throws(Exception::class)
fun insertAndClearDatabase() {
    val film = FilmData(id = 2L)
    filmDao.insertFilm(film)
    assertEquals(filmDao.getFilmsCount(), actual: 1)
    filmDao.clear()
    assertEquals(filmDao.getFilmsCount(), actual: 0)
}

```

Figura 32: Código - Teste do *clear*

A função **insertAndClearDatabase()** da figura 33, insere um filme, verifica se a quantidade de filmes na base de dados local é igual a 1, em seguida a base de dados é limpada utilizando o *clear*, e posteriormente verificado que a quantidade de itens se tornou igual a 0 com sucesso.



```

@Test
@Throws(Exception::class)
fun insertAndUpdateFilm() {
    val film = FilmData(id = 20L, title = "SEM TITULO")
    filmDao.insertFilm(film)
    assertEquals(filmDao.get(20L).title, actual: "SEM TITULO")
    val getFilm = filmDao.get(20L)
    getFilm.title = "COM TITULO"
    filmDao.updateFilm(getFilm)
    assertEquals( filmDao.get(20L).title, actual: "COM TITULO" )
}

```

Figura 33: Código - Teste do *update*

A função **insertAndUpdateFilm()** da figura 34, insere um filme na base de dados local com ID igual a 20 e título igual a “SEM TITULO”, em seguida verificou-se que o filme

foi inserido corretamente. Para completar o teste, então recuperou-se o filme, alterou o seu título para “COM TITULO”, em seguida o *update* foi utilizado e posteriormente foi verificado que o título do filme existente na base de dados foi atualizado corretamente.

O componente *Room* mostrou-se uma biblioteca muito eficaz, capaz de auxiliar na componentização da aplicação, e que também é facilmente testado. Fato é que as ferramentas de testes no Android estão em constantes atualizações, e que essas atualizações podem trazer problemas durante a criação dos testes, todavia, o ponto principal é mostrar que o componente testado é testável, e sem a necessidade da utilização de outros componentes que estão em módulos mais acima na arquitetura apresentada na figura 4, a qual foi abordada e utilizada nesta aplicação.

## 5 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi apresentado conceitos básicos da arquitetura Android, e tem como objetivo principal definir e apresentar novos componentes arquiteturais, capazes de criarem uma aplicação componentizada e melhor construída, em conjunto com *Micro Frontends*.

Os componentes de arquitetura *ViewModel*, *LiveData*, *LifeCycle* *Room* mostraram-se capazes tornar uma aplicação Android componentizada e testável, características indispensáveis para criação de grandes aplicações. Atualmente tem sido muito importante tornar toda a aplicação componentizada, para melhorar a divisão de tarefas entre equipes, atualizar partes de um componente sem interferir o todo, e escalar aplicações de forma organizada e com mais segurança. Além disso, esses componentes se adequaram perfeitamente as técnicas de Micro frontends para composição da interface de usuário, garantido que a aplicação principal se tornasse modular e componentizada.

O aplicativo criado para testar os componentes de arquitetura Android, foi o aplicativo Libflix, e ainda não está disponível para download na loja de aplicativos Android, pois sua principal finalidade é demonstrar e testar a capacidade desses componentes citados.

Alguns pontos já foram identificados para futuras melhorias deste trabalho e da aplicação. As principais são:

- Comparar a arquitetura do trabalho com a arquitetura de componentização Android da Netflix .
- Definir e implementar o conceito de Single Activity com o componente Navigation.
- Implementar e liberar o acesso a um novo *front-end* que apresenta uma lista de séries de TV, também construído utilizando técnicas de *Micro Frontends*.
- Implementar uma base de dados remota para cadastro, login e backup de dados dos usuários.
- Realizar testes unitários em outros componentes de arquitetura Android.

## REFERÊNCIAS

- [1] DEITEL P.; DEITEL H.; DEITEL A. **Android: Como Programar**, 2 ed. Editora Bookman, 2015.
- [2] Android Developers: *Guide to App Architecture*. Disponível em: <<https://developer.android.com/jetpack/docs/guide>>. Acesso em: 24 out. 2019.
- [3] Android Developers: *Arquitetura da Plataforma*. Disponível em: <<https://developer.android.com/guide/platform>>. Acesso em: 24 out. 2019.
- [4] Android Developers: *Application Fundamentals*. Disponível em <<https://developer.android.com/guide/components/fundamentals.html>>. Acesso em: 25 out. 2019.
- [5] LOU, T. *A comparison of Android Native App Architecture - MVC, MVP and MVVM*. 2016, 57f. Tese de Mestrado - Universidade de Tecnologia de Eindhoven. Disponível em: <[https://pure.tue.nl/ws/portalfiles/portal/48628529/Lou\\_2016.pdf](https://pure.tue.nl/ws/portalfiles/portal/48628529/Lou_2016.pdf)>. Acesso em 4 dez. 2019.
- [6] Android Developers: *Handling Lifecycles with Lifecycle-Aware Components*. Disponível em: <<https://developer.android.com/topic/libraries/architecture/lifecycle>>. Acesso em: 27 out. 2019.
- [7] Android Developers: *ViewModel Overview*. Disponível em: <<https://developer.android.com/topic/libraries/architecture/viewmodel>>. Acesso em: 28 out. 2019.
- [8] Android Developers: *Visão Geral do LiveData*. Disponível em: <<https://developer.android.com/topic/libraries/architecture/livedata>>. Acesso em: 28 out. 2019.
- [9] Android Developers: *Salvar dados em um banco de dados local usando o Room*. Disponível em: <<https://developer.android.com/training/data-storage/room/index.html>>. Acesso em: 28 out. 2019.
- [10] SINGH, Rajinder. An Overview of Android Operating System and Its Security Features. **Int. Journal of Engineering Research and Applications**. www.ijera.com, ISSN : 2248-9622, Vol. 4. Fevereiro, 2014, p. 519-521. Disponível em:

- <<https://pdfs.semanticscholar.org/11f4/b8efd1a9af746f17ac5e8d6a789bd3c3a9b7.pdf>>.  
Acesso em: 25 out. 2019.
- [11] Android Developers: *Activity*. Disponível em:  
<<https://developer.android.com/reference/android/app/Activity>>. Acesso em: 9 nov. 2019.
- [12] Android Developers: *Fragments*. Disponível em:  
<<https://developer.android.com/guide/components/fragments>>. Acesso em: 9 nov. 2019.
- [13] Android Developers: *Service*. Disponível em:  
<<https://developer.android.com/reference/android/app/Service.html>>. Acesso em: 12 nov. 2019.
- [14] Android Developers: *Provedores de Conteúdo*. Disponível em:  
<<https://developer.android.com/guide/topics/providers/content-providers.html>>. Acesso em: 12 nov. 2019.
- [15] Android Developers: *Componentes da Arquitetura do Android*. Disponível em:  
<<https://developer.android.com/topic/libraries/architecture>>. Acesso em: 8 nov. 2019.
- [16] Android Developers: *Intents e filtros de Intents*. Disponível em:  
<<https://developer.android.com/guide/components/intents-filters>>. Acesso em: 30 nov. de 2019.
- [17] Android Developers: *Como direcionar o usuário para outro aplicativo*. Disponível em:  
<<https://developer.android.com/training/basics/intents/sending>>. Acesso em: 30 nov. 2019.
- [18] INTERNATIONAL CONFERENCES PervasivE PATTERNS AND APPLICATION  
ANDROID PASSIVE MVC, 15, 2013. *Android Passive MVC: a Novel Architecture  
Model for Android Application Development...* Universidade de Tecnologia de Troyes.  
Disponível em:  
<<https://pdfs.semanticscholar.org/3ffa/594333883a56fc0519072b6615600ec03708.pdf>>.  
Acesso em: 2 dez. 2019.
- [19] Android Developers: *What the Fragment? - Google I/O 2016*. Disponível em:  
<<https://youtu.be/k3IT-IJ0J98>>. Acesso em: 5 dez. 2019.
- [20] Android Developers: *Single Activity: Why, When, and How (Android Dev Summit '18)*.  
Disponível em: <<https://youtu.be/2k8x8V77CrU>>. Acesso em: 5 dez. 2019.

- [21] Vasiliiy: *Netflix Shows The Future of Android Architecture*. Disponível em: <<https://www.techyourchance.com/netflix-shows-the-future-of-android-architecture/>>. Acesso em 10 dez. 2019.
- [22] droidcon SF: *droidcon SF 2018 - Netflix's componentization architecture with RxJava + Kotlin - Part II*. Disponível em: <[https://youtu.be/1cWwfh\\_5ZQs](https://youtu.be/1cWwfh_5ZQs)>. Acesso em 10 dez. 2019.
- [23] Michael Geers: *Micro Frontends - Extending the microservice idea to frontend development*. Disponível em: <<https://micro-frontends.org>>. Acesso em: 9 fev. 2020.
- [24] Cam Jackson: *Micro Frontends*. Disponível em: <<https://martinfowler.com/articles/micro-frontends.html>>. Acesso em 9 fev. 2020.
- [25] GitHub: ednaldomartins - ArchitectureComponentApp. Diponível em: <<https://github.com/ednaldomartins/ArchitectureComponentApp>>. Acesso em 30 Jan. 2020.
- [26] The Movie Database: API Overview. Disponível em: <<https://www.themoviedb.org/documentation/api>>. Acesso em 15 dez. 2019.
- [27] Robolectric: *test-drive your Android code*. Disponível em: <<http://robolectric.org>>. Acesso em 13 mar. 2020.