



MODULARIZAÇÃO DE APPS

LISTAGEM DE PRODUTOS

Heider Pinholi Lopes

Versão 1

HISTÓRICO DE REVISÕES

Versão	Data	Responsável	Descrição
1	31/07/2020	Heider Pinholi Lopes.	Versão inicial do documento.

FICHA CATALOGRÁFICA

[NÃO PREENCHER - PARA USO DO DEPTO DE EAD E BIBLIOTECA]

A000a Sobrenome, Nome

Título [livro eletrônico] / Nome Sobrenome. -- São Paulo : Fiap, 2016.
x MB ; ePUB

Bibliografia.

ISBN 000-00-00000-00-0

Categoria. 2. Subcategoria. S., Nome. II. Título.

CDU 000.000.00

RESUMO

No desenvolvimento de aplicativos Android uma prática adotada pelos desenvolvedores é a modularização que dentre vários benefícios pode-se destacar: reutilização de código entre diferentes projetos, diminuição do tamanho do aplicativo para download para o usuário final, entre outros benefícios. Neste módulo, serão fornecidos conhecimentos essenciais para criar um aplicativo modularizado, de fácil manutenção e escalável.

Palavras-chave: Programação. Mobile. Desenvolvimento. Android. Kotlin. Gradle. Modularização, domain, data, presentation, repository.

LISTA DE FIGURAS

Figura 2.1 – Portal para teste de trechos de código em Kotlin

13

LISTA DE TABELAS

Tabela 2.1 – Tipos pré-definidos

17

LISTA DE CÓDIGOS-FONTE

Código-fonte 2.1 – Comentários	15
Código-fonte 2.2 – Variáveis	16
Código-fonte 2.3 – Exemplo com uso de declaração implícita	16
Código-fonte 2.4 – Criando constantes	17
Código-fonte 2.5 – Utilizando Inteiros	18
Código-fonte 2.6 – Utilizando Double e Float	19
Código-fonte 2.7 – String e Character	20
Código-fonte 2.8 – Caractere especial \$	20
Código-fonte 2.9 – Interpolação de Strings	21
Código-fonte 2.10 – Booleanos	21
Código-fonte 2.11 – Endereço em String	22
Código-fonte 2.12 – Trabalhando com null safety	22
Código-fonte 2.13 – Trabalhando com null safety (Teste de Tipo Nulo)	23
Código-fonte 2.14 – Arrays	24
Código-fonte 2.15 – Acessando itens de um Array	25
Código-fonte 2.16 – List	27
Código-fonte 2.17 – Set	28
Código-fonte 2.18 – Dicionários	31
Código-fonte 2.19 – Operador de atribuição	32
Código-fonte 2.20 – Operadores de atribuição	32
Código-fonte 2.21 – Operadores compostos	33
Código-fonte 2.22 – Operadores lógicos	33
Código-fonte 2.23 – Operadores de comparação	34
Código-fonte 2.24 – Operadores ternário	34
Código-fonte 2.25 – Operador Coalescência nula	34
Código-fonte 2.26 – Operadores Closed Range e Half Closed Range	35
Código-fonte 2.27 – If – else – else if	36
Código-fonte 2.28 – When	37
Código-fonte 2.29 – While, do while	38
Código-fonte 2.30 – For in	40
Código-fonte 2.31 – enum	41
Código-fonte 2.32 – Enum com valores padrões	42
Código-fonte 2.33 – Funções	44
Código-fonte 2.34 – Exemplo de função apresentando Fibonacci	45
Código-fonte 2.35 – Função que retorna outra função	45
Código-fonte 2.36 – Resumo de Map, Filter e Reduce	46
Código-fonte 2.37 – Map	47
Código-fonte 2.38 – Filter	47
Código-fonte 2.39 – Reduce	48
Código-fonte 2.40 – Generics	50
Código-fonte 2.41 – Classes	51
Código-fonte 2.42 – Propriedades computadas	53
Código-fonte 2.43 – Propriedades e métodos de classe	55
Código-fonte 2.44 – Herança	58
Código-fonte 2.45 – Sobrescrita	60

SUMÁRIO

2 TRY KOTLIN	11
2.1 Apresentação	11
2.2 Principais características	11
2.3 Por que desenvolver para Android com Kotlin?	12
2.4 REPL (Read-Eval-Print Loop)	12
2.4.1 Ambiente de estudos	12
2.5 Comentários e variáveis	14
2.5.1 Comentários	14
2.5.2 Variáveis e Constantes	15
2.6 Tipos	17
2.6.1 Tipos Inteiros (Long, Int, Short e Byte)	17
2.6.2 Double e Float (Números com casas decimais)	18
2.6.3 String e Char	19
2.6.4 Bool (Booleanos)	21
2.6.5 Pair (Par)	21
2.6.6 Tipo Nullable (null safety)	22
2.7 Coleções	23
2.7.1 Array	23
2.7.2 List	25
2.7.3 Set	27
2.7.4 Map	29
2.8 Operadores	31
2.8.1 Atribuição (=)	32
2.8.2 Aritméticos (+, -, *, /, %)	32
2.8.3 Compostos (+, -, *, /, %, ++, --)	32
2.8.4 Operadores Lógicos (&&, , !)	33
2.8.5 Operadores de Comparação (>, <, >=, <=, ==, !=)	33
2.8.6 Estrutura de decisão em mesma linha	34
2.8.7 Coalescência nula (?:)	34
2.8.8 Closed Range(..) e Half Closed Range (until)	35
2.9 Estruturas condicionais e de repetição	35
2.9.1 If – else – else if	36
2.9.2 When	36
2.9.3 While / do while	38
2.9.4 For in	39
2.10 Enumeradores	40
2.10.1 Valores padrões	42
2.11 Funções e closures	43
2.11.1 Funções	43
2.11.2 Criando funções	43
2.11.3 Single-Expression functions	45
2.12 Map, Filter e Reduce	45
2.13 Generics	49
2.14 Classes	50
2.14.1 Definição e construção	50
2.14.2 Propriedades computadas	52

2.14.3 Propriedades/métodos de classe	53
2.14.4 Herança	56
2.14.5 Sobrescrita	58
2.15 Considerações sobre a Introdução ao Kotlin	60
REFERÊNCIAS	62

1 Os APLICATIVOS MODULARES - MODULAR APP

Modularizar o aplicativo é o processo de separar componentes lógicos do projeto de aplicativo em módulos discretos, ou seja, partes do nosso aplicativo, que possuem responsabilidades distintas e podem interagir entre si.

1.1 Benefícios da modularização

Criar aplicativos modulares corretamente traz os seguintes benefícios para o desenvolvimento:

1.1.1 Escala e manutenibilidade

Com o crescimento do código do aplicativo e o aumento da equipe que irá manter o projeto, trabalhar em um único módulo pode ocasionar diversos problemas no dia-a-dia e na evolução do produto. Ao separar o app em diferentes componentes, os desenvolvedores podem funcionar melhor dentro do seu domínio e evitar sobreposição de trabalho. Além disso, procurar por uma classe, layout ou um recurso específico em um app modular tende a ser mais rápido uma vez que não é preciso procurar no projeto todo.

1.1.2 Construção do projeto de forma mais rápida

Os aplicativos para serem gerados possuem tempos de build mais rápidos em um aplicativo modular. Quando você altera um único arquivo em um aplicativo monolítico, normalmente é compilado todo o projeto, porém em um aplicativo modular apenas as partes afetadas são compiladas novamente.

1.1.3 Apks menores

Ao distribuir alguns recursos do seu aplicativo sob demanda, ou seja, ele poderá ser muito menor. Neste cenário, pode haver alguns recursos que serão

incluídos posteriormente (por exemplo: recursos pagos ou funcionalidades que são utilizadas somente por um grupo de usuários do seu aplicativo). Recentemente o Google introduziu a entrega dinâmica, onde é possível incluir alguns recursos do seu aplicativo além do módulo base. Essa entrega de recursos do aplicativo pode ser condicional, dependendo do dispositivo ou das necessidades do usuário.

1.1.4 Código reutilizável

Ao modularizar nosso código em seções dissociadas, podemos compartilhar facilmente coisas reutilizáveis, o que nos poupará bastante tempo, em vez de escrever código repetido. Por exemplo, uma biblioteca de componentes.

2 Projeto

Neste projeto será desenvolvido um aplicativo responsável em buscar uma lista de produtos através de uma API. O app será dividido em módulos para que as responsabilidades fiquem separadas e o código fique desacoplado tornando o projeto mais escalável.

Vamos começar pela modularização por camadas e na sequência apresentamos a modularização por recurso (será desenvolvido um módulo dinâmico onde ele só será disponibilizado para o usuário caso ele utilize a funcionalidade tornando o aplicativo menor no momento do download).

Segue as imagens do aplicativo final:



Figura 2 – Menu Principal com botão para baixar módulo
Fonte: Próprio autor (2020)



Figura 2 – Menu Principal com botão para exibir ou excluir
Fonte: Próprio autor (2020)

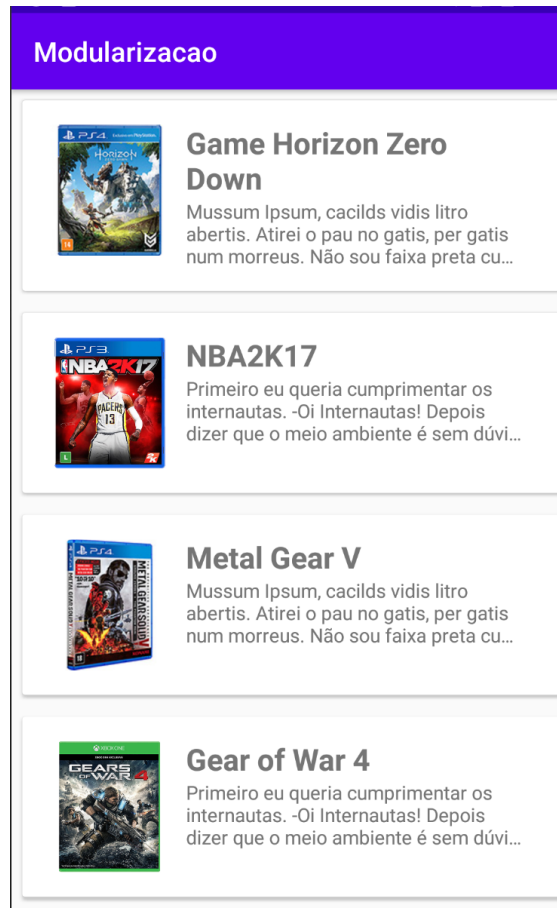


Figura 2 – Listagem de produtos
Fonte: Próprio autor (2020)

2.1 Criando o projeto

Abra o Android Studio e clique em **Start a new Android Studio Project:**

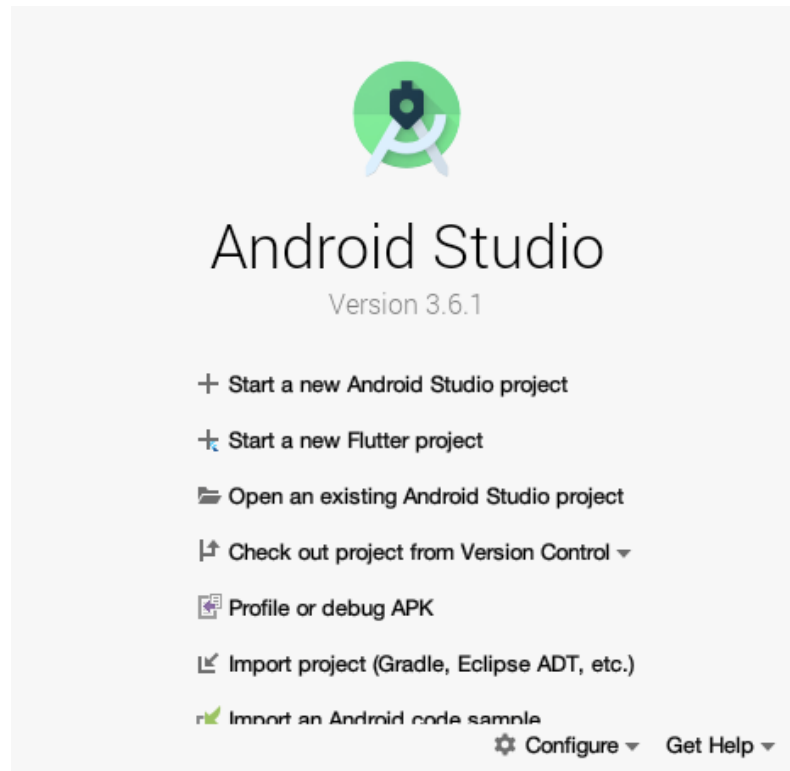


Figura 2.1 – Criação do projeto
Fonte: Próprio autor (2020)

Em seguida, selecione **Empty Activity** e clique em **Next**.

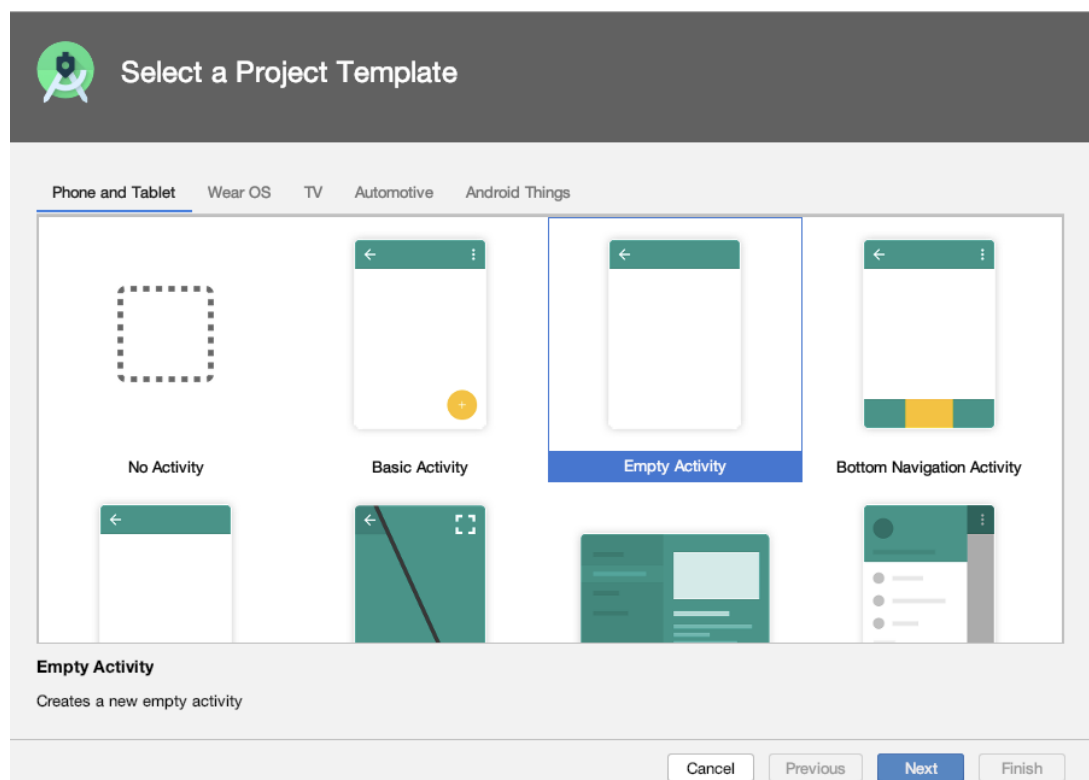
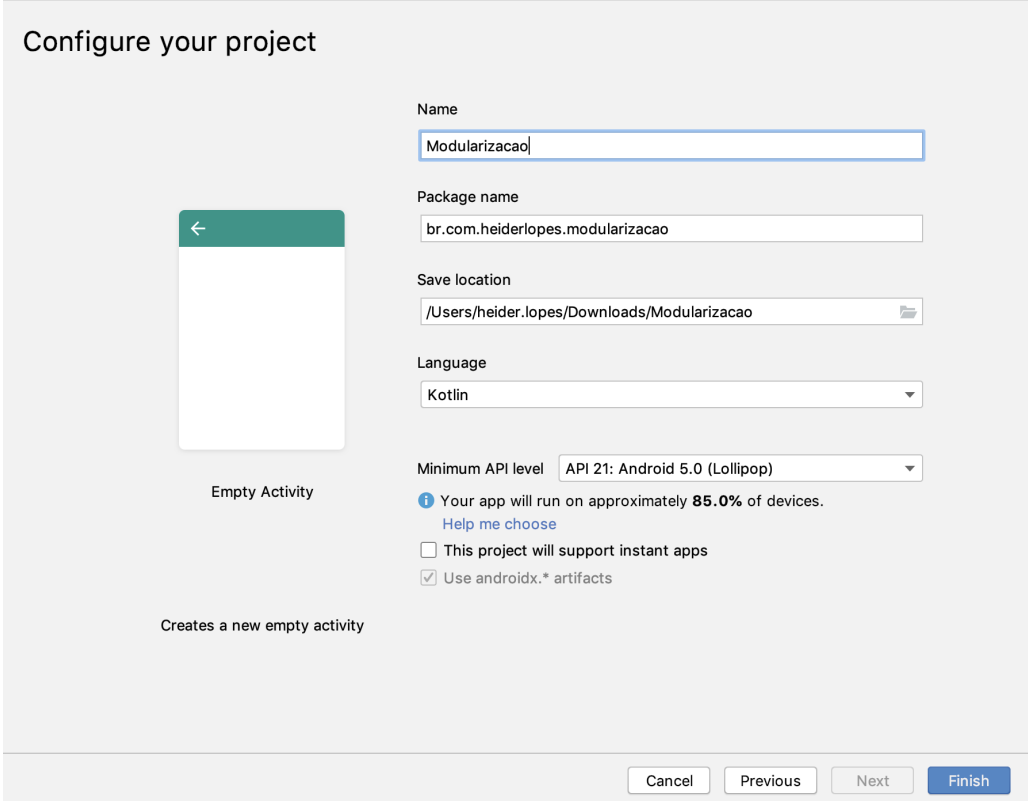


Figura 2.1 – Seleção de Template do Projeto
Fonte: Próprio autor (2020)

Configure o seu projeto definindo o name, package name e language conforme a próxima imagem:



Configure your project

Name
Modularizacao

Package name
br.com.heiderlopes.modularizacao

Save location
/Users/heider.lopes/Downloads/Modularizacao

Language
Kotlin

Minimum API level
API 21: Android 5.0 (Lollipop)

i Your app will run on approximately **85.0%** of devices.
[Help me choose](#)

☐ This project will support instant apps

☒ Use androidx.* artifacts

Empty Activity

Creates a new empty activity

Cancel Previous Next Finish

Figura 2.1 – Seleção de Template do Projeto
Fonte: Próprio autor (2020)

3 Tipos de modularização

Podemos modularizar um aplicativo basicamente de duas maneiras: feature (por recurso) ou por layer (camada).

3.1 Modularização por camadas (layers)

Neste tipo de modularização, cada camada tem uma certa funcionalidade. À maioria dos aplicativos normalmente possuem as seguintes camadas: **domain**, **data** e **presentation**. Tais camadas serão apresentadas durante o desenvolvimento do projeto deste módulo.

3.1.1 Domain Module

O Domain é responsável pela comunicação com o **Presentation Module**. Ele é um module **kotlin/java puro**, ou seja, **não possui dependências Android**. Dentro deste módulo ficam:

Entidades: são as entidades que possuem somente os dados que serão enviados para o ViewModel/Presenter, ou seja, o dado mapeado do backend, por exemplo.

UseCases: é onde são escritas as regras de negócios com base nos dados que são solicitados do repository. Ele é blindado, ou seja, as mudanças feitas aqui não devem afetar outros módulos do projeto, assim como mudanças em outros módulos não devem refletir no UseCase.

Repository: é a interface de comunicação, que solicita dados (seja backend ou do cache).

O módulo domain possui o seguinte diagrama de fluxo:

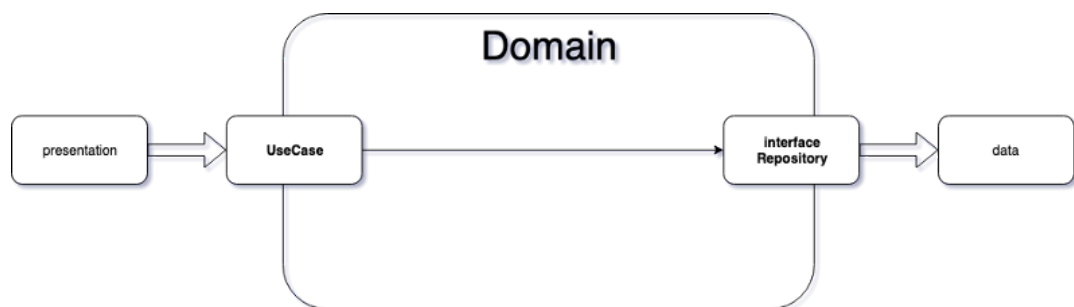


Figura 3.1.1 – Diagrama de fluxo do domain module
Fonte: Próprio autor (2020)

O **presentation** solicita algum dado para **UseCase**, que pede para o **repository**, que vai buscar onde ele está implementado. O dado vem para o **UseCase**, o qual pode aplicar alguma regra de negócio e devolve para o **presentation**.

3.1.1.1 Criando o Domain Module

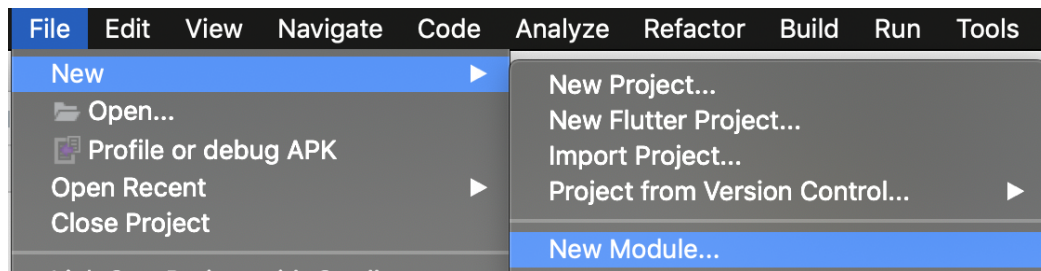


Figura 3.1.1.1 – Criando um novo módulo
Fonte: Próprio autor (2020)

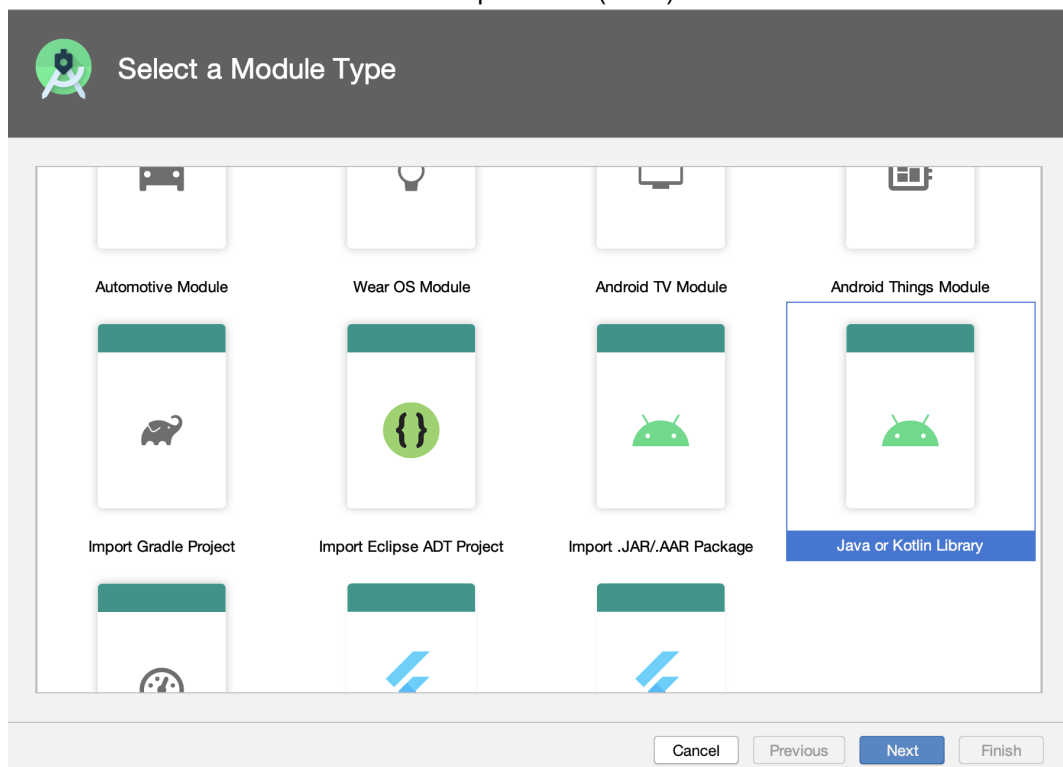
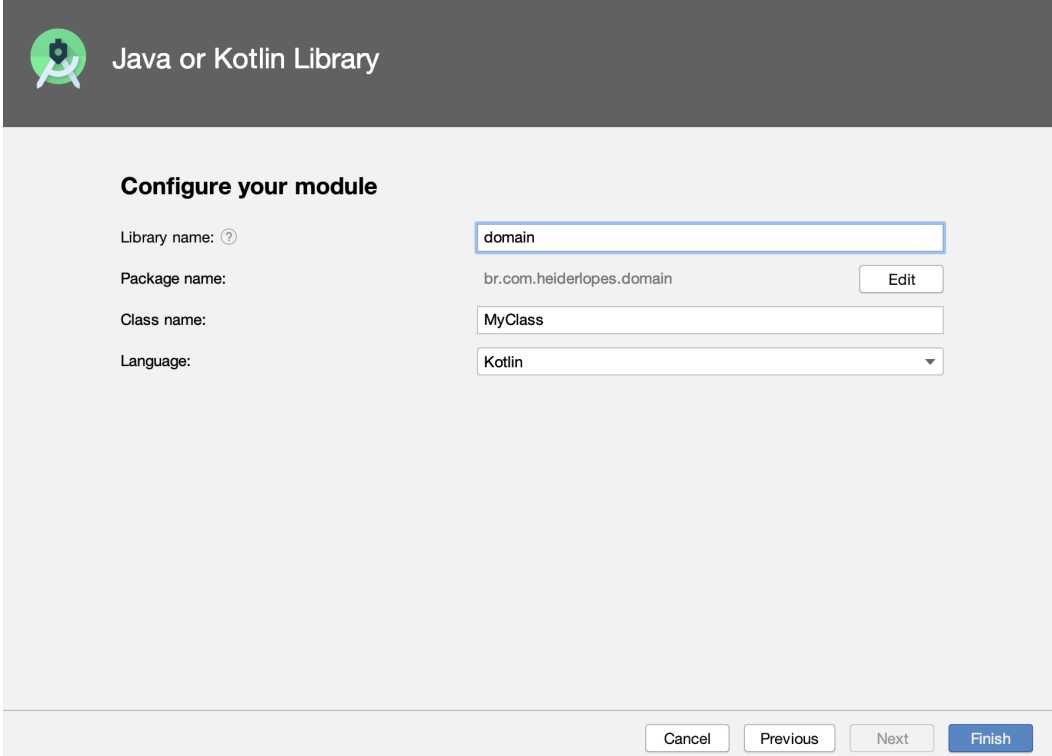


Figura 3.1.1.1 – Criando uma Java or Kotlin Library
Fonte: Próprio autor (2020)



Java or Kotlin Library

Configure your module

Library name:

Package name:

Class name:

Language:

Figura 3.1.1.1 – Definindo o módulo domain
Fonte: Próprio autor (2020)

Para melhorar o gerenciamento das dependências do projeto, é possível criar um arquivo onde serão centralizadas as dependências do projeto. Neste arquivo serão encontradas as libs, suas versões entre outras coisas. Ao longo do projeto esse arquivo será evoluído de acordo com a necessidade.

Crie um arquivo chamado **dependencies.gradle** na raiz do projeto. Para isso, altere o modo de visualização de Android para Project.

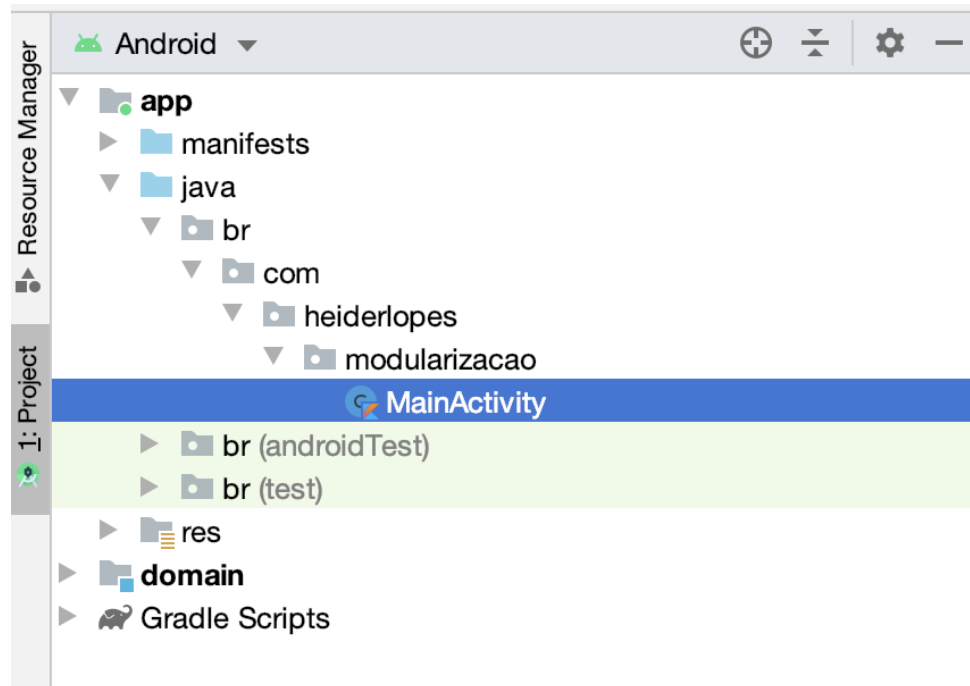


Figura 3.1.1.1 – Modo de visualização Android
Fonte: Próprio autor (2020)

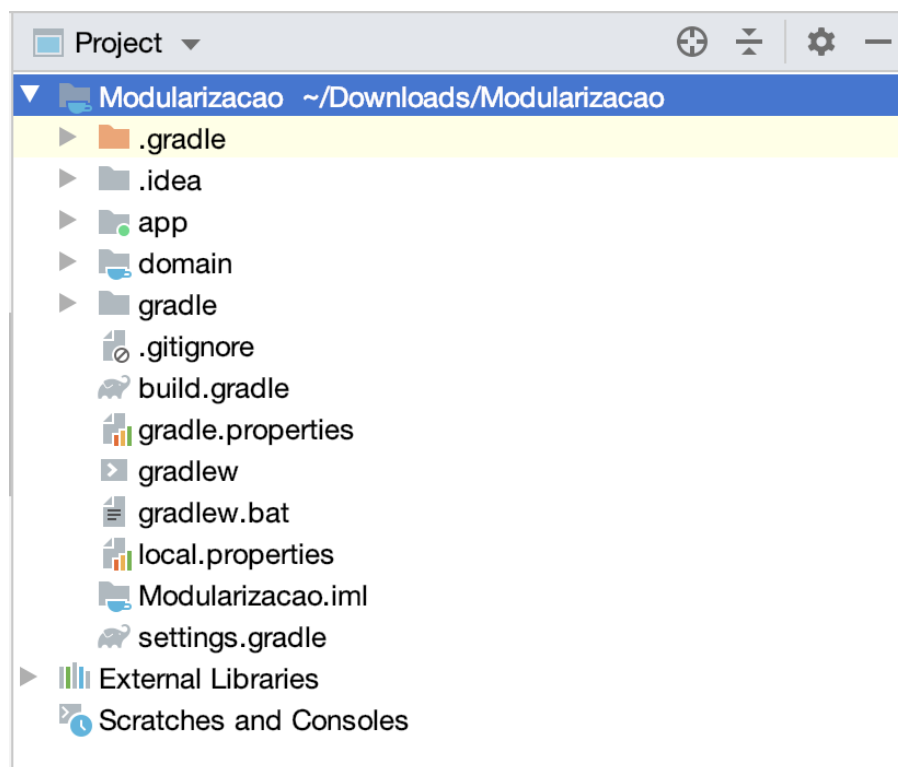


Figura 3.1.1.1 – Modo de visualização Android
Fonte: Próprio autor (2020)

Clique com o botão direito sobre o nome do projeto (Modularização), New ⇒

File

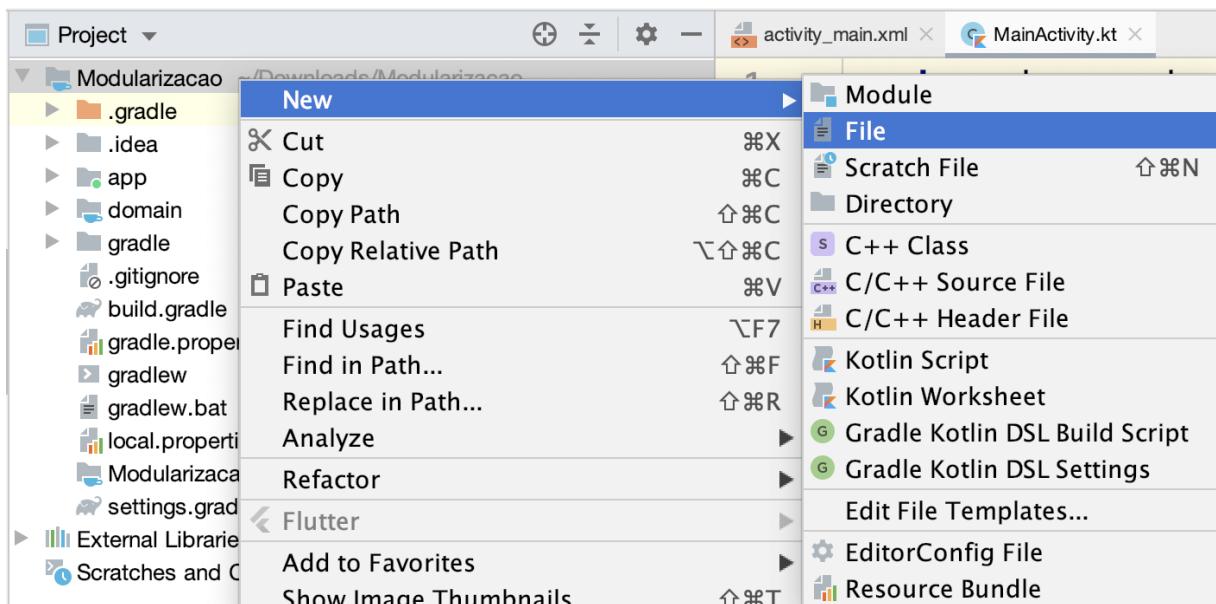


Figura 3.1.1.1 – Criação do arquivo dependencies.gradle

Fonte: Próprio autor (2020)

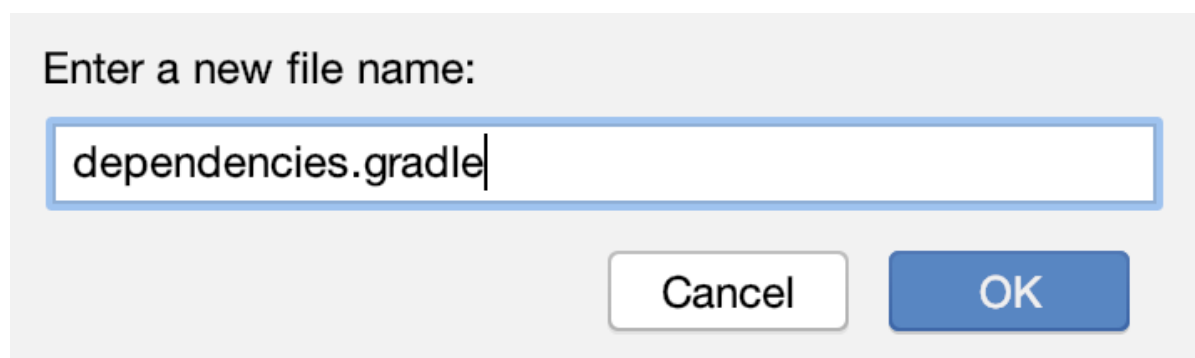


Figura 3.1.1.1 – Nomeação do arquivo dependencies.gradle

Fonte: Próprio autor (2020)

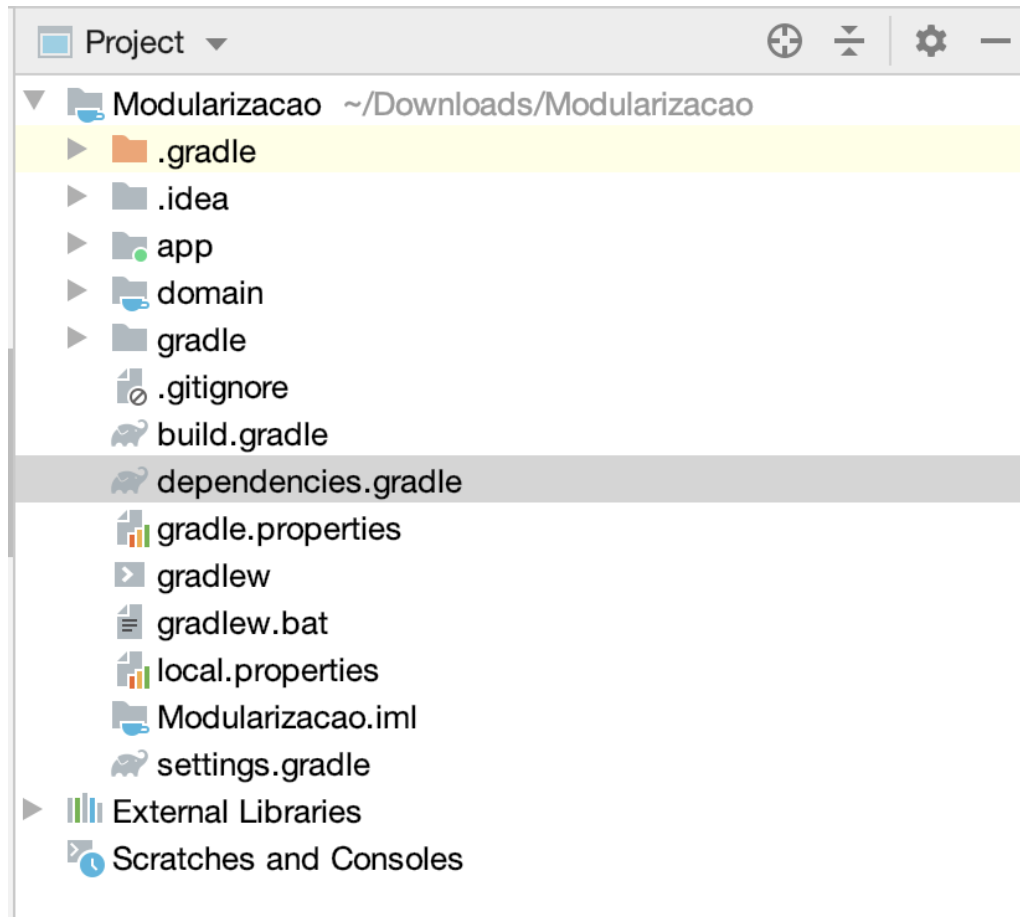


Figura 3.1.1.1 – Arquivo dependencies.gradle na raiz do projeto
Fonte: Próprio autor (2020)

Dentro deste arquivo, coloque as versões das libs e um array de dependências com suas respectivas versões.

```
ext {  
  
    minSDK = 20  
    targetSDK = 28  
    compileSDK = 28  
  
    buildTools = '3.5.0'  
  
    appCompactVersion = '1.0.2'  
    kotlinVersion = '1.3.21'  
  
    AndroidArchVersion = '1.1.1'  
    databindingVersion = '3.1.4'  
    lifecycleVersion = '2.0.0'  
    ktxVersion = '1.0.1'
```

```
constrainVersion = '1.1.3'
cardViewVersion = '1.0.0'
recyclerViewVersion = '1.0.0'

//Rx
rxJavaVersion = '2.2.7'
rxKotlinVersion = '2.4.0'
rxAndroidVersion = '2.1.1'

//Koin
koinVersion = '2.0.1'

//Retrofit
retrofitVersion = '2.3.0'

//Okhttp
okhttpVersion = '3.2.0'

//Gson
gsonVersion = '2.8.5'

//Room version
roomVersion = '2.1.0'

//Test
junitVersion = '4.12'
espressoVersion = '3.1.1'
runnerVersion = '1.1.1'

dependencies = [
    kotlin: "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlinVersion",

    appCompact: "androidx.appcompat:appcompat:$appCompactVersion",
    constraintlayout:
"androidx.constraintlayout:constraintlayout:$constrainVersion",
    cardView: "androidx.cardview:cardview:$cardViewVersion",
    recyclerView: "androidx.recyclerview:recyclerview:$recyclerViewVersion",

    viewModel: "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifeCycleVersion",
    lifeCycle: "android.arch.lifecycle:extensions:$AndroidArchVersion",

    dataBinding: "com.android.databinding:compiler:$databindingVersion",

    ktx: "androidx.core:core-ktx:$ktxVersion",

    rxJava: "io.reactivex.rxjava2:rxjava:$rxJavaVersion",
    rxKotlin: "io.reactivex.rxjava2:rxkotlin:$rxKotlinVersion",
```

```
    rxAndroid: "io.reactivex.rxjava2:rxandroid:$rxAndroidVersion",

    koin: "org.koin:koin-android:$koinVersion",
    koinViewModel: "org.koin:koin-androidx-viewmodel:$koinVersion",

    retrofit: "com.squareup.retrofit2:retrofit:$retrofitVersion",
    retrofitRxAdapter: "com.squareup.retrofit2:adapter-rxjava2:$retrofitVersion",
    retrofitGsonConverter:
"com.squareup.retrofit2:converter-gson:$retrofitVersion",
    gson: "com.google.code.gson:gson:$gsonVersion",

    room: "androidx.room:room-runtime:$roomVersion",
    roomRxJava: "androidx.room:room-rxjava2:$roomVersion",
    roomCompiler: "androidx.room:room-compiler:$roomVersion"
]

testDependencies = [
    junit: "junit:junit:$junitVersion",
    espresso: "androidx.test.espresso:espresso-core:$espressoVersion",
    runner: "androidx.test:runner:$runnerVersion"
]
}
```

Código-fonte 3.1.1.1 – Arquivo dependencies.gradle com as dependências do projeto
Fonte: Próprio autor (2020)

Agora já é possível utilizá-lo no projeto. Primeiro, será configurado o **build.gradle** do projeto:

Configurar: **apply from: 'dependencies.gradle'** dentro do buildScript e configurar as dependências.

```
buildscript {

    apply from: 'dependencies.gradle'

    ext.kotlin_version = '1.3.41'
    repositories {
        google()
        jcenter()
    }

    dependencies {
```

```
classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlinVersion"
classpath "com.android.tools.build:gradle:$buildTools"
classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
}
}

allprojects {
    repositories {
        google()
        jcenter()
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

Código-fonte 3.1.1.1 – Aplicando o dependencies.gradle no projeto
Fonte: Próprio autor (2020)

Abra o arquivo **build.gradle** do **domain** e crie uma variável dependencies que capta todas as dependências do arquivo criado anteriormente.

```
apply plugin: 'java-library'
apply plugin: 'kotlin'

dependencies {

    def dependencies = rootProject.ext.dependencies

    implementation dependencies.kotlin
    implementation dependencies.rxJava
    implementation dependencies.koin
}

sourceCompatibility = JavaVersion.VERSION_1_8
targetCompatibility = JavaVersion.VERSION_1_8
```

Código-fonte 3.1.1.1 – Aplicando o dependencies.gradle no projeto
Fonte: Próprio autor (2020)

Com isso, o projeto terá todas as dependências centralizadas em apenas um lugar, ou seja, se outros módulos utilizam RxJava, dessa forma é mais simples garantir que todos os módulos terão a mesma versão da lib. Com isso, evita-se de ter um módulo com versões diferentes de outros e, quando vamos atualizar para versões mais novas, todos os módulos são atualizados.

3.1.1.2 Pacotes do domain

O projeto irá consumir o seguinte serviço: <http://www.mocky.io/v2/5de6d2643700004f00092633> e ele irá retornar uma lista com várias informações referentes aos jogos que serão exibidos na listagem. Segue um exemplo que será retornado pela api:

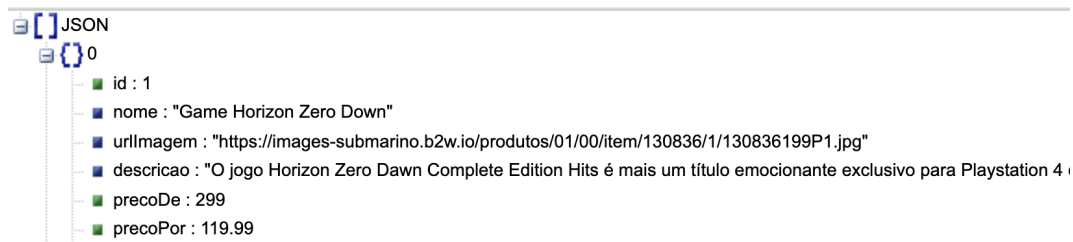


Figura 3.1.1.2 – Exemplo de jogo que será retornado pelo serviço
Fonte: Próprio autor (2020)

Para organizar o projeto de deixá-lo mais estruturado, crie os seguintes pacotes: `entity`, `repository`, `useCases` e `di`.

3.1.1.2.1 Entity

É onde são criadas as classes de dados. A primeira que será criada no projeto será para representar o produto. Crie um pacote chamado **entity** e dentro dele um data class chamado **Product**.

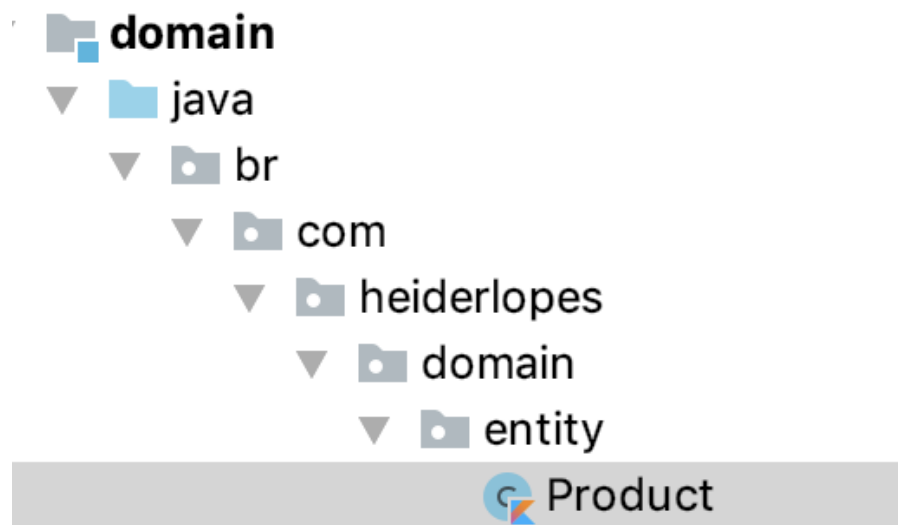


Figura 3.1.1.2.1 – Classe Product dentro do package
Fonte: Próprio autor (2020)

Adicione o seguinte código para representar um **produto** no aplicativo:

```
data class Product(  
    val name: String,  
    val imageURL: String,  
    val description: String  
)
```

Código-fonte 3.1.1.2.1 – Código da classe Product
Fonte: Próprio autor (2020)

3.1.1.2.2 Repository

Aqui ficam as interfaces de comunicação com o módulo **data**. O primeiro repository a ser criado irá retornar um objeto observável de **Product**.

Crie um pacote chamado **repository** e dentro dele uma interface chamada **ProductRepository**.

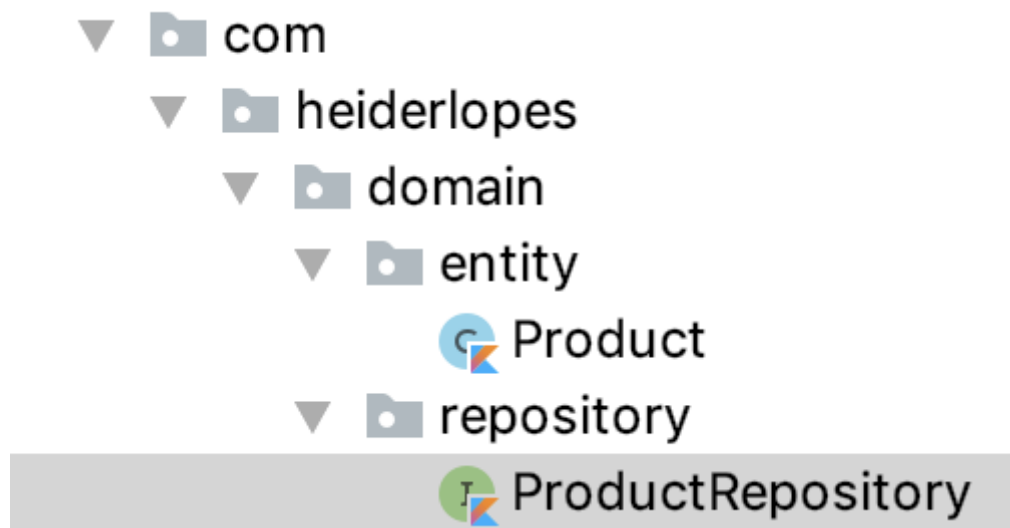


Figura 3.1.1.2.2 – Interface ProductRepository
Fonte: Próprio autor (2020)

```
interface ProductRepository {  
    fun getProducts(forceUpdate: Boolean): Single<List<Product>>  
}
```

Código-fonte 3.1.1.2.2 – Interface ProductRepository
Fonte: Próprio autor (2020)

3.1.1.2.3 UseCases

Os casos de usos serão chamados pela camada de apresentação. O primeiro UseCase será para trazer a lista com os produtos.

Crie um pacote chamado **usecases** e dentro dele uma classe chamada **GetProductsUseCase**

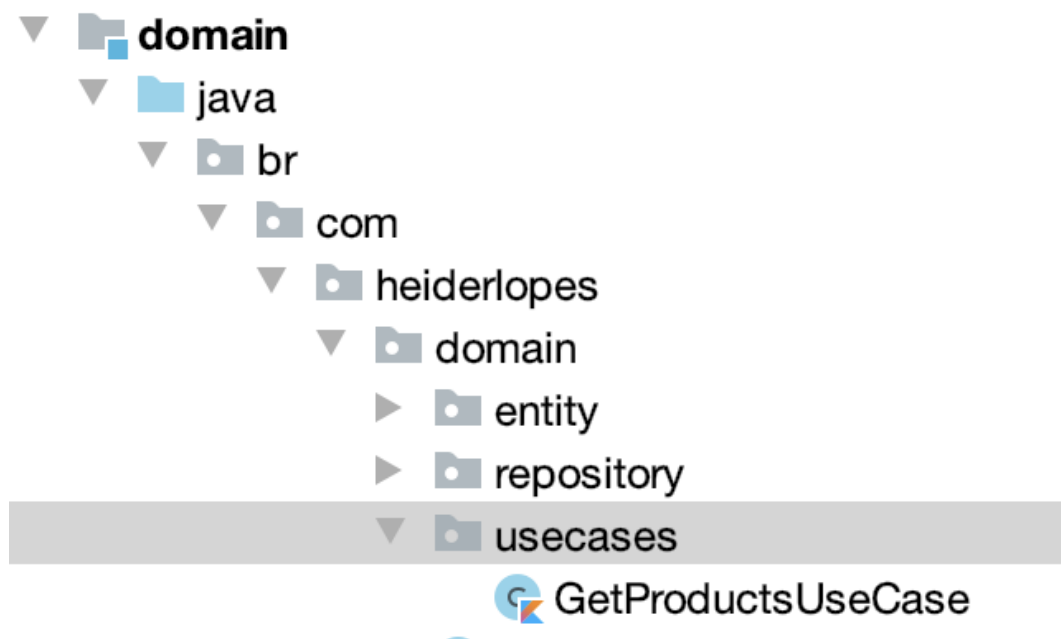


Figura 3.1.1.2.3 – Caso de uso para busca de produtos
Fonte: Próprio autor (2020)

```
class GetProductsUseCase(  
    private val productRepository: ProductRepository,  
    private val scheduler: Scheduler  
) {  
    fun execute(forceUpdate: Boolean): Single<List<Product>> {  
        return productRepository.getProducts(forceUpdate)  
            .subscribeOn(scheduler)  
    }  
}
```

Código-fonte 3.1.1.2.3 – Caso de uso para busca de produtos
Fonte: Próprio autor (2020)

No construtor será utilizada duas dependências necessárias para o **UseCase**: o **repository** (de onde será solicitada a lista de dados) e um **scheduler** (para informar a thread que irá assinar a chamada). Essas duas dependências serão entregues utilizando **injeção de dependência** através do framework **koin**.

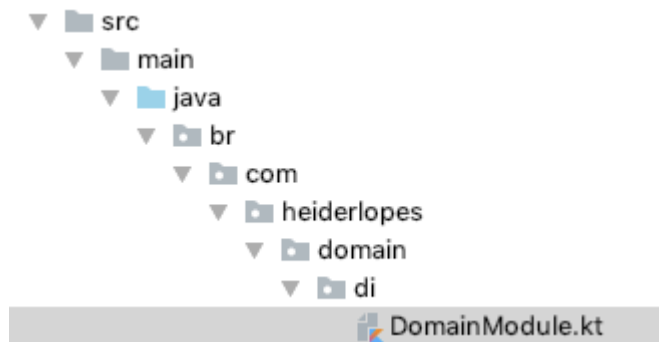


Figura 3.1.1.2.3 – Classe com os módulo para utilização de injeção de dependências
Fonte: Próprio autor (2020)

```
val useCaseModule = module {
    factory {
        GetProductsUseCase(
            productRepository = get(),
            scheduler = Schedulers.io()
        )
    }
}

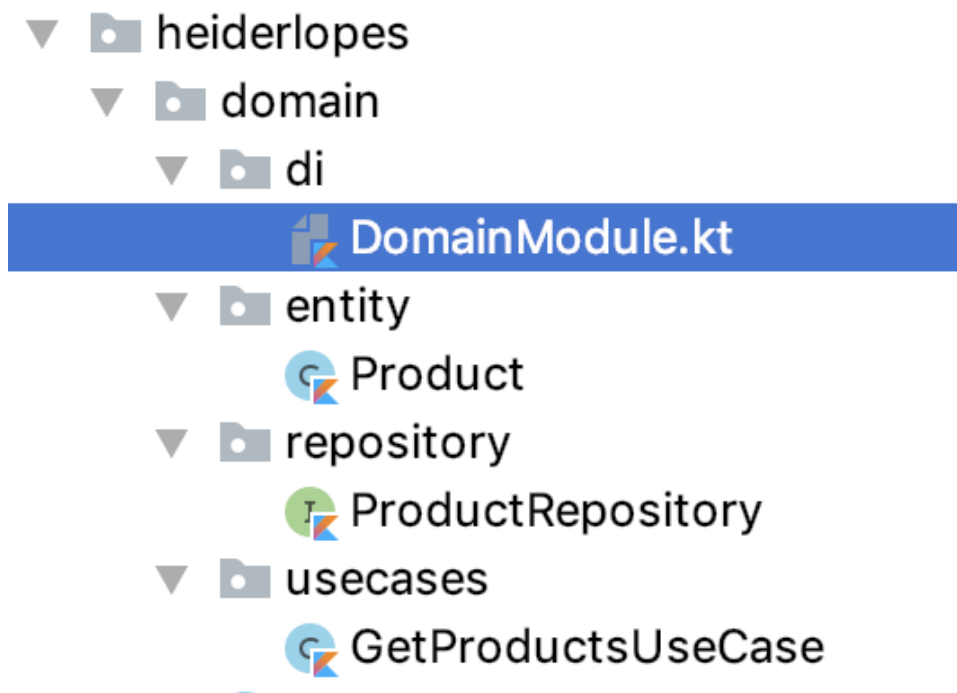
val domainModule = listOf(useCaseModule)
```

Código-fonte 3.1.1.2.3 – Classe com os módulo para utilização de injeção de dependências
Fonte: Próprio autor (2020)

A variável chamada **useCaseModule** receberá um **module koin** e, dentro desse module, estão as dependências que serão providas. O **factory** irá criar uma nova instância toda vez que for requerida essa dependência.

O repository = get() é onde o **get** significa que em algum lugar do projeto essa dependência já foi criada e só vamos pegá-la para utilizar no **UseCase**. E, para o scheduler = Schedulers.io(), será passado o Scheduler que será utilizado. Nesse caso, foi o IO.

O módulo no momento estará da seguinte forma:



Código-fonte 3.1.1.2.3 – Domain module
Fonte: Próprio autor (2020)

3.1.2 Data Module

Todos os projetos Android possuem dados, os quais precisam ser fornecidos de algum lugar, e é justamente isso que o module data faz para nós. Esses dados podem vir de qualquer lugar, como de alguma API ou database.

Quando a domain pede algum dado, ela não sabe de onde eles são fornecidos, pois isso é responsabilidade do módulo data.

Neste módulo são encontrados:

Api: aqui são localizados todos os endpoints que serão utilizados para requisitar dados do backend.

Model: aqui ficam as entidades que vêm do backend ou da cache, ou seja, o dado puro, que só é utilizado no módulo data.

Mapper: onde são mapeados os models para as entidades exigidas pela domain.

RepositoryImpl: aqui fica a implementação da interface repository do domain e é onde é decidido de qual lugar será pego os dados se do cache ou do backend.

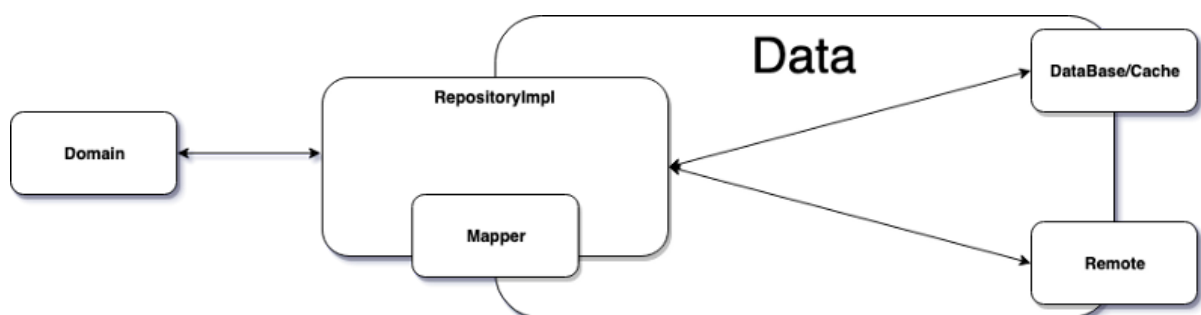
CacheDataSource: interface de comunicação que é implementada na cache para pegar os dados localmente.

CacheDataSourceImpl: aqui será gravados os dados na cache e fornecê-los já mapeados.

RemoteDataSource: interface de comunicação que é implementada no remote para pegar dados do backend.

RemoteDataSourceImpl: aqui será chamada a server api, para pegar os dados do backend e enviar para quem solicitou já mapeados.

Diagrama de fluxo do módulo data:



Código-fonte 3.1.2 – Data module
Fonte: Próprio autor (2020)

A **domain** solicita algum dado para o seu **repository**, que está implementado no **RepositoryImpl**, que então decide de onde vai buscar os dados solicitados.

Primeiro, será chamada o **cache** para verificar se tem algum dado para retornar e, caso não tenha, será chamado o **remote** e será gravado esses dados na **cache** e, então, será retornado os dados requeridos para a domain.

3.1.2.2 Criando o Data Module

Crie um novo módulo Android Library com o nome **data**.

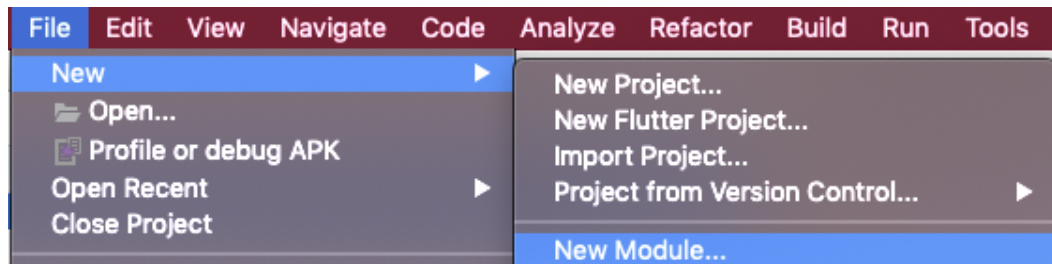


Figura 3.1.2.2 – Criando o Data module
Fonte: Próprio autor (2020)

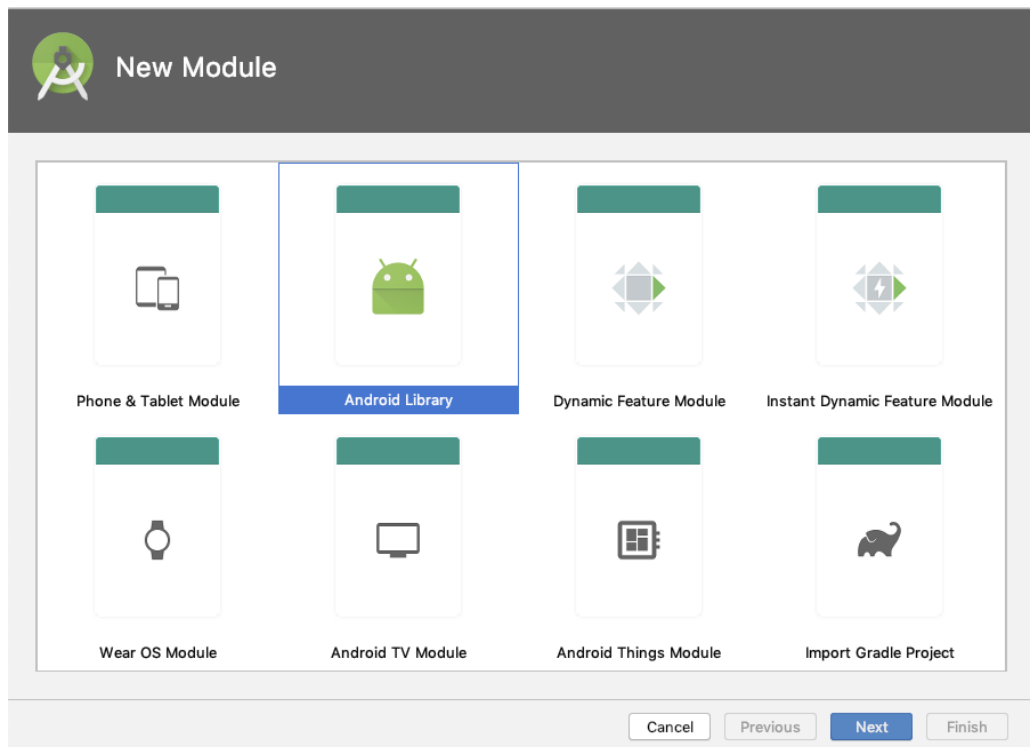


Figura 3.1.2.2 – Criando o novo módulo como Android Library
Fonte: Próprio autor (2020)

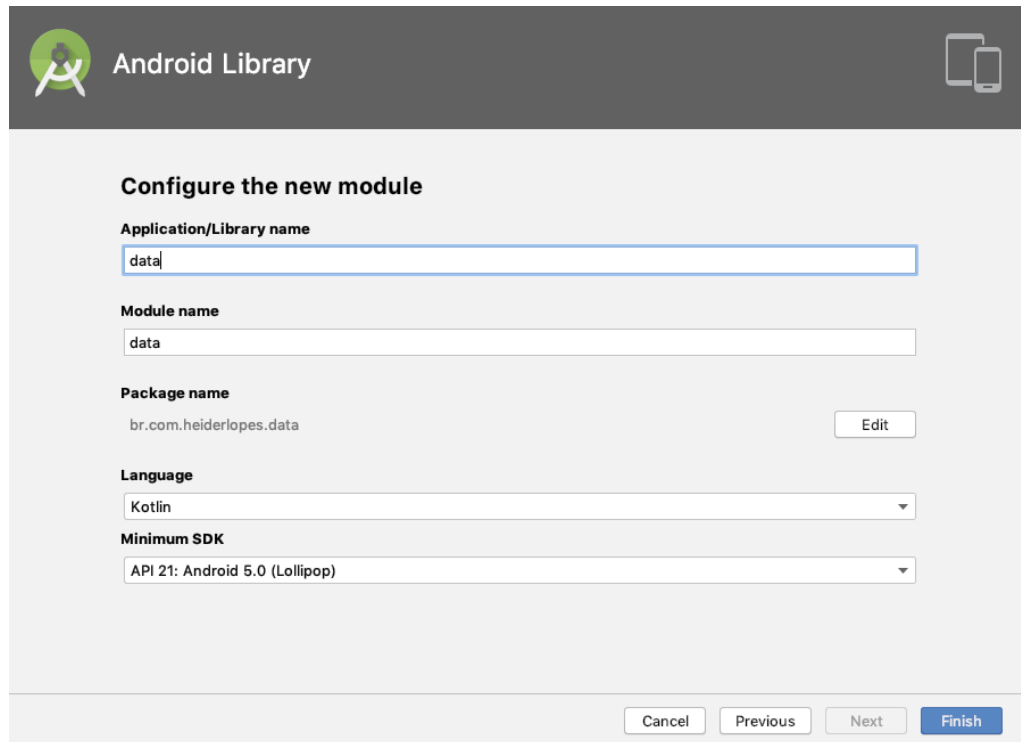


Figura 3.1.2.2 – Definindo o nome do módulo como data

Fonte: Próprio autor (2020)

Nessa parte será utilizada as dependências já adicionada no projeto (arquivo **dependencies.gradle**). Como o projeto irá realizar chamadas para o backend, será utilizada a biblioteca **Retrofit** e, para a nossa cache, vamos utilizar **Room**.

Abra o arquivo **build.gradle** referente ao módulo **data** e realize a seguinte configuração:

```
apply plugin: 'com.android.library'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-kapt'

android {
    def globalConfiguration = rootProject.extensions.getByName("ext")

    compileSdkVersion globalConfiguration["compileSDK"]

    defaultConfig {
        minSdkVersion globalConfiguration["minSDK"]
        targetSdkVersion globalConfiguration["targetSDK"]
    }
}
```

```
compileOptions {  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}  
}  
  
dependencies {  
    def dependencies = rootProject.ext.dependencies  
  
    implementation project(":domain")  
  
    implementation dependencies.kotlin  
    implementation dependencies.rxJava  
  
    implementation dependencies.retrofit  
    implementation dependencies.retrofitRxAdapter  
    implementation dependencies.retrofitGsonConverter  
    implementation dependencies.gson  
  
    implementation dependencies.room  
    implementation dependencies.roomRxJava  
    kapt dependencies.roomCompiler  
  
    implementation 'com.squareup.okhttp3:okhttp:4.2.1'  
  
    implementation dependencies.koin  
}
```

Código-fonte 3.1.2.2 – Arquivo de dependencias do gradle do módulo data
Fonte: Próprio autor (2020)

3.1.2.2.1 Criando o cache

Comece implementando a cache e deixando-a preparada para salvar nossos dados. No **model** será criada a entidade que será utilizada pelo database.

Para isso, crie um pacote chamado **local**, dentro dele uma pasta chamada **model** e dentro dele uma classe chamada **ProductCache**.

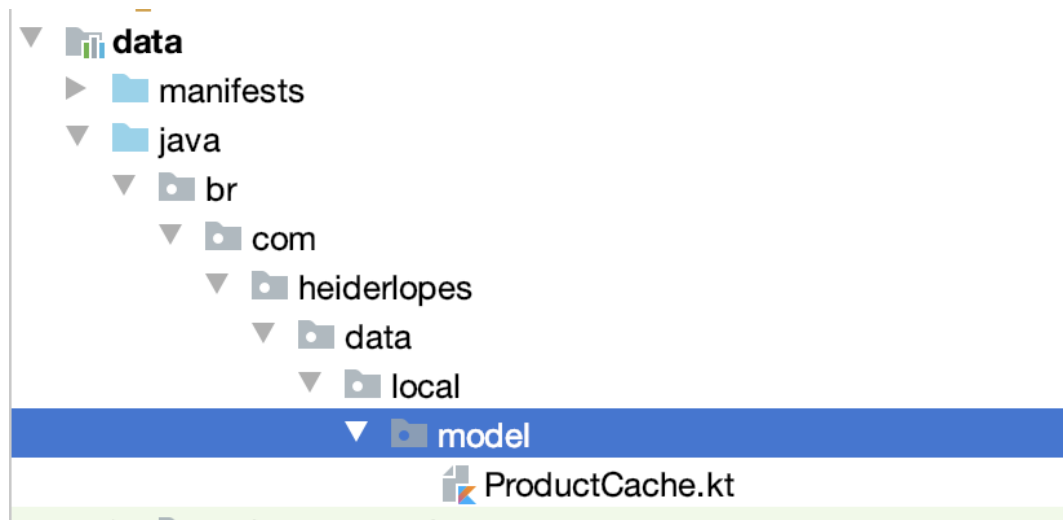


Figura 3.1.2.2.1 – Classe ProductCache
Fonte: Próprio autor (2020)

```
@Entity(tableName = "products")
data class ProductCache(
    @PrimaryKey(autoGenerate = true)
    var id: Int = 0,
    val name: String = "",
    val imageURL: String = "",
    val description: String = ""
)
```

Código-fonte 3.1.2.2.1 – Classe ProductCache
Fonte: Próprio autor (2020)

Crie um pacote **database** e dentro dele adicione dois novos arquivos **ProductsDao** (interface de interação com o banco de dados) e **ProductsDataBase** (classe que irá criar o banco de dados).

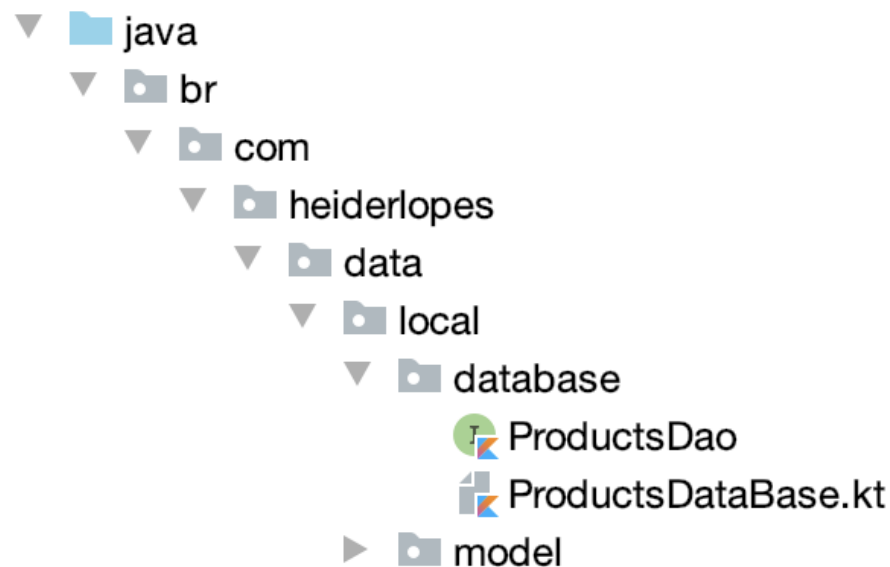


Figura 3.1.2.2.1 – Classes do pacote database

Fonte: Próprio autor (2020)

```

@Dao
interface ProductDao {

    @Query("SELECT * FROM products")
    fun getProducts(): Single<List<ProductCache>>

    @Transaction
    fun updateData(products: List<ProductCache>) {
        deleteAll()
        insertAll(products)
    }

    @Insert
    fun insertAll(products: List<ProductCache>)

    @Query("DELETE FROM products")
    fun deleteAll()
}

```

Código-fonte 3.1.2.2.1 – Classe ProductsDao

Fonte: Próprio autor (2020)

```

@Database(version = 1, entities = [ProductCache::class])
abstract class ProductDataBase: RoomDatabase() {
    abstract fun productDao(): ProductsDao

    companion object {
        fun createDataBase(context: Context): ProductsDao {

```

```

    return Room
        .databaseBuilder(context, ProductDataBase::class.java, "Products.db")
        .build()
        .productDao()
    }
}

```

Código-fonte 3.1.2.2.1 – Classe ProductDataBase
Fonte: Próprio autor (2020)

Feito isso, crie um pacote chamado **mapper** e dentro dele crie uma classe chamada **ProductCacheMapper**. No Mapper será mapeado os dados para salvar na cache e também mapear os dados da cache para serem enviados corretamente.

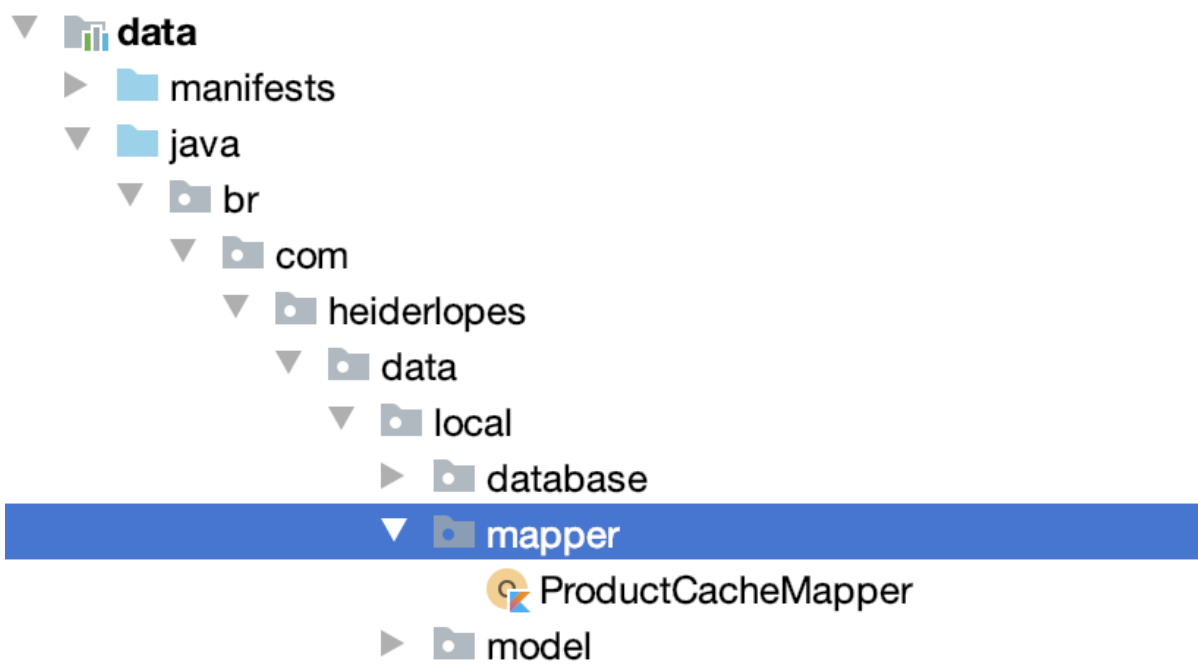


Figura 3.1.2.2.1 – Classe ProductCacheMapper
Fonte: Próprio autor (2020)

```

object ProductCacheMapper {

    fun map(cacheData: List<ProductCache>) = cacheData.map { map(it) }

    private fun map(productCache: ProductCache) = Product(
        name = productCache.name,
        imageURL = productCache.imageURL,

```

```
        description = productCache.description
    )

    fun mapProductToProductCache(products : List<Product>) = products.map {
        map(it) }

    private fun map(product: Product) = ProductCache(
        name = product.name,
        imageURL = product.imageURL,
        description = product.description
    )
}
```

Código-fonte 3.1.2.2.1 – Classe ProductCacheMapper
Fonte: Próprio autor (2020)

Dentro do pacote **local** crie um pacote chamado **datasource** e adicione os seguintes arquivos: **ProductsCacheDataSource** (interface utilizada para que o repository possa solicitar dados da cache). e **ProductsCacheDataSourceImpl** (implementação da interface ProductsCacheDataSource).

```
interface ProductCacheDataSource {

    fun getProducts() : Single<List<Product>>

    fun insertData(products: List<Product>)

    fun updateData(products: List<Product>)

}
```

Código-fonte 3.1.2.2.1 – Interface ProductCacheDataSource
Fonte: Próprio autor (2020)

```
class ProductCacheDataSourceImpl (
    private val productDao: ProductsDao
) : ProductCacheDataSource {
    override fun getProducts(): Single<List<Product>> {
        return productDao.getProducts().map { ProductCacheMapper.map(it) }
    }

    override fun insertData(products: List<Product>) {
        productDao.insertAll(ProductCacheMapper.mapProductToProductCache(products

```

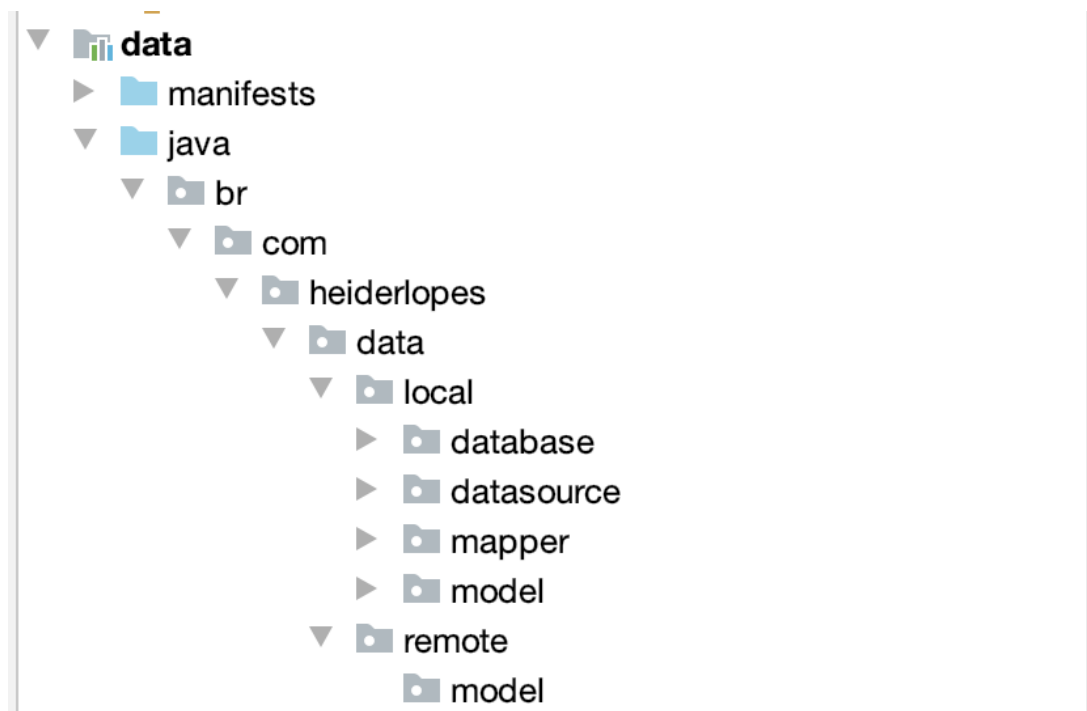
```
))  
}  
  
    override fun updateData(products: List<Product>) {  
  
        productDao.updateData(ProductCacheMapper.mapProductToProductCache(products))  
    }  
}
```

Código-fonte 3.1.2.2.1 – Interface ProductCacheDataSourceImpl
Fonte: Próprio autor (2020)

3.2.2.2.2 Consumindo os dados remotos

Agora será preparada as classes que serão utilizadas para receber os dados do backend.

Crie um pacote **remote** na raiz do **data** e, em seguida, adicione o pacote **model** (dado puro que vem do backend).



Código-fonte 3.1.2.2.2 – Estrutura dos pacotes data com o package remote
Fonte: Próprio autor (2020)

Crie a classe `ProductPayload` dentro de `model` do package `remote`.

```
data class ProductPayload (  
    @SerializedName("nome") val name: String,  
    @SerializedName("urlImagem") val imageURL: String,  
    @SerializedName("descricao") val description: String  
)
```

Código-fonte 3.1.2.2.2 – Classe `ProductPayload`
Fonte: Próprio autor (2020)

Crie um pacote chamado **api** dentro de **remote**, em seguida, crie um arquivo chamado **ProductAPI**.

```
interface ProductAPI {  
    @GET("/v2/5de6d57e3700005f00092640")  
    fun getProducts(): Single<List<ProductPayload>>  
}
```

Código-fonte 3.1.2.2.2 – Interface `ProductAPI`
Fonte: Próprio autor (2020)

Crie um pacote chamado **mapper** dentro do pacote **remote**. Aqui será mapeado os dados puros do backend, nossos payloads, em **Product**, que estão sendo pedidos pela **domain**.

```
object ProductPayloadMapper {  
  
    fun map(products: List<ProductPayload>) = products.map { map(it) }  
  
    private fun map(productPayload: ProductPayload) = Product(  
        name = productPayload.name,  
        imageURL = productPayload.imageURL,  
        description = productPayload.description  
    )  
}
```

Código-fonte 3.1.2.2.2 – Classe `ProductPayloadMapper`
Fonte: Próprio autor (2020)

Crie um pacote chamado **source** e dentro dele os seguintes arquivos: **RemoteDataSource**, **RemoteDataSourceImpl**.

```
interface ProductRemoteDataSource {  
    fun getProducts() : Single<List<Product>>  
}
```

Código-fonte 3.1.2.2.2 – Classe ProductRemoteDataSource
Fonte: Próprio autor (2020)

```
class ProductRemoteDataSourceImpl(private val productAPI: ProductAPI) :  
    ProductRemoteDataSource {  
    override fun getProducts(): Single<List<Product>> {  
        return productAPI.getProducts().map { ProductPayloadMapper.map(it) }  
    }  
}
```

Código-fonte 3.1.2.2.2 – Classe ProductRemoteDataSourceImpl
Fonte: Próprio autor (2020)

Dentro da implementação do método **getProducts**, será chamado o endpoint da API para solicitar os dados do backend, e, quando o chegar os dados eles serão mapeados do payload para o dado que foi solicitado. A **productAPI** é injetada no construtor.

3.1.2.2.3 Criando o repository

Nesta classe será realizada a implementação do **ProductRepository** (criada na domain) dentro do módulo data.

Crie um arquivo chamado **ProductRepositoryImpl** dentro do pacote **data/repository**, e adicione o seguinte código:

```
class ProductRepositoryImpl (  
    private val productsCacheDataSource: ProductCacheDataSource,  
    private val productRemoteDataSource: ProductRemoteDataSource  
) : ProductRepository {
```

```

override fun getProducts(forceUpdate: Boolean): Single<List<Product>> {
    return if (forceUpdate)
        getProductsRemote(forceUpdate)
    else
        productsCacheDataSource.getProducts()
            .flatMap { listJobs ->
                when {
                    listJobs.isEmpty() -> getProductsRemote(false)
                    else -> Single.just(listJobs)
                }
            }
}

private fun getProductsRemote(isUpdate: Boolean): Single<List<Product>> {
    return productRemoteDataSource.getProducts()
        .flatMap { listJobs ->
            if (isUpdate)
                productsCacheDataSource.updateData(listJobs)
            else
                productsCacheDataSource.insertData(listJobs)
            Single.just(listJobs)
        }
}

```

Código-fonte 3.1.2.2.2 – Classe ProductRepositoryImpl
 Fonte: Próprio autor (2020)

Crie um pacote chamado **di** no pacote **data**, em seguida, adicione os seguintes arquivos: **DataCacheModule.kt**, **DataRemoteModule.kt**, **DataModule.kt**. Abaixo os seus respectivos códigos:

```

val cacheDataModule = module {
    single { ProductDataBase.createDataBase(androidContext()) }
    factory<ProductCacheDataSource> { ProductCacheDataSourceImpl(productDao
    = get()) }
}

```

Código-fonte 3.1.2.2.2 – Classe DataCacheModule
 Fonte: Próprio autor (2020)

```

val remoteDataSourceModule = module {
    factory { providesOkHttpClient() }
    single {

```

```
        createWebService<ProductAPI>(  
            okHttpClient = get(),  
            url = "http://www.mocky.io"  
        )  
    }  
  
    factory<ProductRemoteDataSource> {  
        ProductRemoteDataSourceImpl(productAPI = get())  
    }  
}  
  
fun providesOkHttpClient(): OkHttpClient {  
    return OkHttpClient.Builder()  
        .connectTimeout(30, TimeUnit.SECONDS)  
        .readTimeout(30, TimeUnit.SECONDS)  
        .writeTimeout(30, TimeUnit.SECONDS)  
        .build()  
}  
  
inline fun <reified T> createWebService(  
    okHttpClient: OkHttpClient,  
    url: String  
): T {  
    return Retrofit.Builder()  
        .addConverterFactory(GsonConverterFactory.create())  
        .addCallAdapterFactory(RxJava2CallAdapterFactory.create())  
        .baseUrl(url)  
        .client(okHttpClient)  
        .build()  
        .create(T::class.java)  
    }  
}
```

Código-fonte 3.1.2.2.2 – Classe DataRemoteModule
Fonte: Próprio autor (2020)

```
val repositoryModule = module {  
    factory<ProductRepository> {  
        ProductRepositoryImpl(  
            productsCacheDataSource = get(),  
            productRemoteDataSource = get()  
        )  
    }  
}  
  
val dataModules = listOf(remoteDataSourceModule, repositoryModule,  
    cacheDataModule)
```

Código-fonte 3.1.2.2.2 – Classe DataModule
Fonte: Próprio autor (2020)

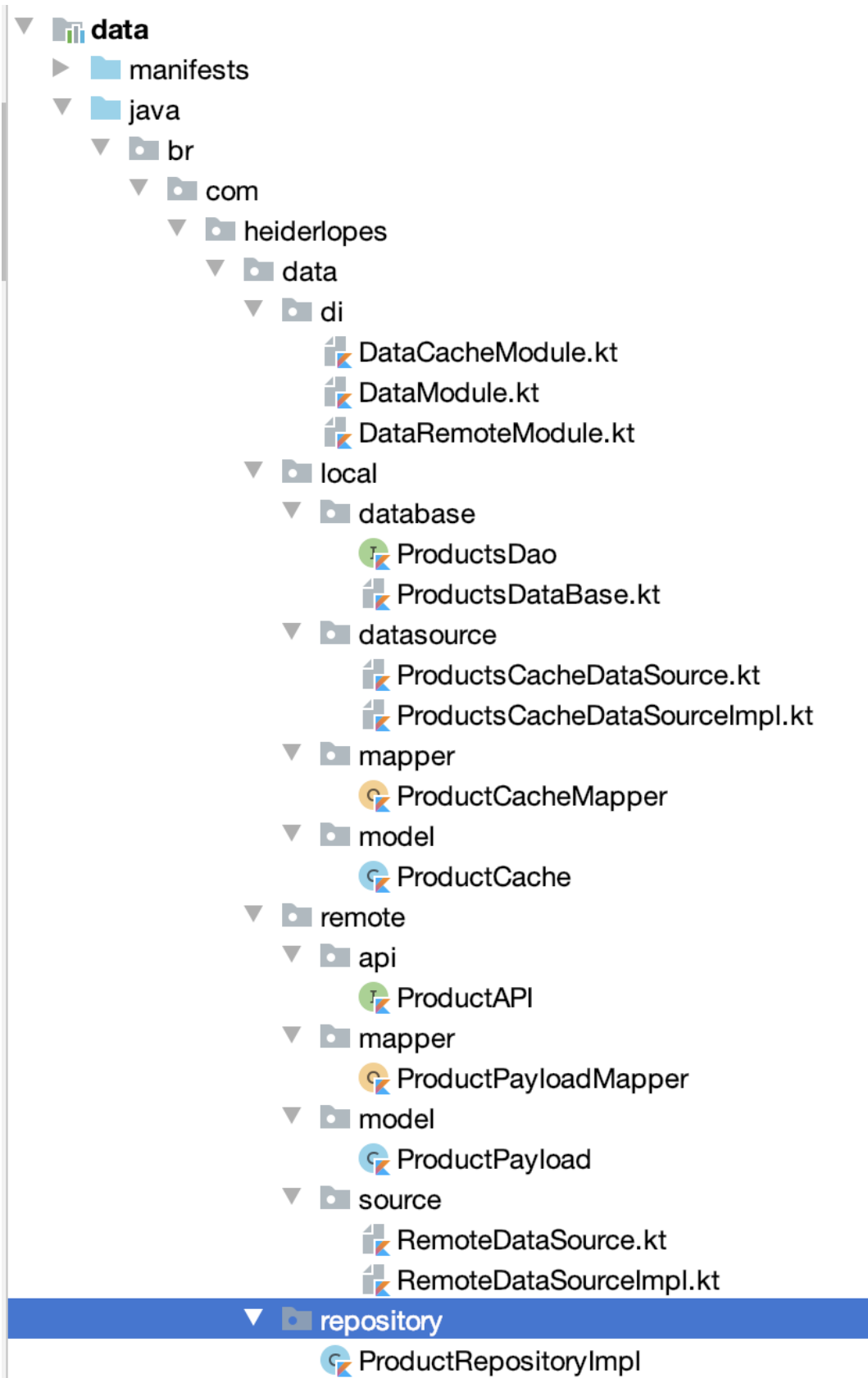


Figura 3.1.2.2.2 – Estrutura de pacote e classes do data module
Fonte: Próprio autor (2020)

3.2.3 Presentation Module

Este módulo é composto por:

View: que são as Activities/Fragments, onde serão apresentados os dados.

ViewModel: é onde serão gerenciados os dados relacionados às nossas Views, ou seja, chamar nosso repositório para consumir dados ou observar dados.

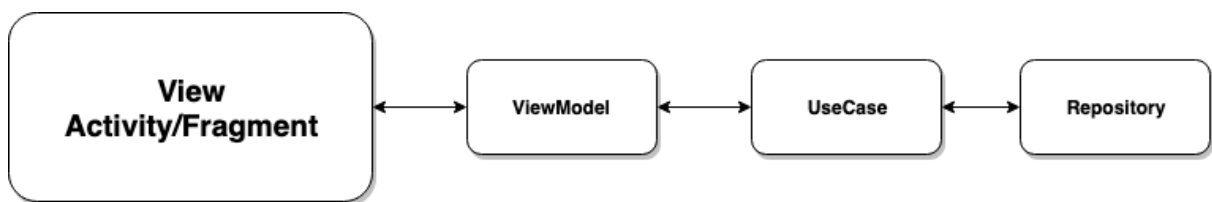


Figura 3.1.2.2.2 – Classe DataModule
Fonte: Próprio autor (2020)

A view solicita o produto, para o ViewModel, que utiliza o repository para buscar os dados a serem apresentados.

No projeto foi criado no início o module **app**, e é ele que será utilizado como presentation module.

Abra o arquivo **build.gradle** do módulo **app**, e adicione à seguinte configuração:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'

android {

    def globalConfiguration = rootProject.extensions.getByName("ext")

    compileSdkVersion globalConfiguration["compileSDK"]
    defaultConfig {
```

```
    applicationId "br.com.heiderlopes.ondeeh"
    minSdkVersion globalConfiguration["minSDK"]
    targetSdkVersion globalConfiguration["targetSDK"]
    versionCode 1
    versionName "1.0"
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

dataBinding {
    enabled true
}

buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
'proguard-rules.pro'
    }
}

dependencies {
    implementation project(path: ':domain')
    implementation project(path: ':data')

    def dependencies = rootProject.ext.dependencies
    def testDependencies = rootProject.ext.testDependencies

    implementation dependencies.appcompat
    implementation dependencies.constraintlayout

    testImplementation testDependencies.junit
    androidTestImplementation testDependencies.runner
    androidTestImplementation testDependencies.espresso

    implementation dependencies.cardview
    implementation dependencies.recyclerview

    implementation dependencies.kotlin

    implementation dependencies.ktx

    implementation dependencies.viewmodel

    implementation dependencies.lifecycle

    implementation dependencies.koin
    implementation dependencies.koinViewModel
}
```

```
implementation dependencies.rxJava
implementation dependencies.rxKotlin
implementation dependencies.rxAndroid

kapt dependencies.dataBinding

}
```

Figura 3.1.3 – build.gradle do presentation module
Fonte: Próprio autor (2020)

Crie os seguintes pacotes **extension** e **di**:

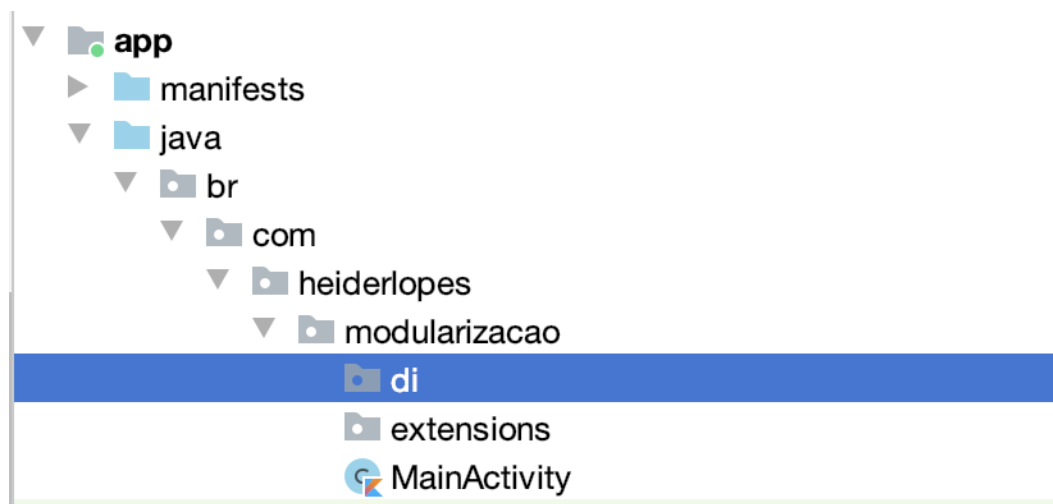


Figura 3.1.3 – Pacote di e extensions do presentation module
Fonte: Próprio autor (2020)

Dentro do pacote extensions crie as seguintes classes: **Context.kt**, **View.kt** e **ViewGroup.kt**. Segue abaixo os respectivos códigos:

```
fun Context.toast(message: CharSequence, duration: Int =
    Toast.LENGTH_SHORT) {
    Toast.makeText(this, message, duration).show()
}
```

Figura 3.1.3 – Classe de extensão Context.kt
Fonte: Próprio autor (2020)

```
fun View.visible(visible: Boolean = false) {
    visibility = if (visible) View.VISIBLE else View.GONE
}
```

Figura 3.1.3 – Classe de extensão View.kt
Fonte: Próprio autor (2020)

```
fun ViewGroup.inflate(layoutId: Int, attachToRoot: Boolean = false): View {
    return LayoutInflater.from(context).inflate(layoutId, this, attachToRoot)
}
```

Figura 3.1.3 – Classe de extensão ViewGroup.kt
Fonte: Próprio autor (2020)

Crie um pacote chamado **viewmodel** e adicione os seguintes arquivos: **BaseViewModel** e **ViewState**.

```
open class BaseViewModel: ViewModel() {

    val disposables = CompositeDisposable()

    override fun onCleared() {
        disposables.clear()

        super.onCleared()
    }
}
```

Figura 3.1.3 – Classe BaseViewModel
Fonte: Próprio autor (2020)

```
sealed class ViewState<out T> {
    object Loading : ViewState<Nothing>()
    data class Success<T>(val data: T) : ViewState<T>()
    data class Failed(val throwable: Throwable) : ViewState<Nothing>()
}

class StateMachineSingle<T>: SingleTransformer<T, ViewState<T>> {

    override fun apply(upstream: Single<T>): SingleSource<ViewState<T>> {
        return upstream
            .map {
                ViewState.Success(it) as ViewState<T>
            }
    }
}
```

```
    }  
    .onErrorReturn {  
        ViewState.Failed(it)  
    }  
    .doOnSubscribe {  
        ViewState.Loading  
    }  
}  
}
```

Figura 3.1.3 – Classe ViewState
Fonte: Próprio autor (2020)

Viewmodel: onde está a **BaseViewModel** que todas as classes do tipo viewmodel irão estender para evitarmos duplicação de códigos comuns. A **StateMachine** vai gerenciar os estados das chamadas do repository dentro do viewmodel.

BaseViewModel: estende a classe ViewModel e também é onde encontra-se um disposables, que é um gerenciador do ciclo de vida dos Observables, os quais são as chamadas do useCase.

No método **onCleared()**, que é chamado quando o **ViewModel** morre, todos os **Observables** armazenados no nosso disposables serão eliminados. É importante fazer isso porque, se não for dado um fim nos observables, eles podem ficar rodando e ocasionar algum leak de memória. O CompositeDisposable() é do que um container de disposables.

ViewState: um gerenciador de estados utilizado para mostrar o estado certo na view, de acordo com o que for emitido. Serão representados os 3 estados possíveis:

Loading: esse estado é emitido no .doOnSubscribe, para mostrar o loader assim que a stream começar. Nesse estado não se precisa de nenhum dado para ser emitido, pois na criação foi usado ViewState<Nothing>.

Success: esse estado é emitido no `.map`, que aplica algo específico no item emitido. Nesse caso, emite o Success juntamente com o dado da stream (lista de produtos).

Failed: quando acontecer algo de errado na stream, vai ser emitido o estado Failed no `onErrorReturn`, junto com o `throwable`(erro) emitido.

StateMachineSingle: aplicada nas chamadas do tipo **Single**, do repository, para nos emitir os estados que mencionados acima.

Crie uma classe chamada **MainViewModel** na raiz do projeto e adicione o seguinte código:

```
class MainViewModel(
    val useCase: GetProductsUseCase,
    val uiScheduler: Scheduler
): BaseViewModel() {

    val state = MutableLiveData<ViewState<List<Product>>>().apply {
        value = ViewState.Loading
    }

    fun getProducts(forceUpdate: Boolean = false) {
        disposables += useCase.execute(forceUpdate = forceUpdate)
            .compose(StateMachineSingle())
            .observeOn(uiScheduler)
            .subscribe(
                {
                    //onSuccess
                    state.postValue(it)
                },
                {
                    //onError
                }
            )
    }
}
```

Figura 3.1.3 – Classe ViewState
Fonte: Próprio autor (2020)

O **LiveData** da **StateMachine** terá inicialmente o estado de **Loading**.

O primeiro método **getProducts** é o local onde será chamado o **useCase** para nos fornecer a lista de produtos. O **useCase.execute**, executa tudo o que já foi criado anteriormente para buscar os dados.

O **compose** adicionado com o **StateMachineSingle()**, vai aplicar alguma função de transformação. No caso, será emitir os **StateMachines** na chamada do **useCase**.

Adicione o **uiScheduler**, no **observerOn**, ou seja, será observado na **ui thread**. Por fim, o resultado final que acontecerá no **subscribe**, ou seja, assim que o chegar os dados, será setado seu valor no **LiveData**. A segunda chave está vazia, pois na **StateMachineSingle** já foi retornado o estado de erro.

Os estados das duas funções abertas no **subscribe** são: a primeira é **onSuccess**, e a segunda é **onError**.

3.1.3.1 Programando a classe principal

Agora abra o arquivo **MainActivity.kt**, em seguida, adicione o seguinte código:

```
class MainActivity : AppCompatActivity() {  
  
    private val viewModel: MainViewModel by viewModel()  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        setupViewModel()  
    }  
  
    private fun setupViewModel() {  
        viewModel.getProducts()  
    }  
}
```



```

viewModel.state.observe(this, Observer { state ->
    when(state) {
        is ViewState.Success -> {
            Log.i("TAG", "Sucesso")
        }
        is ViewState.Loading -> {
            Log.i("TAG", "Loading")
        }
        is ViewState.Failed -> {
            Log.i("TAG", "Failed")
        }
    }
})
}
}

```

Figura 3.1.3.1 – Classe MainActivity
Fonte: Próprio autor (2020)

Primeiro, injete o **viewModel**. Em seguida, crie o método **setupViewModel**, onde será chamada a **viewModel** para nos fornecer os dados, e, em seguida, o **LiveDataObserver**, para cada **state**.

Success: será configurada a lista no adapter e aqui o recyclerView ficará visível, e o resto invisível.

Loading: será mostrado um progressBar, e escondido o resto.

Failed: mostra um botão para tentar novamente, para caso algo dê errado, e esconde as outras views.

3.1.3.2 Di da presentation

Crie um pacote **di** e dentro dele o arquivo **PresentationModule.kt**:

```

val presentationModule = module {
    viewModel { MainViewModel(

```

```
        useCase = get(),  
        uiScheduler = AndroidSchedulers.mainThread()  
    )  
}  
}
```

Figura 3.1.3.2 – Classe Presentation Module
Fonte: Próprio autor (2020)

3.1.3.3 Inicializando o Koin

Crie um arquivo chamado **MyApplication.kt** na raiz do módulo **app** e adicione o seguinte código:

```
class MyApplication: Application() {  
  
    override fun onCreate() {  
        super.onCreate()  
  
        startKoin {  
            androidContext(this@MyApplication)  
  
            modules(domainModule + dataModules + listOf(presentationModule))  
        }  
    }  
}
```

Figura 3.1.3.3 – Classe MyApplication
Fonte: Próprio autor (2020)

Chame o **startkoin** e, primeiro, deverá ser provido o contexto e, em seguida, todos os **koin Modules** criados no projeto.

Em seguida, abra o arquivo **AndroidManifest.xml** e adicione as seguintes linhas em negrito:

```
<uses-permission android:name="android.permission.INTERNET"/>  
  
<application  
    android:name=".MyApplication"  
    android:allowBackup="true"  
    android:icon="@mipmap/ic_launcher"
```

```

        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
<activity android:name=".MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>
    </intent-filter>
</activity>
</application>

```

Figura 3.1.3.3 – AndroidManifest.xml
Fonte: Próprio autor (2020)

3.1.3.4 Melhorando o visual das linhas da lista

Na pasta layout, crie um arquivo chamado **product_row.xml** e crie o seguinte layout:



Figura 3.1.3.4 – Item da lista de produtos
Fonte: Próprio autor (2020)

```

<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_margin="8dp">

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

```

```
        android:orientation="horizontal"
        android:padding="16dp">

        <ImageView
            android:contentDescription="Foto do produto"
            android:id="@+id/ivPhotoProduct"
            android:layout_width="96dp"
            android:layout_height="100dp"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:src="@mipmap/ic_launcher" />

        <TextView
            android:id="@+id/tvTitleProduct"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginStart="8dp"
            android:layout_marginEnd="8dp"
            android:textSize="22sp"
            android:textStyle="bold"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toEndOf="@+id/ivPhotoProduct"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="Nome do Produto" />

        <TextView
            android:id="@+id/tvDescriptionProduct"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:ellipsize="end"
            android:maxLines="3"
            app:layout_constraintEnd_toEndOf="@+id/tvTitleProduct"
            app:layout_constraintStart_toStartOf="@+id/tvTitleProduct"
            app:layout_constraintTop_toBottomOf="@+id/tvTitleProduct"
            tools:text="Alguma coisa falando sobre esse produto ...blablabla" />

    </androidx.constraintlayout.widget.ConstraintLayout>

</androidx.cardview.widget.CardView>
```

Código-fonte 3.1.3.4 – Item produtos da lista
Fonte: Próprio autor (2020)

Como a imagem será exibida através da url que está na internet, adicione a dependência da biblioteca **Picasso**. Abra o arquivo **dependencies.gradle** e adicione as seguintes linhas em negrito:

```
ext {  
  
    minSDK = 20  
    targetSDK = 28  
    compileSDK = 28  
  
    buildTools = '3.4.1'  
  
    appCompactVersion = '1.0.2'  
    kotlinVersion = '1.3.21'  
  
    AndroidArchVersion = '1.1.1'  
    databindingVersion = '3.1.4'  
    lifecycleVersion = '2.0.0'  
    ktxVersion = '1.0.1'  
  
    constrainVersion = '1.1.3'  
    cardViewVersion = '1.0.0'  
    recyclerViewVersion = '1.0.0'  
  
    //Rx  
    rxJavaVersion = '2.2.7'  
    rxKotlinVersion = '2.4.0'  
    rxAndroidVersion = '2.1.1'  
  
    //Koin  
    koinVersion = '2.0.1'  
  
    //Retrofit  
    retrofitVersion = '2.3.0'  
  
    //Okhttp  
    okhttpVersion = '3.2.0'  
  
    //Gson  
    gsonVersion = '2.8.5'  
  
    //Room version  
    roomVersion = '2.1.0'  
  
    //Picasso version  
    picassoVersion = '2.71828'  
    //Test  
    junitVersion = '4.12'  
    espressoVersion = '3.1.1'  
    runnerVersion = '1.1.1'
```

```
dependencies = [  
    kotlin: "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlinVersion",  
  
    appCompact: "androidx.appcompat:appcompat:$appCompactVersion",  
    constraintlayout:  
"androidx.constraintlayout:constraintlayout:$constrainVersion",  
    cardView: "androidx.cardview:cardview:$cardViewVersion",  
    recyclerView: "androidx.recyclerview:recyclerview:$recyclerViewVersion",  
  
    viewModel: "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifeCycleVersion",  
    lifecycle: "android.arch.lifecycle:extensions:$AndroidArchVersion",  
  
    dataBinding: "com.android.databinding:compiler:$databindingVersion",  
  
    ktx: "androidx.core:core-ktx:$ktxVersion",  
  
    rxJava: "io.reactivex.rxjava2:rxjava:$rxJavaVersion",  
    rxKotlin: "io.reactivex.rxjava2:rxkotlin:$rxKotlinVersion",  
    rxAndroid: "io.reactivex.rxjava2:rxandroid:$rxAndroidVersion",  
  
    koin: "org.koin:koin-android:$koinVersion",  
    koinViewModel: "org.koin:koin-androidx-viewmodel:$koinVersion",  
  
    retrofit: "com.squareup.retrofit2:retrofit:$retrofitVersion",  
    retrofitRxAdapter: "com.squareup.retrofit2:adapter-rxjava2:$retrofitVersion",  
    retrofitGsonConverter:  
"com.squareup.retrofit2:converter-gson:$retrofitVersion",  
    gson: "com.google.code.gson:gson:$gsonVersion",  
  
    room: "androidx.room:room-runtime:$roomVersion",  
    roomRxJava: "androidx.room:room-rxjava2:$roomVersion",  
    roomCompiler: "androidx.room:room-compiler:$roomVersion",  
  
    picasso: "com.squareup.picasso:picasso:$picassoVersion"  
  
]  
  
testDependencies = [  
    junit: "junit:junit:$junitVersion",  
    espresso: "androidx.test.espresso:espresso-core:$espressoVersion",  
    runner: "androidx.test:runner:$runnerVersion"  
]  
}
```

Abra o arquivo **build.gradle** referente ao **app** e adicione à dependência criada anteriormente (linha negrito abaixo:)

```
dependencies {  
    implementation project(path: ':domain')  
    implementation project(path: ':data')  
  
    def dependencies = rootProject.ext.dependencies  
    def testDependencies = rootProject.ext.testDependencies  
  
    implementation dependencies.appcompat  
    implementation dependencies.constraintlayout  
  
    testImplementation testDependencies.junit  
    androidTestImplementation testDependencies.runner  
    androidTestImplementation testDependencies.espresso  
  
    implementation dependencies.cardView  
    implementation dependencies.recyclerView  
  
    implementation dependencies.kotlin  
  
    implementation dependencies.ktx  
  
    implementation dependencies.viewModel  
  
    implementation dependencies.lifecycle  
  
    implementation dependencies.koin  
    implementation dependencies.koinViewModel  
  
    implementation dependencies.rxJava  
    implementation dependencies.rxKotlin  
    implementation dependencies.rxAndroid  
  
    implementation dependencies.picasso  
  
    kapt dependencies.databinding  
}
```

Dentro da raiz do projeto, crie um arquivo chamado **MainListAdapter.kt** e adicione o seguinte código:

```
class MainListAdapter(  
    private val picasso: Picasso  
) : RecyclerView.Adapter<MainListAdapter.ViewHolder>() {  
  
    var products: List<Product> = listOf()  
  
    inner class ViewHolder(parent: ViewGroup) :  
        RecyclerView.ViewHolder(parent.inflate(R.layout.product_row)) {  
  
        fun bind(product: Product) = with(itemView) {  
            tvTitleProduct.text = product.name  
            tvDescriptionProduct.text = product.description  
            picasso.load(product.imageURL).into(ivPhotoProduct)  
        }  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):  
        ViewHolder =  
            ViewHolder(parent)  
  
    override fun getItemCount(): Int = products.size  
    override fun onBindViewHolder(holder: ViewHolder, position: Int) =  
        holder.bind(products[position])  
}
```

Código-fonte 3.1.3.4 – Implementação do MainListAdapter
Fonte: Próprio autor (2020)

Abra o arquivo **PresentationModule.kt** e adicione o método responsável por gerar o nosso **adapter**.

```
val presentationModule = module {  
  
    single { Picasso.get() }  
  
    factory { MainListAdapter(picasso = get()) }  
  
    viewModel {  
        MainViewModel(  
            useCase = get(),  
            uiScheduler = AndroidSchedulers.mainThread()  
        )  
    }  
}
```



```
)  
}  
}
```

Código-fonte 3.1.3.4 – Injetando o adapter
Fonte: Próprio autor (2020)

Altere o layout da **MainActivity.kt** para exibir a lista e para que possa exibir os loadings necessários:

```
<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
  xmlns:app="http://schemas.android.com/apk/res-auto">  
  <data>  
    <import type="android.view.View" />  
  
    <variable  
      name="viewModel"  
      type="br.com.heiderlopes.modularizacao.MainViewModel" />  
  
  </data>  
  
  <androidx.constraintlayout.widget.ConstraintLayout  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
    <androidx.recyclerview.widget.RecyclerView  
      android:id="@+id/recyclerView"  
      android:layout_width="match_parent"  
      android:layout_height="0dp"  
      app:layout_constraintBottom_toBottomOf="parent"  
      app:layout_constraintEnd_toEndOf="parent"  
      app:layout_constraintStart_toStartOf="parent"  
      app:layout_constraintTop_toTopOf="parent" />  
  
    <ProgressBar  
      android:id="@+id/progressBar"  
      app:layout_constraintTop_toTopOf="parent"  
      style="?android:attr/progressBarStyle"  
      android:layout_width="wrap_content"  
      android:layout_height="wrap_content"  
      android:layout_marginStart="8dp"  
      android:layout_marginTop="8dp"  
      android:layout_marginEnd="8dp"  
      android:layout_marginBottom="8dp"
```

```
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
    />

    <TextView
        android:id="@+id/tvMessage"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        android:text="Nenhum item encontrado"
        android:visibility="visible"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="@+id/recyclerView" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Código-fonte 3.1.3.4 – Layout da MainActivity
Fonte: Próprio autor (2020)

Como nesse exemplo, está sendo utilizado o **DataBinding**, abra o arquivo **build.gradle** e adicione a seguinte linha em negrito:

```
android {
    compileSdkVersion 29
    buildToolsVersion "29.0.0"
    defaultConfig {
        applicationId "com.example.modularizacao"
        minSdkVersion 20
        targetSdkVersion 29
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    dataBinding {
        enabled true
    }

    buildTypes {
```

```
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),
            'proguard-rules.pro'
        }
    }
}
```

Código-fonte 3.1.3.4 – Habilitando o databinding
Fonte: Próprio autor (2020)

Feito isso, clique em **Build**, em seguida, **Rebuild Project**.

Após o rebuild, altere a MainActivity.

```
class MainActivity : AppCompatActivity() {

    private val viewModel: MainViewModel by viewModel()
    private val mainListAdapter: MainListAdapter by inject()

    private lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        binding = DataBindingUtil.setContentView(this, R.layout.activity_main)
        binding.viewModel = viewModel
        binding.lifecycleOwner = this

        setupRecyclerView()
        setupViewModel()
    }

    private fun setupViewModel() {
        viewModel.getProducts()

        viewModel.state.observe(this, Observer { state ->
            when (state) {
                is ViewState.Success -> {
                    mainListAdapter.products = state.data
                    setVisibilities(showList = true)
                }
                is ViewState.Loading -> {
                    setVisibilities(showProgressBar = true)
                }
                is ViewState.Failed -> {
                    binding.tvMessage.text = state.throwable.message
                }
            }
        })
    }
}
```

```
        setVisibilities(showMessage = true)
    }
}

}

}

private fun setupRecyclerView() = with(binding.recyclerView) {
    layoutManager = LinearLayoutManager(context)
    adapter = mainListAdapter
}

private fun setVisibilities(
    showProgressBar: Boolean = false,
    showList: Boolean = false,
    showMessage: Boolean = false
) {
    binding.progressBar.visible(showProgressBar)
    binding.recyclerView.visible(showList)
    binding.tvMessage.visible(showMessage)
}
}
```

Código-fonte 3.1.3.4 – MainActivity para exibir os dados
Fonte: Próprio autor (2020)

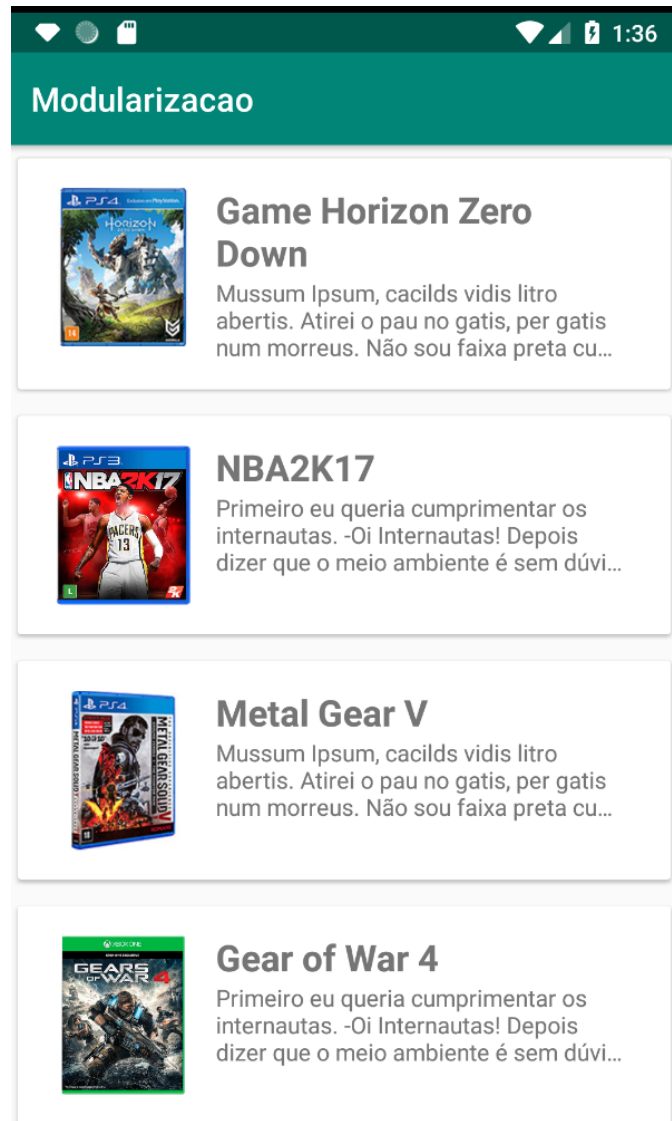


Figura 3.1.3.4 – Exibição dos dados no aplicativo
Fonte: Próprio autor (2020)

3.2 Modularização de recursos (features)

Existem dois tipos de módulos no android: módulos de biblioteca (abordados no módulo referente ao Gradle) e módulos de recursos dinâmicos.

Módulos de biblioteca: são incorporados ao nosso aplicativo e são essenciais. São módulos comuns que conhecemos e usamos em nossos aplicativos. O módulo do aplicativo dependerá dos módulos da biblioteca e eles fornecem

funcionalidades essenciais, por exemplo, bibliotecas para gerenciamento de requisições, manipulação de imagens, animações entre outros.

Módulos dinâmicos: são módulos que podem ser instalados sob demanda e não devem incluir nenhum recurso básico. Nesses módulos os usuários têm a opção de removê-los mais tarde ou instalar recursos dinâmicos, se quiserem. A principal limitação dos módulos de recursos dinâmicos é que o módulo de aplicativos não pode depender dos módulos de recursos dinâmicos.

3.2.1 Criando módulos dinâmicos

Crie um novo pacote chamado **feature**, e dentro dele um pacote chamado de **listproducts**.

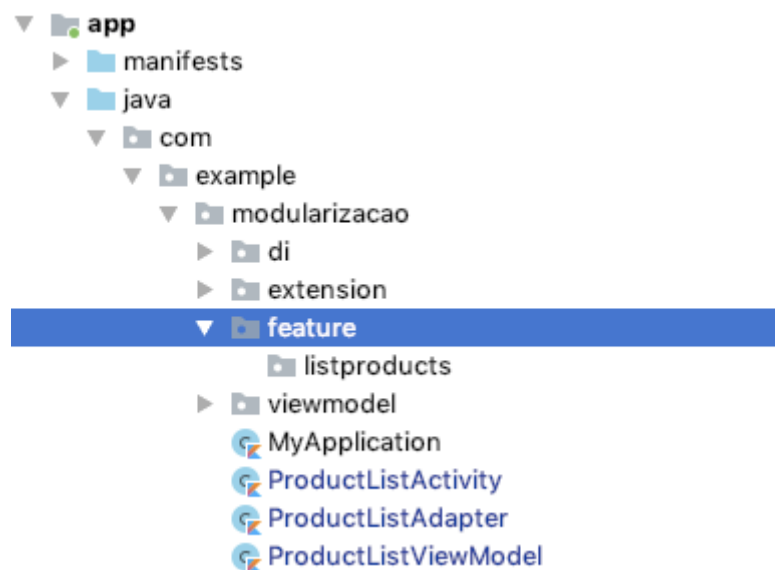


Figura 3.2.1 – Estrutura de pacotes com a modularização por funcionalidade
Fonte: Próprio autor (2020)

Altere os arquivos criados para que remetam o que realmente são (caso necessário altere os nomes das variáveis):

MainActivity ⇒ **ProductListActivity**

MainViewModel => **ProductListViewModel**

MainListAdapter ⇒ ProductListAdapter

activity_main.xml ⇒ activity_product_list

Após isso, clique em **Build ⇒ Rebuild project**

Mova os arquivos renomeados para dentro dessa nova pasta:

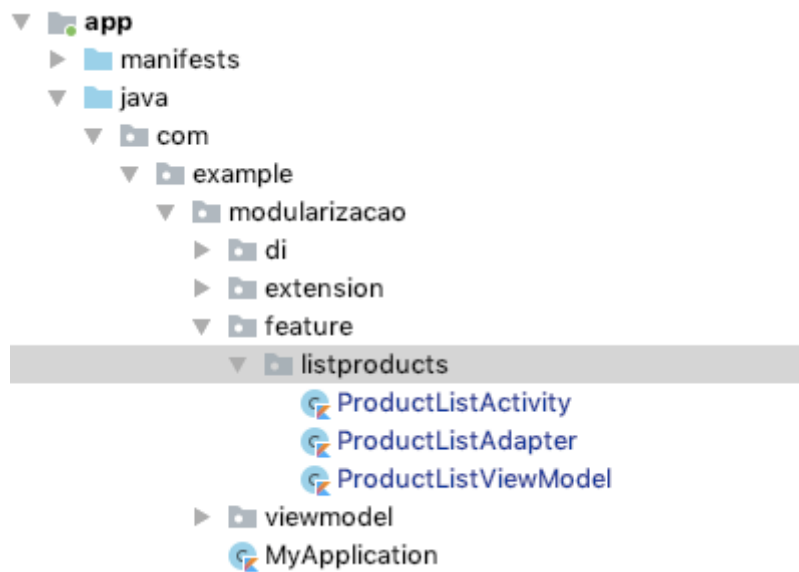


Figura 3.2.1 – Estrutura de pacotes com a modularização por funcionalidade com as classes
Fonte: Próprio autor (2020)

Crie um novo pacote chamado de **main** dentro do pacote **feature**. Dentro deste pacote crie uma **Empty Activity** chamada **MainActivity**.

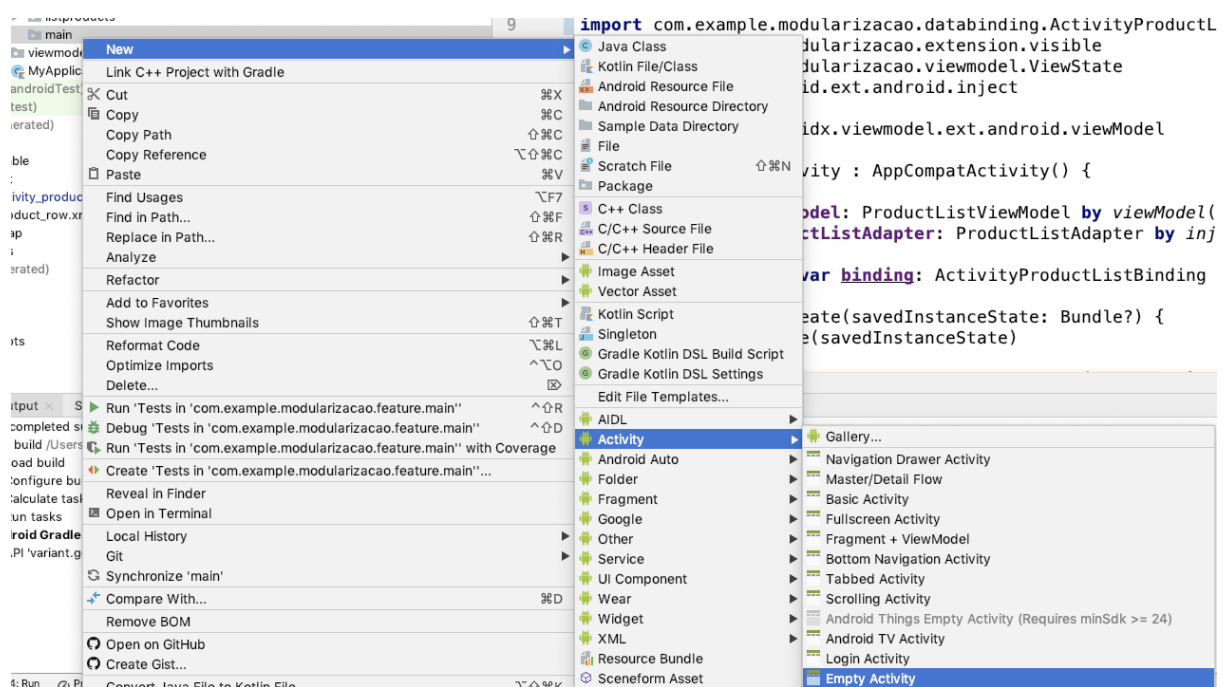


Figura 3.2.1 –Criando uma Empty Activity
Fonte: Próprio autor (2020)

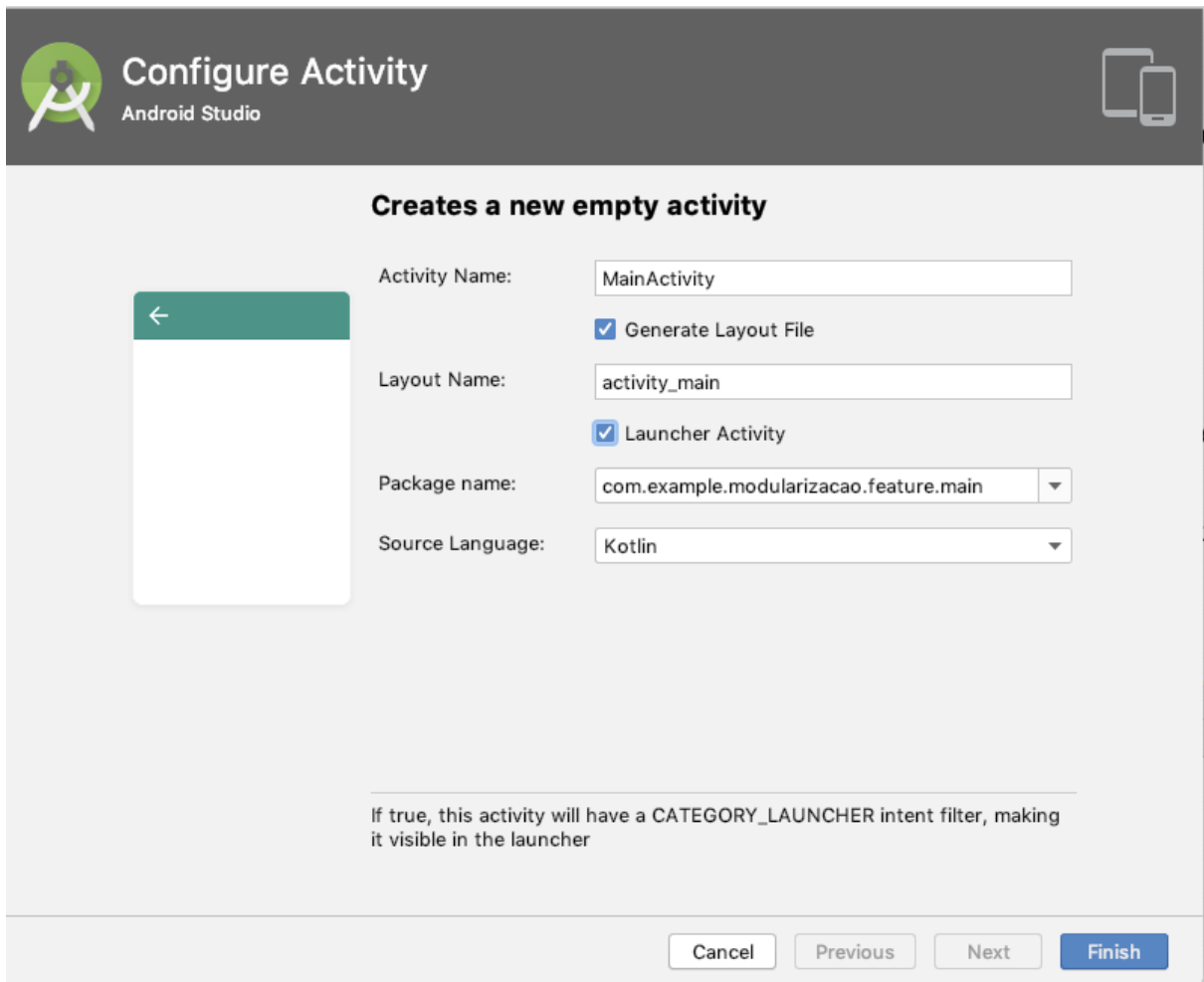


Figura 3.2.1 –Definindo o nome da EmptyActivity como MainActivity
Fonte: Próprio autor (2020)

Abra o arquivo **AndroidManifest.xml** e certifique-se que somente à **MainActivity** contém o intent-filter relacionado à abertura do aplicativo.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.modularizacao">

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:name=".MyApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
```



```

        android:supportRtl="true"
        android:theme="@style/AppTheme">
<activity android:name=".feature.main.MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity android:name=".feature.listproducts.ProductListActivity">
</activity>
</application>

</manifest>

```

Figura 3.2.1 – Definindo a MainActivity como a principal
Fonte: Próprio autor (2020)

Agora crie um novo módulo:

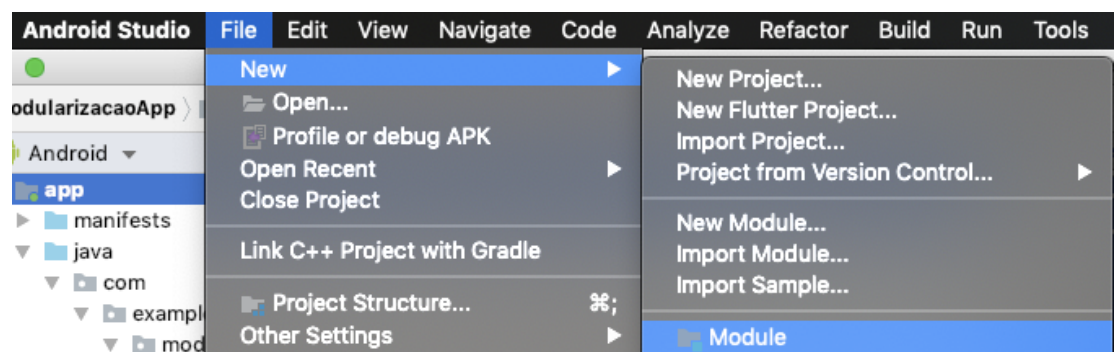


Figura 3.2.1 – Criando um novo módulo
Fonte: Próprio autor (2020)

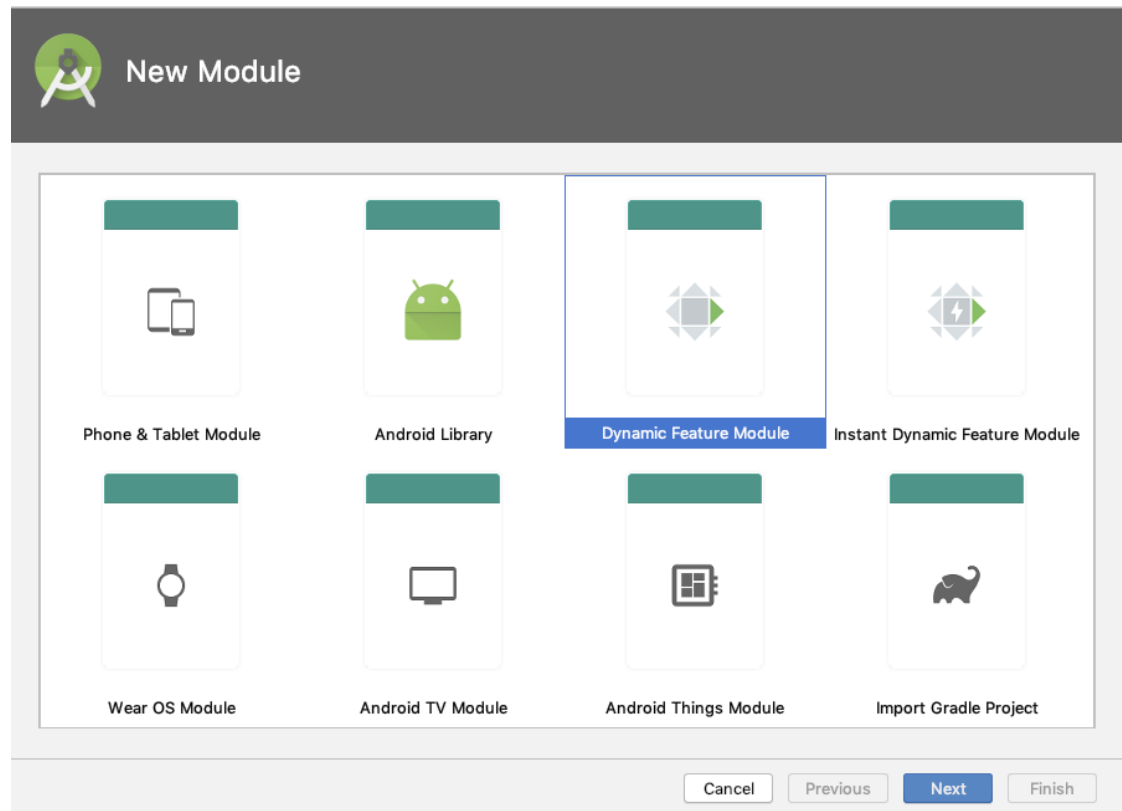


Figura 3.2.1 – Definindo o módulo com Dynamic Feature Module

Fonte: Próprio autor (2020)

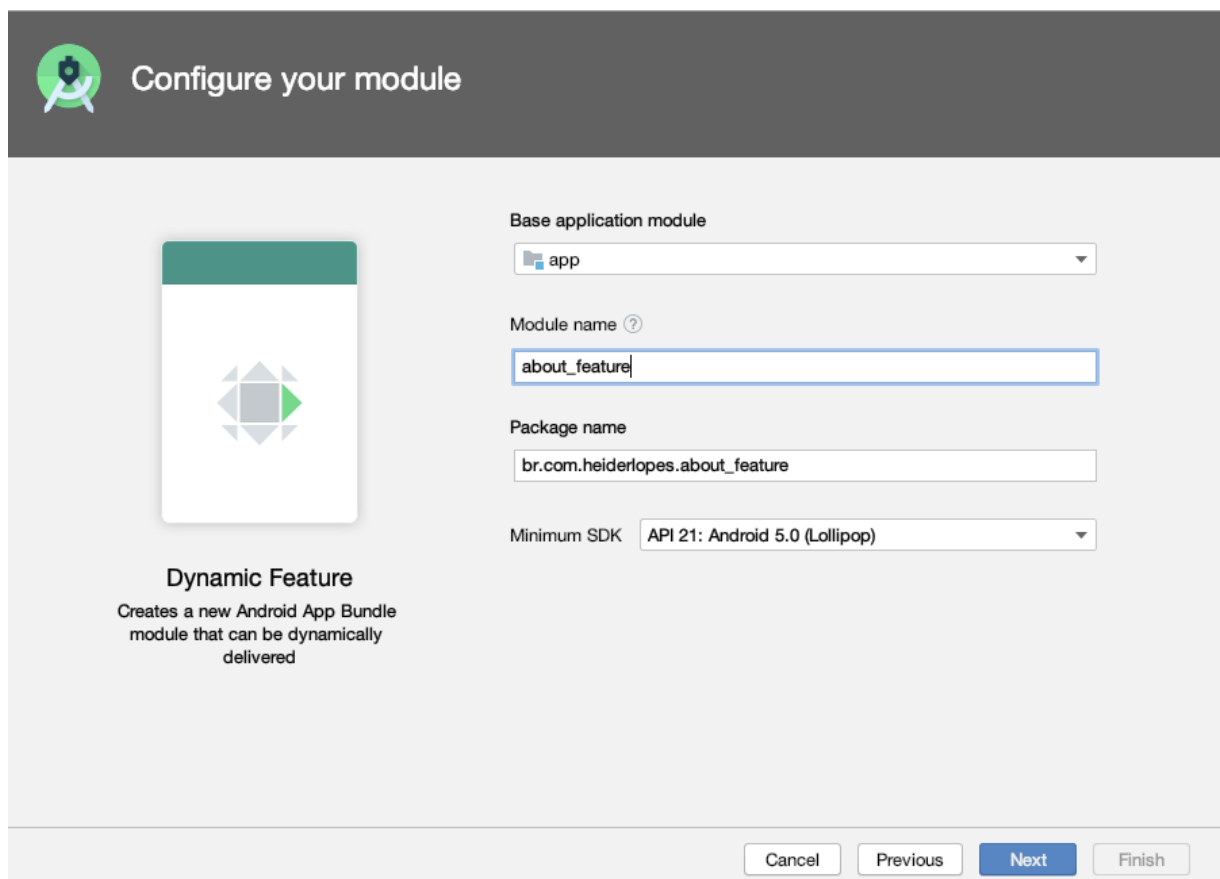


Figura 3.2.1 – Definindo o nome do módulo

Fonte: Próprio autor (2020)

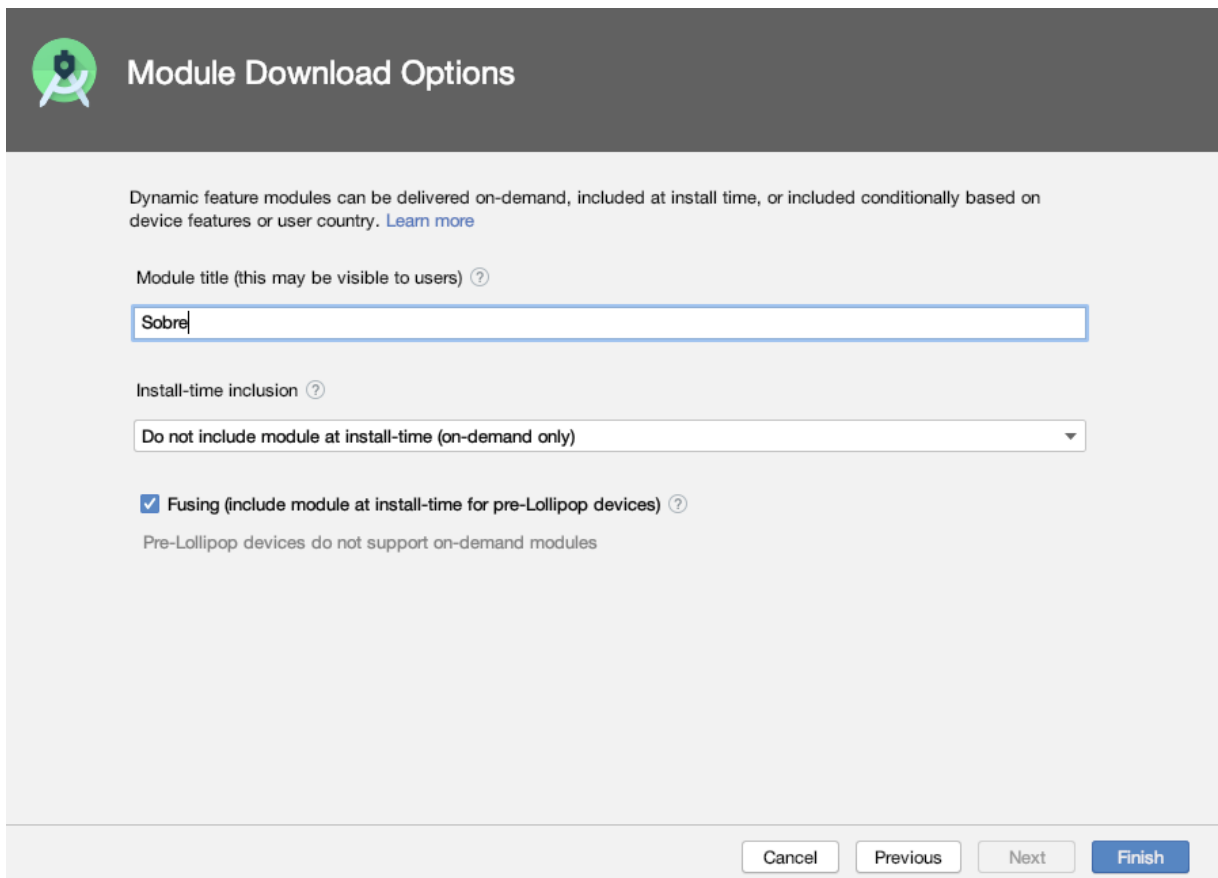


Figura 3.2.1 – Definindo o nome visível para o usuário
Fonte: Próprio autor (2020)

Module Title: a plataforma usa esse título para identificar o módulo para os usuários quando, por exemplo, confirmar se o usuário deseja fazer o download do módulo.

Install-time inclusion: especifique se este módulo deve ser incluído no momento da instalação incondicionalmente ou com base no recurso do dispositivo.

Na nossa **MainActivity**, além do botão para abrir a lista de produtos, a tela principal terá um botão e, ao clicar nele, faremos o download do módulo de recurso about que irá constar informações sobre o aplicativo.

Abra o arquivo **activity_main.xml** e adicione o seguinte código:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="32dp"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/btProducts"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Ver Produtos" />

    <Button
        android:id="@+id/btDownloadAbout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Baixar o módulo Sobre" />

    <Button
        android:id="@+id/btOpenNewsModule"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Sobre"
        android:visibility="gone" />

    <Button
        android:id="@+id/btDeleteNewsModule"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Apagar módulo Sobre"
        android:visibility="gone" />
</LinearLayout>
```

Figura 3.2.1 – Layout da tela principal
Fonte: Próprio autor (2020)

Na tela principal existem 4 botões nos quais 2 botões estão ocultos e somente serão visíveis quando o módulo dinâmico for baixado corretamente. E será usado para abrir a activity no módulo de recurso **about** e outro é para excluir o módulo dinâmico.

Ao ser criado o módulo de recurso about, no **build.gradle** do **app** foi adicionada a seguinte referência:

```
dynamicFeatures = [":about_feature"]
```

Figura 3.2.1 – Layout da tela principal

Fonte: Próprio autor (2020)

Isso significa que foi adicionado um módulo dinâmico no projeto. No **build.gradle** do recurso sobre, foi adicionado o seguinte código:

```
dependencies {  
    ....  
    implementation project(':app')  
}
```

Figura 3.2.1 – Adição do módulo do aplicativo

Fonte: Próprio autor (2020)

O módulo de recurso dinâmico implementa o módulo de aplicativo e o usa como uma dependência. Além disso, no recurso dinâmico temos o seguinte Plug-in Gradle sendo usado:

```
apply plugin: 'com.android.dynamic-feature'
```

Figura 3.2.1 – Aplicação do plugin dynamic feature

Fonte: Próprio autor (2020)

Segue o arquivo **AndroidManifest.xml** referente ao módulo about.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:dist="http://schemas.android.com/apk/distribution"  
    package="br.com.heiderlopes.about_feature">  
  
    <dist:module  
        dist:instant="false"  
        dist:title="@string/title_about_feature">  
        <dist:delivery>  
            <dist:on-demand />  
        </dist:delivery>  
        <dist:fusing dist:include="true" />  
    </dist:module>  
</manifest>
```

```
</dist:module>  
</manifest>
```

Código-fonte 3.2.1 – AndroidManifest.xml do módulo about.

Fonte: Próprio autor (2020)

A tag <dist> ajuda o projeto a saber como o módulo está sendo compactado dist: onDemand = "true": especifica se o módulo está disponível como um download sob demanda.

Para tornar seu módulo disponível para download sob demanda, Precisamos adicionar à dependência: *'com.google.android.play:core:1.6.4'*, no **build.gradle** do módulo do app.

```
implementation 'com.google.android.play:core:1.8.0'
```

Código-fonte 3.2.1 – Adição da biblioteca para distribuição sob demanda

Fonte: Próprio autor (2020)

Abra o arquivo **build.gradle** e troque de **implementation** para **api** as bibliotecas **appcompat** e **constraint_layout**.

Quando usamos a **api**, as bibliotecas podem ser usadas pelos outros módulos, assim como os módulos dinâmicos implementam o módulo do aplicativo. Se usarmos a **implementation**, a biblioteca será usada apenas pelo módulo em que é implementada.

Agora no pacote **about_feature** do módulo **about_feature** adicione uma nova **Activity** chamada **AboutActivity**. Clique com o botão direito sobre o pacote citado → **New** → **Empty Activity**

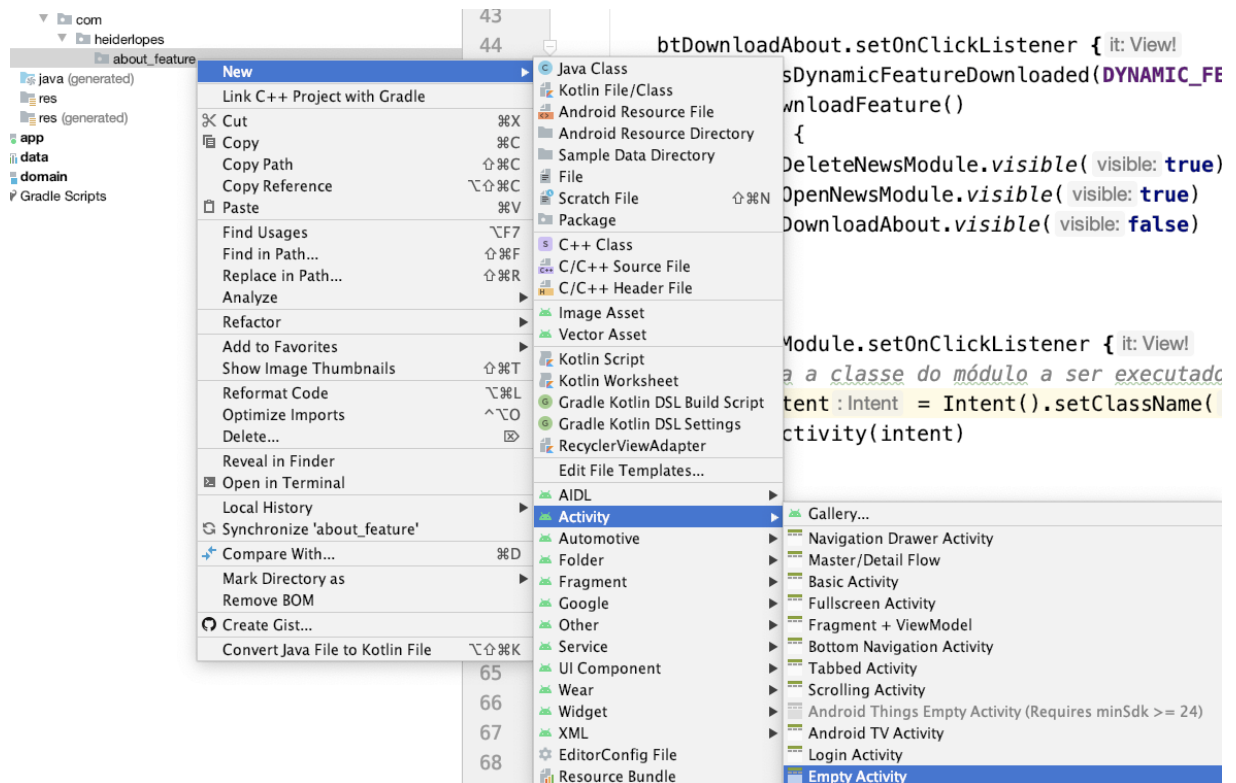


Figura 3.2.1 – Criação da Activity Sobre
Fonte: Próprio autor (2020)

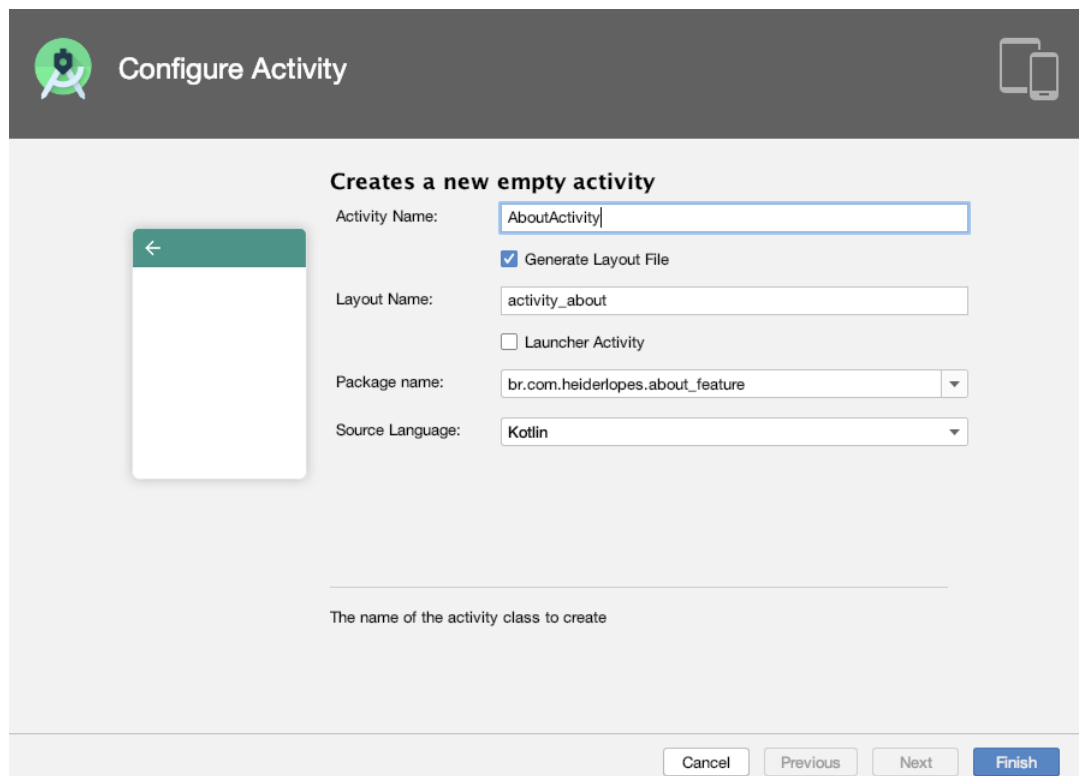


Figura 3.2.1 – Criação da Activity Sobre
Fonte: Próprio autor (2020)

Abra o arquivo **activity_about.xml** e adicione o seguinte código:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".AboutActivity">

    <ImageView
        android:id="@+id/imageView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.38"
        app:srcCompat="@drawable/ic_launcher_foreground" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="16dp"
        android:layout_marginTop="32dp"
        android:layout_marginEnd="16dp"
        android:text="Modularização"
        android:gravity="center"
        android:textSize="16sp"
        android:textStyle="bold"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/imageView" />

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="8dp"
        android:text="Versão: 1.0"
        app:layout_constraintEnd_toEndOf="@+id/textView"
        app:layout_constraintStart_toStartOf="@+id/textView"
        app:layout_constraintTop_toBottomOf="@+id/textView" />
</androidx.constraintlayout.widget.ConstraintLayout>
```


Figura 3.2.1 – Layout da tela Sobre
Fonte: Próprio autor (2020)

Agora, para tornar o recurso dinâmico para download no módulo do app, escreva a lógica para fazer o download dos módulos no arquivo **MainActivity.kt**.

Primeiramente, criaremos 3 variáveis:

```
lateinit var splitInstallManager: SplitInstallManager
lateinit var request: SplitInstallRequest

val DYNAMIC_FEATURE = "about_feature"
```

Código-fonte 3.2.1 – Declaração de variáveis
Fonte: Próprio autor (2020)

SplitInstallManager é responsável por baixar o módulo. O aplicativo deve estar em primeiro plano para baixar o módulo dinâmico.

SplitInstallRequest conterá as informações de solicitação que serão usadas para solicitar nosso módulo de recurso dinâmico do Google Play.

Agora, inicialize as variáveis lateinit em **onCreate** da **MainActivity**:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    initDynamicModules()
}

private fun initDynamicModules() {
    splitInstallManager = SplitInstallManagerFactory.create(this)
    request = SplitInstallRequest
        .newBuilder()
        .addModule(DYNAMIC_FEATURE)
        .build();
}
```

Código-fonte 3.2.1 – Declaração de variáveis
Fonte: Próprio autor (2020)

Onde:

Primeiro, foi criado um factory para **splitInstallManager** e, em seguida, criada a instância **SplitInstallRequest**.

É possível adicionar 1 ou vários módulos através do addModule. Segue abaixo o código completo da nossa classe:

```
class MainActivity : AppCompatActivity() {

    lateinit var splitInstallManager: SplitInstallManager
    lateinit var request: SplitInstallRequest
    val DYNAMIC_FEATURE = "about_feature"

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        initDynamicModules()

        setClickListeners()
    }

    private fun initDynamicModules() {
        splitInstallManager = SplitInstallManagerFactory.create(this)
        request = SplitInstallRequest
            .newBuilder()
            .addModule(DYNAMIC_FEATURE)
            .build()
    }

    private fun setClickListeners() {
        btProducts.setOnClickListener {
            startActivity(Intent(this@MainActivity, ProductListActivity::class.java))
        }

        btDownloadAbout.setOnClickListener {
            if (!isDynamicFeatureDownloaded(DYNAMIC_FEATURE)) {
                downloadFeature()
            } else {
                btDeleteNewsModule.visible(true)
                btOpenNewsModule.visible(true)
                btDownloadAbout.visible(false)
            }
        }
    }
}
```

```
    }  
}  
  
btOpenNewsModule.setOnClickListener {  
    //Chama a classe do módulo a ser executado  
    val intent = Intent().setClassName(this,  
    "br.com.heiderlopes.about_feature.AboutActivity")  
    startActivity(intent)  
}  
  
btDeleteNewsModule.setOnClickListener {  
    val list = ArrayList<String>()  
    list.add(DYNAMIC_FEATURE)  
    uninstallDynamicFeature(list)  
}  
}  
  
private fun uninstallDynamicFeature(list: List<String>) {  
    splitInstallManager.deferredUninstall(list)  
        .addOnSuccessListener {  
            btDeleteNewsModule.visible(false)  
            btOpenNewsModule.visible(false)  
            btDownloadAbout.visible(true)  
        }  
}  
  
private fun isDynamicFeatureDownloaded(feature: String): Boolean =  
    splitInstallManager.installedModules.contains(feature)  
  
private fun downloadFeature() {  
    splitInstallManager.startInstall(request)  
        .addOnFailureListener {  
        }  
        .addOnSuccessListener {  
            btOpenNewsModule.visible(true)  
            btDeleteNewsModule.visible(true)  
            btDownloadAbout.visible(false)  
        }  
        .addOnCompleteListener {  
        }  
}  
  
//Check status download dynamic module  
/*var mySessionId = 0  
val listener = SplitInstallStateUpdatedListener {  
    mySessionId = it.sessionId()  
    when (it.status()) {  
        SplitInstallSessionStatus.DOWNLOADING -> {
```

```
        val totalBytes = it.totalBytesToDownload()
        val progress = it.bytesDownloaded()
        // Update progress bar.
    }
    SplitInstallSessionStatus.INSTALLING -> Log.d("Status", "INSTALLING")
    SplitInstallSessionStatus.INSTALLED -> Log.d("Status", "INSTALLED")
    SplitInstallSessionStatus.FAILED -> Log.d("Status", "FAILED")
    SplitInstallSessionStatus.REQUIRES_USER_CONFIRMATION ->
    Log.d("Status", "REQUIRES_USER_CONFIRMATION")
    }
    }*/
}
```

Código-fonte 3.2.1 –MainActivity com gerenciamento de módulos

Fonte: Próprio autor (2020)

Segue as imagens do aplicativo:



Figura 3.2.1 – Tela do aplicativo para baixar o módulo Sobre
Fonte: Próprio autor (2020)

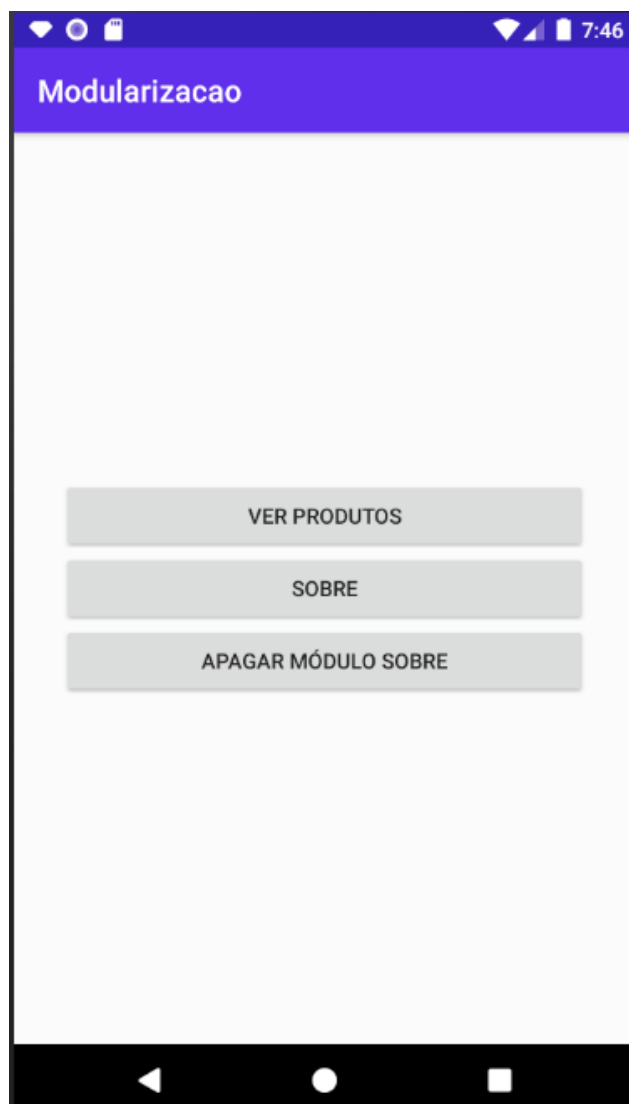


Figura 3.2.1 – Tela do aplicativo com módulo Sobre baixado
Fonte: Próprio autor (2020)

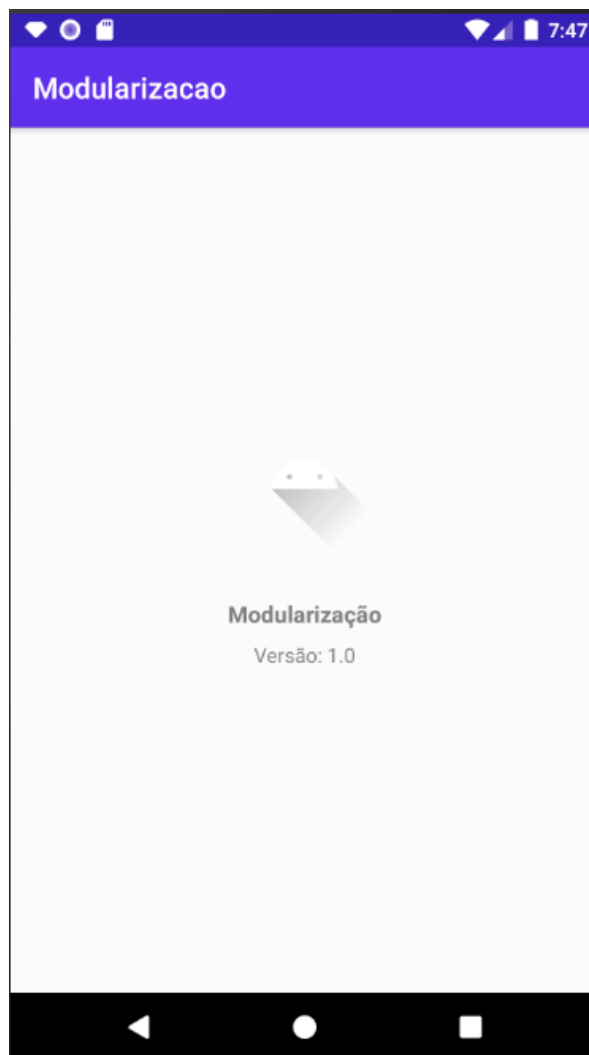


Figura 3.2.1 – Tela Sobre do aplicativo
Fonte: Próprio autor (2020)

4 Considerações sobre modularização

Neste capítulo, foi possível conhecer o que é e como criar um aplicativo modular. Com esse conhecimento será possível construir aplicativos cada vez menos acoplados, mais testáveis, apks menores e com builds mais rápidos.

REFERÊNCIAS

<https://medium.com/android-dev-br/modulariza%C3%A7%C3%A3o-android-parte-1-b69b509571c9>

<https://developer.android.com/studio/projects/dynamic-delivery?hl=pt-br>