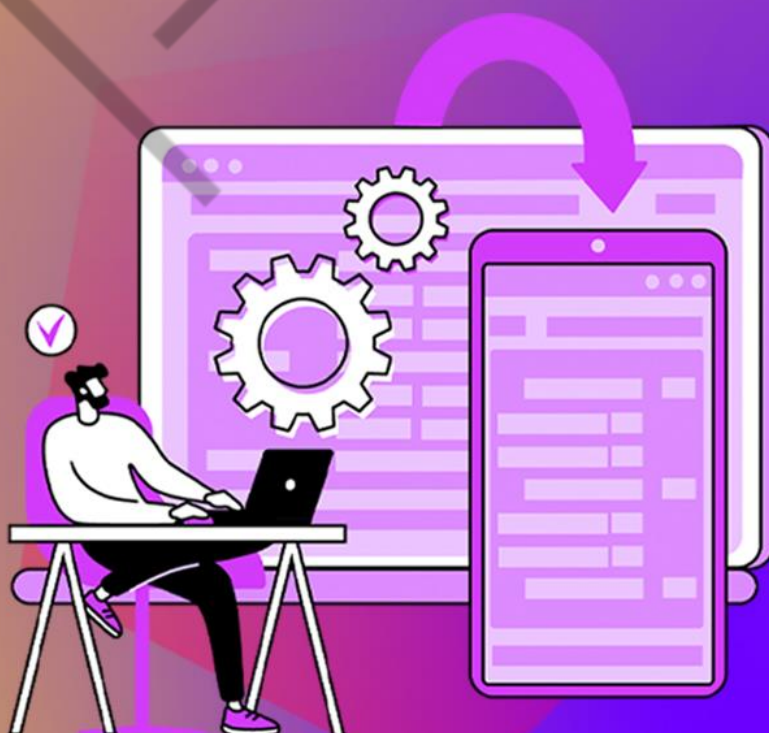


TÓPICOS AVANÇADOS DE
DESENVOLVIMENTO ANDROID

MODULARIZAÇÃO DE APLICAÇÕES ANDROID

HEIDER PINHOLI LOPES



4

LISTA DE FIGURAS

Figura 4.1 - Menu Principal com botão para baixar módulo	7
Figura 4.2 - Menu Principal com botão para exibir ou excluir.....	8
Figura 4.3 - Listagem de produtos.....	9
Figura 4.4 - Criação do projeto.....	10
Figura 4.5 - Seleção de Template do Projeto	10
Figura 4.6 - Seleção de Template do Projeto	11
Figura 4.7 - Diagrama de fluxo do domain module.....	12
Figura 4.8 - Criando um novo módulo	13
Figura 4.9 - Criando uma Java or Kotlin Library	13
Figura 4.10 - Definindo o módulo domain.....	14
Figura 4.11 - Modo de visualização Android	14
Figura 4.12 - Modo de visualização Android	15
Figura 4.13 - Criação do arquivo dependencies.gradle	15
Figura 4.14 - Nomeação do arquivo dependencies.gradle	15
Figura 4.15 - Arquivo dependencies.gradle na raiz do projeto	16
Figura 4.16 - Exemplo de jogo que será retornado pelo serviço	20
Figura 4.17 - Classe Product dentro do package	20
Figura 4.18 - Interface ProductRepository	21
Figura 4.19 - Caso de uso para busca de produtos	22
Figura 4.20 - Classe com os módulos para utilização de injeção de dependências..	22
Figura 4.21 - Domain module	23
Figura 4.22 - Data module.....	24
Figura 4.23 - Criando o Data module	25
Figura 4.24 - Criando o novo módulo como Android Library	25
Figura 4.25 - Definindo o nome do módulo como data.....	26
Figura 4.26 - Classe ProductCache	28
Figura 4.27 - Classes do pacote database	28
Figura 4.28 - Classe ProductCacheMapper	30
Figura 4.29 - Estrutura dos pacotes data com o package remote	32
Figura 4.30 - Estrutura de pacote e classes do data module	36
Figura 4.31 - Classe DataModule.....	37
Figura 4.32 - Pacote di e extensions do presentation module.....	39
Figura 4.33 - Item da lista de produtos.....	45
Figura 4.34 - Exibição dos dados no aplicativo	54
Figura 4.35 - Estrutura de pacotes com a modularização por funcionalidade	55
Figura 4.36 - Estrutura de pacotes com a modularização por funcionalidade com as classes	56
Figura 4.37 - Criando uma Empty Activity	56
Figura 4.38 - Definindo o nome da EmptyActivity como MainActivity	57
Figura 4.39 - Definindo o módulo com Dynamic Feature Module.....	58
Figura 4.40 - Definindo o nome do módulo	59
Figura 4.41 - Definindo o nome visível para o usuário	59
Figura 4.42 - Criação da Activity Sobre.....	63
Figura 4.43 - Criação da Activity Sobre.....	63
Figura 4.44 - Tela do aplicativo para baixar o módulo Sobre	68
Figura 4.45 - Tela do aplicativo com módulo Sobre baixado.....	68
Figura 4.46 - Tela Sobre do aplicativo.....	69

LISTA DE CÓDIGOS-FONTE

Código-fonte 4.1 – Arquivo dependencies.gradle com as dependências do projeto	18
Código-fonte 4.2 – Aplicando o dependencies.gradle no projeto	19
Código-fonte 4.3 – Aplicando o dependencies.gradle no projeto	19
Código-fonte 4.4 – Código da classe Product	21
Código-fonte 4.5 – Interface ProductRepository	21
Código-fonte 4.6 – Caso de uso para busca de produtos	22
Código-fonte 4.7 – Classe com os módulos para utilização de injeção de dependências	23
Código-fonte 4.8 – Arquivo de dependencies do gradle do módulo data	27
Código-fonte 4.9 – Classe ProductCache	28
Código-fonte 4.10 – Classe ProductsDao	29
Código-fonte 4.11 – Classe ProductDataBase	29
Código-fonte 4.12 – Classe ProductCacheMapper	30
Código-fonte 4.13 – Interface ProductCacheDataSource	31
Código-fonte 4.14 – Interface ProductCacheDataSourceImpl	31
Código-fonte 4.15 – Classe ProductPayload	32
Código-fonte 4.16 – Interface ProductAPI	32
Código-fonte 4.17 – Classe ProductPayloadMapper	33
Código-fonte 4.18 – Classe ProductRemoteDataSource	33
Código-fonte 4.19 – Classe ProductRemoteDataSourceImpl	33
Código-fonte 4.20 – Classe ProductRepositoryImpl	34
Código-fonte 4.21 – Classe DataCacheModule	35
Código-fonte 4.22 – Classe DataRemoteModule	35
Código-fonte 4.23 – Classe DataModule	36
Código-fonte 4.24 – build.gradle do presentation module	38
Código-fonte 4.25 – Classe de extensão Context.kt	39
Código-fonte 4.26 – Classe de extensão View.kt	39
Código-fonte 4.27 – Classe de extensão ViewGroup.kt	39
Código-fonte 4.28 – Classe BaseViewModel	40
Código-fonte 4.29 – Classe ViewState	40
Código-fonte 4.30 – Classe ViewState	42
Código-fonte 4.31 – Classe MainActivity	43
Código-fonte 4.32 – Classe Presentation Module	44
Código-fonte 4.33 – Classe MyApplication	44
Código-fonte 4.34 – AndroidManifest.xml	45
Código-fonte 4.35 – Item produtos da lista	46
Código-fonte 4.36 – Inclusão da biblioteca Picasso no dependencies.gradle	48
Código-fonte 4.37 – Inclusão da biblioteca Picasso no módulo app	49
Código-fonte 4.38 – Implementação do MainListAdapter	50
Código-fonte 4.39 – Injetando o adapter	50
Código-fonte 4.40 – Layout da MainActivity	52
Código-fonte 4.41 – Habilitando o databinding	52
Código-fonte 4.42 – MainActivity para exibir os dados	54
Código-fonte 4.43 – Definindo a MainActivity como a principal	58
Código-fonte 4.44 – Criando um novo módulo	58
Código-fonte 4.45 – Layout da tela principal	60
Código-fonte 4.46 – Layout da tela principal	61

Código-fonte 4.47 – Adição do módulo do aplicativo	61
Código-fonte 4.48 – Aplicação do plugin dynamic feature.....	61
Código-fonte 4.49 – AndroidManifest.xml do módulo about.	62
Código-fonte 4.50 – Adição da biblioteca para distribuição sob demanda	62
Código-fonte 4.51 – Layout da tela Sobre	64
Código-fonte 4.52 – Declaração de variáveis.....	65
Código-fonte 4.53 – Declaração de variáveis.....	65
Código-fonte 4.54 – MainActivity com gerenciamento de módulos	67

EXEMPLO

SUMÁRIO

4 MODULARIZAÇÃO DE APLICAÇÕES ANDROID	6
4.1 Projeto de Aplicação Modular.....	7
4.2 Modularização por camadas (layers).....	11
4.2.1 Domain Module	12
4.2.2 Criando o Domain Module.....	13
4.2.3 Pacotes do domain.....	19
4.2.4 Definindo a Entity	20
4.2.5 Criando o Repository.....	21
4.2.6 Criando os UseCases.....	21
4.2.7 Data Module	24
4.2.8 Criando o cache	27
4.2.9 Consumindo os dados remotos.....	31
4.2.10 Criando o Repository.....	33
4.2.11 Presentation Module.....	36
4.2.12 Programando a classe principal	42
4.2.13 Inicializando o Koin.....	44
4.2.14 Melhorando o visual das linhas da lista	45
4.3 Modularização de recursos (features)	54
4.3.1 Criando módulos dinâmicos	55
CONCLUSÃO.....	70

4 MODULARIZAÇÃO DE APLICAÇÕES ANDROID

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.

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

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 em todo o projeto.

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.

APKs menores: distribuí 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, em que 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.

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.

4.1 Projeto de Aplicação Modular

Para compreendermos todos os benefícios de um projeto modular, vamos criar nosso próprio projeto.

Nesse projeto será desenvolvido um aplicativo responsável em buscar uma lista de produtos por meio 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, o qual só será disponibilizado para o usuário caso ele utilize a funcionalidade, tornando o aplicativo menor no momento do download).

Seguem as imagens do aplicativo final.



Figura 4.1 - Menu Principal com botão para baixar módulo
Fonte: Elaborado pelo autor (2020)

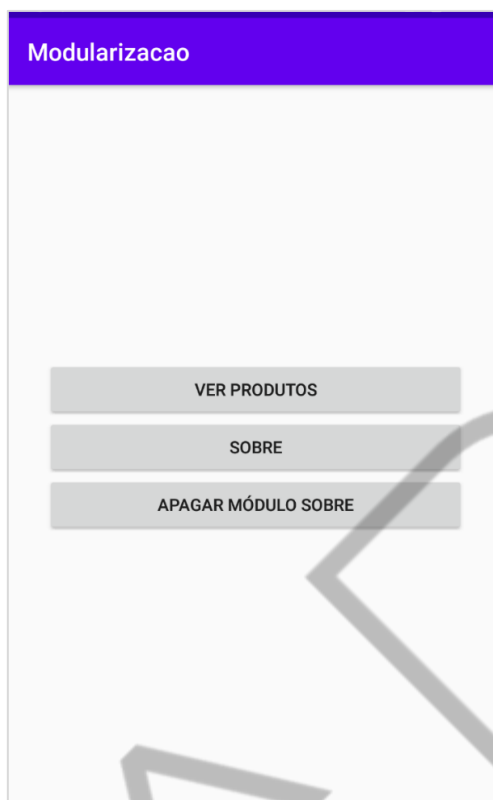


Figura 4.2 - Menu Principal com botão para exibir ou excluir
Fonte: Elaborado pelo autor (2020)

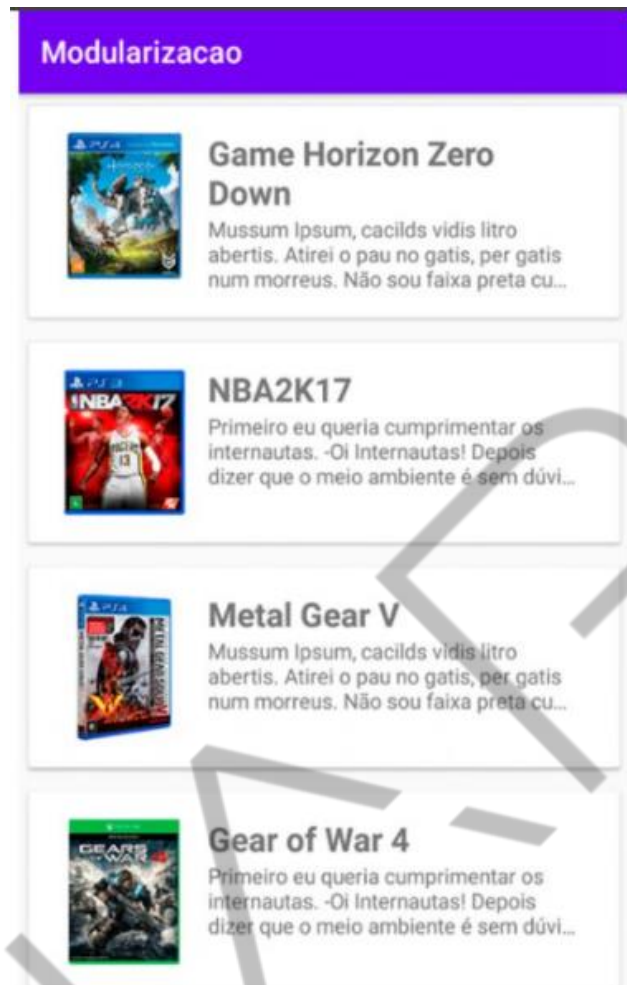


Figura 4.3 - Listagem de produtos
Fonte: Elaborado pelo autor (2020)

Nosso projeto buscará uma lista de produtos a partir de uma API e organizará em uma lista visual. Cada parte dessa aplicação será separada em módulos com a finalidade de poder ser reutilizada em outros projetos.

Para isso, abra o Android Studio e clique em **Start a new Android Studio Project:**

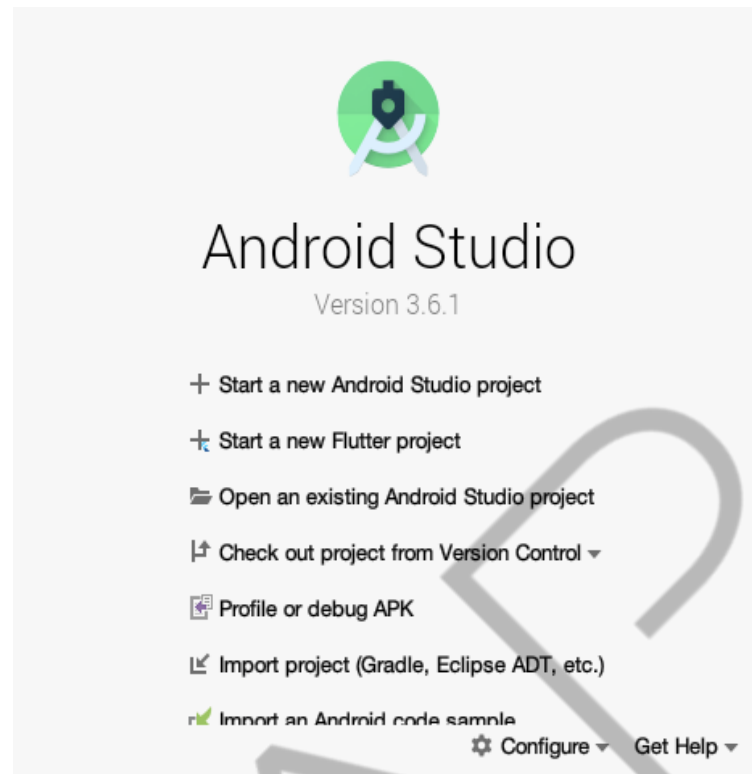


Figura 4.4 - Criação do projeto
Fonte: Elaborado pelo autor (2020)

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

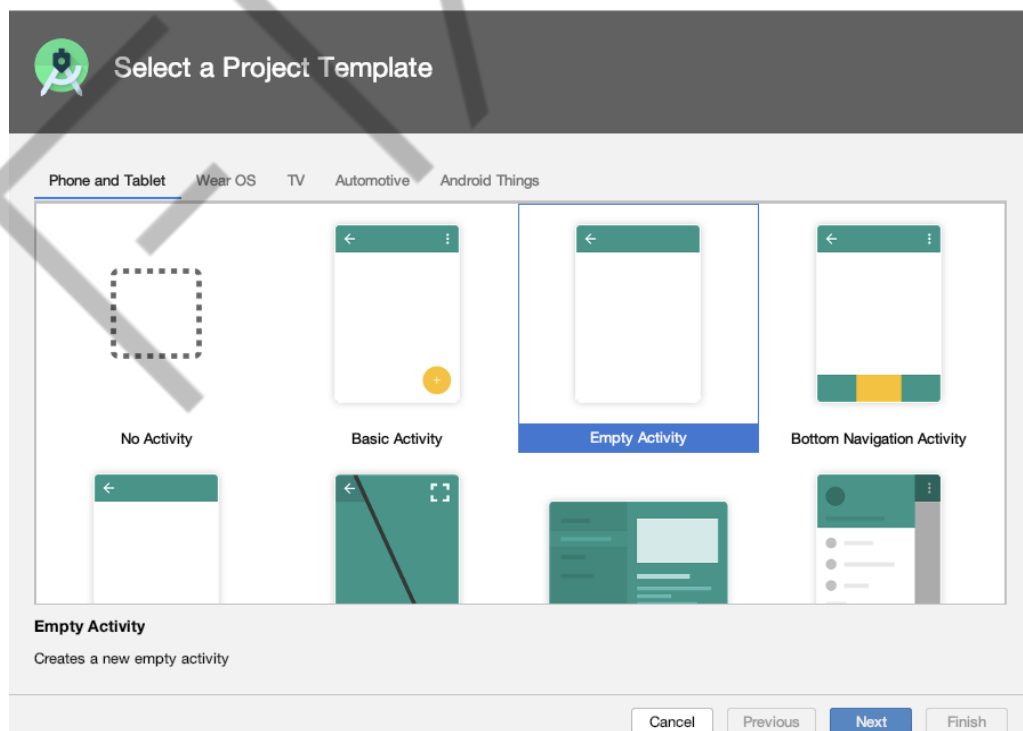


Figura 4.5 - Seleção de Template do Projeto
Fonte: Elaborado pelo autor (2020)

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

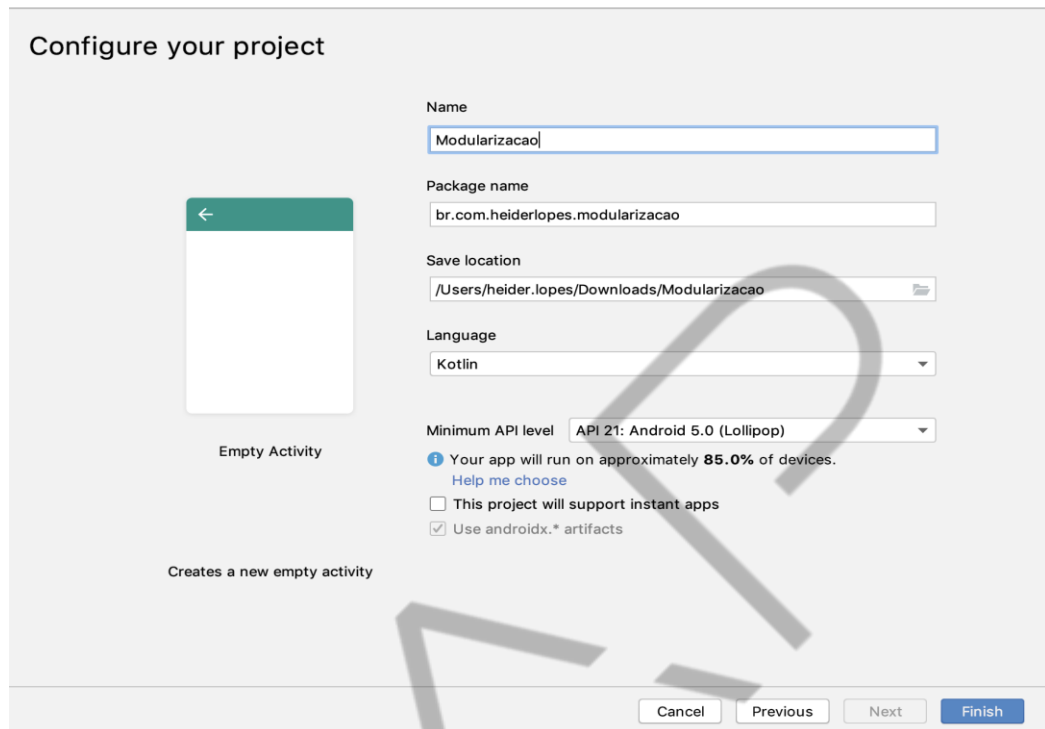


Figura 4.6 - Seleção de Template do Projeto
Fonte: Elaborado pelo autor (2020)

Podemos modularizar um aplicativo de duas maneiras: por layer (camada) ou por feature (recurso). Vamos começar pela modularização por camadas e na sequência apresentamos a modularização por recurso que, inclusive, pode ser entregue dinamicamente.

4.2 Modularização por camadas (layers)

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

4.2.1 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 desse 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 4.7 - Diagrama de fluxo do domain module
Fonte: Elaborado pelo 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**.

4.2.2 Criando o Domain Module

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

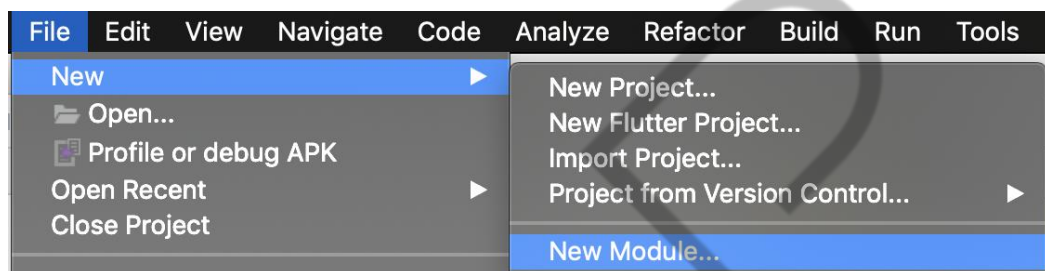


Figura 4.8 - Criando um novo módulo
Fonte: Elaborado pelo autor (2020)

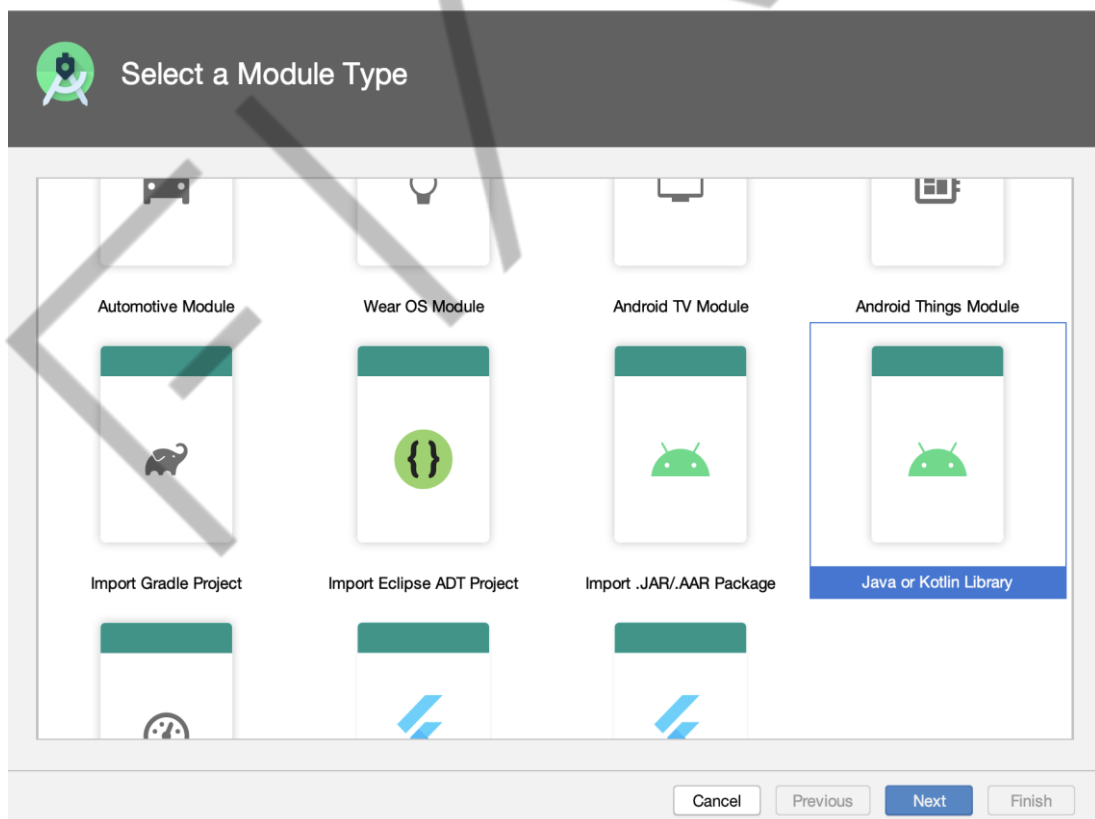


Figura 4.9 - Criando uma Java or Kotlin Library
Fonte: Elaborado pelo autor (2020)

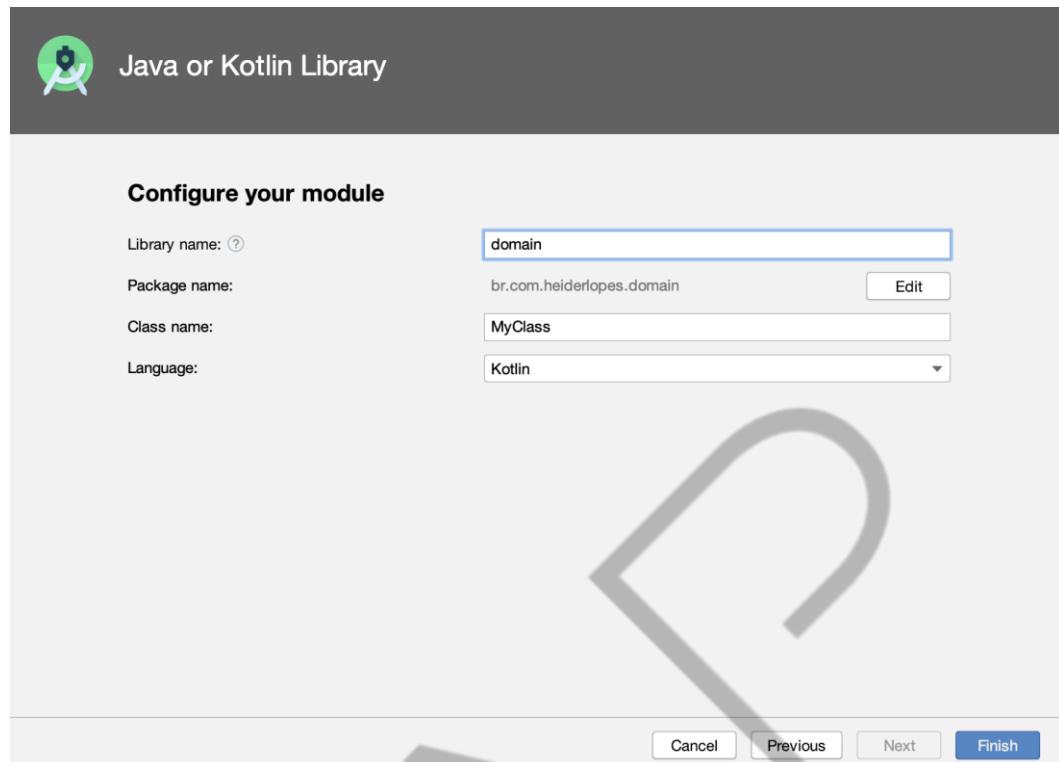


Figura 4.10 - Definindo o módulo domain

Fonte: Elaborado pelo autor (2020)

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

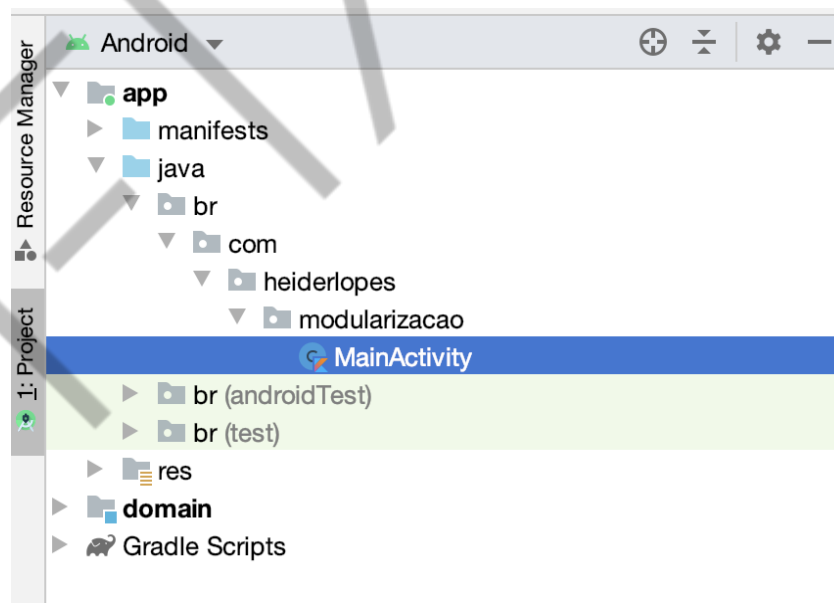


Figura 4.11 - Modo de visualização Android

Fonte: Elaborado pelo autor (2020)

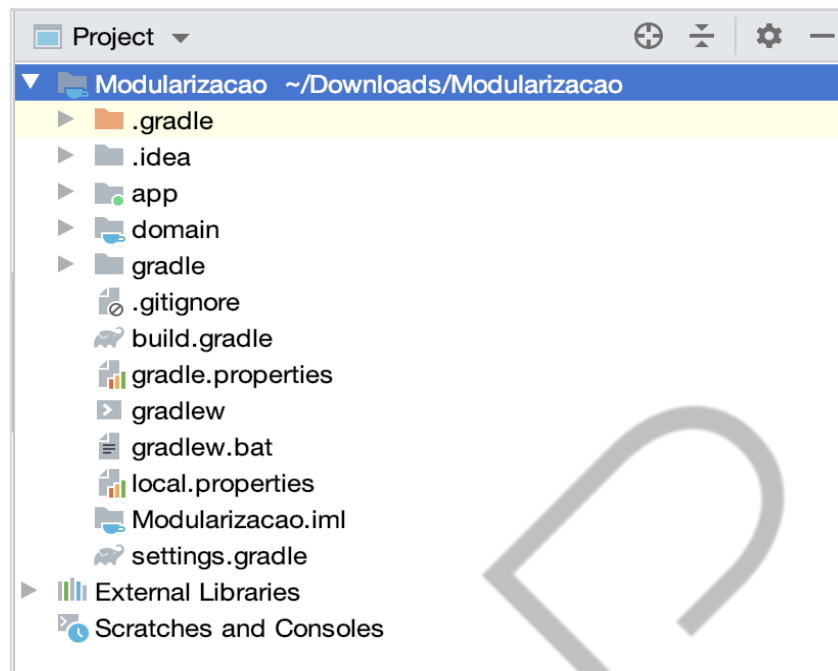


Figura 4.12 - Modo de visualização Android

Fonte: Elaborado pelo autor (2020)

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

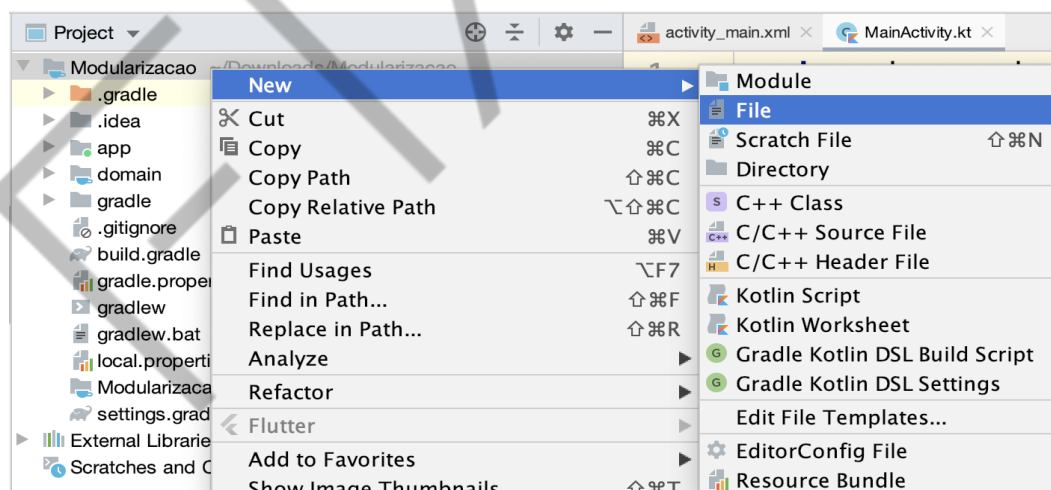


Figura 4.13 - Criação do arquivo dependencies.gradle

Fonte: Elaborado pelo autor (2020)

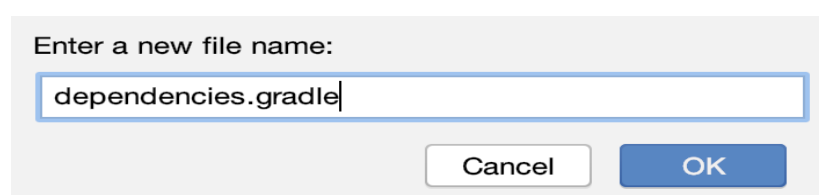


Figura 4.14 - Nomeação do arquivo dependencies.gradle

Fonte: Elaborado pelo autor (2020)

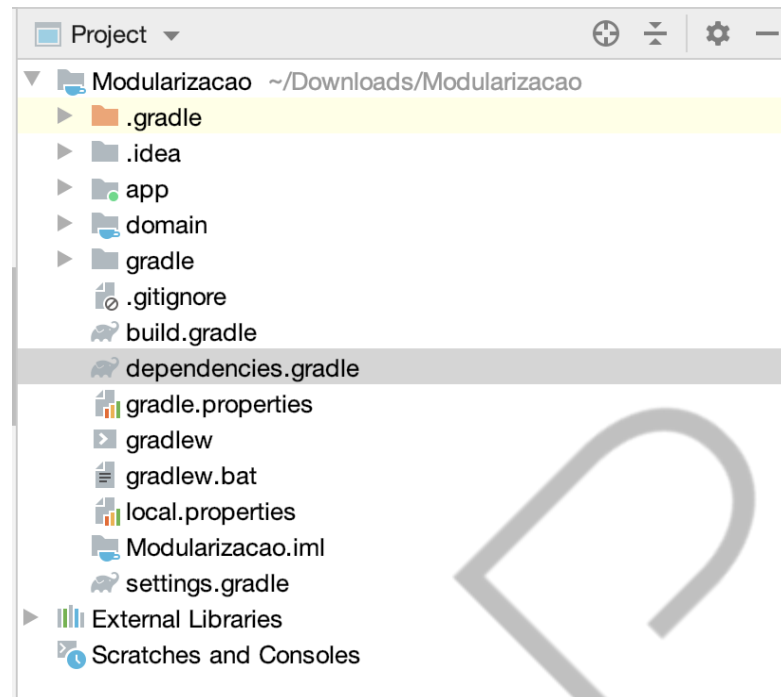


Figura 4.15 - Arquivo dependencies.gradle na raiz do projeto
Fonte: Elaborado pelo autor (2020)

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 4.1 — Arquivo dependencies.gradle com as dependências do projeto
Fonte: Elaborado pelo 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 4.2 — Aplicando o dependencies.gradle no projeto
Fonte: Elaborado pelo 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 4.3 — Aplicando o dependencies.gradle no projeto
Fonte: Elaborado pelo 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.

4.2.3 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:

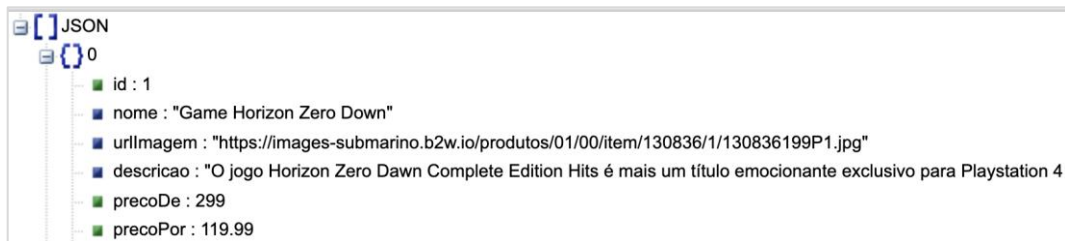


Figura 4.16 - Exemplo de jogo que será retornado pelo serviço

Fonte: Elaborado pelo autor (2020)

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

Utilizamos o **Mocky.io**, acessível em <<https://designer.mocky.io/>>, para criar protótipos de retornos de API sem precisar desenvolver todo o backend. É bem útil, quando queremos testar retornos para consumir em front-end e aplicações nativas.

4.2.4 Definindo a Entity

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

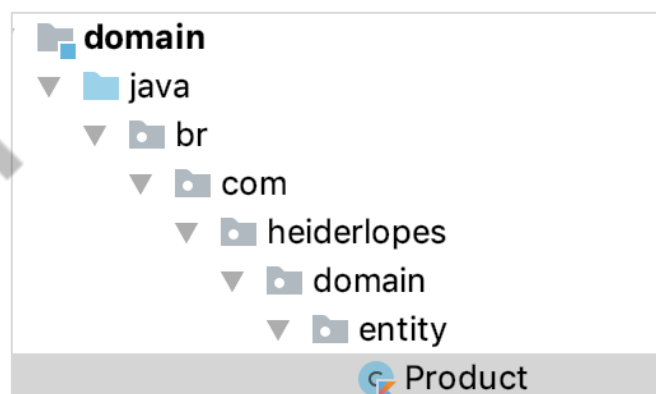


Figura 4.17 - Classe Product dentro do package

Fonte: Elaborado pelo 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 4.4 — Código da classe Product
Fonte: Elaborado pelo autor (2020)

4.2.5 Criando o 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**.

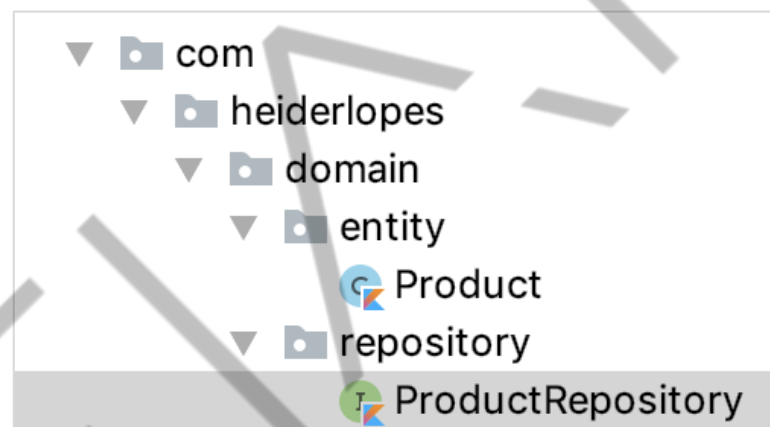


Figura 4.18 - Interface ProductRepository
Fonte: Elaborado pelo autor (2020)

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

Código-fonte 4.5 — Interface ProductRepository
Fonte: Elaborado pelo autor (2020)

4.2.6 Criando os 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**

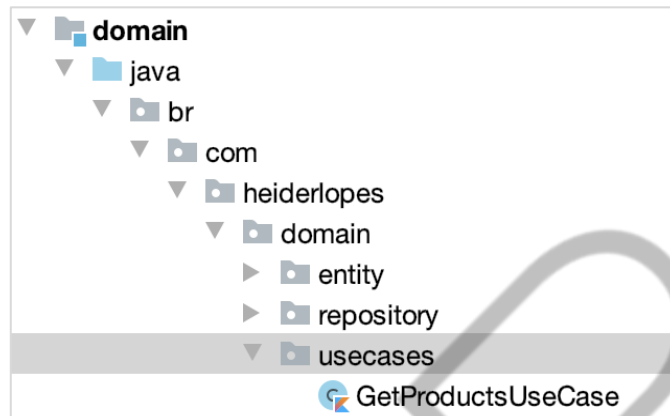


Figura 4.19 - Caso de uso para busca de produtos
Fonte: Elaborado pelo autor (2020)

```
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 4.6 – Caso de uso para busca de produtos
Fonte: Elaborado pelo autor (2020)

No construtor serão utilizadas 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** por meio do framework **koin**.

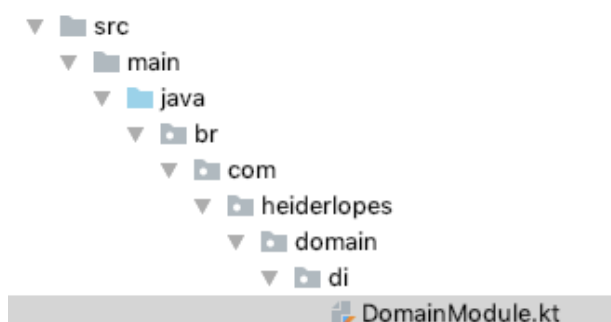


Figura 4.20 - Classe com os módulos para utilização de injeção de dependências

Fonte: Elaborado pelo autor (2020)

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

Código-fonte 4.7 – Classe com os módulos para utilização de injeção de dependências

Fonte: Elaborado pelo autor (2020)

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

O repository = get(), no qual 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:

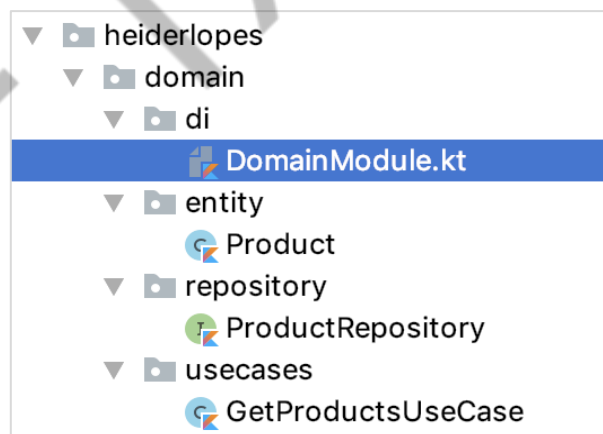


Figura 4.21 - Domain module

Fonte: Elaborado pelo autor (2020)

4.2.7 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.

Nesse módulo são encontrados:

Api: localiza todos os endpoints que serão utilizados para requisitar dados do backend.

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

Mapper: mapeia os models para as entidades exigidas pela domain.

RepositoryImpl: implementa a interface repository do domain e decide de qual lugar serão pegos os dados, se do cache ou do backend.

CacheDataSource: interface de comunicação que é implementada no cache para pegar os dados localmente.

CacheDataSourceImpl: grava os dados no cache e os fornece já mapeados.

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

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

Diagrama de fluxo do módulo data:

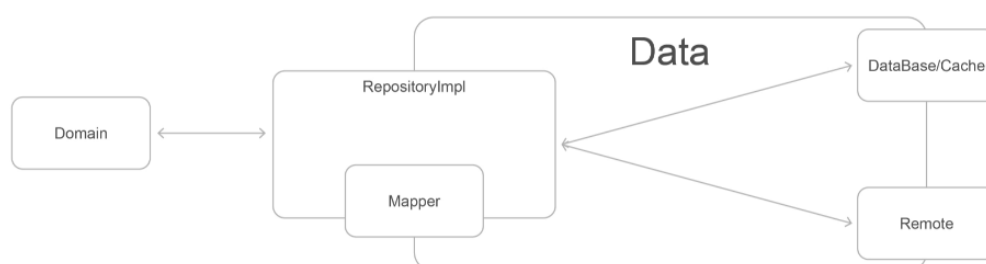


Figura 4.22 - Data module
Fonte: Elaborado pelo autor (2020)

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 chamado o **cache** para verificar se tem algum dado para retornar e, caso não tenha, o **remote** é chamado, esses dados serão gravados no **cache**, então os dados requeridos serão retornados para a domain.

3.1.2.2 Criando o Data Module

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

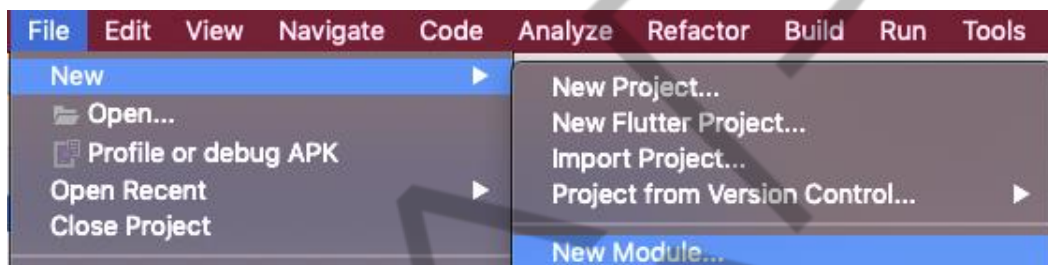


Figura 4.23 - Criando o Data module
Fonte: Elaborado pelo autor (2020)

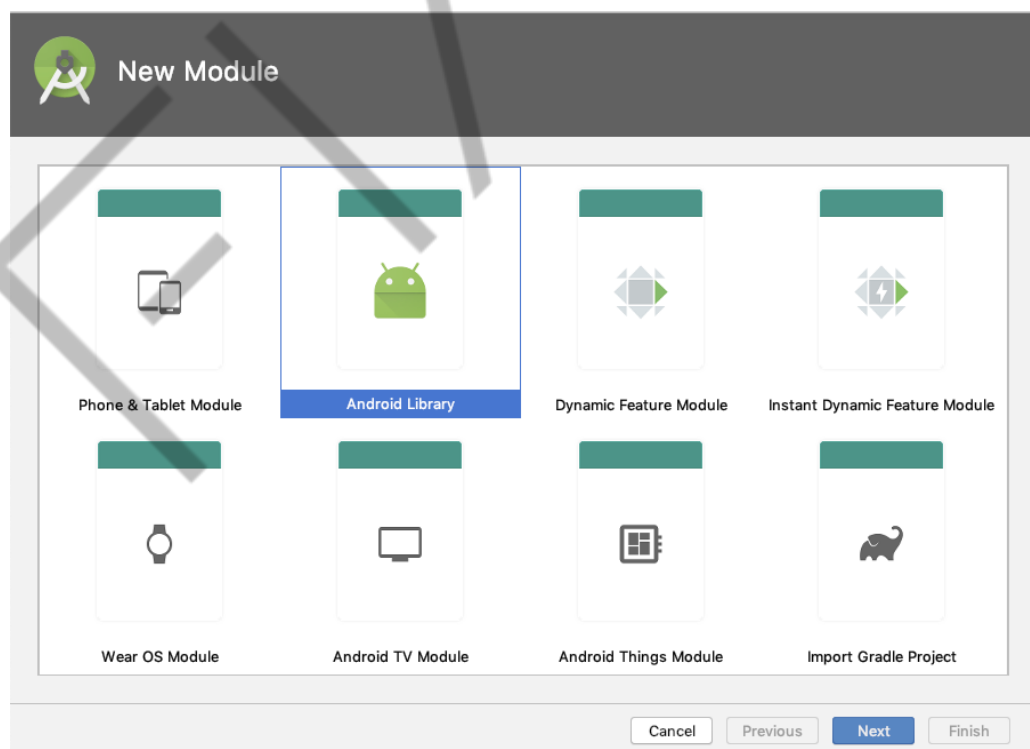


Figura 4.24 - Criando o novo módulo como Android Library
Fonte: Elaborado pelo autor (2020)

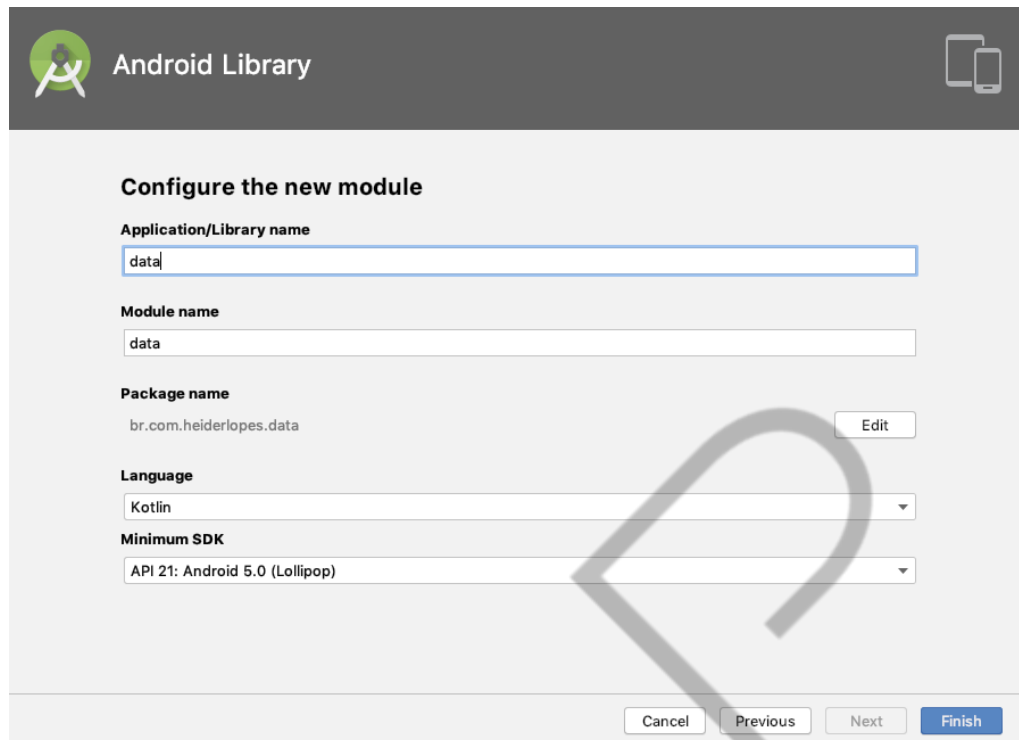


Figura 4.25 - Definindo o nome do módulo como data

Fonte: Elaborado pelo autor (2020)

Nessa parte serão utilizadas as dependências já adicionadas no projeto (arquivo **dependencies.gradle**). Como o projeto realizará chamadas para o backend, será utilizada a biblioteca **Retrofit** e, para o nosso 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 4.8 – Arquivo de dependencias do gradle do módulo data
Fonte: Elaborado pelo autor (2020)

4.2.8 Criando o cache

Comece implementando o cache, deixando-o preparado para salvar nossos dados. No **model** criaremos a entidade que será utilizada pelo database.

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

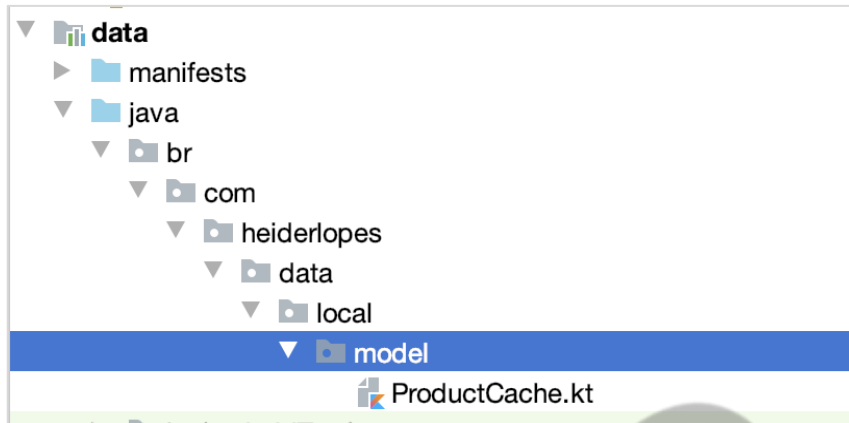


Figura 4.26 - Classe ProductCache
Fonte: Elaborado pelo autor (2020)

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

Código-fonte 4.9– Classe ProductCache
Fonte: Elaborado pelo 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).

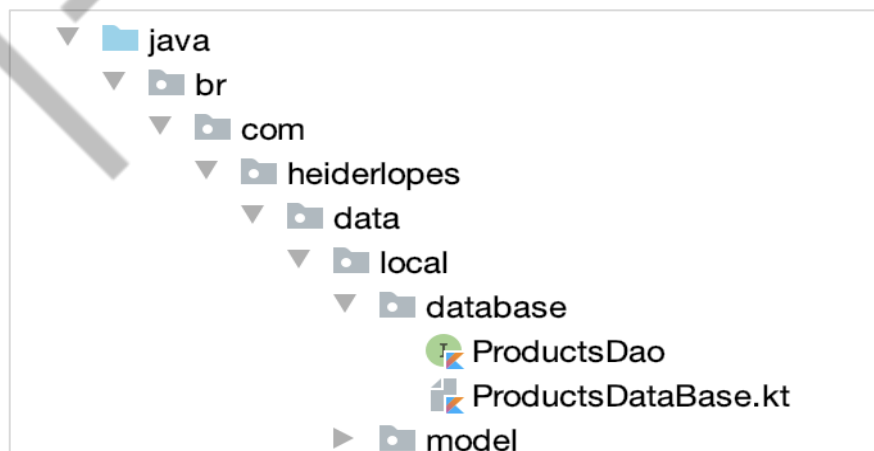


Figura 4.27 - Classes do pacote database
Fonte: Elaborado pelo autor (2020)

```
@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 4.10 – Classe ProductsDao
Fonte: Elaborado pelo 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 4.11 – Classe ProductDataBase
Fonte: Elaborado pelo autor (2020)

Feito isso, crie um pacote chamado **mapper** e, dentro dele, crie uma classe chamada **ProductCacheMapper**. No Mapper, serão mapeados os dados para salvar no cache e também serão mapeados os dados do cache para serem enviados corretamente.

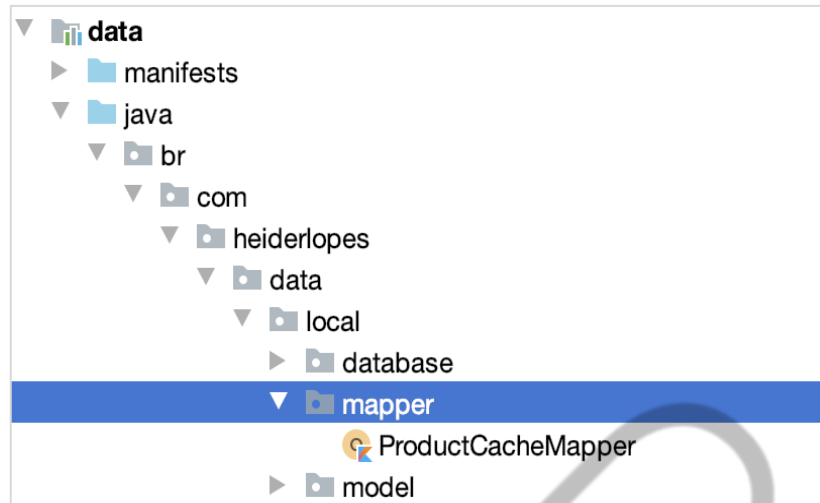


Figura 4.28 - Classe ProductCacheMapper
Fonte: Elaborado pelo autor (2020)

```
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 4.12 — Classe ProductCacheMapper
Fonte: Elaborado pelo 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 do 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 4.13 – Interface ProductCacheDataSource
Fonte: Elaborado pelo 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(prod  
        ucts))  
    }  
}
```

Código-fonte 4.14 – Interface ProductCacheDataSourceImpl
Fonte: Elaborado pelo autor (2020)

4.2.9 Consumindo os dados remotos

Agora vamos preparar 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).

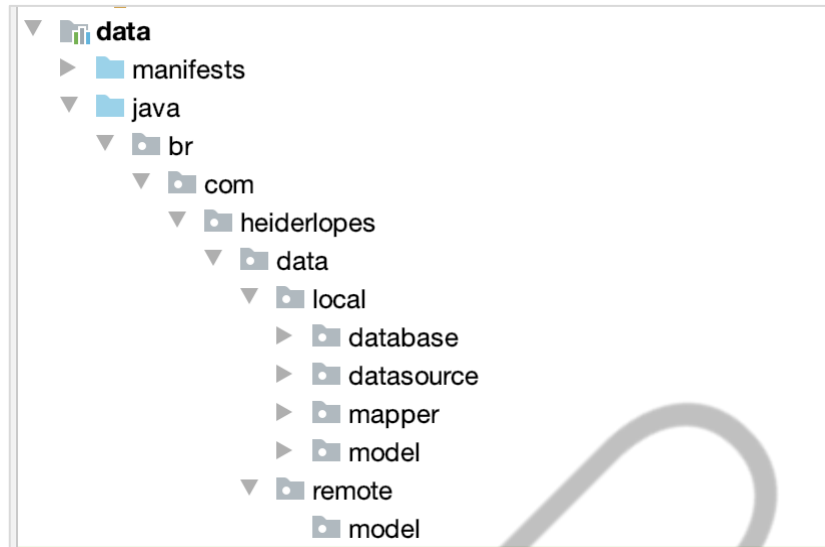


Figura 4.29 - Estrutura dos pacotes data com o package remote
Fonte: Elaborado pelo autor (2020)

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 4.15 – Classe `ProductPayload`
Fonte: Elaborado pelo autor (2020)

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

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

Código-fonte 4.16 – Interface `ProductAPI`
Fonte: Elaborado pelo autor (2020)

Crie um pacote chamado **mapper** dentro do pacote **remote**. Aqui serão mapeados 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 4.17 – Classe ProductPayloadMapper
Fonte: Elaborado pelo autor (2020)

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

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

Código-fonte 4.18 – Classe ProductRemoteDataSource
Fonte: Elaborado pelo autor (2020)

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

Código-fonte 4.19 – Classe ProductRemoteDataSourceImpl
Fonte: Elaborado pelo autor (2020)

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

4.2.10 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 4.20 – Classe ProductRepositoryImpl
Fonte: Elaborado pelo autor (2020)

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

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

```
= get()) }  
}
```

Código-fonte 4.21 – Classe DataCacheModule

Fonte: Elaborado pelo 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 4.22 – Classe DataRemoteModule

Fonte: Elaborado pelo autor (2020)

```
val repositoryModule = module {  
    factory<ProductRepository> {  
        ProductRepositoryImpl(  
            productsCacheDataSource = get(),
```

```
        productRemoteDataSource = get()
    }
}

val dataModules = listOf(remoteDataSourceModule, repositoryModule,
    cacheDataModule)
```

Código-fonte 4.23 – Classe DataModule
Fonte: Elaborado pelo autor (2020)

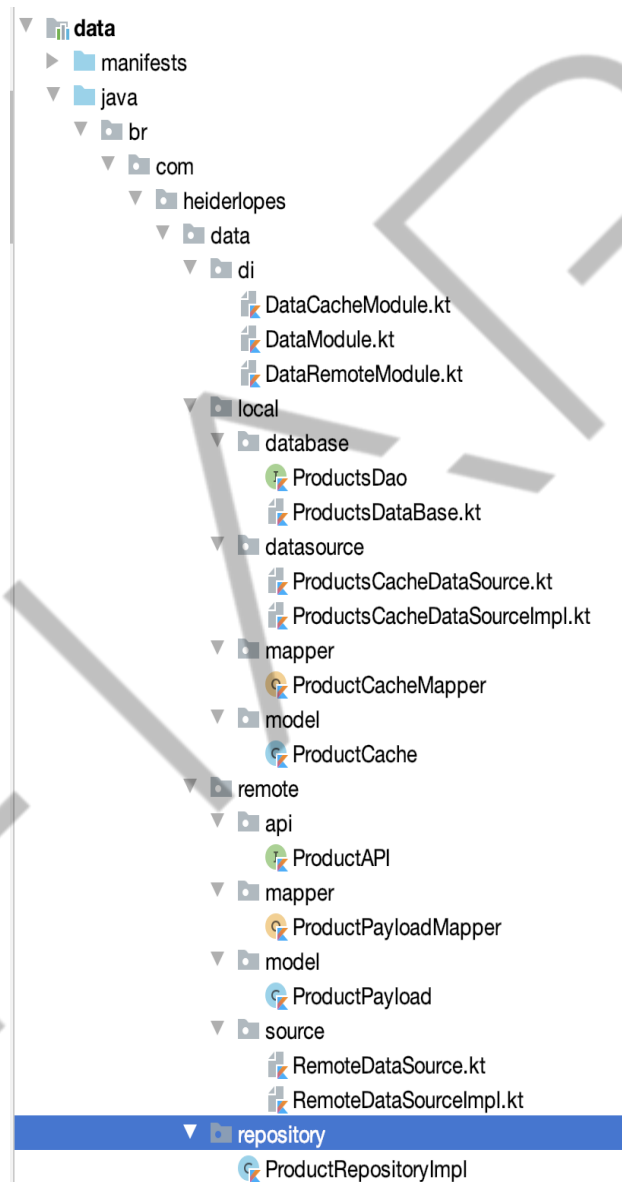


Figura 4.30 - Estrutura de pacote e classes do data module
Fonte: Elaborado pelo autor (2020)

4.2.11 Presentation Module

O Presentation Module é composto por:

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

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

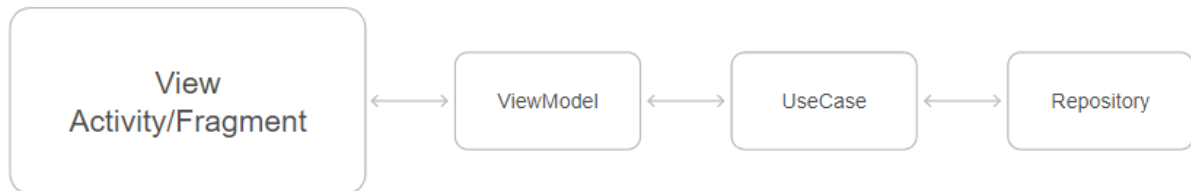


Figura 4.31 - Classe DataModule
Fonte: Elaborado pelo autor (2020)

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 ele será utilizado como presentation module.

Abra o arquivo **build.gradle** do módulo **app**, e adicione a 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.appCompact
    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
}
```

Código-fonte 4.24 — build.gradle do presentation module

Fonte: Elaborado pelo autor (2020)

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

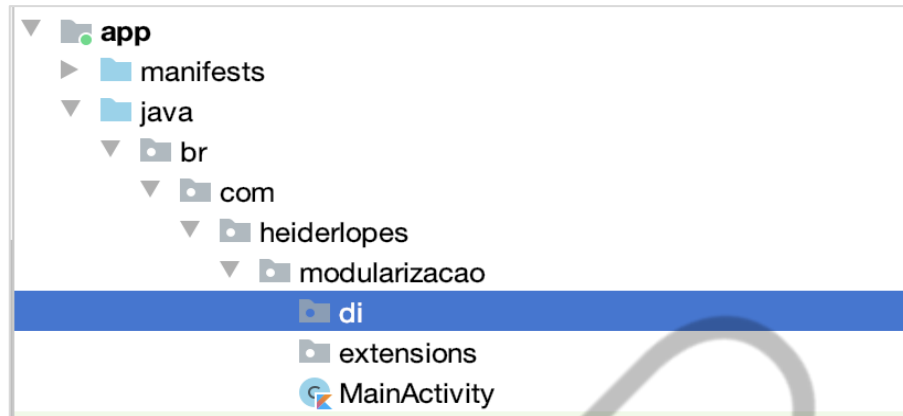


Figura 4.32 - Pacote di e extensions do presentation module
Fonte: Elaborado pelo autor (2020)

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

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

Código-fonte 4.25 – Classe de extensão Context.kt
Fonte: Elaborado pelo autor (2020)

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

Código-fonte 4.26 – Classe de extensão View.kt
Fonte: Elaborado pelo autor (2020)

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

Código-fonte 4.27 – Classe de extensão ViewGroup.kt
Fonte: Elaborado pelo 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()  
    }  
}
```

Código-fonte 4.28 – Classe BaseViewModel
Fonte: Elaborado pelo 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  
            }  
    }  
}
```

Código-fonte 4.29 – Classe ViewState
Fonte: Elaborado pelo 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 é nela também que se encontra 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 **Observeables** 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() nada mais 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, será 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 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
    }
)
}
```

Código-fonte 4.30 – Classe ViewState
Fonte: Elaborado pelo 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 fornecer, a nós, 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. Nesse 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 chegarem 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**.

4.2.12 Programando a classe principal

Agora abra o arquivo **MainActivity.kt**, e, 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")
            }
        }
    })
}
```

Código-fonte 4.31 — Classe MainActivity
Fonte: Elaborado pelo autor (2020)

Primeiro injete o **viewModel**. Em seguida, crie o método **setupViewModel**, no qual será chamada a **viewModel** para nos fornecer os dados e, na sequência, 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, caso algo dê errado, e esconde as outras views.

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

```
val presentationModule = module {

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

```
)  
}  
}
```

Código-fonte 4.32 – Classe Presentation Module
Fonte: Elaborado pelo autor (2020)

4.2.13 Inicializando o Koin

O Koin é uma biblioteca externa que nos ajuda a configurar dependência em nossos módulos de uma forma mais efetiva, que é chamado de **injeção de dependência**.

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))  
        }  
    }  
}
```

Código-fonte 4.33 – Classe MyApplication
Fonte: Elaborado pelo 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-filteraction android:name="android.intent.action.MAIN"/>

    <category android:name="android.intent.category.LAUNCHER"/>
  </intent-filteractivityapplication

```

Código-fonte 4.34 — AndroidManifest.xml
 Fonte: Elaborado pelo autor (2020)

4.2.14 Melhorando o visual das linhas da lista

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



Figura 4.33 - Item da lista de produtos
 Fonte: Elaborado pelo 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 4.35 — Item produtos da lista

Fonte: Elaborado pelo autor (2020)

Como a imagem será exibida por meio 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:$appCompactVersion",
    constraintlayout:
"androidx.constraintlayout:constraintlayout:$constraintVersion",
    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"
]
}

```

Código-fonte 4.36 — Inclusão da biblioteca Picasso no dependencies.gradle

Fonte: Elaborado pelo autor (2020)

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.reactivex
    implementation dependencies.rxkotlin
    implementation dependencies.rxandroid

    implementation dependencies.picasso

    kapt dependencies.databinding
}
```

Código-fonte 4.37 – Inclusão da biblioteca Picasso no módulo app
Fonte: Elaborado pelo autor (2020)

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 4.38 — Implementação do MainListAdapter
 Fonte: Elaborado pelo 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 4.39 — Injetando o adapter
 Fonte: Elaborado pelo autor (2020)

Altere o layout da **MainActivity.kt** para exibir a lista e 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 4.40 — Layout da MainActivity
Fonte: Elaborado pelo 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 4.41 — Habilitando o databinding
Fonte: Elaborado pelo autor (2020)

Feito isso, clique em **Build** e, 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 4.42 – MainActivity para exibir os dados
Fonte: Elaborado pelo autor (2020)

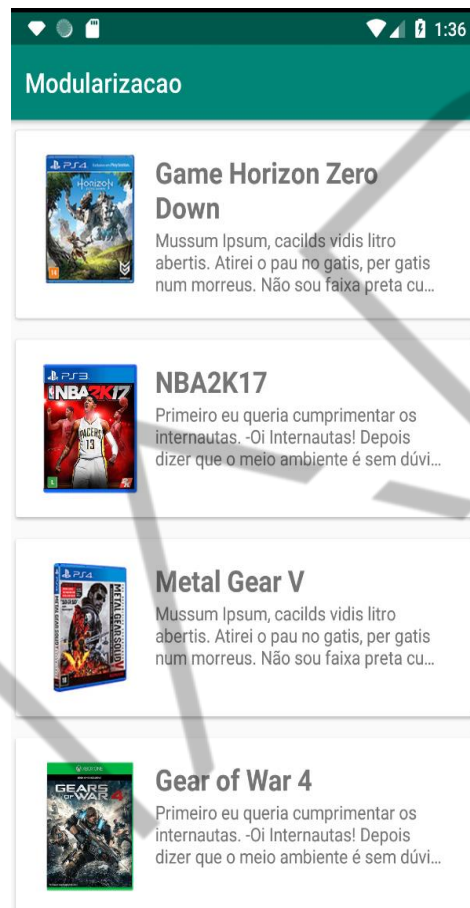


Figura 4.34 - Exibição dos dados no aplicativo
Fonte: Elaborado pelo autor (2020)

4.3 Modularização de recursos (features)

Para modularizar a aplicação por recursos, antes precisamos conhecer os tipos de módulos existentes no Android: módulos de biblioteca 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 de recursos 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.

4.3.1 Criando módulos dinâmicos

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

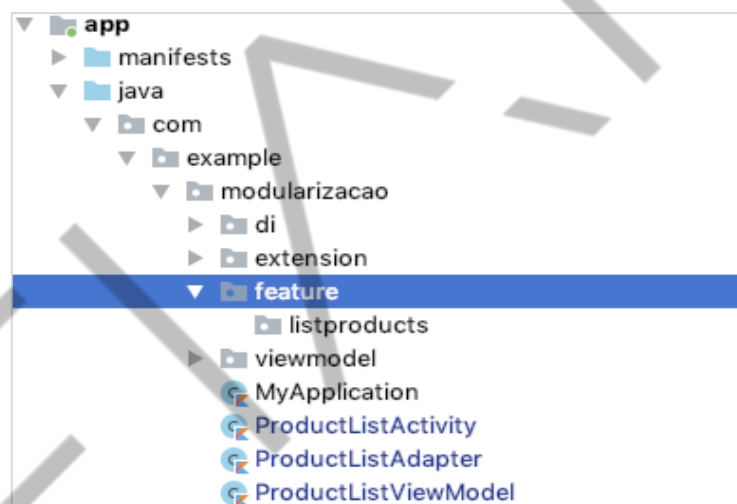


Figura 4.35 - Estrutura de pacotes com a modularização por funcionalidade
Fonte: Elaborado pelo autor (2020)

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

- MainActivity ⇒ ProductListActivity
- MainViewModel ⇒ ProductListViewModel
- MainListAdapter ⇒ ProductListAdapter
- activity_main.xml ⇒ activity_product_list

Após essas ações, clique em **Build** ⇒ **Rebuild project**

Mova os arquivos renomeados para dentro desta nova pasta:

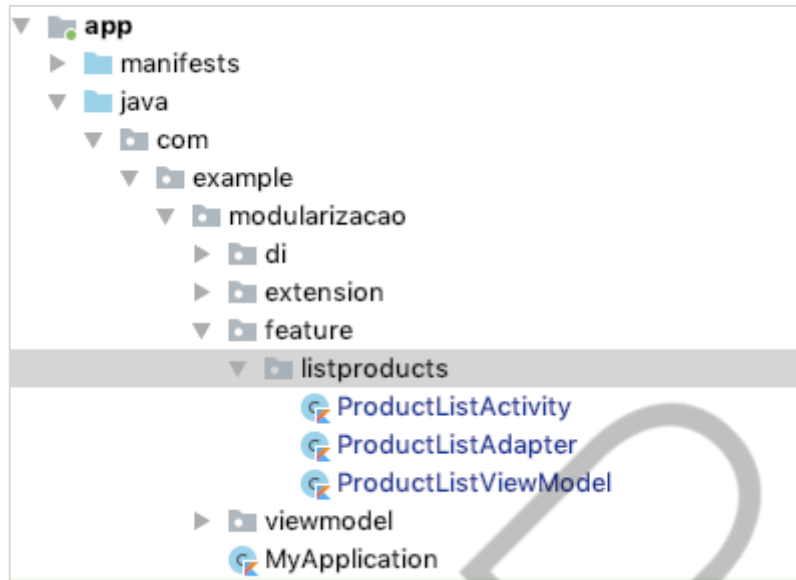


Figura 4.36 - Estrutura de pacotes com a modularização por funcionalidade com as classes
Fonte: Elaborado pelo autor (2020)

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

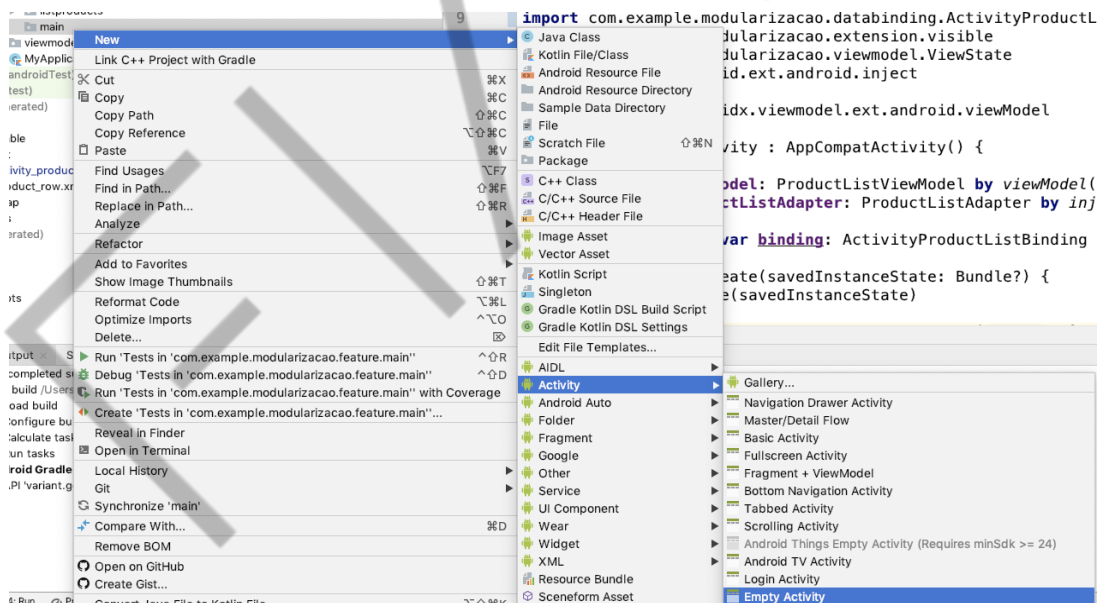


Figura 4.37 - Criando uma Empty Activity
Fonte: Elaborado pelo autor (2020)

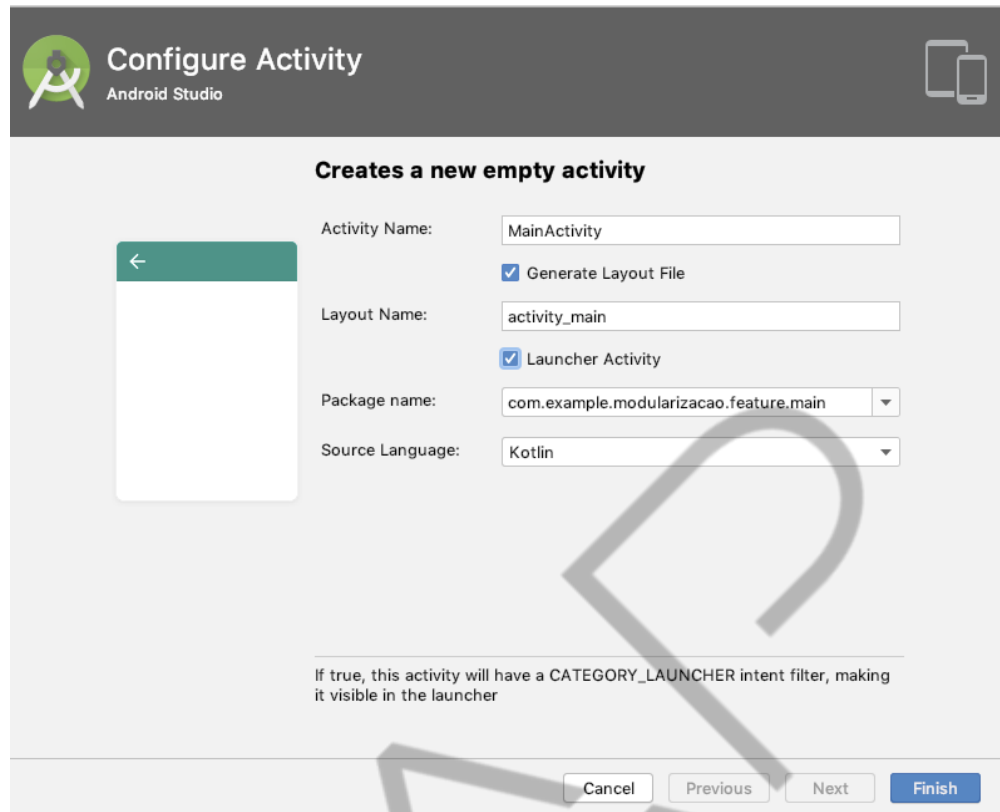


Figura 4.38 - Definindo o nome da EmptyActivity como MainActivity
 Fonte: Elaborado pelo autor (2020)

Abra o arquivo **AndroidManifest.xml** e certifique-se de que somente a **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" />

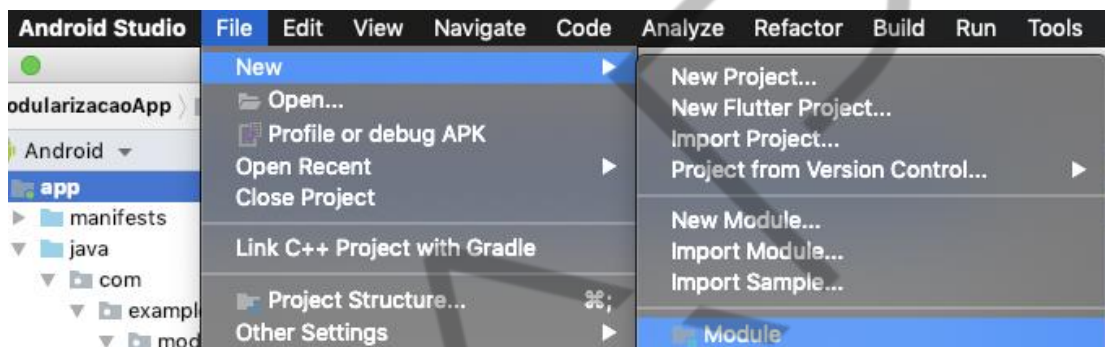
    <application
        android:name=".MyApplication"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <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>
    </application>
</manifest>
```

```
</activity>  
<activity android:name=".feature.listproducts.ProductListActivity">  
</activity>  
</application>  
  
</manifest>
```

Código-fonte 4.43 – Definindo a MainActivity como a principal
Fonte: Elaborado pelo autor (2020)

Agora crie um novo módulo:



Código-fonte 4.44 – - Criando um novo módulo
Fonte: Elaborado pelo autor (2020)

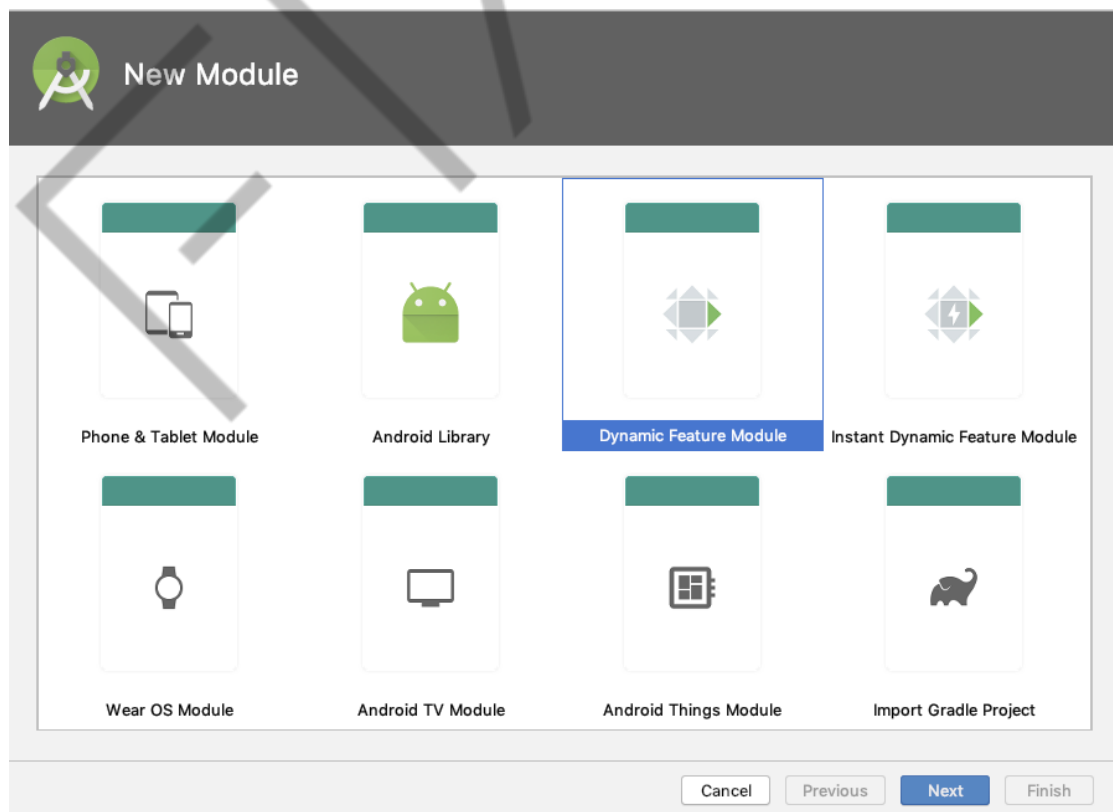


Figura 4.39 - Definindo o módulo com Dynamic Feature Module
Fonte: Elaborado pelo autor (2020)

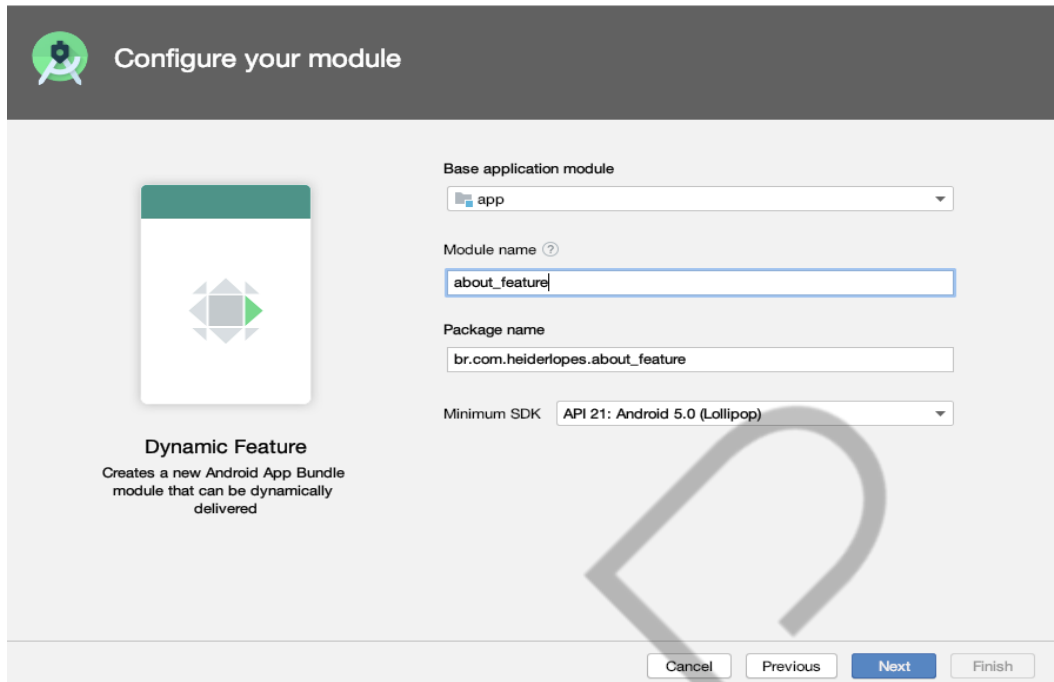


Figura 4.40 - Definindo o nome do módulo
Fonte: Elaborado pelo autor (2020)

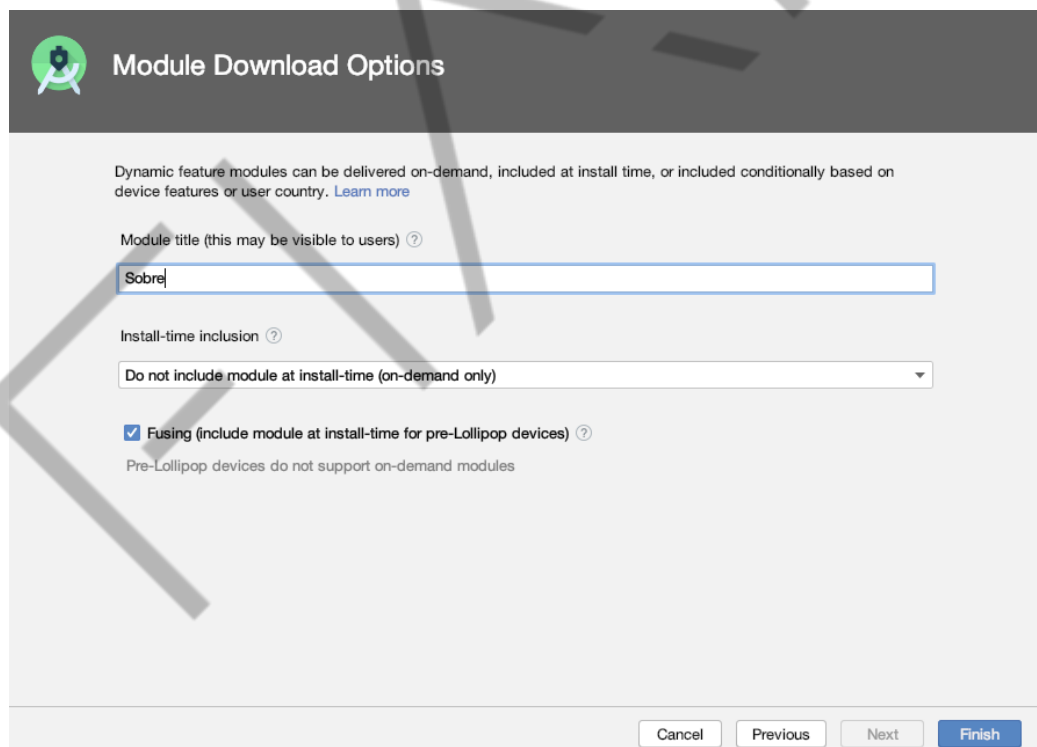


Figura 4.41 - Definindo o nome visível para o usuário
Fonte: Elaborado pelo autor (2020)

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>
```

Código-fonte 4.45 — Layout da tela principal
Fonte: Elaborado pelo autor (2020)

Na tela principal existem 4 botões dos quais 2 deles 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 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"]
```

Código-fonte 4.46 – Layout da tela principal
Fonte: Elaborado pelo 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')  
}
```

Código-fonte 4.47 – Adição do módulo do aplicativo
Fonte: Elaborado pelo 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'
```

Código-fonte 4.48 – Aplicação do plugin dynamic feature
Fonte: Elaborado pelo 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>
```

Código-fonte 4.49 — AndroidManifest.xml do módulo about.

Fonte: Elaborado pelo 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 4.50 — Adição da biblioteca para distribuição sob demanda

Fonte: Elaborado pelo 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 por 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**:

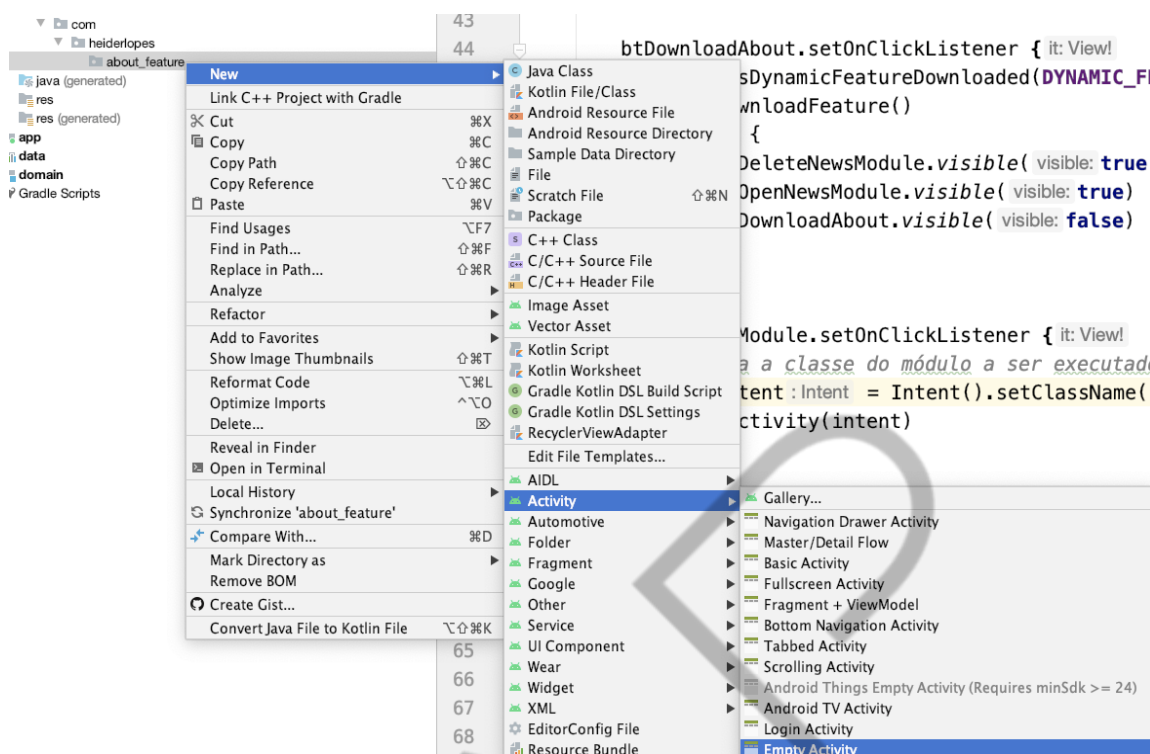


Figura 4.42 - Criação da Activity Sobre
Fonte: Elaborado pelo autor (2020)

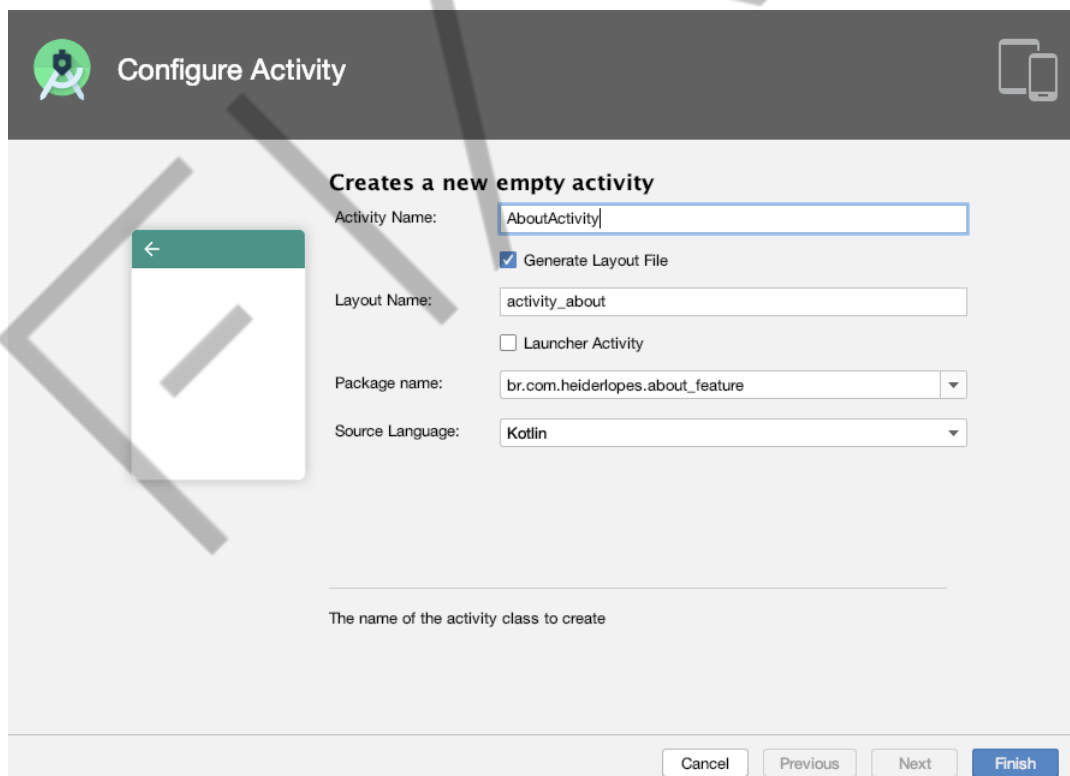


Figura 4.43 - Criação da Activity Sobre
Fonte: Elaborado pelo autor (2020)

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>
```

Código-fonte 4.51 – Layout da tela Sobre

Fonte: Elaborado pelo autor (2020)

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 4.52 – Declaração de variáveis
Fonte: Elaborado pelo 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.

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 4.53 – Declaração de variáveis
Fonte: Elaborado pelo autor (2020)

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

É possível adicionar um ou vários módulos com o uso 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 {
        }
}
}
```

Código-fonte 4.54 – MainActivity com gerenciamento de módulos

Fonte: Elaborado pelo autor (2020)

Seguem também as imagens do aplicativo:



Figura 4.44 - Tela do aplicativo para baixar o módulo Sobre
Fonte: Elaborado pelo autor (2020)



Figura 4.45 - Tela do aplicativo com módulo Sobre baixado

Fonte: Elaborado pelo autor (2020)



Figura 4.46 - Tela Sobre do aplicativo

Fonte: Elaborado pelo autor (2020)

Para eventuais consultas, as duas formas de modularizar uma aplicação você encontra disponíveis neste repositório: [<https://github.com/FIAPON/AppModularizadoAndroid/>](https://github.com/FIAPON/AppModularizadoAndroid/).

CONCLUSÃO

Neste capítulo, foi possível conhecer um aplicativo modular e também como criá-lo. Com esse conhecimento será possível construir aplicativos cada vez menos acoplados, mais testáveis, APK menores e com builds mais rápidos.

EXEMPLO

REFERÊNCIAS

DEVELOPERS, Google. **Sobre o Dynamic Delivery**. 2020. Disponível em: <<https://developer.android.com/studio/projects/dynamic-delivery?hl=pt-br>>. Acesso em: 14 set. 2020.

FUCOLO, I. M. **Modularização Android**. 2019. Disponível em: <<https://medium.com/android-dev-br/modulariza%C3%A7%C3%A3o-android-parte-1-b69b509571c9>>. Acesso em: 14 set. 2020.

KOIN. **Documentação**. 2020. Disponível em: <<https://insert-koin.io/>>. Acesso em: 14 set. 2020.