RAW App

Pedro H. Pino, Samuel S. Rocha, Filipe C. de Jesus

Instituto de Ensino Superior de Brasília

Ciência da Computação

pedro.haluch@gmail.com, samuelsilv.rocha@gmail.com, filipe.cdj@gmail.com

Resumo. *Neste trabalho os conhecimentos adquiridos ao longo do semestre foram usados para conceber o* MVP *de um aplicativo Android com temática de review de produtos. Dentre as ferramentas empregadas estão construção de APIs, CRUD em banco de dados, framework MVVMi, e requisições HTTPs, além da elaboração de um roadmap geral de próximos passos.*

1. Introdução e Contextualização

A temática inicial do projeto foi definida como uma plataforma para avaliação de artigos sexuais, trazendo um ambiente mais livre para a discussão sobre esse tipo de produto que é considerado tabu. Durante o desenvolvimento, no entanto, chegou-se à conclusão que o tema era muito expansivo, e que focar na lógica de negócio e valor comercial afetaria o foco e a capacidade de utilizar o ferramental adquirido ao longo do curso. Decidimos então usar a lógica de *MVP*s (*mininum viable products*): fazer a implementação de cada feature e suas possíveis integrações, mesmo que de forma simples.

2. Motivações

Para oferecer um serviço como esse, alguns requisitos são necessários. Do ponto de vista do usuário temos, portanto:

- Necessidade de autenticação,
- cadastro e leitura de informações,
- acesso a localizações, e
- suporte via chatbot.

Já do ponto de vista técnico, precisamos:

- manter ambientes autenticado e não-autenticado distintos,
- delegar a parte de autenticação a um serviço dedicado,
- capacidade de fazer requisições web,
- lidas com essas últimas em camadas que possam ser responsáveis por partes distintas das nossas validações,
- conectar a um banco de dados e fazer as operações básicas,
- acessar serviços de localização geográfica.

A expectativa é, portanto, demonstrar como essas ferramentas trabalham em conjunto para oferecer seus respectivos serviços e facilitar o desenvolvimento desse tipo de produto.

3. Referencial Teórico

MVVMi, Fragmentos, Injeção de Dependências e ViewModels e RecyclerViews

A escolha de como estruturar o aplicativo de forma a facilitar escala e integração com outras ferramentas passa por decidir como os componentes e suas funções serão distribuídos e abstraídos. Um pilar claro desde o início é a extrema dependência de programação orientada a objetos para conseguir executar esse tipo de desenvolvimento. Sem as conveniências de abstração oferecidas por objetos dentro do código seria muito difícil implementar a maioria das ferramentas utilizadas.

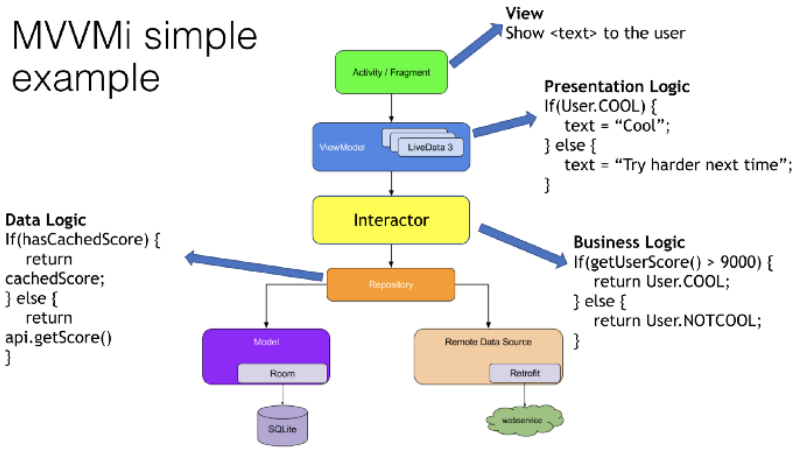


Fig 1. Arquitetura MVVMi (Simpson 2017)

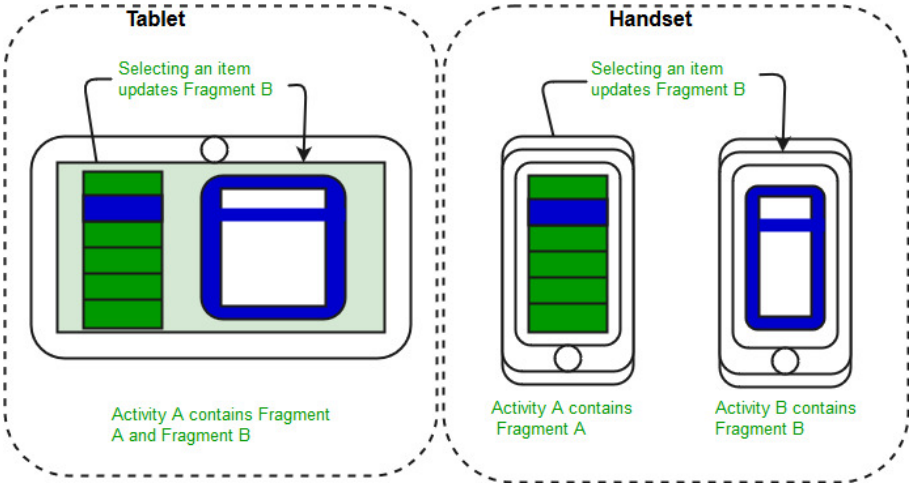


Fig 2. Benefícios do Uso de Fragmentos ("Introduction to Fragments" 2020)

O grande diferencial estrutural das escolhas feitas é a modularização e abstração das estruturas em código. O uso da dependência hilt-dagger e suas injeções de dependência permite criar cadeias de classes que não precisam de construtores sendo instanciados a todo momento. O benefício é imediatamente notado quando se usa a arquitetura MVVMi (Figura 1). Aqui o objetivo é criar camadas de responsabilidades em relação aos dados e objetos sendo transitados. Tanto o desenvolvimento quanto a manutenção ficam mais diretos definindo classes específicas para tratar por exemplo de validação de negócio e validação técnica. O código fica mais modularizado, e os métodos sofrem menos efeito tanto "do que vem antes" quanto "do que vem depois".

Ainda em busca de maior modularização ("Introduction to Fragments" 2020) e abstração fica motivado o uso de fragmentos (*Fragments*) em vez de Activities para criar tanto as classes internas quanto as próprias telas. Os fragmentos são versáteis: podem desde ocupar completamente a tela no lugar de uma Activity como ser apenas um elemento dela. Além disso, são mais plásticos em sua habilidade de comunicar e transitar entre si usando navegação entre fragmentos com a AndroidX Navigation e seus

grafos de navegação. Eles, também, abstraem as “tecnicalidades” de integrar esses objetos e seus atributos.

Para gerenciar atributos e estados visuais faz-se o uso de ViewModels. Essa classe trata esses elementos de forma mais focada no ciclo de vida da aplicação (Google 2020) e de seus elementos, e não apenas baseada em eventos estáticos e gatilhos por interações de usuário. Isso permite ao desenvolvedor trabalhar melhor com as etapas do ciclo de vida de uma tela, fragmento, ou qualquer outro elemento, não sofrendo por exemplo com o recomeço de algumas etapas de um elemento ao mudar a orientação da tela do dispositivo.

Na estruturação uma escolha importante a ser feita é a do uso de RecyclerViews. Esses elementos são responsáveis por dispor na tela estruturas list-like, em diversos formatos distintos (Android Docs 2020). O benefício do uso desse recurso específico está na habilidade de, trabalhando em conjunto com um Adapter (classe responsável por gerenciar os dados dos elementos da lista, entre outras coisas), gerenciar muito melhor a carga desse tipo de estrutura no aplicativo, instanciando e removendo itens de memória à medida que entram e saem da tela.

Data Binding, Recursos de Cores e Temas, Constraint Layouts, Requisições

Uma vez definidas e organizadas as estruturas que recebem o projeto é hora de utilizar ferramentas para dinamizar seu funcionamento, tanto externo quanto interno.

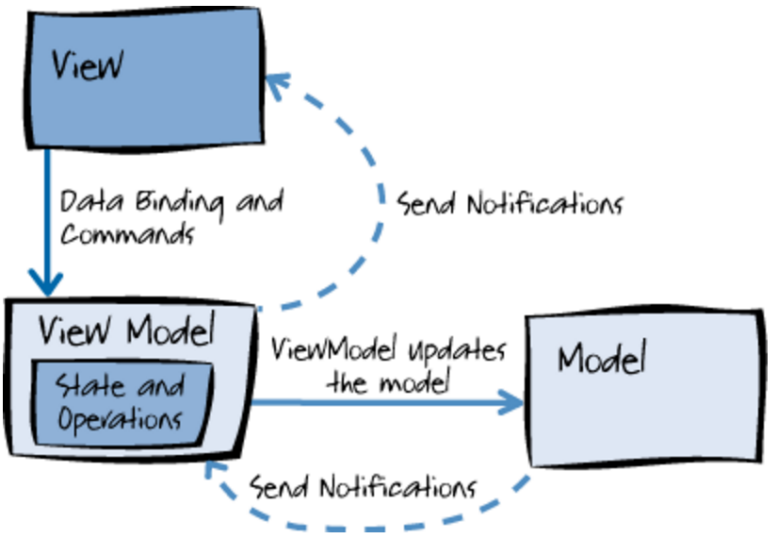


Fig 3. Relação entre ViewModel e Data Binding: trabalho em conjunto para melhor a experiência de desenvolvimento e execução

Do ponto de vista técnico o uso de Data Binding (que traz ,mais uma vez, uma camada de abstração entre os arquivos .xml de layout dos fragmentos e as próprias classes .kt) é essencial. A habilidade de usar ViewModels e *manter* essa abstração mesmo lidando com trânsito de dados entre a parte visual e a parte interna da aplicação dão muito mais flexibilidade. (Google Developers 2020)

Sobre a parte visual duas features que somam à experiência de desenvolvimento são os temas e os ConstraintLayouts. Os primeiros permitem de forma simples manter um padrão visual de cores, fontes, tamanhos, e vários outros aspectos visuais para dar uma visão de unicidade ao aplicativo. Já os ConstraintLayouts somam no dinamismo. Como são elementos visuais construídos de forma relativa aos outros elementos da tela (e também permitem ancoragens “fixas”, se necessário) o desenvolvimento para diferentes dispositivos fica muito mais simples e remove a necessidade de buscar preparar a mesma tela para diversas possibilidades de dimensões.

Serviços: Banco de Dados, Requisições, APIs, Autenticação, GoogleMaps e DialogFlow

Não é realista esperar que um único dispositivo, principalmente dispositivos móveis que por natureza têm limitações em processamento e duração de bateria, sejam capazes de entregar nativamente todas as funcionalidades que um usuário moderno espera.

Nesse caso buscamos ‘fora’ da aplicação as soluções que precisamos. Essas conexões não são imediatas, no entanto. Algumas etapas são necessárias para acessar essas funcionalidades, como:

- preparar o ambiente para receber essas conexões,
- modelar os serviços às necessidades da aplicação, e
- estabelecer os canais de comunicação.

Esses requisitos, apesar de não serem os únicos, são necessários para a implementação de qualquer serviço *online*.

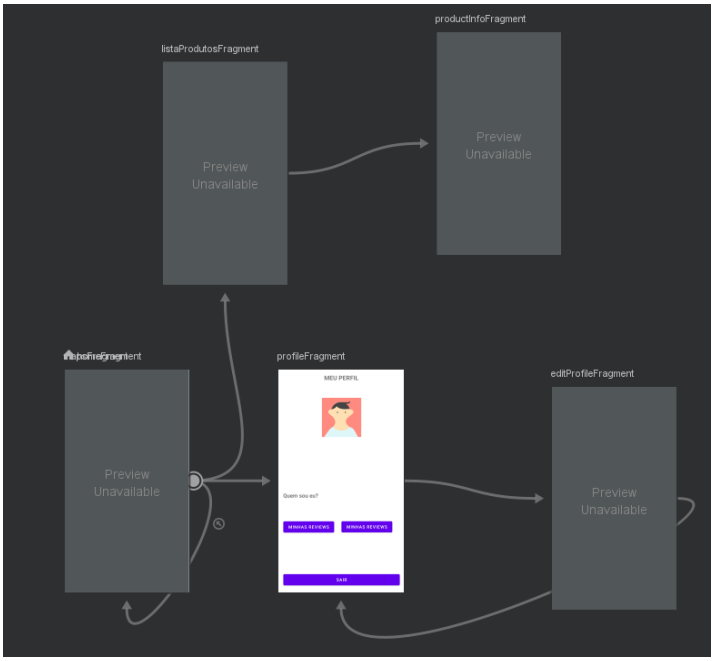
As requisições web foram feitas usando a dependência Retrofit. Ela, junto ao uso de Data Classes, mais uma vez fornece ao desenvolvedor uma camada de abstração na hora de usar as funcionalidades. Usando interfaces para mapear os elementos comuns entre métodos podemos criar objetos que fazem diversos tipos de requisições removendo das mãos do desenvolvedor a criação de código repetido.

As próprias DataClasses são uma ótima abstração para bancos de dados. Sua estrutura é diretamente mapeável ao banco de dados local do sistema Android. É difícil até justificar a necessidade do uso de bancos de dados nas nossas aplicações: são um serviço esperado em qualquer tipo de sistema atualmente, desde embarcados até em nuvem.

O uso de APIs também é fundamental por ser uma forma de se comunicar com esses serviços. FirebaseAuth, Google Maps e Dialog Flow se comunicam via APIs. Este último inclusive requer um serviço intermediário para tratar as requisições. Isso não é uma limitação da linguagem Kotlin ou do sistema Android: é uma decisão com finalidade de balanceamento de carga comum em desenvolvimento mobile. Remover o máximo possível de esforço computacional das mãos do aplicativo aumenta sua usabilidade e experiência do usuário.

4. Execução

O projeto começou do zero, e todas as estruturas foram criadas ao longo do desenvolvimento. Os pacotes foram estruturados de forma a respeitar a estrutura MVVMi e as necessidades de Adapter para as RecycleViews e Fragments como elementos visuais principais.




```
3   import ...
18
19  @HiltViewModel
20  public class RawViewModel @Inject constructor(
21      app: Application,
22      private val interactor: RawAppInteractor) : AndroidViewModel(app){
```

```
@HiltAndroidApp
class RawApp : Application() {
```

Figs. 4, 5 e 6 - Grafo de navegação, Classe ViewModel e ClassApp, geral. As anotações indicam ao Hilt-Dagger que é nesse escopo que funcionarão as injeções de dependência.

As estruturas básicas foram criadas em seguida, para receber as próximas iterações. O grafo de navegação foi criado, assim como a classe ViewModel para o aplicativo e as injeções de dependência para começar de forma padronizada a criar as classes (Figs). Duas Activities foram criadas, para representar informalmente os ambientes autenticado e não autenticado.

```
val taskDeLogin = autenticacao.signInWithEmailAndPassword(email, password)

taskDeLogin.addOnCompleteListener { resultado ->
    if(resultado.isSuccessful) {

        val sharedPref = requireActivity().getSharedPreferences( name: "GlobalVar", Context.MODE_PRIVATE)
        val editor = sharedPref.edit()

        viewmodel.loadUserByEmail(email, editor)
        Log.d( tag: "VIEWMODEL",  msg: "resultadoLoadUserByEmail.value=${viewmodel.resultadoLoadUserByEmail.value}")
        val objectId = viewmodel.resultadoLoadUserByEmail.value?.objectId

        Log.d( tag: "VIEWMODEL",  msg: "btLoginListener > Saved ${email}")
        val intencaoDeChamada = Intent(activity, AppActivity::class.java)
        activity?.startActivity(intencaoDeChamada)
```

Fig. 7 - Trecho do código acessando o Firebase

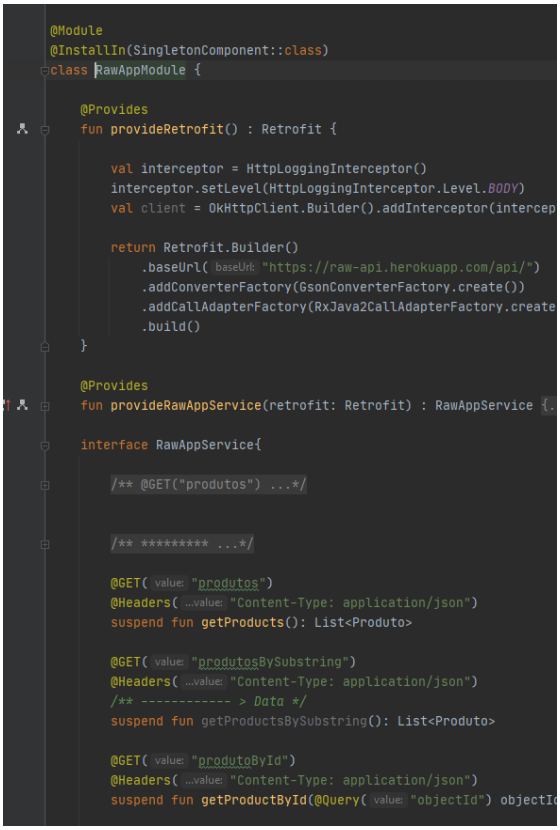
A partir deste momento o desenvolvimento passou a ser feito usando fragmentos. O uso do FirebaseAuthenticator foi implementado paralelamente à criação das classes responsáveis pela estrutura MVVM. O objetivo será utilizar essas classes para fazer a 'escada' de passagem de dados quando o aplicativo precisar se comunicar com o mundo externo.

```
class RawAppRepository @Inject constructor(
    private val request : RawAppModule.RawAppService
) {

    /** PRODUTOS */
    suspend fun loadProdutos(): List<Produto> {...}

    suspend fun loadProdutoById(oid : String): Produto {...}

    /** USERS */
    suspend fun loadUserByEmail(email : String): RawUser {...}

    suspend fun loadCurrentUserByEmail(email : String): RawUser {...}

    suspend fun writeUser(u : RawUser): RawUser {...}

    suspend fun loadCurrentUserReviews(email: String): List<Avaliation> {...}
```

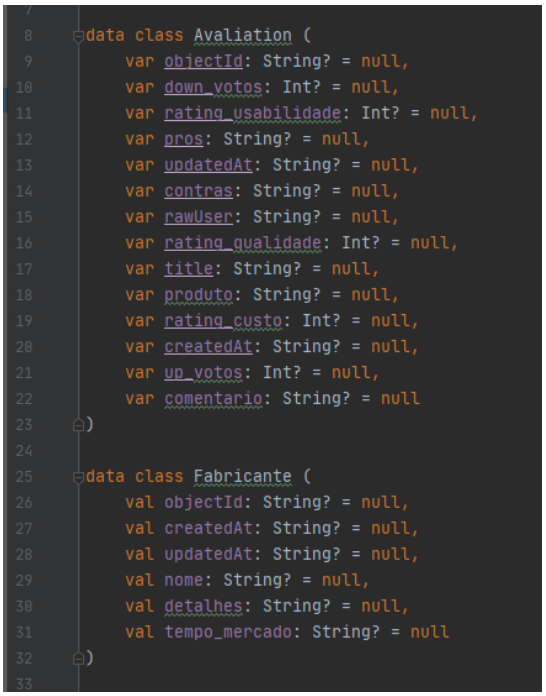
Fig. 8 - Camada repository, fazendo chamadas para o Retrofit e recebendo injeções

Neste momento notou-se que o nível de complexidade da execução do projeto era bem mais alto do que o antecipado. Nessa hora tomamos algumas decisões acerca do escopo, e de como executá-lo. A conexão com o DialogFlow teria de ser preterida, e isso deixaria de demonstrar uma importante parte do projeto que é a conexão via API. Para contornar essa falta tomamos a decisão de, em vez de hospedar localmente o banco de dados, fazê-lo online na plataforma https://parse-dashboard.back4app.com/. Apesar de o Parse oferecer um SDK para desenvolvimento em Kotlin decidimos por criar a nossa própria API, usando um servidor rodando online em uma instância do Heroku.

Para preparar essa conexão criou-se tanto a interface para que o Retrofit fizesse as chamadas, como o mapeamento das Data Classes para receber os objetos do banco de acordo com as nossas necessidades.



```
@Module
@InstallIn(SingletonComponent::class)
class RawAppModule {

    @Provides
    fun provideRetrofit() : Retrofit {

        val interceptor = HttpLoggingInterceptor()
        interceptor.setLevel(HttpLoggingInterceptor.Level.BODY)
        val client = OkHttpClient.Builder().addInterceptor(intercep

        return Retrofit.Builder()
            .baseUrl( baseUrl: "https://raw-api.herokuapp.com/api/")
            .addConverterFactory(GsonConverterFactory.create())
            .addCallAdapterFactory(RxJava2CallAdapterFactory.create
            .build()
    }

    @Provides
    fun provideRawAppService(retrofit: Retrofit) : RawAppService {.

    interface RawAppService{

        /** @GET("produtos") ...*/


        /** ******** ...*/

        @GET( value: "produtos")
        @Headers( ...value: "Content-Type: application/json")
        suspend fun getProducts(): List<Produto>

        @GET( value: "produtosBySubstring")
        @Headers( ...value: "Content-Type: application/json")
        /** ------------ > Data */
        suspend fun getProductsBySubstring(): List<Produto>

        @GET( value: "produtoById")
        @Headers( ...value: "Content-Type: application/json")
        suspend fun getProductById(@Query( value: "objectId") objectI
```

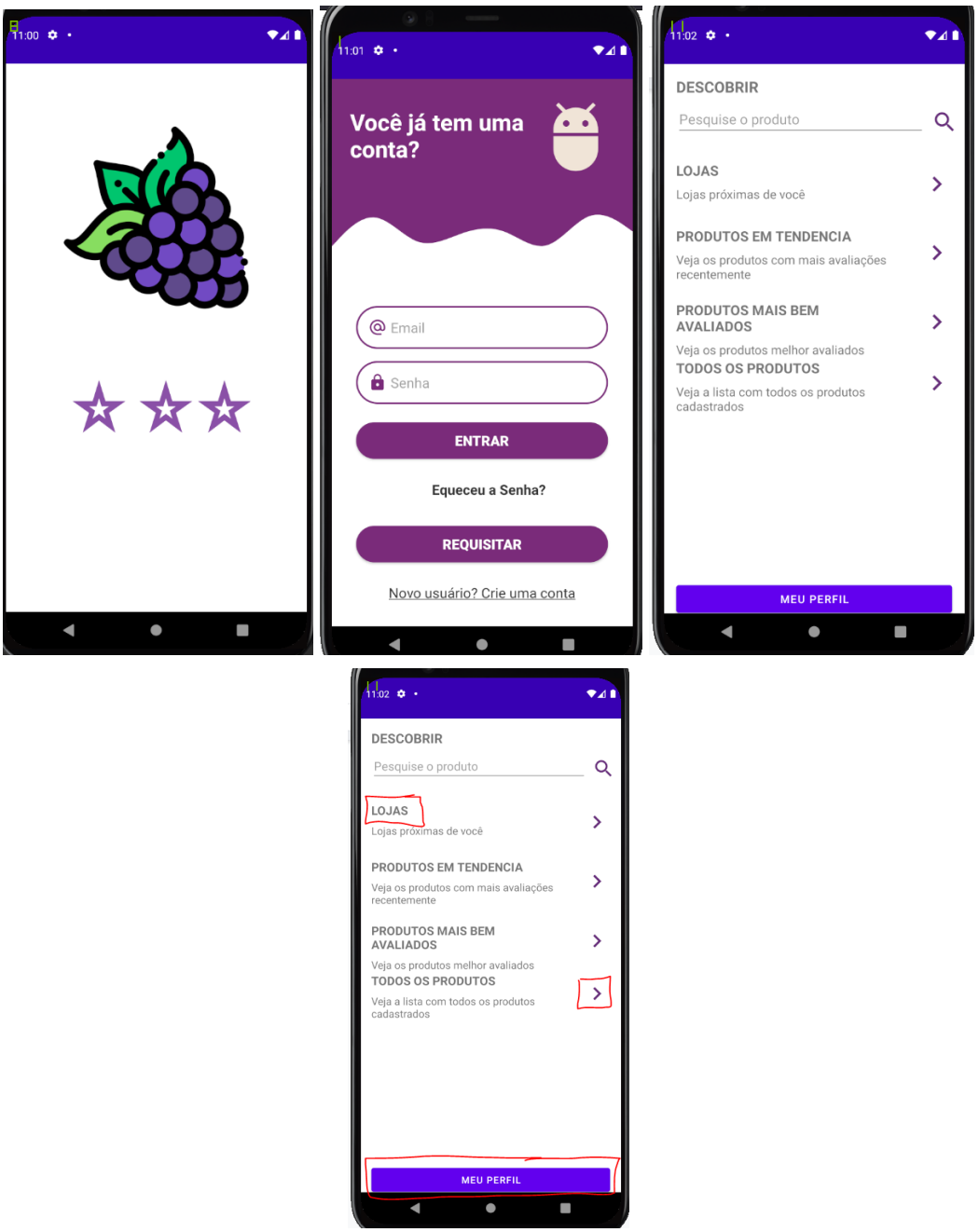


```
data class Avaliation (
    var objectId: String? = null,
    var down_votos: Int? = null,
    var rating_usabilidade: Int? = null,
    var pros: String? = null,
    var updatedAt: String? = null,
    var contras: String? = null,
    var rawUser: String? = null,
    var rating_qualidade: Int? = null,
    var title: String? = null,
    var produto: String? = null,
    var rating_custo: Int? = null,
    var createdAt: String? = null,
    var up_votos: Int? = null,
    var comentario: String? = null
)

data class Fabricante (
    val objectId: String? = null,
    val createdAt: String? = null,
    val updatedAt: String? = null,
    val nome: String? = null,
    val detalhes: String? = null,
    val tempo_mercado: String? = null
)
```

Figs. 9 e 10 - Interface para Retrofit, e Data Classes modelando nossas entidades.

Para escrever na tela contamos com data binding e Observers de objetos MutableLiveData, instanciados na nossa ViewModel. O fato de a viewModel nos oferecer subrotinas onde lidar paralelamente com requisições, por exemplo, sem 'sequestrar' os métodos do ciclo de vida padrão dos fragmentos foi imediatamente valioso para o desenvolvimento.

O onboarding usando fragmentos foi adicionado, assim como a tela de login com as funcionalidades de criação de conta, reset de senha e o próprio login.



Figs. 11 a 14 - Splash Screen, Login, Home e destaque os elementos com os quais se pode interagir

Uma tela Home foi adicionada pós-login, e alguns listeners foram adicionados para levar às próximas telas. A decisão de colocar em poucos elementos foi tanto fruto da revisão de escopo, quanto da ideia de experimentação com gatilhos a partir de diferentes tipos de elementos em tela.

Adicionada à tela de Perfil do usuário está a opção de alteração de campos do perfil. Isso foi feito usando data binding e observers nos objetos MutableLiveData da viewModel.

A requisição é feita ao abrir a tela de perfil, e os campos são alterados para editáveis com o clique do botão de editar. Assim que o usuário confirma a edição é feita a requisição PUT ao banco de dados Parse via API, e a resposta é enviada de volta ao fragmento de Perfil para que os dados mais atualizados sejam exibidos.

O uso de RecyclerView foi feito na tela de produtos. Uma funcionalidade que decidimos implementar é levar o usuário à página de um produto ao clicar em seu item na lista. Aqui foi fundamental o entendimento da estrutura da viewModel, injeção de dependência e do próprio Adapter responsável pelos itens. Como no Adapter não pode-se (pelo menos não de forma tão intuitiva) passar o Context do fragmento acabamos sobrescrevendo o método de criação do Adapter no fragmento para receber um parâmetro adicional, que é o contexto do próprio fragmento. Isso não teria sido possível sem a natureza extremamente orientada a objeto de todo esse framework de trabalho, e da linguagem num geral.

```
viewmodelListaProdutos.resultadoLoadProdutos.observe(viewLifecycleOwner) { listProduto ->
    binding.rvProduto.adapter = RawAdapter(listProduto, cont: this)
}
```

```
class RawAdapter @Inject constructor(
    private val lista: List<Produto>,
    private val cont : ListaProdutosFragment
) : RecyclerView.Adapter<ProdutoViewHolder>(){
```

Figs. 15 e 16 - Recebendo novo construtor na RawAdapter para usar o context.

O próximo desafio aqui foi passar como parâmetro de um fragmento para o outro um atributo (a chave primária no BD) para ser usado ao buscar as informações do produto no nosso banco de dados.

Esse objetivo foi atingido usando o conceito das Destinations e navArgs do AndroidX Navigation. Enquanto em um primeiro momento usamos sharedPrefs essa solução passa longe de ser ideal por ser globalmente acessível no app. Usando os argumentos de navegação a facilidade da sintaxe de ‘resgatar’ os parâmetros acelerou muito o processo, que daí pra frente seguiu a mesma lógica de alimentação de tela usando data binding.

A conexão com o Maps, que pode ser acessada clicando no texto Lojas na home screen, não atingiu o nível esperado de desenvolvimento. Por motivos similares aos que justificaram a decisão de preterir do DialogFlow decidimos ser bastante, para demonstrar a funcionalidade, a conexão com a API e disposição do elemento do Maps em tela. Foi feito o cadastro do projeto na GCP e a conexão com a aplicação. Nesse caso consideramos ser mais merecedora de foco a parte técnica, de conexão, do que a parte de visualização dos *pins* em tela. A colocação deles é paramétrica e relativamente simples quando se tem os dados estruturados.

Fig. 18 - Tela do Google Maps disponível na aplicação

5. Resultado e Conclusões

O desenvolvimento dessa aplicação se mostrou mais complexo e exigente do que era nossa expectativa. Atribuímos isso a não haver conhecimento prévio algum sobre essa linguagem e plataforma, somado ao grau altíssimo de abstração desse framework de trabalho. Apesar de incontestavelmente acelerar o desenvolvimento ele acaba ocultando muito da real ciência do sistema. O processo de debugging foi particularmente oneroso já que apesar de por óbvio a linha que causa o erro ser criada pelos desenvolvedores, o *caminho* que as chamadas fazem até levantar o erro é complexo e passa por muitos métodos que são criados pelo hilt-dagger dinamicamente baseados no próprio código. Isso quer dizer que as sintaxes não são padrão e buscar por informação em canais comuns como StackOverflow fica mais difícil.

Esse tipo de percalço motivou o visual pobre do aplicativo, e o fato de só haver poucas telas implementadas. A decisão do grupo foi a de implementar a parte “por trás dos panos”, partindo do pressuposto que a roupagem é mais facilmente alterável/configurável.

Em contrapartida foi muito enriquecedor poder aplicar as noções e intuições que aprendemos em disciplinas como Programação Orientada a Objetos, Estrutura de Dados e Engenharia de Software. Sentimos que graças a isso dominamos todos os aspectos teóricos e práticos das implementações que fizemos. Sem esses conhecimentos é seguro dizer que teria sido impossível elaborar esse aplicativo. Não só isso, como a própria disciplina de Programação de Dispositivos Móveis apresentou um volume enorme de conteúdo, que deu uma visão bem atualizada do mercado de desenvolvimento Android.

6. Referências

Google Developers. 2020. “Android Kotlin Fundamentals: 8.1 Getting data from the internet.” Android Kotlin Fundamentals. https://developer.android.com/codelabs/kotlin-android-training-internet-data?index=..%2F..android-kotlin-fundamentals&hl=es-419#0.

Google Developers. 2020. “Data Binding Library.” Android Docs. https://developer.android.com/topic/libraries/data-binding.

Google Developers. 2020. “Guide - Fragments.” Android Docs. https://developer.android.com/guide/fragments.

Google Developers. 2020. “RecycleView Overview.” Android Docs. https://developer.android.com/jetpack/androidx/releases/recyclerview.

Google Developers. 2020. “ViewModel Overview.” Android Docs. https://developer.android.com/topic/libraries/architecture/viewmodel.

“Introduction to Fragments.” 2020. GeeksForGeeks. https://www.geeksforgeeks.org/introduction-fragments-android/.

Simpson, Luke. 2017. “Clean Architecture with MVVMi, Architecture Components & RxJava.” Medium. https://medium.com/@thereallukesimpson/clean-architecture-with-mvvmi-architecture-components-rxjava-8c5093337b43.