Fast Market - Aplicativo para se fazer compras mais rápido

Ernesto Gonçalves Costa[1], André Ferreira Henriques de Souza[2]

[1]1712130014

[2]1712130065

Abstract. *In this work it is explained the development of a supermarket shop list app with the purpose of spending less time inside the supermarket during the pandemic.*

Resumo. *Neste trabalho é exemplificado o desenvolvimento de um aplicativo de lista de compras com a intenção de diminuir o tempo dentro de mercados durante a pandemia.*

1. Introdução

Durante a pandemia, diversos aspectos do cotidiano foram alterados. Devido a possibilidade de contágio [1], que em mercados pode ser maior, passar mais tempo fora de casa aumenta o risco de infecção.

No lockdown do Distrito Federal, foi definido no Diário Oficial [2] que apenas Serviços Essenciais iriam permanecer em funcionamento, sendo que o mais importante desses serviços são os mercados. Atualmente diversos mercados estão fazendo entregas via aplicativos, porém, por diversos motivos, pessoas ainda preferem, ou precisam, ir pessoalmente ao mercado, o que pode se tornar um risco.

O fluxo de pessoas tocando nos produtos pode aumentar o risco de contágio. Isso entra em conflito com o método no qual os mercados são pensados [3], que são organizados de modo a fazer com que o cliente passe mais tempo dentro dele e que ele compre mais produtos que não havia planejado.

Pensando nisso, neste trabalho será proposto um aplicativo que tem por objetivo gerar uma lista de compras em que o usuário irá marcar os produtos que precisa, e irá mostrar a localização dos produtos dentro do mercado, para que ele vá direto aos corredores que contém os produtos desejados, a fim de minimizar o tempo de exposição dentro do mercado.

2. Contextualização

Fazer compras é uma tarefa essencial na rotina de casa, e com o passar dos anos tem se transformado seguindo os acontecimentos sociais, como impactos econômicos e de saúde pública.

O padrão de consumo dos brasileiros sofreu mudanças significativas nos últimos anos, segundo pesquisa publicada pela Popai Brasil [5] em meados de 2010, "Comportamento do Consumidor em Super e Hipermercados", direcionado ao Marketing, o tempo gasto nos supermercados variou de trinta minutos a uma hora, com visitas mais frequentes e menor número de produtos comprados por vez.

Em 2020, com a pandemia do *Covid-19*, o comportamento de consumo do brasileiro sofreu grande alteração, apresentando preferências por compras online e padrão focado em estoque de alimentos e remédios [6,7,8,9,10], assim como muitos serviços iniciando serviços online, alguns inclusive migrando totalmente para este novo formato de comércio [11].

Temos o Brasil como um dos países mais afetados pela pandemia, onde se há um número grande e excessivo de mortes [12], sendo que o Brasil, em fevereiro de 2021, foi o país com o segundo maior número de mortes pelo *covid-19* do mundo, sendo que em março de 2021 houve registro de 2286 óbitos no período de um dia [13].

Porém, ainda é comum pessoas fazerem compras pessoalmente, por motivos diversos, ou por simples preferência, às vezes até como uma desculpa para sair de casa nos períodos de quarentena. Por conta disso, seria bom minimizar o tempo de exposição em lugares fechados com bastante gente, tais como supermercados.

3. Problema

Durante a pandemia, é complicado ficar muito tempo fora de casa, principalmente em locais muito cheios ou que há muito fluxo de pessoas, logo, as idas ao mercado se tornam um método comum de contágio. Levando em conta o fato de os mercados serem organizados de modo a fazer o cliente ficar o máximo de tempo possível dentro dele, o aplicativo a ser desenvolvido tem como intenção a resolução desse problema.

4. Objetivos

4.1. Objetivo Geral

Produzir um aplicativo de listas de compras que tem o intuito de acelerar as visitas aos mercados ao mostrar ao usuário a localização dos itens que ele precisa comprar para que ele vá direto ao seu interesse sem perder tempo olhando produtos que não planejava comprar.

4.1. Objetivo Específico

- Criar o design visual (protótipo) das telas.
- Criação da tela de login, cadastro e recuperação de senha.
- Criação das telas da *HomeActivity* (visualizar lista, criar nova lista, mercados próximos, a tela de “compra”, perfil e tela de *home*)
- Fazer *backend* do *login* utilizando Autenticação do *Firebase*
- Visualização das gôndolas do mercado através de polígonos desenhados no *Google Maps*
- Posição dos corredores no *Google Maps*
- Salvar listas em banco de dados local
- Recuperação de listas do banco de dados local para uso nos algoritmos
- Desenvolvimento de uma lógica para a passagem de produtos durante as compras

5. Referencial Teórico

Nesse momento serão descritas as ferramentas utilizadas no desenvolvimento do aplicativo.

5.1. FirebaseAuth

O *Firebase* é uma plataforma de aplicação web que possui o intuito de ajudar desenvolvedores a criarem apps melhores, sendo que ela guarda dados no formato de *JSON* (*JavaScript Object Notation*) [14]. Ele possui alguns serviços disponíveis: *Analytics*, *Cloud Messaging*, *Auth*, *Real-time Database*, *Firebase Storage*, *Test Lab for Android*, *Crash Reporting* e *Notifications*. Para esse app em específico, utilizou-se apenas o FirabaseAuth, que é um serviço que oferece a funcionalidade de login com email e senha, além de também oferecer suporte para login com Facebook, Gmail, Github, dentre outros.

Para sua ativação é necessário utilizar uma chave de API específica no seu aplicativo que é acessada pelo arquivo *google-services.json* que lhe é fornecido no momento de criação do projeto. Esse arquivo deve ser colocado na pasta *app* do projeto no *Android Studio*.

5.2. Maps SDK

O Google Maps é um serviço oferecido pela *Google*, que possui uma API que possibilita aos desenvolvedores adicionar informações dos *Maps* em seus aplicativos. om o *Maps SDK* (*Maps Software Development Kit*) oferece a possibilidade de se integrar mapas às aplicações, mostrar a localização em tempo real no mapa do usuário através do GPS do celular, marcação pontos específicos no mapa, traçagem de rotas e a possibilidade de se fazer desenhos no mapa [15].

Para se adicionar o *Maps* ao projeto, é necessário a geração de uma chave de API, que é colocada no *Android Manifest*. Essa API é colocada diretamente no manifesto, e por isso, torna-se necessário a restrição de acessos no console da *Google*, para que mesmo que terceiros consigam a chave, eles ainda não terão acesso pois isso dependerá de informações específicas que servem para restringir o acesso à API.

5.3. ROOM Database

Room é uma biblioteca do banco de dados local do *Android* (SQLite), que guarda tabelas, tal como no SQL, na memória do dispositivo do usuário, sendo acessadas apenas pelo aplicativo em questão.

Para sua utilização, algumas partes são necessárias. Primeiro é necessário ter uma Entidade, que é uma *Data Class*, que possui os nomes e tipos de variáveis, além da designação de qual variável será a chave primária, indicada pela *tag* *@PrimaryKey*, que pode receber um modificador de *autoGenerate* = *true* para assim o id inteiro inicializado em zero possa ter os ids gerados de forma automática.

Outra parte importante é DAO (*Data Access Object*), que é o objeto responsável pelas expressões de interação com o banco de dados, sendo ele uma interface e os métodos declarados como *suspend fun*, e com as tags em cada um indicando o que elas fazem, assim como passando o código em SQL equivalente ao que se deseja que seja feito em cada caso.

E por último, é necessário uma classe abstrata de *Database*, que irá identificar as entidades e a DAO referente a elas, ou seja, é ela que faz o controle do banco de dados.

5.4. Heroku

A plataforma escolhida para disponibilizar o servidor foi o Heroku, a qual é uma plataforma cloud disponibilizada como um serviço (ou na denominação em inglês: *PaaS, Plataform as a Service*), que permite ao usuário disponibilizar suas aplicações online, tendo suporte escritas em diversas linguagens de programação diferentes, além de possuir diversas ferramentas para flexibilizar os processos de manutenção, atualização, *deployment* e dimensionamento do programa.

Todas as aplicações no Heroku são executadas em contêineres virtuais denominados *dynos*, cada *dyno* representando uma instância individual da aplicação online e disponível para receber e responder pedidos [16], isso é feito para possibilitar que a aplicação seja dimensionada mais rapidamente e sem causar impactos negativos ao usuário ou para as outras instâncias da aplicação. A comunicação com os *dynos* é organizada por um sistema de *router*, que recebe e direciona todas as requisições *HTTP* tanto vindas da web quanto por meio de *API* 's para a instância correta da aplicação[17].

5.5. NodeJS e JSON

O NodeJS é um interpretador de *JavaScript* específico para construção de sistemas *server-side*, ou seja, funciona sem ter um navegador web vinculado. O NodeJS inclui funcionalidades estendidas especificamente para essa área disponibilizados por meio de módulos, libraries e plugins, um dos exemplos mais conhecidos de repositórios de libraries open-source é o NPM. Devido a sua arquitetura, os servidores em NodeJS são consideravelmente menos complexos de se escrever, e possuem um *throughput* alto para ações de I/O intensivas com menos gastos computacionais. A arquitetura do NodeJS é baseada em uma única thread executando o código do aplicativo, a qual lida com todas as requisições que chegam ao servidor, e opera em um sistema de fila de eventos, os quais são cada requisição vinda dos usuários, verifica o tipo da requisição, se será necessário algum tipo de acesso a recursos externos, utilizando apenas uma thread para acessos menores, e na necessidade de algum recurso externo, então se utilizam recursos extras caso necessário [18].

O JSON (*JavaScript Object Notation*), é um formato de intercâmbio de dados, leve, independente de linguagem e baseado em texto, definindo identificadores e valores, além de estruturas com arrays, tanto de objetos como de valores simples, padronizados e de estrutura simples [19]. Em uma requisição a um servidor que utiliza esse formato, o objeto JSON é montado no ambiente no qual o usuário realizou a requisição, é serializado e enviado ao servidor, e cabe ao servidor desserializar o arquivo, gerar o

JSON da resposta e repetir o processo com sua resposta ao usuário, e ao aplicativo no lado *client-side* realizar sua desserialização.

5.6. Retrofit

O Retrofit é uma library para a construção de um cliente HTTP para a realização de requisições utilizando esse protocolo. O objetivo dessa library é de definir o formato das requisições HTTP, da estrutura dos arquivos JSON de requisição e de resposta, e dentro das aplicações, facilmente formatar e acessar as informações de cada um, podendo também trabalhar em conjunto com outras ferramentas de serialização e desserialização.[20]

O objetivo do Retrofit no aplicativo é de simplificar o processo de utilização das REST APIs (*Representational State Transfer Application Programming Interfaces*) dentro do sistema do aplicativo, tornando o acesso muito mais dinâmico e seguro. Devido a sua performance e facilidade de uso, o Retrofit ganhou uma popularidade considerável entre desenvolvedores Android.[21]

5.7. DialogFlow

A plataforma escolhida para o desenvolvimento do chatbot foi o Dialogflow, a plataforma da Google de processamento de linguagem natural para a construção de interfaces conversacionais com o usuário, por meio de voz ou texto [22].

A API do Dialogflow permite que o sistema possa receber e responder requisições de várias formas por meio de sua API própria da Google possibilitando que sua integração com diversos serviços e plataformas de comunicação seja muito mais eficaz [23]. Na estrutura do projeto, o servidor hosteado no Heroku contém os dados de acesso ao serviço de Cloud da Google, e faz a comunicação entre o usuário e o Chatbot, direcionando a requisição feita pelo usuário e a resposta gerada pelo dialogflow. aos seus devidos locais.

6. Metodologia

O aplicativo foi totalmente desenvolvido na linguagem de programação *Kotlin* e inicialmente está separado em duas atividades distintas: a *LoginActivity* e a *MainActivity*. As telas em si são todas feitas em Fragmentos e é utilizado o Data Binding para gerenciar os cliques na tela, para melhor eficiência e desenvolvimento do código. Ele também possui um chatbot feito no *DialogFlow*, que é acessado do *HomeFragment* e possui o intuito de ajudar o usuário em caso de dúvidas na utilização do aplicativo. Também é importante enfatizar que o aplicativo não busca por marcas específicas de cada produto, ele busca o produto como um todo, então, a pesquisa na tabela de produtos do mercado não é feita como, por exemplo, Axe, e sim como desodorante, pois o objetivo é levar o usuário ao local onde os desodorantes estão.

A LoginActivity possui três fragmentos apenas: *LoginFragment*, *SignUpFragment* e *ForgotFragment*. Toda a parte de login foi feita utilizando-se a autenticação do Firebase, com as funcionalidades feitas nos próprios fragmentos.

Ao se inicializar o aplicativo, antes dos fragmentos da *LoginActivity*, é mostrado as telas de *Onboarding*, que dão algumas breves informações sobre o aplicativo e possui um botão na parte de baixo da tela que ao clicá-lo se vai direto para a tela de *Login*. E na tela principal do *Onboarding* está implementado uma função *checkUser*() para verificar se o usuário já está conectado, assim, caso ele feche o aplicativo e volte a abrí-lo, a *LoginActivity* inteira é pulada e ele é direcionado direto para a o *HomeFragment*, que está na *MainActivity*.

Após o Onboarding e o LoginActivity, o aplicativo entra na *MainActivity*, onde todas os outros fragmentos são apresentados, sendo esses: *HomeFragment*, *ListFragment*, *NewListFragment*, *MarketFragment*, *MapsFragment*, *ProfileFragment* e *ChatbotFragment*. O *HomeFragment* é apenas a tela inicial, onde se pode ir visualizar a lista de compras ou ir para o chatbot para se tirar dúvidas.

O *NewListFragment* foi desenvolvido antes do *ListFragment*, pois a funcionalidade de ver a lista foi feita após a de salvar a lista. A tabela da lista de compras é salva localmente no celular do usuário, sendo acessada e visualizada apenas por ele, utilizando-se o banco de dados *SQLite* do *Android*, *ROOM*. O plano inicial era ter três tabelas, uma guarda os produtos escolhidos pelo usuário, uma guarda o nome de cada lista que o usuário fez e a terceira tabela relacionaria o produto ao nome de uma lista, assim, o usuário poderia criar várias listas e utilizar uma novamente ou criar uma nova. Porém, devido a problemas de tempo no desenvolvimento, parte disso foi cortado e a funcionalidade ficou da seguinte forma: o usuário ao logar no aplicativo pode visualizar sua lista na *Home* do aplicativo, sendo que caso a lista esteja vazia, ele então pode ir para a tela de criar uma nova lista, assim, ele adiciona os produtos que deseja, sendo esses verificados numa tabela de produtos no mercado, para que não haja problemas na identificação do produto, como por exemplo, se o usuário escrever feijao em vez de feijão, ou trocar alguma letra de lugar. A busca feita é simples, pega-se a palavra escrita pelo usuário e é verificada se a mesma está presente na tabela de produtos, caso não esteja, um *Toast* aparece pedindo para ele verificar a escrita. Caso esteja tudo certo na escrita, o produto é adicionado numa estrutura lista do *Kotlin*, onde caso o usuário deseje remover algum item, ele então pode simplesmente arrastar o item para o lado para removê-lo. Ao terminar de se adicionar os itens, o usuário clica no botão no final da tela para salvar e cada linha da lista é adicionada no banco de dados local, gerando assim uma tabela no celular que somente esse aplicativo pode acessar. Ao salvar uma nova lista, a anterior tem todos os seus itens removidos do banco de dados e a nova é salva por cima - pelo problema do tempo, optou-se por se ter esse funcionamento simplificado. Ao salvar a lista, o usuário retorna à tela de visualização da lista de compras e assim pode ir para a compra.

Nesse ponto, ele é direcionado para uma tela onde o plano original era a listagem de todos os mercados próximos do usuário, porém, devido ao *SDK Maps* ter uma grande complicação com o retorno dessas informações, foi sugerido pelo professor apenas mostrar como seria no produto final e devido ao tempo optou-se por essa alternativa.

Nessa tela é mostrado na parte de cima um mapa com alguns pontos de mercado e na metade de baixo é adicionado um *RecyclerView* com os nomes desses mercados marcados por pinos. Utilizou-se o Pão de Açúcar de Águas Claras como modelo para demonstrar o funcionamento do aplicativo. Ao se clicar no item do Pão de Açúcar no RecyclerView se é direcionado para a tela *MapsFragment*.

A tela atual, portanto, é onde ocorre o propósito principal do aplicativo, que é o de se mostrar as localizações dos produtos contidos na lista. Quando se entra nessa tela, a primeira parte a ser feita é resgatar a lista do banco de dados, colocando-a em uma estrutura de lista. Após isso então se é verificado item por item em qual corredor do mercado que ele está localizado. Para isso, utilizou-se uma tabela de produtos do mercado, que possui duas variáveis: nome e corredor. Os corredores no código são listados a partir de zero e por haver vários produtos diferentes do mesmo corredor, a variável corredor se repete. Então, pega-se o primeiro item da lista do usuário e é procurado na lista de produtos do mercado por ele, e quando se ele é achado, adiciona-se então o nome do produto e o corredor dele em uma nova lista. Essa lista, então, é ordenada usando o método do *Kotlin .sortBy()* - uma vez que se é uma ordenação simples, podendo conter itens com mesmo valor inteiro e que não terão muitos itens - e ordenando pelo valor inteiro de cada corredor. *Kotlin* implementa os mesmos algoritmos de busca que o Java, portanto, para um simples *array* de inteiros, ele usa o *Quicksort* que é bem mais eficiente, porém, ao se ter uma *array* de objetos, torna-se mais eficiente usar outros tipos de ordenação, e nesses casos o *Kotlin*, e o *Java*, usam o *TimSort*, que é uma junção entre *MergeSort* e *InsertionSort*, também chamado de *MergeSort* iterativo, e para trabalhar com objetos ele tende a ser mais rápido. O *TimSort* se apresenta mais rápido em casos de arrays semi organizados, o que uma lista de compras costuma ser, portanto, a utilização do método de ordenação padrão do *Kotlin* funciona em cima do esperado para a situação do aplicativo.

Logo, a ideia básica para se acelerar as compras é: vá direto onde o que você deseja está. Para isso, decidiu-se o seguinte método: partindo da entrada do mercado, caso o usuário queira um produto que está no primeiro corredor e um que está no quinto, ele vai passar no primeiro e ir direto para o quinto, assim, ao se pegar todos os produtos que ele quer e ordená-los baseados em quais corredores estão, o cliente irá passar apenas por esses corredores. Por exemplo, se ele quer cinco produtos, dois estão no corredor um, dois no três e um no sétimo, ele vai passar primeiro no corredor um, pegar tudo que precisa nele, ir direto pro três, pegar os produtos, então ir direto ao sétimo e então para o caixa para pagar as compras. Assim, no topo da tela, então, é mostrado qual o primeiro produto da lista já ordenada e no mapa abaixo é colocado um pino no local do corredor, sendo que as gôndolas foram desenhadas com uma propriedade do *Google Maps* de se desenhar polígonos no mapa.

O mapa, em específico, possui as funções de *zoom* e rotação totalmente desligadas e a câmera é movida para o local dele no mapa, para assim mostrar apenas o mercado e o usuário não sair dessa visão por engano. Nela são colocadas as posições das gôndolas numa tabela e são guardadas as latitude e longitude de cada corredor.

Ao ir nesse corredor e pegar o produto, o usuário clica em um botão abaixo do nome do produto que remove da lista aquele produto e pega o produto que agora está no

topo da lista. Uma vez que a lista está organizada por corredores, caso o próximo produto seja no mesmo corredor, o pino no mapa não muda. Caso o produto esteja em outro corredor, o pino anterior é removido e um novo pino é adicionado no corredor desse novo produto. Esse processo é repetido até que a lista esteja vazia. Se ela estiver vazia, o usuário pega todos os produtos que desejava e ele é redirecionado para a *Home* do aplicativo.

As tabelas de produtos do mercado e posição de seus corredores originalmente seria armazenado em um banco de dados remoto, para ser baixada quando o usuário escolhe o mercado, porém, novamente devido a problemas com tempo e outros membros do grupo, optou-se por simular como seria após baixar a tabela, pois, depois de baixá-la, ela seria colocada em uma estrutura de lista para poder ser manuseada junto com as outras listas e tabelas. Outro problema com o qual nos deparamos foi o de não termos um sistema de localização preciso o suficiente. Foram feitos testes para capturar a latitude e longitude do ponto central de cada corredor do mercado escolhido (Pão de Açúcar de Águas Claras) porém, a localização recuperada pelo celular não era precisa o suficiente, e mesmo variando nossa posição 10 metros, a localização não variava, portanto, mostrou-se necessário fazer uma visualização do local e então fazer um mapeamento aproximado das posições.

Antes da construção das demais partes do fragmento de interação com o Chatbot, foram desenvolvidos o sistema de backend, que está hospedado na plataforma *Heroku*, e o próprio *Chatbot*, o qual foi criado utilizando o sistema *Dialogflow*, da *Google*. O servidor foi desenvolvido utilizando *NodeJS*, e a comunicação é realizada por *HTTP*, com as requisições escritas no formato *JSON* contendo a mensagem do usuário e uma ID específica da sessão, e as respostas contendo o texto de resposta do chatbot no *Dialogflow*.

Dentro do aplicativo, a interação com o chatbot é feita pelo *ChatbotFragment.kt*, podendo enviar perguntas em texto para o servidor, e visualizar tanto as perguntas realizadas quanto às respostas recebidas, assim que o fragmento é ativado, o aplicativo gera uma ID única para a sessão utilizando o método do kotlin UUID.randomUUID() e convertendo o resultado para uma *string* e dispara uma requisição padrão para o servidor, inicializando a conversa entre usuário e chatbot. A requisição em *JSON* é construída pelo *Retrofit*, em conjunto com as data classes *ChatbotRequest* e *ChatbotResponse*, que definem a estrutura dos campos do *JSON* das mensagens de requisição e de resposta do servidor, respectivamente e da interface *DialogFlowService*, que contém a estrutura da função de requisição *HTTP (Hypertext Transfer Protocol)* que será chamada pelo Retrofit, para enviar a mensagem do usuário e realizar a requisição, o *ChatbotFragment.kt* pega o texto escrito pelo usuário, apresenta o texto da pergunta no histórico da conversa, e realiza a requisição utilizando o texto e a ID gerada na ativação do fragmento, a resposta é apresentada no histórico da conversa assim que recebida, caso o campo de texto da mensagem do usuário esteja vazia no momento da tentativa de envio, um *Toast* aparece indicando que é necessário que seja escrito um texto. Todo o histórico da conversa é apresentado no *layout* do *fragment* por meio de um *RecyclerView* específico do chatbot.

7. Resultados Obtidos

A primeira parte do aplicativo feita foram os visuais das telas de *Login*, Cadastro e Esqueci Minha Senha, que podem ser visualizadas na imagem abaixo. O aplicativo foi todo feito apenas em duas atividades: uma *MainActivity* e uma *LoginActivity*, para que assim, ao entrar pela primeira vez no aplicativo você primeiro passe pela *LoginActivity*, que tem os fragmentos do *Onboarding*, *Login*, Cadastro e Esqueci Minha Senha. Cada uma dessas telas foi desenvolvida como um fragmento e utilizando o *Data Binding*.

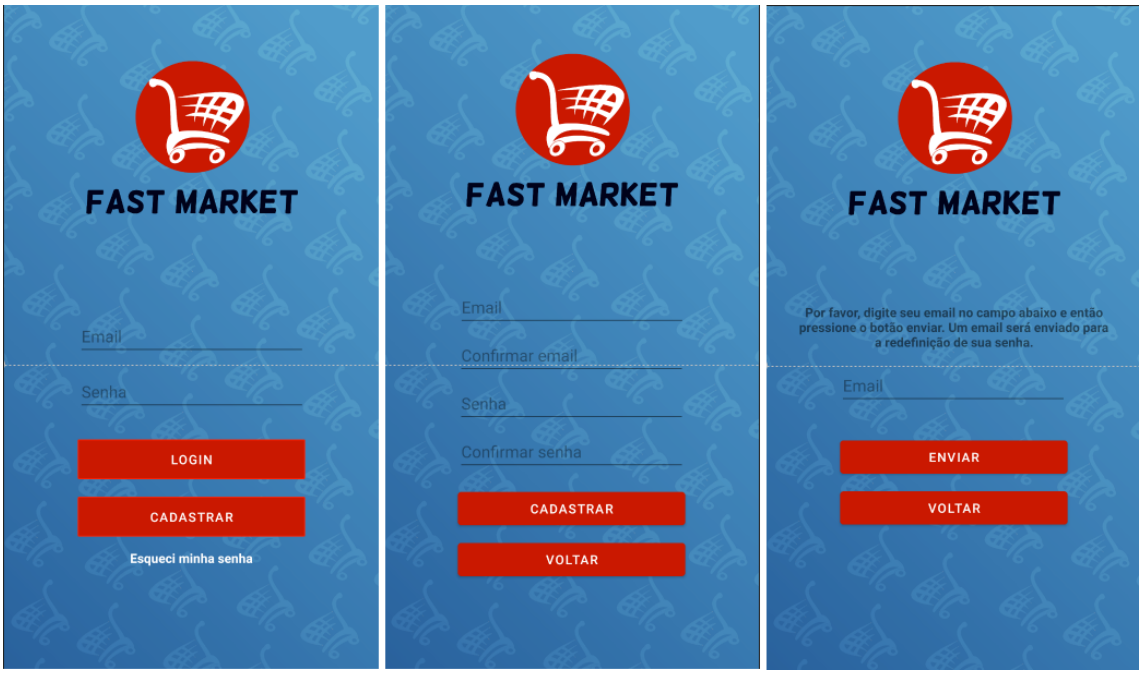

Figure 1. Telas de Login, Cadastro e Esqueci Minha Senha.

Uma imagem para o fundo foi feita para o aplicativo em três cores, uma azul no tom de azul principal utilizado por toda a identidade visual do aplicativo, porém, foram feitas também em outras duas cores: cinza, para a tela do *chatbot*, e vermelho para a tela de perfil.

Após isso, foi desenvolvido o *backend* do *login* ao mesmo tempo que o *Onboarding*. Para o *login*, utilizou-se o *Firebase Auth*, apenas com a funcionalidade de email e senha. O plano original seria de se usar uma *ViewModel* em conjunto com a injeção de dependência utilizando-se a biblioteca *Hilt*, porém, devido a problemas com tempo e a divisão de trabalho entre os membros, após algumas tentativas de aplicar a injeção de dependência, optou-se por fazer o login pelo método antigo que consiste em fazer as funções de cada fragmento dentro dele próprio. Logo, a função de cadastro está feita dentro do fragmento de Cadastro e assim para todas as telas de *login*. Também foi inserido no fragmento de *Onboarding*, que é o primeiro fragmento apresentado ao se abrir o aplicativo, uma função para checar se o usuário já está conectado no aplicativo, para que assim, caso esteja, ele seja redirecionado para a tela *Home*.

O sistema de *onboarding* funcionou dentro das expectativas esperadas, introduzindo de forma breve o objetivo principal do aplicativo ao usuário, o sistema utiliza quatro fragmentos, um principal (*OnboardingFragment*) onde foram apresentadas as páginas do *onboarding*, e três outros fragmentos (*OnboardingPage1, OnboardingPage2, OnboardingFragmentPage3*) que representam cada página do *onboarding*, e um adaptador (*OnboardingAdapter*), o qual coordena a sequência das páginas a serem apresentadas. Não foram encontrados problemas com a implementação do sistema no aplicativo, e a estrutura permite que possam ser adicionadas mais páginas a serem apresentadas futuramente, caso necessário.

Figure 2. Fragmentos de Onboarding.

Nesse ponto, começou-se o desenvolvimento da criação das listas de compras. Para isso, seria necessário a utilização de um banco de dados local, uma vez que a lista do usuário seria salva apenas no dispositivo dele. O plano original se ter uma lista de listas feitas pelo usuário e uma listas de produtos que o usuário já adicionou, e então uma terceira lista que teria a chave primária da lista de listas e a chave primária dos produtos, assim, pelo SQLite seria possível, quando o usuário escolher uma lista para ser aberta, recuperar quais produtos estavam nela usando a lista de chaves. Porém, devido a problemas com tempo, decidiu-se simplificar o sistema. O usuário dificilmente irá sempre comprar os mesmos produtos, portanto, o modelo optado foi o de se utilizar apenas uma lista. Ele pode visualizar essa lista e escolher fazer as compras, ou criar uma nova lista. Caso opte por criar uma nova lista, a antiga é apagada do banco e o usuário é direcionado para a área de criação de uma lista.

O usuário, para adicionar itens na nova lista, escreve o nome do produto num bloco de EditText no topo. Ao se clicar para adicionar o item na lista, é feita uma busca em uma lista de produtos disponíveis no mercado, então, caso seja encontrado o produto na lista do mercado, o produto é adicionado, e se caso não seja encontrado, o usuário é informado que o produto não foi encontrado e pede para ele verificar o nome do produto.

Primeiramente, o aplicativo foca em fazer a compra ser mais rápida mostrando onde os produtos ficam localizados, ignorando assim marcas e preços. Por isso, na tabela de produtos do mercado há apenas produtos como: arroz, feijão, macarrão, azeitona; e nenhum deles indicado pela marca ou preço. Em segundo lugar, o plano inicialmente era ter algum servidor de onde receber essa tabela, mas devido a problemas com tempo e desenvolvimento, precisou-se simular como seria a tabela após ser recebida do servidor. O funcionamento do algoritmo foi feito como se a tabela tivesse sido recebida do servidor, porém, ela é inicializada junto com a criação da View da tela.

Ao terminar de criar a lista, o usuário é redirecionado para a tela que mostra a lista, sendo possível ele recriar a lista ou ir para o mercado. Um problema encontrado nessa etapa foi que para recuperar a lista de produtos do banco de dados local usando o Data Binding implicou que o CardView (funcionalidade do RecyclerView para criar as células da lista) não funciona com o Binding, assim, embora tenha sido usado na criação da lista, que não precisou ser usado o Binding, para recuperar a lista do banco de dados após ela ser guardada acabou sendo necessário usar um visual diferente para a as listas.

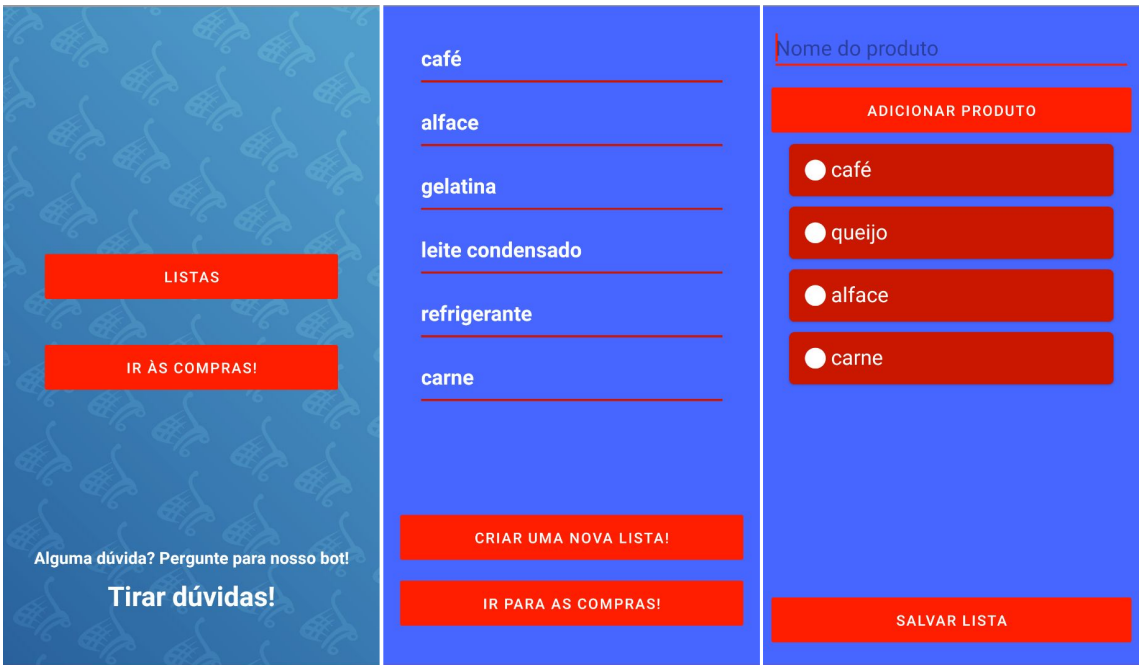

Figure 3. Telas de Home, Lista e Nova Lista.

Figure 5. Banco de dados local, primeiro a entidade, depois o DAO e então o Database.

Nesse ponto, o usuário, ao escolher ir para o mercado é levado a uma tela onde teriam as opções de mercado disponíveis, mas, como sugerido nas aulas de dúvidas, devido a dificuldade para se recuperar as latitudes e longitudes de locais próximos ao usuário, essa tela foi simulada de como seria no aplicativo fora da etapa de desenvolvimento. Assim, tentou-se deixar a tela parecida com o modo que se era esperado, assim, na parte superior é exibido um mapa com pinos para os mercados próximos, e embaixo um botão que simula a escolha do mercado pelo usuário.

Na tela atual, é mostrado o mapa do Google Maps com um zoom em cima do mercado, onde estão desenhadas as gôndolas dos produtos. No topo, é mostrado o produto a se pegar e um botão para pegar o próximo produto. Ao entrar nessa tela, é recuperada a lista do usuário é feita uma busca de cada produto na lista de produtos disponíveis no mercado, e nessa lista estão os corredores do mercado, indicando em qual corredor está cada produto. Assim, cria-se uma nova lista com o produto do usuário e a posição dele no mercado e então é feita uma organização levando-se em conta apenas os corredores, portanto, o resultado será os produtos dispostos na lista do primeiro para o último. Assim, é mostrado um marcador no corredor que o cliente deve ir, e ao se clicar para pegar o próximo produto, caso ele esteja em outro corredor, o marcador é removido e um novo é adicionado no corredor em questão.

O problema encontrado nessa etapa é o mesmo da nova lista. Os produtos do mercado foram simulados aqui novamente, mas além deles, as latitudes e longitudes de cada corredor, e os desenhos das gôndolas (feitos usando LatLng no Maps) também foi simulado pelo mesmo motivo.

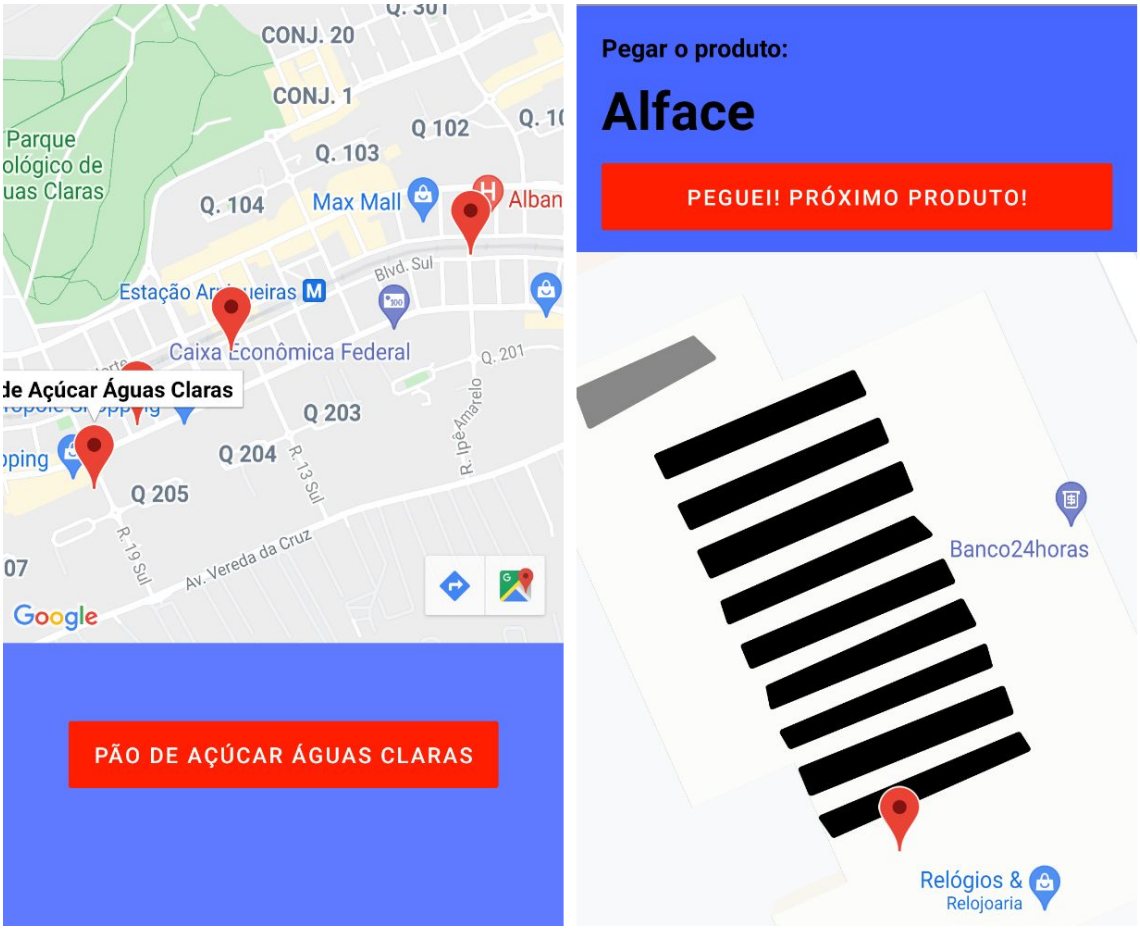

Figure 5. Telas de Mercados e Compras.

Os resultados do chatbot e do servidor foram dentro do esperado, porém inicialmente, o sistema apresentou problemas para a realização da comunicação entre o Usuário, o servidor e o *Dialogflow*, as requisições seguiam bem sucedidas, mas o arquivo *JSON* de resposta enviado pelo servidor ao aplicativo era muito mais extenso e estruturalmente complexo do que inicialmente esperado, o que gerou uma série de problemas tanto para o acesso às informações necessárias ao funcionamento da interface de conversa com o chatbot quanto para moldagem das requisições do usuário. Devido a isso, foram feitas alterações no servidor para que fosse gerado um arquivo de resposta do servidor com apenas os dados necessários ao aplicativo, feito isso, não foram encontrados novos problemas com o funcionamento da interface de conversa, das respostas do Dialogflow ou do servidor hosteado no Heroku nos testes subsequentes.

Figure 6. Tela do Chatbot.

Vale lembrar que com o aplicativo rodando, é visualizado também um *Bottom Navigation View*, que possui botões para Home, Lista e Perfil, sendo que o acesso ao Chatbot é feito por um link na *Home*, a Nova Lista só é acessível pela tela de Lista e o mesmo é válido para a tela de Mercados e Compras.

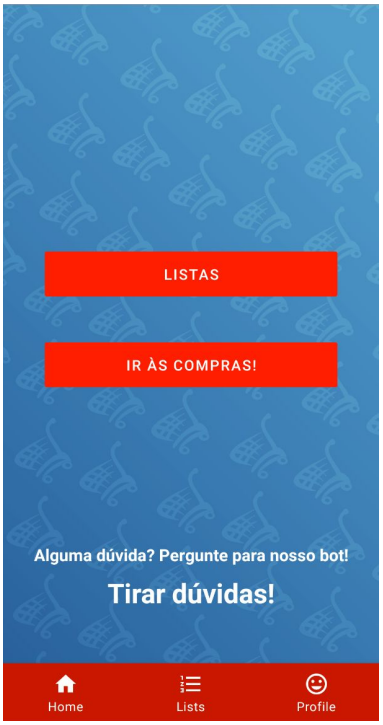

Figure 7. Tela de Home com o Bottom Navigation View.

8. Conclusão

Observa-se que o aplicativo funcionou como esperado, porém ele possui problemas devido ao fato de terem sido feitas simulações de como seriam. Apesar das dificuldades encontradas com o grupo e o tempo disponível, optou-se por terminar o aplicativo, mas mesmo terminando-o observa-se esses problemas. Porém, ele está funcionando, e caso tivesse o banco de dados externo implementado, o funcionamento teria sido o mesmo, portanto, considera-se um sucesso.

Para trabalhos futuros seria a implementação do restante do banco de dados local, para a criação de várias listas em vez de sempre se trabalhar apenas com uma. Outra funcionalidade seria a implementação do já citado banco de dados remoto, sendo que com esse poderia ser feita também um armazenamento das listas de compras do usuário, não guardando nenhuma outra informação pessoal, apenas o que foi comprado, para assim poder fazer mineração de dados em cima dos produtos mais comprados e adicionados às listas dos usuários, sendo assim algo que os mercados poderiam se beneficiar ao planejar as promoções, pois o cliente pode acabar comprando mais do que planejou, porém, com as listas, se tem o plano original do usuário, assim sabendo o que ele acha mais importante, logo, sendo uma informação mais interessante para o mercado.

References

[1] A Gazeta. “Covid-19: os riscos de contaminação com compras de supermercado”. <https://www.agazeta.com.br/es/cotidiano/covid-19-os-riscos-de-contaminacao-com-compras-de-supermercado-1020>. Acesso em: 21 de março de 2021.

[2] Diário Oficial do Distrito Federal. ANO L EDIÇÃO EXTRA Nº 22-A. BRASÍLIA - DF, SEXTA-FEIRA, 19 DE MARÇO DE 2021. <https://dodf.df.gov.br/index/visualizar-arquivo/?pasta=2021%7C03_Março%7CDODF%20022%2019-03-2021%20EDICAO%20EXTRA%20A%7C&arquivo=DODF%20022%2019-03-2021%20EDICAO%20EXTRA%20A.pdf>. Acesso em: 21 de março de 2021.

[3] Helio Print. “10 truques para organizar prateleiras de supermercado e vender mais”. <https://helioprint.com.br/blog/organizar-prateleiras-supermercado/>. Acesso em: 21 de março de 2021.

[4] Agência Brasil. “DF prorroga fechamento do comércio e restrições de circulação”. <https://agenciabrasil.ebc.com.br/saude/noticia/2021-03/df-prorroga-fechamento-do-comercio-e-restricoes-de-circulacao>. Acesso em: 21 de março de 2021.

[5] Folha de São Paulo. “Brasileiro reduz pela metade tempo gasto em compra no supermercado”. <https://www1.folha.uol.com.br/fsp/mercado/me0610201018.htm#:~:text=O%20consumidor%20brasileiro%20prefere%20ir,em%201998%20superava%20uma%20hora>. Acesso em: 22 de março de 2021.

[6] SEBRAE. “Estudo mostra novo comportamento do consumidor diante da pandemia”. <https://www.sebrae.com.br/sites/PortalSebrae/artigos/estudo-mostra-novo-comportamento-do-consumidor-diante-da-pandemia,9388ad41eab21710VgnVCM1000004c00210aRCRD>. Acesso em: 22 de março de 2021.

[7] Estadão. “Perguntas e Respostas: Veja como evitar contaminação de coronavírus na ida ao mercado”. <https://saude.estadao.com.br/noticias/geral,perguntas-e-respostas-veja-como-evitar-contaminacao-de-coronavirus-na-ida-ao-mercado,70003245391>. Acesso em: 20 de março de 2021.

[8] E-Commerce Brasil. “46% dos brasileiros fizeram mais compras online na pandemia, indica Mastercard. <https://www.ecommercebrasil.com.br/noticias/brasileiros-compras-online-pandemia-coronavirus/>. Acesso em: 21 de março de 2021.

[9] CNN Business. “Online shopping has been turbocharged by the pandemic. There's no going back”. <https://edition.cnn.com/2020/10/11/investing/stocks-week-ahead/index.html>. Acesso em: 21 de março de 2021.

[10] CNBC. “More people are doing their holiday shopping online and this trend is here to stay”.

<https://www.cnbc.com/2020/12/15/coronavirus-pandemic-has-pushed-shoppers-to-e-commerce-sites.html> . Acesso em: 21 de março de 2021.

[11] UN News. “Pandemic has forever changed online shopping, UN-backed survey reveals” <https://news.un.org/en/story/2020/10/1074982> Acesso em: 21 de março de 2021.

[12] ORELLANA, Jesem Douglas Yamall; CUNHA, Geraldo Marcelo da; MARRERO, Lihsieh; MOREIRA, Ronaldo Ismeiro; LEITE, Iuri da Costa; HORTA, Bernardo Lessa. “Excesso de mortes durante a pandemia de COVID-19: subnotificação e desigualdades regionais no Brasil”. Scielo Saúde Pública. DOI: https://doi.org/10.1590/0102-311X00259120. Disponível em: . Acessado em: 10 de junho de 2021.

[13] SANCHEZ, Mauro Niskier et al. Mortalidade por COVID-19 no Brasil: uma análise do Registro Civil de óbitos de janeiro de 2020 a fevereiro de 2021. SciELO Preprints, 2021. DOI: https://doi.org/10.1590/SciELOPreprints.2012. Disponível em: https://preprints.scielo.org/index.php/scielo/preprint/view/2012. Acesso em: 10 de junho de 2021.

[14] KHAWAS, Chunnu; SHAH, Pritam. Application of firebase in android app development-a study. **International Journal of Computer Applications**, v. 179, n. 46, p. 49-53, 2018.

[15] MEDEIROS JUNIOR, Gilmar Margoti De. Implementação de um protótipo de aplicativo móvel com a Integração com a ferramenta Google Maps e o componente GPS dos smartphones voltado ao auxílio no transporte público da cidade de Criciúma. Disponível em: <http://repositorio.unesc.net/handle/1/8156>. Acessado em: 10 de junho de 2021.

[16] HANJURA, Anubhav. **Heroku Cloud Application Development**. Packt Publishing Ltd, 2014.

[17] MIDDLETON, Neil; SCHNEEMAN, Richard. **Heroku: up and running: effortless application deployment and scaling**. " O'Reilly Media, Inc.", 2013.

[18] MEAD, Andrew. **Learning Node. js Development: Learn the fundamentals of Node. js, and deploy and test Node. js applications on the web**. Packt Publishing Ltd, 2018.

[19] BRAY, Tim. **The javascript object notation (json) data interchange format (No. RFC 8259)**. Technical Report, 2017.

[20] SOURCE, Square Open. Retrofit. **URL: http://square. github. io/retrofit**.

[21] BELKHIR, Abdelkarim et al. An observational study on the state of REST API uses in android mobile applications. In: **2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)**. IEEE, 2019. p. 66-75.

[22] SOURCE, Google. Dialogflow ES Documentation. **URL: https://cloud.google.com/dialogflow/es/docs**

[23] SOURCE, Google. Cloud Dialogflow ES Documentation. **URL: https://cloud.google.com/dialogflow/es/docs/reference/api-overview**