# Apostila de Desenvolvimento Mobile - Criando um aplicativo de agenda de contatos com Android Jetpack

Prof. Bruno Crestani Calegaro

**Coordenação Gráfica**
Bruno Crestani Calegaro

**Capa, Projeto gráfico e Diagramação**
Bruno Crestani Calegaro

**Revisão Textual**
Bruno Crestani Calegaro
Diovane da Silva

## SUMÁRIO

# 1. Introdução

Este material é destinado a alunos cursando disciplinas de Programação para Dispositivos Móveis. A apostila tem o objetivo de ensinar programação para Android através do estudo prático de um aplicativo de Agenda de Contatos. O aplicativo será desenvolvido usando os recursos mais recentes do Android Jetpack, resultando em um aplicativo robusto, moderno e eficiente.

Todo o processo de construção do aplicativo será ilustrado e a cada novo passo será explicado os recursos envolvidos em tal operação. Ao longo da apostila diversos assuntos da programação para Android serão abordados como: atividades, ciclos de vida, fragmentos, elementos de interface gráfica, biblioteca Room, padrão MVVM, entre outros.

Acompanhe o desenvolvimento do aplicativo do início ao fim seguindo os passos da apostila. Ao final você terá desenvolvido um aplicativo de Agenda de Contatos funcional, com persistência de dados usando o banco de dados SQLite, múltiplas telas para criação, edição e visualização dos contatos.

Por questões de organização o conteúdo da apostila é organizado em três partes:

- Parte 1: Desenvolvimento da Interface Gráfica
- Parte 2: Transição entre telas e apresentação de dados fictícios
- Parte 3: Integração com banco de dados com a biblioteca Room

# Parte 1 - Criação da Interface Gráfica do Usuário

## 2. Apresentação

Na parte 1 vamos iniciar o desenvolvimento do aplicativo Agenda de Contatos com a construção da Interface Gráfica do Usuário. Neste material vamos conhecer componentes avançados para a construção de aplicativos usando o **Android Jetpack**. Também vamos fazer uma preparação inicial dos componentes de arquitetura (*Android Arch Components*) do Android Jetpack que serão usados no projeto, bem como a biblioteca **Room** para a manipulação do banco de dados SQLite da **Agenda de Contatos**.

O aplicativo **Agenda de Contatos** fornece acesso a informações de contatos armazenadas em um banco de dados SQLite no dispositivo. No aplicativo podemos:

- ver uma lista em ordem alfabética dos contatos
- ver os detalhes de um contato tocando no nome dele na lista de contatos
- adicionar novos contatos
- editar ou excluir contatos existentes

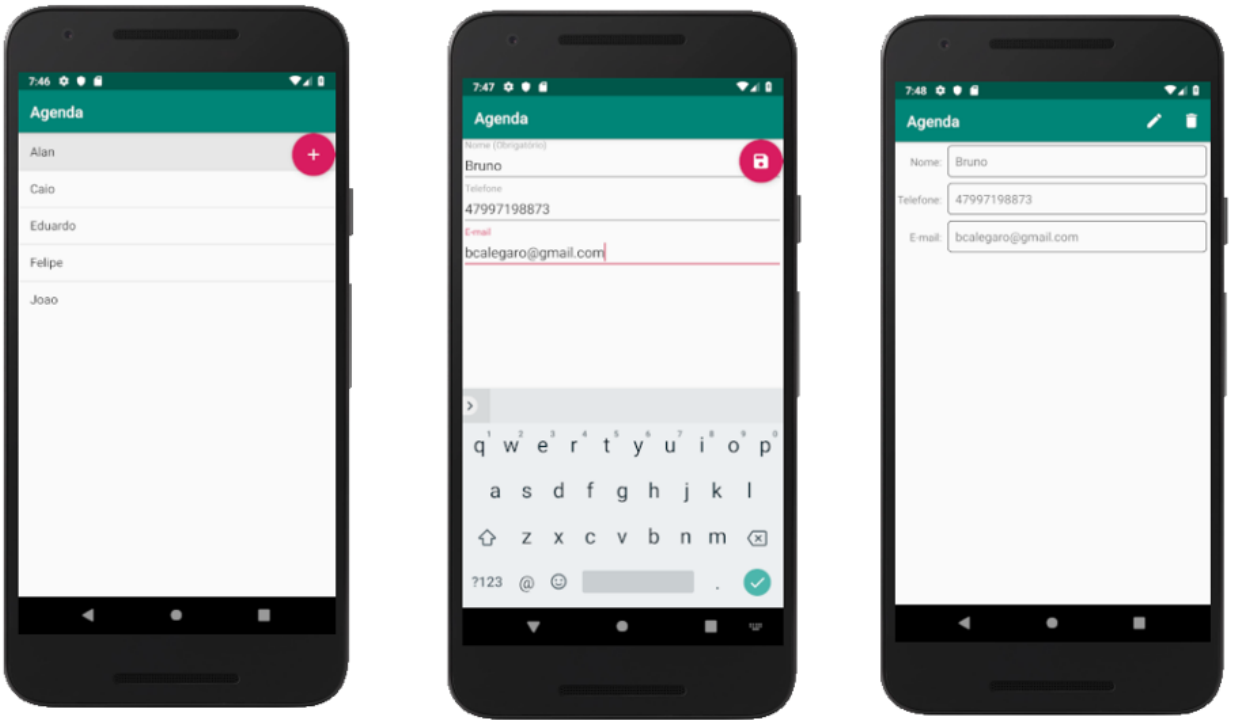


Figura 1. Aplicativo Agenda de Contatos e suas múltiplas telas

Devido a complexidade desse projeto e a quantidade de recursos novos a serem aprendidos a apostila é organizada em três partes. Sendo a parte 1 a criação da interface gráfica com **Fragmentos**, **RecyclerView**, **Material Icons** e **Menus.**

# 3. Recursos Envolvidos

Nesse projeto usaremos os seguintes recursos na construção do aplicativo **Agenda de Contatos**:

- **Fragments**
  - *Fragments* representam partes reutilizáveis para a criação de interfaces gráficas de uma Activity e também podem conter lógica de programação.
  - Originalmente, *fragments* foram criadas para construir interfaces mais elegantes para os tablets. Por exemplo, dado um aplicativo de celular, o mesmo poderia ser construído fazendo o uso de duas telas, logo com a interação do usuário essas telas seriam sobrepostas quando necessário. No entanto, em um tablet essa construção parece estranha pois havia muito espaço sobrando na tela do dispositivo. Assim, com o uso de fragmentos é possível construir uma única tela no tablet que incorpore essas duas telas menores de um celular, de uma única vez.


Figura 2 - Representação de Fragments em um Tablet e um Smartphone

Fonte https://developer.android.com/training/basics/fragments/fragment-ui

- Note que ambas as versões acima, apenas uma Activity estará rodando. No entanto, a versão do tablet faz o uso de dois *fragments* ao mesmo tempo e na versão do celular cada Activity usa apenas um *fragment*.
- No projeto **Agenda de Contatos** usaremos *fragments* para criar nosso aplicativo com mais de uma tela. Serão criados três fragmentos:
  - **ContactsFragment** – fragmento para exibir a lista de contatos através de uma RecyclerView.
  - **DetailFragment** – fragmento para exibir as informações do contato selecionado.
  - **AddEditFragment** – fragmento para adicionar um novo contato ou editar um contato já existente.

- **Ciclo de Vida**
  - Similar a uma Activity, um fragmento também possui ciclo de vida. Neste projeto daremos atenção aos métodos onCreate e onCreateView.
    - **onCreate** – método invocado quando um fragmento é criado.
    - **onCreateView** – método invocado após a criação do fragmento para retornar uma *View* contendo a definição da interface gráfica do fragmento.
    - **onActivityCreated** – método invocado após a criação da atividade do fragmento. Nesse local iremos fazer a vinculação dos dados da *ViewModel* aos elementos da tela do fragmento.
- **Navegação entre Fragmentos**
  - Neste projeto vamos fazer o uso dos componentes de arquitetura disponíveis no **Android Jetpack** (pacote **androidx**). Entre eles está o novo componente de navegação, **Navigation**, que oferece uma solução moderna e simples de se implementar, desde a configuração de gráfico de transições para a programação das transições.
  - A navegação se refere às interações que permitem aos usuários navegar, entrar e sair de diferentes partes do conteúdo no aplicativo. O componente de navegação do **Android Jetpack** ajuda a implementar a navegação, desde simples cliques em botões até padrões mais complexos, como barras de aplicativos e a gaveta de navegação.

- **RecyclerView, RecyclerViewAdapter e padrão ViewHolder**
  - Este aplicativo exibe uma lista de contatos com um componente *RecyclerView* (pacote **androidx.recyclerview**) – uma lista rolante de itens reutilizáveis. Para preencher essa lista vamos precisar criar uma subclasse de **RecyclerViewAdapter**, cuja finalidade é preencher o elemento *RecyclerView* utilizando dados de um objeto ArrayList.
  - Quando o aplicativo atualiza os dados da lista de contatos, ele chama o método **notifyDataSetChanged** do *RecyclerViewAdapter* para indicar que os dados mudaram. Então, o adaptador notificará o componente *RecyclerView* para que atualize sua lista de itens exibidos. Isso é conhecido como **vinculação de dados** (*data binding*).
  - Cada item adicionado à lista envolve a execução do processo de criar novos objetos dinamicamente. Para listas grandes, nos quais o usuário rola rapidamente, a quantidade de itens gera uma sobrecarga que pode impedir uma rolagem suave. Logo, para reduzir essa sobrecarga, quando os itens do componente RecyclerView rolam para fora da tela, vamos fazer a reutilização desses itens de lista para os novos que estão entrando na tela. Para isso, usamos o padrão **ViewHolder**, no qual criamos uma classe (normalmente chamada *ViewHolder*) para conter variáveis de instância para as views que exibem os dados dos itens na *RecyclerView*.
- **TextInputLayout**
  - Os componentes *EditText* são usados para criar caixas de texto de modo que o usuário possa digitar algum conteúdo. Para ajudar o usuário a entender a finalidade da caixa de texto, podemos especificar a propriedade *hint* (dica) para esse elemento. Isso mostrará uma mensagem dentro da caixa de texto que desaparecerá assim que o usuário começar a digitar o texto. Por causa disso, pode acontecer do usuário esquecer da finalidade do elemento *EditText*, uma vez que a dica não aparecerá novamente. Para evitar isso, usamos **TextInputLayout** (pacote **com.google.android.material.textfield.TextInputLayout**) da *Android Design Support Library*.
  - Em um *TextInputLayout*, quando o elemento *EditText* recebe foco, o *TextInputLayout* anima o texto da dica, mudando seu tamanho original para um menor e exibindo-o acima da caixa de texto. Desse modo, o usuário pode digitar os dados e ver a dica ao mesmo tempo.
- **FloatingActionButton**
  - Os botões de ação flutuantes, comumente chamados de **FAB** (*FloatingActionButton*), foram introduzidos pelo Material Design e são botões que "flutuam", isto é, possuem elevação maior que o restante dos elementos da interface gráfica do aplicativo. A partir do Android 6.0, esse componente passou a ser disponibilizado pela *Android Design Support Library*.
  - É comum usar esse botão para uma ação única, mas importante para o aplicativo. Alguns exemplos de seu uso: enviar um email (Gmail) e, no caso da **Agenda de Contatos**, ir para a tela de adição de um novo contato.
  - *FloatingActionButton* é uma subclasse de *ImageView*, portanto, é possível usar esse botão para exibir uma imagem. Dessa forma, podemos usar ícones do Material Design nos **FAB 's.**
  - As diretrizes do Material Design sugerem posicionar esses botões a pelo menos 16 dp das margens do celular e a pelo menos 24 dp das margens em um tablet. Naturalmente, o Android Studio configura esses valores padrões aos botões do

projeto, mas nada impede deles serem modificados.

- **Material Design Icons**
  - O Android Studio oferece uma ferramenta chamada **Vector Asset** para adicionar ao projeto os ícones disponíveis do *Material Design*. A lista de ícones é grande e abrange desde ícones de navegação, para ações, acessibilidade, arquivos, play e muitos outros.
  - Esses ícones são imagens vetoriais capazes de serem redimensionadas para qualquer resolução sem perder a qualidade pois não são exatamente imagens, mas sim definições de formas. Eles também podem ser configurados para diferentes cores.
- **Menus**
  - Menus podem ser adicionados em atividades ou fragmentos. Eles são incluídos na barra do aplicativo (AppBar) e podem assumir forma de um ícone com uma lista de opções (menu) ou oferecer diretamente ícones para realizar as ações. Esse ícone fica normalmente no lado direito da barra e usa um arquivo do tipo *menu.xml* para definir seu comportamento.
  - Neste projeto iremos configurar um Menu com duas opções: editar e excluir um contato.
- **Diretório RES**
  - Na pasta RES de um projeto Android, encontram-se todos os recursos que o aplicativo irá usar: imagens, textos, cores, dimensões, entre outros. Neste aplicativo vamos operar sobre a pasta drawable para organizar as imagens dos ícones e a pasta values com os arquivos de recursos string. Além disso, vamos usar a pasta menu, que contém os arquivos .xml que definem o conteúdo dos menus da aplicação.
- **Android Jetpack**
  - O Android Jetpack é uma **coleção** de componentes de software Android que buscam facilitar o desenvolvimento de excelentes aplicativos Android. Esses componentes ajudam você a seguir as práticas recomendadas, acabando com os códigos clichê e simplificando tarefas complexas, para que você possa se concentrar na parte do código do seu interesse.
  - O Jetpack é composto por bibliotecas de pacotes do "**androidx.***", separadas das APIs da plataforma. Isso significa que ele oferece retrocompatibilidade e é atualizado com mais frequência do que a plataforma Android, garantindo que você sempre tenha acesso às versões mais recentes dos componentes do Jetpack.
  - Nesse projeto iremos usar uma série de componentes de arquitetura (*Android Arch Components*):
  - **Fragment**: uma unidade básica de interface gráfica combinável
  - **ViewModel**: gerencia os dados para interface gráfica considerando o ciclo de vida
  - **LifeCycles**: para gerenciar os ciclos de vida de atividade e fragmentos
  - **LiveData**: para manipular as informações do banco de dados
  - **Room**: biblioteca para facilitar o acesso ao banco de dados SQLite
- Nas próximas partes do projeto será documentado de maneira mais profunda os recursos envolvidos em cada parte.

# 4. Criando o projeto

Para a construção deste projeto devemos fazer o uso da **template "Fragment + ViewModel"** para a criação da atividade principal e os fragmentos. Dessa forma, siga os passos abaixo para a criação do projeto base da Agenda de Contatos.

- Abra o Android Studio e clique na opção **Start a new Android Studio Project** na tela de abertura ou, caso já exista um projeto em aberto, clique em **File > New > New Project**
- Na primeira janela escolha a template "**Fragment + ViewModel**":

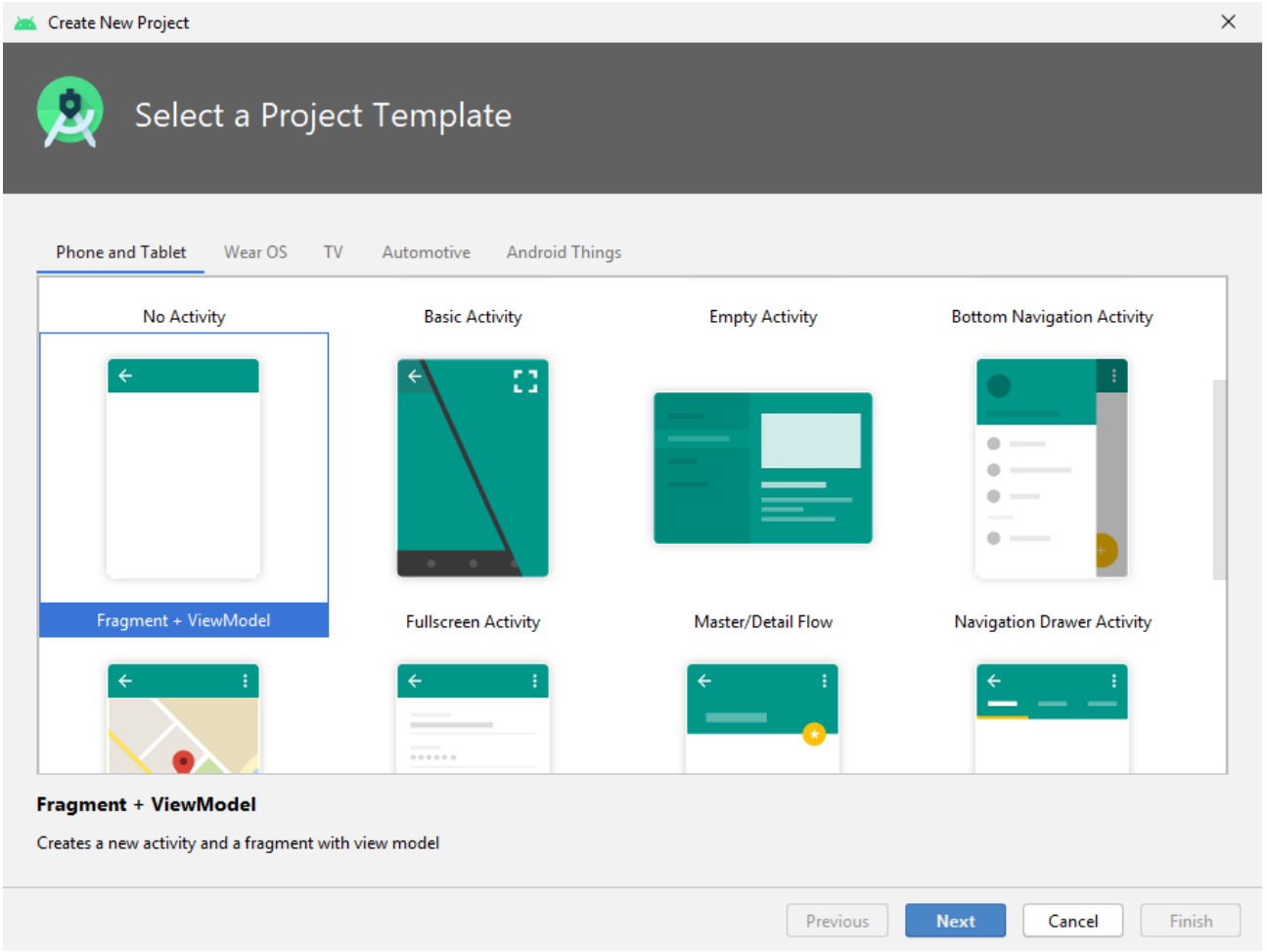


Figura 3 - Selecionando a template Fragment + ViewModel

- Na próxima janela configure o nome do projeto como **Agenda de Contatos** e modifique o local do projeto se achar necessário. Selecione a linguagem de programação Java e API 23. Clique em **Finish**.

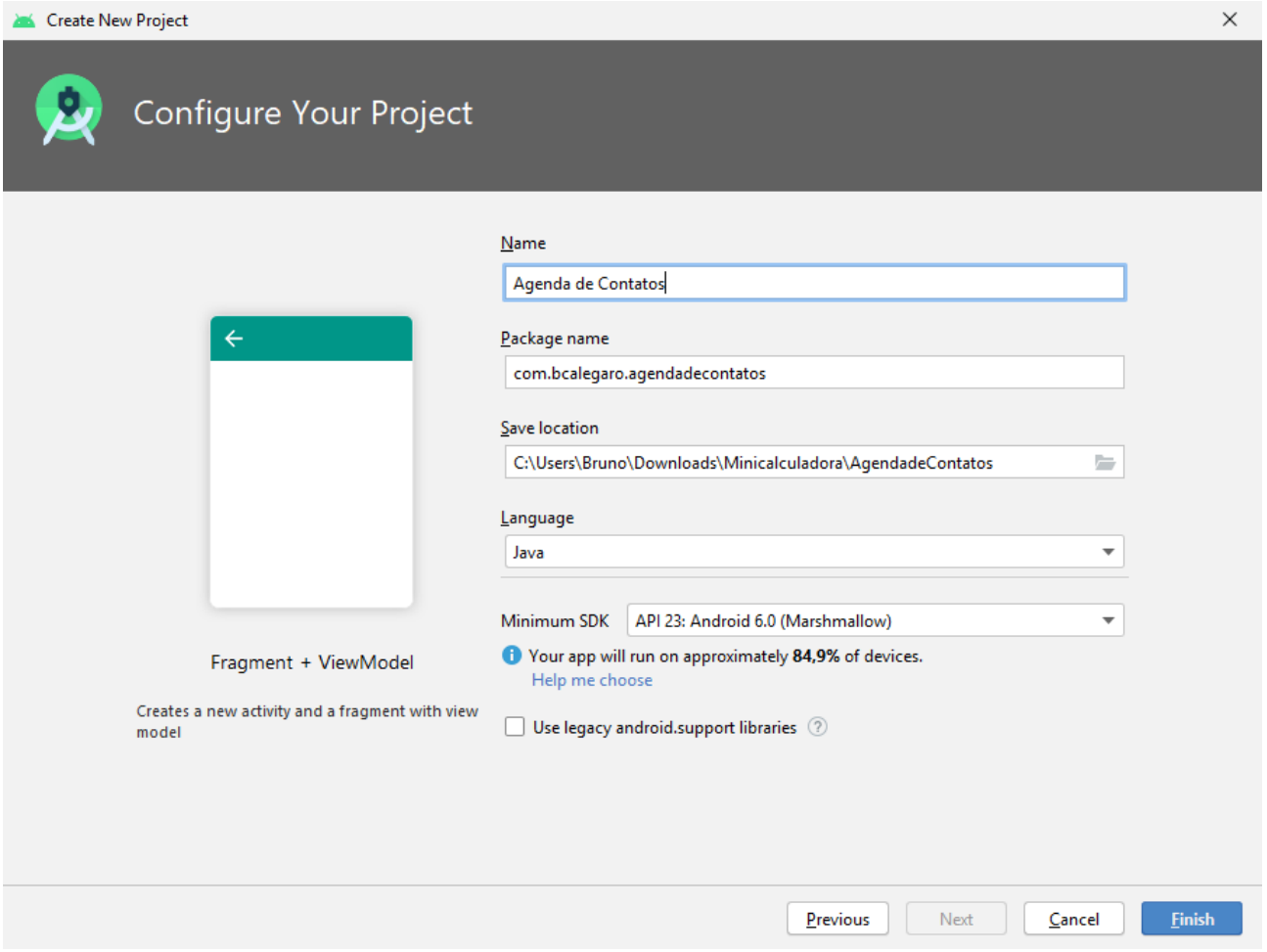


Figura 4 - Criando o projeto Agenda de Contatos

# 5. Configurações Iniciais do Projeto

Nesta primeira parte do projeto vamos construir o desenho da interface gráfica e a configuração dos recursos necessários para a construção do aplicativo. Vamos realizar as seguintes etapas:

- Configuração do Android JetPack
- Adicionar novas bibliotecas (RecyclerView, Room, etc) ao projeto
- Criação das classes
- Configuração de ícones, recursos strings e estilos de borda

## 5.1. Configurando o projeto para o Android Jetpack

Primeiramente, vamos configurar o nosso projeto para fazer o uso dos componentes presentes no **Android Jetpack**. Esses componentes estão presentes no pacote **androidx** e devem ser adicionados ao projeto através da configuração do Gradle.

**Nesta última versão do Android Studio (4.x) o projeto criado já está fazendo o uso do Android JetPack, portanto os próximos passos podem ser pulados. Para o caso de você ter um projeto antigo e queira migrar para o Android Jetpack considere os passos abaixo.**

No Android Studio existe uma ferramenta para fazer migração automática de projetos já existentes ao novo Android Jetpack. Essa ferramenta corrige todas as dependências para fazer o uso novo pacote **androidx** e adiciona novas configurações ao projeto. Dessa forma, para usar a ferramenta siga os passos abaixo:

- No menu Refactor e escolha a opção "Migrate to AndroidX"

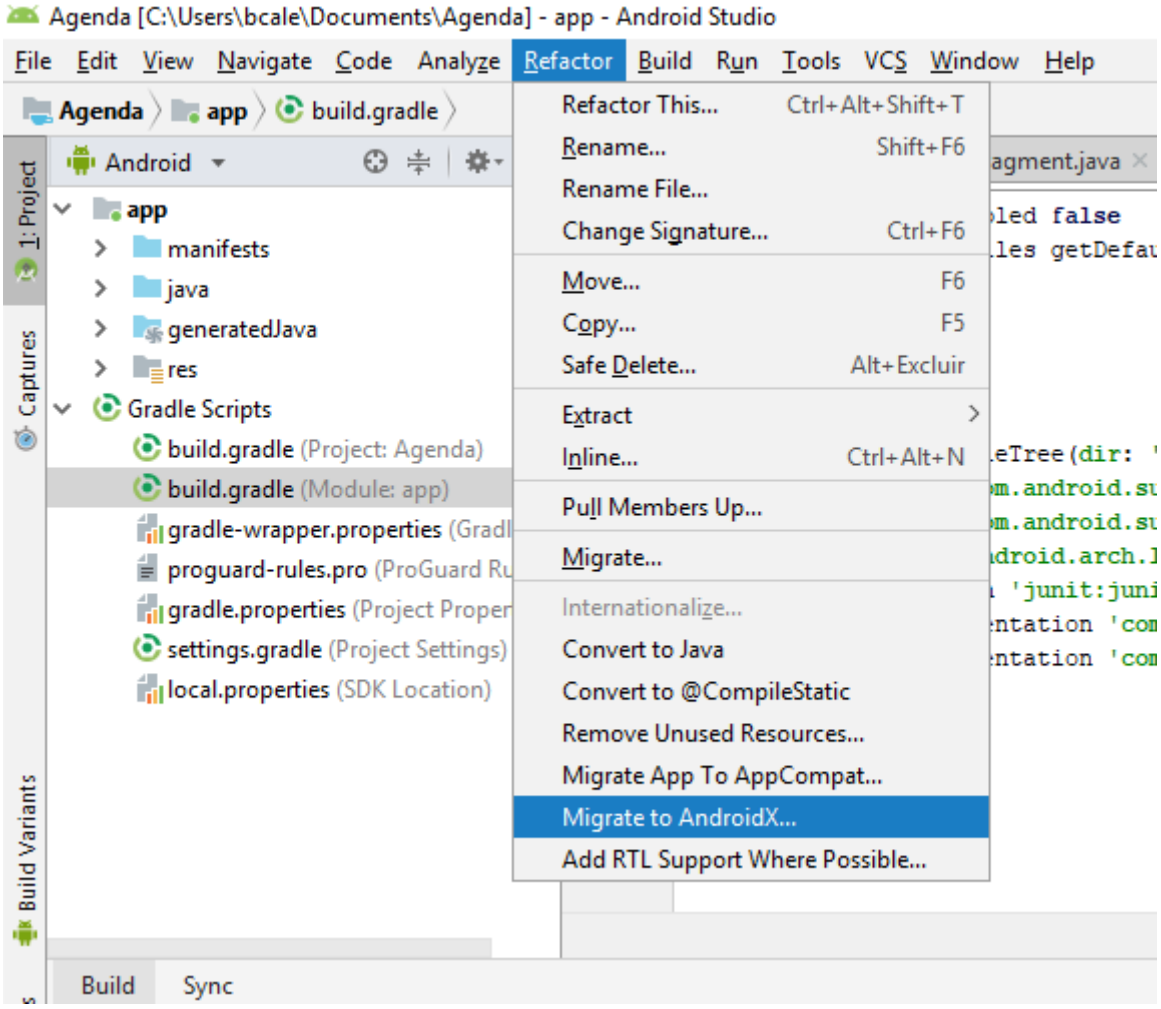


Figura 5 - Selecionando a opção de migração automática para o AndroidX

- Na Janela de Notificação "*Migrate do AndroidX*", desmarque a opção de backup e confirme migração clicando no botão "*Migrate*"
    - Como recém criamos o projeto não há necessidade de fazer uma cópia de segurança. A situação seria diferente se você deseja-se migrar um projeto completo já existente pois a migração pode acarretar erros e estragar o projeto.

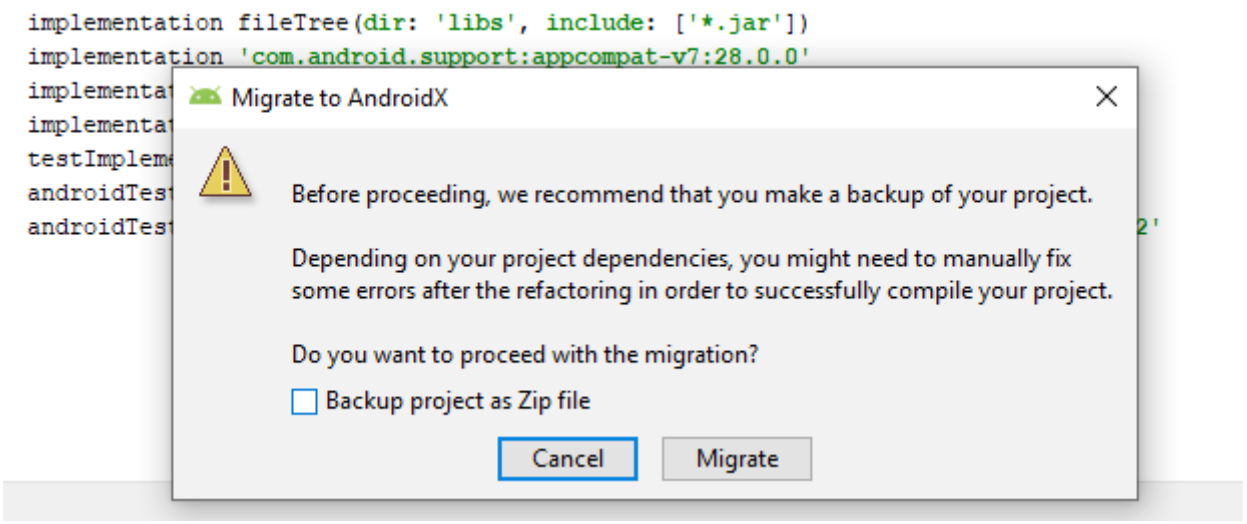


Figura 6 - Janela de confirmação antes da migração para o AndroidX

- Neste momento o Android Studio começa a verificar as alterações a serem feitas e exibe um relatório nas abas inferiores. Confirme a migração clicando em "*Do Refactor*"

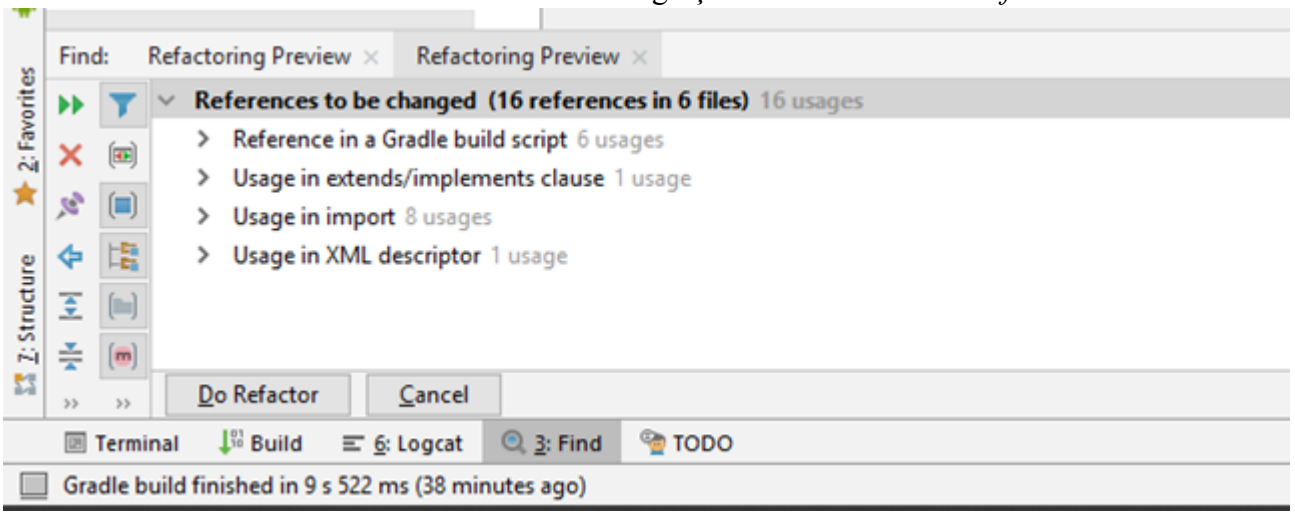


Figura 7 - Após a análise do projeto é necessário confirmar as alterações proposto com o botão "Do Refactor"

- Pronto. Seu projeto está configurado e fazendo o uso dos componentes do **Android Jetpack**. Você pode conferir as alterações inspecionado o conteúdo do arquivo localizado em *Gradle Script -> build.gradle (Module: app)*.

## 5.2. Adicionando novas bibliotecas ao projeto com o Gradle

Nesta etapa vamos configurar o nosso projeto para incorporar as novas bibliotecas a serem utilizadas. No caso do projeto **Agenda de Contatos** vamos usar as bibliotecas: **RecyclerView**, **Room** e **LifeCycle**. Abra o arquivo localizado em *Gradle Scripts -> build.gradle (Module: app)* para fazer a configuração. Edite no arquivo o campos *dependencies* com os valores abaixo:

Observação: essa etapa pode ser pulada pois durante a programação o Android Studio é capaz de completar automaticamente as dependências que faltam. De todo modo, assim é como

deve ficar a versão final do arquivo.

```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.3.0'
    implementation 'com.google.android.material:material:1.3.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
    implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.3.1'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
    implementation 'androidx.gridlayout:gridlayout:1.0.0'
    implementation 'androidx.navigation:navigation-fragment:2.3.4'
    implementation 'androidx.navigation:navigation-ui:2.3.4'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'

    implementation "androidx.room:room-runtime:2.2.6"
    annotationProcessor "androidx.room:room-compiler:2.2.6"
    testImplementation "androidx.room:room-testing:2.2.6"
}
```

## 5.3. Criação das classes do aplicativo

Nesta etapa vamos preparar as classes para a construção do aplicativo. A template *Fragment + ViewModel* cria três classes durante a inicialização do projeto: *MainActivity*, *MainActivityFragment* e *MainActivityViewModel*. Se você realizou as etapas conforme solicitado no **Capítulo 3** o seu projeto deve possuir essas três classes como mostra a figura:

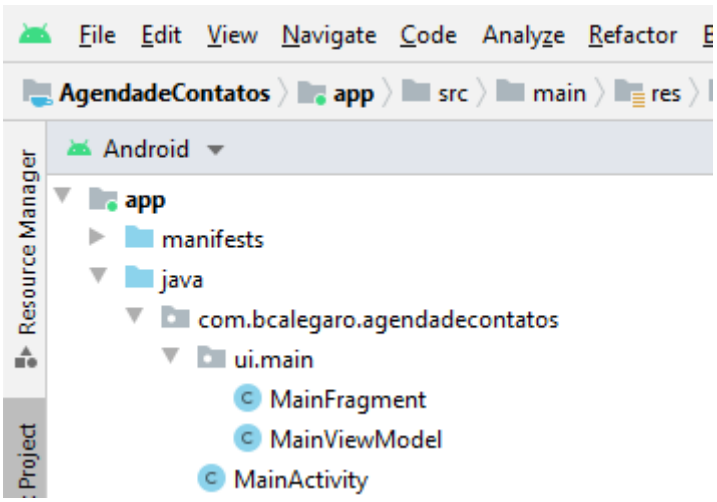


Figura 8 - Classes criadas durante a inicialização do projeto

Naturalmente, podemos associar cada fragmento a uma atividade separada, mas neste projeto vamos usar apenas uma atividade, e adicionar ou remover dinamicamente fragmentos nela. Dessa forma, siga os passos abaixo para a criação inicial dos dois fragmentos restantes:

- Adicione dois novos fragmentos ao pacote "**ui.main**" do projeto. Para adicionar use o menu de contexto sobre o pacote e selecione **New -> Fragment -> Fragment (with View Model)**

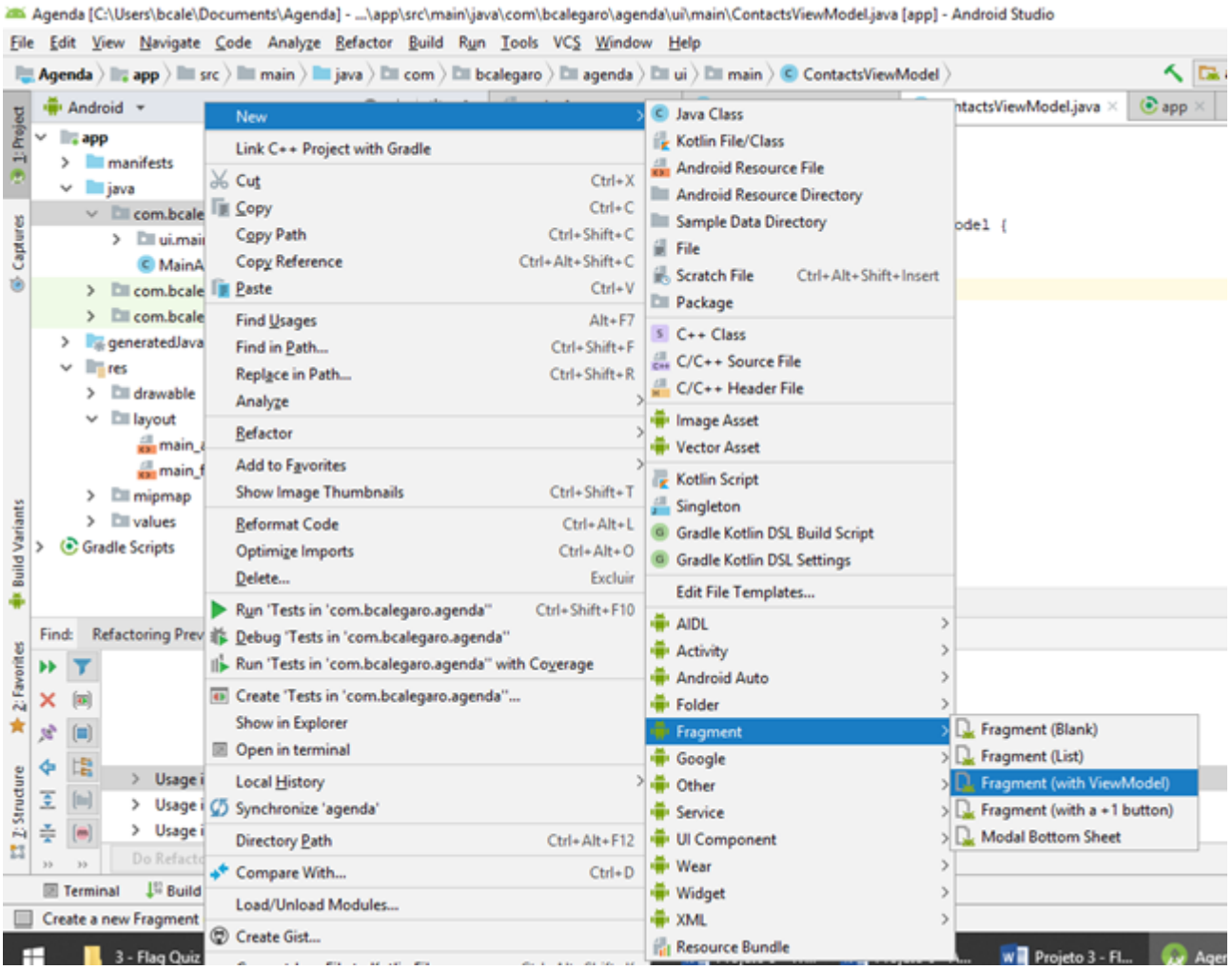


Figura 9 - Menu de contexto para inserir um novo fragmento com a template Fragment + ViewModel

- Na janela "*New Android Component*" modifique o campo *Fragment Name* para **AddEditFragment**. Clique em **Finish.**
    - Este fragmento será usado para criar a interface para adicionar um novo contato ou editar um já existente

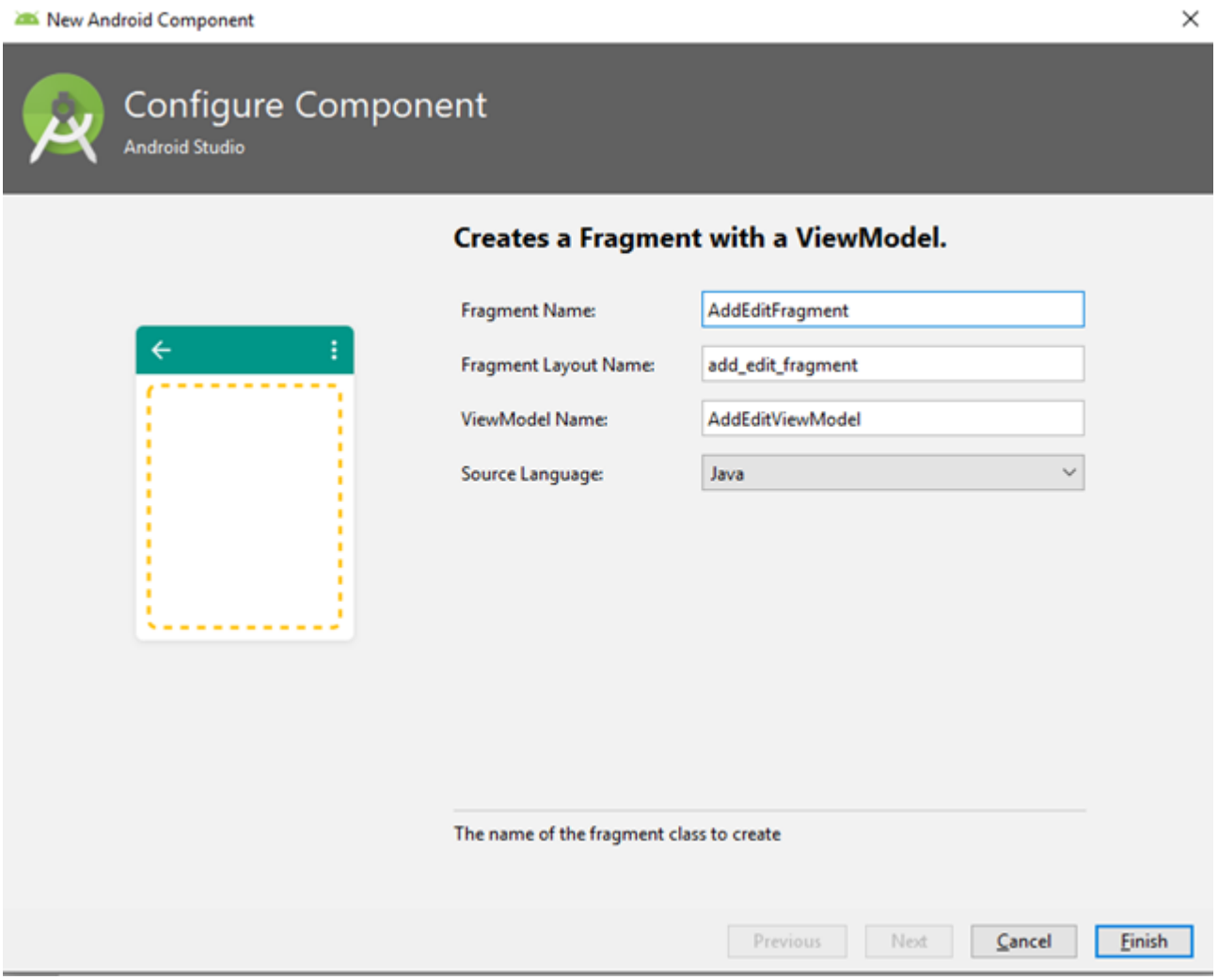


Figura 10 - Janela de criação do novo fragmento

- Repita os passos acima e crie o **DetailFragment**
  - Este fragmento será usado para exibir os dados de um contato e fornecer itens de menu para editar ou excluir o contato.
- Adicione também as seguintes classes para a criação da lista de contatos, para adicionar uma nova classe clique com o botão direito do mouse sobre a pasta **ui.main** e selecione **New > Java Class**.

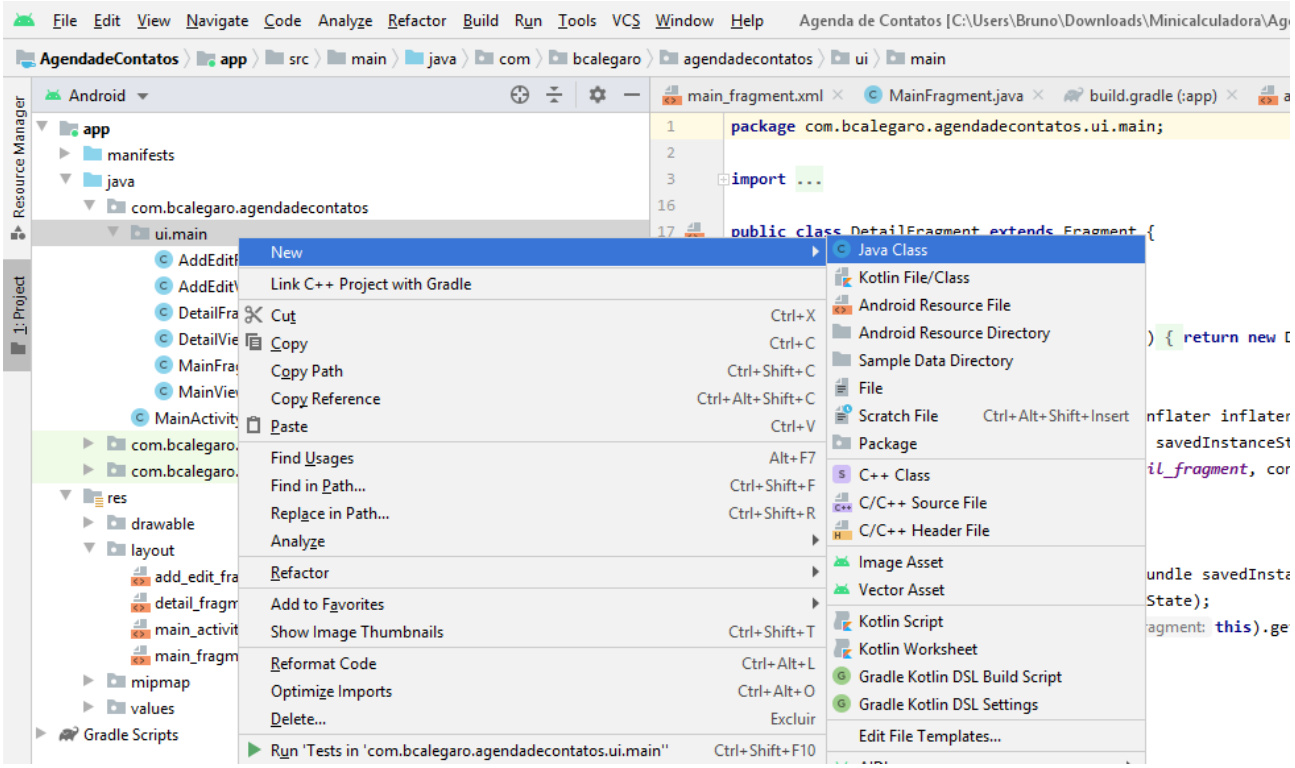


Figura 11 - Menu de contexto para adicionar uma nova classe Java

- Na nova janela coloque o nome da nova classe e clique em **OK**. Crie as classes:
  - **ContactsAdapter** – adaptador que será usado para preencher uma *RecyclerView* com uma lista de contatos
  - **ItemDivider** – classe para desenhar uma linha horizontal entre os itens da *RecyclerView*

- Crie um novo pacote para o projeto. Para fazer isso, clique com o botão direito do mouse sobre o pacote principal e selecione **New > Package**. Coloque o nome como "**data**".

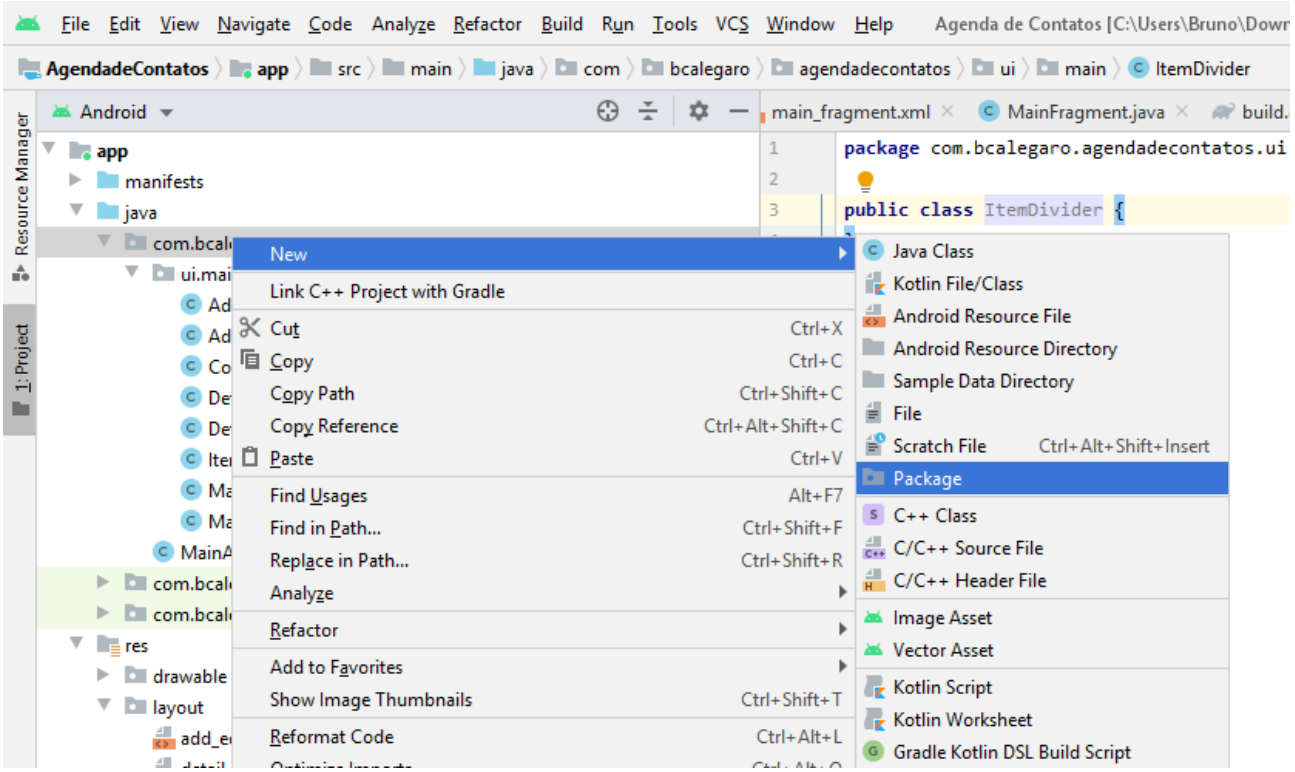


Figura 12 - Criando um novo pacote

- Clique com o botão direito sobre o novo pacote (pasta) criada e adicione novas classes como anteriormente:
  - **Contact** – classe representando um contato no banco de dados
  - **ContactsDAO** – classe usada para manipular as iterações ao banco de dados como consultas e atualizações. DAO significa Database Access Object.
  - **ContactsDatabase** – classe para fazer a criação do banco de dados
  - **ContactsRepository** – classe para servir como uma API de acesso ao banco para o resto da aplicação.

Após a criação das classes iniciais, o projeto deve ficar estruturado como:

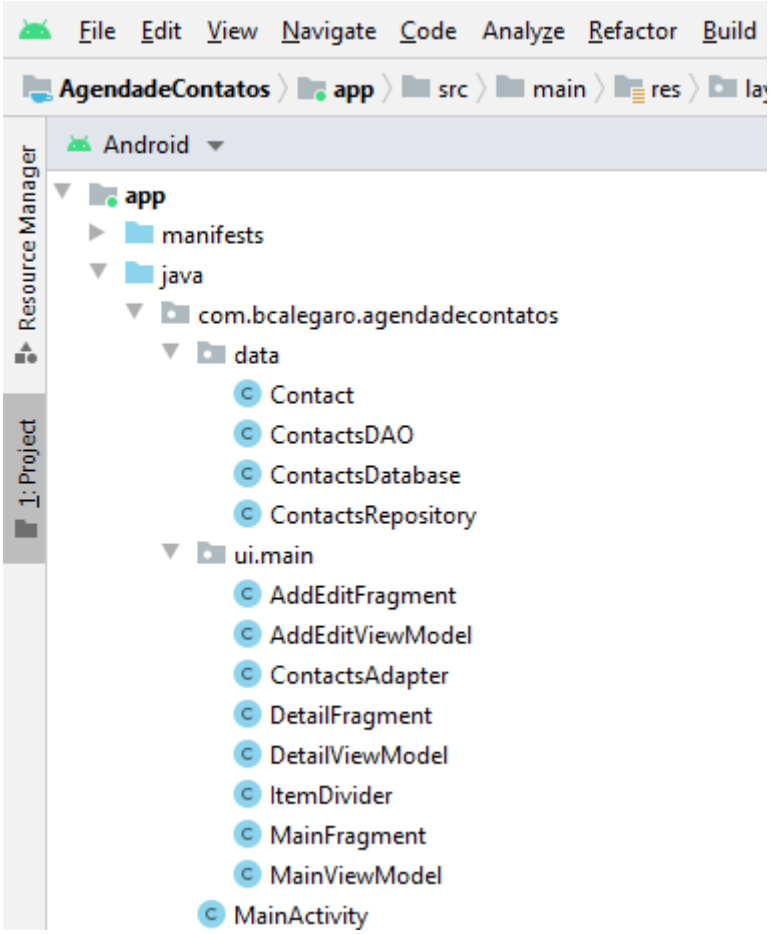


Figura 13 - Estrutura final das classes do projeto

## 5.4. Adicionando Ícones

O Material Design opera nativamente no Android Studio e oferece diferentes tipos de recursos para serem incluídos nas aplicações Android. Um desses recursos é o uso de ícones. O Material Design Icons é um kit com diversos tipos de ícones e está presente dentro do Android Studio pronto para ser utilizado. A vantagem de se usar esses ícones é que eles são imagens vetoriais, isto é, não perdem a qualidade ao escalar o tamanho na tela.

Para adicionar um novo ícone ao projeto no Android Studio, siga os seguintes passos:

- Selecione **File > New > Vector Asset** para abrir a ferramenta **Vector Asset Studio**

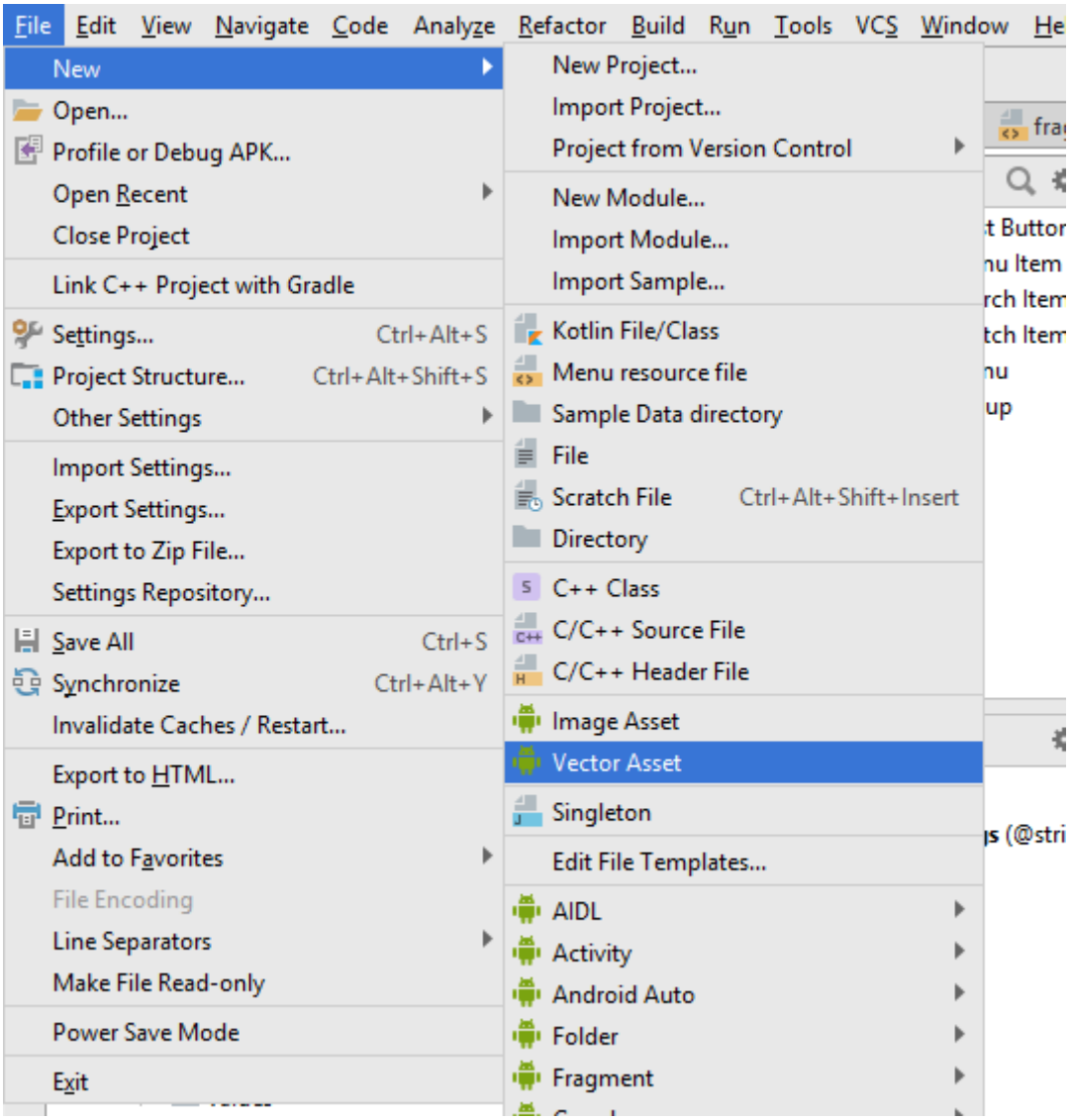


Figura 14 - Caminho para a ferramenta Vector Asset

- Clique no botão ao lado do campo *Clip Art* e na próxima janela procure o ícone com nome "*save*", clique em OK

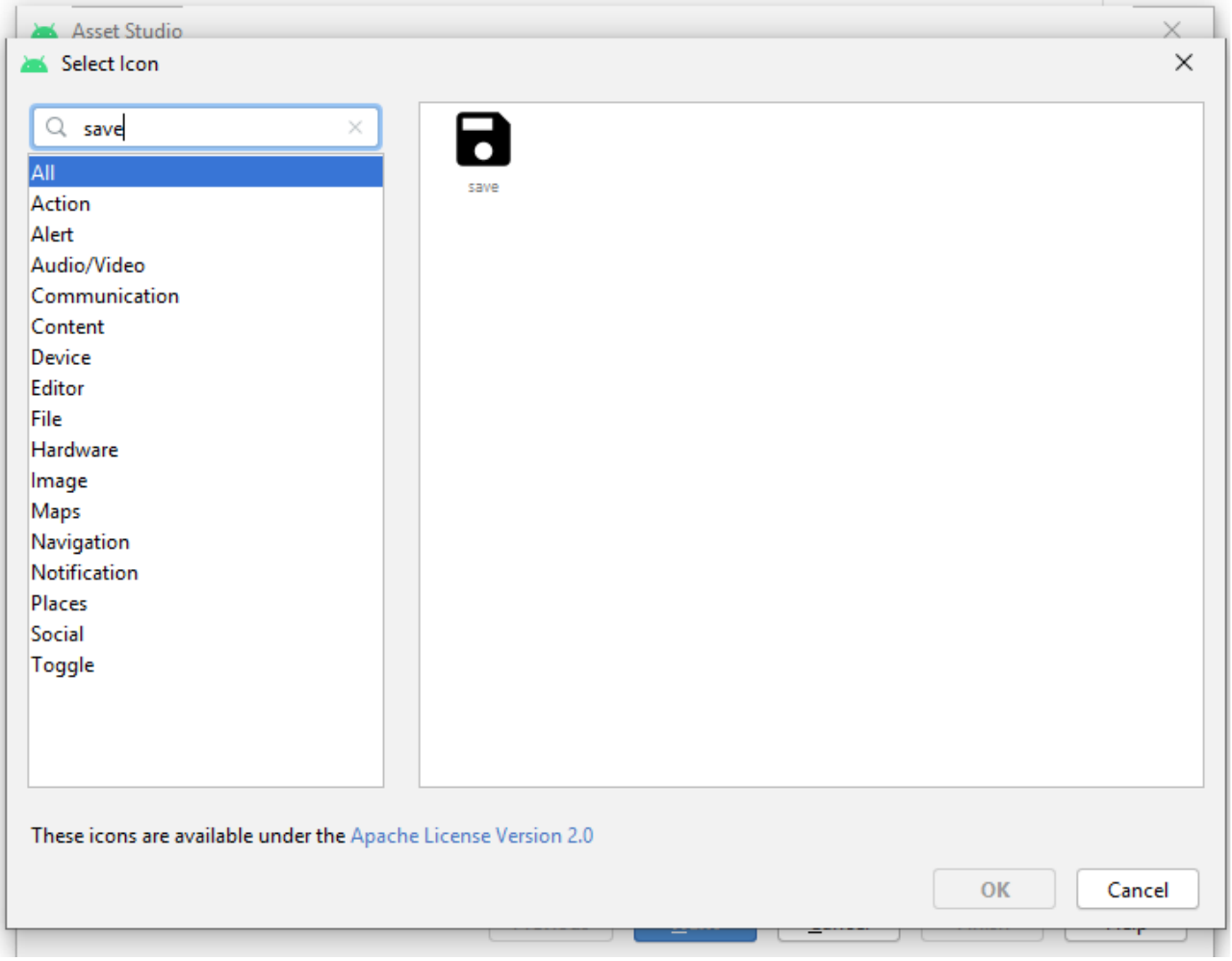


Figura 15 - Selecionando o ícone "save"

- Altera o nome do ícone para “ic_save_24dp” e modifique o campo *Color* para branco (#FFFFFF). Clique em **Next** e **Finish** a seguir.

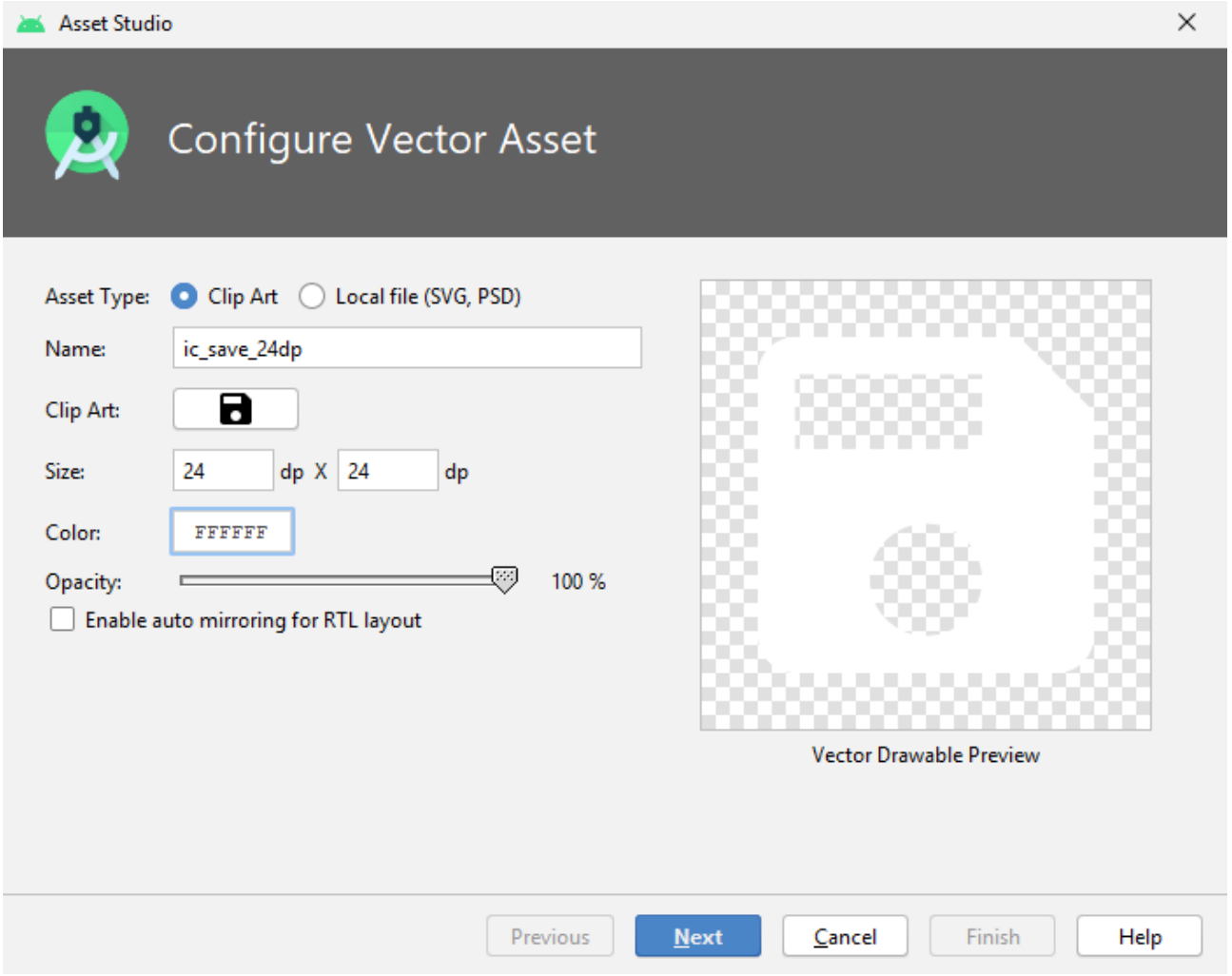


Figura 16 - Adicionando o ícone na cor branca

- Repita o passo 3 para adicionar os ícones *add*, *edit* e *delete*. Respectivamente com os nomes: "ic_add_24dp", "ic_edit_24dp" e "ic_delete_24dp".
- Lembre-se que, imagens vetoriais não são exatamente figuras, mas sim definições de formas. Assim, é possível modificar suas propriedades como cor dos elementos e plano de fundo.

Feito isso, estamos prontos para usar o novo ícone na nossa aplicação em diversos tipos de componentes. Neste projeto, vamos usar esses ícones para os FABs (Float Action Button) e os itens de menu do **DetailFragment**.

Como resultado final vamos ter todos os ícones para a aplicação dentro da pasta drawable dos recursos do projeto:

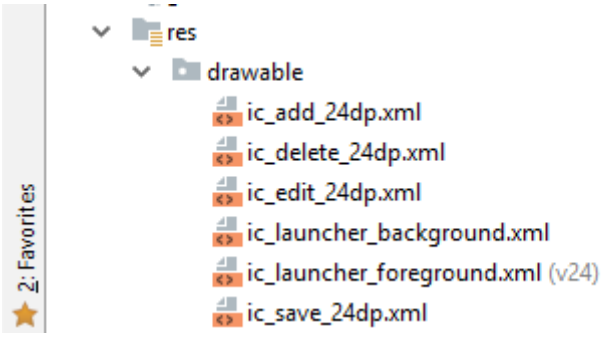


Figura 17 - Pasta drawable com os ícones add, edit, delete e save

## 5.5. Recurso strings.xml

Para o aplicativo vamos precisar de recursos string para armazenar os textos apresentados pela aplicação. Nós poderíamos criar isso junto com a interface gráfica, mas por simplicidade vamos já preencher todos os recursos necessários agora. Para tanto, abra o arquivo *strings.xml* localizado na pasta *res/values* do projeto e preencha com os seguintes campos:

```
<resources>
    <string name="app_name">Agenda de Contatos</string>
    <string name="menuitem_edit">Editar</string>
    <string name="menuitem_delete">Deletar</string>
    <string name="hint_name_required">Nome (Obrigatório)</string>
    <string name="hint_email">E-mail</string>
    <string name="hint_phone">Telefone</string>
    <string name="label_name">Nome:</string>
    <string name="label_email">E-mail:</string>
    <string name="label_phone">Telefone:</string>
</resources>
```

## 5.6. Configurando um estilo de borda para os componentes

Nos componentes *TextView* do aplicativo será aplicado um estilo de borda. Para isso vamos adicionar um elemento *drawable* chamado **textview_border** para a propriedade *background*. Esse novo elemento irá criar uma borda em volta da views. Para definir como será o desenho da borda temos que adicionar um novo recurso *Drawable* ao projeto. Dessa forma, execute os passos abaixo:

- Clique com o botão direito do mouse na pasta *res/drawable* e selecione **New > Drawable resource file**
- Especifique *textview_border* para **File name** e clique em **OK**
- Abra o arquivo e insira o código:

```
<?xml version="1.0" encoding="utf-8"?>

<shape xmlns:android="http://schemas.android.com/apk/res/android"

        android:shape="rectangle">

        <corners android:radius="5dp"/>

        <stroke android:width="1dp" android:color="#555"/>

        <padding android:top="10dp" android:left="10dp"

        android:bottom="10dp" android:right="10dp"/>

</shape>
```

O atributo *android:shape* do elemento **shape** pode ter o valor “rectangle”, “oval”, “line” ou “ring”. O elemento **corners** especifica o raio do canto do retângulo, nesse caso, deixando arredondado. O elemento **stroke** define a largura e a cor da linha do retângulo. O elemento **padding** especifica o tamanho em torno do conteúdo no elemento em que **Drawable** é aplicado.

# 6. Desenhando as telas do aplicativo

Nesta etapa vamos criar e configurar os layouts das telas do aplicativo. O projeto **Agenda de Contatos** possui três telas: a tela principal exibe a lista de contatos, a tela de detalhes exibe as informações de um contato selecionado e a tela de adição/edição insere ou altera as informações do contato.

## 6.1. Layout do arquivo main_activity.xml

Neste aplicativo vamos usar o componente **Navigation** para adicionar ou remover os fragmentos na tela. Dessa forma, nesse projeto vamos usar um tipo de layout novo: o **FrameLayout**. O FrameLayout vai se comportar como um recipiente a ser preenchido pelos fragmentos que serão criados. Na parte 2 da apostila vamos atualizar esse arquivo, por ora, podemos abrir o arquivo *activity_main.xml* e alterar o campo **id** do *FrameLayout* de container para *fragmentContainer*.

Como resultado temos:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:tools="http://schemas.android.com/tools"
      android:id="@+id/fragmentContainer"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      tools:context=".MainActivity" />
```

## 6.2. Layout do arquivo main_fragment.xml

No fragmento de contatos vamos apenas exibir uma lista contendo o nome dos contatos salvos e um FAB para adicionar novos contatos. Dessa forma, siga as etapas abaixo para a configuração da tela de contatos:

- Abra o arquivo em modo texto e substitua o *ConstraintLayout* para **FrameLayout**.
- Apague o elemento *TextView* pré-definido
- Adicione uma *RecyclerView*, aba "*Common*", e configure seu **id** para recyclerView
- Caso solicitado a inclusão da biblioteca ao projeto, selecione **OK** na caixa de diálogo. Espere a operação terminar e continue a criação da interface gráfica.

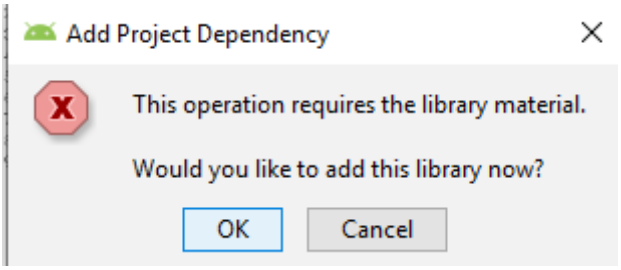


Figura 18 Janela de confirmação para adicionar biblioteca ao projeto

- Adicione um FAB (*Float Action Button*), aba *Button*, escolha o ícone de adicionar (ic_add_24dp). Novamente, se aparecer um tela solicitando o download da biblioteca (*library*) clique em **OK** e aguarde o término da operação,
- Configure o FAB como: **id** para addButton e **layout:gravity** para "top|end"
- Como resultado temos a tela:

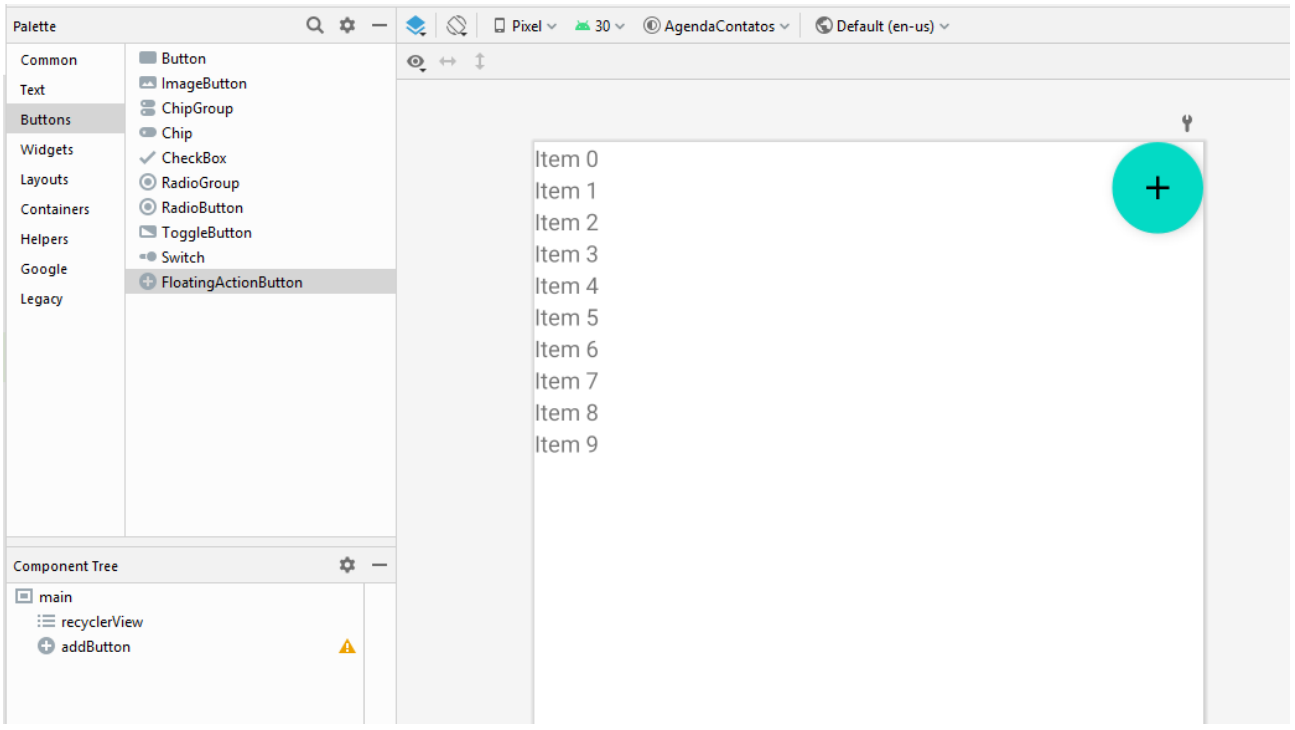


Figura 19 - Tela da lista de contatos

## 6.3. Layout detail_fragment.xml

Quando o usuário tocar em um contato da *RecyclerView* o aplicativo exibirá o fragmento **DetailFragment**. O layout desse fragmento consiste em um elemento *GridLayout* vertical com duas colunas de componentes *TextView*.

Para criar esse layout vamos modificar o arquivo *detail_fragment*. Abra o arquivo e apague o elemento *TextView* criado automaticamente. Adicione um componente *GridLayout* ao *FrameLayout*, aba *Legacy*. Novamente, caso solicitado a inclusão da biblioteca ao projeto, selecione **OK** na caixa de diálogo e espere a operação terminar para continuar a criação da interface gráfica.

*Obs: O FrameLayout é necessário para o uso correto de fragmentos, por isso, adicione o GridLayout dentro do FrameLayout, ou seja, não escreva por cima!*

Configure as propriedades do **GridLayout** para:

- **columnCount** => 2
- **useDefaultMargins** como *true*

Adicione ao *GridLayout* os elementos *TextView* organizados como na próxima figura:

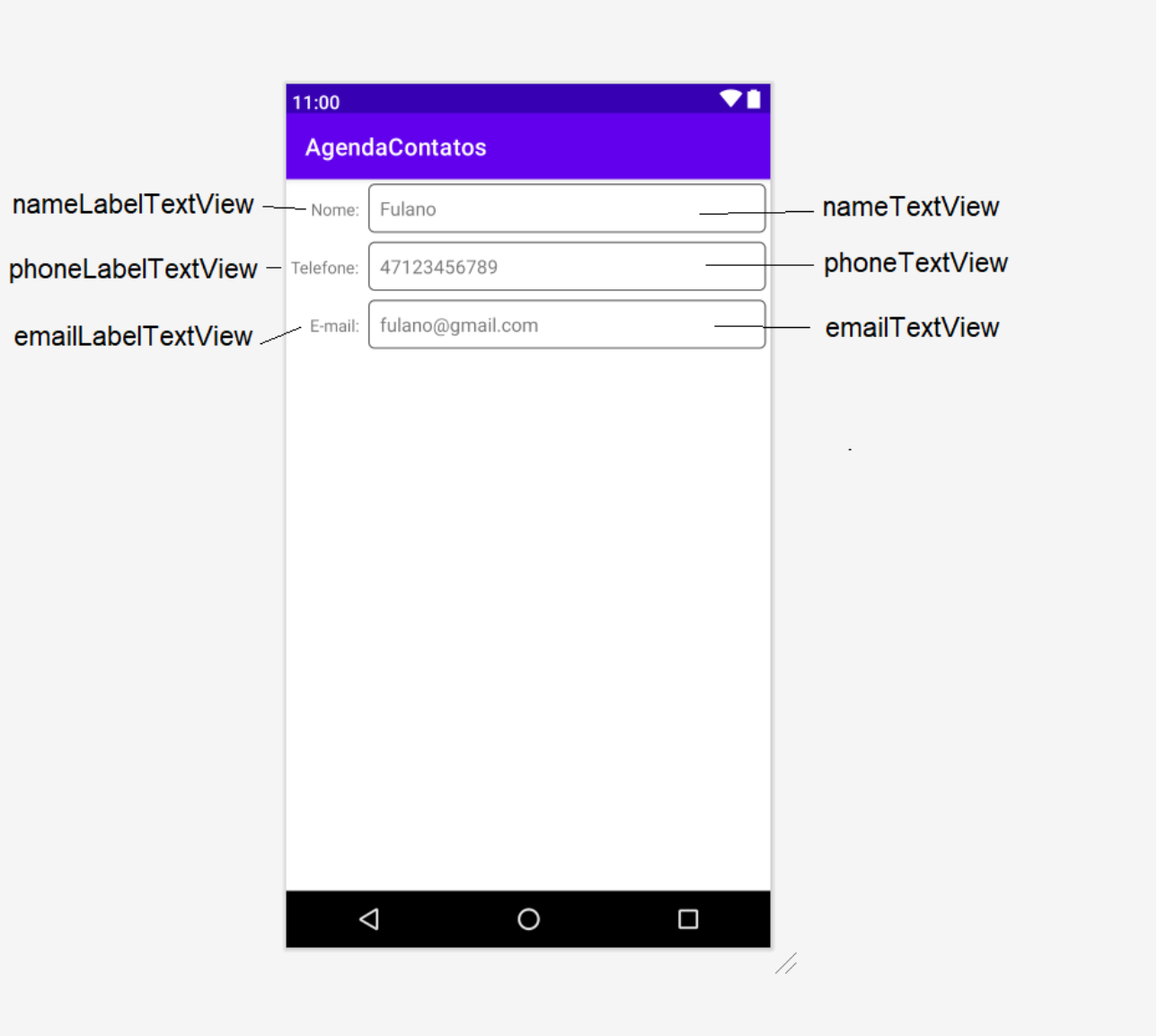


Figura 20 - Organização das Views no detail_fragment.xml

- Configure a propriedade **id** de cada elemento como na figura acima.
- Configure as TextViews da coluna da esquerda como:
  - **layout:row** com valor de 0 a 2, dependendo da linha
  - **layout:column** => 0
  - **text** com o o recurso strings apropriado (*label_nome, label_email, label_phone*)
  - **layout_gravity**: *end* e *center_horizontal*
- Configure as *TextViews* da coluna da **direita** como:
  - **layout:row** com valor de 0 a 2, dependendo da linha
  - **layout:column** => 1
  - **textSize**: 16sp
  - **layout_gravity**: *fill_horizontal*
  - **background**: utilize o botão "[]" para selecionar o "*textview_border*"

Como resultado final temos a seguinte tela:

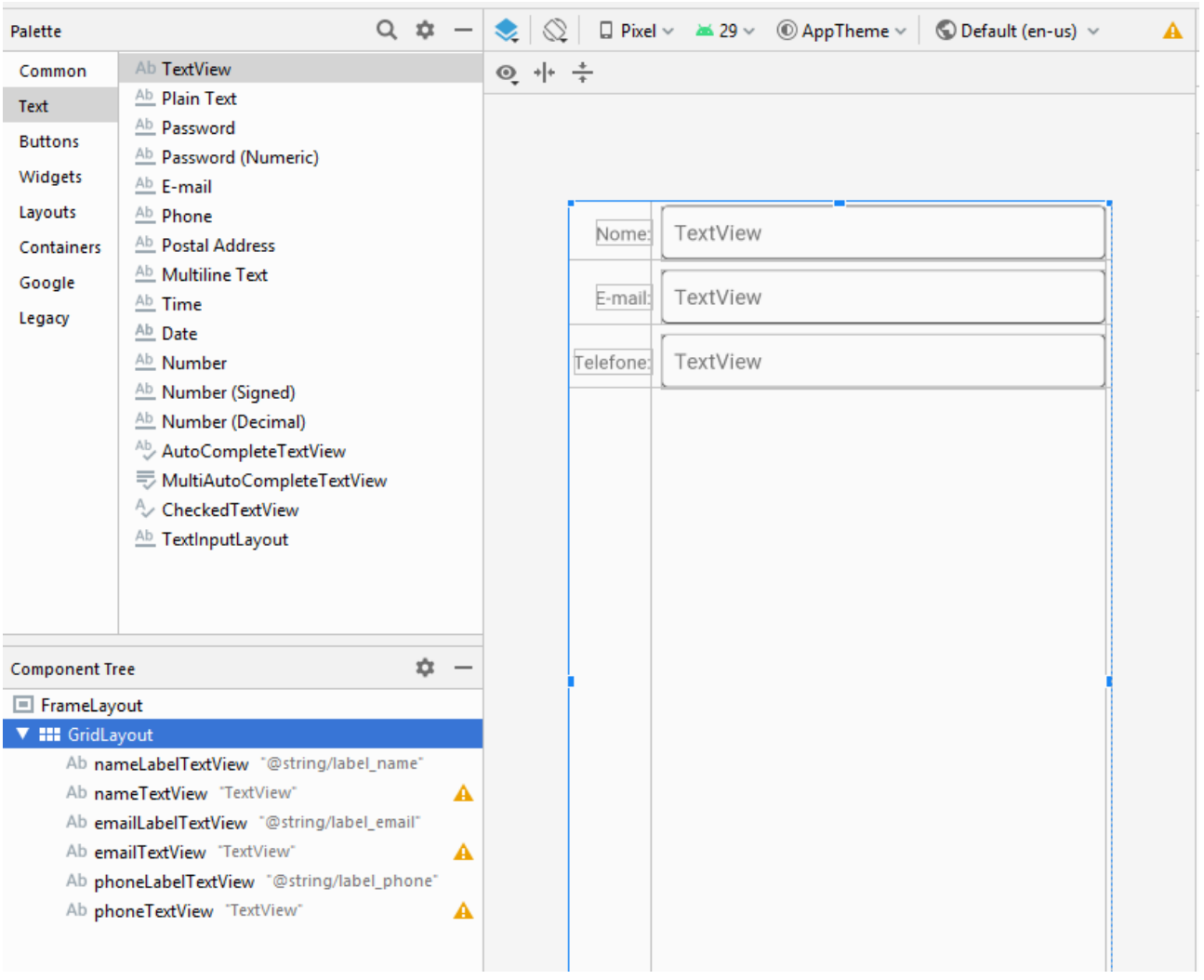


Figura 21 - Tela exibindo as informações de um contato específico

## 6.4. Layout add_edit_fragment.xml

Quando o usuário clicar no FAB do *MainFragment* ou no botão de editar do menu de

*DetailFragment*, o fragmento deve mostrar a tela *add_edit_fragment.xml*. Essa tela contém um **FrameLayout** com um LinearLayout e um botão flutuante. O layout linear possui *TextInputLayouts* para o usuário inserir os dados do contato.

Abra o arquivo *add_edit_fragment.xml* e realize as seguintes alterações:

- Apague a *TextView* criada automaticamente
- Adicione um *LinearLayout* e configure sua orientação para a **vertical** e *layout_height* para *wrap_content*.
- Adicione ao *LinearLayout* três elementos **TextInputLayout**, aba *Text,* para cada novo elemento será criado também uma *EditText*.
- Para cada *TextInputLayout* adicionado, configure a propriedade **id** como mostrado na figura a seguir e escolha a propriedade **hint** apropriada para cada *EditText.*

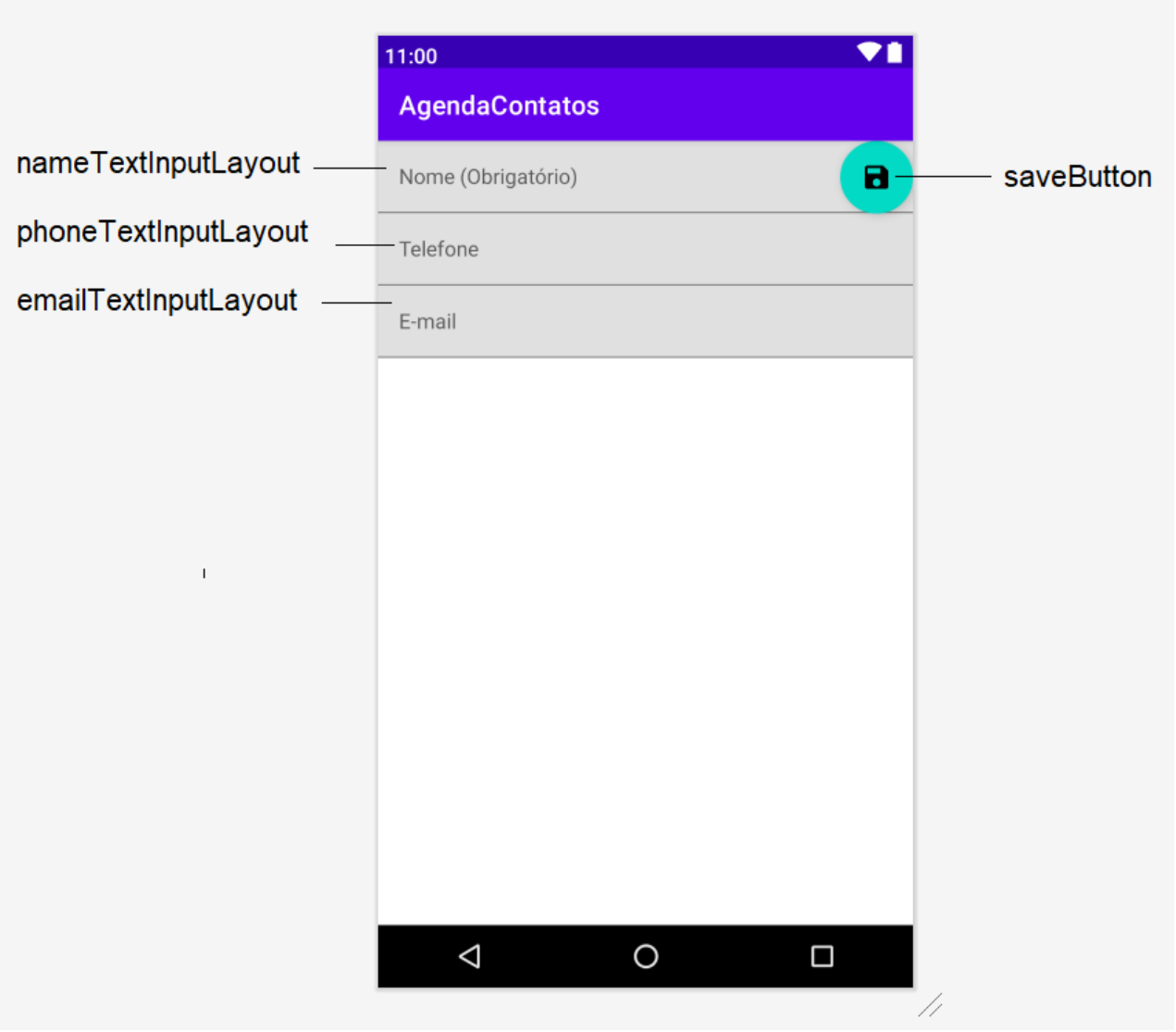


Figura 22 - Organização das views no fragment_add_edit.xml

- Selecione a *EditText* dos elementos *nameTextInputLayout* e *phoneTextInputLayout* para configurar a propriedade **imeOptions** para *actionNext*. Com essa configuração ficará mais fácil para o usuário preencher os dados usando o teclado.
- Selecione a *EditText* do último elemento, *emailTextInputLayout*, e configure a propriedade **imeOptions** para *actionDone*. Com essa configuração finalizamos o preenchimento e o

teclado irá se esconder quando o usuário clicar em prosseguir.

- Configure cada *EditText* para um tipo de dado específico. Isso ajuda no preenchimento dos dados pelo usuário. Assim, modifique a propriedade **inputType** como:
    - EditText do **nameTextInputLayout** – selecione *textPersonName* e *textCapWords* (coloca automaticamente letras maiúsculas)
    - EditText do **phoneTextInputLayout** – selecione *phone*
    - EditText do **emailTextInputLayout** – selecione textEmailAddress
- Adicione um FAB ao FrameLayout, escolha o ícone de salvar (ic_save_24dp). Configure seu **id** para *saveButton* e **layout:gravity** para "top|end"

Como resultado temos a tela:

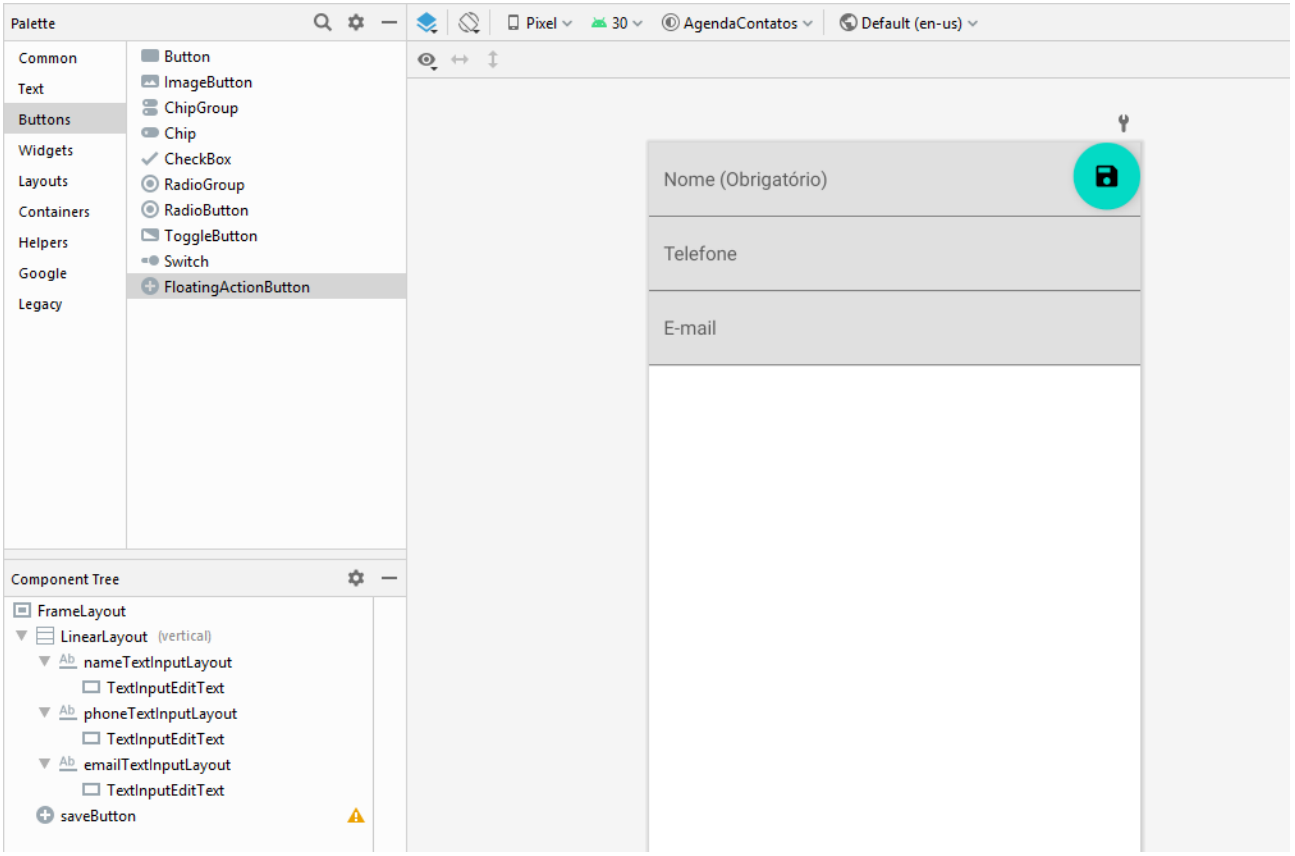


Figura 23 - Tela para adicionar ou atualizar um contato

## 6.5. Menu fragment_details_menu.xml

Nesta aplicação, apenas um fragmento irá precisar de menus, o **DetailFragment**. Dessa forma, execute os seguintes passos para incluir um menu a aplicação:

- Para adicionar um novo menu clique com o botão direito nas pastas do projeto e selecione **New > Android resource file**
- Na janela *New resource file* preencha com o nome *fragment_details_menu*. No campo "*Resource type*" selecione **Menu**. Clique em **OK**.
- Abra o arquivo criado e adicione um novo Menu Item da paleta. Configure suas propriedades como:
    - **id** => action_edit

  - **icon** => @drawable/ic_edit_24dp
  - **title** => @string/menuitem_edit
  - **showAsAction** => Always
- Adicione outro elemento Menu Item e configure suas propriedades como:
  - **id** => action_delete
  - **icon** => @drawable/ic_delete_24dp
  - **title** => @string/menuitem_delete
  - **showAsAction** => Always

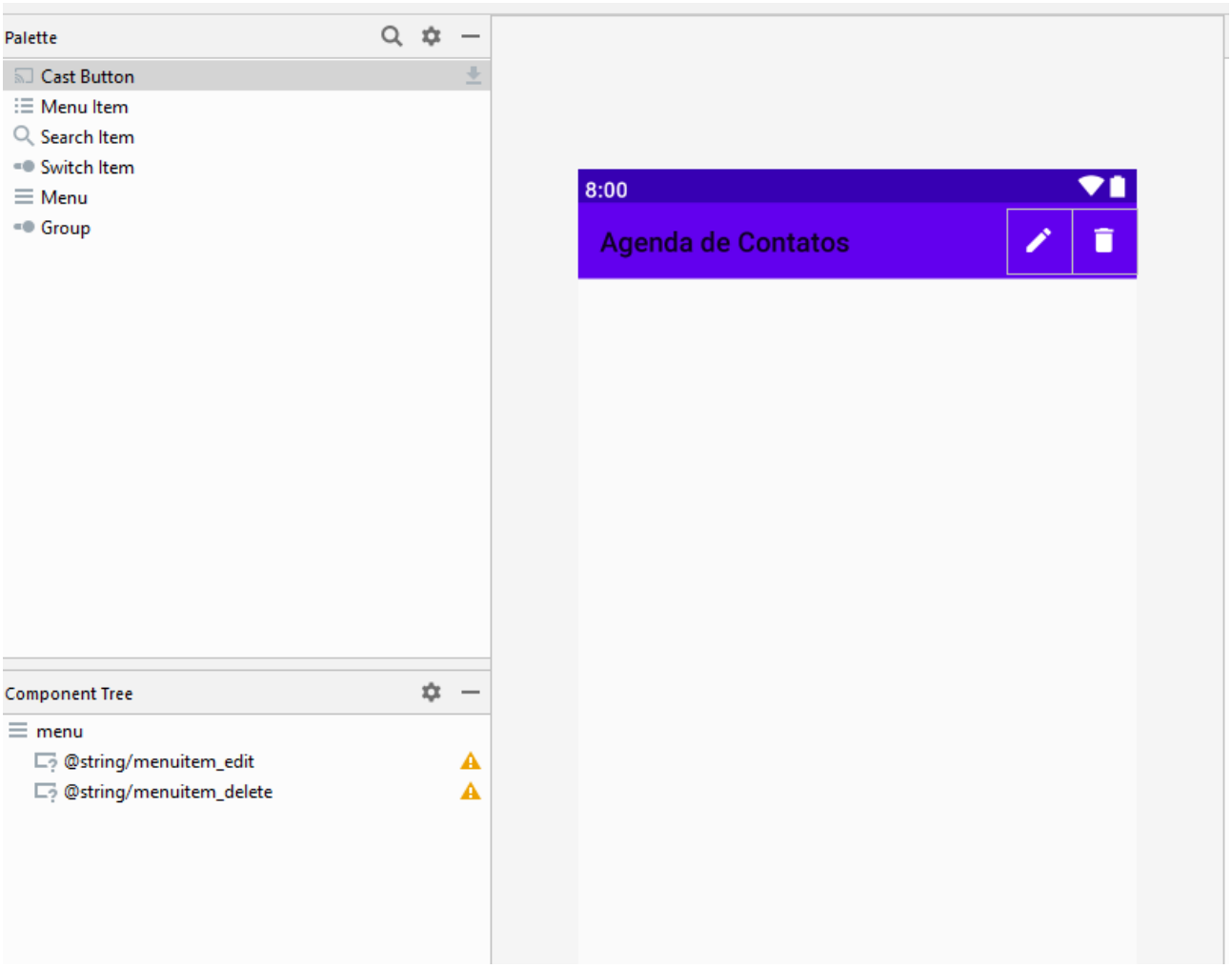


Figura 24 - Menu para o *DetailFragment*

Pronto. Construímos um menu para o **DetailsFragment** com as opções de editar e deletar um contato.

Com esse último passo concluímos a criação dos recursos iniciais do projeto e o desenho das telas do aplicativo. Nas próximas partes do projeto iremos fazer a transação entre as telas e a programação dos eventos e o acesso ao banco de dados.

# Parte 2 - Transição entre telas e apresentação de dados fictícios

## 7. Apresentação

Na primeira parte do projeto foram construídas as telas do aplicativo e as classes referentes a **Atividade**, **Fragmentos**, **ViewModels**, **Adapters** e classes necessárias para a configuração e execução do banco de dados.

Devido ao tamanho do aplicativo, vamos revisar a finalidade de cada classe:

- **Pacote data**
  - **Contact**: classe para conter a descrição de um objeto a ser manipulado pelo banco de dados.
  - **ContactsDAO**: classe padrão da biblioteca Room para conter as operações a ser manipuladas pelo banco de dados
  - **ContactsDatabase**: classe padrão da biblioteca Room para conter a descrição do banco de dado
  - **ContactsRepository**: esta classe não é exigida pela biblioteca Room mas é usada devido boas práticas de programação pois ela irá oferecer a aplicação uma interface (API) para o restante do aplicativo.
- **Pacote principal**
  - **MainActivity**: classe principal que gerencia os fragmentos do aplicativo
- **Pacote ui.main**
  - **MainFragment**: essa classe gerencia a *RecyclerView* da lista de contatos e o *FloatingActionButton* para adicionar novos contatos.
  - **MainViewModel**: *ViewModel* para vincular os dados do banco a uma lista de contatos
  - **ContactsAdapter**: subclasse de *RecylerView.Adapter* usada pelo **ContactsFragment** para vincular a lista ordenada de nomes de contato na *RecyclerView*.
  - **ItemDivider**: essa classe define o divisor exibido entre os itens da *RecyclerView*.
  - **AddEditFragment**: essa classe gerencia os *TextInputLayouts* e um *FloatingActionButton* para adicionar um novo contato ou editar um já existente.
  - **AddEditViewModel**: *ViewModel* para armazenar os dados de um novo contato ou um já existente
  - **DetailFragment**: essa classe gerencia os componentes *TextView* estilizados que exibem os detalhes de um contato selecionado e os itens da barra de aplicativo que permitem ao usuário editar ou excluir o contato que está sendo exibi
  - **DetailViewModel**: *ViewModel* para vincular os dados de um contato específico

Nesta segunda parte do projeto, vamos fazer a configuração das interações entre os fragmentos com o componente de navegação **Navigation** e apresentação de dados fictícios.

## 8. Recursos Envolvidos

No desenvolvimento da segunda parte do aplicativo, vamos usar as seguintes tecnologias envolvidas na criação de aplicativos de múltiplas telas. Na primeira parte do projeto foram criados três **Fragments** para o aplicativo. Nessa segunda parte, vamos lidar com a transição entre as telas e a configuração das interações entre os objetos.

- **Navegação entre Fragmentos**

- A navegação se refere às interações que permitem aos usuários navegar, entrar e sair de diferentes partes do conteúdo no aplicativo. O componente de navegação do Android Jetpack ajuda a implementar a navegação, desde simples cliques em botões até padrões mais complexos, como barras de aplicativos e a gaveta de navegação.
- Neste aplicativo vamos ter as seguintes transições de tela:
  - selecionar um contato para exibição;
  - tocar no *FloatActionButton* de adição (+) do fragmento da lista de contatos;
  - tocar nas ações de editar ou salvar o contato exibido no fragmento de detalhes;
  - tocar em salvar para finalizar a edição de um contato existente ou a adição de um novo contato

- **Componente de Navegação**
  - Neste projeto vamos fazer o uso dos componentes de arquitetura disponíveis no **Android Jetpack** (pacote **androidx**). Entre eles está o novo componente de navegação, Navigation, que oferece uma solução moderna e simples de se implementar, desde a configuração de gráfico de transições para a programação das transições.
  - O componente de navegação consiste de três partes principais, descritas abaixo:
    - **Grafo de navegação**: é um recurso XML que contém todas as informações relacionadas à navegação em um local centralizado. Isso inclui todas as áreas de conteúdo individual no aplicativo (telas), chamadas destinos, e todos os caminhos que podem ser percorridos pelo usuário no aplicativo.
    - **NavHost**: é um contêiner vazio que mostra destinos do gráfico de navegação. O componente de navegação contém uma implementação *NavHost* padrão, *NavHostFragment*, que mostra os destinos do fragmento. Esse componente gráfico será adicionado à atividade principal e funcionará como o recipiente a ser preenchido com as telas do aplicativo.
    - **NavController**: é um objeto que gerencia a navegação do aplicativo em um *NavHost*. O NavController organiza a troca do conteúdo de destino no *NavHost* conforme os usuários se movem pelo aplicativo

# 9. Navegação

Neste projeto vamos fazer o uso do componente **Navigation** para configurar a transição das telas. Para tanto precisamos primeiro criar um novo gráfico de navegação e criarmos as transições entre as telas. Também precisamos configurar a atividade principal e programar as ações dos botões.

## 9.1. Criando um grafo de navegação

Para criar o grafo de navegação precisamos adicionar um novo recurso no projeto. Para tanto, clique com o botão direito sobre a pasta res e selecione *New -> Android Resource File*.

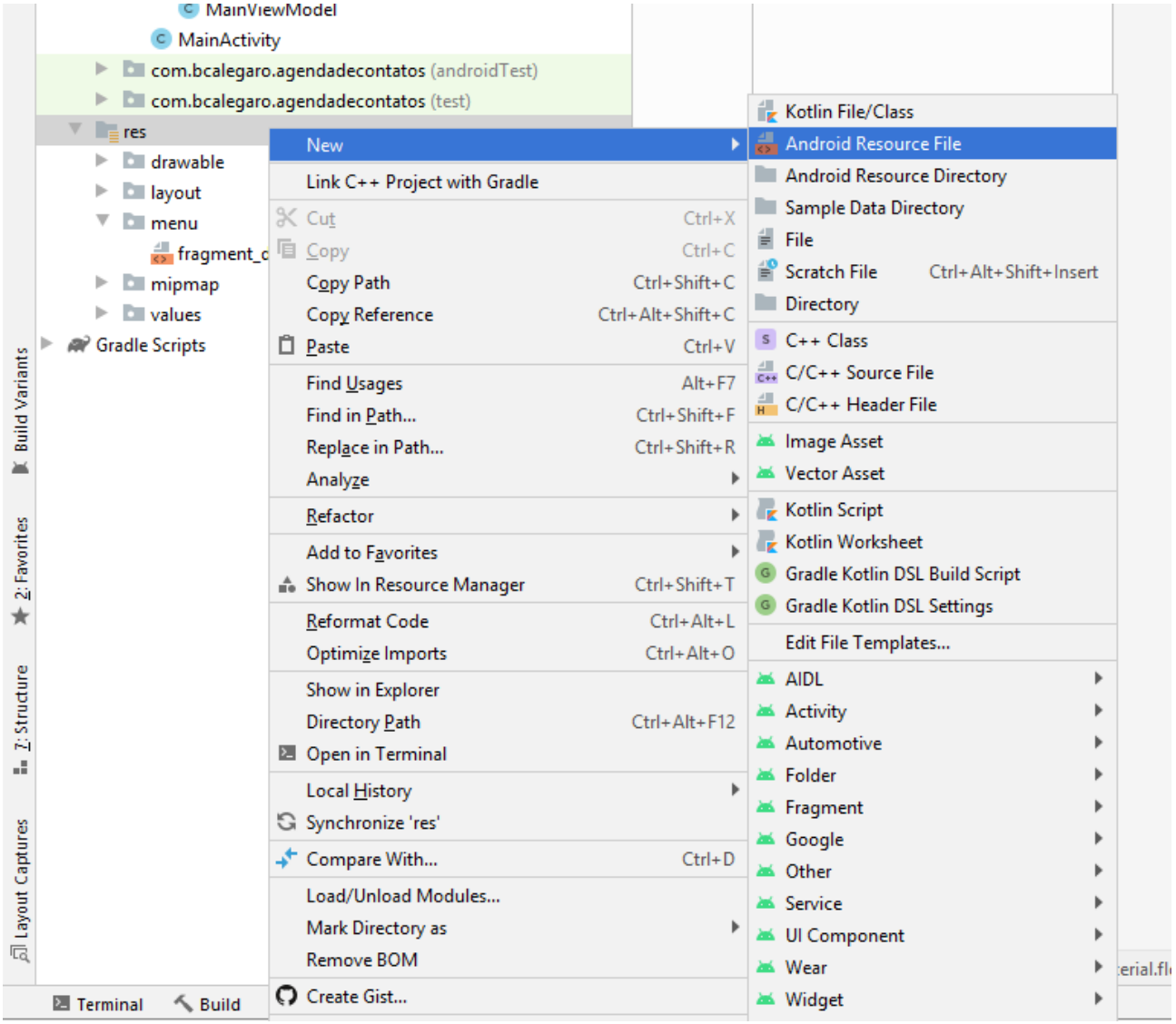


Figura 25 - Adicionando um novo recurso Android

Na próxima janela preencha o nome do recurso como *nav_graph* e selecione o campo "*Resouce type*" como **Navigation**. Clique em **OK**.

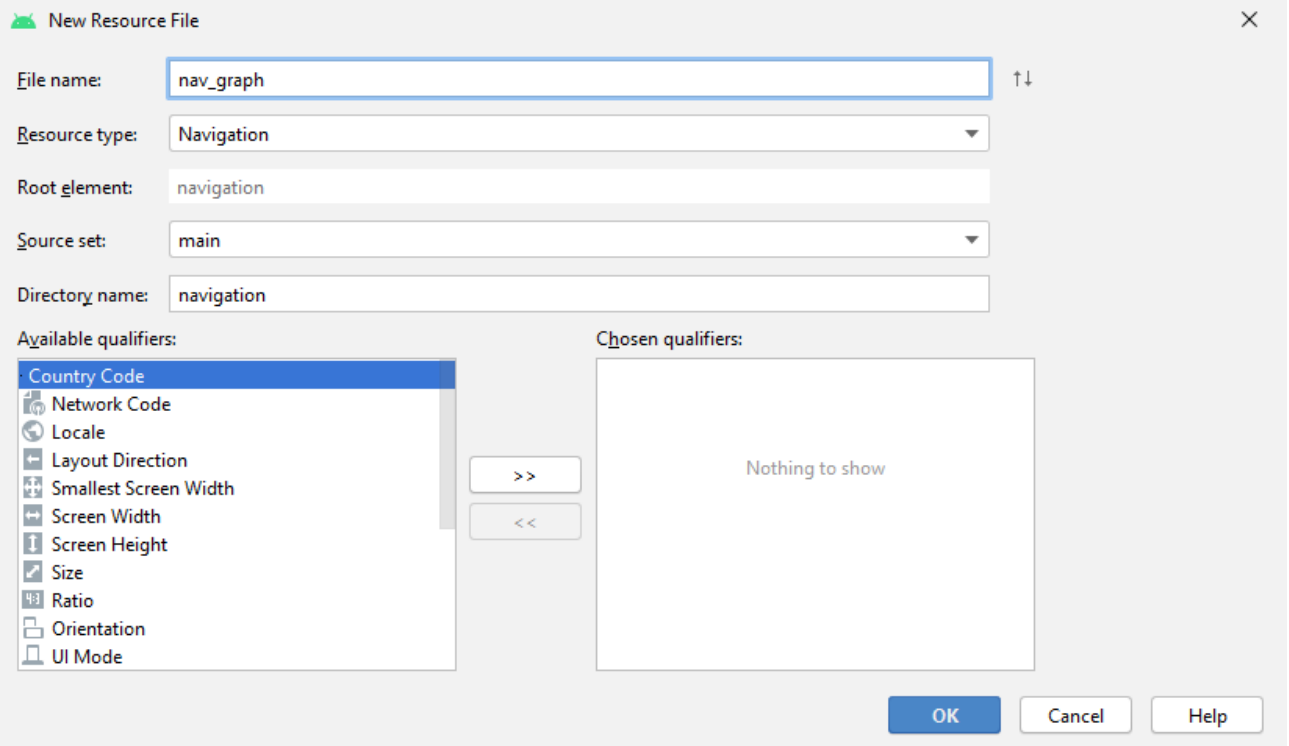


Figura 26 - Criação do recurso grafo de navegação

Provavelmente, será notificada uma janela de aviso dizendo para adicionar as dependências do novo componente ao projeto, clique em **OK** e aguarde. O componente **Navigation** pertence ao pacote **androidx.navigation** é deve ser adicionado ao projeto. Se você seguir as orientações do Android Studio isso ocorrerá automaticamente.

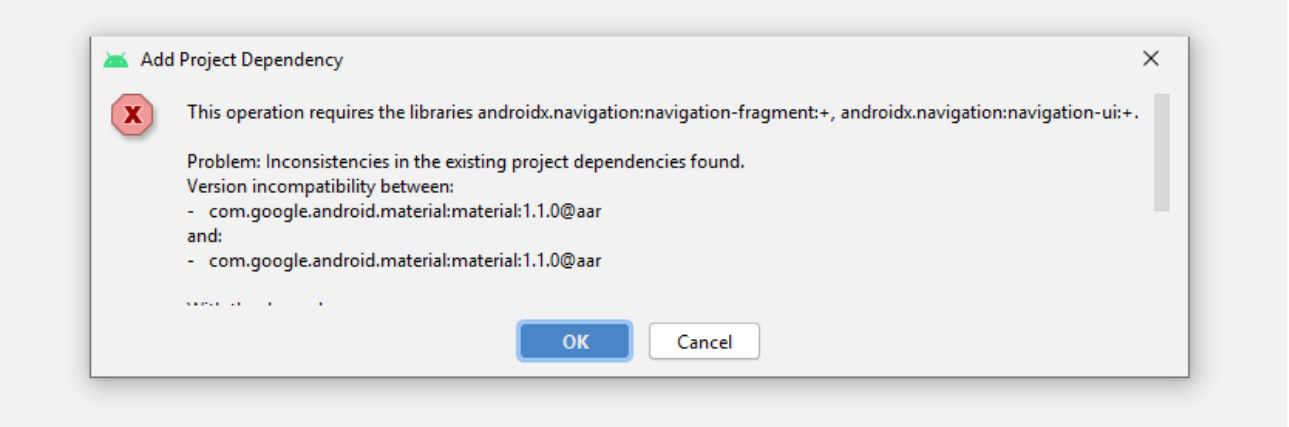


Figura 27 - Adicionando dependências no projeto para usar o componente Navigation

Com a nova biblioteca uma nova pasta para armazenar grafos de navegação foi criada. Se tudo estiver certo, ao abrir o arquivo *nav_graph*, localizado na pasta *res/navigation* será mostrada a seguinte interface gráfica.

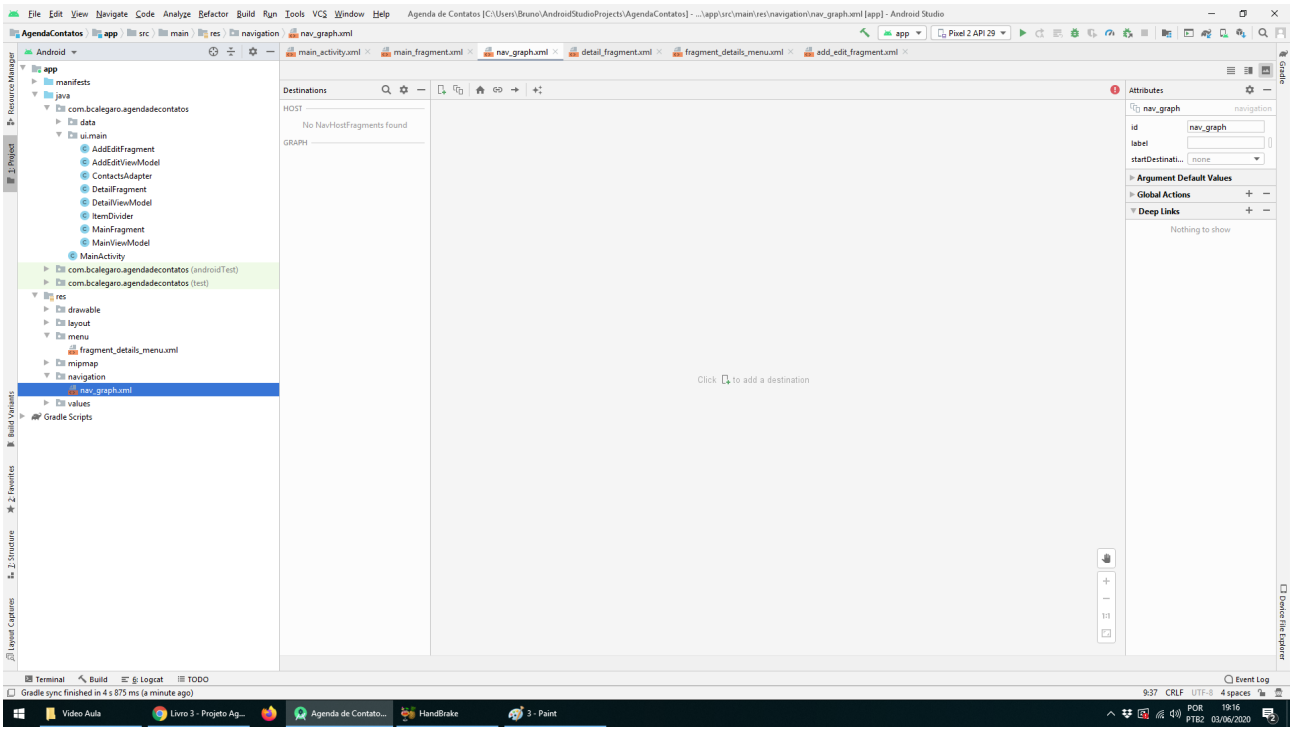

Figura 28 - Localização do arquivo nav_graph

A navegação pode ser configurada via arquivo xml ou graficamente, por simplicidade, iremos ilustrar graficamente como fazer o processo passo a passo. Para o projeto da **Agenda de Contatos** vamos adicionar primeiramente as telas dos fragmentos e depois fazer as transições entre elas.

## 9.2. Configurando a navegação

Inicialmente o grafo está em branco e não mostra nenhuma tela. Vamos adicionar a primeira tela como o **MainFragment** e consequentemente configurar ela como a tela inicial. A tela inicial pode ser configurada manualmente, mas como o grafo está em branco automaticamente ele vai definir a tela inicial como a primeira tela adicionada à navegação.

Clique no botão ilustrado na figura para adicionar um novo destino (New Destination) e selecione o fragmento principal do nosso projeto.

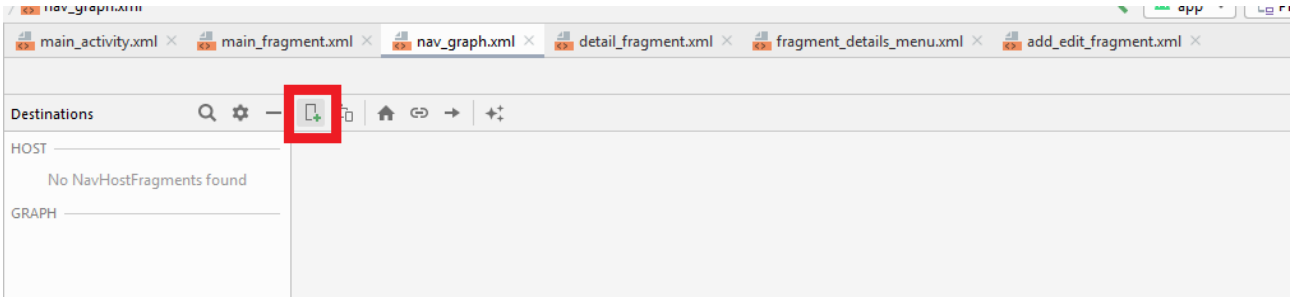


Figura 29 - Adicionando um novo destino

As possíveis telas serão mostradas como opção de destino. Selecione *main_fragment* para adicionar a tela ao grafo.

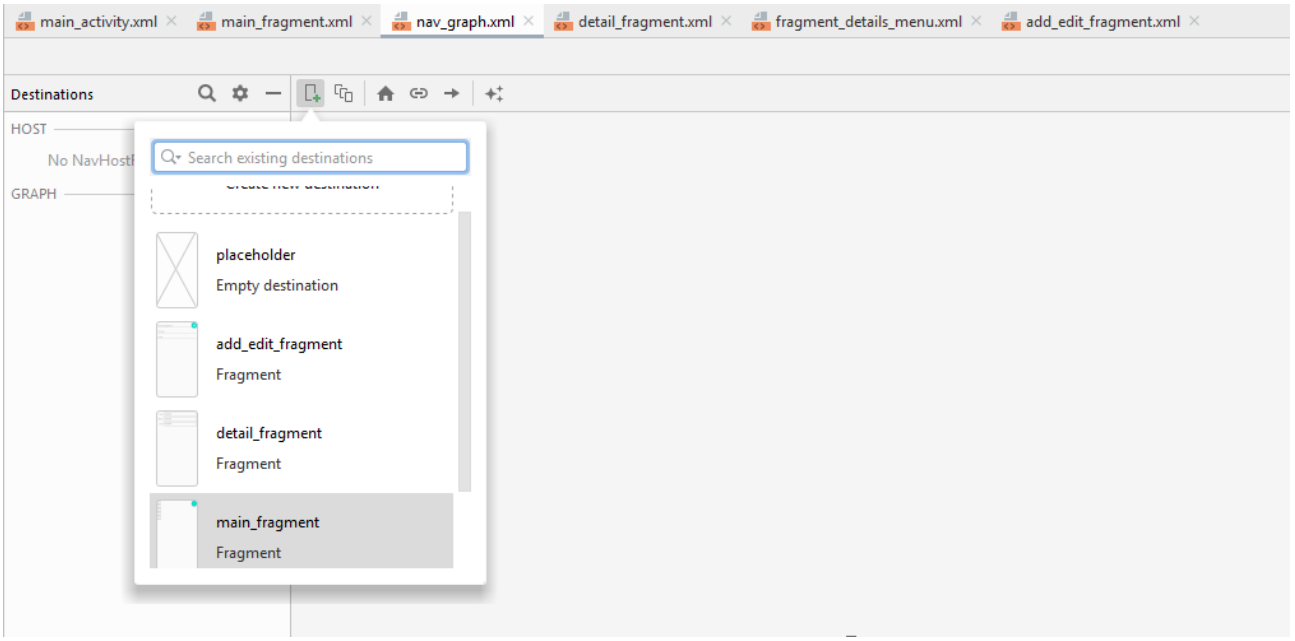


Figura 30 - Definindo o destino

Após adicionar o fragmento principal o grafo se apresentará da seguinte forma, observe que está anotado *Start* mostrando que a tela principal da navegação será o fragmento adicionado.

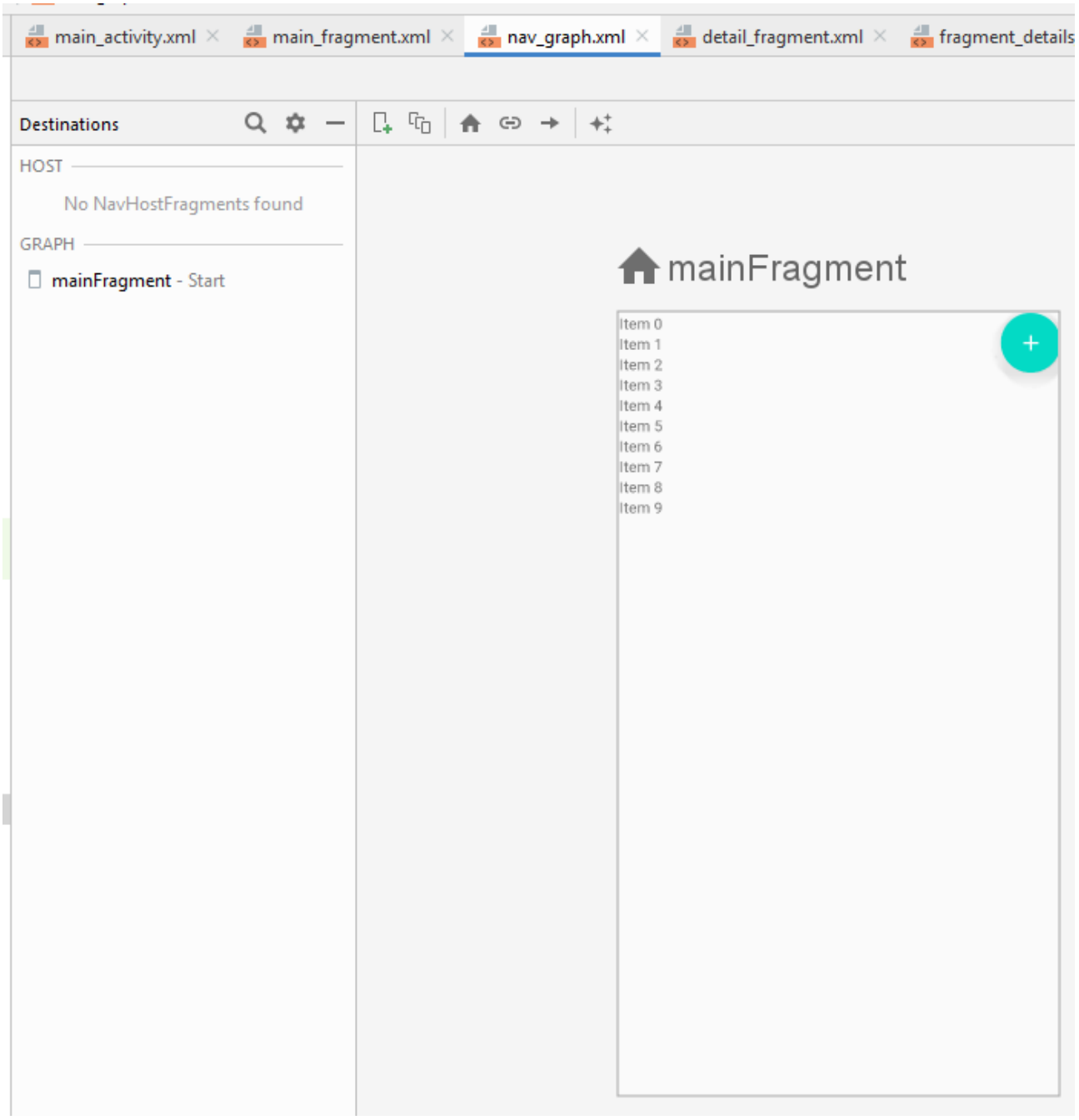


Figura 31 - Visualização da navegação com o *mainFragment* como destino inicial

Repita os passos anteriores e adicione as demais telas ao grafo. Como resultado final, após organizar a posição das telas você poderá ter a seguinte tela:

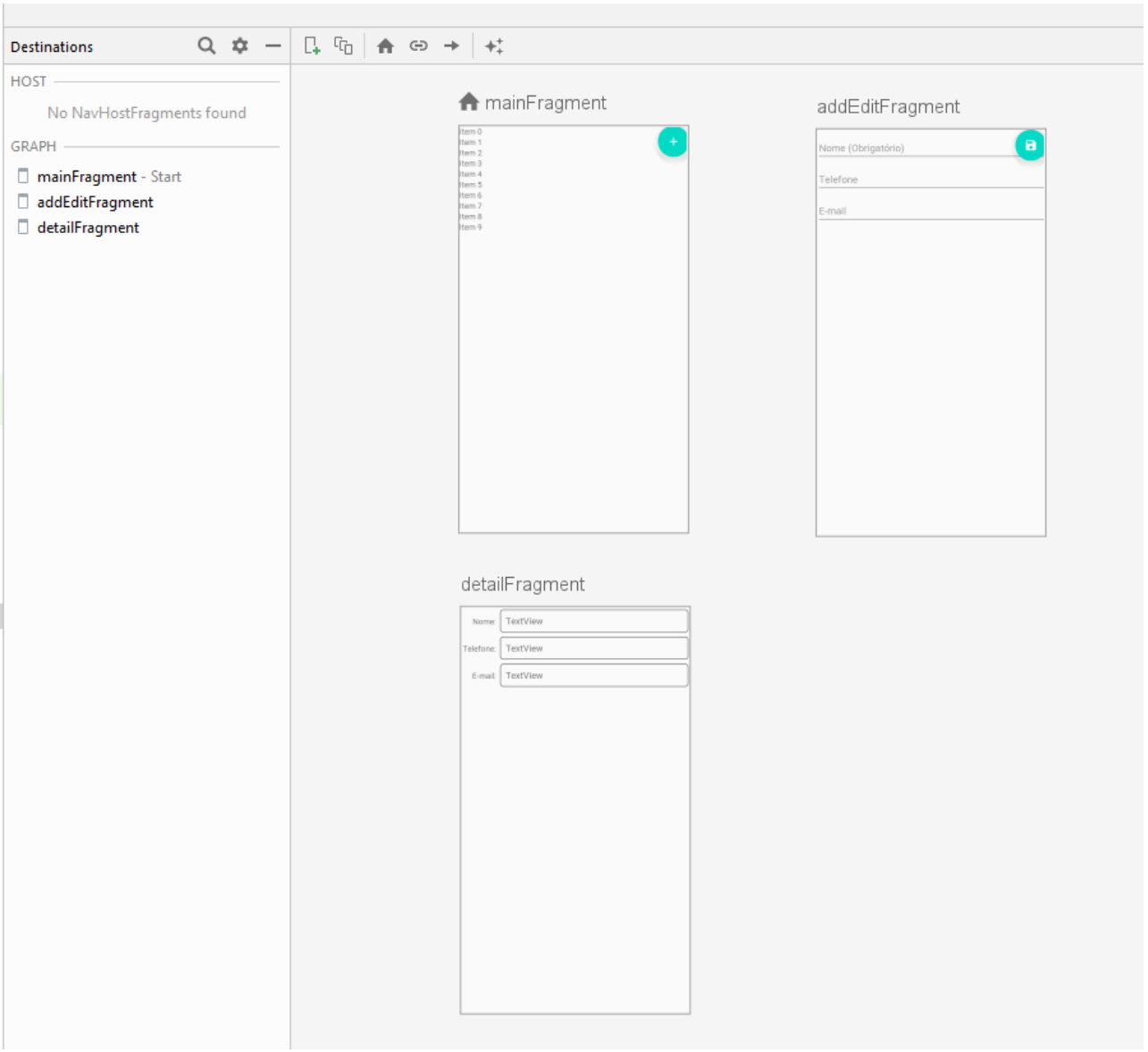


Figura 32 - Telas do aplicativo no grafo de navegação

A próxima etapa é configurar as possíveis transições de tela, pois a navegação se dará por meio das transições configuradas nesse momento. Assim, selecione uma tela e clique sobre a bolinha azul que aparece na borda da tela, segure e clique na tela a qual será o destino da transição. Segue a ilustração de como fazer a transição do fragmento principal para o fragmento de editar:

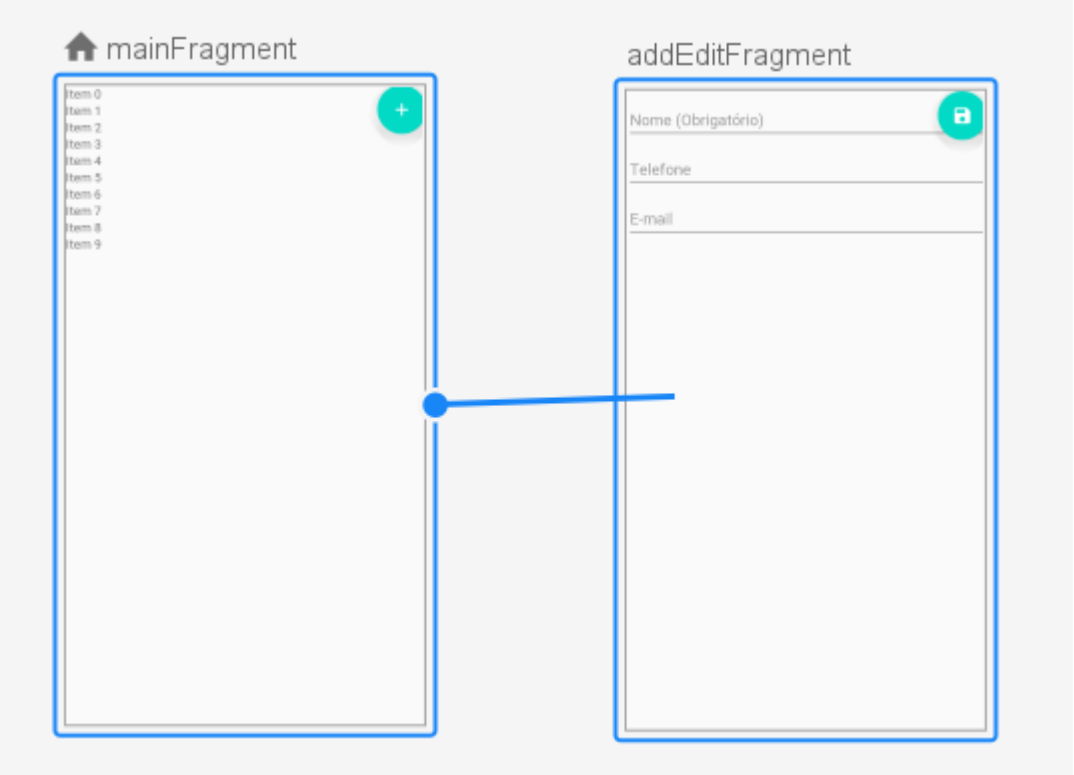


Figura 33 - Adicionando uma ligação entre duas telas

Repita isso para fazer mais duas transições:

- Transição do fragmento principal ao fragmento de detalhes
- Transição do fragmento de detalhes ao fragmento de adicionar/editar

Como resultado final teremos o seguinte grafo:

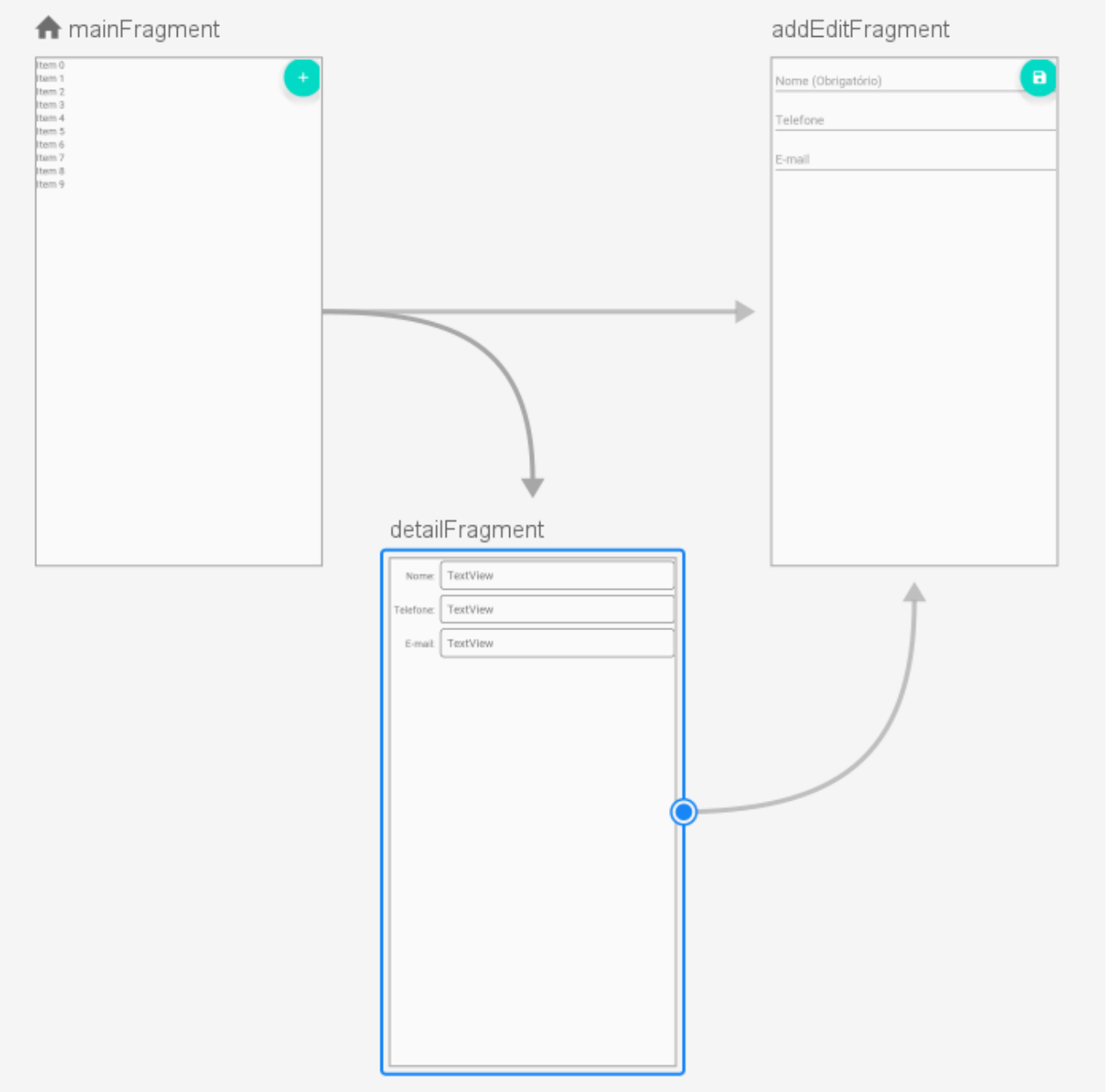


Figura 34 - Interações entre as telas do aplicativo

Por questão de simplicidade iremos omitir opções avançadas de navegação como animações e definição de argumentos, assim isso é tudo que você precisa fazer para terminar a configuração da transição das telas. Cada transição adicionado foi configurada automaticamente com o nome das telas origem e destino como: **action_mainFragment_to_addEditFragment**, **action_mainFragment_to_detailFragment** e **action_detailFragment_to_addEditFragment**. Através do nome da ação, o **NavController** consegue efetuar a navegação entre as telas. Na próxima seção vamos ver como programar os eventos para chamar essas transições.

## 10. Programando transições

Para fazer as transições entre as telas do aplicativo vamos acessar o **NavController** da atividade principal e invocar as ações das transições criadas no grafo. Por exemplo, ao se clicar no

botão de adição de um novo contato (+) o fragmento **ContactsFragment** dispara a transição para ir ao **AddEditFragment**, o **NavController** se encarrega de realizar as operações necessárias para que a transição de telas ocorra de maneira fluída.

Nas próximas etapas vamos alterar o layout da atividade principal para incorporar o elemento **NavHost** para a ativar a navegação e atualizar a programação. Na sequência vamos configurar cada fragmento e vincular a ação dos botões com as transições de telas.

## 10.1. Classe MainActivity

A classe **MainActivity** deve hospedar o elemento **NavHost** e vincular o grafo de navegação. Todas as operações de transições de tela ficaram a cargo do componente **Navigation** então a programação deve ser atualizada também.

### 4.1.1 Alteração no arquivo main_activity.xml

Precisamos adicionar o elemento **NavHost** a raiz do projeto dentro do layout da atividade principal. Assim abre o arquivo *main_activity.xml* e procure na paleta na aba *Containers* o **NavHostFragment**. Adicione ao layout principal o novo container.

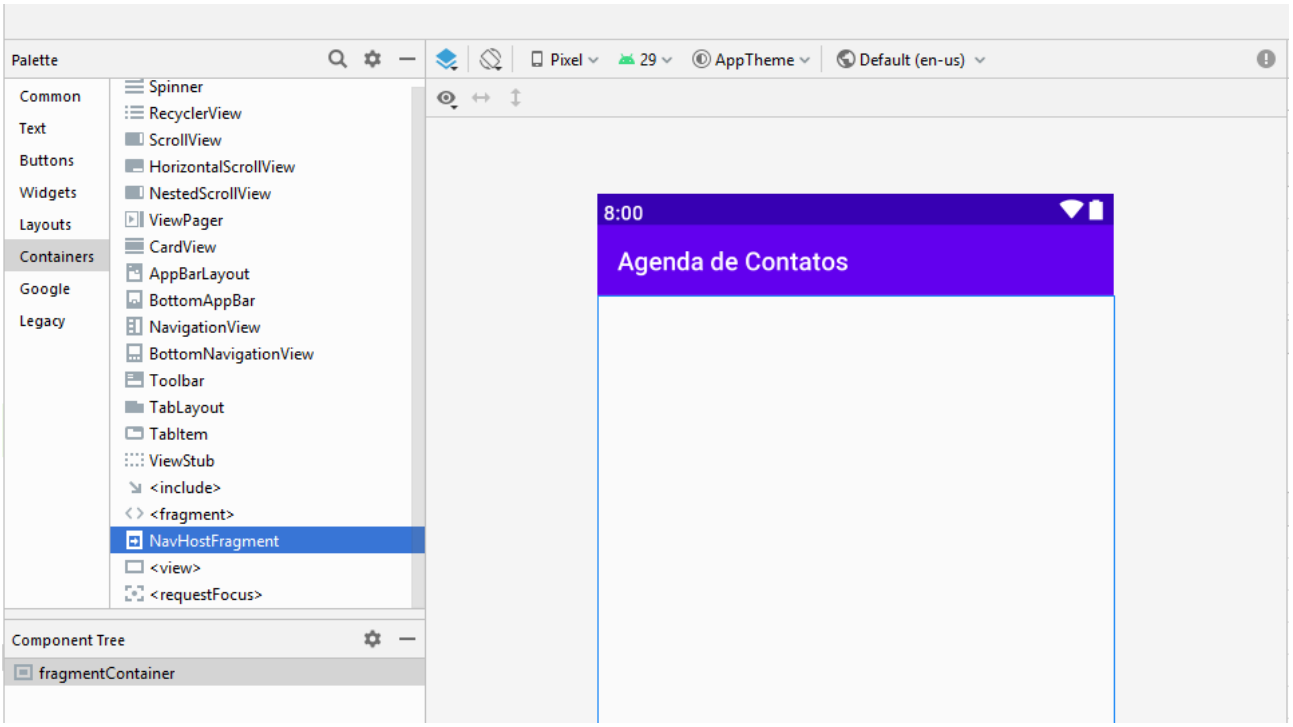


Figura 35 - Adicionando um NavHostFragment

Ao arrastar e soltar o novo elemento uma nova janela será aberta solicitando a escolha de qual grafo de navegação usar, selecione o grafo criado anteriormente, *nav_graph*, e clique em **OK**.

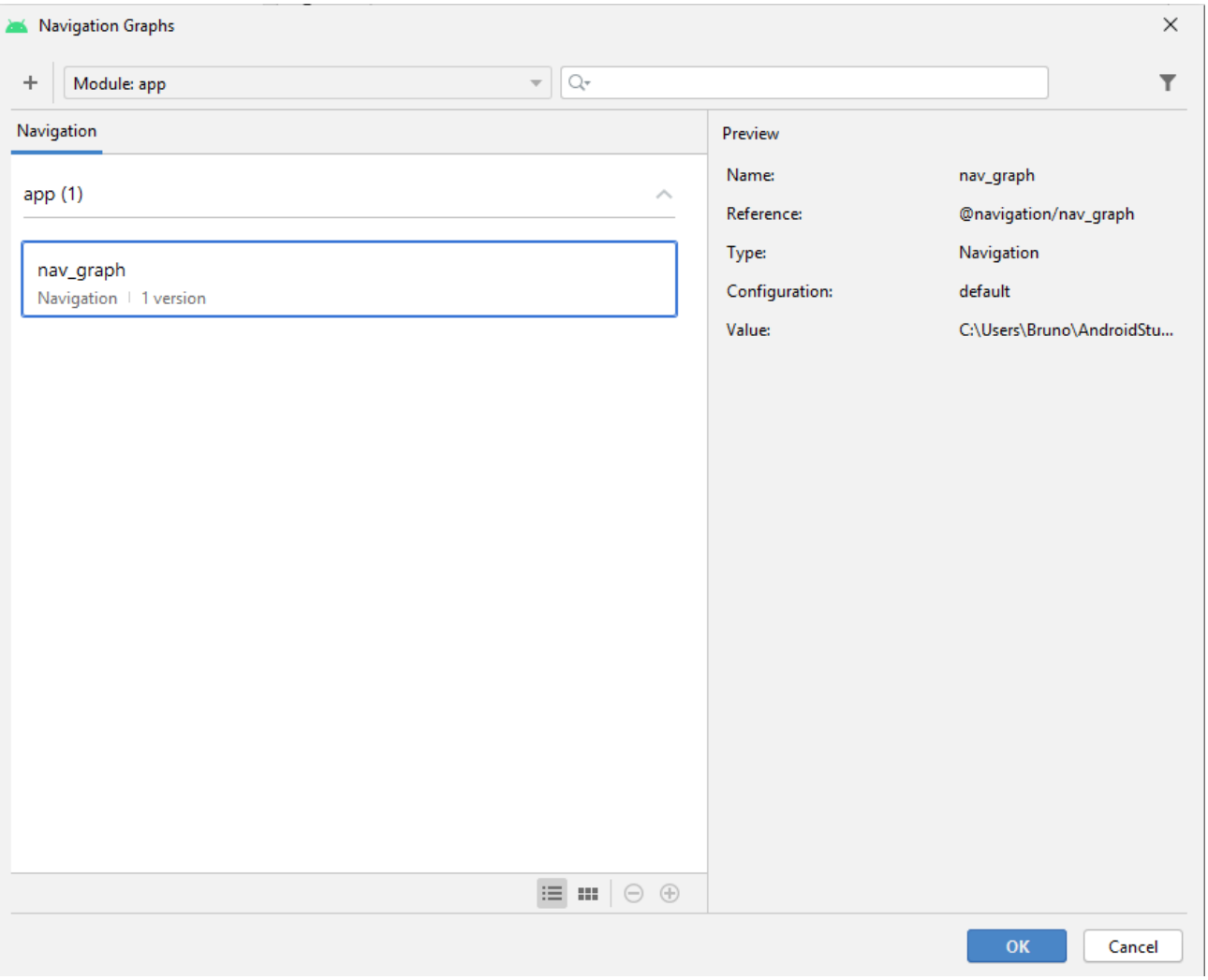


Figura 36 - Selecionando *nav_graph* como o grafo de navegação do NavHostFragment

Como resultado final você deverá ver a tela inicial da navegação, **MainFragment**, sendo apresentada na atividade principal. Isso ocorre porque a grafo configurou a tela inicial como está sendo assim não cabe mais a atividade principal controlar quais fragmentos devem ser mostrados.

### 4.1.2 Atualização do arquivo MainActivity.java

Como tudo será gerenciado pelo **NavController** a partir de agora, é necessário limpar o código fonte da atividade principal porque ele está manualmente solicitando o carregamento do fragmento principal ao se iniciar a aplicação. Assim, abra o arquivo **MainActivity.java** e remova o código anterior que fazia o carregamento do fragmento principal usando o método **getSupportFragmentManager**. Também, deve ser adicionado à atividade principal duas constantes, *CONTACT_ID* e *NEW CONTACT,* que serão usadas posteriormente pelos fragmentos para fazer a passagem de argumentos na transição das telas. Como resultado o arquivo deve ficar como:

```
public class MainActivity extends AppCompatActivity {
```

```
  public static final String CONTACT_ID = "CONTACT_ID";

  public static final int NEW_CONTACT = -1;


  @Override

  protected void onCreate(Bundle savedInstanceState) {

     super.onCreate(savedInstanceState);

     setContentView(R.layout.main_activity);

  }

}
```

## 10.2. Classe MainFragment

A classe **MainFragment** exibe a lista de contatos em uma *RecyclerView* e fornece um *FloatingActionButton* para a adição de novos contatos.

### 4.2.1 Importações e Campos

O fragmento **MainFragment** possui uma **ViewModel** para vincular os dados de agenda de contatos a uma lista. Esse campo é importante pois será usado para configurar a integração dos dados do banco de dados com a *RecyclerView*. Renomeie a variável da **ViewModel** para *mainViewModel* pois posteriormente escreveremos os demais fragmentos de maneira similar. Também, para a configuração da transição das telas precisamos fazer o uso das constantes declaradas na classe **MainActivity** por isso certifique-se de colocar as importações corretas.

```
//OBSERVAÇÃO: A localização exata dos pacotes para as importações abaixo varia de acordo com cada projeto

import static br.com.bcalegaro.agendacontatos.MainActivity.NEW_CONTACT;

import static br.com.bcalegaro.agendacontatos.MainActivity.CONTACT_ID;


public class MainFragment extends Fragment {

  private MainViewModel mainViewModel;

  ...

}
```

### 4.2.2 Método onCreateView

No método **onCreateView** vamos inicializar os componentes da interface gráfica definidos no arquivo de layout *main_fragment.xml*. Futuramente vamos vincular o adaptador a *RecyclerView*., mas nesta segunda etapa do projeto vamos apenas adicionar o tratamento de eventos do FAB.

```
@Nullable
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,
                         @Nullable Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.main_fragment, container, false);
    // TODO recyclerView, adapter, itemdivider
    // cria uma referência ao FAB e vincula uma ação
    FloatingActionButton addButton = view.findViewById(R.id.addButton);
    addButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            // Quando for clicado no FAB o método onAddContact será invocado
            onAddContact();
        }
    });
    return view;
}
```

### 4.2.3 Método onAddContact

O método **onAddContact** é disparado no toque do botão flutuante de adicionar um novo contato. Esse método deve usar a *navegação* para solicitar a transição da tela atual para o fragmento **addEditFragment**. Como se trata de um novo contato, será usada para a passagem de informações (argumentos) as constantes registradas na **MainActivity**.

```
private void onAddContact() {

  //cria um pacote de argumentos

  Bundle arguments = new Bundle();

  // adiciona o ID do contato como argumento a ser passado ao fragmento

  // como se trate de um novo contato o valor a ser encaminhado será  NEW_CONTACT

  arguments.putInt(CONTACT_ID, NEW_CONTACT);

  //solicita a transição de tela para o fragmento de adicionar um novo contato

  Navigation.findNavController(getView()).navigate(R.id.action_mainFragment_to_addEditFragment, arguments);

}
```

Para invocar uma nova tela, **fragmento**, usando o componente de arquitetura do Android Jetpack **Navigation** usamos as ações (*action*) definidas no grafo de navegação. A ação em questão é a da transição da tela *mainFragment* para *addEditFragment*. Além de informar qual ação o navegador deve executar podemos passar argumentos, dados que podem ser transmitidos de uma tela a outra.

A classe **Bundle** é um pacote de transmissão de informações entre os fragmentos e atividades que pode também ser utilizada como uma maneira de passar dados de um lado para o outro. No aplicativo da **Agenda de Contatos** as informações a serem compartilhadas entre as telas são o código único de cada contato. Neste caso específico, o botão adicionar deve enviar um código para o fragmento **addEditFragment** que sinaliza que se trata de um novo contato a ser inserido. Assim, o novo fragmento ao receber o valor *NEW_CONTACT* consegue se adaptar a situação de adição de um novo contato e não a edição de um contato já existente. Para fazer o registro de um novo argumento ao pacote (bundle) se usa o método putInt para registrar um número inteiro com a chave *CONTACT_ID*.

### 4.2.4 Método onContactSelected

O método **onContactSelected** é disparado no toque de um item da lista de contatos e usa a navegação para solicitar a transição da tela atual para a tela de detalhes, fragmento **detailFragment**. Esse método será vinculado a lista e operará da seguinte forma, ao se tocar no item da lista de contatos, a lista invoca o método passando como argumento para a função o código do contato selecionado, *contactID* (o código de um contato é um valor de identificação única usado no banco de dados, similar ao RG das pessoas, só pode haver um), navegador, então, encaminha o código do contato para o fragmento de detalhes, onde será exibido todas as informações do contato selecionado.

```
// chamado quando o contato é selecionado

private void onContactSelected(int contactID){

   //cria um pacote de argumentos

   Bundle arguments = new Bundle();

   // adiciona o contactID informado como argumento a ser passado ao fragmento

   arguments.putInt(CONTACT_ID, contactID);

   //solicita a transição de tela para o fragmento de detalhes do contato

   Navigation.findNavController(getView()).navigate(R.id.action_mainFragment_to_detailFragment, arguments);

}
```

### 4.2.5 Método onActivityCreated

O método de ciclo de vida **onActivityCreated** de Fragment é chamado depois que a atividade hospedeira de um fragmento foi criada e o método *onCreateView* terminou de executar. Usamos esse método para configurar a **ViewModel**. No entanto, nesta segunda parte do projeto apenas precisamos arrumar a nomenclatura do campo para conferir com novo nome da *mainViewModel*.

```
@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);
  mainViewModel = new ViewModelProvider(this).get(MainViewModel.class);
  // TODO: Use the ViewModel
}
```

## 10.3. Classe AddEditFragment

A classe **AddEditFragment** fornece uma interface para adicionar novos contatos ou editar os já existentes.

### 4.3.1 Importação e Campos

O fragmento **AddEditFragment** possui variáveis para fazer a referência aos seus componentes gráficos, pois o usuário deve digitar três campos: nome, telefone e e-mail. Além disso, a mesma interface é usada para adicionar e editar um contato existente, por isso será usada uma *tag* para sinalizar quando o fragmento deve ler os dados de um contato já existente e iniciar sua edição ou apenas criar um contato novo. O fragmento usará a **ViewModel** para manipular os dados do contato já existente ou novo. Também, como no fragmento anterior é necessária a adição das importações das constantes para configuração da transição das telas precisamos.

```
//OBSERVAÇÃO: A localização exata dos pacotes para as importações abaixo varia de acordo com cada projeto

import static br.com.bcalegaro.agendacontatos.MainActivity.CONTACT_ID;

import static br.com.bcalegaro.agendacontatos.MainActivity.NEW_CONTACT;


public class AddEditFragment extends Fragment {

  private int contactID; // ID do contato selecionado

  private boolean addingNewContact; // flag para sinalizar adição ou edição

  // componente FAB para salvar o contato

  private FloatingActionButton saveContactFAB;

  // ViewModel do fragmento

  private AddEditViewModel addEditViewModel;

  ...

}
```

### 4.3.2 Método onCreateView

No método **onCreateView** vamos inicializar os componentes da interface gráfica definidos no arquivo de layout *add_edit_fragment*. Futuramente vamos obter as referências aos componentes da interface gráfica. Mas para esta segunda parte do projeto vamos apenas vincular o tratamento de eventos do FAB.

```
@Override

public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,

                         @Nullable Bundle savedInstanceState) {

  super.onCreateView(inflater, container, savedInstanceState);

  // cria o fragmento com o layout do arquivo add_edit_fragment.xml

  View view = inflater.inflate(R.layout.add_edit_fragment, container, false);


  // TODO componentes TextInputLayout
```

```
    // configura o receptor de eventos do FAB
    saveContactFAB = view.findViewById(R.id.saveButton);
    saveContactFAB.setOnClickListener(saveContactButtonClicked);
    // acessa a lista de argumentos enviada ao fragmento em busca do ID do contato
    Bundle arguments = getArguments();
    contactID = arguments.getInt(CONTACT_ID);
    // verifica se o fragmento deve criar um novo contato ou editar um já existente
    if (contactID == NEW_CONTACT) {
        // usa a flag para sinalizar que é um novo contato
        addingNewContact = true;
    } else {
        // usa a flag para sinalizar que é uma edição
        addingNewContact = false;
    }
    return view;
}
```

Ao criar o fragmento, criamos um objeto *arguments* da classe **Bundle** para conseguirmos ler as informações repassadas pelo **NavController** na transição de telas. Buscamos o argumento com a chave *CONTACT_ID* e verificamos seu valor, assim conseguimos detectar se o fragmento foi invocado para uma operação de adição ou edição de um contato. A tag **addingNewContact** será usada na parte 3 do projeto, vamos programar para que em caso de edição os dados do contato existente sejam carregados do banco de dados e apresentados nos campos de edição. Caso contrário, por se tratar de um contato novo, a interface deve apresentar os campos de edição em branco.

### 4.3.3 Implementação de interface onClickListener com saveContactButtonClicked

O tratamento de eventos para o FAB do fragmento, botão flutuante salvar (saveButton), é cadastrado como a implementação da interface **saveContactButtonClicked**. O código executado acessa o sistema nativo do Android para solicitar para esconder o teclado virtual (*hideSoftInputFromWindow*). Por fim, é invocado o método **saveContact** para efetivamente salvar o novo contato.

```
private final View.OnClickListener saveContactButtonClicked = new View.OnClickListener() {

  @Override

  public void onClick(View view) {

     // oculta o teclado virtual

     ((InputMethodManager) getActivity().getSystemService(

           Context.INPUT_METHOD_SERVICE)).hideSoftInputFromWindow(

           getView().getWindowToken(), 0);

     saveContact();

  }

};
```

### 4.3.4 Método saveContact

O método **saveContact** deve ler os dados preenchidos e salvar o contato criado/editado. Sua implementação completa se dará na parte 3 deste projeto. Por ora, vamos fazer apenas a configuração da transição de telas, que neste caso é solicitar ao **NavController** voltar uma tela com o método **popBackStack**.

```
// salva informações de um contato no banco de dados

private void saveContact() {

  // TODO ler dados inseridos

  // TODO salvar dados

  // solicita o retorno à tela anterior

  Navigation.findNavController(getView()).popBackStack();

}
```

O **BackStack** é uma pilha, como o próprio nome diz, que armazena estados das telas em relação às transações. O *BackStack* permite, de forma transparente, a navegação entre os fragmentos ou atividades decorrente do empilhamento desses. O elemento no topo da pilha sempre será a tela atual, assim, remover o elemento do topo significa você voltar para a tela anterior, pop é ação de remover o estado no topo da pilha. Assim, não é necessário executar uma ação identificando origem

e destino na navegação neste caso, apenas voltar uma tela é o suficiente..

### 4.3.5 Método onActivityCreated

O método de ciclo de vida **onActivityCreated** de *Fragment* é chamado depois que a atividade hospedeira de um fragmento foi criada e o método onCreateView terminou de executar. Usamos esse método para configurar a **ViewModel**. No entanto, nesta segunda parte do projeto apenas precisamos arrumar a nomenclatura do campo para conferir com novo nome da *addEditViewModel*.

```
@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);
  addEditViewModel = new ViewModelProvider(this).get(AddEditViewModel.class);
  // TODO: Use the ViewModel
}
```

## 10.4. Classe DetailFragment

A classe **DetailFragment** fornece os detalhes de um contato selecionado.

### 4.4.1 Importações e Campos

O fragmento **DetailFragment** deve apresentar os dados de um contato específico portanto deve armazenar o código do contato selecionado em uma variável. Além disso, para prosseguir com as transições entre telas é preciso fazer a importação correta da constante *CONTACT_ID* presente na atividade principal.

```
//OBSERVAÇÃO: A localização exata dos pacotes para as importações abaixo varia de acordo com cada projeto
import static br.com.bcalegaro.agendacontatos.MainActivity.CONTACT_ID;

public class DetailFragment extends Fragment {
  private int contactID; // ID do contato selecionado
  // TODO componentes TextView
  private DetailViewModel detailViewModel; // ViewModel do fragmento
```

```
    ...
}
```

O fragmento usará a **ViewModel** para manipular os dados do contato selecionado.

### 4.4.2 Método onCreateView

O método **onCreateView** configura o fragmento para possuir um menu e faz a leitura do ID do contato recebido. A leitura do contato recebido é realizada com o objeto *arguments* da classe **Bundle**.

```
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,
                         @Nullable Bundle savedInstanceState) {
    // cria o fragmento com o layout do arquivo details_fragment.xml
    View view = inflater.inflate(R.layout.detail_fragment, container, false);
    // configura o fragmento para exibir itens de menu
    setHasOptionsMenu(true);
    // TODO componentes textview
    // acessa a lista de argumentos enviada ao fragmento em busca do ID do contato
    Bundle arguments = getArguments();
    if (arguments != null)
        contactID = arguments.getInt(CONTACT_ID);
    return view;
}
```

Para vincular um menu à uma atividade/fragmento é preciso ativar a opção através do método *setHasOptionsMenu*. A correta configuração do menu será realizada através de dois métodos auxiliares **onCreateOptionsMenu** e **onOptionsItemSelected**, implementados logo a seguir.

### 4.4.3 Métodos onCreateOptionsMenu e onOptionsItemSelected

O método **onCreateOptionsMenu** define qual menu será adicionado à atividade ou fragmento e inicia seu carregamento. Para este fragmento vamos usar recurso xml criado na parte 1 do projeto chamado *fragment_details_menu*. Para programar a ação a ser executada ao se clicar nos ícones do menu precisamos implementar o método **onOptionsItemSelected**. Esse método seleciona qual item de menu foi clicado e encaminha para a ação desejada.

```
@Override
public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
  super.onCreateOptionsMenu(menu, inflater);
  // seleciona o menu a ser mostrado no fragmento
  inflater.inflate(R.menu.fragment_details_menu, menu);
}
@Override
public boolean onOptionsItemSelected(MenuItem item) {
  switch (item.getItemId()) {
     case R.id.action_edit:
        editContact();
        return true;
     case R.id.action_delete:
        deleteContact();
        return true;
  }
  return super.onOptionsItemSelected(item);
}
```

### 4.4.4 Método editContact

Quando o usuário tocar um item de menu para excluir um contato o método **editContact** é invocado. Esse método solicita ao **NavController** a transição para o fragmento *addEditFragment* repassando o ID do contato.

```
// passa o ID do contato para editar no DetailFragmentListener
private void editContact(){
   //cria um pacote de argumentos
   Bundle arguments = new Bundle();
   // adiciona o ID do contato como argumento a ser passado ao fragmento
   arguments.putInt(CONTACT_ID, contactID);
   //solicita a transição de tela para o fragmento de editar um novo contato
   Navigation.findNavController(getView()).navigate(R.id.action_detailFragment_to_addEditFragment, arguments);
}
```

### 4.4.5 Método deleteContact

Quando o usuário tocar no item de menu para excluir um contato, o método **deleteContact** é invocado. Sua implementação completa se dará na parte 3 deste projeto, por ora, vamos programar apenas a transição de tela que neste caso será a de voltar à tela anterior. Novamente, não é necessário acionar uma ação no **NavController**, apenas invocar o método **popBackStack**.

```
// exclui um contato
private void deleteContact() {
   // TODO deletar contato
   // solicita o retorno à tela anterior
   Navigation.findNavController(getView()).popBackStack();
}
```

### 4.4.6 Método onActivityCreated

O método de ciclo de vida **onActivityCreated** de Fragment é chamado depois que a atividade hospedeira de um fragmento foi criada e o método **onCreateView** terminou de executar. Usamos esse método para configurar a **ViewModel**. No entanto, nesta segunda parte do projeto apenas arrumar a nomenclatura do campo para conferir com **detailViewModel**.

```
@Override

public void onActivityCreated(@Nullable Bundle savedInstanceState) {

  super.onActivityCreated(savedInstanceState);

  detailViewModel = new ViewModelProvider(this).get(DetailViewModel.class);

  // TODO: Use the ViewModel

}
```

# 11. Manipulação de Listas

Para finalizar a segunda parte do aplicativo vamos configurar o elemento RecyclerView do fragmento da tela inicial do aplicativo. A lista deve exibir o nome dos contatos armazenados e ao se clicar em um item deve ir para o fragmento de detalhes.

Para essa finalidade será configurada as três classes:

- **Contact** - Para representar os dados de um contato a serem exibidos na tela
- **ContactsAdapter** - Para vincular os dados de uma lista de contatos a uma RecyclerView
- **ItemDivider** - Elemento gráfico para desenhar uma linha de separação entre os nomes da lista de contato

## 11.1. Classe Contact

A classe **Contact** faz parte do pacote data e abstrair as informações de um contato. Para a segunda parte do aplicativo vamos criar o classe para conter o id, nome, telefone e e-mail. Ademais, devem ser adicionados os referidos métodos get e o construtor padrão.

```
public class Contact {

  private int id;

  private String name;

  private String phone;

  private String email;

  public Contact(int id, String name, String phone, String email) {

    this.id = id;

    this.name = name;

    this.phone = phone;
```

```
    this.email = email;
  }
  public int getId() {
    return this.id;
  }
  public String getName() {
    return this.name;
  }
  public String getPhone() {
    return this.phone;
  }
  public String getEmail() {
    return this.email;
  }
}
```

Essa classe vai ser usada para criar uma lista de contatos e futuramente será definida com anotações da biblioteca **Room** para especificar uma tabela do banco de dados.

## 11.2. Classe ContactsAdapter

Ao lidar com listas de itens no **Android** um adaptador deve ser implementado, ele é o responsável por vincular uma lista de valores aos elementos gráficos de uma *ListView*. No caso específico de uma *RecyclerView* o adaptador deve ser implementado com a herança da classe **RecyclerView.Adapter**. Neste projeto, criamos o **ContactsAdapter** para preencher o conteúdo da *RecyclerView* da aplicação.

```
public class ContactsAdapter extends RecyclerView.Adapter<ContactsAdapter.ContactsViewHolder> {
  // interface implementada por ContactsFragment para responder
  // quando o usuário toca em um item na RecyclerView
```

```
public interface ContactClickListener {

    void onClick(int contactID);

}

public class ContactsViewHolder extends RecyclerView.ViewHolder {

    private final TextView contactItemView;

    private int contactID;

    private ContactsViewHolder(View itemView) {

        super(itemView);

        contactItemView = itemView.findViewById(android.R.id.text1);

        // anexa receptor a itemView

        itemView.setOnClickListener(new View.OnClickListener() {

            @Override

            public void onClick(View view) {

                clickListener.onClick(contactID);

            }

        });

    }

    // configurar o identificador de linha do banco de dados para o contato

    public void setContactID(int contactID) {

        this.contactID = contactID;

    }

}

private List<Contact> mContacts; // Cached copy of contacts

private final ContactClickListener clickListener;

// construtor

public ContactsAdapter(ContactClickListener clickListener) {
```

```
    this.clickListener = clickListener;
}
@Override
public ContactsViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {
    // infla o layout android.R.layout.simple_list_item_1
    View view = LayoutInflater.from(
            parent.getContext()).inflate(
            android.R.layout.simple_list_item_1,
            parent, false);
    return new ContactsViewHolder(view);
}
@Override
public void onBindViewHolder(ContactsViewHolder holder, int position) {
    if (mContacts != null) {
        Contact current = mContacts.get(position);
        holder.contactItemView.setText(current.getName());
        holder.setContactID(current.getId());
    }
}
void setContacts(List<Contact> contacts){
    mContacts = contacts;
    notifyDataSetChanged();
}
@Override
public int getItemCount() {
    if (mContacts != null)
```

```
        return mContacts.size();

    else return 0;

  }

}
```

Dentro desta classe também serão definidos os seguintes elementos:

- **Interface ContactClickListener**
  - Interface que a classe *ContactsFragment* implementa para ser notificada quando o usuário toca em um contato da *RecyclerView*
  - Cada item da lista, pode disparar esse evento. Ao se clicar em um contato, o método deve notificar a *MainActivity* que um contato foi selecionado, para que a atividade principal possa exibir o componente *DetailFragment*
- **Classe ContactsViewHolder**
  - Classe para usar o padrão *ViewHolder*. Utiliza a herança do elemento *RecyclerView.ViewHolder*.
  - *ViewHolder* é um elemento que armazena os dados do item em cada posição da lista. No caso do contato, cada ViewHolder vai armazenar o ID do contato e assimilar a ação onClick indicando esse valor de ID.
- **Método onCreateViewHolder**
  - Método que infla a interface gráfica usando um layout predefinido no Android. O android.R.layout.simple_list_item_1, pode ser usado para mostrar listas simples, com apenas um texto descrevendo o item.
  - Neste aplicativo esse template é o suficiente pois iremos apenas mostrar o nome do Contato na lista.
- **Método onBindViewHolder**
  - Esse método usa o valor da posição atual, position, na lista para identificar o contato correto a partir de uma lista de contatos armazenada no adaptador.
  - Os dados do contato encontrado são repassados a *ViewHolder* informando o nome do contato para exibir como item da lista ao mesmo tempo que se armazena o ID do contato no *ViewHolder*.
  - Observe que é necessário anexar o valor do ID do contato a *ViewHolder* pois a mesma usará esse valor para encaminhar a ação ao se clicar nesse contato específico.
- **Método setContacts**
  - Esse método anexa uma nova lista de contatos ao adaptador e sinaliza para a lista ser redesenhada através do método **notifyDataSetChanged**.

## 11.3. Classe ItemDivider

A classe **ItemDivider** é utilizada no desenho da *RecyclerView*. Sua definição é simples e genérica. Ela cria um desenho de uma linha reta para separar os itens da lista.

```
public class ItemDivider extends RecyclerView.ItemDecoration{
```

```
private final Drawable divider;

// o construtor carrega a divisória de itens de lista interna

ItemDivider (Context context) {

   int[] attrs = {android.R.attr.listDivider};

   divider = context.obtainStyledAttributes(attrs).getDrawable(0);

}

// desenha as divisórias de itens de lista na RecyclerView

@Override

public void onDrawOver(Canvas c, RecyclerView parent,

                       RecyclerView.State state) {

   super.onDrawOver(c, parent, state);

   // calcula as coordenadas x esquerda/direita de todas as divisórias

   int left = parent.getPaddingLeft();

   int right = parent.getWidth() - parent.getPaddingRight();


   // para todos os itens, menos o último, desenha uma linha abaixo dele

   for (int i = 0; i < parent.getChildCount() - 1; i++) {

      View item = parent.getChildAt(i);

      // calcula as coordenadas y superior/inferior da divisória atual

      int top = item.getBottom() + ((RecyclerView.LayoutParams)

            item.getLayoutParams()).bottomMargin;

      int bottom = top + divider.getIntrinsicHeight();

      // desenha a divisória com os limites calculados

      divider.setBounds(left, top, right, bottom);

      divider.draw(c);

   }
```

```
  }

}
```

## 12. Finalizando a integração entre as telas e exibindo dados fictícios

Para finalizar o aplicativo vamos configurar o fragmento inicial para exibir uma lista com alguns dados fictícios. Assim, será possível finalizar a transição entre as telas. Para tanto, modifique a classe **MainFragment** como a seguir.

```
public class MainFragment extends Fragment {

  private MainViewModel mainViewModel;

  // adaptador para ser uma usada com a lista de contatos

  ContactsAdapter contactsAdapter;



  public static MainFragment newInstance() {

     return new MainFragment();

  }



  public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,

                           @Nullable Bundle savedInstanceState) {

     View view = inflater.inflate(R.layout.main_fragment, container, false);

     // cria uma referência a recyclerview

     RecyclerView recyclerView = view.findViewById(R.id.recyclerView);

     // Configura a recycler view para exibir os itens na vertical

     recyclerView.setLayoutManager(new LinearLayoutManager(getActivity().getBaseContext()));

     // anexar um ItemDecorator personalizado para desenhar divisórias entre os itens da lista

     recyclerView.addItemDecoration(new ItemDivider(getContext()));

     // cria um adaptador para preencher os dados da lista
```

```
    // o adaptar também vincula uma ação para quando for clicado sobre um contato
    contactsAdapter = new ContactsAdapter(new ContactsAdapter.ContactClickListener() {
        @Override
        public void onClick(int contactID) {
            // Quando clicado sobre um contato, o método onContactSelected é invocado
            onContactSelected(contactID);
        }
    });
    recyclerView.setAdapter(contactsAdapter); // vincula o adaptador à lista
    // cria uma referência ao FAB e vincula uma ação
    FloatingActionButton addButton = view.findViewById(R.id.addButton);
    addButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            // Quando for clicado no FAB o método onAddContact será invocado
            onAddContact();
        }
    });
    return view;
}

...

public void onActivityCreated(@Nullable Bundle savedInstanceState) {
    super.onActivityCreated(savedInstanceState);
```

```
        mainViewModel = new ViewModelProvider(this).get(MainViewModel.class);

        // TODO: Use the ViewModel



        ArrayList<Contact> list = new ArrayList<Contact>();

        list.add(new Contact(1,"Fulano", "1234", "a@a.com"));

        list.add(new Contact(2,"Beltrano", "4567", "b@a.com"));

        list.add(new Contact(3,"Ciclano", "8888", "c@a.com"));

        contactsAdapter.setContacts(list);

    }

}
```

As modificações incluem a inicialização e configuração do adaptador para a **RecylerView** e a adição de três contatos fictícios no método *onActivityCreated*. Neste método, uma lista de contatos é criada e repassada ao adaptador através do método *setContacts*. Lembre que o método *setContacts* recebe uma lista de valores do tipo contato, configura seus elementos gráficos e solicita o redesenho da lista na tela.

Agora, com a segunda parte do projeto, podemos rodar o aplicativo e navegar entre as telas corretamente. Ainda não serão cadastrados nem exibidos os dados corretos de cada contato, mas podemos visualizar as telas e conferir o desenho da interface gráfica. Na próxima etapa do projeto vamos configurar o banco de dados para a **Agenda de Contatos** e implementar perfeitamente as funcionalidades de criar um novo contato, exibir os dados de um contato existente, editar um contato já existente e excluir um contato selecionado.

# Parte 3 - Integração com banco de dados com a biblioteca Room

## 1. Apresentação

Na segunda parte do projeto, fizemos a configuração das transições entre os fragmentos com o componente **Navigation** e apresentação de dados fictícios.

Por fim, nesta terceira e última etapa do projeto vamos aplicar a persistência de dados da agenda de contatos com a biblioteca **Room** manipulando os dados salvos em um banco de dados **SQLite**. Também iremos implementar o padrão **ViewModel** e usar a classe **LiveData**, ambas presentes nos componentes de ciclo de vida do **Android Jetpack**.

## 2. Recursos envolvidos

No desenvolvimento da terceira parte do aplicativo, vamos usar as seguintes tecnologias presentes no **Android Jetpack**: biblioteca **Room** para manipulação de banco de dados **SQLite**,

padrão **ViewModel** e **LiveData** para fazer a vinculação de dados aos elementos da interface gráfica.

- **Componentes de Arquitetura**
  - Foi introduzido pelo **Android Jetpack** o pacote de componentes de Arquitetura (*Architecture Components*) para servir como um guia para construção de aplicativos. Neste pacote estão presentes diversas bibliotecas para tarefas comuns como gerenciamento dos ciclos de vida e persistência de dados.
  - Os componentes de arquitetura ajudam você a construir um aplicativo **robusto**, **testável** e de **fácil manutenção**.
  - Os componentes de arquitetura oferecem uma abordagem simples, flexível e prática nos livrando de ter que lidar com problemas comuns e nos focar nas funcionalidades do aplicativo.
- **Projeto de Arquitetura**
  - O projeto de Agenda de Contatos faz o uso dos componentes de arquitetura do Android e exemplifica a estrutura padrão recomendado pelo Android Jetpack:

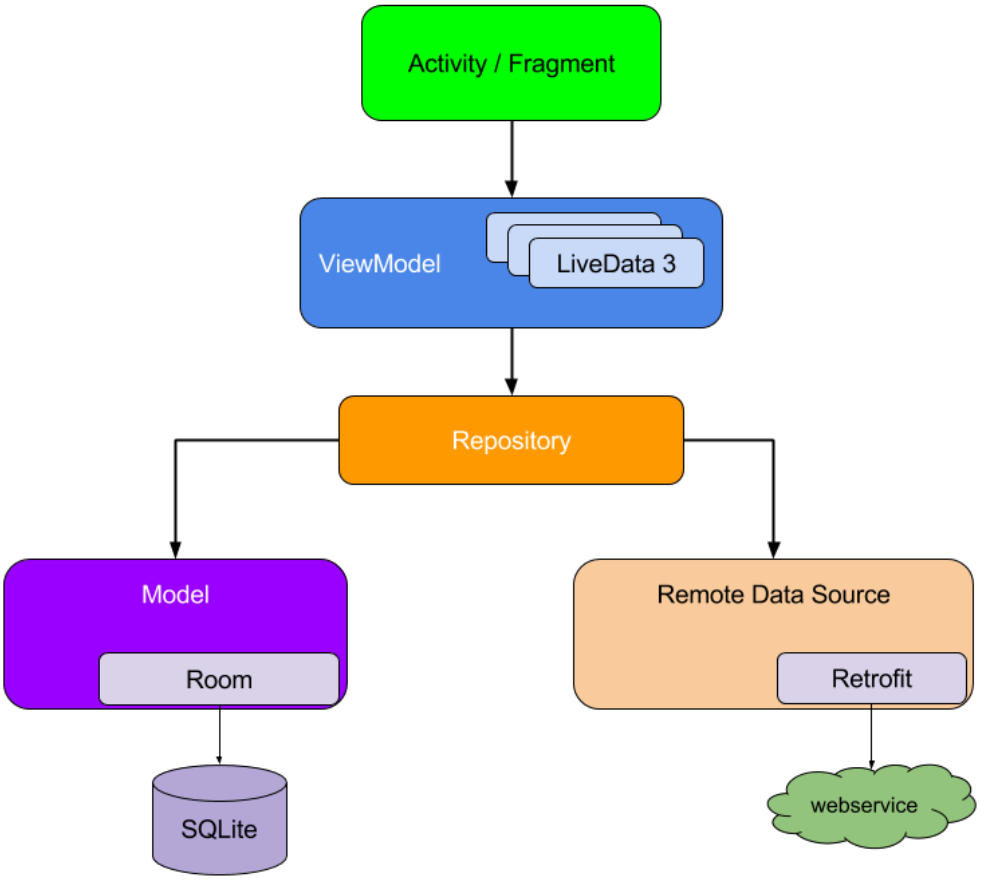


Figura 37 - Arquitetura padrão
Fonte: https://developer.android.com/jetpack/guide

  - Na primeira e segunda parte do aplicativo fizemos a implementação da primeira camada da arquitetura. Nela usamos os **fragmentos** para construir

as telas do aplicativo e configuramos as classes junto com a **atividade principal** (*main activity*) para fazer a transição entre as telas.

- Nesta terceira parte do aplicativo vamos implementar as demais camadas e introduzir novos conceitos adotados na arquitetura proposta.
  - **Entity**: Essa é uma classe de anotação utilizada para descrever uma tabela do banco de dados. Neste projeto específico, a classe **Contact** será utilizada para descrever as informações que devem ser armazenadas no banco de dados.
  - **SQLite**: banco de dados nativo do Android. Cada aplicativo pode criar e gerenciar as suas tabelas em um banco de dados *sandbox* (isto é, apenas o aplicativo pode ter acesso ao banco criado, você não pode acessar um banco de dados de outro aplicativo do seu celular).
  - **DAO**: *Data Access Object*, padrão de projeto onde se cria uma classe para conter todas as operações (SQL) de um dado objeto. Nesse projeto específico, a classe **ContactsDAO** será empregada para definir as operações sobre a tabela de contatos.
  - **RoomDatabase**: camada da biblioteca **Room** para abstrair as operações sobre um banco de dados **SQLite**. Esta classe usa os **DAOs** criados para fazer a manipulação dos dados salvos no banco de dados. Nesse projeto específico, a classe **ContactsDatabase** será empregada para fazer a criação da tabela no banco de dados.
  - **Repository**: a classe repositório é um padrão de projeto empregado para que seja uma interface de comunicação entre a interface do aplicativo e o banco de dados. A vantagem de usar esse tipo de estratégia é que o repositório pode ser programado para gerenciar dados de múltiplas fontes, por exemplo, sincronizar uma banco de dados local com um Web Service. Na Agenda de Contatos, vamos implementar a classe **ContactsRepository**.
  - **ViewModel**: uma ViewModel é uma classe cujo propósito é servir dados a uma interface gráfica (fragmento). A ViewModel fará o meio termo entre a comunicação dos dados apresentados na interface gráfica e o repositório. No aplicativo Agenda de Contatos vamos ter três ViewModels uma para cada fragmento: **MainViewModel**, **AddEditViewModel** e **DetailViewModel**.
  - **LiveData**: classe especial armazenar informações e notificar alterações em seu conteúdo. Esse tipo de classe pode ser "observado" por outro objeto e receber notificações quando seu valor é alterado garantindo que o objeto que está observando sempre tenha a última versão do conteúdo. Usamos essa classe para manipular a lista de contatos, pois a cada contato adicionado, editado ou removido, o fragmento será notificado e mostrará os dados sempre atualizados.

## 3. Biblioteca Room e persistência de dados

Para a finalização do aplicativo **Agenda de Contatos** vamos primeiramente fazer as configurações das classes utilizadas pela biblioteca **Room**. Ao configurar a persistência de dados para o lista de contatos poderemos realizar as operações de criação, edição e remoção de contatos em um banco de dados **SQLite**.

Para tanto, vamos implementar as seguintes classes:

- **Contact**
- **ContactsDAO**
- **ContactsDatabase**
- **ContactsRepository**

## 3.1. Adicionando a biblioteca Room ao projeto

Antes de começar a usar a biblioteca **Room** no projeto, primeiro precisamos adicionar suas dependências ao projeto. Para tanto, localize na pasta "Gradle Scripts" o arquivo *build.gradle (Module ...)*.

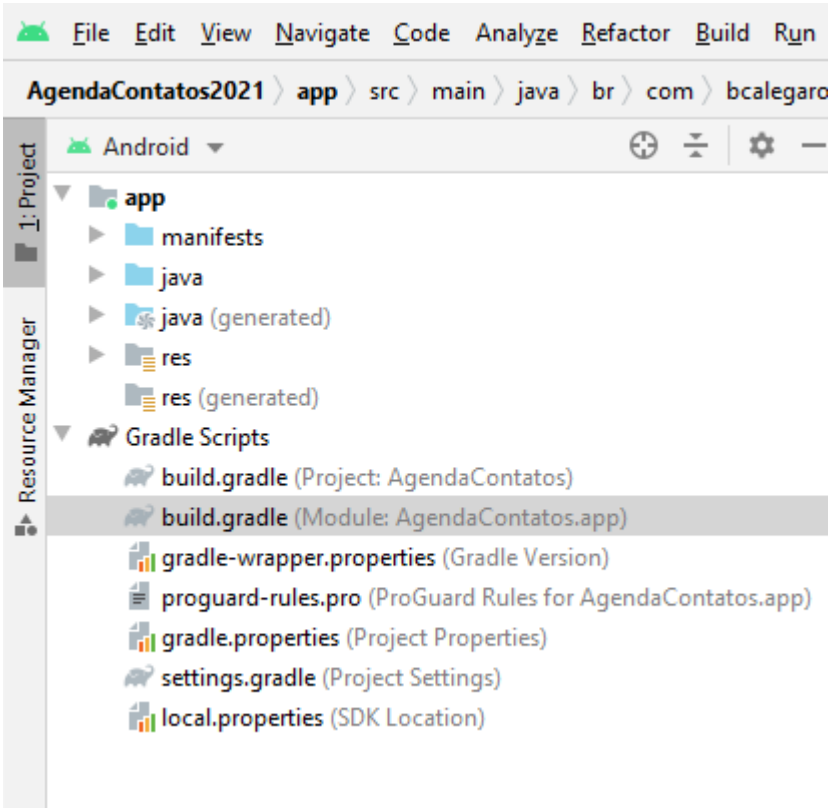


Figura 38 - Localização do arquivo *build.gradle*

Dentro deste arquivo estão presentes diversas configurações para o nosso projeto como a versão Android mínima, a versão do compilador e as dependências do projeto. O que nós precisamos fazer é adicionar os componentes da biblioteca **Room** ao campo dependencies. Se você já realizei esse passo nas configurações do projeto na parte 1 ignore essa etapa, senão certifique se de configurar o arquivo como:

```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.3.0'
    implementation 'com.google.android.material:material:1.3.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.0.4'
    implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.3.1'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1'
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
    implementation 'androidx.gridlayout:gridlayout:1.0.0'
    implementation 'androidx.navigation:navigation-fragment:2.3.4'
```

```
    implementation 'androidx.navigation:navigation-ui:2.3.4'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.2'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.3.0'

    implementation "androidx.room:room-runtime:2.2.6"
    annotationProcessor "androidx.room:room-compiler:2.2.6"
    testImplementation "androidx.room:room-testing:2.2.6"
}
```

Após adicionar as novas linhas em destaque ao arquivo, clique no botão "*Sync now*" para sinalizar ao Android Studio fazer o download e adicionar as dependências ao projeto. Ao final do processo, seu projeto estará pronto e configurado para usar a biblioteca **Room**.

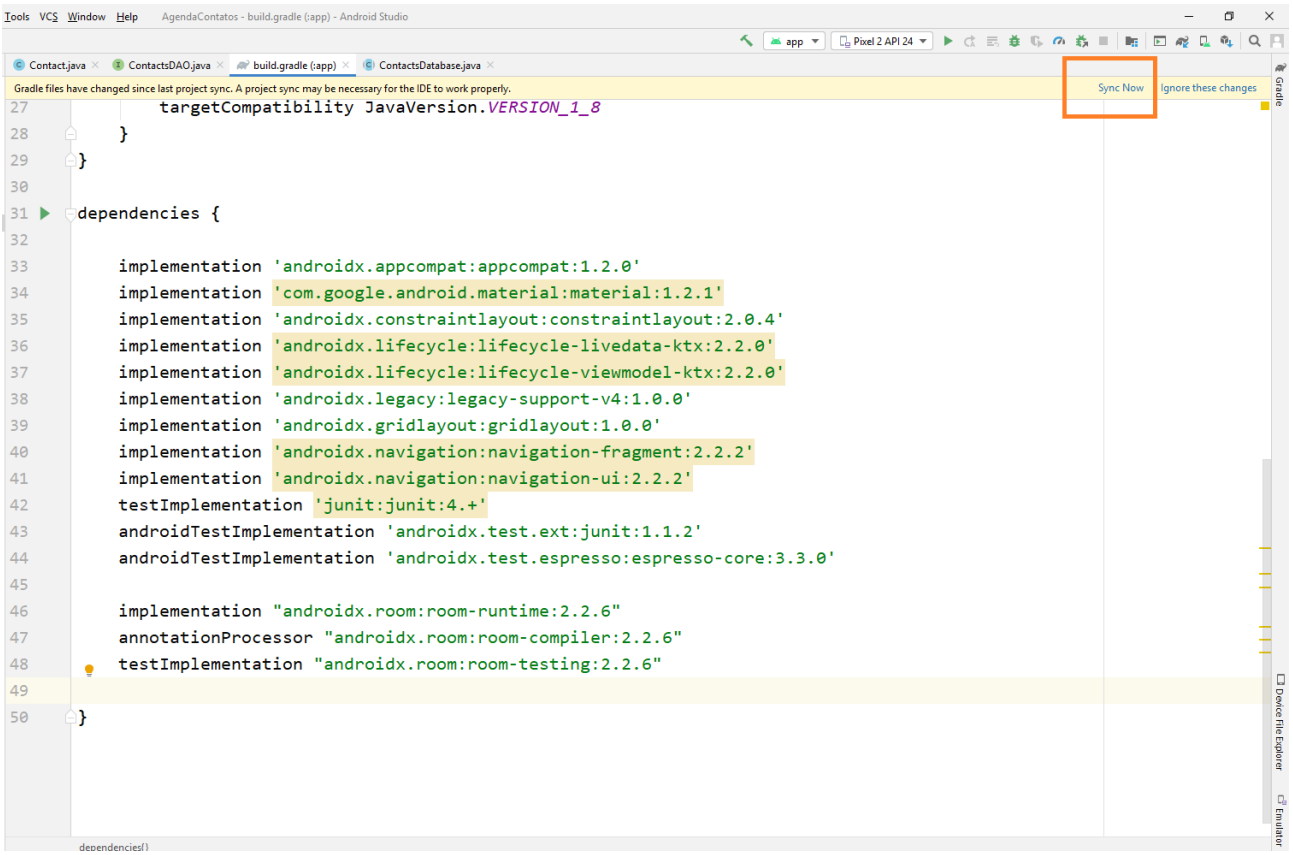


Figura 39 - Localização do botão para iniciar a sincronização do projeto com as novas alterações do arquivo build.gradle

## 3.2. Classe Contact (Entity)

A agenda de contatos proposta por este aplicativo deve armazenar informações de um contato contendo seu **nome**, **telefone** e **e-mail**. Para a biblioteca **Room**, cada contato deve representar uma **entidade** (*Entity*) e, assim, ser armazenado no banco de dados. Para conseguir fazer a persistência desses dados em um banco precisamos criar uma classe chamada **Contato** com seus métodos *get* e construtor pois é assim que a biblioteca entende como instanciar esses objetos.

Observe, no entanto, que apenas a definição clássica de uma classe não é o suficiente para a biblioteca "entender" a classe. Usamos assim um conjunto de **anotações** (*annotations*) para escrever

essa classe de modo que a biblioteca **Room** entenda exatamente como queremos que a tabela seja armazenada no banco de dados.

Dessa forma, abra o arquivo **Contact.java** e implemente o código a seguir:

```
import androidx.annotation.NonNull;

import androidx.room.ColumnInfo;

import androidx.room.Entity;

import androidx.room.Ignore;

import androidx.room.PrimaryKey;


@Entity(tableName = "contacts_table")

public class Contact {

  @PrimaryKey(autoGenerate = true)

  @NonNull

  @ColumnInfo(name = "id")

  private int id;

  @NonNull

  @ColumnInfo(name = "name")

  private String name;

  @ColumnInfo(name = "phone")

  private String phone;

  @ColumnInfo(name = "email")

  private String email;


  public Contact(int id, String name, String phone, String email) {

    this.id = id;

    this.name = name;
```

```
    this.phone = phone;

    this.email = email;

  }


  @Ignore

  public Contact(String name, String phone, String email) {

    this.name = name;

    this.phone = phone;

    this.email = email;

  }

  public int getId() {return this.id;}

  public String getName(){return this.name;}

  public String getPhone() {return this.phone;}

  public String getEmail() {return this.email;}

}
```

- **@Entity(tableName = "conacts_table")**
  - Cada classe com a anotação *entity* representa uma tabela no banco de dados. Por padrão, o próprio nome da classe identifica o nome da tabela, mas você pode especificar manualmente como no caso acima.
- **@PrimaryKey**
  - Toda entidade precisa de uma chave primária. Para a tabela de contatos será adotado como chave primária o campo **id**
- **@NonNull**
  - Identifica que o parâmetro, campo ou o valor de retorno do método nunca seja nulo
- **@ColumnInfo(name = "phone")**
  - Especifica o nome da coluna na tabela. Por padrão, se adota o próprio nome do campo, mas você pode definir manualmente.
- Observe também que todos os campos devem possuir um método get e deve existir ao menos um construtor padrão com todos os campos para se poder instanciar os objetos. Nesta classe declaramos um construtor padrão e outro com a omissão do campo **id**. O construtor padrão vai ser invocado quando usarmos comandos como *"select **"*, ou seja, queremos retornar uma lista de contatos do banco de dados. E o outro será usado quando formos adicionar um novo contato, pois ao omitir o campo

**id** a biblioteca **Room** entende e fica automaticamente carregada de criar um valor **id** válido. Ademais, para a biblioteca **Room** entender qual é o construtor padrão devemos usar a anotação **@Ignore** nos demais construtores.

Você pode encontrar a lista completa de anotações na documentação da biblioteca **Room**:

- https://developer.android.com/reference/android/arch/persistence/room/package-summary.html

## 3.3. Classe ContactsDAO (DAO)

### O que é DAO?

Em um objeto DAO (*Data Access Object*) vão ser declaradas as consultas SQL bem como sua associação a métodos específicos. O compilador verifica os comandos SQL e gera as consultas por conveniência através de anotações como **@Insert**, por exemplo.

Um objeto DAO deve ser implementado como uma *interface* ou uma classe *abstrata*. Assim, a biblioteca **Room** cria automaticamente uma API sem você precisar escrever muito código. Como resultado temos um código mais limpo.

Por padrão, todas as consultas são executadas em uma *thread* separada. Dessa forma, o programador não precisa de preocupar em implementar essa tarefa tão comum para cada consulta criada, tornando o processo do desenvolvimento do aplicativo mais rápido e produtivo!

### Implementação da classe ContactsDAO

Na Agenda de Contatos vamos usar o banco de dados **SQLite** para armazenar contatos com nome, telefone e e-mail. Nesse aplicativo, vamos usar três telas: uma para exibir a lista de contatos, uma para adicionar ou editar um contato já existente e uma para exibir os detalhes de um contato selecionado. Portanto, precisamos implementar o **DAO** para realizar as seguintes operações:

- **Pesquisar** todos os contatos salvos no banco de dados
  - e também pesquisar todas as informações de um **contato específico**
- **Inserir** um novo contato
- **Editar** um contato já existente
- **Apagar** um contato específico
  - e também apagar todos os contatos da agenda. Usaremos essa consulta na fase de testes do aplicativo.
  -

O código completo da classe **ContactsDAO** fica como:

```
import androidx.lifecycle.LiveData;

import androidx.room.Dao;

import androidx.room.Delete;

import androidx.room.Insert;
```

```
import androidx.room.Query;

import androidx.room.Update;

import java.util.List;



@Dao

public interface ContactsDAO {

  @Query("SELECT * from contacts_table ORDER BY name ASC")

  LiveData<List<Contact>> getAllContacts();

  @Query("SELECT * from contacts_table WHERE id=:id")

  LiveData<Contact> getContactById(int id);

  @Insert

  void insert(Contact contact);

  @Update

  void update(Contact contact);

  @Delete

  void delete(Contact contact);

  @Query("DELETE FROM contacts_table")

  void deleteAll();

}
```

- **@DAO**
  - Anotação usada para identificar a classe como um **DAO**
- **@Query**
  - Anotação usada para declarar uma consulta manualmente ao banco de dados. Declare entre os parênteses a consulta desejada. Logo abaixo da anotação deve ser implementado um método para executar a consulta.
  - Neste projeto duas consultas são declaradas, uma para buscar todos os contatos e outra para buscar apenas o contato com o **id** selecionado.
  - Observe que em ambos os casos o objeto de retorno é um dado do tipo **LiveData<Tipo>**. Esse é o retorno padrão dos métodos da biblioteca **Room** e é um dos componentes de arquitetura do **Android Jetpack**.

- **@Insert, @Update, @Delete**
  - Anotação para uma consulta do do tipo *insert* (inserção), *update* (atualização) e *delete* (remoção). Esse tipo de operação é extremamente comum, logo não é necessário especificar nenhum comando SQL. Mas se for necessária a criação de uma consulta personalizada você teria que incluir o comando SQL específico de maneira similar como foi definida na *query* para um contato específico..
- **Método deleteAll**
  - Não existe uma anotação para deletar todos os dados de uma tabela, logo devemos usar **@Query** para declarar manualmente o comando SQL para esse fim. De fato, para todos os casos em que não exista uma anotação conveniente, o comando SQL deve ser declarado explicitamente com a anotação **@Query**.

### Classe LiveData

Quando os dados mudam você provavelmente deseja executar alguma ação, como exibir os dados atualizados na tela. Isso significa que você deve observar esses dados para perceber essas mudanças e ter uma reação. Dependendo de como os dados são armazenados, isso pode ser complicado. Observar mudanças nos dados através de múltiplos componentes dentro do aplicativo pode criar, explicitamente, dependências rígidas entre os componentes. Isso torna a tarefa de depurar e testar o código mais difícil, entre outras coisas.

**LiveData**, classe para observar dados, presente na biblioteca de componentes de ciclo de vida (*lifecycle*) dentro do **Android Jetpack**, representa uma solução para esse problema. A biblioteca **Room** usa esse tipo de dado para ser o valor de retorno das consultas ao banco e gera automaticamente todo o código necessário para atualizar a **LiveData** quando o banco de dados é atualizado.

## 3.4. Classe ContactsDatabase (Room database)

### O que é uma Room database?

**Room database** é uma camada acima do banco de dados **SQLite**. A biblioteca **Room** simplifica a manipulação de banco de dados e se encarrega das tarefas comuns de acessar ou criar um banco de dados **SQLite**

- **Room** usa os objetos **DAO** para fazer consultas a seu banco de dados.
- Por padrão, não se deve fazer operações demoradas na *thread* principal do aplicativo, tais como as consultas SQL a um banco de dados, pois isso fará com que a interface gráfica trave. Você já deve ter visto a mensagem do Android: "O aplicativo está demorando e responder deseja encerrar?", não queremos isso em nossos aplicativos! .Portanto, para evitar essa situação devemos SEMPRE realizar operações demoradas em segundo plano. Felizmente, o casamento entre a biblioteca **Room** e a classe **LiveData** aplica essa regra automaticamente e executa todas as consultas em uma thread em segundo plano.
- **Room** oferece verificações em tempo de compilação para os comandos SQL. Sem mais erros de sintaxe em tempo de execução!
- Sua classe *Room* deve ser declarada como *abstrata* e extender a classe

**RoomDatabase**.

- Normalmente, você precisará apenas de uma única instância da *Room database* em todo aplicativo. Assim, é comum empregar o padrão de projeto **Singleton** para garantir a existência de apenas um objeto acessando o banco de dados.

### Implementação da classe ContactsDatabase

A criação da classe **ContactsDatabase** usa anotações para declarar as tabelas do banco de dados e o padrão **Singleton** para garantir a existência de apenas uma instância da classe em todo o aplicativo. Além disso, nesta fase de construção do aplicativo vamos criar um *callback*, que nada mais é que um método para ser executado junto a criação do banco de dados, para gerar automaticamente alguns valores iniciais para o nosso banco.

```
import android.content.Context;

import android.os.AsyncTask;



import androidx.annotation.NonNull;

import androidx.room.Database;

import androidx.room.Room;

import androidx.room.RoomDatabase;

import androidx.sqlite.db.SupportSQLiteDatabase;



@Database(entities = {Contact.class}, version = 1, exportSchema = false)

public abstract class ContactsDatabase extends RoomDatabase {

  public abstract ContactsDAO contactsDao();



  private static volatile ContactsDatabase INSTANCE;



  static ContactsDatabase getDatabase(final Context context) {

    if (INSTANCE == null) {

      synchronized (ContactsDatabase.class) {
```

```
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(context.getApplicationContext(),
                        ContactsDatabase.class, "contacts_database")
                        // remover essa linha na versão final
                        .addCallback(sRoomDatabaseCallback)
                        .build();
            }
        }
    }
    return INSTANCE;
}

private static RoomDatabase.Callback sRoomDatabaseCallback =
        new RoomDatabase.Callback(){

            @Override
            public void onOpen (@NonNull SupportSQLiteDatabase db){
                super.onOpen(db);
                new PopulateDbAsync(INSTANCE).execute();
            }
        };
private static class PopulateDbAsync extends AsyncTask<Void, Void, Void> {
    private final ContactsDAO mDao;

    PopulateDbAsync(ContactsDatabase db) {
        mDao = db.contactsDao();
```

```
    }


    @Override
    protected Void doInBackground(final Void... params) {
        mDao.deleteAll();
        Contact word = new Contact("Hello", "", "");
        mDao.insert(word);
        word = new Contact("World", "", "");
        mDao.insert(word);
        return null;
    }
  }
}
```

- Todo banco de dados deve usar a herança da classe **RoomDatabase**
- **@Database**
  - Anotação para identificar a criação de uma **Room database**. Você deve declarar todas as *entidades* para usar no banco e também o número de versão. Cada entidade vai ser usada para criar as tabelas no banco de dados.
- Declaramos todos os objetos **DAO** para a manipulação das consultas ao banco de dados provendo um método *abstrato* para cada **@DAO**
- O padrão **Singleton** exige que a classe possua uma referência a uma instância dela mesma e um método *getDatabase*. Dessa forma, para obter um novo objeto ContactsDatabase deve ser solicitado esse método get onde ele verifica a existência de uma instância existente e retorna esse valor ou, se é a primeira vez que o método é invocado (logo a classe ainda não foi instanciada) então ele cria uma instância do objeto
- O método **databaseBuilder** é o responsável pela efetiva criação do banco de dados. Ele cria para o contexto da aplicação um banco de dados a partir da definição da classe **ContactsDatabase** e nomeia-o para "*contacts_database*"
- Após a criação do banco é configurado uma ação para executar através do método *addCallBack*. A ação a ser executada irá através de uma **AsyncTask** adicionar alguns valores iniciais para o banco de dados.

## 3.5. Classe ContactsRepository (Repository)

### O que é um Repositório?

A classe *Repository* é uma classe que abstrai o acesso a múltiplas fontes de dados. O repositório não é um componente de arquitetura do Android JetPack mas é uma prática recomendada. Um repositório manipula operações sobre as fontes de dados e oferece uma API limpa para ser usada pelo resto da aplicação. A figura abaixo ilustra como a camada repositório se encaixa na arquitetura do projeto. Note que UI se refere a *User Interface*, ou seja, a interface gráfica do usuário.

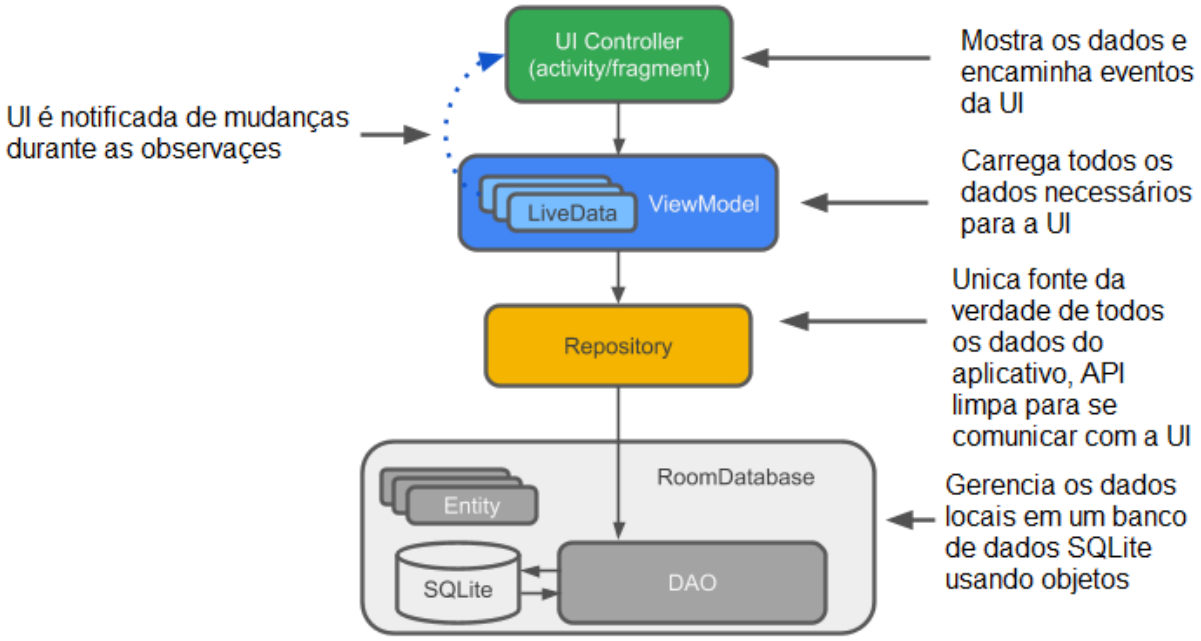


Figura 40 - Arquitetura padrão com a biblioteca Room
Fonte: https://google-developer-training.github.io/android-developer-fundamentals-course-concepts-v2/unit-4-saving-user-data/lesson-10-storing-data-with-room/10-1-c-room-livedata-viewmodel/10-1-c-room-livedata-viewmodel.html

### Porque usar um Repositório?

Um repositório gerencia as *threads* das consultas e permite o uso de múltiplas fontes de dados. Em um caso de uso mais comum, o repositório implementa a lógica para decidir se os dados devem ser obtidos através de um Web Service ou de uma *cache* armazenada em um banco de dados local.

### Implementando a classe ContactsRepository

A classe **ContactsRepository** implementará todas as operações necessárias para o aplicativo **Agenda de Contatos**. Para cada consulta ao banco será usada uma **AsyncTask** para executar a operação em plano de fundo.

```
import android.app.Application;

import android.os.AsyncTask;

import androidx.lifecycle.LiveData;

import java.util.List;


public class ContactsRepository {

  private ContactsDAO mContactsDao;

  private LiveData<List<Contact>> mAllContacts; // create a cached data


  public ContactsRepository(Application application) {

     ContactsDatabase db = ContactsDatabase.getDatabase(application);

     mContactsDao = db.contactsDao();

     mAllContacts = mContactsDao.getAllContacts();

  }


  /**********************************************

   GET ALL CONTACTS

   **********************************************/

  public LiveData<List<Contact>> getAllContacts() {

     return mAllContacts;

  }


  /**********************************************

   GET CONTACT BY ID

   **********************************************/

  public LiveData<Contact> getContactById(int id) {
```

```
    return mContactsDao.getContactById(id);
}

/**********************************************
 INSERT CONTACT TASKS
 **********************************************/
public void insert (Contact contact) {
    new insertAsyncTask(mContactsDao).execute(contact);
}

private static class insertAsyncTask extends AsyncTask<Contact, Void, Void> {
    private ContactsDAO mAsyncTaskDao;

    insertAsyncTask(ContactsDAO dao) {
        mAsyncTaskDao = dao;
    }

    @Override
    protected Void doInBackground(final Contact... params) {
        mAsyncTaskDao.insert(params[0]);
        return null;
    }
}

/**********************************************
 UPDATE CONTACT TASKS
```

```
 *********************************************/
public void update (Contact contact) {
    new updateAsyncTask(mContactsDao).execute(contact);
}

private static class updateAsyncTask extends AsyncTask<Contact, Void, Void> {
    private ContactsDAO mAsyncTaskDao;

    updateAsyncTask(ContactsDAO dao) {
        mAsyncTaskDao = dao;
    }

    @Override
    protected Void doInBackground(final Contact... params) {
        mAsyncTaskDao.update(params[0]);
        return null;
    }
}

/**********************************************
 DELETE CONTACT TASKS
 *********************************************/
public void delete (Contact contact) {
    new deleteAsyncTask(mContactsDao).execute(contact);
}
```

```
    private static class deleteAsyncTask extends AsyncTask<Contact, Void, Void> {

        private ContactsDAO mAsyncTaskDao;



        deleteAsyncTask(ContactsDAO dao) {

            mAsyncTaskDao = dao;

        }



        @Override

        protected Void doInBackground(final Contact... params) {

            mAsyncTaskDao.delete(params[0]);

            return null;

        }

    }

}
```

- O repositório cria dois objetos em seu construtor para serem usados ao longo do aplicativo. **mContactsDAO** representa o **DAO** para executar as operações no banco de dados e **mAllContacts** representa a lista de contatos.
- Observe que não foi implementado um método para buscar todos os contatos. Isso ocorre durante o construtor e será invocado logo no início do aplicativo apenas uma vez. Quando um contato for adicionado, alterado ou removido, você deve imaginar que a lista de contatos deveria ser atualizada e um nova consulta ao banco de dados solicitado para exibir na tela essas novas informações. No entanto, estamos usando **LiveData** e assim não precisamos fazer uma nova solicitação pois qualquer mudança na lista de contatos será automaticamente processada! Assim, ao ser chamado o método *getAllContacts* podemos simplesmente retornar a lista de objetos **LiveData** sem ter que fazer explicitamente uma nova solicitação ao banco de dados com o **DAO**.
- Ademais, no caso de ser buscado as informações de um contato específico precisamos apenas fazer a solicitação ao **DAO** pois ele irá automaticamente buscar o valor salvo na cache da consulta já feita logo não será uma tarefa demorada.
- Para as demais operações, usamos uma **AsyncTask** para configurar uma ação em segundo plano e executar o método apropriado do **DAO**.

### Classe AsyncTask

A classe **AsyncTask** é um classe que permite facilmente a declaração de ações para serem executadas em segundo plano sem você ter que manipular explicitamente *threads*. Essa classe é criada com a declaração de três tipos respectivamente um tipo associado aos parâmetros, progresso e resultado. Esses tipos serão usados nos métodos *onPreExecute*, *doInBackground*, *onProgressUpdate* e *onPostExecute*. Todo código declarado dentro do método *doInBackground* será executado em uma *thread* paralela. Você pode então fazer uma solicitação a um Web Service em paralelo e após receber os dados executar uma ação, como atualizar os valores na tela, com o método *onPostExecute*.

No caso do aplicativo da **Agenda de Contatos** não precisamos de uma ação de retorno pois apenas iremos fazer solicitações para o banco de dados. Logo, usar apenas o método *doInBackground* é o suficiente. Você poderia, no entanto, mostrar uma mensagem de sucesso após uma consulta no banco usando o método *onPostExecute*.

# 4. ViewModel e vinculação de dados

### O que é uma ViewModel?

O papel de uma *ViewModel*, é prover dados para a interface gráfica do usuário e sobreviver a mudanças nas configurações (Ciclo de Vida). Uma *ViewModel* atua como uma comunicação entre o **repositório** e a interface gráfica (**atividade/fragmento**). Você também pode usar uma *ViewModel* para compartilhar dados entre fragmentos diferentes. A classe **ViewModel** é um dos componentes da biblioteca *LifeCycle* do **Android Jetpack**.

### Por que usar uma ViewModel?

Uma *ViewModel* guarda os dados de um aplicativo e observa os seus ciclos de vida de forma para garantir que eles sobrevivem a mudanças nas configurações. Uma simples mudança como girar a tela do dispositivo faz com que uma atividade/fragmento seja reconstruído o que pode acarretar perda das informações. *ViewModel* resolvem esse problema pois separa os dados da **Activity/Fragment**.

Separar os dados da interface gráfica é uma maneira elegante de seguir o princípio da **responsabilidade única**: a atividade e o fragmento ficam responsáveis em cuidar da apresentação dos dados na tela, enquanto a *ViewModel* se encarrega de armazenar e processar os dados necessários na interface gráfica.

Em uma *ViewModel*, se usa a classe *LiveData* para armazenar e observar os valores para serem usados ou mostrados pela interface gráfica. Usar *LiveData* trás alguns benefícios:

- Você pode observar os dados e configurar para que mudanças sejam automaticamente processadas e mostradas na interface gráfica
- O repositório e a interface gráfica estão separados da *ViewModel*. Não existem chamadas diretas ao banco de dados pela *ViewModel*, o que facilita a criação de testes unitários.

## 4.1. MainViewModel

### Implementação da MainViewModel

A **MainViewModel** deve ser implementada para armazenar uma lista de contatos. Para tanto, usamos um elemento *LiveData<List<Contact>>* de forma que seja armazenada uma lista de contatos usando a classe *LiveData*. Durante a criação da ViewModel, é solicitado ao repositório da aplicação a lista de contatos. Como esses dados são representados por uma *LiveData* o fragmento pode solicitar esses dados com o método *getAllContacts* e configurar a observação desses dados para que mudanças no banco de dados reflitam nos dados apresentados na interface gráfica.

```
import android.app.Application;

import androidx.lifecycle.AndroidViewModel;

import androidx.lifecycle.LiveData;

import java.util.List;


//OBSERVAÇÃO: A localização exata dos pacotes para as importações abaixo varia de acordo com cada projeto

import br.com.bcalegaro.agendacontatos.data.Contact;

import br.com.bcalegaro.agendacontatos.data.ContactsRepository;


public class MainViewModel extends AndroidViewModel {

  private LiveData<List<Contact>> mAllContacts;

  private ContactsRepository mRepository;


  public MainViewModel (Application application) {

     super(application);

     mRepository = new ContactsRepository(application);

     mAllContacts = mRepository.getAllContacts();

  }

  public LiveData<List<Contact>> getAllContacts() { return mAllContacts; }

}
```

## Vinculação dos dados no MainFragment

Para fazer a vinculação dos dados da *ViewModel* ao fragmento vamos modificar o método onActivityCreated de **MainFragment** da seguinte forma:

```
@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);
  mainViewModel = new ViewModelProvider(this).get(MainViewModel.class);
  // configura a observação da lista de contatos para atualizar a lista
  // quando detectada uma mudança
  mainViewModel.getAllContacts().observe(getViewLifecycleOwner(), new Observer<List<Contact>>() {
    @Override
    public void onChanged(@Nullable final List<Contact> contacts) {
      // Atualiza a lista de contatos do adaptador
      contactsAdapter.setContacts(contacts);
    }
  });
}
```

Ao fazer essa configuração, configuramos que mudanças na lista de contato disparam automaticamente a atualização dos dados no adaptador e, por tanto, nos dados exibidos na interface gráfica.

## 4.2. AddEditViewModel

### Implementação da AddEditViewModel

A **AddEditViewModel** deve ser implementada para armazenar um contato específico e abstrair as solicitação ao repositório para adicionar um novo contato ou editar um já existente. Criamos o método *getContactById* para solicitar ao repositório um contato a partir de seu **id** e os métodos *insert* e *update* para o encaminhamento das operações ao repositório.

```
import android.app.Application;

import androidx.lifecycle.AndroidViewModel;
```

```
import androidx.lifecycle.LiveData;


//OBSERVAÇÃO: A localização exata dos pacotes para as importações abaixo varia de acordo com cada projeto
import br.com.bcalegaro.agendacontatos.data.Contact;
import br.com.bcalegaro.agendacontatos.data.ContactsRepository;


public class AddEditViewModel extends AndroidViewModel {
  private LiveData<Contact> contact;
  private ContactsRepository mRepository;

  public AddEditViewModel (Application application) {
     super(application);
     mRepository = new ContactsRepository(application);
  }

  public LiveData<Contact> getContactById(int id) {
     contact = mRepository.getContactById(id);
     return contact;
  }

  public void insert(Contact contact) { mRepository.insert(contact); }

  public void update(Contact contact) { mRepository.update(contact); }
}
```

## Vinculação dos dados na AddEditFragment

Para fazer a vinculação dos dados da *ViewModel* ao fragmento vamos modificar o método *onActivityCreated* de **AddEditFragment**. Nesse fragmento devemos, ou criar um novo contato e assim apresentar dados em branco nas caixas de texto, ou buscar as informações de um contato específico e apresentá-las na tela. Fazemos isso através da solicitação a *ViewModel* com o método *getContactById* e configurando que ao ser retornado os dados as informações na tela seja atualizada.

```
@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);
  // cria uma ViewModel para o fragmento
  addEditViewModel = new ViewModelProvider(this).get(AddEditViewModel.class);
  // se estiver editando um contato existente atualiza a tela com os valores
  if (addingNewContact == false) {
    // usa a ViewModel para solicitar a busca pelo novo contato
    addEditViewModel.getContactById(contactID).observe(getViewLifecycleOwner(), new Observer<Contact>() {
      @Override
      public void onChanged(@Nullable final Contact contact) {
        // atualiza as informações da tela com os dados do contato lido
        nameTextInputLayout.getEditText().setText(contact.getName());
        phoneTextInputLayout.getEditText().setText(contact.getPhone());
        emailTextInputLayout.getEditText().setText(contact.getEmail());
      }
    });
  }
}
```

Ademais, precisamos modificar o método *saveContact* para encaminhar as solicitações corretas ao repositório. Ou seja, as solicitações de inserção de um novo contato ou edição de um contato já existente.

Primeiramente adicione os seguintes campos a classe:

```
// componentes EditText para informações de contato
private TextInputLayout nameTextInputLayout;
private TextInputLayout phoneTextInputLayout;
private TextInputLayout emailTextInputLayout;
```

Inicialize os campos corretamente na criação do fragmento:

```
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,
                         @Nullable Bundle savedInstanceState) {
  super.onCreateView(inflater, container, savedInstanceState);
  // cria o fragmento com o layout do arquivo add_edit_fragment.xml
  View view = inflater.inflate(R.layout.add_edit_fragment, container, false);
  // obtém as referências dos componentes
  nameTextInputLayout = view.findViewById(R.id.nameTextInputLayout);
  phoneTextInputLayout = view.findViewById(R.id.phoneTextInputLayout);
  emailTextInputLayout = view.findViewById(R.id.emailTextInputLayout);
  // configura o receptor de eventos do FAB
  saveContactFAB = view.findViewById(R.id.saveButton);
  saveContactFAB.setOnClickListener(saveContactButtonClicked);
  // acessa a lista de argumentos enviada ao fragmento em busca do ID do contato
  Bundle arguments = getArguments();
  contactID = arguments.getInt(CONTACT_ID);
  // verifica se o fragmento deve criar um novo contato ou editar um já existente
  if (contactID == NEW_CONTACT) {
     // usa a flag para sinalizar que é um novo contato
```

```
        addingNewContact = true;
    } else {
        // usa a flag para sinalizar que é uma edição
        addingNewContact = false;
    }
    return view;
}
```

Por fim, modifique o método *saveContact*.

```
// salva informações de um contato no banco de dados
private void saveContact() {
    // faz a leitura dos dados inseridos
    String name = nameTextInputLayout.getEditText().getText().toString();
    String phone = phoneTextInputLayout.getEditText().getText().toString();
    String email = emailTextInputLayout.getEditText().getText().toString();
    // caso for adição de uma novo contato
    if (addingNewContact) {
        // cria um contato sem um ID pois ele será adicionado automaticamente no banco de dados
        Contact contact = new Contact(name, phone, email);
        // solicita a ViewModel a inserção do novo contato
        addEditViewModel.insert(contact);
    } else {
        // cria um contato com o mesmo ID e atualiza o seus valores
        Contact contact = new Contact(contactID, name, phone, email);
        // solicita a ViewModel a atualização do contato
```

```
    addEditViewModel.update(contact);
  }
  //Solicita a navegação voltar uma tela
  Navigation.findNavController(getView()).popBackStack();
}
```

## 4.3. DetailViewModel

### Implementação da DetailViewModel

A **DetailViewModel** deve ser implementada para armazenar um contato específico e abstrair a solicitação ao repositório de apagar um contato já existente. Criamos o método *getContactById* para solicitar ao repositório um contato a partir de seu **id** e o método delete para o encaminhamento da operação ao repositório.

```
import android.app.Application;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;

//OBSERVAÇÃO: A localização exata dos pacotes para as importações abaixo varia de acordo com cada projeto
import br.com.bcalegaro.agendacontatos.data.Contact;
import br.com.bcalegaro.agendacontatos.data.ContactsRepository;

public class DetailViewModel extends AndroidViewModel {
  private LiveData<Contact> mContact;
  private ContactsRepository mRepository;

  public DetailViewModel (Application application) {
    super(application);
```

```
    mRepository = new ContactsRepository(application);
  }

  public LiveData<Contact> getContactById(int id) {
    mContact = mRepository.getContactById(id);
    return mContact;
  }

  public void delete() { mRepository.delete(mContact.getValue()); }
}
```

### Vinculação dos dados na DetailFragment

Para fazer a vinculação dos dados da *ViewModel* ao fragmento vamos modificar o método *onActivityCreated* de **DetailFragment**. Nesse fragmento devemos, ou criar um novo contato e assim apresentar dados em branco nas caixas de texto, ou buscar as informações de um contato específico e apresentá-las na tela. Fazemos isso através da solicitação a *ViewModel* com o método *getContactById* e configurando que ao ser retornado os dados as informações na tela sejam atualizadas.

Primeiramente, adicione os seguintes campos na classe:

```
// componentes TextView para informações de contato
private TextView nameTextView;
private TextView phoneTextView;
private TextView emailTextView;
```

Configure a inicialização dos campos:

```
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container,
```

```
                @Nullable Bundle savedInstanceState) {
  super.onCreateView(inflater, container, savedInstanceState);
  // cria o fragmento com o layout do arquivo details_fragment.xml
  View view = inflater.inflate(R.layout.detail_fragment, container, false);
  // configura o fragmento para exibir itens de menu
  setHasOptionsMenu(true);
  // obtém as referências dos componentes
  nameTextView = (TextView) view.findViewById(R.id.nameTextView);
  phoneTextView = (TextView) view.findViewById(R.id.phoneTextView);
  emailTextView = (TextView) view.findViewById(R.id.emailTextView);
  // acessa a lista de argumentos enviada ao fragmento em busca do ID do contato
  Bundle arguments = getArguments();
  if (arguments != null)
     contactID = arguments.getInt(CONTACT_ID);
  return view;
}
```

Por fim, configure a *ViewModel* para buscar as informações de um contato e atualizar os dados na tela.

```
@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
  super.onActivityCreated(savedInstanceState);
  detailViewModel = new ViewModelProvider(this).get(DetailViewModel.class);
  detailViewModel.getContactById(contactID).observe(getViewLifecycleOwner(), new Observer<Contact>() {
     @Override
     public void onChanged(@Nullable final Contact contact) {
```

```
        // atualiza as informações da tela com os dados do contato lido
        nameTextView.setText(contact.getName());
        phoneTextView.setText(contact.getPhone());
        emailTextView.setText(contact.getEmail());
    }
  });
}
```

Por fim, modifique o método *deleteContact* para fazer o uso da *ViewModel*:

```
// exclui um contato
private void deleteContact() {
  // usa a ViewModel para apagar o contato aberto
  detailViewModel.delete();
  Navigation.findNavController(getView()).popBackStack();
}
```

Tudo pronto. Agora você pode executar o aplicativo e rodar a sua mais nova Agenda de Contatos. Sinta-se livre para adicionar novos temas e estilos ao aplicativo, bem ícones e plano de fundo. Todas as operações são executadas e armazenadas no **SQLite** do aparelho, por tanto os dados não serão perdidos. Você pode adicionar, editar ou remover seus contatos da maneira que desejar.

# 5. Conclusão

Se você seguiu todos os passos ilustrados pela apostila, você finalizou a criação do aplicativo **Agenda de Contatos** para adicionar, ver, editar e excluir informações de contato armazenadas em um banco de dados SQLite.

Você usou uma única atividade para armazenar e gerenciar todos os fragmentos do aplicativo. Usou **Navigation** para exibir fragmentos dinamicamente. Usou a pilha de retrocesso (**BackStack**) para fornecer suporte automático ao botão de voltar do Android.

Você usou a biblioteca **Room** para abstrair o uso de banco de dados **SQLite** no dispositivo. Também criou um objeto **DAO** para manipular as operações SQL de consulta, inserção, atualização e remoção de contatos no banco de dados. Para acessar o banco de dados de forma assíncrona, fora da *thread* da interface gráfica do usuário, você fez o uso da classe **AsyncTask**. Você configurou a vinculação dos dados proveniente do repositório da aplicação com o uso de **ViewModels** e a classe

**LiveData**.

No final, temos um aplicativo robusto fazendo o uso dos melhores componentes de arquitetura disponibilizados no **Android Jetpack**. Seguindo o princípio da responsabilidade única temos uma arquitetura onde os fragmentos estão responsáveis em como apresentar os dados ao usuário, as *ViewModels* se encarregam de armazenar as informações e fazer solicitações ao repositório, o repositório se encarrega de se comunicar com a biblioteca Room e encaminhar solicitação ao objeto **DAO**, e a os componentes da biblioteca **Room** se encarregam de se comunicar com o banco de dados **SQLite** em si. Dessa forma, construímos um aplicativo robusto, testável e eficiente.