CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO ITAJAÍ - UNIDAVI

LUIZ ARTUR BOING IMHOF

PROTÓTIPO DE APLICATIVO PARA A BUSCA E REGISTRO DE DESCONTOS EM LOCAIS DE ALIMENTAÇÃO E ENTRETENIMENTO

RIO DO SUL
2020

CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO ITAJAÍ - UNIDAVI

LUIZ ARTUR BÖING IMHOF

PROTÓTIPO DE APLICATIVO PARA A BUSCA E REGISTRO DE DESCONTOS EM LOCAIS DE ALIMENTAÇÃO E ENTRETENIMENTO

Trabalho de Conclusão de Curso a ser apresentado ao curso de Sistemas da Informação, da Área das Ciências Naturais, da Computação e das Engenharias, do Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí, como condição parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Prof. Orientador: M.e Marcondes Maçaneiro

RIO DO SUL
2020

CENTRO UNIVERSITÁRIO PARA O DESENVOLVIMENTO DO ALTO VALE DO ITAJAÍ - UNIDAVI

LUIZ ARTUR BOING IMHOF

PROTÓTIPO DE APLICATIVO PARA A BUSCA E REGISTRO DE DESCONTOS EM LOCAIS DE ALIMENTAÇÃO E ENTRETENIMENTO

Trabalho de Conclusão de Curso a ser apresentado ao curso de Sistemas da Informação, da Área das Ciências Naturais, da Computação e das Engenharias, do Centro Universitário para o Desenvolvimento do Alto Vale do Itajaí- UNIDAVI, a ser apreciado pela Banca Examinadora, formada por:

_____

Professor Orientador: M.e Marcondes Maçaneiro

Banca Examinadora:

_____

Professor M.e Fernando Andrade Bastos

_____

Professor M.e Jeancarlo Visentainer

Rio do Sul, 03 de Dezembro de 2020.

Dedico este trabalho aos meus pais, Raul e Éden, a minha irmã, Iris e a minha namorada Sarah, que me apoiaram e me fizeram chegar até aqui.

AGRADECIMENTOS

Agradeço aos meus pais Raul e Éden, por permitirem estudar para chegar onde estou, por sempre apoiarem minhas escolhas e acreditarem em mim.

Agradeço a minha irmã Iris, que foi, e sempre será meu exemplo.

Agradeço a minha namorada Sarah, pelo amor, paciência e ajuda, em especial durante o período de desenvolvimento deste trabalho.

Agradeço também a todos os professores que contribuíram com a minha formação, em especial ao meu orientador Marcondes e aos membros da banca Fernando Bastos e Jeancarlo, pois sem eles o desenvolvimento deste projeto não seria possível.

E por último o meu agradecimento aos meus familiares e todos amigos que passaram pela minha vida, pois cada um do seu jeito, fizeram parte do meu desenvolvimento colaborando para que esta etapa fosse possível.

RESUMO

Hoje em dia, a maioria da população brasileira tem um dispositivo móvel, e o utiliza como apoio para decisões diárias. Entre elas onde ir para se alimentar ou se divertir, e também para economizar o máximo possível sem abrir mão dessas atividades. O intuito deste trabalho foi desenvolver um protótipo de um aplicativo que auxilie o usuário a chegar nesse objetivo de encontrar um local próximo que tenha descontos atrativos para o cliente. Para o desenvolvimento do protótipo foi utilizado a linguagem Kotlin e bibliotecas disponibilizadas pela Google especificamente para o desenvolvimento na plataforma Android. As bibliotecas escolhidas foram selecionadas com o propósito de utilizar a melhor tecnologia disponível atualmente para garantir que a aplicação não fique obsoleta e que seja sempre atualizada para manter-se relevante. A aplicação armazena as informações em um banco de dados SQLite utilizando a biblioteca Room, disponibilizada no conjunto de bibliotecas e boas práticas de programação da Android *Jetpack.* Além disso as telas foram pensadas utilizando conceitos de usabilidade e baseado no *Material design*, que é um dos principais padrões para aplicações móveis atualmente. Ao final do desenvolvimento do protótipo foi analisado o resultado e comparado com os objetivos, com isso foi verificado uma possibilidade de trabalhos futuros que venham a melhorar o protótipo e possivelmente levar o desenvolvimento adiante.

Palavras-Chave: Android, Kotlin, aplicativo, desconto, alimentação, entretenimento.

ABSTRACT

Nowadays, most of the Brazilian population has a mobile device, and uses it as a support for daily decisions, including where they choose to eat or where they go to have fun, saving as much Money as possible without giving up these activities. The purpose of this project was to develop a prototype of an application that helps the user to reach this goal of finding a nearby location that has attractive discounts for the customer. The prototype was developed using the Kotlin language and libraries made available by Google that are intended specifically for Android development. The chosen libraries were selected with the purpose of using the best technology currently available to ensure that the application don't become obsolete and that it is always updated to remain relevant. The application stores the information in a SQLite database using the Room library, available as part of Android Jetpack, wich is a set of libraries and programing patterns. In addition to that, the screens were designed using usability concepts and based on Material design, which is one of the standards for mobile applications. At the end of the prototype development, the results were analyzed and compared with the objectives, revealing a possibility for future works that could improve the prototype and possibly carry out the development of the application.

Keywords: Android, Kotlin, application, discount, food, entertainment.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE ABREVIATURAS E SIGLAS

AOT	*Ahead of Time*
APK	Android *Application Pack*
APP	Aplicativo
ART	Android *Runtime*
CNDL	Confederação Nacional de Dirigentes Lojistas
DAO	*Data Access Objects*
HAL	Camada de Abstração de Hardware
IBGE	Instituto Brasileiro de Geografia e Estatística
IDE	Ambiente de desenvolvimento integrado
IU	Interface de Usuário
JIT	*Just in Time*
MIME	*Multipurpose Internet Mail Extensions*
SPC	Serviço de Proteção ao Crédito
SDK	Software *Development* Kit
URI	*Uniform Resource Identifiers*
XML	*Extensible Markup Language*

SUMÁRIO

1. INTRODUÇÃO

Segundo pesquisa do Diário do Comércio, o setor de alimentação fora do lar movimenta cerca de 170 bilhões de reais no Brasil, e é uma área que não costuma sofrer influência do momento econômico do país. Já o mercado de entretenimento, segundo pesquisa do Mundo Marketing, continua em crescimento, visto que os brasileiros não abrem mão de momentos de lazer. Juntas, essas duas áreas correspondem a média de 25% do gasto mensal em uma família no Brasil, segundos dados do Índice Nacional de Preços ao Consumidor Amplo, 2012/2018 – IBGE. Ou seja, são áreas que independentemente da situação financeira, sempre terão importância na vida do Brasileiro.

Aliado a essas informações, temos o fato de que uma das maiores metas do brasileiro para 2020 é economizar, conforme demonstrado pela pesquisa realizada pela CNDL e o SPC Brasil. A informação de que o uso de cupons de desconto demonstra ser benéfica tanto para o cliente quanto para os comércios conforme artigo do *E-commerce* Brasil. E o resultado da pesquisa da PwC - PricewaterhouseCoopers Brasil Ltda, que indica que 50% dos brasileiros fazem compras via smartphone ao menos uma vez por mês.

Com essas informações fica claro que a união dessas duas áreas com a oferta de descontos que possam ser encontrados facilmente pelo usuário com o uso do seu smartphone, beneficiará os clientes que buscam economizar e os empresários que desejam que seus produtos chamem a atenção de cada vez mais pessoas. É essa a proposta do aplicativo que será desenvolvido neste trabalho.

O aplicativo será desenvolvido apenas na plataforma Android, pelo fato de que a grande maioria dos usuários utilizam esse sistema operacional. Dados da StatCounter, mostram que atualmente 73% dos usuários de smartphone no mundo usam o Android, no Brasil essa porcentagem atinge 87%. As ferramentas para o desenvolvimento são gratuitas, o que também é um fator importante na escolha, facilitando o desenvolvimento nativo que, como resultado, gerará um aplicativo rápido e mais fácil de manter nos padrões sugeridos pela Google.

1.1 PROBLEMA DE PESQUISA

É possível desenvolver um aplicativo para agrupar vários descontos disponibilizados por locais de alimentação e entretenimento de uma determinada região?

1.2 OBJETIVOS

1.2.1 Geral

Desenvolver um protótipo de aplicativo Android em Kotlin, possibilitando utilizar descontos em locais de entretenimento e alimentação de uma região.

1.2.2 Específicos

- Avaliar os componentes da plataforma Android e suas bibliotecas utilizadas para o desenvolvimento de aplicações móveis;
- Desenhar as telas do aplicativo de acordo com heurísticas de usabilidade;
- Desenvolver integração com banco de dados para persistir os dados utilizados pelo usuário;
- Usar ferramentas de teste automatizado para testar o banco de dados do aplicativo;
- Desenvolver o aplicativo possibilitando o uso em telas maiores, como *tablets*;

1.3 JUSTIFICATIVA

No Brasil, são movimentados cerca de 170 bilhões de reais com alimentação fora do lar, o mercado de entretenimento continua em crescimento, pois os brasileiros não abrem mão de momentos de lazer. Essas áreas correspondem, em média a 25% do gasto mensal familiar no país, são áreas que independentemente da situação financeira, tem importância na vida do Brasileiro.

Diante disso, surgiu a ideia de criar um aplicativo para que empresas desses setores pudessem aumentar o seu alcance para novos clientes, através de cupons de desconto para os usuários de smartphone que buscam opções de atividades de lazer ou alimentação fora de casa. Como a ampla maioria de usuários de smartphone no Brasil possuem celulares Android, o desenvolvimento será apenas nessa plataforma. Sendo assim possível a aplicação real do resultado do TCC, possibilitando ainda novas pesquisas e trabalhos futuros focando na melhoria do aplicativo desenvolvido.

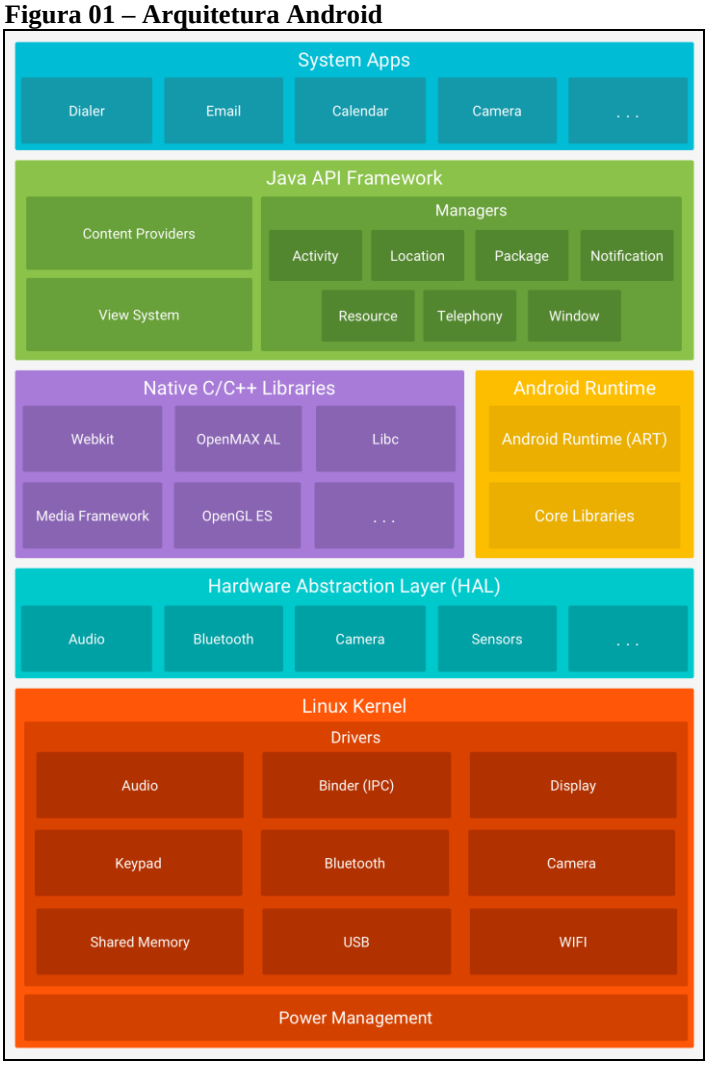
2. REFERENCIAL TEÓRICO

Nessa seção serão expostos alguns dos temas envolvidos no desenvolvimento deste trabalho e no aplicativo. Primeiramente serão descritos conceitos sobre o sistema operacional Android, as tecnologias envolvidas no desenvolvimento de aplicações para esta plataforma. Em sequência, serão explicadas brevemente as ferramentas utilizadas para o desenvolvimento, como o Android Studio, o Kotlin e o Android Jetpack.

2.1 ANDROID

O Android, segundo a Google (2019), é um sistema operacional, desenvolvido e mantido pela Google, de código aberto, com base no núcleo do Linux. A arquitetura da plataforma Android é demonstrada na Figura 01.

Figura 01 – Arquitetura Android



Fonte: Google (2019)

Na figura 01, podemos observar algumas camadas importantes para o desenvolvimento de aplicativos Android:

A Hardware *Abstraction Layer* (HAL) segundo a Google (2019), é o que fornece as interfaces padrões para que permita que os aplicativos tenham acesso as capacidades do hardware. Ainda de acordo com a Google (2019), "A HAL consiste em módulos de biblioteca, que implementam uma interface para um tipo específico de componente de hardware, como o módulo de câmera ou Bluetooth.".

O Android *Runtime*, conforme descrito pela Google (2019), em dispositivos com a versão Android 5.0 ou superior, sugere que "... cada aplicativo executa o próprio processo com uma instância própria do Android *Runtime* (ART).". Além disso a ART possui alguns recursos, como compilação *Ahead-of-time* (AOT) e *Just-in-time* (JIT), e a Coleta de lixo otimizada.

Os celulares Android vêm com alguns aplicativos de sistema para as tarefas mais comuns, como e-mail e navegador de internet. Esses aplicativos padrões não possuem nenhuma diferença entre outros aplicativos instalados pelo usuário, como explica a Google (2019). Mas, o mais interessante para desenvolvedores, ainda com base na Google (2019), é que esses aplicativos podem ser chamados pela aplicação desenvolvida, ou seja, não é necessário incluir uma função para, por exemplo, enviar e-mail, basta utilizar o aplicativo padrão para essa tarefa, simplificando a tarefa dos desenvolvedores.

2.2 FUNDAMENTOS DO APLICATIVO ANDROID

Os aplicativos para Android podem ser escritas de forma nativa em três linguagens, Kotlin, Java e C++, conforme a Google (2019), as ferramentas disponibilizadas pela SDK compilam esse código em um conjunto com todas as informações, que é o arquivo APK, esse arquivo contêm o aplicativo todo e é utilizado para instalar o aplicativo no Smartphone.

Sobre a segurança nos aplicativos:

> O sistema Android implementa o princípio do privilégio mínimo. Ou seja, cada aplicativo, por padrão, tem acesso somente aos componentes necessários para a execução do seu trabalho e nada mais. Isso cria um ambiente muito seguro em que o aplicativo não pode acessar partes do sistema a que não tem permissão. (GOOGLE, 2019)

Porém pode ser que o aplicativo utilize outros aplicativos, ou acesso a dados, para essa situação a Google (2019) diz que é necessário que o usuário conceda essas permissões explicitamente.

2.3 ARQUIVO DE MANIFESTO DO APLICATIVO

Segundo a Google (2020), o arquivo de manifesto é lido antes de iniciar qualquer componente do aplicativo, pois esse arquivo declara todos os componentes do aplicativo, e assim o sistema sabe quais os componentes existentes no aplicativo, o manifesto é um arquivo XML, deve ter o nome AndroidManifest.xml e deve estar localizado na pasta raiz do projeto do aplicativo. Este arquivo contém informações importantes para a compilação do aplicativo, para o sistema operacional e também para a loja de aplicativos Google Play.

Entre as informações disponibilizadas neste arquivo, uma das mais importantes, de acordo com a Google (2020), é o nome do pacote e o ID do aplicativo, essa informação é de extrema importância, pois é usada para a compilação do aplicativo, e será usado para acessar os recursos no código, além disso é com essa informação que é gerado o identificador único e universal do aplicativo desenvolvido. Esse ID é a informação que é usada na loja de aplicativo para identificar o aplicativo, por isso é único para cada *app* existente na loja. Devido a necessidade dessas informações o atributo *package* é parâmetro obrigatório na tag raiz do arquivo de manifesto.

Cada componente do *app* deve estar descrito em uma *tag* no manifesto, por exemplo se o aplicativo for composto de 4 *Activities*, será necessário conter 4 *tags* <*activity*>, cada uma com o atributo *name* definindo nome da classe dessa atividade, conforme a Google (2020). Outras *tags* para representação de componentes são <*service*>, <*receiver*> e <*provider*>, que representam os 3 outros componentes de um aplicativo, componentes estes que serão descritos com mais detalhes posteriormente no trabalho

As permissões para dados do usuário ou recursos do sistema, como explicado pela Google (2020), também são especificados no arquivo de manifesto, como por exemplo dados da agenda do usuário ou acesso a câmera, por exemplo. Cada permissão é descrita em uma *tag* específica, <*uses-permission*> com o atributo *name* referente ao recurso solicitado. Cada uma dessas *tags* gerará uma solicitação para o usuário aprovar, caso seja aprovado o funcionamento será como previsto e caso rejeitado, ao tentar acessar o recurso, ocorrerá uma falha.

Ainda no arquivo de manifesto, como definido pela Google (2020), são declarados quais os dispositivos compatíveis com o aplicativo, definindo quais versões de sistema são necessários para a instalação do aplicativo e quais os recursos de hardware obrigatórios. Assim a loja de aplicativos impede que uma aplicação que necessite de uma versão do Android específica, ou de um recurso específico, seja instalado em um celular sem essa

capacidade. Por exemplo um aplicativo que define que necessita ao menos do Android 8, não poderá ser instalado em um aparelho com a versão 7 ou menor, ou um aplicativo que necessite do sensor de bússola, não poderá ser instalado em um aparelho sem essa funcionalidade de hardware.

2.4 *INTENTS* E FILTROS DE *INTENTS*

Os *intents*, segundo a Google (2019), são objetos de mensagem, utilizados para solicitar uma ação de outro componente. Existem três funções básicas para o uso, iniciar uma atividade, iniciar um serviço ou fornecer uma transmissão. Existem dois tipos de *Intents*, o *Intent* implícito, conforme a Google (2019), "Os *intents* implícitos não nomeiam nenhum componente específico, mas declaram uma ação geral a realizar, o que permite que um componente de outro aplicativo a processe.". O segundo tipo de *intent* é o explícito:

> Os intents explícitos especificam qual aplicativo atenderá ao *intent*, fornecendo o nome do pacote do aplicativo de destino ou o nome da classe de um componente totalmente qualificado. Normalmente, usa-se um *intent* explícito para iniciar um componente no próprio aplicativo porque se sabe o nome de classe da atividade ou do serviço que se quer iniciar. (GOOGLE, 2019)

A diferença entre os tipos de *intents* é simples de se entender, de acordo com a Google (2019), o explícito é utilizado para passar informações e chamar outro componente quando se sabe exatamente qual o componente se deseja utilizar, por isso é normalmente utilizado dentro do próprio aplicativo. Já o implícito é utilizado quando se sabe qual ação se deseja realizar, porém não se sabe qual componente em específico será utilizado, neste caso componente que realizará tal ação pode ser dentro do próprio aplicativo ou de um processo exterior.

Para o caso dos *intents* implícitos, o sistema Android utiliza os filtros de *intent* para saber qual componente iniciar:

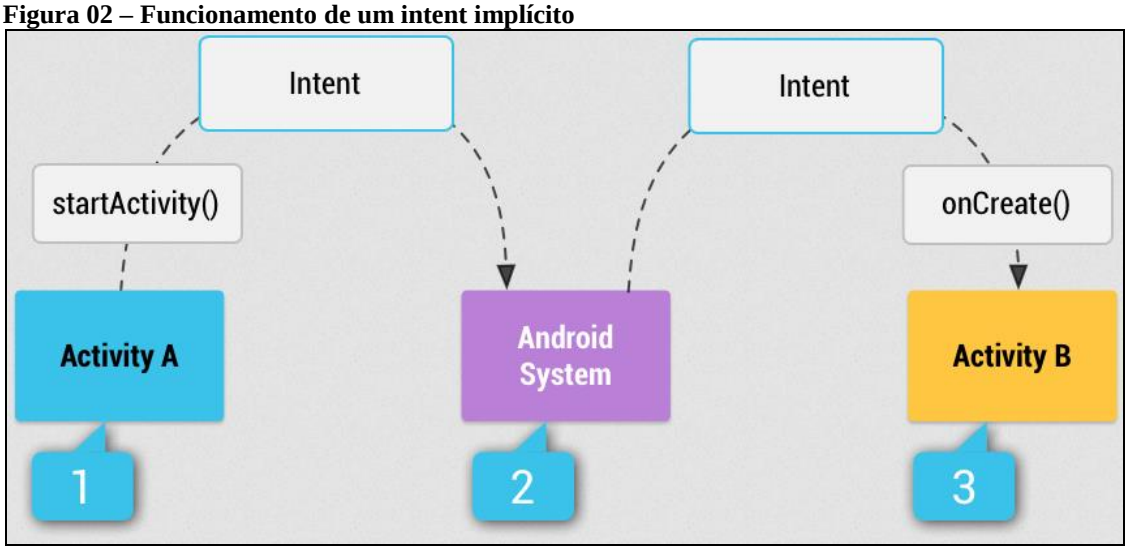
> Ao criar um *intent* implícito, o sistema Android encontra o componente adequado para iniciar, comparando o conteúdo do intent aos filtros de intents declarados no arquivo de manifesto de outros aplicativos no dispositivo. Se o intent corresponder a um filtro de intents, o sistema iniciará esse componente e entregará o objeto Intent. Se diversos filtros de intents corresponderem, o sistema exibirá uma caixa de diálogo para que o usuário selecione o aplicativo que deseja usar. (GOOGLE, 2019)

O filtro de *intents* ainda como explicado pela mesma documentação, é explicitado no arquivo de manifesto de cada aplicativo, com o uso de expressões que definem quais *intents* o aplicativo é capaz de processar. Portanto, ao declarar um filtro de *intent* para alguma

atividade, outros aplicativos poderão iniciar diretamente essa atividade com determinado tipo de *intent*. Da mesma forma, se não existir nenhum filtro definido no manifesto, todos os componentes do aplicativo serão passíveis de inicialização apenas com *intent* explícito.

A Figura 02 demonstra como funciona a inicialização de uma atividade a partir de um *intent* implícito criado por outra atividade.

Figura 02 – Funcionamento de um intent implícito



Fonte: Google (2019)

Na figura 02 podemos ver a sequência de uma atividade de um aplicativo, que cria um *intent*, e o sistema operacional define, a partir dos filtros de *intents* de todos os aplicativos instalados no celular, qual atividade chamará passando para ela a informação do *intent*, conforme a Google (2019) explica.

A criação de um objeto *intent*, como explicado pela Google (2019), pode ser feita por qualquer atividade e é possível incluir algumas informações nesse objeto, para ajudar o sistema a encontrar a atividade que será executada, as informações estão explicitadas no Quadro 01.

Quadro 01 – Informações do objeto *intent*

Nome do componente	É o nome do componente a iniciar. É opcional, mas é a informação fundamental que torna um *intent* explícito, o que significa que o *intent* deve ser entregue somente ao componente do aplicativo definido pelo nome do componente. Sem nome de componente, o *intent* será implícito, e o sistema decidirá qual componente deve receber o *intent*.
Ação	*String* que especifica a ação genérica a realizar (como exibir ou selecionar). No caso de um *intent* de transmissão, essa é a ação que entrou em vigor e que está sendo relatada. A ação determina amplamente como o resto do *intent* é estruturado especificamente e o que está contido nos dados e em extras.
Dados	É o URI que referencia os dados a serem aproveitados e/ou o tipo MIME desses dados. O tipo dos dados fornecidos geralmente é determinado pela ação do *intent*. Por exemplo, se a ação for para editar, os dados deverão conter o URI do documento a editar.
Categoria	É uma *string* que contém informações adicionais sobre o tipo de componente que deve processar o *intent*. Qualquer número de descrições de categoria pode ser inserido em um *intent*, mas a maioria dos *intents* não requer nenhuma categoria.
Extras	São pares de chave-valor que carregam informações adicionais necessárias para realizar a ação solicitada. Assim como algumas ações usam determinados tipos de URIs de dados, outras também usam determinados extras. Por exemplo se o *intent* é enviar um e-mail, pode ser passado um extra que informa o assunto desse e-mail.
Sinalizadores	Sinalizadores são definidos na classe *Intent* e funcionam como metadados para o *intent*. Os sinalizadores podem instruir o sistema Android a inicializar uma atividade (por exemplo, a qual tarefa a atividade deve pertencer) e como tratá-la após a inicialização.

Fonte: Elaborado a partir de Google (2019)

2.5 COMPONENTES DE UM APLICATIVO ANDROID

Conforme descrito pela Google (2019) um aplicativo Android é construído a partir de 4 componentes, onde cada componente é uma porta de entrada ao aplicativo, seja essa entrada para o sistema ou para o usuário. Alguns desses componentes dependem dos outros. Cada um tem um ciclo de vida distinto, que define como cada um é criado e destruído. Esses componentes serão explicados a seguir.

2.5.1 Atividades

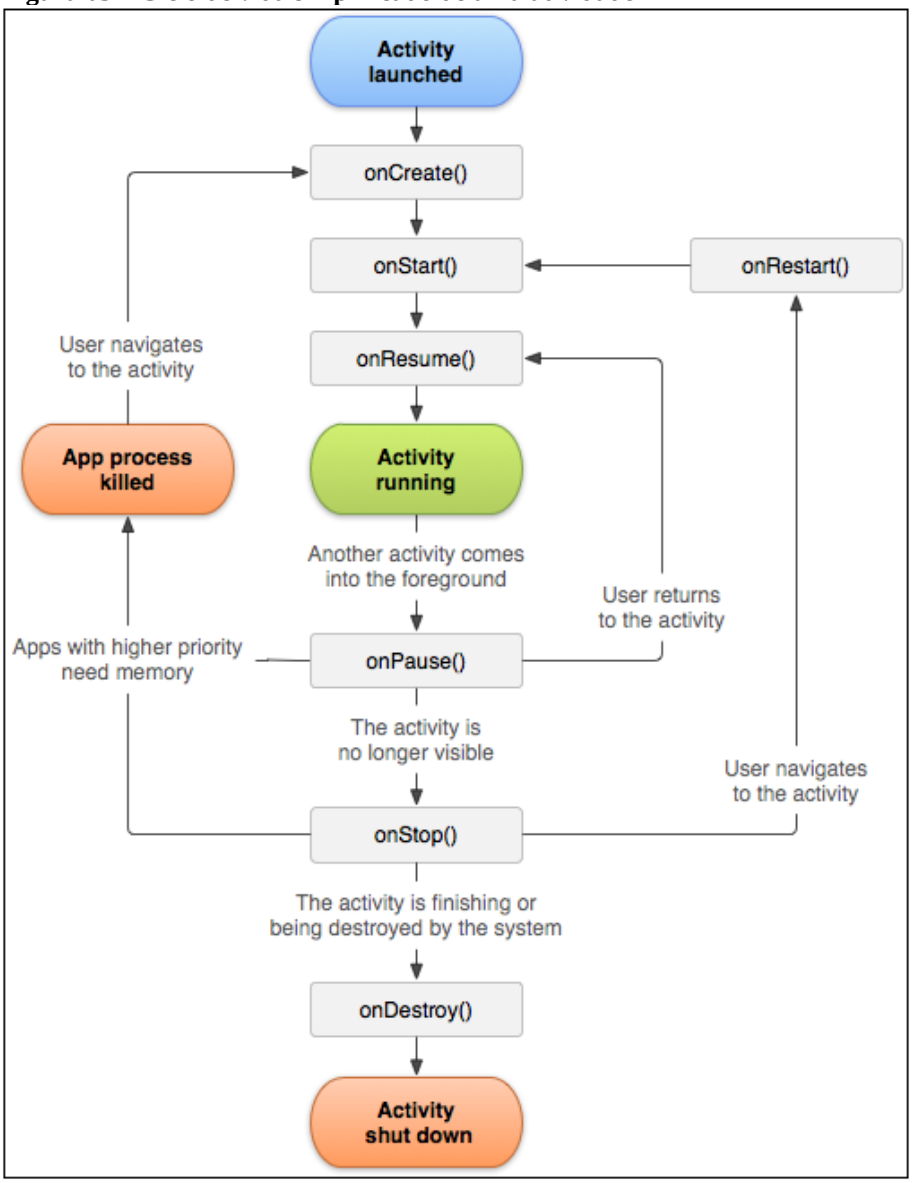
De acordo com os textos da Google (2019), “Uma atividade é o ponto de entrada para a interação com o usuário. Ela representa uma tela única com uma interface do usuário.” Por exemplo, um aplicativo de e-mail pode ter uma atividade que mostre a lista de e-mails, uma que cria e-mails para serem enviados e outra que lê os e-mails não lidos. Conforme a Google (2019) essas atividades funcionam de forma independente, mas juntas elas formam uma experiência coerente para o usuário do aplicativo de e-mail. Dessa forma, um outro aplicativo que necessite utilizar alguma função do e-mail pode iniciar em qualquer uma dessas atividades, desde que o aplicativo de e-mail permita, ou seja, a atividade de um aplicativo pode iniciar a atividade de outro aplicativo.

Ainda sobre atividades:

> Uma atividade fornece a janela na qual o app desenha a própria IU. Essa janela normalmente preenche a tela, mas pode ser menor do que a tela e flutuar sobre outras janelas. Geralmente, uma atividade implementa uma tela em um app. Por exemplo, uma das atividades de um app pode implementar uma tela Preferências, enquanto outra atividade implementa uma tela selecionar foto. (GOOGLE, 2019)

A Google (2019) explica que todo aplicativo normalmente tem diversas atividades, e várias telas, e uma delas, definida como atividade principal, é a primeira tela exibida ao usuário, e cada atividade chama outra atividade dentro do próprio aplicativo. Conforme o usuário navega pelo aplicativo e suas telas, essas atividades transitam por estados dentro de ciclo de vida de atividades. O desenvolvedor implementa algumas funções disponíveis na classe de Atividade para tratar a navegação entre esse ciclo, essas funções e o ciclo de vida estão descritas na Figura 03.

Figura 03 – Ciclo de vida simplificado de uma atividade



Fonte: Google (2019)

Na figura 03 é possível verificar os estados e quais as funções que são executadas quando a atividade entra em um novo estado. De acordo com a Google (2019), dependendo da complexidade da atividade, pode não ser necessário implementar todos esses métodos, ou seja, nem todos os métodos são obrigatórios para o bom funcionamento da atividade. O método *onCreate* é sempre obrigatório, pois é a porta de entrada da atividade, conforme a Google (2019), "No método *onCreate*(), você executa a lógica básica de inicialização do aplicativo. Isso deve acontecer somente uma vez durante todo o período que a atividade durar.".

Após essa inicialização o método *onStart*, de acordo com a Google (2019), "A chamada *onStart*() torna a atividade visível ao usuário, à medida que o aplicativo prepara a atividade para inserir o primeiro plano e se tornar interativa.". Ou seja, esse é o método que

possibilita a visualização da atividade pelo usuário. Em sequência é chamado o método *onResume,* e após essa execução, segundo a Google (2019), o aplicativo entra no estado que o usuário pode interagir com o aplicativo, e se mantém nesse estado até que o aplicativo perca o foco por alguma ação, como por exemplo o desligamento da tela, uma ligação recebida ou a navegação para outra atividade.

2.5.2 Serviços

"O serviço é um ponto de entrada para manter um aplicativo em execução no segundo plano, seja qual for o motivo" (Google, 2019). Segundo Lecheta (2010) geralmente um serviço tem alto consumo de recursos e não precisa interagir com o usuário, por conta disso não precisa de uma interface gráfica. A classe de serviço, conforme Lecheta (2010) pode ser usada para fazer downloads mais demorados sem que o usuário perceba, ou tocar música em segundo plano enquanto o usuário utiliza outros aplicativos, por exemplo. "O serviço é sempre executado em segundo plano, e o próprio sistema operacional cuida de seu processo e do gerenciamento de memória." (LECHETA, 2010, p.317).

Ou seja, não temos o mesmo controle do ciclo de vida que temos na atividade, pois, de acordo com Lecheta (2010), esta classe faz parte dos processos cujo ciclo de vida é controlado pelo sistema operacional, e, enquanto estiver em execução, não será finalizado, a não ser que o celular esteja em condições de memória extremamente baixas, ocasião onde o Android pode encerrar processos para liberar recursos.

Outro fato interessante sobre o ciclo de vida de um serviço:

> "[...]Mesmo que um *service* seja encerrado devido à falta de memória, o Android posteriormente tentará reiniciá-lo para que o processamento continue assim que as condições de memória e os recursos utilizados estejam normais. Nesse caso, o serviço deve ser codificado de uma forma que seja possível recuperar o estado em que o processamento foi encerrado anteriormente." (LECHETA, 2010, p.318)

Portanto o desenvolvedor que usar essa classe, deve programar sabendo que o sistema pode precisar parar o processamento para depois retomar, e deve levar isso em conta quando desenvolver utilizando essa classe.

A maneira mais comum de iniciar um serviço, conforme Google (2019) é chamando o método *startService*() passando como parâmetro um *Intent*. O serviço permanecerá em execução até que seja interrompido por outro componente, utilizando o método *stopService*(),

ou ainda o próprio serviço chega ao final de sua função e chama o método *stopSelf*(), que o encerra, conforme descrito pela Google (2019).

2.5.3 *Broadcast receivers*

A classe *BroadcastReceivers*, de acordo com Lecheta (2010), é utilizada para permitir que aplicações possam reagir a eventos enviados pelo sistema operacional, é, assim como os *Services*, executada sempre em segundo plano e sem interface gráfica, porém são executadas em pouco tempo. "Seu objetivo é receber uma mensagem (*intent*) e processá-la sem que o usuário perceba. Isso é um importante passo para integrar aplicações, uma vez que elas podem trocar mensagens em segundo plano sem atrapalhar o usuário." (LECHETA, 2010, p.290).

O ciclo de vida de um *Broadcast Receiver* é muito menor que o de um *Service*:

> Um *BroadcastReceiver* é válido somente durante a chamada ao método *onReceive(context, intent)*. Depois disso o sistema operacional encerrará seu processo para liberar memória. O código do método *onReceive(context, intent)* deve executar brevemente, e assim que o método terminar o Android vai considerar que o *BroadcastReceiver* não está mais ativo e está pronto para ser destruído." (LECHETA, 2010, p.296)

Então a chamada para essa classe deve ser de retorno extremamente rápido, Lecheta (2010) diz que caso demore mais de 10 segundos para executar, o sistema operacional exibirá uma mensagem de erro informando que a aplicação não está respondendo. Por essa limitação, a classe *Service* deve ser usada quando uma atividade em segundo plano mais longa é necessária.

Portanto, conforme a Google (2019), esse tipo de classe é utilizado para receber e reagir a eventos enviados pelo próprio sistema operacional, como por exemplo quando o usuário entra no modo avião, é disparado uma mensagem informando dessa troca de estado, um *Broadcast Receiver* pode, por exemplo, decidir parar algum processamento.

A Google (2019), define duas formas de receber essas transmissões, uma sendo declarada no arquivo de manifesto, faz com que o aplicativo seja iniciado, caso não esteja iniciado, quando o sistema dispara o *Broadcast*. O outro método é a recepção por contexto, por exemplo uma atividade pode ser programada para agir ao receber a transmissão, neste caso o aplicativo apenas responderá à transmissão caso o aplicativo esteja iniciado e a atividade em questão esteja em execução.

Além de definir como receber os *Broadcasts*, a Google (2019) também possibilita a transmissão deles de três maneiras distintas descritos, demonstrados no Quadro 02:

Quadro 02 – Como enviar transmissões

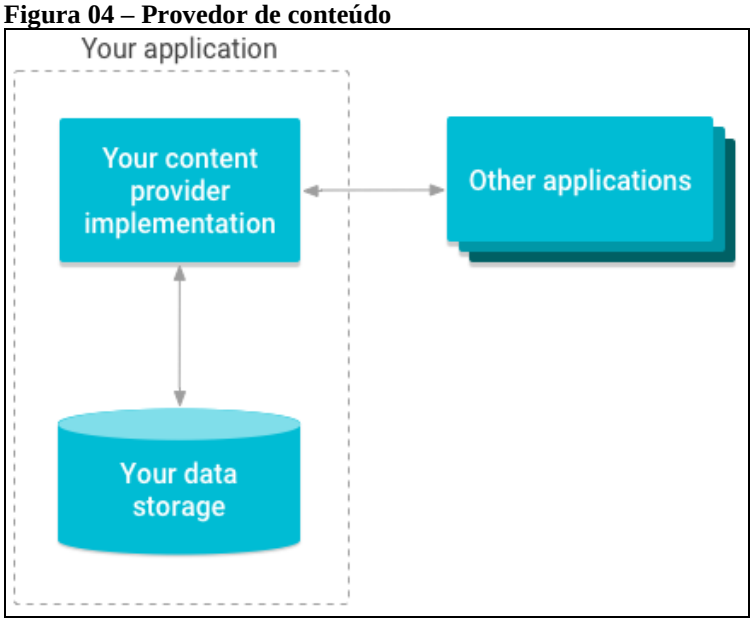
sendOrderedBroadcast (Intent, String)	Envia transmissões para um receptor de cada vez. À medida que cada receptor é executado, ele pode propagar um resultado para o próximo receptor ou pode interromper completamente a transmissão para que não seja passada para outros receptores. Os receptores de pedidos executados podem ser controlados com o atributo *android:priority* do filtro de *intent* correspondente. Os receptores com a mesma prioridade serão executados em uma ordem arbitrária.
sendBroadcast(Intent)	Envia transmissões para todos os receptores em uma ordem indefinida. Isso é chamado de *Normal Broadcast.* É mais eficiente, mas significa que os receptores não podem ler resultados de outros receptores, propagar dados recebidos ou abortar a transmissão.
LocalBroadcastManager .sendBroadcast	Envia transmissões para receptores que estão no mesmo app que o remetente. Se você não precisa enviar transmissões entre apps, use transmissões locais. A implementação é muito mais eficiente porque não é necessária uma comunicação entre processos e você não precisa se preocupar com problemas de segurança relacionados a outros apps conseguindo receber ou enviar suas transmissões.

Fonte: Elaborado a partir de Google (2019)

2.5.4 Provedores de conteúdo

De acordo com a Google (2019), provedores de conteúdo servem para ajudar o aplicativo a gerenciar o acesso aos próprios dados ou aos dados de outros aplicativos, além de facilitar o compartilhamento dessas informações. “Eles encapsulam os dados e fornecem mecanismos para definir a segurança deles. Os provedores de conteúdo são a interface padrão que conecta dados em um processo com código em execução em outro processo.” (GOOGLE, 2019). O mais importante de utilizar um provedor de conteúdo, conforme a Google (2019) é a possibilidade de permitir que outros aplicativos acessem e modifiquem os dados do aplicativo de forma segura, conforme a Figura 04.

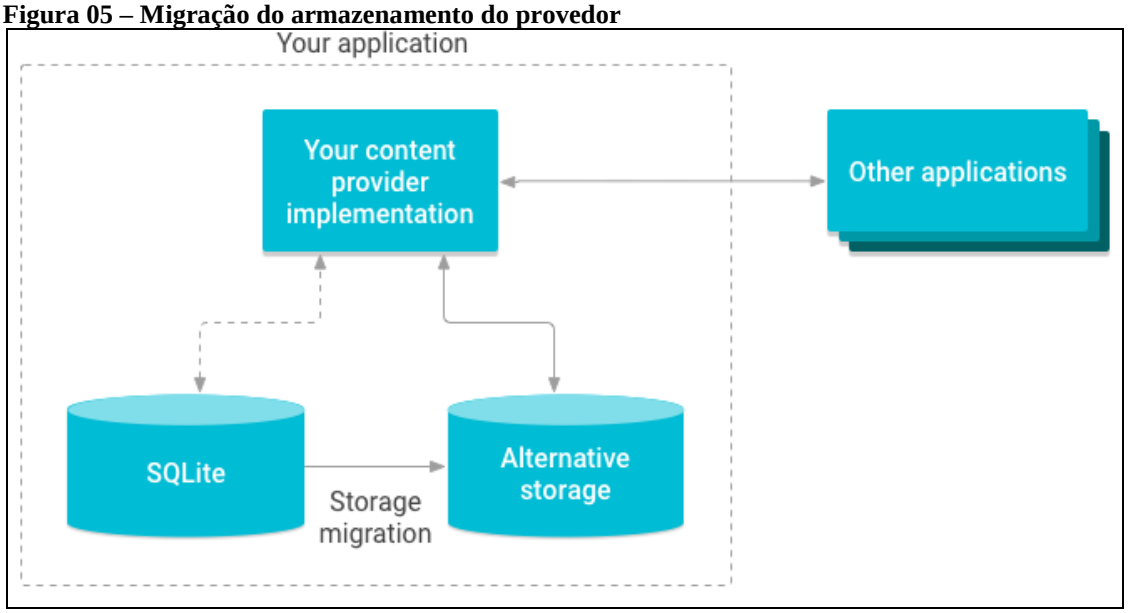
Figura 04 – Provedor de conteúdo



Fonte: Google (2019)

Outra função do provedor de conteúdo, conforme a Google (2019) é o compartilhamento de dados, mesmo não sendo essa sua função principal. Porém, por oferecer um bom nível de abstração, é possível modificar a implementação do armazenamento de dados do seu aplicativo, sem necessitar alterar nenhum outro aplicativo que, por ventura, utilize os dados do seu aplicativo, como por exemplo a Figura 05, onde foi feito a migração de um armazenamento *SQLite* para uma forma alternativa, mas outras aplicações não notam essa mudança.

Figura 05 – Migração do armazenamento do provedor



Fonte: Google (2019)

2.6 KOTLIN

Kotlin, de acordo com Vasić (2017), é uma linguagem com tipagem estática, ou seja, os tipos das variáveis são conhecidos em tempo de compilação, roda na Máquina Virtual Java, Android, navegadores web ou nativamente em binário, é orientada a objetos, porém possibilita também a programação funcional. É gratuita é *open source*, mas desenvolvida e mantida pela JetBrains, e, desde de 2019, oficializada pela Google como linguagem principal para aplicativos do Sistema Operacional Android.

O uso do Kotlin é interessante, em especial para desenvolvimento Android, por algumas questões. Segundo a Fundação Kotlin (2020), é uma linguagem em constante evolução com uma grande comunidade trabalhando nela, é compatível com Java, possibilitando chamadas de classes Java, e uso de bibliotecas desenvolvidas em ambas as linguagens. Além disso, possui elementos de programação funcional, como funções *inline* e *lambda.*

Pode ser utilizada não apenas para desenvolvimento mobile, mas também para desenvolvimento Web, seja ele no lado do servidor ou do cliente, para desenvolvimento desktop e nativo, podendo ser compilada para *Bytecode* compatível com Java, para uma linguagem intermediária que depois gera código compatível com *JavaScript*, ou nativamente para código específico da plataforma em questão.

2.7 ANDROID STUDIO

O Android Studio, segundo a Google (2019), “é o ambiente de desenvolvimento integrado (IDE, na sigla em inglês) oficial para o desenvolvimento de apps para Android e é baseado no IntelliJ IDEA”. Essa IDE é focada para desenvolvimento Android e traz diversos recursos específicos para essa tarefa, como a visualização da árvore de arquivos organizada por módulos, cada módulo contendo arquivos específicos, sejam eles os arquivos de código fonte, scripts de compilação, ou arquivos essenciais para o aplicativo, como o arquivo manifesto.

O editor de layout do Android Studio permite que o desenvolvedor faça o layout da tela desejada arrastando elementos de interface com o usuário, onde será gerado o arquivo XML referente ao layout automaticamente, acelerando o processo de desenvolvimento, pois o desenvolvedor foca na funcionalidade de cada elemento e seus atributos, ao invés de ter que controlar essa parte manualmente, conforme explicado pela Google (2020).

Com o Android Studio, de acordo com a Google (2020), também auxilia no processo de compilação do aplicativo, principalmente com o uso do Gradle, que é um automatizador do processo de compilação, gratuito e *open source* como descrito pela Gradle Inc. (2020). Dessa forma, “O sistema de compilação do Android compila os recursos e o código-fonte do app e os empacota em APKs que você pode testar, implantar, assinar e distribuir” (GOOGLE, 2020). Essa integração entre a IDE e a ferramenta de compilação, permite configurar vários perfis de compilação para cada situação desejada, o mais comum é um perfil para a depuração do aplicativo, onde ficam registrados *logs* da aplicação, e normalmente se executam os testes, e outro perfil para a versão de lançamento, onde é feito uma limpeza automática do código, removidos os testes e *logs* para gerar o arquivo final menor possível. Outras configurações personalizadas de compilação são demonstradas no Quadro 03.

Quadro 03 – Opções de compilação

Tipos de build	Os tipos de build definem algumas propriedades que o Gradle usa ao criar e empacotar o aplicativo. Geralmente, eles são configurados para diferentes fases do ciclo de desenvolvimento. É necessário definir pelo menos um tipo de build para. Por padrão, o Android Studio cria os tipos de build de depuração e de lançamento.
Variações de produtos	As variações de produtos representam diferentes versões do seu app que podem ser lançadas para os usuários, como versões gratuitas e pagas da aplicação. Os tipos de produto são opcionais e precisam ser criados manualmente.
Variantes de compilação	Uma variante de compilação é o produto resultante da combinação de um tipo de build e uma variação de produto, portanto, a criação de outros tipos de build ou variações de produtos também faz com que outras variantes de compilação sejam criadas.
Entradas do manifesto	É possível especificar valores para algumas propriedades do arquivo de manifesto na configuração da variante de compilação. Esses valores de compilação modificam os valores existentes no arquivo de manifesto.
Dependências	O sistema de compilação gerencia as dependências do projeto a partir do seu sistema de arquivos local e de repositórios remotos. Dessa maneira evitando a necessidade de pesquisar, fazer o download e copiar manualmente pacotes binários das suas dependências para o diretório do projeto.
Assinaturas	O sistema de compilação permite especificar opções de assinatura nas configurações da compilação e ele pode assinar seus APKs automaticamente durante o processo de compilação. O Android studio assina a versão de depuração com uma chave e um certificado padrão, usando credenciais conhecidas para evitar solicitações de senha durante o tempo de compilação.
Suporte a vários APKs	O sistema de compilação permite criar diferentes APKs automaticamente, cada um contendo apenas os códigos e recursos necessários para uma densidade de tela ou Interface binária do aplicativo (ABI, na sigla em inglês) específica.

Fonte: elaborado a partir de Google (2020)

2.8 ANDROID JETPACK

De acordo com a Google (2020), “O Jetpack é um conjunto de bibliotecas, ferramentas e orientações para ajudar os desenvolvedores a criar apps de alta qualidade com mais facilidade.”. Além disso a Google (2020) explica que “O Jetpack compõe as bibliotecas de pacotes androidx*, separadas das APIs da plataforma. Isso significa que ele oferece compatibilidade com versões anteriores e é atualizado com mais frequência que a Plataforma Android.”

O Android Jetpack é, segundo a Google (2020) separado em 4 componentes, e juntas formam uma coleção de bibliotecas que podem ser adotadas individualmente ou por completo, usando todas as disponíveis e combinando cada uma delas. Os componentes estão descritos no quadro 04.

Quadro 04 – Componentes do Android Jetpack

Base	Os componentes de base fornecem funcionalidade transversal, como compatibilidade com versões anteriores, testes e compatibilidade com a linguagem Kotlin.
Arquitetura	Os componentes de arquitetura ajudam a criar aplicativos robustos, testáveis e de fácil manutenção.
Comportamento	Os componentes de comportamento ajudam o aplicativo a se integrar aos serviços padrão do Android, como notificações, permissões, compartilhamento e o Assistente.
Interface de Usuário	Os componentes de IU fornecem *widgets* e assistentes para tornar o aplicativo não apenas fácil, como também agradável de usar.

Fonte: elaborado a partir de Google (2020)

Entre as bibliotecas importantes do Android *Jetpack*, algumas se destacam pelo maior uso.

2.8.1 Room

O banco de dados *Room*, de acordo com a Google (2020), "oferece uma camada de abstração sobre o *SQLite* para permitir acesso fluente ao banco de dados, e, ao mesmo tempo, aproveitar toda a capacidade do *SQLite*." Essa abstração é feita a partir de três componentes principais, conforme explica a Google (2020) no quadro 05.

Quadro 05 – Componentes do banco de dados Room

Banco de dados	É o principal ponto de acesso para a conexão com dados relacionais e persistentes do aplicativo. Obrigatoriamente deve ter uma lista de entidades e um método que retorne a classe do DAO
Entidade	Representa uma tabela dentro do banco de dados
DAO	Em inglês, a sigla significa Objetos de Acesso a Dados, contém os métodos utilizados para acessar o banco de dados

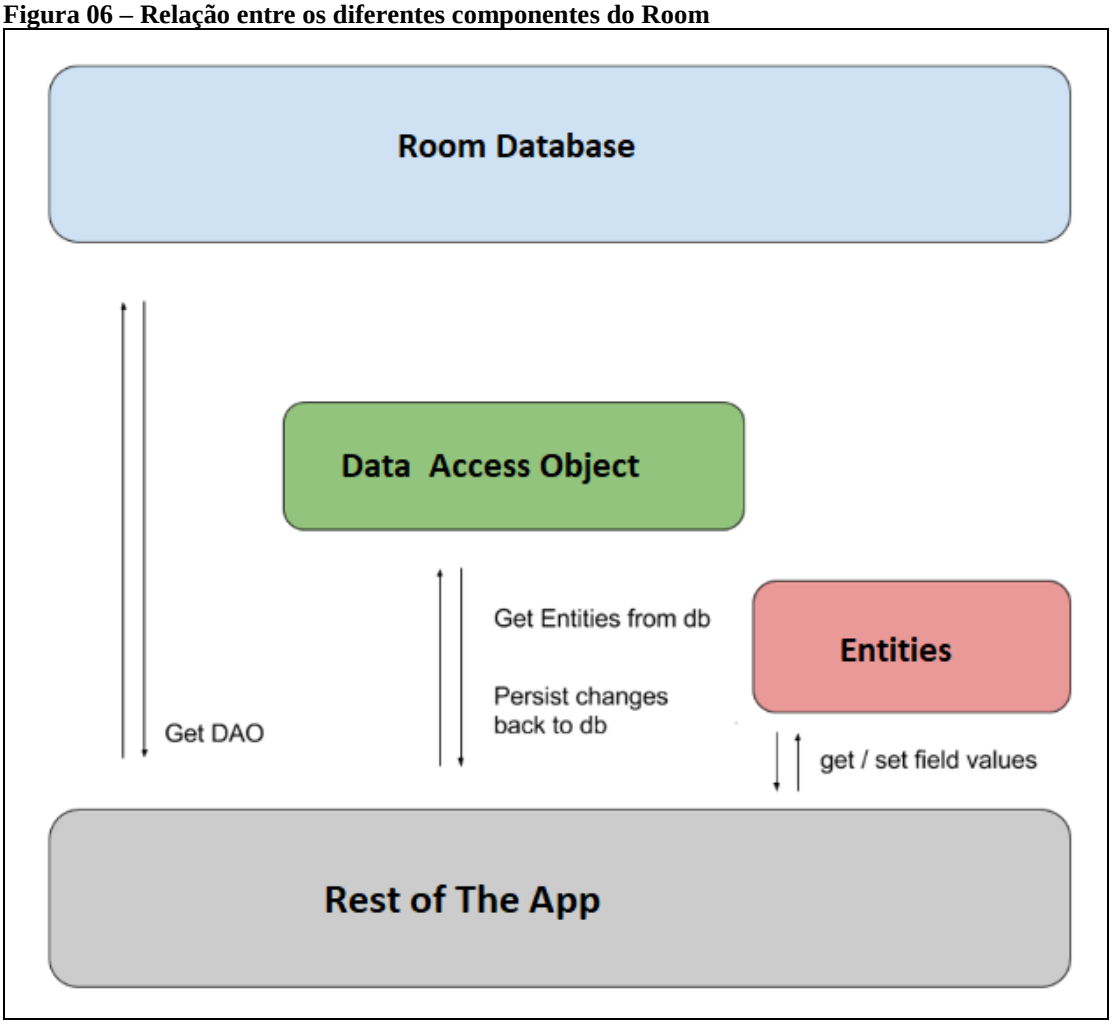
Fonte: elaborado a partir de Google (2020)

Com esses componentes:

> "O app usa o banco de dados do Room para conseguir os objetos de acesso a dados associados a esse banco de dados, em seguida o app usa cada DAO para conseguir entidades do banco de dados e salvar as alterações dessas entidades de volta no banco de dados. Por fim, o app usa uma entidade para conseguir e definir valores que correspondam a colunas da tabela no banco de dados." (GOOGLE, 2020)

Ainda sobre a relação entre esses componentes do *Room* com o Aplicativo, temos a figura 06, onde é notado a interação do aplicativo com cada uma dessas partes que compõem o banco de dados *Room*.

Figura 06 – Relação entre os diferentes componentes do Room



Fonte: Google (2020)

2.8.2 R*ecyclerView*

O *RecyclerView* é uma parte do Android *Jetpack*, que é uma evolução mais flexível do *ListView*, usado para quando necessário exibir uma lista de rolagem de um grande conjunto de elementos onde os dados mudem com frequência, conforme a Google (2019).

Conforme a Google (2019) explica, o modelo *RecyclerView*, usa de diferentes componentes para exibir os dados, o contêiner geral, que é o próprio recyclerView, é adicionado diretamente no layout. Este contêiner é preenchido com as *views* fornecidas por um gerenciador de layout, existem layouts padrões disponibilizados na biblioteca Android, como o *LinearLayoutManager* e o *GridLayoutManager*, ou podem ser implementados manualmente. Cada *view* dessa lista é representada por um objeto fixador de visualização, ou seja, um *ViewHolder*, que é implementado estendendo a classe *RecyclerView.ViewHolder*. Cada um desses fixadores exibe um único item, uma *view*, e cada fixador é gerenciador por um adaptador, que é criado estendendo a classe *RecyclerView.Adapter*. Os adaptadores são

responsáveis por ligar os dados em si aos fixadores de visualização e também controla a quantidade de fixadores necessários na tela.

Para que esse modelo seja eficiente, o modelo *RecyclerView* realiza diversas otimizações automaticamente algumas delas são explicadas a seguir.

> Quando a lista é preenchida pela primeira vez, ela cria e vincula alguns fixadores de visualização em ambos os lados da lista. Por exemplo, se a visualização estiver exibindo as posições de lista de 0 a 9, o RecyclerView criará e vinculará esses fixadores de visualização e também poderá criar e vincular o fixador de visualização para a posição 10. Dessa forma, se o usuário rolar a lista, o próximo elemento estará pronto para ser exibido (GOOGLE, 2019)

Outra otimização importante, conforme descrito pela Google (2019) "À medida que o usuário rola a lista, o RecyclerView cria novos fixadores de visualização conforme necessário.". Além disso, os fixadores que foram rolados para fora da tela são salvos para que possam ser reutilizados, supondo que o usuário continue rolando para a mesma direção, esses fixadores são redefinidos com novos dados, e caso a seja rolado para a direção oposta, os mesmos podem ser trazidos de volta, portanto não é gasto o tempo de criar os novos fixadores a cada rolagem, pois são reutilizados os que já existem, como explica a Google (2019).

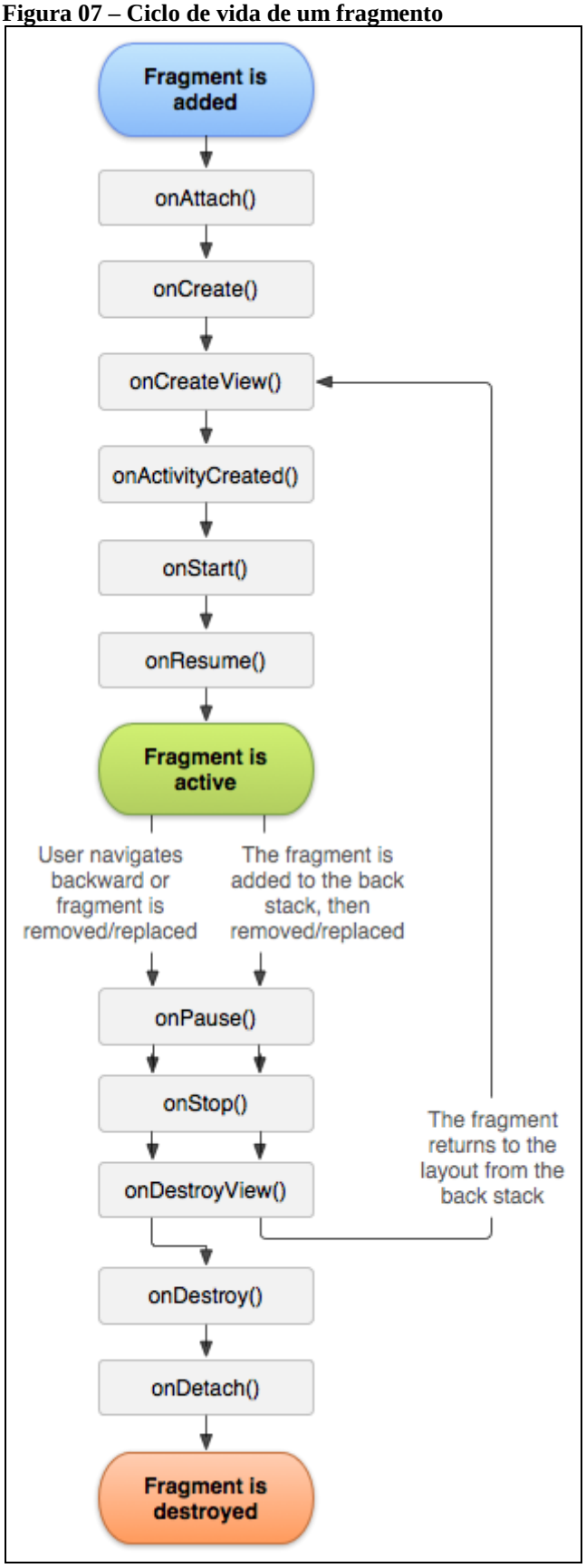
Portanto, com essas otimizações, a rolagem acontece de forma suave, já que os próximos dados já estão pré-carregados e os fixadores já existem, fazendo com o que o usuário nem perceba essa transição de informação da lista.

2.8.3 Fragmentos

Um fragmento, "representa o comportamento ou uma parte da interface do usuário em uma F*ragmentAcivity*" (GOOGLE, 2019), ele é como uma seção modular de uma atividade, com ciclo de vida distinto, eventos de entrada próprios, que pode ser adicionado ou removido durante a execução de uma atividade, uma espécie de subatividade que pode ser reutilizado em diversas atividades, conforme explicado pela Google (2019).

O ciclo de vida de um fragmento é bastante distinto quando comparado com o clico de uma atividade, visível na figura 07.

Figura 07 – Ciclo de vida de um fragmento



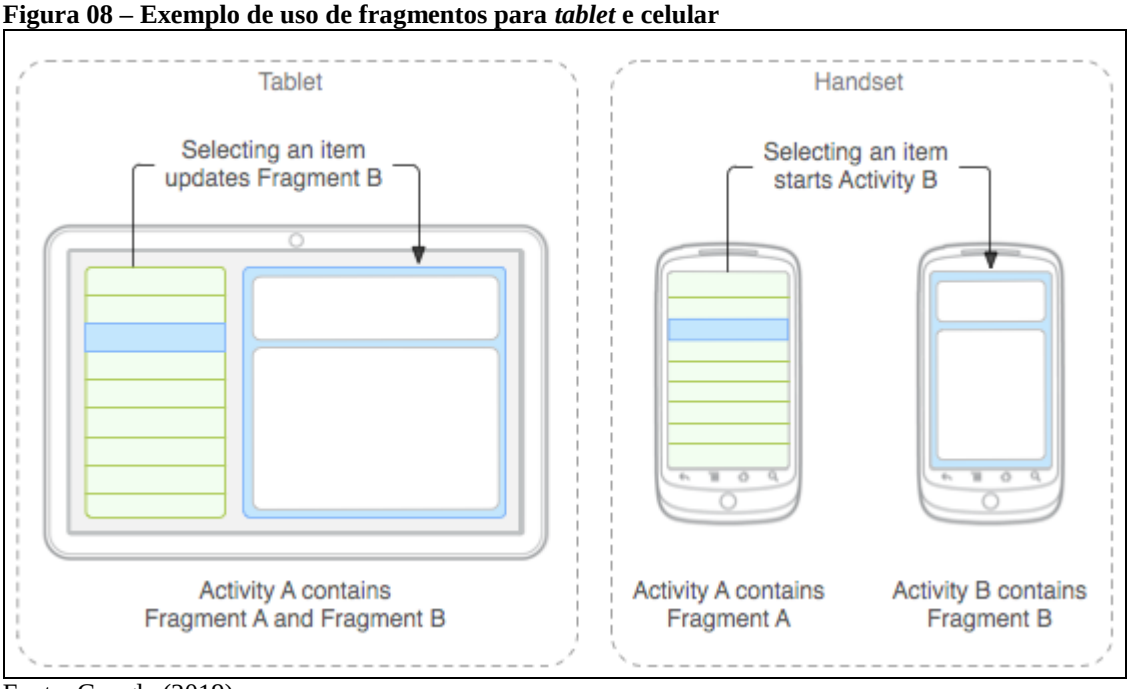
Fonte: Google (2019)

Apesar de fragmentos terem um ciclo de vida distinto, eles são impactados pelo ciclo de vida da atividade, visto que obrigatoriamente, uma atividade dever hospedá-lo, como explica a Google (2019). Ainda sobre o seu ciclo de vida:

> "[...]quando a atividade é pausada, todos os fragmentos também são, e, quando a atividade é destruída, todos os fragmentos também são. No entanto, enquanto uma atividade estiver em execução (estiver no estado do ciclo de vita retomado), é possível processar cada fragmento independentemente, como adicioná-los ou removê-los." (GOOGLE, 2019)

Com essa flexibidade de permitir ter os fragmentos dentro do ciclo de vida de uma atividade, conforme a Google (2019) explica, é mais simples realizar projetos que funcionem tanto em smartphones quanto em *tablets* onde a tela é maior, e pode, por exemplo, exibir dois fragmentos relacionados lado a lado, onde no celular é exigido uma navegação entre um e outro, pelo fato de sua tela ser mais limitada em tamanho. Como demonstrado na figura 08.

Figura 08 – Exemplo de uso de fragmentos para *tablet* e celular



Fonte: Google (2019)

2.8.4 D*ata binding*

Como explicado pela Google (2020) "A Data Binding Library é uma biblioteca de apoio que permite vincular componentes de IU dos seus layouts a fontes de dados do app usando um formato declarativo, em vez de programático" essa forma de realizar o vínculo substitui as várias chamadas de *findViewByID()* necessários para preencher as informações da interface com dados obtidos no programa

> “A vinculação de componentes no arquivo de layout permite remover muitas chamadas de framework da IU presentes nas suas atividades, tornando-as mais simples e fáceis de manter. Ela também pode melhorar o desempenho do app e ajudar a evitar vazamentos de memória e exceções de ponteiro nulo.” (GOOGLE, 2020)

A ligação de dados com a interface de forma programática é demonstrada na figura 09.

Figura 09 – Exemplo de ligação com a interface de forma programática

```
findViewById<TextView>(R.id.sample_text).apply {
    text = viewModel.userName
}
```

Fonte: Google (2020)

O exemplo de como realizar mesma ligação de forma declarativa é mostrado na figura 10.

Figura 10 – Exemplo de ligação com a interface de forma declarativa

```
<TextView
        android:text="@{viewmodel.userName}" />
```

Fonte: Google (2020)

Nas figuras 09 e 10, vemos o exemplo de uma *View* apenas, então pode parecer que a diferença é pouca, mas na maioria dos casos um *layout* possui muitas *views* que dependem de dados obtidos programaticamente, e ao usar *data binding* esse vínculo fica todo no arquivo XML do layout, eliminando grande parte de código *boilerplate*, o que facilita a manutenção e escalabilidade do código, como explicado pela Google (2020).

3. METODOLOGIA DA PESQUISA

Este trabalho é caracterizado como pesquisa aplicada e descritiva, pois foi desenvolvido um protótipo da aplicação e também teve como um dos objetivo saber se o protótipo desenvolvido tem as funcionalidades necessárias para possibilitar o uso comercial. O trabalho buscou responder o seguinte problema: É possível desenvolver um aplicativo para agrupar vários descontos disponibilizados por locais de alimentação e entretenimento de uma determinada região?

Após a finalização do levantamento bibliográfico, foi realizado um estudo mais aprofundado sobre o desenvolvimento de aplicações nativas em Android atualmente, utilizando as ferramentas e padrões de desenvolvimentos mais modernos possíveis, visando deixar a aplicação relevante por mais tempo, evitando futuros retrabalhos, migrações para novos padrões e facilitando possíveis adições futuras ao trabalho.

Com base nessa pesquisa foi definido que a aplicação seria feita na IDE Android Studio, com a linguagem Kotlin, usando as bibliotecas do Android *jetpack* e *design* baseado no conjunto de boas práticas definidos pela Google no *Material Design*. Essas escolhas foram feitas por serem sugeridas pela Google no ano de 2020, portanto a manutenção da tecnologia será constante.

4. PROTÓTIPO DE APLICATIVO PARA A BUSCA E REGISTRO DE DESCONTOS EM LOCAIS DE ALIMENTAÇÃO E ENTRETENIMENTO

Antes de iniciar com o desenvolvimento do aplicativo, foi pensado nas informações básicas que deveriam ser armazenadas em um banco de dados para que seja possível ter todas as informações necessárias sobre os descontos a serem exibidos. Após feito o modelo do banco de dados, foram definidas as funcionalidades básicas, desenhado as telas e o fluxo de navegação que o usuário teria ao utilizar o aplicativo. Então, com essas duas partes definidas foi desenvolvido o protótipo com o funcionamento principal da aplicação e definido o nome do aplicativo, Conto descontos.

4.1 ESTADO DA ARTE

Foi pesquisado na loja de aplicativos da Google, aplicativos que tem a funcionalidade semelhante com a ideia do trabalho foram encontrados alguns, entre os mais baixados na loja temos o Pelando, o Peixe Urbano e o Cuponeria.

O Pelando é um aplicativo similar, onde a lista de descontos é definida com base em votos dos usuários, tentando assim mostrar o maior número possível de descontos ativos para o usuário, esses descontos variam entre várias categorias, desde eletrodomésticos até joias e em geral são disponibilizados para compras online. Por serem descontos indicados por usuários da plataforma, e não diretamente pelos vendedores, é difícil saber se os descontos realmente estão ativos ou não.

O Peixe urbano também é um agrupador de descontos, mas estes são disponibilizados pela própria empresa do aplicativo, ou seja, a compra é feita direto no aplicativo e retirado na loja física, dessa forma não é disponibilizado diretamente pelo vendedor. Além disso, o Peixe urbano tem como diferencial o filtro de descontos por região e o pagamento online.

O Cuponeria é o aplicativo mais similar, além de ser disponibilizado por região os cupons são exibidos com base em uma categoria, uma loja ou uma marca escolhida. Também mostra a validade do cupom de desconto e permite utilizar o desconto mostrando o código do cupom na hora do pagamento da compra no estabelecimento. A diferença aqui é o foco para compras online.

Todos os aplicativos buscados tem um foco semelhante, o de possibilitar a compra de produtos com descontos, facilitando a compra para o usuário, porém o diferencial maior que o Conto teria é que os descontos são focados mais em pequenas áreas, dessa forma as pequenas empresas locais são beneficiadas, pois uma empresa do tamanho da Amazon, ou do McDonalds, por exemplo, não precisa desse tipo de visibilidade, ao passo que restaurantes

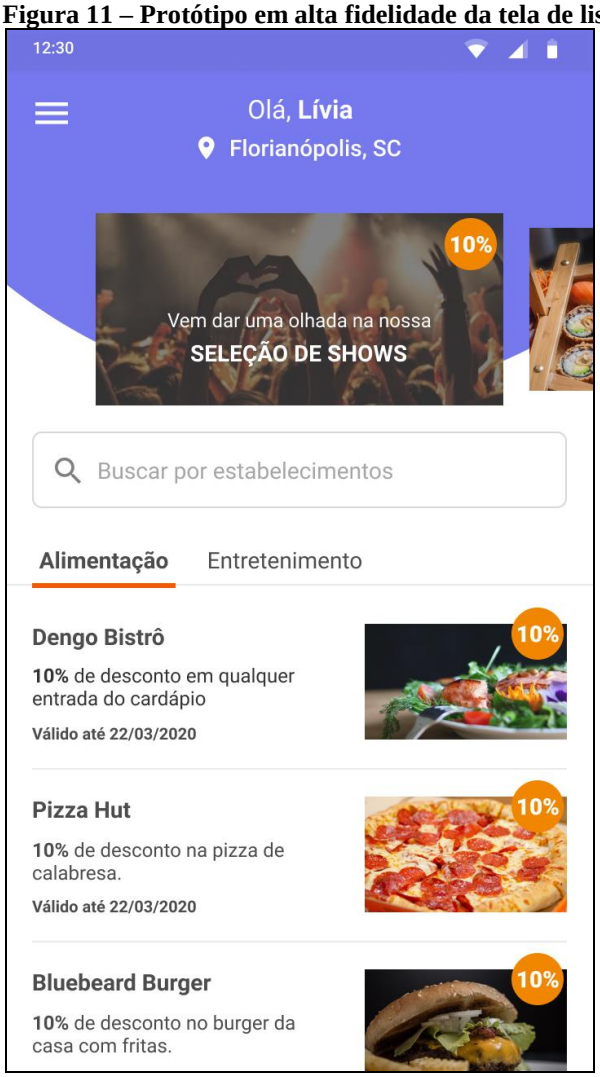
locais, ou casas de show locais passam despercebidos por clientes que estão acostumados a comprar de grandes cadeias. Dessa forma o Conto descontos não apenas ajuda no cliente, que encontra descontos e dessa forma economiza, mas também no comércio local, que ganha clientes.

O aplicativo possibilita a exibição de todos os outros produtos disponibilizados pelo vendedor, seja ele com desconto ou não, dessa forma o cliente pode escolher comprar outros produtos ou voltar outros dias. Portanto a economia local seria impactada positivamente com o uso do aplicativo por moradores de regiões não tão focadas pelos aplicativos maiores e mais conhecidos que acabam beneficiando empresas que já tem nome e capital formado.

4.2 PROJETO DO PROTÓTIPO

Para iniciar o desenvolvimento do protótipo, foram definidas as telas que seriam essenciais para a funcionalidade *core* do aplicativo, e a partir disso começar o protótipo. Então foi percebido que, para o protótipo, seria necessária uma tela que listasse todos os descontos ativos na região, possibilitando selecionar a categoria de estabelecimento, que são estabelecimentos de alimentação (restaurantes, lanchonetes, bares) e estabelecimentos de entretenimento (casas de show, teatro, cinema).

Figura 11 – Protótipo em alta fidelidade da tela de listagem de descontos



Fonte: Acervo do Autor (2020)

A partir da listagem, é necessário ter a tela de detalhes da oferta, que exibe a descrição do desconto, a sua porcentagem, seu código único, se ele é válido para qualquer opção do estabelecimento ou para algum produto específico, se o desconto se encontra ativo e sua validade. Ainda nessa tela temos as regras do uso do desconto e a localização do estabelecimento.

Figura 12 – Protótipo em alta fidelidade da tela de detalhes do desconto

Fonte: Acervo do Autor (2020)

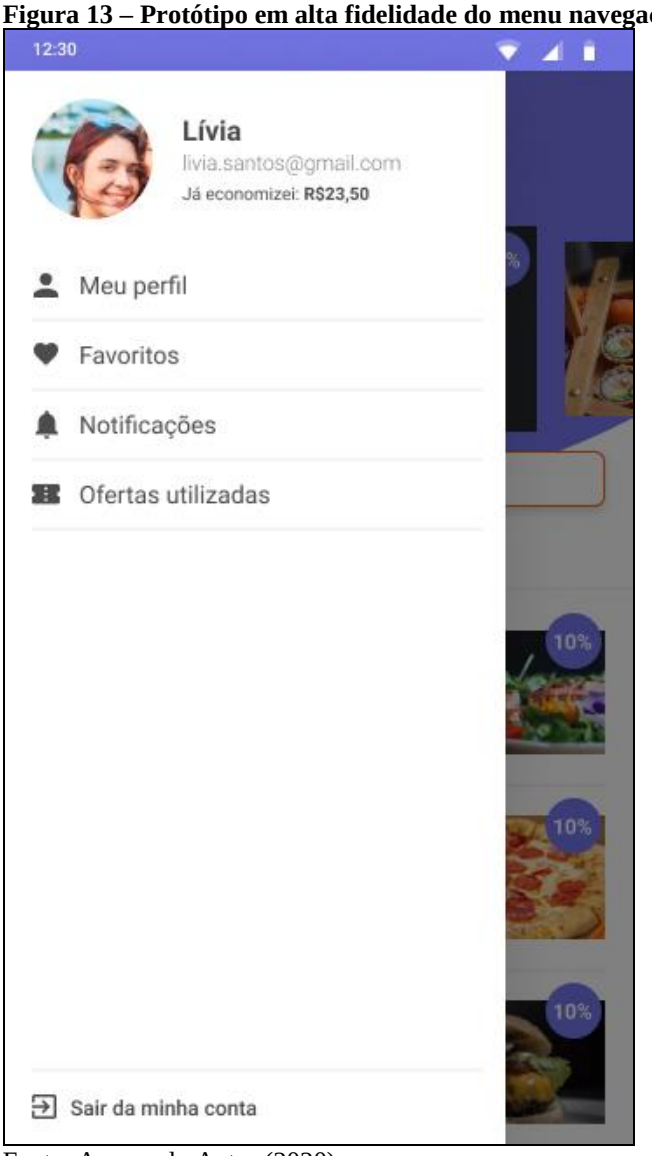
A tela dos detalhes do desconto serve como porta de entrada para todas as opções do estabelecimento, então, a partir dela temos uma navegação em páginas, onde a primeira página mostra os detalhes, conforme descrito anteriormente, a segunda página mostra todos os produtos do estabelecimento, como um menu de um restaurante, com suas categorias e seus valores, aqui não importa se os produtos tem desconto no momento ou não, serve para que o estabelecimento possa mostrar ao cliente tudo que oferece.

A terceira página serve para mostrar todas as outras ofertas ativas do estabelecimento, assim como a listagem da página inicial, porém apenas com as ofertas do estabelecimento atual, e, por consequência, da mesma categoria do desconto selecionado. Os itens dessa lista têm a mesma funcionalidade da lista inicial, onde ao clicar em um dos descontos o aplicativo navega para a tela de detalhes do desconto selecionado.

Além disso, em qualquer uma das páginas, é possível marcar como favorito o desconto atual para que possa ser encontrado de maneira mais rápida no futuro, e um botão de navegação no topo que sempre retorna para a tela inicial do aplicativo.

Na página inicial, existe um botão de menu, que abre uma *navigation drawer* com as opções relacionadas ao perfil do usuário, um botão para o Meu perfil, que permite alterar a senha atual; um botão Favoritos, que mostra os descontos que o usuário marcou como favorito, onde é possível clicar em um dos descontos dessa lista e será levado para a tela de detalhes do desconto; o botão para as Notificações do usuário; um botão para as Ofertas utilizadas onde é exibido as ofertas que o usuário já utilizou, aqui a lista não tem cliques, pois as ofertas já não estão ativas; e por último a opção para sair da conta.

Figura 13 – Protótipo em alta fidelidade do menu navegação



Fonte: Acervo do Autor (2020)

Com as telas em mente, foi possível iniciar o desenvolvimento do protótipo, para seguir com o desenvolvimento, foi feito o modelo inicial do banco de dados.

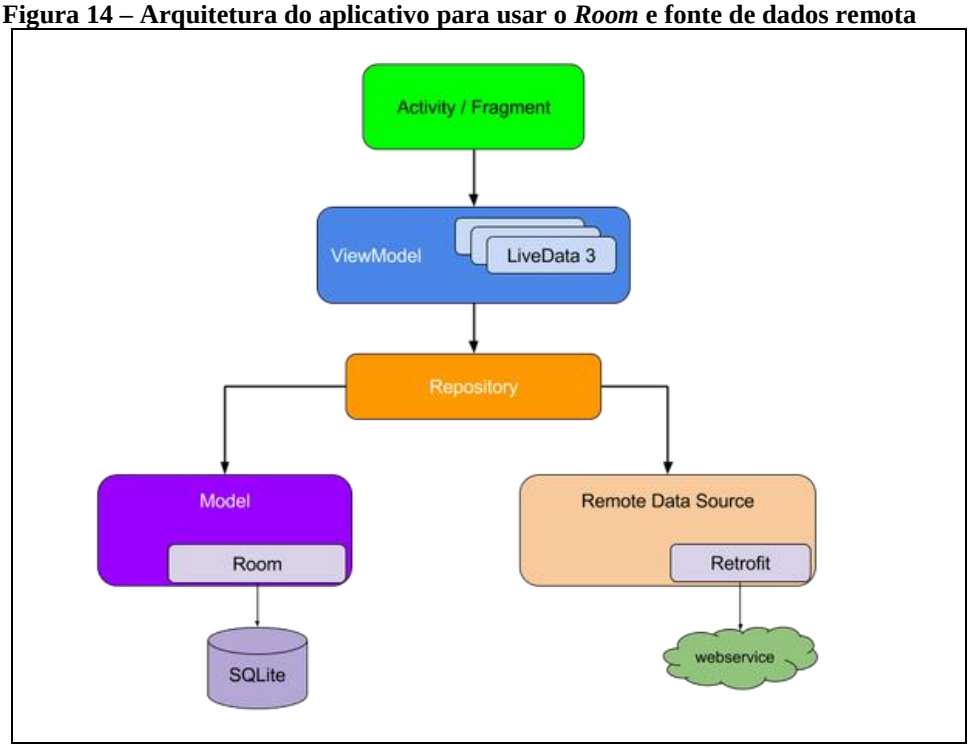
4.3 MODELO DE DADOS

O modelo de dados foi construído baseado nas funcionalidades definidas anteriormente, para o protótipo esses dados ficam armazenados localmente, mas para trabalhos futuros e para que o aplicativo possa ser publicado, é necessário armazenar esses dados em um servidor remoto, para que cada usuário possa receber as informações em qualquer lugar, porém como a ideia era realizar o protótipo com as funcionalidades básicas, essa parte não foi desenvolvida.

Com isso em mente foi buscado qual o banco de dados seria utilizado, como o Android tem uma integração já pronta com o SQLite a partir da biblioteca *Room*, parte da Android *Jetpack*, foi decidido utilizar essa ferramenta. Além de já estar incluso na Android SDK, outra vantagem de utilizar o *Room*, é que, em conjunto com outras ferramentas, como o uso do padrão *Observer* e o uso de *LiveData*, outra biblioteca inclusa no Android *Jetpack*, faz com o que o Android execute a atualização e busca da base dados já de forma assíncrona, não utilizando a *Thread* principal, e dessa forma a busca dos dados não trava a parte visual do aplicativo.

Outra vantagem dessa busca ser realizada fora da *thread* principal, é que, futuramente, ao fazer a ligação do banco de dados com um *Web Service* que grave os dados na internet, o aplicativo já está pronto para esta funcionalidade, pois o *Room* trabalha como um repositório local, que pode ser preenchido a partir de uma fonte de dados remota sem ser necessário alterar a forma como a aplicação busca e salva as informações. Como pode ser verificado na figura 14.

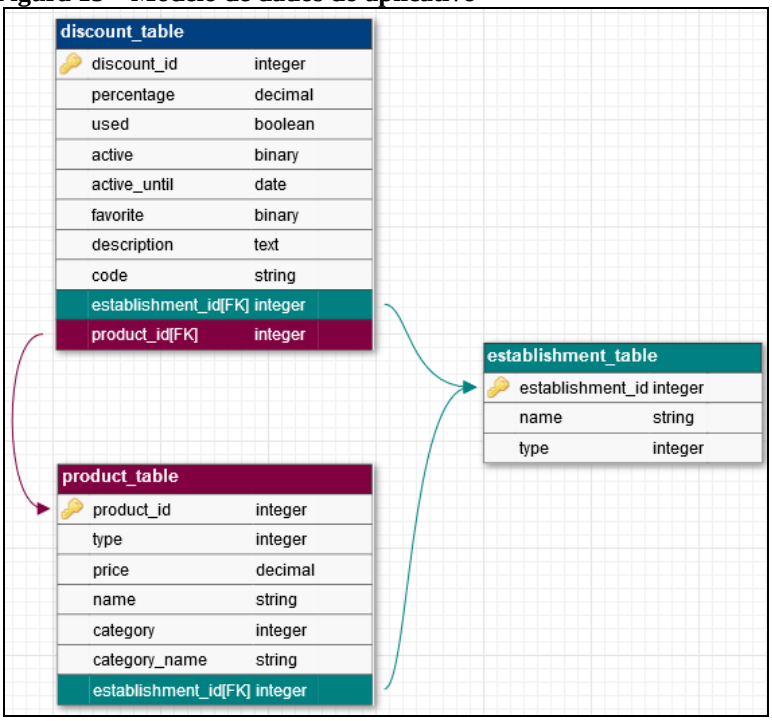
Figura 14 – Arquitetura do aplicativo para usar o *Room* e fonte de dados remota



Fonte: Google (2020)

Então, com o banco de dados escolhido, foi pensado nas tabelas e informações que cada entidade necessita, pensando no protótipo, foi decidido que o usuário não teria, no momento, uma tabela, pois para o protótipo o que importa são os estabelecimentos, produtos e descontos. Portanto foi chegado no modelo como demonstrado na figura 15.

Figura 15 – Modelo de dados do aplicativo



Fonte: Acervo do Autor (2020)

Pode ser visto no modelo (Figura 15) que foi decidido usar a língua inglesa para nomenclatura das entidades e das informações. Essa decisão se estende para todo o código, como será visto nas próximas seções do trabalho. Foi escolhido fazer dessa maneira pelo fato de o inglês ser a língua padrão do Android, e para o desenvolvimento em geral. Dessa maneira o aplicativo fica mais bem estruturado, escalável e apresentável para uma audiência maior. Apesar disso, todas as telas e qualquer interface com o usuário é feita somente em português.

Como exibido no modelo, temos três tabelas essenciais para a aplicação: desconto; produto e estabelecimento. Todo produto obrigatoriamente tem uma ligação de 1 para 1 com estabelecimento, pois não existe um produto sem um vendedor. Já o desconto tem uma ligação obrigatória com o estabelecimento, porém opcional para o produto, isso significa que uma oferta pode servir para qualquer produto oferecido pelo estabelecimento ou apenas um produto em específico.

Ainda sobre o desconto, o modelo permite apenas descontos baseados em porcentagem, ou seja, não irá existir descontos do tipo “10 reais de desconto na primeira compra” ou outras situações como essa, apenas porcentagem para facilitar o uso e entendimento do comprador. Também pode ser visto o campo *code* que é uma *String* composta de 5 valores alfanuméricos gerados aleatoriamente um traço e a porcentagem do desconto, conforme código na figura 16.

Figura 16 – Método de geração do código do desconto

```
1 fun generateDiscountCode(length: Int, percentage:Int) : String {
2     val allowedChars = ('A'..'Z') + ('a'..'z') + ('0'..'9')
3     val code = (1..length)
4         .map { allowedChars.random() }
5         .joinToString("")
6     return "$code-$percentage"
7 }
```

Fonte: Acervo do Autor (2020)

Neste código, pode ser notado os dois parâmetros na linha 1, o tamanho da *string* aleatória a ser gerada, que foi definido que sempre será 5. O segundo parâmetro é a porcentagem do desconto, que é concatenada posteriormente com o código aleatório, na linha 6. A linha 2 exibe os valores aceitos pelo gerador de texto aleatória. O principal deste código é na linha 4, com o método *random*, que é código padrão disponibilizado pelo Kotlin para retornar um padrão aleatório partindo da coleção usada.

Outro ponto a ser notado no modelo de dados, é o campo *type* presente na tabela de estabelecimentos e de produtos, esses campos representam se o estabelecimento vende produtos de alimentação ou de entretenimento, os dois tipos aceitos no aplicativo. Esses valores são definidos por constantes no código, e caso, o aplicativo passe a aceitar novas categorias, estas deverão ser incluídas.

Mais uma observação sobre a tabela de produtos são os campos *category* e *category_name* esses valores também servem para identificar os produtos mais especificamente além do tipo, é usado para listar os outros produtos do estabelecimento de forma categorizada e organizada, conforme definido no código na figura 17.

Figura 17 – Categorias disponíveis para produtos

```
enum class FoodCategory (val intValue: Int, val stringValue: String){
    ENTRADA_FRIA(1, "Entrada Fria"),
    ENTRADA_QUENTE(2, "Entrada Quente"),
    PRINCIPAL(3, "Prato Principal"),
    SOBREMESA(4, "Sobremesa"),
    BEBIDA(5, "Bebida")
}
enum class EntertainmentCategory (val intValue: Int, val stringValue: String){
    SHOW(1, "Show"),
    TEATRO(2, "Teatro"),
    CINEMA(3, "Cinema")
}
```

Fonte: Acervo do Autor (2020)

Na figura 17, podemos verificar que as categorias são divididas para cada um dos tipos, e também que o *enum* em questão recebe um valor inteiro, para salvar como valor inteiro no banco, e o campo em *string* para ser utilizado posteriormente na exibição na tela de produtos.

Uma última decisão importante quanto a nomenclatura das tabelas, foi fazer a nomenclatura genérica, para que fosse possível usar tanto para estabelecimentos de alimentação ou de entretenimento, assim como qualquer futura adição. Foi escolhido *establishment* e *product* ao invés de nomes como restaurante e prato, pois dessa forma não é confuso para situações onde o tipo não fosse de alimentação.

Também, pode ser verificado na tabela de desconto os campos *used*, *active* e *active_until* esses campos representam se o desconto foi utilizado, se está ativo e até qual data permanecerá ativo. Um desconto se torna inativo quando a data atual é maior que a data de

active_until ou quando foi utilizado pelo usuário, nesse caso além de inativo, ele se torna usado.

4.4 IMPLEMENTAÇÃO DO BANCO DE DADOS

Com o modelo de dados pronto e o banco de dados selecionado, foi iniciado a implementação do modelo com o *Room*. Para implementar um banco de dados usando o *Room* são necessárias ao menos três partes. Primeiro uma classe representando cada uma das tabelas, depois uma ou mais interface que represente o DAO do banco de dados, e por último uma classe abstrata que represente o banco de dados por completo, que estenda a classe *RoomDatabase*, essa classe deve ter acesso aos DAOs e uma função que possa ser chamada para receber uma instancia do banco de dados. Essas partes serão demonstradas nas seções seguintes.

4.4.1 *Entities*

Essa seção do modelo usado pelo Room tem o objetivo de representar cada uma das tabelas do banco com seus respectivos campos. O código para a entidade de desconto está visível na Figura 18.

Figura 18 – Classe Kotlin representando a tabela de desconto

```kotlin
package com.unidavi.tc.conto.database

import androidx.room.*

@Entity(tableName = "discount_table")
data class Discount(
    @PrimaryKey(autoGenerate = true)
    var discountId: Long = 0L,
    var percentage: Int = -1,
    var active: Boolean = true,
    var favorite: Boolean = false,
    var description: String = "",
    var code: String = "",
    @ColumnInfo(name = "active_until")
    var activeUntil: Long = System.currentTimeMillis(),
    @ForeignKey(
        entity = Establishment::class,
        parentColumns = ["establishment_owner_id"],
        childColumns = ["discountId"],
        onDelete = ForeignKey.RESTRICT
    )
    @ColumnInfo(name = "establishment_owner_id")
    var establishmentOwnerId: Long = 0,
    @Embedded var establishment: Establishment = Establishment()
)
```

Fonte: Acervo do Autor (2020)

Na figura 18 podemos ver como funciona o *Room* para criar uma tabela no banco, primeiro é necessário importar as bibliotecas do Room, para que seja possível usar as anotações necessárias. Então a classe deve estar anotada com *@Entity* (linha 5), para que o Android saiba que essa classe representa uma tabela da base de dados a anotação necessita o parâmetro que define o nome da tabela no modelo de dados, esse é o nome que é usado posteriormente nos comandos SQL para selecionar as informações da tabela.

Depois de anotado a classe pode ser aberta, no caso do Kotlin existe um modificador de classe específico para classes cujo único objetivo é armazenar informação, sem realizar nenhum cálculo sobre eles, como é o caso de qualquer entidade do banco de dados. Esse modificador é o *data* antes do *class*, como verificado na linha 6.

Em seguida é obrigatório ter ao menos uma variável que esteja anotada com *@PrimaryKey* (linha 7) com o parâmetro *autoGenerate*, que no caso de ser verdadeiro, o Room se encarrega de gerar o valor sempre que uma entidade for inserida, nesse caso é um valor inteiro sequencial.

Todas as outras variáveis podem ter uma anotação opcional *@ColumnInfo* com um parâmetro *name*, esse parâmetro é opcional pois serve para definir o nome do campo na tabela, caso a variável não seja anotada, o nome da tabela será o mesmo nome da variável, como visto na linha 14.

Outra anotação opcional é *@ForeignKey* (linha 16), como o nome indica, ela informa que a variável anotada é uma chave estrangeira para outra entidade. Ela necessita dos parâmetros *Entity* (linha 17), que é a classe que representa a tabela apontada pela chave estrangeira; o parâmetro *parentColumns* (linha 18) que é o campo, ou conjunto de campos, que é a chave estrangeira da tabela pai; o *childColumns* (linha 19) é a chave primária da tabela filha; e por fim o parâmetro *onDelete* (linha 20), que informa ao *Room* o que deve ser feito caso a entidade pai seja excluída.

A última anotação utilizada é a *@Embedded* (linha 24), esta anotação traz as informações de um *join* junto na pesquisa. Para exemplificar, no caso do desconto, ao selecionar o campo *establishment* de um *discount* é possível acessar os campos da tabela *establishment* sem precisar realizar um *join* explicito com a tabela filha.

As outras tabelas seguem o mesmo modelo do exemplo demonstrado na figura 18, portanto não serão detalhadas no trabalho.

4.4.2 D*ata Access Object*

As classes DAO são o ponto de acesso principal as interações com o banco de dados, são anotadas com @Dao e a classe deve ser abstrata ou uma interface. O *Room* então utiliza essas informações para gerar uma classe de implementação em tempo de compilação quando referenciada pela base de dados. A figura 19 mostra um exemplo de uma classe DAO em Kotlin para acessar a tabela de produto.

Figura 19 – Classe Kotlin representando o ponto de acesso a tabela de produtos

```
1 @Dao
2 interface ProductDao {
3     @Insert(onConflict = OnConflictStrategy.REPLACE)
4     fun insertProduct(product: Product) : Long
5
6     @Update
7     fun updateProduct(product: Product)
8
9     @Delete
10    fun deleteProduct(product: Product)
11
12    @Query("SELECT * FROM product_table " +
13           " WHERE type = :type " +
14           " AND establishment_owner_id = :establishmentKey ORDER BY category")
15    fun getProductsFromTypeOrderedByCategory(type: Int, establishmentKey: Long): LiveData<List<Product>>
16
17    @Query("SELECT establishment_owner_id FROM product_table WHERE productId = :key")
18    fun getEstablishmentFromDiscount(key: Long): Long?
19 }
```

Fonte: Acervo do Autor (2020)

Com o exemplo da figura 19, é possível verificar algumas situações, como o fato de ser uma interface anotada com @Dao (linha 1), essa é a maneira de o Android saber que esse é o ponto de acesso à tabela do banco. Não é necessário ter uma interface para cada entidade, porém, para o código ficar mais organizado e simples de manter, foi escolhido fazer uma para cada tabela.

Também é visível no exemplo outras anotações: a anotação *@Insert* (linha 3) que gera automaticamente o comando para inserir na tabela, dessa maneira não é necessário se preocupar com esse tipo de query. O parâmetro onConflict é opcional, caso não seja passado nenhum, por padrão, o Room usa a estratégia de abortar caso haja um conflito ao inserir. Foi decidido utilizar a estratégia de substituir caso haja um conflito, pois caso seja adicionado um produto com uma chave primária já existente, é necessário substituir todas as informações do registro.

Essa anotação insere o elemento por inteiro, com todas as informações presentes no objeto. No exemplo é retornado um valor *Long* (linha 4), no caso o campo definido como a

chave primária da entidade, não é necessário retornar essa informação, mas foi escolhido retornar para que seja possível atualizar posteriormente as outras informações do produto.

A anotação *@Update* (linha 6) é semelhante com a *Insert*, pois o código da atualização da entidade é gerado automaticamente pelo *Room*. O registro a ser atualizado é buscado a partir da chave primária do objeto passado como parâmetro e todas as informações presentes no objeto são alteradas, o parâmetro obrigatoriamente deve ser um objeto de uma classe identificada pela anotação *@Entity*. Caso não seja encontrada um registro com a chave primária do objeto passado, nada acontece na base de dados.

O *@Delete* (linha 9) simplesmente apaga o registro a partir da chave primária do objeto passado como parâmetro, caso o registro tenha alguma chave estrangeira, é usado a estratégia definida pela anotação da classe de entidade.

Por último a anotação *@Query* (linhas 12 e 17) que recebe como parâmetro uma *String*, que é o comando SQL a ser executado. Aqui é possível utilizar quantos parâmetros forem necessários para construir o *query* e também é definido o tipo de retorno da consulta. A consulta pode ser tão complexa quanto necessária, dentro dos limites do SQLite.

Com essas anotações podem ser feitos todos os acessos necessários à tabela de banco de dados, aqui é importante notar que dessa maneira não é necessário passar um comando SQL toda vez que for necessário alterar a base de dados, facilitando o acesso aos dados e também diminuindo a manutenção posterior, além de separar as camadas da aplicação e dados, aumentando, dessa forma, a escalabilidade do código.

Todas as anotações, exceto a *@Query*, podem receber uma lista de objetos ao invés de um objeto único, no caso do protótipo não foi necessário, pois toda inclusão de informação é feita de maneira unitária.

4.4.3 *Database*

A classe que representa todo o banco de dados é a classe que agrupa todas as entidades e todos os DAOs necessários para compor o banco de dados como um todo. Para informar ao Android que essa classe representa um banco de dados *Room* ela é anotada com *@Database*, passando como parâmetros um *array* das entidades e a versão atual do banco, que deve ser alterada quando é feita alguma alteração no modelo de dados.

Além disso, a classe é abstrata e estende à *RoomDatabase*. Tem todos os DAOs necessários definidos como variáveis de classe e uma função que cria uma instância do banco de dados quando chamada, ou retorna a instância já existente, evitando duplicidade.

Essa classe não varia muito entre aplicativos, pois é um padrão onde as únicas informações que devem ser alteradas são as entidades e os DAOs utilizados, o restante acaba sendo sempre semelhante.

4.5 TESTES NO BANCO DE DADOS

A parte de banco de dados é essencial em qualquer aplicação, portanto, para garantir que o armazenamento, recuperação e atualização de dados no banco estão funcionando corretamente, foram desenvolvidos métodos de teste para validar os métodos de manipulação de banco de dados. Os métodos mais importantes como o de inserir descontos, buscar descontos ativos e inativos, atualizar os descontos foram testados conforme demonstrado na figura 20.

Figura 20 – Teste automático para verificação de métodos no banco de dados

```
1  fun insertAndGetActiveDiscounts(){
2          Timber.i("Insert and get all active discounts test beginning")
3          val activeDiscount = Discount()
4          val inactiveDiscount = Discount ()
5
6          activeDiscount.discountId = discountDao.insertDiscount(activeDiscount)
7          inactiveDiscount.discountId = discountDao.insertDiscount(inactiveDiscount)
8          var activeDiscounts = discountDao.getActiveDiscounts()
9          var inactiveDiscounts = discountDao.getInactiveDiscounts()
10
11         assertEquals(activeDiscounts.size, 2)
12         assertEquals(inactiveDiscounts.size, 0)
13
14         inactiveDiscount.active = false
15         discountDao.updateDiscount(inactiveDiscount)
16
17         activeDiscounts = discountDao.getActiveDiscounts()
18         inactiveDiscounts = discountDao.getInactiveDiscounts()
19
20         assertEquals(activeDiscounts.size, 1)
21         assertEquals(inactiveDiscounts.size, 1)
22
23         Timber.i("Insert and get all active discounts test ending")
24     }
```

Fonte: Acervo do Autor (2020)

Aqui verificamos que o método de inserir um desconto (linha 6) está funcionando corretamente, visto que um desconto por padrão é criado como ativo, a consulta de descontos ativos retorna dois num primeiro momento (linha 11). Em sequência um dos descontos é inativado (linha 14) e atualizado com o método anotado com o *@Update* na classe da entidade desconto (linha 15). Portanto ao consultar os descontos ativos novamente, o tamanho deve ser 1 (linha 20) assim como o tamanho da lista de descontos inativos (linha 21).

O Android Studio separa as classes de teste automaticamente do restante do projeto, separando em um pacote distinto que não é levado em conta quando compilado o projeto para gerar a APK do aplicativo, dessa forma não ocupa espaço no celular cuja instalação do aplicativo seja efetuada.

4.6 DESENVOLVIMENTO DO PROTÓTIPO

Com o banco de dados preparado, foi iniciado o desenvolvimento do protótipo em si. A primeira tela a ser desenvolvida foi a tela inicial com a listagem dos descontos ativos separados em duas *tabs*, uma para o tipo alimentação e outra para o tipo entretenimento. A decisão nesse momento foi a de fazer uma *Activity* que contêm a barra de topo do aplicativo, seguido pelas páginas de tipo e então dois *Fragments*, um para cada lista de descontos por tipo.

Então com a lista pronta foi passado para a tela de detalhes do desconto. De modo similar a tela principal, foi feito uma atividade que contem a barra de topo do aplicativo, com o nome do estabelecimento, uma imagem de fundo e três *tabs*, uma que mostra a lista de produtos e outra que mostra a lista de outros descontos do estabelecimento.

Depois foi adicionado o menu de navegação lateral na tela inicial, que levam para informações do usuário, seu perfil, suas notificações, uma lista de descontos favoritos e uma de descontos utilizados. Aqui a navegação para cada uma dessas opções inicia uma nova atividade, exceto para as duas listas que levam para uma atividade que contêm um fragmento, podendo dessa forma reutilizar o design definido para as outras listas do aplicativo.

Por fim foi decidido incluir ao iniciar o aplicativo uma *splash screen* que exibe a logo e o nome do aplicativo por um curto espaço de tempo.

4.6.1 Tela inicial

Ao desenvolver a tela inicial, a maior questão foi como criar duas páginas com informações distintas. Para realizar esta função de maneira correta é necessário ter, no arquivo de *design* da atividade, após a definição da barra do tipo do aplicativo, primeiramente o elemento da Google *Material Design* chamado *TabLayout*, e logo em seguida um outro elemento da Android *Jetpack* chamando *ViewPager*

Para fazer com que as páginas sejam construídas e exibam cada uma sua lista, foi necessário programar uma classe que estenda a classe, disponibilizada pelo Android SDK, *FragmentPagerAdapter*, o que essa classe faz é definir a quantidade de *tabs*, seus títulos e o

fragmento para cada uma delas. Então, ao criar a atividade, foi chamado um método que configura este adaptador e adiciona os fragmentos desejados, no caso o fragmento que lista os descontos de alimentação e o que lista os de entretenimento, conforme demonstrado na Figura 21.

Figura 21 – Configuração das *tabs* na atividade principal

```
1  class MainActivity : AppCompatActivity(), NavigationView.OnNavigationItemSelectedListener {
2      override fun onCreate(savedInstanceState: Bundle?) {
3          super.onCreate(savedInstanceState)
4          setContentView(R.layout.activity_main)
5          setSupportActionBar(toolBar_main)
6          [...]
7          setUpTabs()
8      }
9
10     private fun setUpTabs(){
11         val adapter = ViewPageAdapter(supportFragmentManager)
12         adapter.addFragment(FoodTypeFragment(), "Alimentação")
13         adapter.addFragment(EntertainmentTypeFragment(), "Entretenimento")
14         viewPager_main.adapter = adapter
15         tabs_main.setupWithViewPager(viewPager_main)
16     }
```

Fonte: Acervo do Autor (2020)

A figura 21 mostra apenas as partes relevantes para a criação das *tabs* no método *setUpTabs()*. Como pode ser notado, esse método é chamado após toda a configuração inicial da atividade no o*nCreate* (linha 7). Primeiro é criado o adaptador (linha 11), e nesse adaptador adicionado os dois fragmentos, com seu respectivo título (linhas 12 e 13). Com o adaptador pronto, é feito a ligação dele com o *ViewPager* (linha 14) e, por fim, com o *TabLayout* (linha 15).

Outro detalhe importante do código da figura 21 é o parâmetro do construtor de *ViewPageAdapter* (linha 11), *o supportFragmentManager*, que é automaticamente criado pelas atividades Android, justamente com o intuito de facilitar o trabalho de fazer um ou mais fragmentos filhos da atividade.

Em sequência, na figura 22 será mostrado o código do adaptador, para que fique claro como é feita a criação das páginas.

Figura 22 – *FragmentPagerAdapter*

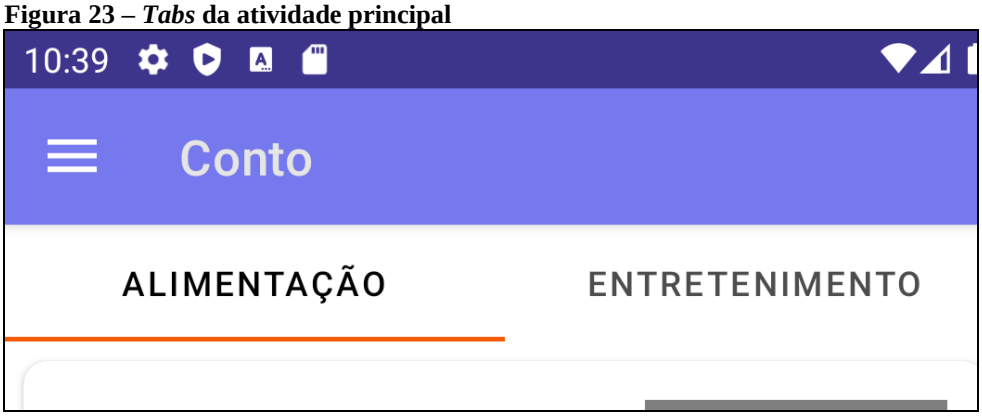
```
1 class ViewPageAdapter(supportFragmentManager: FragmentManager) :
2  FragmentPagerAdapter(supportFragmentManager, BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT){
3     private val mFragmentList = ArrayList<Fragment>()
4     private val mFragmentTitleList = ArrayList<String>()
5
6     override fun getCount(): Int {
7         return mFragmentList.size
8     }
9
10    override fun getItem(position: Int): Fragment {
11        return mFragmentList[position]
12    }
13
14    override fun getPageTitle(position: Int): CharSequence? {
15        return mFragmentTitleList[position]
16    }
17
18    fun addFragment(fragment: Fragment, tittle : String){
19        mFragmentList.add(fragment)
20        mFragmentTitleList.add(tittle)
21    }
22 }
```

Fonte: Acervo do Autor (2020)

O adaptador tem duas listas, uma para os fragmentos (linha 4) e uma para os títulos de cada página (linha 5), o método *addFragmente* (linha 18) serve para adicionar um fragmento e relacioná-lo ao título. As outras funções são utilizadas pelo Android para criar a quantidade de *tabs* correta, na posição correta dentro do *layout* da tela e com o título correspontende. Vale notar também o parâmetro *BEHAVIOR_RESUME_ONLY_CURRENT_FRAGMENT* (linha 2), o que ele faz é forçar o fragmento a chegar no estado *resumed* do ciclo de vida, dessa forma passando por todos os métodos importantes da classe do fragmento.

O resultado dessa parte é demonstrado pela tela do aplicativo na figura 23, que mostra apenas o topo do aplicativo, para exibir as duas páginas criadas. Em seguida no trabalho será explicado como foi desenvolvido cada um dos fragmentos das páginas de alimentação e entretenimento.

Figura 23 – *Tabs* da atividade principal



Fonte: Acervo do Autor (2020)

4.6.2 Fragmentos filhos da atividade principal

Os dois fragmentos utilizados nas páginas criadas anteriormente são *layouts* que contém um *RecyclerView* (linha 17), disponível a partir da Android *Jetpack*. Envolto por um *FrameLayout* (linha 12), que serve para usar toda a tela disponível. Além disso foi decidido usar *data binding* (linha 7) para vincular os dados com a interface do usuário automaticamente, para isso o XML do layout do fragmento deve possuir uma variável (linha 8) para gravar um *ViewModel* (linha 9 e linha 10). A figura 24 exibe o código XML do fragmento.

Figura 24 – *Layout* do fragmento de descontos do tipo alimentação

```
1  <?xml version="1.0" encoding="utf-8"?>
2
3  <layout xmlns:android="http://schemas.android.com/apk/res/android"
4      xmlns:app="http://schemas.android.com/apk/res-auto"
5      xmlns:tools="http://schemas.android.com/tools">
6
7      <data>
8          <variable
9              name="discountViewModel"
10             type="com.unidavi.tc.conto.viewModels.DiscountViewModel" />
11     </data>
12     <FrameLayout
13         android:layout_width="match_parent"
14         android:layout_height="match_parent"
15         tools:context=".main.FoodTypeFragment">
16
17         <androidx.recyclerview.widget.RecyclerView
18             android:id="@+id/food_product_offer_list"
19             android:layout_width="match_parent"
20             android:layout_height="match_parent"
21             app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager" />
22     </FrameLayout>
23 </layout>
```

Fonte: Acervo do Autor (2020)

O *recyclerView* (linha 17) funciona com uma lista, que armazena vários elementos, nesse caso cada elemento é um desconto do tipo alimentação. Para que seja exibido a lista de forma correta, e que faça sentido para o usuário é necessário um segundo *layout* em XML para um elemento da lista, este segundo layout está demonstrado na figura 25.

Para este layout foi decidido em uma visualização de *cards* (linha 16), que contém as informações necessárias para o desconto, no caso temos uma imagem (linha 25 a 36) que, para o protótipo é apenas um *placeholder* (linha 32), mas para desenvolvimento futuro essa imagem será armazenada em um *web service*, para que seja diferente para cada um dos descontos. Além da imagem temos três textos (linhas 38, 51 e 52), o principal com o nome do estabelecimento um secundário com a porcentagem do desconto e outro com a validade do

desconto, onde no protótipo foi decidido, por praticidade adicionar o texto Validade indeterminada ou Desconto inativo caso o desconto não esteja ativo.

Este layout também utiliza de *data binding* (linha 6) onde possui uma variável que guarda um registro de desconto (linha 7) e um *clickListener* (linha 11) que é usado para a função de clique que leva para a tela de detalhe do desconto. A figura 25 exibe o código XML do *layout* individual do desconto, com algumas partes omitidas para demonstrar apenas as partes relevantes e não repetidas.

Figura 25 – *Layout* de um desconto

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <layout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto"
4      xmlns:tools="http://schemas.android.com/tools">
5
6      <data>
7          <variable
8              name="discount"
9              type="com.unidavi.tc.conto.database.Discount" />
10
11         <variable
12             name="clickListener"
13             type="com.unidavi.tc.conto.adapters.DiscountListener" />
14     </data>
15
16     <androidx.cardview.widget.CardView
17         android:id="@+id/card_discount"
18         [...]
19         android:onClick="@{() -> clickListener.onClick(discount)}">
20
21         <androidx.constraintlayout.widget.ConstraintLayout
22             android:layout_width="match_parent"
23             android:layout_height="match_parent">
24
25             <ImageView
26                 android:id="@+id/imageView_discount"
27                 android:layout_width="124dp"
28                 android:layout_height="71dp"
29                 android:layout_marginTop="16dp"
30                 android:layout_marginEnd="16dp"
31                 android:layout_marginBottom="16dp"
32                 android:src="@drawable/discount_image_placeholder"
33                 app:layout_constraintBottom_toBottomOf="parent"
34                 app:layout_constraintEnd_toEndOf="parent"
35                 app:layout_constraintTop_toTopOf="parent"
36                 android:contentDescription="@string/str_image_discount_description" />
37
38             <TextView
39                 android:id="@+id/text_discount_establishment_name"
40                 android:layout_width="0dp"
41                 android:layout_height="wrap_content"
42                 android:layout_marginStart="16dp"
43                 android:layout_marginTop="16dp"
44                 android:layout_marginEnd="16dp"
45                 android:textSize="14sp"
46                 android:textStyle="bold"
47                 app:discountEstablishmentName="@{discount}"
48                 app:layout_constraintEnd_toStartOf="@+id/imageView_discount"
49                 app:layout_constraintStart_toStartOf="parent"
50                 app:layout_constraintTop_toTopOf="parent" />
51             <TextView ... />
52             <TextView ... />
53         </androidx.constraintlayout.widget.ConstraintLayout>
54     </androidx.cardview.widget.CardView>
55 </layout>
```

Fonte: Acervo do Autor (2020)

Observa-se então que este layout tem a variável que aponta para a classe da entidade de desconto no *Room data base* (linha 9), portanto pode ser utilizado qualquer campo do

desconto para preencher o *layout*, como será demonstrado nas próximas seções do trabalho. Outro detalhe importante é o atributo *android:onClick* (linha 19) da *tag cardView* que recebe como parâmetro uma função lambda definida no *clickListener*. Além disso, na *tag TextView* o atributo *app:discountEstablishmentName* (linha 47) recebe como parâmetro a variável *discount* que é utilizada para preencher o texto de forma dinâmica usando *data binding*. Importante notar que os outros *TextViews* levam um atributo parecido, porém omitido na figura, pois são semelhantes ao código exibido. O código usado para preencher essas informações será exibido no decorrer do trabalho. As definições do estilo de cada uma das *views* não é tão relevante para o trabalho, mas é importante notar que cada uma possui um *Android:id* distinto, essa informação é importante para posteriormente vincular cada informação com a *view* correta.

Com esses dois arquivos XML o layout do fragmento de listagem de desconto está pronto e é necessário o código Kotlin para preencher as informações necessárias antes da exibição da tela ao usuário. Ao verificar o ciclo de vida de um fragmento, que pode ser visto na figura 07, nota-se que o código responsável por essa parte deve ser incluso no método *onCreateView*.

Neste método será realizado o *data binding*, ou seja, o vínculo entre os dados e a interface com o usuário. Também são usados os padrões *model-view-viewmodel*, *ViewModelFactory* e *observer*, todos aconselhados pela parte de arquitetura de projeto do Android *Jetpack*, por isso foi decidido seguir com essa arquitetura. Além desses modelos, também está presente o *adapter* que adapta os valores de cada item da lista para o layout único.

Na figura 26 está o método *onCreateView* do fragmento de listagem de descontos do tipo alimentação.

Figura 26 – Método *onCreateView* do fragmento

```
1 class FoodTypeFragment : Fragment() {
2     override fun onCreateView(
3         inflater: LayoutInflater,
4         container: ViewGroup?,
5         savedInstanceState: Bundle?
6     ): View? {
7         val binding: FragmentFoodTypeBinding = DataBindingUtil.inflate(
8             inflater, R.layout.fragment_food_type, container, false
9         )
10        val application = requireNotNull(this.activity).application
11        val dataSource = ContoDataBase.getInstance(application)
12        val viewModelFactory = DiscountViewModelFactory(dataSource, application, TYPE_FOOD, 0)
13        val discountViewModel =
14            ViewModelProvider(this, viewModelFactory).get(DiscountViewModel::class.java)
15        val adapter = DiscountAdapter(DiscountListener { discount ->
16            discountViewModel.onDiscountClicked(discount)
17        })
18        binding.foodProductOfferList.adapter = adapter
19        discountViewModel.discounts?.observe(viewLifecycleOwner, {
20            it?.let {
21                adapter.submitList(it)
22            }
23        })
24        discountViewModel.navigateToDiscountDetail.observe(viewLifecycleOwner, { discount ->
25            discount?.let {
26                val intent = Intent(this.context, DiscountActivity::class.java)
27                intent.putExtra("discountKey", discount.discountId)
28                intent.putExtra("establishmentName", discount.establishment.name)
29                intent.putExtra("establishmentKey", discount.establishment.establishmentId)
30                startActivity(intent)
31                discountViewModel.onDiscountDetailNavigated()
32            }
33        })
34        return binding.root
35    }
36 }
```

Fonte: Acervo do Autor (2020)

Temos de importante no código, a variável *binding* (linha 7), que é responsável por inflar o layout XML, ou seja, preencher as informações de cada *view* do *layout* e mostrar elas na tela, temos como um dos parâmetros o XML do fragmento exibido na figura 24. Em seguida a variável *dataSource* (linha 11) que é a classe anotada com *@Database* conforme explicado nas seções anteriores do trabalho.

O v*iewModelFactory* (linha 12) é construído com os parâmetros *datasource*, explicado anteriormente; a aplicação, que serve como contexto para garantir que a fonte de dados seja sempre a mesma; e dois inteiros: o primeiro à partir de uma constante *TYPE_FOOD*, que é usada para filtrar os descontos; e o último parâmetro, que é usado para reaproveitar o ViewModel em outros fragmentos, como será exibido posteriormente no trabalho.

Então usando o *ViewModelProvider* disponibilizado pela Android *Jetpack*, é construído um *ViewModel* (linha 14) baseado no próprio fragmento como contexto, e na classe *factory* passada como parâmetro. A classe *DiscountViewModel* será detalhada no decorrer do trabalho.

O *adapter* (linha 15) também está sendo construído nesse método. Ele recebe como parâmetro uma função lambda que registra qual o item da lista que foi clicado e executa a função definida no parâmetro (linha 16). Esse adaptador então é vinculado à lista de descontos

do *RecyclerView* da variável *binding* (linha 18) que faz a ligação de cada elemento da lista para um elemento do *layout* XML.

E por fim no método *onCreateView* temos os dois métodos *observe*, que observa se há alguma alteração no elemento observado e, quando houver, executa a função lambda passada como parâmetro. No exemplo temos o observador da lista de descontos (linha 19), que sempre que acontece alguma alteração é atualizada na tela do *smartphone*. E o *observer* da variável do *ViewModel navigateToDiscountDetail* (linha 24), que é alterada no clique, e então é iniciado a segunda atividade do programa (linha 30), que exibe os detalhes do desconto selecionado, passando algumas informações importantes (linhas 26 a 29) para a atividade chamada. São essas informações: o Id do desconto (linha 27); o nome do estabelecimento do desconto (linha 28) e o Id do estabelecimento (linha 29).

A classe *DiscountViewModel* está detalhada na figura 27 e a *DiscountAdapter* na figura 28. O *ViewModelFactory* não terá detalhamento pois é um código padrão que pouco muda quando utilizado em qualquer modelo.

Figura 27 – Classe *DiscountViewModel*

```
1  class DiscountViewModel(
2      dataBase: ContoDataBase,
3      application: Application,
4      type: Int,
5      discountId: Long
6  ) : AndroidViewModel(application) {
7      val establishmentId = dataBase.discountDao.getEstablishmentIdFromDiscount(discountId)
8      val discounts = when (type){
9          -1 -> establishmentId?.let {
10             dataBase.discountDao.getAllDifferentActiveDiscountsFromEstablishment(
11                 establishmentId,
12                 discountId
13             )
14         }
15         -2 -> dataBase.discountDao.getFavoritesDiscounts()
16         -3 ->dataBase.discountDao.getInactiveDiscounts()
17         else -> dataBase.discountDao.getAllActiveDiscountsFromEstablishmentType(type)
18     }
19     private val _navigateToDiscountDetail = MutableLiveData<Discount>()
20     val navigateToDiscountDetail
21         get() = _navigateToDiscountDetail
22
23     fun onDiscountClicked(discount: Discount) {
24         _navigateToDiscountDetail.value = discount
25     }
26
27     fun onDiscountDetailNavigated() {
28         _navigateToDiscountDetail.value = null
29     }
30 }
```

Fonte: Acervo do Autor (2020)

No *DiscountViewModel* o ponto principal é a variável discounts (linha 8), que é preenchido buscando as informações da base de dados que é passada como parâmetro no construtor. Aqui, para reutilizar a classe para todos os modelos de desconto usados no programa, essa variável é preenchida com base no parâmetro *type*. No caso da tela inicial a

busca realizada é a do caso *else* (linha 17) pois é passado um dos valores de tipo de desconto, comida ou entretenimento que tem como valores 0 e 1, respectivamente. O uso de valores negativos nas outras condições é para que, caso seja adicionado tipos novos de estabelecimentos, não será necessário alterar essa classe. Uma breve explicação das outras possíveis condições: quando -1, é buscado todos os outros descontos ativos do estabelecimento atual (linha 9); quando -2, busca todos os favoritos (linha 15); e quando -3 busca os descontos já utilizados (linha 16).

Como o Kotlin utiliza variáveis com tipo inferido, não é possível perceber na figura 27, porém a variável *discounts* é do tipo LiveData<List<Discount>>, o retorno da consulta é feito dessa forma para que a lista seja um LiveData, e dessa forma possa ser observada pelo padrão *observer*. Além de garantir que a consulta seja feita em uma *thread* separada da principal.

A variável *navigateToDiscountDetail* (linhas 19 e 20) segue a mesma lógica, pois ela é observada para navegar para a atividade do desconto selecionado.

Figura 28 – Classe *DiscountAdapter*

```
1  class DiscountAdapter(val clickListener: DiscountListener): ListAdapter<Discount, DiscountAdapter.ViewHolder>
   (DiscountDiffCallback()) {
2
3      override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
4          return ViewHolder.from(parent)
5      }
6      override fun onBindViewHolder(holder: ViewHolder, position: Int) {
7          val item = getItem(position)
8          holder.bind(item, clickListener)
9      }
10
11     class ViewHolder private constructor(val binding: ListItemDiscountBinding) :
   RecyclerView.ViewHolder(binding.root){
12         fun bind(item: Discount, clickListener: DiscountListener) {
13             binding.discount = item
14             binding.clickListener = clickListener
15             binding.executePendingBindings()
16         }
17         companion object {
18             fun from(parent: ViewGroup): ViewHolder {
19                 val layoutInflater = LayoutInflater.from(parent.context)
20                 val binding = ListItemDiscountBinding.inflate(layoutInflater, parent, false)
21                 return ViewHolder(binding)
22             }
23         }
24     }
25 }
26
27 class DiscountDiffCallback : DiffUtil.ItemCallback<Discount>() {
28     override fun areItemsTheSame(oldItem: Discount, newItem: Discount): Boolean {
29         return oldItem.discountId == newItem.discountId
30     }
31
32     override fun areContentsTheSame(oldItem: Discount, newItem: Discount): Boolean {
33         return oldItem == newItem
34     }
35 }
36
37 class DiscountListener(val clickListener: (discount: Discount) -> Unit) {
38     fun onClick (discount:Discount) = clickListener(discount)
39 }
```

Fonte: Acervo do Autor (2020)

Na figura 28 temos algumas particularidades do Kotlin, que permite classes distintas no mesmo arquivo, como é o caso da classe D*iscountDiffCallback* (linha 27) que é usada para verificar se dois itens da lista são iguais e fazer a mudança das informações do layout com base nisso. E também da classe *DiscountListener* (linha 37) que passa o desconto que foi clicado para a função de navegação para a próxima atividade.

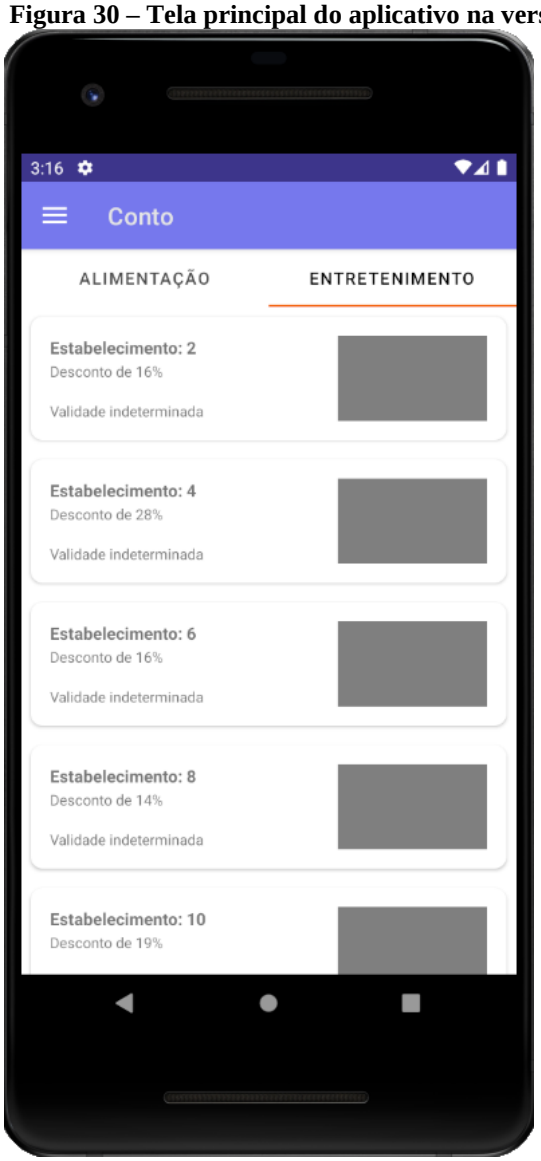
Além dessas duas temos a classe principal do arquivo *DiscountAdapter* (linha 1), e uma classe interna *ViewHolder* (linha 11). A classe externa serve basicamente para chamar os métodos de ligação da classe interna *ViewHolder* que realiza a ligação entre os dados do desconto e o layout a ser inflado na função *bind* (linha 12). O método *binding.executePendingBindings()* é gerado automaticamente pelo Android, quando usado *data binding*. Tudo o que é necessário fazer para que a ligação seja feita então, é criar métodos anotados com *@BindingAdapter* e um parâmetro *string* que é usado no XML para saber qual view deve ser preenchida, como exemplificado na figura 29. Esses métodos podem estar presentes em qualquer arquivo, no meu caso fiz um arquivo *bindingUtils.kt* que contém todos os métodos necessários para a ligação de todas as telas.

Figura 29 – Exemplo de funcionamento do *data binding*

Fonte: Acervo do Autor (2020)

Dessa forma temos a tela inicial de listagem de descontos pronta, usando *data binding*, para que o vínculo de dados com a interface seja mais eficaz do que se fosse feito manualmente utilizando a metodologia de *findViewById*, usando *ViewModel* para ficar de acordo com os padrões de desenvolvimento sugeridos pela Google, garantindo suporte mais longo do aplicativo, e pelo mesmo motivo usando adaptadores e observadores para, junto com o *data binding*, realizar a atualização da tela e a navegação do aplicativo de forma correta e segura. O resultado final da tela se encontra na figura 30.

Figura 30 – Tela principal do aplicativo na versão final do protótipo



Fonte: Acervo do Autor (2020)

4.6.3 *Navigation drawer*

Para encerrar a tela inicial ficar totalmente pronta, é necessário o menu de navegação lateral, a *navigation drawer*. Para a criação desse menu, é necessário primeiro incluir novas tags no arquivo XML do *layout* da atividade principal. No topo da hierarquia do XML é necessário adicionar o *DrawerLayout*, parte do Android *Jetpack*, e no final da hierarquia um *NavigationView*, que é parte da biblioteca *Material design*. Conforme exibido na figura 31.

Figura 31 – Parte do XML referente ao menu de navegação lateral

```
1  <androidx.drawerlayout.widget.DrawerLayout
2          android:id="@+id/drawer_main"
3          android:layout_width="match_parent"
4          android:layout_height="match_parent"
5          android:fitsSystemWindows="true">
6
7          <RelativeLayout ... </RelativeLayout>
8
9          <com.google.android.material.navigation.NavigationView
10             android:id="@+id/navigation_drawer_main"
11             android:layout_width="wrap_content"
12             android:layout_height="match_parent"
13             android:layout_gravity="start"
14             android:background="@color/white"
15             android:fitsSystemWindows="true"
16             app:headerLayout="@layout/nav_header"
17             app:itemIconTint="@color/gray"
18             app:itemTextAppearance="@style/style_nav_item"
19             app:itemTextColor="@color/gray"
20             app:menu="@menu/menu_nav" />
21
22     </androidx.drawerlayout.widget.DrawerLayout>
```

Fonte: Acervo do Autor (2020)

No XML da figura 31 temos de importante, além das *tags*, os elementos *app:headerLayout* (linha 16) e o *app:menu* (linha 20) que recebem como parâmetro dois arquivos XML, um representando o cabeçalho do menu lateral e o outro o menu contendo os elementos que realizam a navegação. Esses arquivos são simples e contém informações que já foram apresentadas no decorrer do trabalho, portanto não serão detalhados.

O código para lidar com a navegação está demonstrado na figura 32, e será explicado em sequência.

Figura 32 – Controle da navegação pela navigation *drawer*

```
1 class MainActivity : AppCompatActivity(), NavigationView.OnNavigationItemSelectedListener {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         [...]
4         val toggle = ActionBarDrawerToggle(
5             this,
6             drawer_main,
7             toolBar_main,
8             R.string.open_drawer,
9             R.string.close_drawer
10        )
11        drawer_main.addDrawerListener(toggle)
12        navigation_drawer_main.setNavigationItemSelectedListener (this)
13        setUpTabs()
14    }
15    [...]
16    override fun onNavigationItemSelected(item: MenuItem): Boolean {
17        when (item.itemId){
18            R.id.nav_item_profile -> {
19                val intent = Intent(this.applicationContext, ProfileActivity::class.java)
20                startActivity(intent)
21                return true
22            }
23            R.id.nav_item_favorites -> {
24                val intent = Intent(this.applicationContext, FavoritesActivity::class.java)
25                startActivity(intent)
26                return true
27            }
28            R.id.nav_item_notifications -> {
29                val intent = Intent(this.applicationContext, NotificationActivity::class.java)
30                startActivity(intent)
31                return true
32            }
33            R.id.nav_item_used_offers -> {
34                val intent = Intent(this.applicationContext, UsedOffersActivity::class.java)
35                startActivity(intent)
36                return true
37            }
38            else -> return false
39        }
40    }
41 }
```

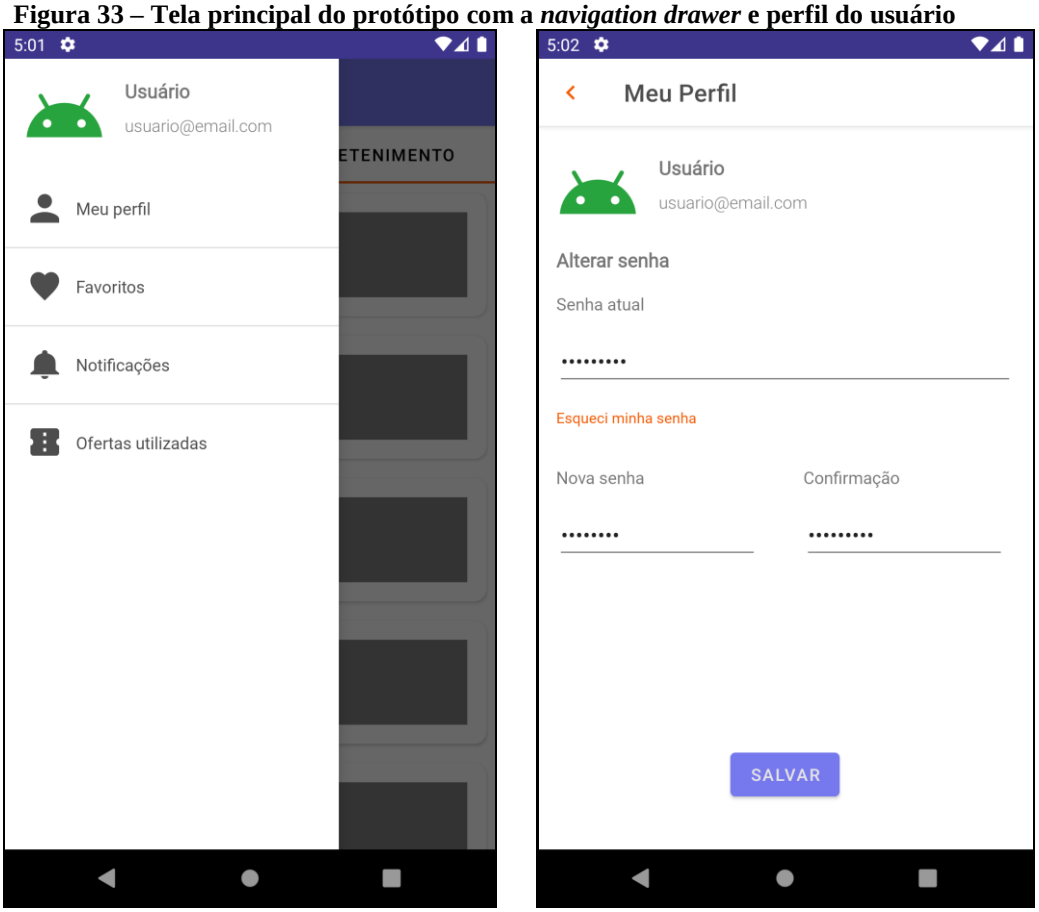
Fonte: Acervo do Autor (2020)

A variável *toggle* (linha 4) é relacionado à ação de expandir e esconder o menu lateral, ela recebe como parâmetro, a própria atividade como contexto (linha 5), o menu lateral e a barra do topo do aplicativo (linhas 6 e 7) e duas s*trings* (linhas 8 e 9) que servem como descrição da funcionalidade, no caso abrir e fechar o menu lateral, por questões de acessibilidade. Essa variável é então vinculada ao *DrawerLayout* da atividade (linha 11) através de um listener, que é aberta ao arrastar para a direita, ou ao clicar no botão do topo esquerdo da atividade. A segunda parte importante é o método *onNavigationItemSelected* (linha 16 – 40) que simplesmente inicia a atividade referente a opção selecionada.

Aqui foram tomadas algumas decisões, como o trabalho é um protótipo, e como explicado anteriormente, foi escolhido não controlar o usuário no protótipo, as navegações para o Meu perfil e para as Notificações tem apenas o layout, mas sem a funcionalidade. Já a navegação para os Favoritos e para as Ofertas utilizadas tem a funcionalidade correta e exibem a lista de descontos relacionada.

A listagem de descontos utilizados e favoritos segue o mesmo princípio explicado anteriormente na seção 4.6.2, portanto não se faz necessário o mesmo detalhamento. Quanto aos arquivos de layout do Meu perfil e das Notificações, estes também seguem modelos já apresentados, então não convém nova explicação. A figura 33 exibe o resultado do menu de navegação lateral, com o exemplo da navegação para o layout da tela de perfil do usuário

Figura 33 – Tela principal do protótipo com a *navigation drawer* e perfil do usuário



Fonte: Acervo do Autor (2020)

4.6.4 Tela de detalhes do desconto

A atividade de detalhes do desconto, assim como a inicial é composta por uma barra de topo do aplicativo, onde, nesse caso contém o nome do estabelecimento e uma imagem, que, assim como a do desconto, por motivos de praticidade, no protótipo é apenas um *placeholder*, e para desenvolvimento futuro essa imagem será armazenada remotamente e obtida através de um *web service*.

Abaixo da barra de topo, foram desenvolvidas três páginas, uma para os detalhes do desconto, uma para exibir todos os produtos do estabelecimento e outra para os outros descontos ativos do estabelecimento. Cada *tab* com um fragmento. A forma de controle dessas páginas e dos fragmentos é igual à explicação da atividade principal.

A novidade nessa tela é a barra de topo, que ao invés de conter o botão para abrir a n*avigation drawer*, possui um botão que retorna para a atividade inicial e outro botão, no lado direito para favoritar o desconto. Além disso o título da tela é dinâmico, baseado no nome do estabelecimento do desconto.

A figura 34 contém o código da atividade do desconto, omitindo as partes que já foram vistas na explicação da atividade principal.

Figura 34 – Atividade de desconto

```
1  class DiscountActivity : AppCompatActivity() {
2
3      lateinit var discount: Discount
4      override fun onCreate(savedInstanceState: Bundle?) {
5          super.onCreate(savedInstanceState)
6          setContentView(R.layout.activity_discount)
7          val intent = intent
8          val discountKey = intent.extras?.getLong("discountKey")
9          discount = discountKey?.let {
10             ContoDataBase.getInstance(application).discountDao.getDiscountWithIdNoLiveData(
11                 it
12             )
13         }!!
14         setUpTabs()
15         onPrepareOptionsMenu(toolBar_discount.menu)
16         setSupportActionBar(toolBar_discount)
17     }
18     [...]
19     override fun setSupportActionBar(toolbar: Toolbar?) {
20         super.setSupportActionBar(toolbar)
21         supportActionBar?.title = discount.establishment.name
22         toolbar?.menu?.findItem(R.id.favorite)?.isVisible = true
23         toolbar?.setNavigationOnClickListener {
24             val intent = Intent(applicationContext, MainActivity::class.java)
25             intent.flags = FLAG_ACTIVITY_NEW_TASK or FLAG_ACTIVITY_CLEAR_TASK
26             startActivity(intent)
27         }
28
29     }
30
31     override fun onOptionsItemSelected(item: MenuItem): Boolean {
32         return when (item.itemId) {
33             R.id.favorite -> {
34                 if (discount.favorite) {
35                     item.icon = ContextCompat.getDrawable(
36                         this,
37                         R.drawable.ic_favorite_inactive_24dp
38                     )
39                     discount.favorite = false
40                 } else {
41                     item.icon = ContextCompat.getDrawable(this, R.drawable.ic_favorite_24dp)
42                     discount.favorite = true
43                 }
44                 ContoDataBase.getInstance(application).discountDao.updateDiscount(discount)
45                 return true
46             }
47             else -> super.onOptionsItemSelected(item)
48         }
49     }
50 }
```

Fonte: Acervo do Autor (2020)

Nesta atividade, temos o uso de uma variável com o modificador *lateinit* (linha 3). Esta é a forma do Kotlin garantir o *null safety* da variável, sem exigir que ela seja instanciada

no momento de criação do objeto da classe. O que acontece é que o compilador, ao ler esse modificador, aceita a iniciação da variável como *null*, desde que o programador saiba disso e garanta que ela não será nula no momento que for usada. No caso essa variável é utilizada no método *onCreate*, que sempre é executado. Além disso a busca do desconto está denotada com o '?.*let'* (linha 9) e encerrado com '!!' (linha 13) que são as duas formas que fazem o Kotlin garantir que não seja nulo, pois o '?' gera uma exceção de *NullPointer* caso o retorno seja nulo, e o '!!' é como um *assert not null*, dessa forma fica garantido que o desconto nunca será nulo quando for necessário usá-lo.

Ainda sobre a variável de desconto, é possível notar no código que a variável é buscada a partir do campo extra enviado junto com o i*ntent* de iniciar a atividade (linha 8). Esse extra, o *discountKey* é um valor *long*, e é usado para buscar o desconto específico.

O método *setSupportActionBar* (linha 19) é responsável por alterar dinamicamente o título da barra do aplicativo de acordo com o nome do estabelecimento (linha 21), além de configurar o funcionamento do botão de navegação do topo esquerdo da tela (linha 23). Esse botão limpa todas as atividades na fila de atividades do aplicativo (linha 25) e inicia a atividade principal novamente (linha 26). Essa funcionalidade é importante pois garante que o botão de navegação, que é chamada de *up button* seja diferente do botão de retorno físico, o *back button.* A diferença é visível quando o usuário navega entre os outros descontos oferecidos pelo estabelecimento, o botão físico volta para a tela de detalhe do desconto anterior, enquanto que o *up button* sempre volta para a tela inicial. Essa limpeza da fila de atividades acontece devido ao uso do *intent* com as *flags FLAG_ACTIVITY_NEW_TASK* e *FLAG_ACTIVITY_CLEAR_TASK*.

Outra situação visível no código é o funcionamento do botão de favorito, que é configurado em *onOptionsItemSelected* (linha 31) e atualiza no banco de dados se o desconto foi marcado como favorito (linha 42) ou desmarcado (linha 39), além de alterar o ícone do botão para que represente o estado atual do desconto, um coração preenchido caso o desconto seja favorito (linha 41) ou o coração contornado (linha 37) caso não.

4.6.5 Fragmentos da atividade de detalhe do desconto

A atividade de detalhe do desconto é composta por 3 fragmentos, o que contém os detalhes do desconto, o que contém os outros produtos do estabelecimento e o que contém os outros descontos ativos.

Nesses fragmentos também é utilizado *data binding*, *ViewModel* e *viewModelFactory*, e, nos fragmentos de outras ofertas e outros descontos também é utilizado um adaptador. Apenas no fragmento de detalhes que não é necessário o adaptador, pois este fragmento mostra a informação de apenas um desconto, e não de uma lista.

Na tela de detalhes, é exibido o código único do desconto, onde, na aplicação real, ao utilizar o desconto, este código seria inserido na aplicação do vendedor, e isso faria com que o desconto passasse a ser usado. Como no protótipo não temos o aplicativo do vendedor, foi decidido que ao clicar no código, o desconto se torna inativo e usado, removendo, dessa forma, da tela inicial e adicionando na lista do fragmento de descontos utilizados. Essa interação é feita com o padrão *observer* assim como explicado anteriormente no trabalho. A função do *ViewModel* dos detalhes que inativa o desconto está exibida na figura 35.

Figura 35 – Inativando o desconto na classe *DiscountDetailsViewModel*

```
1  class DiscountDetailsViewModel(private  val discountKey: Long = 0L,
2                                 dataSource: ContoDataBase) : ViewModel(){
3
4      val database = dataSource
5
6      private val discount = database.discountDao.getDiscountWithId(discountKey)
7
8      private val _inactivateDiscount = MutableLiveData<Boolean?>()
9      val inactivateDiscount: LiveData<Boolean?>
10         get() = _inactivateDiscount
11
12     fun getDiscount() = discount
13
14     fun doneInactivatingDiscount(){
15         _inactivateDiscount.value = null
16     }
17
18     fun onInactivateDiscount (){
19         viewModelScope.launch {
20             val discount1 = database.discountDao.getDiscountWithIdNoLiveData(discountKey) ?: return@launch
21             discount1.active = false
22             database.discountDao.updateDiscount(discount1)
23
24             _inactivateDiscount.value = true
25         }
26     }
27 }
```

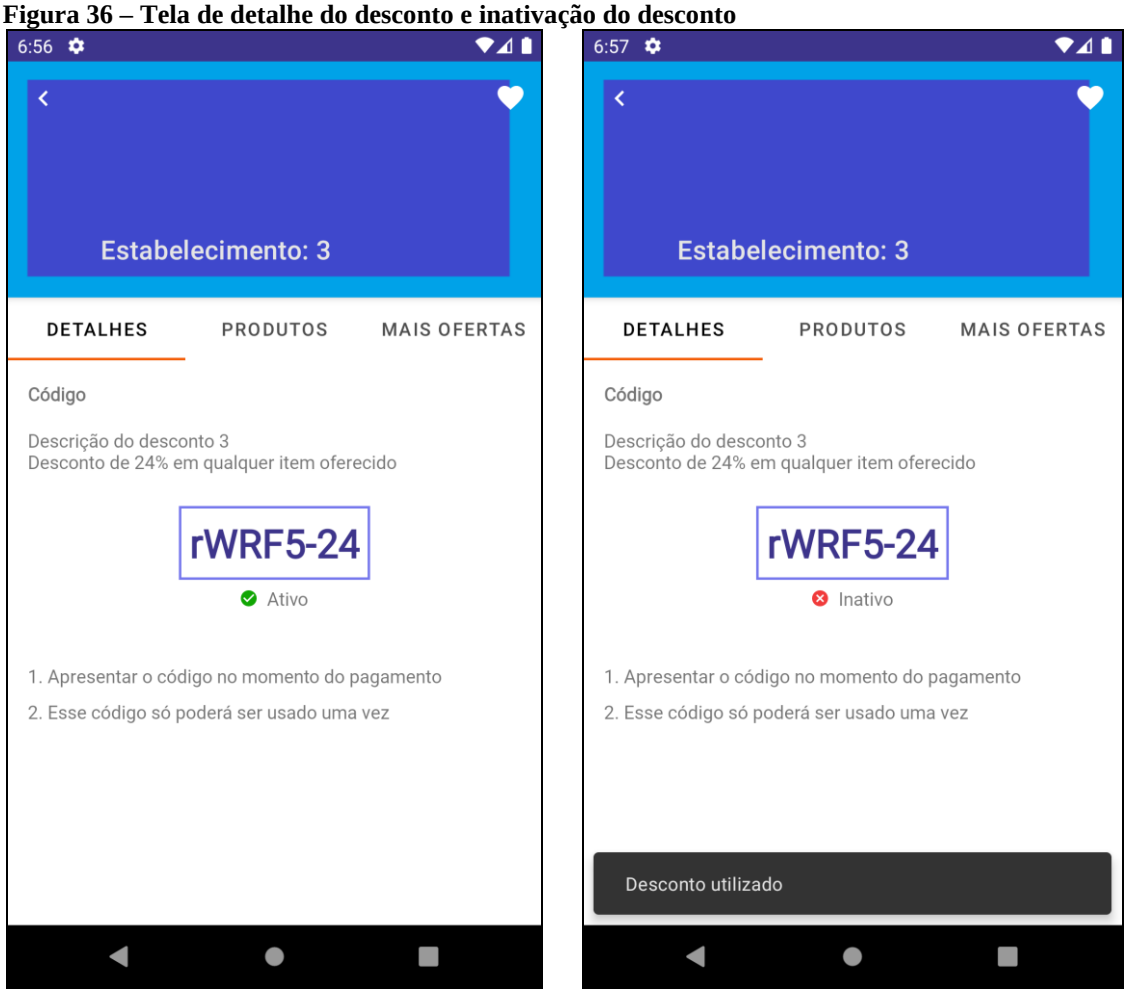
Fonte: Acervo do Autor (2020)

O método relevante para a inativação do desconto é o *onInactivateDiscount* (linha 18), aqui usamos a função dentro do *viewModelScope.launch* (linha 19), isso força a execução do que estiver dentro desse escopo em uma *thread* separada, afim de evitar o travamento da interface com o usuário. Nesse método se faz necessário o uso da função dessa forma pois a consulta ao banco utilizada retorna diretamente um desconto, e não um *liveData* do desconto. A anotação *return@launch* (linha 20) indica que a thread deve aguardar o retorno antes de continuar a execução do método, é necessário nessa situação para garantir que o desconto retornado pode ser alterado no decorrer do método. Uma vantagem do método desse modo é

que ele se torna observável, pois altera a variável *inactivateDiscount*, que é um *liveData* (linha 8) e, por consequência, observável.

A figura 36 exibe o resultado final da tela do fragmento de detalhe do desconto e a função de inativação do desconto.

Figura 36 – Tela de detalhe do desconto e inativação do desconto



Fonte: Acervo do Autor (2020)

O fragmento da página de produtos tem poucas novidades em relação ao que já foi apresentado até agora no trabalho. Ele tem um *productViewModel* contendo a lista para ser usada no *recycler view*, e um adaptador *productAdapter* que adapta cada item da lista para um layout específico. Esse layout também é bem simples, são apenas três *views* de texto onde uma representa a categoria do produto, uma o nome e a última o preço. A parte mais interessante desse fragmento é a função que controla a visibilidade da *view* que representa a categoria, que a remove caso já tenha algum produto na mesma categoria. Este método consta no adaptador e está demonstrado na figura 37.

Figura 37 – Método do adaptador de produto para esconder categorias repetidas

```
1 class ProductAdapter() : ListAdapter<Product, ProductAdapter.ViewHolder>(ProductDiffCallback()) {
2
3     override fun onBindViewHolder(holder: ViewHolder, position: Int) {
4         val item = getItem(position)
5         if (position > 0 && getItem(position - 1).categoryString == item.categoryString) {
6             holder.binding.textMenuCategory.visibility = View.GONE
7         } else {
8             holder.binding.textMenuCategory.visibility = View.VISIBLE
9         }
10        holder.bind(item)
11    }
12    [...]
13 }
```

Fonte: Acervo do Autor (2020)

O método verifica se a posição do item é maior que 0 e compara com a categoria do item anterior (linha 5), caso a categoria seja a mesma a visibilidade da *view* é passada para *GONE* (linha 6), esse nível de visibilidade é diferente de *INVISIBLE*, a diferença entre as duas é que a invisível ainda ocupa espaço no layout, já a *gone* é como se a *view* não existisse no layout, portanto não fica um espaço vazio na tela do usuário. Este método funciona, pois, a consulta realizada para buscar os produtos foi deliberadamente feita ordenando pela categoria, conforme a figura 38.

Figura 38 – Query de busca para a lista de produtos do estabelecimento

```
class ProductViewModel (val database: ContoDataBase,
                        application: Application,
                        establishmentId: Long) : AndroidViewModel(application) {

    val products = database.productDao.getProductsFromEstablishment(establishmentId)
    [...]
}
```

```
@Query("SELECT * FROM product_table " +
       " WHERE establishment_owner_id = :establishmentKey ORDER BY category")
fun getProductsFromEstablishment(establishmentKey: Long): LiveData<List<Product>>
```

Fonte: Acervo do Autor (2020)

A figura 39 exibe a versão final da tela do fragmento de outros produtos do estabelecimento.

Figura 39 – Tela de produtos do estabelecimento na versão final do protótipo

Fonte: Acervo do Autor (2020)

E por fim temos o fragmento mais ofertas. Tudo o que foi utilizado no desenvolvimento desse fragmento já foi explicado anteriormente no trabalho, o uso de *ViewModel*, adaptadores, *observers*, navegação com uso de *intents*. A diferença é unicamente o parâmetro *type* passado para o *DiscountViewModelFactory*, que é utilizado para buscar a lista de descontos a partir do id do estabelecimento.

4.6.6 *Splash screen*

Após o desenvolvimento de todos as telas definidas no protótipo, foi decidido programar uma tela inicial, chamada de *splash screen*. A função desta tela é mostrar ao usuário a marca do aplicativo, e usar este tempo de inicialização para buscar as informações importantes no repositório de dados. No caso do protótipo essa função não é tão necessária, visto que não é acessado a internet para a busca dos dados, porém futuramente esse tempo pode ser utilizado para buscar os dados do *web service* remotamente e validá-los com os dados atualmente em *cache*, evitando que o usuário inicie a aplicação e a mesma esteja travada acessando a internet.

O layout da atividade *splash* é composto apenas de um *background* que recebe um vetor de imagem contendo a logo e o nome da aplicação, e o código em Kotlin apenas exibe esta tela com um *delay* de 2 segundos antes de chamar a atividade principal, como demonstrado na figura 40.

Figura 40 – Código da tela de *splash*

```
1 class SplashActivity : AppCompatActivity() {
2     override fun onCreate(savedInstanceState: Bundle?) {
3         super.onCreate(savedInstanceState)
4         setContentView(R.layout.activity_splash)
5         supportActionBar?.hide()
6         Handler(Looper.getMainLooper()).postDelayed({
7             val intent = Intent(this@SplashActivity, MainActivity::class.java)
8             startActivity(intent)
9             finish()
10        },2000)
11    }
12 }
```

Fonte: Acervo do Autor (2020)

A classe não tem nada de muito complicado, apenas esconde a barra do aplicativo (linha 5), para utilizar a tela inteira do dispositivo e cria um *intent* para chamar a *MainActivity* (linha 7) depois de 2000 milisegundos (linha 10). Importante notar a chamada do método *finish* após o início da atividade seguinte (linha 9), isto se faz necessário para que ao voltar pelo botão físico do smartphone, quando na tela inicial, o aplicativo seja encerrado, e não retornado para a tela de *splash.*

Feito isso é necessário alterar o filtro de *intent* do aplicativo para que seja iniciado nesta atividade e não na *mainActivity*. Esta alteração é feita no manifesto do aplicativo, o arquivo gerado automaticamente pelo Android *Studio.* A versão final do

AndroidManifest.XML, e a tela de *splash* final do protótipo estão exibidas nas figuras 41 e 42 respectivamente.

Figura 41 – Manifesto do aplicativo

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <manifest xmlns:android="http://schemas.android.com/apk/res/android"
3     package="com.unidavi.tc.conto">
4
5     <application
6         android:name=".ContoApplication"
7         android:allowBackup="true"
8         android:icon="@mipmap/ic_launcher"
9         android:label="@string/app_name"
10        android:roundIcon="@mipmap/ic_launcher_round"
11        android:supportsRtl="true"
12        android:theme="@style/Theme.Conto">
13        <activity android:name=".UsedOffersActivity" />
14        <activity android:name=".FavoritesActivity" />
15        <activity android:name=".NotificationActivity" />
16        <activity android:name=".main.ProfileActivity" />
17        <activity android:name=".SplashActivity">
18            <intent-filter>
19                <action android:name="android.intent.action.MAIN" />
20
21                <category android:name="android.intent.category.LAUNCHER" />
22            </intent-filter>
23        </activity>
24        <activity android:name=".discount.DiscountActivity" />
25        <activity android:name=".main.MainActivity" />
26
27        <meta-data
28            android:name="preloaded_fonts"
29            android:resource="@array/preloaded_fonts" />
30    </application>
31 </manifest>
```

Fonte: Acervo do Autor (2020)

No manifesto, é percebido que a S*plashActivity* passou a ser chamada quando é clicado no ícone do aplicativo (linha 21) ou iniciada através de um possível *intent* externo (linha 19). Outros pontos a serem notados é o tema da aplicação, que é definido também no manifesto (linha 12) e é o que configura, por exemplo, as fontes do aplicativo e o esquema de cores. Aqui também é definido o ícone da aplicação (linha 8 e linha 10).

Figura 42 – *Splash screen*

Fonte: Acervo do Autor (2020)

Aqui é notado que a barra de aplicativo foi escondida para preencher a tela toda com a imagem.

5. CONCLUSÃO

Este trabalho apresentou o desenvolvimento de um protótipo de aplicativo em Kotlin para a plataforma Android, com o intuito de exibir descontos em estabelecimentos para que usuários possam utilizar de forma simples ao sair para se alimentar ou se entreter. Para alcançar o objetivo geral, foram usadas diversas bibliotecas disponibilizadas diretamente pela Google, de forma que o protótipo esteja atualizado com a melhor tecnologia disponível no momento, e garantindo que será possível manter por um longo tempo, sem se preocupar com o abandono das bibliotecas, como acaba acontecendo ao utilizar ferramentas de menor escopo.

O trabalho conseguiu cumprir com o objetivo específico de avaliar os componentes da plataforma Android, visto que cada elemento utilizado no desenvolvimento do programa foi pesquisado diretamente nas documentações disponibilizadas diretamente pela Google, então a informação é a mais atual possível, inclusive com páginas que foram atualizadas na mesma semana da leitura. Esta foi a principal parte do trabalho, pois estas pesquisas continuarão servindo para futuros trabalhos.

As telas do aplicativo seguiram heurísticas de usabilidade visto que foram baseados nos princípios do material Design, mesmo que o resultado final não tenha sido exatamente como o protótipo desenhado antes do início do desenvolvimento. O uso de banco de dados foi realizado de forma que futuramente, a persistência desses dados possa ser feita remotamente de forma simples, pois se baseou nos princípios de programação sugeridos pela Google para separar as camadas da aplicação e a camada de informação.

Os métodos de acesso ao banco de dados foram todos testados com o uso de testes automatizados. Cumprindo com o objetivo de usar testes no banco de dados.

O aplicativo cumpre com o objetivo de atender *tablets*, mesmo que não seja otimizado para esse tipo de mídia, pois nenhum layout foi feito com tamanhos fixos, e todos se baseiam da densidade de pixel da tela, logo eles ocuparão a tela de um *tablet* corretamente, mesmo que não tenha sido definido layouts específicos para este tamanho de tela.

Infelizmente não foi possível levar o protótipo a testes públicos, o que dificulta a avaliação final do protótipo, porém não tira a validez do trabalho que demonstrou a possibilidade de realizar um aplicativo de tamanho moderado com ferramentas atualizadas usando os conhecimentos adquiridos ao longo do curso

5.1 TRABALHOS FUTUROS

A principal evolução do protótipo, e essencial caso seja levado à público, é o uso de um *web service* para armazenar as informações em um servidor remoto. Junto com essa implementação é necessário também o controle de usuários e o uso da API de localização, que será o ponto principal da versão final do aplicativo, visto que o objetivo, além de ser benéfico para o comprador, é servir de vitrine para os estabelecimentos locais, não tão focados por outros aplicativos similares que preferem dar espaço à lojas multinacionais já conhecidas por todos.

Ainda ligado ao foco dado as lojas, se faz necessário em um trabalho futuro, o aplicativo para que as empresas possam controlar os seus descontos e visualizar dados, como quais desses descontos estão sendo mais utilizados, mais visualizados, mais marcados como favorito e outras informações pertinentes que podem auxiliar na evolução do seu negócio.

REFERÊNCIAS

GOOGLE. **Arquitetura da plataforma,** 2020. Disponível em: https://developer.android.com/guide/platform/index.html?hl=pt-br. Acesso em: 20 jun. 2020.

_____. **Fundamentos de aplicativos,** 2019. Disponível em: https://developer.android.com/guide/components/fundamentals?hl=pt_br. Acesso em: 20 jun. 2020.

_____. **Introdução a atividades,** 2019. Disponível em: https://developer.android.com/guide/components/activities/intro-activities?hl=pt_br. Acesso em: 20 jun. 2020.

_____. **Ciclo de vida de atividades,** 2019. Disponível em: https://developer.android.com/guide/components/activities/activity-lifecycle?hl=pt_br. Acesso em: 20 jun. 2020.

_____. **Provedores de conteúdo,** 2019. Disponível em: https://developer.android.com/guide/topics/providers/content-providers?hl=pt_br. Acesso em: 20 jun. 2020.

_____. **Serviços,** 2019. Disponível em: https://developer.android.com/guide/components/services?hl=pt_br Acesso em: 22 jun. 2020

_____. **Transmissões,** 2019. Disponível em: https://developer.android.com/guide/components/broadcasts?hl=pt-br Acesso em: 22 jun. 2020

_____. **Manifesto do aplicativo,** 2020. Disponível em: https://developer.android.com/guide/topics/manifest/manifest-intro Acesso em: 22 jun. 2020

_____. **Intents e filtro de intents,** 2019 Disponível em: https://developer.android.com/guide/components/intents-filters Acesso em: 22 jun. 2020

_____. **Android Studio,** 2020. Disponível em: https://developer.android.com/studio/intro Acesso em: 22 jun. 2020

_____. **Criar uma IU com o editor de layout,** 2020. Disponível em: https://developer.android.com/studio/write/layout-editor Acesso em: 22 jun. 2020

_____. **O processo de compilação,** 2020. Disponível em: https://developer.android.com/studio/build Acesso em: 23 jun. 2020

_____. **Conceitos básicos de testes,** 2020. Disponível em: https://developer.android.com/training/testing/fundamentals Acesso em: 23 jun. 2020

_____. **Android Jetpack,** 2020. Disponível em: https://developer.android.com/jetpack Acesso em: 23 jun. 2020

_____. **Salvar dados em um banco de dados local usando o Room,** 2020. Disponível em: https://developer.android.com/training/data-storage/room Acesso em: 20 nov. 2020

_____. **Como criar uma lista com o RecyclerView,** 2019. Disponível em: https://developer.android.com/guide/topics/ui/layout/recyclerview Acesso em: 20 nov. 2020

_____. **Fragmentos,** 2019. Disponível em: https://developer.android.com/guide/components/fragments Acesso em: 20 nov. 2020

_____. **Data Binding Library,** 2020. Disponível em: https://developer.android.com/topic/libraries/data-binding Acesso em: 20 nov. 2020

_____. **Guia para a arquitetura do app,** 2020. Disponível em: https://developer.android.com/jetpack/guide?hl=pt-br Acesso em: 20 nov. 2020

INSTITUTO BRASILEIRO DE GEOGRAFIA E ESTATÍSTICA. **Setor cultural ocupa 5,2 milhões de pessoas em 2008,** 2019. Disponível em: https://agenciadenoticias.ibge.gov.br/agencia-sala-de-imprensa/2013-agencia-de-noticias/releases/26235-siic-2007-2018-setor-cultural-ocupa-5-2-milhoes-de-pessoas-em-2018-tendo-movimentado-r-226-bilhoes-no-ano-anterior Acesso em: 28 jun. 2020

JORNADA, Izabela; AQUINO, Bruna. **Planejamento financeiro:** guardar dinheiro é a principal meta para 2020**,** 2020. Disponível em: https://www.correiodoestado.com.br/economia/planejamento-financeiro-guardar-dinheiro-e-a-principal-meta-para-2020/366180/ Acesso em: 28 jun. 2020

KLAUMANN, Thiago. **Como e por que o cupom de desconto deve ser utilizado por e-commerces e varejistas,** 2019. Disponível em: https://www.ecommercebrasil.com.br/artigos/como-e-por-que-o-cupom-de-desconto-deve-ser-utilizado-por-e-commerces-e-varejistas/ Acesso em: 28 jun. 2020

KOTLIN FOUNDATION. **Kotlin Language Documentation,** 2020. Disponível em: https://kotlinlang.org/docs/kotlin-docs.pdf. Acesso em: 22 jun. 2020

LECHETA, Ricardo R**. Google Android:** aprenda a criar aplicações para dispositivos móveis com Android SDK. São Paulo: Novatec, 2010. 608 p.

MISSIAGGIA, Mariana. **Como os brasileiros se alimentam fora de casa,** 2019. Disponível em: https://dcomercio.com.br/categoria/vida-e-estilo/como-os-brasileiros-se-alimentam-fora-de-casa Acesso em: 28 jun. 2020

MUNDO DO MARKETING. **Mercado do entretenimento:** prosperidade no Brasil**,** 2020. Disponível em: https://www.mundodomarketing.com.br/inteligencia/estudos/443/mercado-de-entretenimento-prosperidade-no-brasil.html Acesso em: 28 jun. 2020

PRICEWATERHOUSECOOPERS BRASIL LTDA. **Global consumers insights survey,** 2019. Disponível em: https://www.pwc.com.br/pt/estudos/setores-atividades/varejo/2019/con-insight-19.pdf Acesso em: 28 jun. 2020

STATCOUNTER. **Mobile Operating Systems market share Brazil,** 2020. Disponível em: https://gs.statcounter.com/os-market-share/mobile/brazil Acesso em: 28 jun. 2020

STATCOUNTER. **Mobile Operating Systems market share Worldwide,** 2020. Disponível em: https://gs.statcounter.com/os-market-share/mobile/worldwide Acesso em: 28 jun. 2020

VASIĆ, Miloš. **Fundamental Kotlin**: everything you need to know about kotlin. 2. ed. Miloš Vasić, 2017. E-book.

ANEXOS