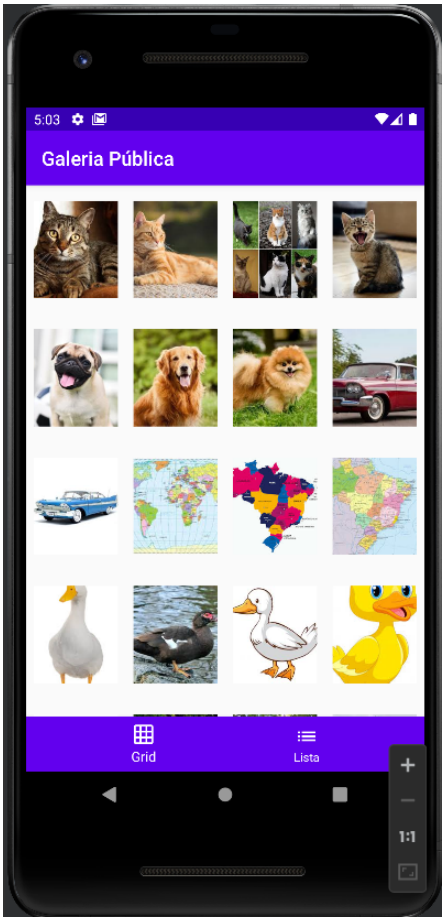
Dispositivos Móveis - Prof. Daniel Ribeiro Trindade

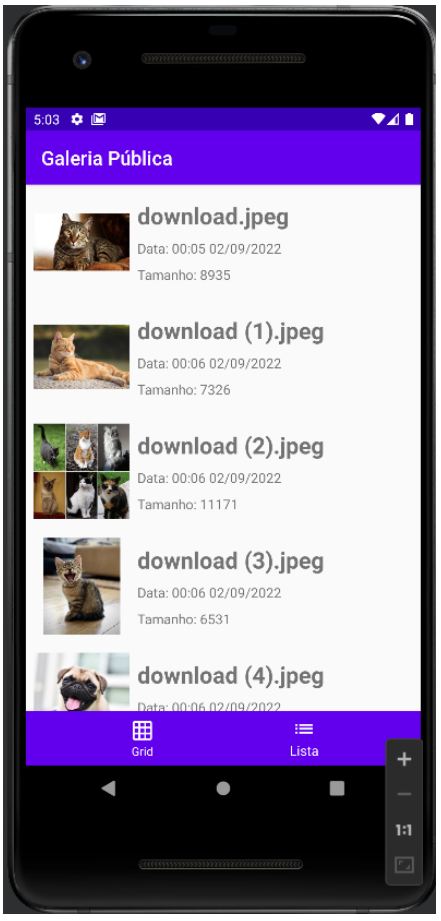
Roteiro para criação da APP Galeria Pública

Neste roteiro iremos criar uma app para Android do tipo Galeria para visualizar as fotos presentes na galeria pública do celular.

A app será composta por somente uma tela/Activity:

- **MainActivity**: a tela principal, que exibe as fotos presentes na galeria pública do celular. O usuário conta com um menu na parte inferior da tela (**BottonViewNavigation**), onde é possível escolher entre duas opções diferentes de visualização:
    a. **Grid** -> neste tipo de visualização será mostrado uma lista em formato de grade, onde cada item é uma miniatura da foto;
    b. **Lista** -> neste tipo de visualização será exibida uma lista no formato padrão. Cada item irá conter uma miniatura da foto, o nome original do arquivo, a data de criação do arquivo e o tamanho em bytes dos arquivos.





Neste projeto iremos aprender como alterar a interface de usuário dinamicamente. Isso significa que, ao selecionar uma das opções do menu, a app não realiza uma transição

entre diferentes Activities. Somente o conteúdo principal da tela, delimitado pela região em branco nas figuras acima, é trocado. Para realizar essa mudança na UI, nós iremos usar um componente do Android chamado **Fragment**.

Um **fragment,** como o próprio nome indica, é um fragmento de UI que pode ser substituído enquanto a app está sendo executada. Assim como uma Activity, um **fragment** possui um arquivo de código java e um arquivo de layout associado. Na nossa app criaremos dois fragments:

1. **GridViewFragment** -> esse fragmento é composto de um RecycleView configurado para exibir as fotos em formato de grade. Ele é setado na Activity toda vez que o usuário escolhe a opção Grid no menu inferior.
2. **ListViewFragment** -> esse fragmento é composto de um RecycleView configurado para exibir as fotos em formato de lista. Ele é setado na Activity toda vez que o usuário escolhe a opção Lista no menu inferior.

Outro conceito que iremos aprender neste projeto é o de **paginação de dados**. Quando estamos exibindo dados em uma lista, é comum que a quantidade total de itens seja grande demais para ser totalmente carregada em memória. Outra situação é quando os dados são provenientes de um servidor web: transferir todos os dados do servidor para a aplicação implicaria em um tempo muito grande para concluir a operação.

A **paginação de dados** tem como objetivo justamente resolver esse problema. Ao invés de carregar os dados todos de uma só vez, a app obtém uma nova página sempre que for necessário. Uma **página** de dados é um subconjunto da lista total. Por exemplo, para uma lista de 300 itens, podemos estabelecer que cada página é composta por 30 desses itens. Logo que a app abre, são exibidos os itens referentes a uma página (30 itens). Uma vez que usuário role a lista, novas páginas vão sendo solicitadas. Essa estratégia garante que a app tenha um desempenho melhor, ao mesmo tempo em que economiza recursos de memória e banda de rede. No Android, a biblioteca Paging 3 é utilizada para fornecer uma forma integrada com o RecycleView o recurso de **paginação de dados**.

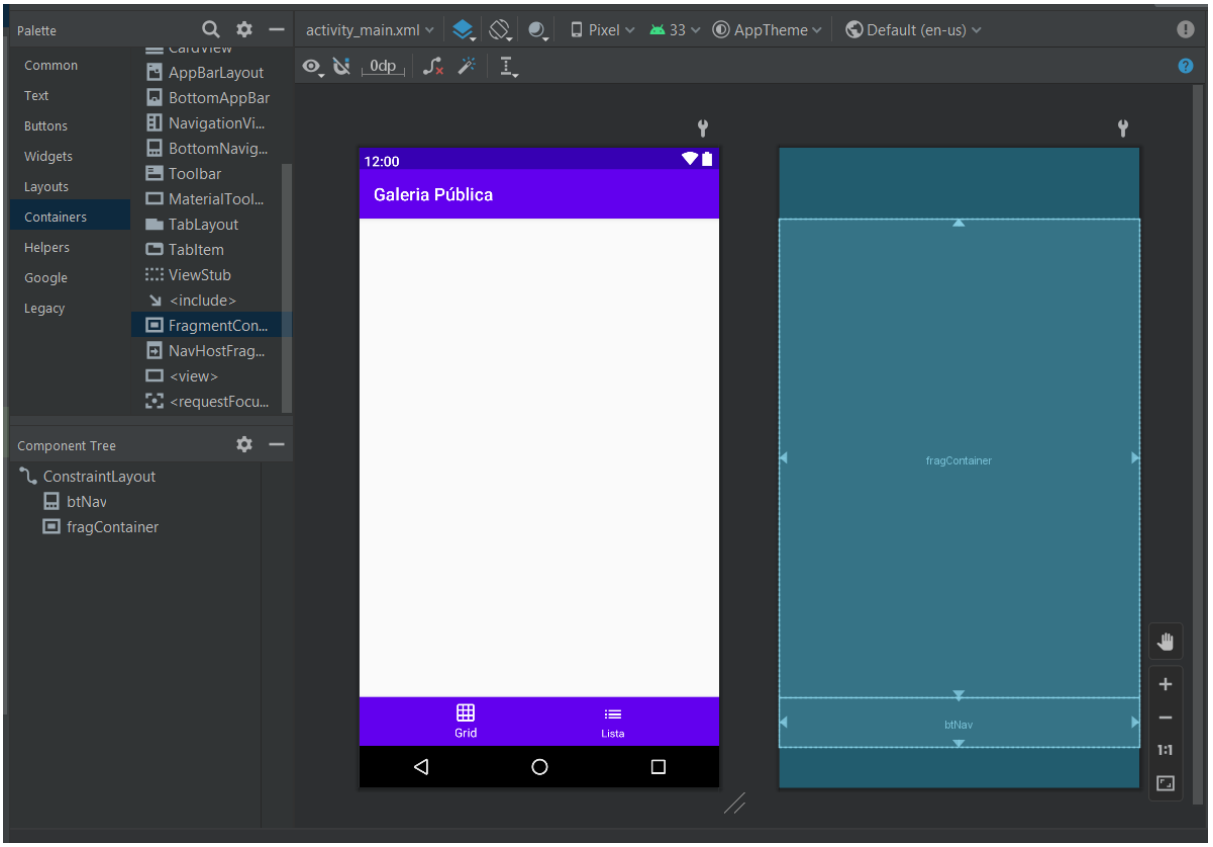
Passo 1 - Criação do Projeto

- Crie um novo projeto no Android Studio
  - O nome da APP deve ser Galeria Pública;
  - O package deve ser sobrenome.nome.galeria. Por exemplo: **trindade.daniel.galeriapublica**;
  - A linguagem deve ser Java;
  - Escolha o template “Empty”;
- Depois de criar o projeto, compartilhe ele no GitHub;
- Dê o commit inicial no projeto e de um push para que ele suba para o GitHub.

Passo 2 - Criação da Interface

A app é composta somente por uma Activity, a MainActivity. Ela é composta por 2 elementos principais de UI.

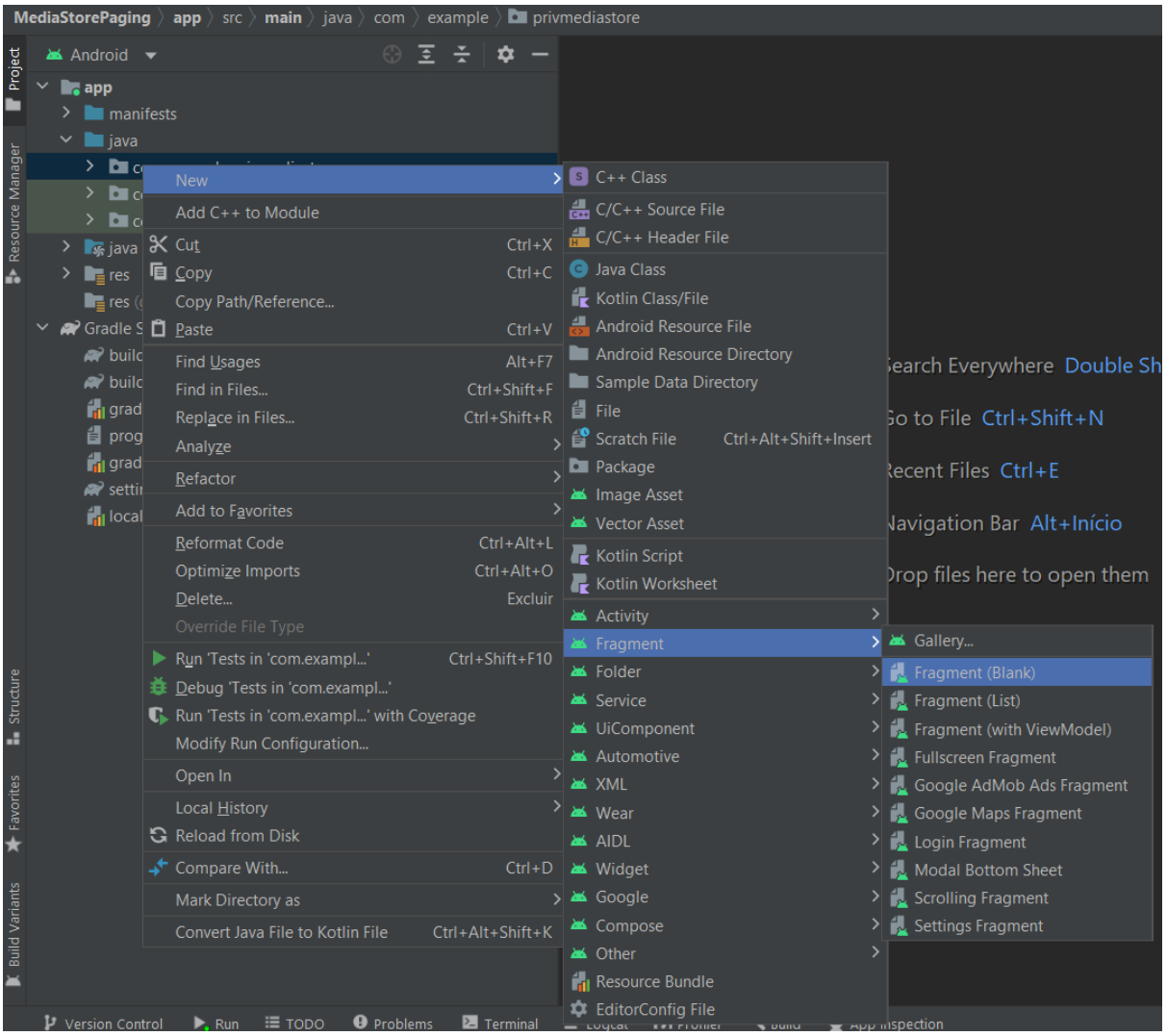
- Modifique o layout de **MainActivity** para ficar como mostrado na figura abaixo. Esse layout possui dois elementos de UI:
    - **fragContainer** (Palette -> Layouts -> FrameLayout) -> esse tipo de elemento estabelece um espaço “em branco”, no qual iremos setar diferentes fragmentos no futuro;
    - **btNav** (Palette -> Containers -> BottonNavigationView) -> elemento de UI para definir um menu que aparece na parte inferior da tela. Cada opção do menu é composta pelo nome da opção e um ícone.



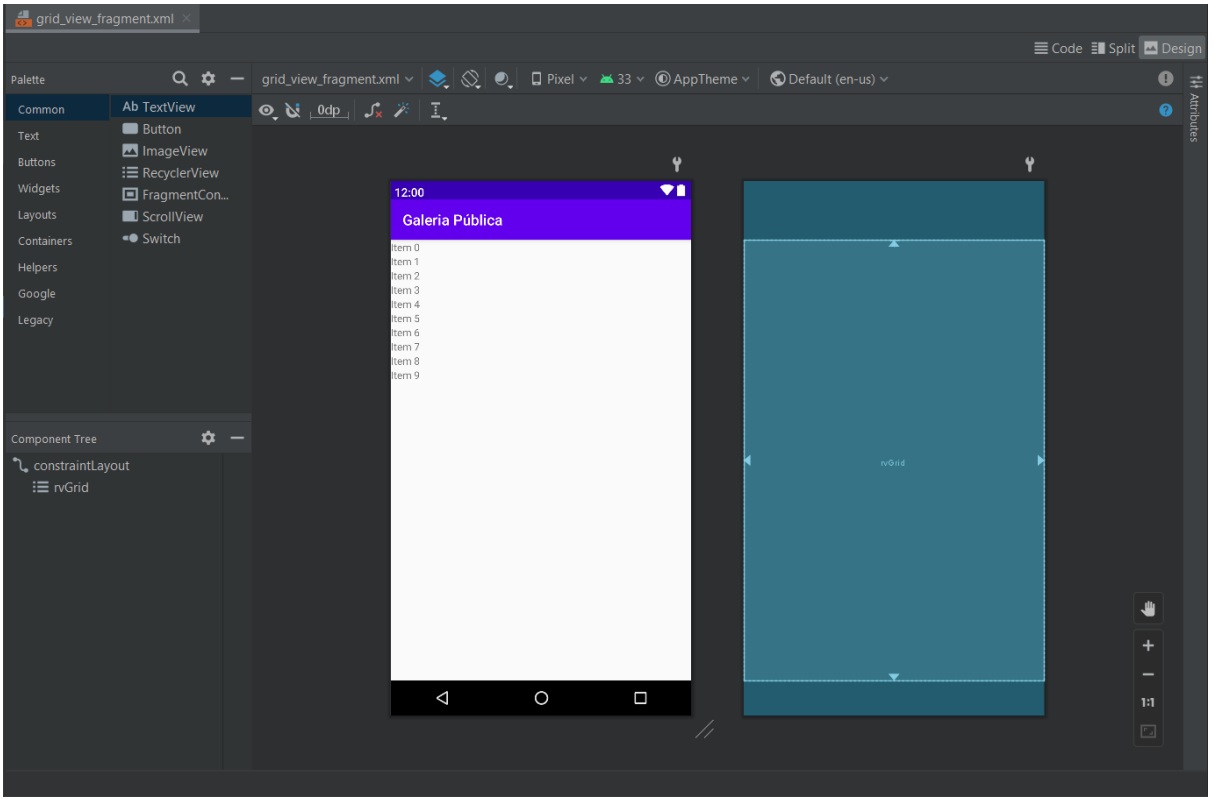
Como já mencionado antes, o elemento fragContainer é apenas um local onde serão setados diferentes tipos de interface, dependendo de qual opção o usuário escolhe no menu interior da tela. Caso o usuário escolha a opção Grid, fragContainer vai exibir uma lista no formato grid. Caso o usuário escolha a opção Lista, fragContainer vai exibir uma lista no formato padrão. Para definir esses diferentes tipos de interface, usaremos um componente do Android chamado Fragment.

Um fragment representa um pedaço de UI que pode ser modificado enquanto a app está sendo executada. Ele é composto de um arquivo de código java e um arquivo de layout associado, assim como ocorre com uma activity. A diferença é que fragments precisam necessariamente estar associados, em um dado momento, a uma activity em específico. Além disso, diferente de uma activity, um fragment não precisa ser declarado no arquivo AndroidManifest.xml. Na nossa app iremos criar dois fragmentos, um para a lista em formato grid e outro para a lista em formato padrão.

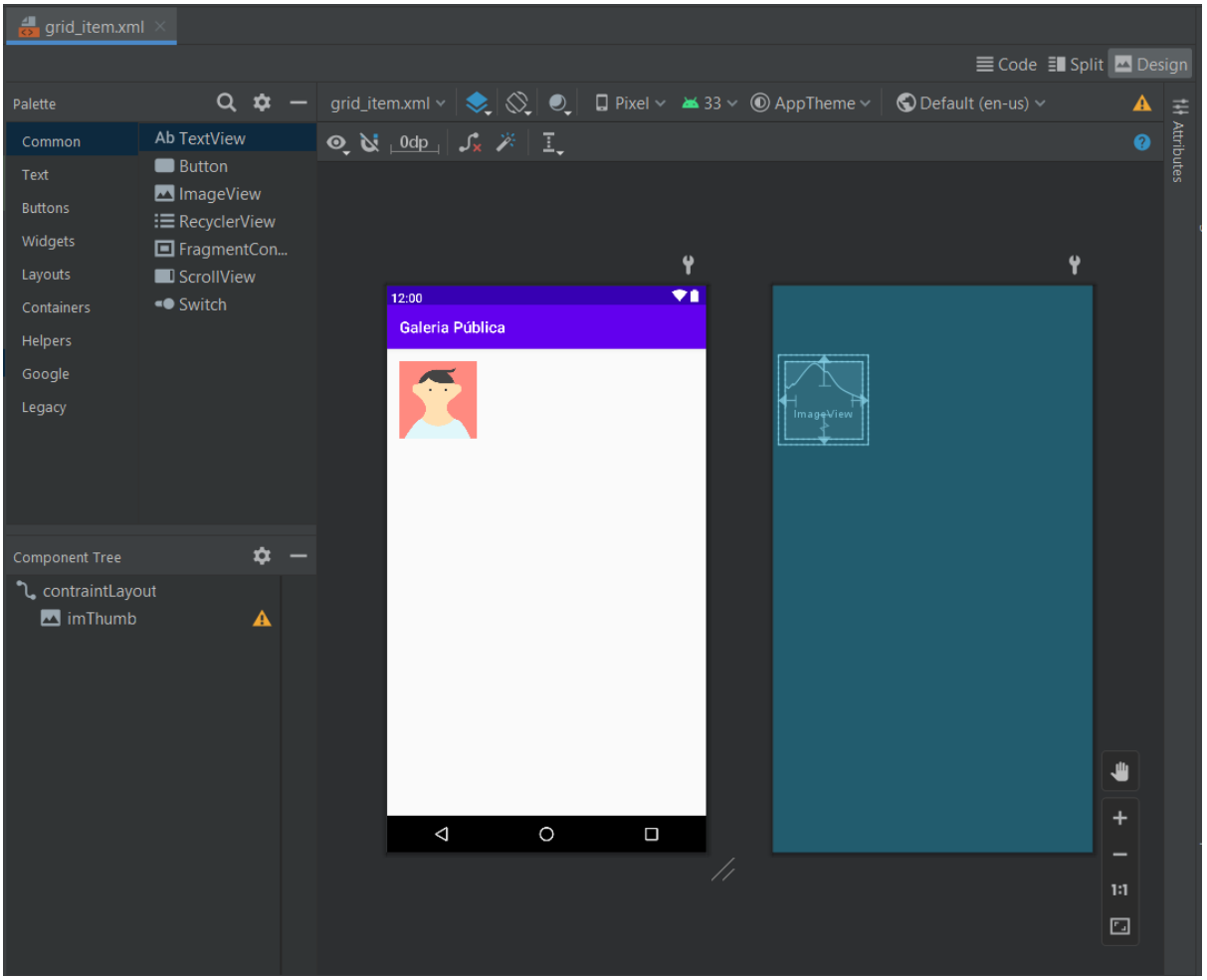
- Como na figura abaixo, crie dois fragments do tipo Blank (vazio) e os nomeie da seguinte forma:
    - **GridViewFragment** -> contém um recycleview para exibir a lista no formato grid;
    - **ListViewFragment** -> contém um recycleview para exibir a lista no formato lista;



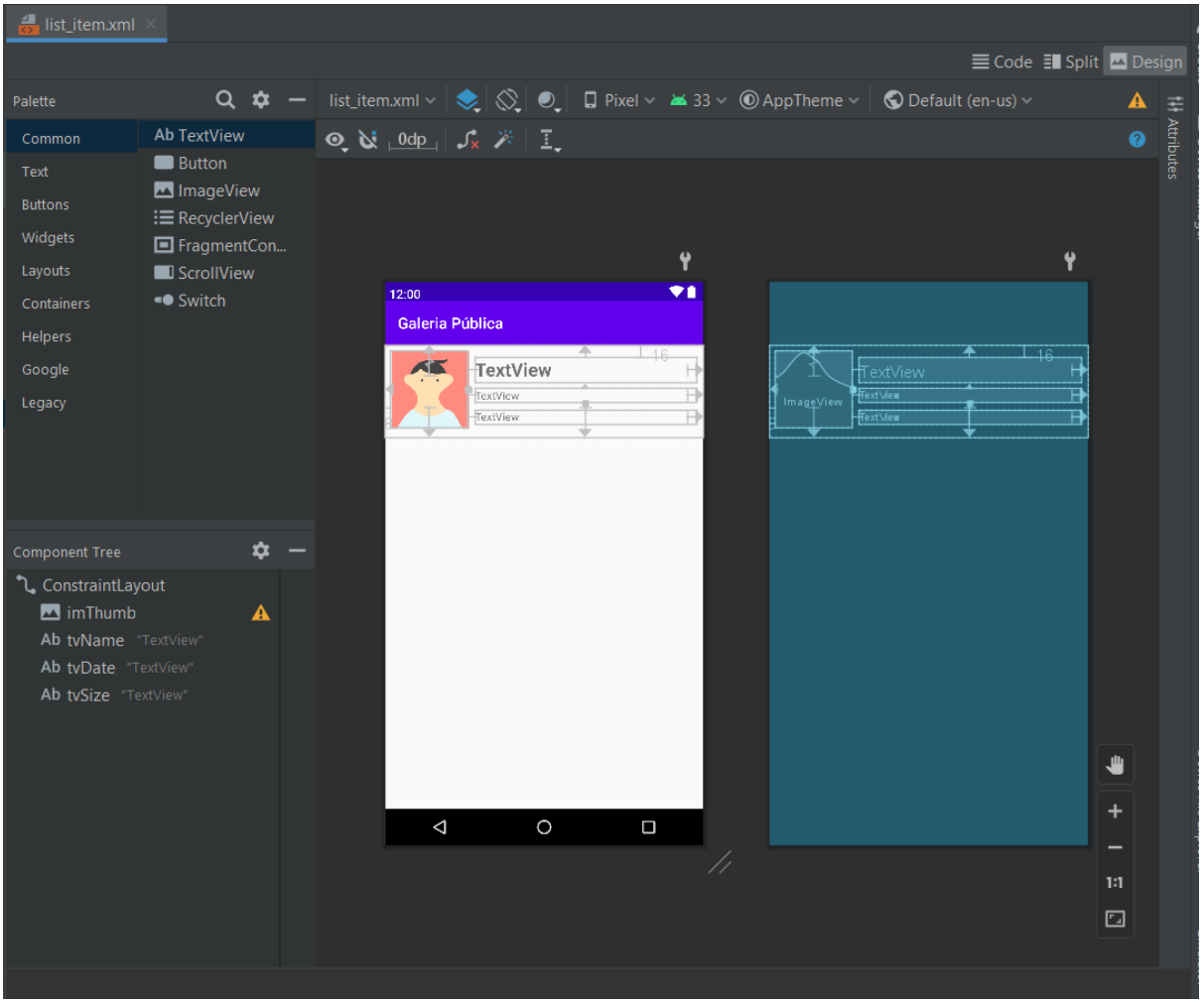
- Para cada arquivo de layout dos fragmentos criados, adicione um elemento do tipo RecycleView, como na figura abaixo.



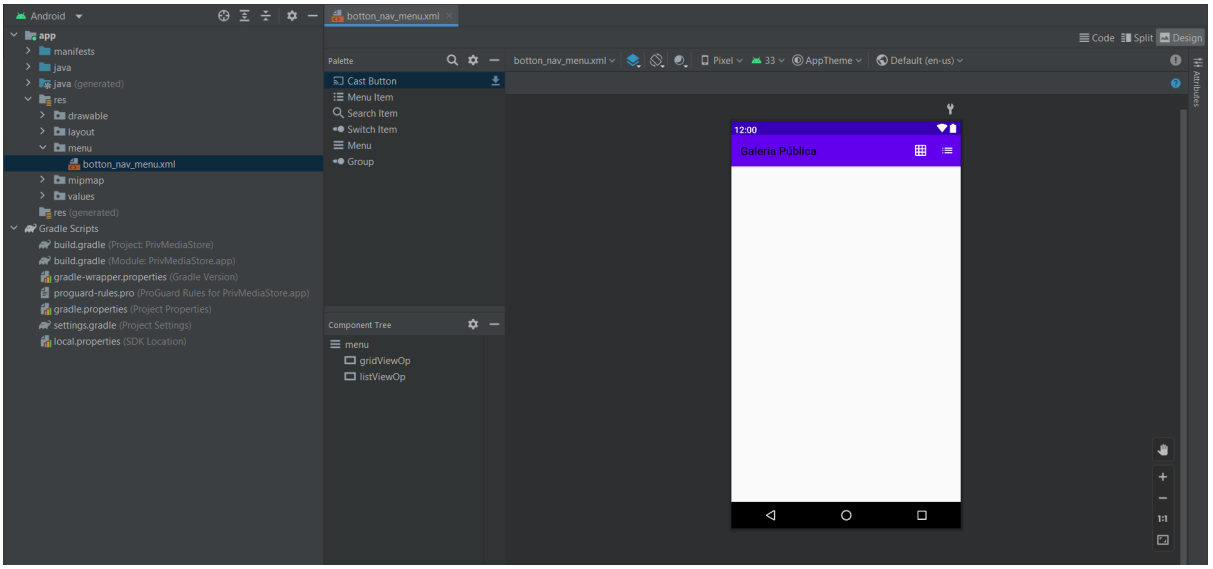
- Crie os layouts referentes a um item da lista, como nas figuras abaixo:
  - **grid_item.xml** -> layout referente a somente um item da lista no formato grd, composto unicamente por um ImageView (**imThumb**);



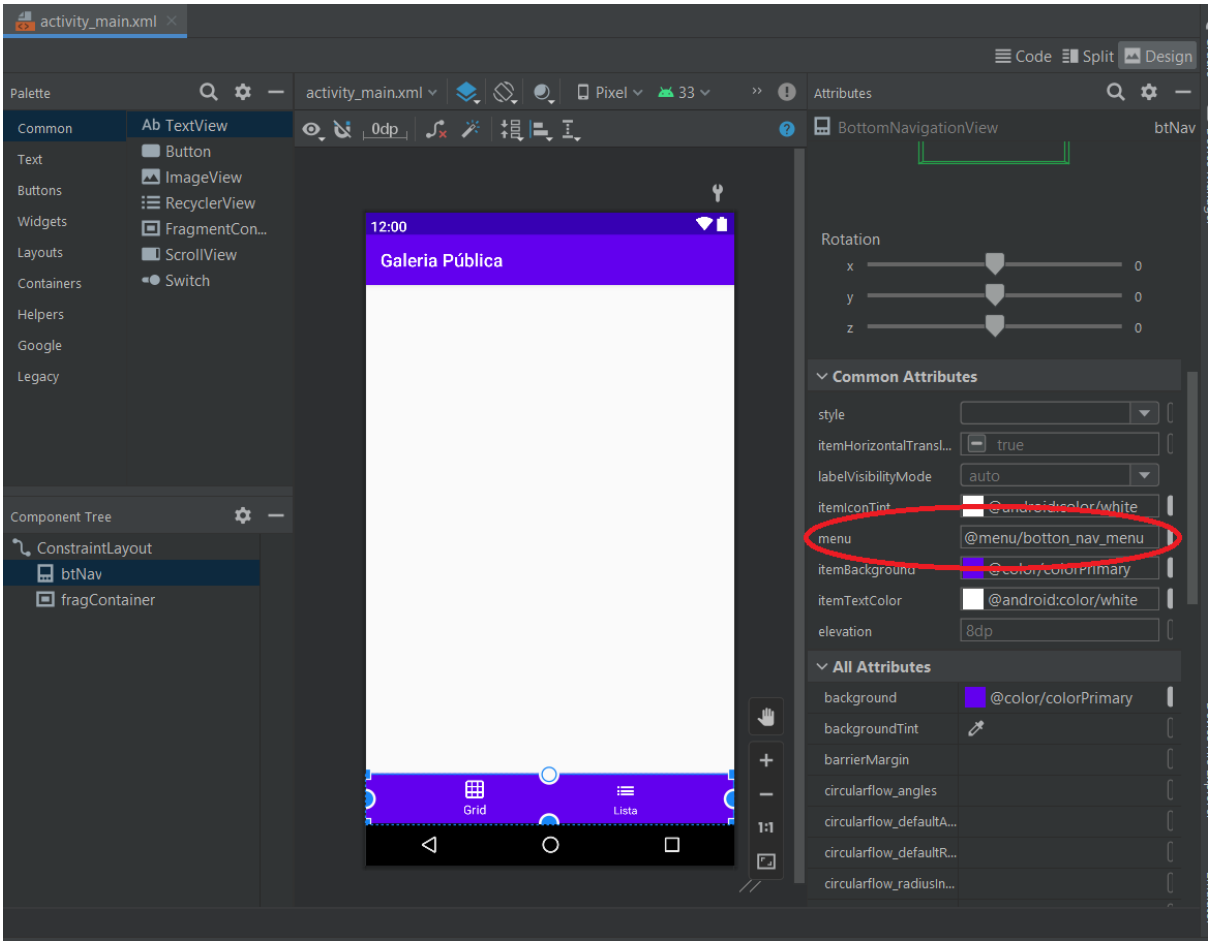
- **list_item.xml** -> layout referente a somente um item da lista no formato padrão, composto por um ImageView (**imThumb**), um textView para exibir o nome do arquivo (**tvName**), um textView para exibir a data de criação da foto (**tvDate**) e um textView para exibir o tamanho da foto (**tvSize**).



- Crie um novo diretório para guardar os arquivos de menu do projeto. Dentro desse diretório adicione **botton_nav_menu.xml**, o arquivo com as definições do menu que aparece no canto inferior da tela. Adicione dois itens de menu:
    - **gridViewOp** -> para definir a opção de menu referente a Grid (o nome da opção é "**Grid**"). Escolha um ícone de acordo;
    - **listViewOp** -> para definir a opção de menu referente a lista padrão (o nome da opção é "**Lista**"). Escolha um ícone de acordo;



- No layout referente a MainActivity, selecione o elemento btNav. Na lista de atributos, ache a opção menu. Selecione o menu que você criou no anteriormente. Esse atributo indica para o BottonNavigationView qual menu ele deve usar para disponibilizar opções para o usuário.



Passo 3 - Modificando a Aparência da APP: Cores, Background

- Modifique a aparência dos elementos de interface de acordo com suas preferências

- Escolhe uma nova paleta de cores;
- Defina o formato/aparência de cada itém da lista através do uso de drawables;

Passo 4 - Criação do ModelView

- Crie uma nova classe java no projeto para funcionar como ViewModel de MainActivity

```
1. public class MainViewModel extends AndroidViewModel {
2.
3.     int navigationOpSelected = R.id.gridViewOp;
4.
5.     public MainViewModel(@NonNull Application application) {
6.         super(application);
7.     }
8.
9.     public int getNavigationOpSelected() {
10.        return navigationOpSelected;
11.    }
12.
13.    public void setNavigationOpSelected(int navigationOpSelected) {
14.        this.navigationOpSelected = navigationOpSelected;
15.    }
16.}
```

Olhando o código acima, temos que **MainViewModel** herda de **AndroidViewModel** (linha 1) e não de ViewModel. A classe **AndroidViewModel** é uma especialização de **ViewModel** que possui como parâmetro de entrada em seu construtor uma instância da aplicação (linha 5). Isso nos permite acessar dentro de **MainViewModel** o contexto da aplicação, o que será importante quando estivermos trabalhando com a parte de paginação de dados.

Por enquanto **MainViewModel** guarda somente um dado: **navigationOpSelected** (linha 3). Esse é do tipo inteiro e serve para guardar a opção escolhida pelo usuário no menu **btNav**. Assim, caso a app saia fora de foco e volte novamente, podemos exibir o último tipo de lista que foi selecionado pelo usuário. As linhas 9 e 13 mostram os métodos para pegar e setar esse valor.

Passo 5 - Configuração do BottonNavigationView e Como Escutar os Eventos de Menu

- Configure as ações que serão executadas quando o usuário selecionar uma das opções do BottonavigationView.

```
1. public class MainActivity extends AppCompatActivity {
2.
3. ...
4.
5.     BottomNavigationView bottomNavigationView;
```

```
6.
7.      @Override
8.      protected void onCreate(Bundle savedInstanceState) {
9.          super.onCreate(savedInstanceState);
10.         setContentView(R.layout.activity_main);
11.
12.         final MainViewModel vm = new
    ViewModelProvider(this).get(MainViewModel.class);
13.
14.         bottomNavigationView = findViewById(R.id.btNav);
15.         bottomNavigationView.setOnItemSelectedListener(new
    NavigationBarView.OnItemSelectedListener() {
16.             @Override
17.             public boolean onNavigationItemSelected(@NonNull MenuItem
    item) {
18.                 vm.setNavigationOpSelected(item.getItemId());
19.                 switch (item.getItemId()) {
20.                     case R.id.gridViewOp:
21.                         GridViewFragment gridViewFragment =
    GridViewFragment.newInstance();
22.                         setFragment(gridViewFragment);
23.                         break;
24.                     case R.id.listViewOp:
25.                         ListViewFragment listViewFragment =
    ListViewFragment.newInstance();
26.                         setFragment(listViewFragment);
27.                         break;
28.                 }
29.                 return true;
30.             }
31.         });
32.     }
33.
34....
```

No código acima, começamos definindo o **bottonViewNavigation** como um atributo da classe MainActivity (linha 5). Isso é necessário pois iremos utilizar a referência em outros métodos da classe.

Na linha 12 obtemos uma referência para **MainViewModel**, enquanto na linha 13 obtemos a referência para o **BottonNavigationView**.

Na linha 15 setamos em **bottonNavigationView** o “escutador” de eventos de seleção do menu. Assim, toda vez que o usuário selecionar uma das opções, o método **onNavigationItemSelected** será chamado, indicando qual opção foi escolhida (linha 17).

Na linha 18 nós guardamos dentro de **MainViewModel** a opção que foi escolhida pelo usuário.

Nas linhas de 20 a 29 nós definimos as ações que serão realizadas para cada opção. Caso o usuário selecione **gridViewOp** (Grid) (linha 20), será criado um fragmento do tipo **GridViewFragment** (linha 21) e o mesmo será setado em **MainActivity** (linha 22). Caso o usuário selecione **listViewOp** (Grid) (linha 24), será criado um fragmento do tipo **ListViewFragment** (linha 25) e o mesmo será setado em **MainActivity** (linha 26).

Passo 6 - Como Setar um Fragmento

- Implemente o método que seta um fragment dentro do fragContainer de MainActivity.

```
1. public class MainActivity extends AppCompatActivity {
2. ...
3.
4.     void setFragment(Fragment fragment) {
5.         FragmentTransaction fragmentTransaction =
   getSupportFragmentManager().beginTransaction();
6.         fragmentTransaction.replace(R.id.fragContainer, fragment);
7.         fragmentTransaction.addToBackStack(null);
8.         fragmentTransaction.commit();
9.     }
10.
11....
```

No código acima, o método **setFragment** recebe como parâmetro um **fragment** (linha 4). Esse **fragment** será setado no espaço definido pelo elemento de UI **fragContainer** (linha 6). Para isso, primeiro é iniciada uma transação do gerenciador de fragmentos (linha 5). Depois de substituido o fragmento em **fragContainer** (linha 6), indicamos que esse fragmento agora faz parte da pilha de tela do botão voltar do Android (linha 7). Ao final, é realizado o commit da transação (linha 8).

Passo 7 - Permissões

- No arquivo AndroidManifest.xml adicione a permissão abaixo. Ela garante acesso de leitura ao espaço público de amazenamento do celular.

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

- Implemente o método **onResume** e dentro dele peça pela permissão de acesso de leitura ao armazenamento público.

```
public class MainActivity extends AppCompatActivity {

...

    @Override
    protected void onResume() {
        super.onResume();
        List<String> permissions = new ArrayList<>();
        permissions.add(Manifest.permission.READ_EXTERNAL_STORAGE);
        checkForPermissions(permissions);
```

```
    }

...
```

Só é possível acessar as imagens do armazenamento público do celular depois que o usuário explicitamente fornece a permissão. Assim, nós devemos disparar o código que lê as imagens do armazenamento público somente quando temos certeza que a app possui todas as permissões necessárias. Para isso, temos que modificar as funções responsáveis por verificar e pedir as permissões do usuário.

- Modifique os métodos de permissões como indicado abaixo:

```
1. private void checkForPermissions(List<String> permissions) {
2.     List<String> permissionsNotGranted = new ArrayList<>();
3.
4. ...
5.
6.     if(permissionsNotGranted.size() > 0) {
7.
8. ...
9.
10.    }
11.    else {
12.        MainViewModel vm = new
   ViewModelProvider(this).get(MainViewModel.class);
13.        int navigationOpSelected = vm.getNavigationOpSelected();
14.        bottomNavigationView.setSelectedItemId(navigationOpSelected);
15.    }
16.}
17.
18....
19.
20.@Override
21.public void onRequestPermissionsResult(int requestCode, @NonNull
   String[] permissions, @NonNull int[] grantResults) {
22.    super.onRequestPermissionsResult(requestCode, permissions,
23.
24....
25.
26.    if(permissionsRejected.size() > 0) {
27.
28....
29.
30.    }
31.    else {
32.        MainViewModel vm = new
   ViewModelProvider(this).get(MainViewModel.class);
33.        int navigationOpSelected = vm.getNavigationOpSelected();
34.        bottomNavigationView.setSelectedItemId(navigationOpSelected);
35.    }
36.}
```

O código acima mostra apenas as modificações que devem ser realizadas nos métodos de solicitação de permissões. No método **checkForPermissions** (linha 1), caso a app já possua todas as permissões que precisa (linha 11), acessamos a **MainViewModel** e obtemos qual a opção que foi escolhida pelo usuário (linhas 12 e 13). Em seguida, setamos essa opção em **bottonNavgationView** (linha 14). Isso faz com que o fragmento seja setado em **MainActivity**. A criação do fragmento irá provocar a leitura das fotos no armazenamento público. Observe que isso só acontece depois que o usuário já deu todas as permissões necessárias. Da mesma forma, no método **onRequestPermissionResult** verificamos se todas as permissões foram fornecidas pelo usuário (linha 31). em caso positivo, é realizado o mesmo procedimento (linhas 32 a 34).

Passo 8 - Paginação de Dados Usando a Biblioteca Paging 3

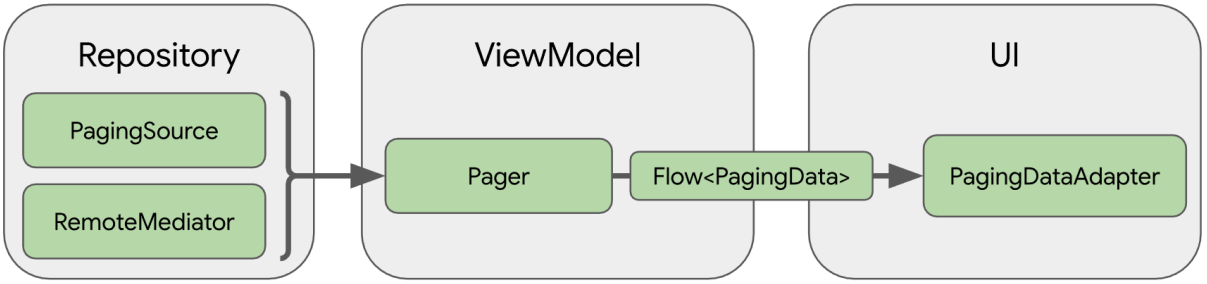
Como já explicado anteriormente, **paginação de dados** consiste em realizar o carregamento de dados em pequenos blocos sempre que haja necessidade. Essa técnica deve ser aplicada preferencialmente sempre que a quantidade de dados a ser exibida/carregada é muito elevada. Com ela nossa app obtém os seguintes benefícios:

- **Melhor uso dos recursos de memória do celular** -> ao exibir uma lista, os dados da lista estarão carregados em memória. Para conjuntos muito grandes de itens em lista, isso pode fazer com que uma quantidade excessiva de memória seja usada. Ao usar paginação de dados, garantimos que somente os dados que estão sendo visualizados pelo usuários estarão presentes na memória, economizando assim recursos do dispositivo.
- **Menor tempo de carregamento dos dados** -> para listas muito grandes, o carregamento dos dados pode levar um tempo considerável para finalizar. Ao carregar somente blocos/páginas que representam uma fração do total, a app é capaz de realizar essa operação mais rápido, ao mesmo tempo em que o usuário tem feedback mais rápido na interface.
- **Economia de banda de transmissão de internet** -> o uso mais comum da técnica de paginação de dados é quando esses são transmitidos de um servidor web para a aplicação. Assim, sempre que a app precisa de novos dados, ela realiza uma solicitação ao servidor web que, por sua vez, entrega apenas uma quantidade suficiente para aquele momento. Como a quantidade de dados transmitida é bem menor que o todo, então a transferência é feita de forma muito mais rápida ao mesmo tempo em que há economia da banda de transmissão.

No nosso projeto, por exemplo, estamos lendo os arquivos de fotos que estão presentes na galeria pública do celular. É comum que um usuário tenha centenas de fotos e, carregar isso de uma única vez, levaria muito tempo e consumiria muito memória. A paginação de dados permite carregar pequenas quantidades de fotos (páginas) sempre que o usuário rolar a página para baixo. Neste projeto configuramos o tamanho de cada página como 10. Isso quer dizer que, toda vez que o usuário rolar a tela e chegar em seu final, serão carregadas mais 10 fotos da galeria pública do celular.

Para implementar a técnica de paginação de dados, o Android fornece uma biblioteca chamada Paging 3. Ela fornece integração com o RecycleView de forma que o desenvolvedor não precisa se preocupar com os momentos em que novas páginas de

dados devem ser solicitadas. A figura abaixo mostra os componentes que devem ser implementados para usar a Paging 3.



- **Repository** -> o repositório é a fonte dos dados. É nessa classe que iremos colocar a lógica de acesso aos dados e montar uma lista com a quantidade de itens de uma página. Os dados geralmente são obtidos de um banco de dados armazenado no próprio dispositivo ou, mais comum, obtidos via rede através de solicitações a um servidor web.
- **PagingSource** -> e a classe onde definimos o método para carregar uma página. O **PagingSource** calcula qual página deve ser pedida e faz a solicitação ao **repository** para obter os dados. Uma vez obtidos, esses dados são montados em um **PagingData** e submetidos ao **PagingDataAdapter**.
- **Pager** -> o pager é criado dentro do **ViewModel**. É nele que configuramos qual o tamanho de cada página (quantos itens cada página vai ter), e qual **PagingSource** será usando para obter as páginas.
- **PagingData** -> estrututa de dados que guarda uma página de dados.
- **PagingDataAdapter** -> uma especialização da classe RecycleView.Adapter para exibir listas de dados paginados.

- Adicione os pacotes da biblioteca Paging 3 no projeto. Abra o arquivo gradle.build e adicione as linhas abaixo no bloco dependencies. Em seguida, clique no botão “sync” que aparece na barra superior ao arquivo para baixar a biblioteca Paging 3.

```
1. ...
2.
3. dependencies {
4.
5.     implementation fileTree(dir: "libs", include: ["*.jar"])
6.     def paging_version = "3.1.1"
7.     // optional - Guava ListenableFuture support
8.     implementation "androidx.paging:paging-guava:$paging_version"
9.     implementation "androidx.paging:paging-runtime:$paging_version"
10.
11....
```

Passo 8.1 - Definição do Modelo de Dados

- Crie uma nova classe para representar/guardar os dados de uma foto.

```
1. public class ImageData {
```

```
2.     public Uri uri;
3.     public Bitmap thumb;
4.     public String fileName;
5.     public Date date;
6.     public int size;
7.
8.     public ImageData(Uri uri, Bitmap thumb, String fileName, Date
   date, int size) {
9.         this.uri = uri;
10.        this.thumb = thumb;
11.        this.fileName = fileName;
12.        this.date = date;
13.        this.size = size;
14.    }
15.}
```

No código acima, iremos guardar os seguintes dados referente a um arquivo de foto:

- uri (linha 2) -> endereço uri do arquivo de foto;
- thumb (linha 3) -> imagem em minitura da foto;
- filename (linha 4) -> nome do arquivo de foto;
- date (linha 5) -> data em que a foto foi criada;
- size (linha 6) -> tamanho em bytes do arquivo de foto.

Passo 8.2 - Configuração do Repositório de Dados

- Crie uma classe **GalleryRepository**, como abaixo:

```
1. public class GalleryRepository {
2.
3.     Context context;
4.
5.     public GalleryRepository(Context context) {
6.         this.context = context;
7.     }
8.
9.     public List<ImageData> loadImageData(Integer limit, Integer
   offSet) throws FileNotFoundException {
10.
11.        List<ImageData> imageDataList = new ArrayList<>();
12.        int w = (int)
   context.getResources().getDimension(R.dimen.im_width);
13.        int h = (int)
   context.getResources().getDimension(R.dimen.im_height);
14.
15.        String[] projection = new String[]
   {MediaStore.Images.Media._ID,
16.                MediaStore.Images.Media.DISPLAY_NAME,
17.                MediaStore.Images.Media.DATE_ADDED,
18.                MediaStore.Images.Media.SIZE};
19.        String selection = null;
20.        String selectionArgs[] = null;
21.        String sort = MediaStore.Images.Media.DATE_ADDED;
22.
```

```
        Cursor cursor = null;
        if(Build.VERSION.SDK_INT > Build.VERSION_CODES.Q) {
            Bundle queryArgs = new Bundle();
queryArgs.putString(ContentResolver.QUERY_ARG_SQL_SELECTION,
selection);
            queryArgs.putStringArray(
                    ContentResolver.QUERY_ARG_SQL_SELECTION_ARGS,
                    selectionArgs
            );
            // sort
            queryArgs.putString(
                    ContentResolver.QUERY_ARG_SORT_COLUMNS,
                    sort
            );
            queryArgs.putInt(
                    ContentResolver.QUERY_ARG_SORT_DIRECTION,
                    ContentResolver.QUERY_SORT_DIRECTION_ASCENDING
            );
            // limit, offset
            queryArgs.putInt(ContentResolver.QUERY_ARG_LIMIT, limit);
            queryArgs.putInt(ContentResolver.QUERY_ARG_OFFSET,
offSet);

            cursor =
context.getContentResolver().query(MediaStore.Images.Media.EXTERNAL_C
ONTENT_URI,
                    projection,
                    queryArgs,
                    null
            );

        }
        else {
            cursor =
context.getContentResolver().query(MediaStore.Images.Media.EXTERNAL_C
ONTENT_URI,
                    projection,
                    selection,
                    selectionArgs,
                    sort + " ASC + LIMIT " + String.valueOf(limit) + "
OFFSET " + String.valueOf(offSet)
            );
        }

        int idColumn =
cursor.getColumnIndexOrThrow(MediaStore.Images.Media._ID);
        int nameColumn =
cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DISPLAY_NAME);
        int dateAddedColumn =
cursor.getColumnIndexOrThrow(MediaStore.Images.Media.DATE_ADDED);
        int sizeColumn =
cursor.getColumnIndexOrThrow(MediaStore.Images.Media.SIZE);
```

```
64.
65.         while (cursor.moveToNext()) {
66.             // Get values of columns for a given image.
67.             long id = cursor.getLong(idColumn);
68.             Uri contentUri = ContentUris.withAppendedId(
69.                     MediaStore.Images.Media.EXTERNAL_CONTENT_URI, id);
70.             String name = cursor.getString(nameColumn);
71.             int dateAdded = cursor.getInt(dateAddedColumn);
72.             int size = cursor.getInt(sizeColumn);
73.             Bitmap thumb = Util.getBitmap(context, contentUri, w, h);
74.
75.             // Stores column values and the contentUri in a local
    object
76.             // that represents the media file.
77.             imageDataList.add(new ImageData(contentUri, thumb, name,
    new Date(dateAdded*1000L), size));
78.         }
79.         return imageDataList;
80.     }
81.}
```

No código acima, o construtor de **GalleryRepository** recebe um objeto do tipo **Context** (linha 5). É através do context da app que seremos capazes de obter acesso ao conteúdo da galeria de fotos do celular.

Na linha 9 definimos o método **loadImageData**. Ele recebe como parâmetros dois números inteiros:

- **limit** -> o número de elementos que devem ser carregados;
- **offset** -> o índice a partir do qual os elementos devem ser carregados.

Por exemplo, se cada página deve ter 10 fotos, então a primeira vez que **loadImageData** é chamado, limit=10 e offSet=0, indicando assim que serão carregados as 10 primeiras fotos da galeria (página 1 de dados). Na segunda chamada, limit=10 e offSet=10 (página 2 de dados). Na terceira chamada, limit=10 e offSet=20…

O método **loadImageData** retorna uma lista de objetos **ImageData**, a qual contém somente a quantidade de itens referentes a uma página.

Na linha 11 criamos a lista de **ImageData**.

Nas linhas 12 e 13 pegamos as dimensões que cada miniatura de foto deve ter. As dimensões devem estar definidas no arquivo “**res->values->dimens.xml**”.

No Android os dados são armazenados no espaço público do celular como se fosse um banco de dados. Cada arquivo possui um **id** e está armazenado em uma tabela com nome específico. Cada tabela possui colunas que dependem do tipo de dado que ela armazena.

No nosso caso iremos acessar a tabela que guarda as imagens (**MediaStore.Images.Media**), e para cada fotos iremos obter as seguintes colunas (linhas 15 a 18):

- **_ID** -> o id do arquivo de fotos, usado para construir o endereço uri;
- **DISPLAY_NAME** -> o nome do arquivo de foto;
- **DATE_ADDED** -> a data em que a foto foi criada;
- **SIZE** -> tamanho do arquivo em bytes;

A linha 19 podemos definir qual subconjunto dos dados queremos obter (**selection**). No nosso caso, queremos todas as fotos, então setamos esse parâmetro como nulo.

A linha 20 são definidos os argumentos para o selection da linha 19. Como não definimos um **selection**, então também setamos **selectionArgs** como nulo.

Na linha 21 definimos qual coluna da tabela será usada para ordenar os resultados da pesquisa. Nós setamos **DATE_ADDED**, indicando que os dados serão ordenados de acordo com a data de criação do arquivo.

Entre as linhas 23 e 58 é executada a query ao banco de dados de imagens do celular. A partir da versão 11 do Android, a forma de se realizar essa query mudou, então nós temos que verificar qual versão de Android o dispositivo tem instalado para realizar corretamente a consulta (linha 23).

Para versões posteriores ao Android 11 (linha 24 a 50), os parâmetros da consulta devem ser passados via Bundle (linha 25). Um **Bundle** é uma estrutura que guarda tuplas do tipo chave=valor. Nas linhas 26 e 27 são definidos os parâmetros de **selection** e **selectionArgs**. Linhas 32 a 35 definem quais colunas devem ser usadas para a ordenação do resultado da consulta. Linhas 36 a 39 indicam qual será a direção da ordenação (ascendente ou descendente). Linhas 41 e 42 indicam os parâmetros de **LIMIT** e **OFFSET** respectivamente.

Uma vez definidos os parâmetros da consulta, ela é realizada através do **ContentResolver**, o qual é obtido do contexto da aplicação (linha 44). O resultado é retornado em um objeto do tipo Cursor.

Nas linhas 60 a 80 são obtidos os dados para as fotos. Na linha 67 é obtido o ID do arquivo de foto. Esse ID é usado para construir o endereço URI da fotos (linha 68). Nas linhas 70, 71 e 72 são obtidos o nome, data e tamanho do arquivo de foto. Na linha 73 é gerado uma versão em miniatura da foto (thumb). Na linha 77 é criado um objeto do tipo ImageData e o mesmo é adicionado na lista de ImageData.

Uma vez construída a lista de ImageData, ela é retornada pela função (linha 79). Essa lista contém apenas a quantidade de itens referentes a uma página de dados.

Passo 8.3 - Configuração do PageSource

- Crie uma nova classe GalleryPagingSource, como abaixo:

```
1. public class GalleryPagingSource extends
   ListenableFuturePagingSource<Integer, ImageData> {
2.
```

```
3.      GalleryRepository galleryRepository;
4.
5.      Integer initialLoadSize = 0;
6.
7.      public GalleryPagingSource(GalleryRepository galleryRepository) {
8.          this.galleryRepository = galleryRepository;
9.      }
10.
11.     @Nullable
12.     @Override
13.     public Integer getRefreshKey(@NonNull PagingState<Integer,
    ImageData> pagingState) {
14.         return null;
15.     }
16.
17.     @NonNull
18.     @Override
19.     public ListenableFuture<LoadResult<Integer, ImageData>>
    loadFuture(@NonNull LoadParams<Integer> loadParams) {
20.         Integer nextPageNumber = loadParams.getKey();
21.         if (nextPageNumber == null) {
22.             nextPageNumber = 1;
23.             initialLoadSize = loadParams.getLoadSize();
24.         }
25.
26.         Integer offSet = 0;
27.         if(nextPageNumber == 2) {
28.             offSet = initialLoadSize;
29.         }
30.         else {
31.             offSet = ((nextPageNumber - 1) * loadParams.getLoadSize())
    + (initialLoadSize - loadParams.getLoadSize());
32.         }
33.
34.         ListeningExecutorService service =
    MoreExecutors.listeningDecorator(Executors.newSingleThreadExecutor())
    ;
35.         Integer finalOffSet = offSet;
36.         Integer finalNextPageNumber = nextPageNumber;
37.         ListenableFuture<LoadResult<Integer, ImageData>> lf =
    service.submit(new Callable<LoadResult<Integer, ImageData>>() {
38.             @Override
39.             public LoadResult<Integer, ImageData> call() {
40.                 List<ImageData> imageDataList = null;
41.                 try {
42.                     imageDataList =
    galleryRepository.loadImageData(loadParams.getLoadSize(),
    finalOffSet);
43.                     Integer nextKey = null;
44.                     if(imageDataList.size() >=
    loadParams.getLoadSize()) {
45.                         nextKey = finalNextPageNumber + 1;
46.                     }
```

```
47.                     return new LoadResult.Page<Integer,
   ImageData>(imageDataList,
48.                             null,
49.                             nextKey);
50.                 } catch (FileNotFoundException e) {
51.                     return new LoadResult.Error<>(e);
52.                 }
53.
54.             }
55.         });
56.
57.         return lf;
58.     }
59.}
```

O código acima mostra a implementação da classe **GalleryPagingSource**. O objetivo dessa classe é acessar dados usando **GalleryRepository** e, com esses dados, montar uma página de dados que será entregue no futuro para o **Adapter** do **RecycleView**.

A classe **GalleyPagingSource** herda de **ListenableFuturePagingSource** (linha 1). Essa última é uma classe que pertence à biblioteca **Paging 3**. Ao definir o **PagingSource**, identificamos qual é o tipo usado para identificar uma página/chave e o tipo de dado que uma página irá conter. No nosso caso, um **Integer** identifica o número da página e **ImageData** o tipo de dado.

O construtor de **GalleryPagingSource** recebe como parâmetro uma instância de **GalleryRepository** (linha 7), que será usada para consultar os dados e montar as páginas de dados.

ListenableFuturePagingSource é uma especialização da classe **PagingSource**. Ao herdar de **ListenableFuturePagingSource**, somos obrigados a implementar 2 métodos:

- **getRefreshKey** (linha 13) -> não é importante para o nosso caso, então deixamos a implementação padrão que retorna nulo;
- **loadFuture** (linha 19) -> esse método é responsável por carregar uma página do **GalleryRepository** e retorná-lo encapsulado em um objeto **ListenableFuture**. Ele será explicado em detalhes abaixo.

O método **loadFuture** recebe como parâmetro um objeto do tipo **LoadParams** (linha 19). Esse objeto é preenchido automaticamente pela biblioteca **Paging 3**. Ele contém dois atributos que serão utilizados para calcular os parâmetros de **LIMIT** e **OFFSET** que temos que fornecer para o **GalleryRepository**:

- **key** -> obtido através do método **getKey** (linha 20), esse atributo corresponde à página de dados que deve ser obtida neste momento. Na primeira vez em que é chamado, ou seja, quando estamos tentando obter a página 1, o método retorna nulo (linha 21);
- **loadSize** -> obtido através do método **getLoadSize** (linha 23), esse atributo guarda o número de itens que devem ser carregados para a página corrente. Geralmente esse atributo possui um valor fixo para todas as páginas, que é configurado na hora

em que iniciamos a biblioteca de **Paging**. Entretanto, por padrão, a biblioteca **Paging 3** estabelece que, para a página 1, **loadSize** é 3 vezes o valor que foi configurado pelo desenvolvedor.

Os atributos **key** e **loadSize** são utilizados para calcular o parâmetros de **LIMIT** e **OFFSET** do **GalleryRepository** (linhas 21 a 32). Por conta do caso especial envolvendo a página 1, quando loadSize é 3 vezes o valor configurado, temos que levar isso em conta para calcular o **offset** (linhas 27 a 32).

O próximo passo é obter uma página de dados (bloco de fotos) de **GalleryRepository**. A biblioteca **Paging 3** nos obriga a realizar essa operação em uma **thread** separada. Uma **thread** é uma linha de execução que será realizada em paralelo à linha de execução principal. Na linha 34 nós criamos a nova **thread** e guardamos sua referência na variável **service**.

Depois de criada a **thread**, nós temos que indicar o que será executado dentro dela. Isso é feito na linha 37: usamos o método **submit** para setar um **Callable**, que por sua vez define a função que será chamada no momento em que a **thread** for iniciada. No nosso caso, a função a ser chamada será **call** (linha 39). O objetivo desse método é obter uma página de dados de **GalleryRepository** (linha 42), calcular qual será a próxima página que deve ser carregada no futuro (linhas 43 a 46) e por fim montar um objeto do tipo **LoadResult**. Esse objeto é retornado pelo método call e é responsável por armazenar os dados da página obtidos e o valor para a próxima página calculado.

O objeto **LoadResult** criado e preenchido por **call** será guardado em um objeto do tipo **ListenableFuture**, na variável **lf** (linha 37). Esse objeto é um **container** que serve para guardar qualquer tipo de dado e que pode ser observado por outros objetos. Assim, quando um nova página de dados é obtida e guardada em **lf**, outros objetos são avisados disso e podem tomar ações como, por exemplo, atualizar a interface de usuário com os novos dados obtidos.

Passo 8.4 - Configuração do ViewModel e da Biblioteca Paging3

- Adicione no construtor da classe MainViewModel o código de configuração para usar a biblioteca de Paging 3:

```
1. public class MainViewModel extends AndroidViewModel {
2. ...
3.
4.     LiveData<PagingData<ImageData>> pageLv;
5.
6.     public MainViewModel(@NonNull Application application) {
7.         super(application);
8.         GalleryRepository galleryRepository = new
   GalleryRepository(application);
9.         GalleryPagingSource galleryPagingSource = new
   GalleryPagingSource(galleryRepository);
10.         Pager<Integer, ImageData> pager = new Pager(new
   PagingConfig(10), () -> galleryPagingSource);
```

```
11.         CoroutineScope viewModelScope =
    ViewModelKt.getViewModelScope(this);
12.         pageLv =
    PagingLiveData.cachedIn(PagingLiveData.getLiveData(pager),
    viewModelScope);
13.     }
14.
15.     public LiveData<PagingData<ImageData>> getPageLv() {
16.         return pageLv;
17.     }
18....
```

O código acima mostra o código necessário para configurar o uso da biblioteca **Paging 3**, para paginação de dados. Como nós estamos falando de dados, então a **ViewModel** é o lugar onde iremos guardar a página de dados referente ao bloco de fotos. Na linha 4 criamos um atributo novo para guardar a página de dados atual. Esse atributo é do tipo **LiveData<PagingData<ImageData>>**.

Um **LiveData** é um objeto container. Ele serve para guardar qualquer tipo de objeto. Aqui no nosso caso, estamos guardando dentro do **LiveData** um objeto do tipo **PagingData**, que por sua vez também é um container de dados genérico para um página de dados onde nós guardamos itens do tipo **ImageData**. Um container do tipo **LiveData** tem a capacidade de avisar outros objetos sempre que um novo dado é colocado em seu interior. Assim, a ideia aqui é que a interface de usuário da app irá “observar” o **LiveData**. Assim que uma nova página de dados for setada no **LiveData**, a interface será avisada e atualizará a interface do **RecycleView** para mostrar os novos itens. Na linha 15 há apenas um método para que o container **LiveData** possa ser obtido por outros objetos da app.

Dentro do construtor de **MainViewModel** iniciamos a configuração da **Paging 3** criamos uma instância de **GalleryRepository**, classe responsável por ler um bloco de fotos da galeria pública do celular (linha 8). Em seguida, criamos o **GalleryPagingSource** e passamos para ele a instância de **GalleryRepository** (linha 9). O **GalleryPagingSource** tem como função calcular qual bloco de dados será pedido para **GalleryRepository**, calcular qual será a próxima página de dados e montar um objeto de resposta contendo os dados da página atual e o número da próxima página calculado. Na linha 10 nós iniciamos a biblioteca de **Paging 3** passado um objeto de configuração de paginação e a instância de **GalleryPagingSource**. Aqui nós configuramos a biblioteca **Paging 3** para obter 10 itens de fotos cada vez que uma nova página for pedida. Na linha 11 nós obtemos o objeto de escopo de **MainViewModel**, o qual será usado para guardar páginas de dados no cache. Na linha 12, nós obtemos o **LiveData** que foi gerado pela biblioteca de **Paging 3** e guardamos esses dados no cache de **MainViewModel**.

Passo 8.5 - Configuração do PagingDataAdapter

- Crie a classe padrão MyViewHolder;
- Crie a classe ListAdapter como abaixo, para cuidar do preenchimento do RecycleView para a lista em formato padrão.

```
1. public class ListAdapter extends PagingDataAdapter<ImageData,
   MyViewHolder> {
2.
3.
4.     public ListAdapter(@NonNull DiffUtil.ItemCallback<ImageData>
   diffCallback) {
5.         super(diffCallback);
6.     }
7.
8.     @NonNull
9.     @Override
10.    public MyViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
   int viewType) {
11.        LayoutInflater layoutInflater =
   LayoutInflater.from(parent.getContext());
12.        View view = layoutInflater.inflate(R.layout.list_item, parent,
   false);
13.        return new MyViewHolder(view);
14.    }
15.
16.    @Override
17.    public void onBindViewHolder(@NonNull MyViewHolder holder, int
   position) {
18.
19.        ImageData imageData = getItem(position);
20.
21.        TextView tvName = holder.itemView.findViewById(R.id.tvName);
22.        tvName.setText(imageData.fileName);
23.
24.        TextView tvDate = holder.itemView.findViewById(R.id.tvDate);
25.        tvDate.setText("Data: " + new SimpleDateFormat("HH:mm
   dd/MM/yyyy").format(imageData.date));
26.
27.        TextView tvSize = holder.itemView.findViewById(R.id.tvSize);
28.        tvSize.setText("Tamanho: " + String.valueOf(imageData.size));
29.
30.        Bitmap thumb = imageData.thumb;
31.        ImageView imageView =
   holder.itemView.findViewById(R.id.imThumb);
32.        imageView.setImageBitmap(thumb);
33.    }
34.}
```

O código acima contém o código referente ao Adapter do **RecycleView** que mostra a lista de fotos no formato padrão. Esse Adapter herda de um classe específica de Paging 3, a classe **PagingDataAdapter** (linha 1).

PagingDataAdapter é uma especialização da classe Adapter preparada para trabalhar de forma integrada com a biblioteca Paging 3. No geral, ela não possui muitas diferenças de um Adapter normal: como se pode ver acima, os métodos **onCreateViewHolder** (linha 10) e **onBindViewHolder** (linha 17) tem o mesmo comportamento de um **Adapter** normal. Aqui,

entretanto, não temos que implementar o método **getItemCount**. Isso ocorre pois no caso de **PagingDataAdapter** o número de itens da lista é variável.

- O código acima mostra somente a implementação de ListAdapter. Crie a classe GridAdapter para preencher os itens da lista em formato Grid.

Outra diferença é que o construtor de **PagingDataAdapter** recebe como parâmetro de entrada um objeto **diffCallback**. Esse objeto serve para indicar ao Adapter como comparar dois elementos da lista. A implementação de uma classe desse tipo segue abaixo.

- Crie a classe ImageDataComparator como abaixo:

```
1. class ImageDataComparator extends
   DiffUtil.ItemCallback<ImageData> {
2.     @Override
3.     public boolean areItemsTheSame(@NonNull ImageData oldItem,
4.                                    @NonNull ImageData newItem)
   {
5.         // Id is unique.
6.         return oldItem.uri.equals(newItem.uri);
7.     }
8.
9.     @Override
10.      public boolean areContentsTheSame(@NonNull ImageData
   oldItem,
11.                                        @NonNull ImageData
   newItem) {
12.          return oldItem.uri.equals(newItem.uri);
13.      }
14.  }
```

A classe **ImageDataComparator** tem como objetido mostrar ao **Adapter** como verificar se dois itens do tipo **ImageData** são iguais. Ela possui dois métodos que devem ser reimplementados:

- **areItemsTheSame** (linha 3) -> esse método recebe dois objetos do tipo ImageData e retorna true caso eles sejam iguais e false caso contrário;
- **areContentesTheSame** (linha 10) -> esse método recebe dois objetos do tipo ImageData e retorna true caso eles tenham o mesmo conteúdo e false caso contrário;

Passo 9 - Configuração dos Fragmentos

Ao criar os fragmentos **ListViewFragment** e **GridViewFragment**, cada um deles vem com um arquivo .**java** e um arquivo de **layout** associado. Nas seções anteriores já criamos o layout dos fragmentos. Agora é hora de configurar o .**java** referente a eles.

- Modifique a classe ListViewFragment para ficar como abaixo:

```
1. public class ListViewFragment extends Fragment {
2.
```

```
3.    private MainViewModel mViewModel;
4.    private View view;
5.
6.    public static ListViewFragment newInstance() {
7.        return new ListViewFragment();
8.    }
9.
10.   @Override
11.   public View onCreateView(@NonNull LayoutInflater inflater,
   @Nullable ViewGroup container,
12.                            @Nullable Bundle savedInstanceState) {
13.       view = inflater.inflate(R.layout.list_view_fragment,
   container, false);
14.       return view;
15.   }
16.
17.   @Override
18.   public void onViewCreated(@NonNull View view, @Nullable Bundle
   savedInstanceState) {
19.       super.onViewCreated(view, savedInstanceState);
20.       mViewModel = new
   ViewModelProvider(getActivity()).get(MainViewModel.class);
21.       ListAdapter listAdapter = new ListAdapter(new
   ImageDataComparator());
22.       LiveData<PagingData<ImageData>> liveData =
   mViewModel.getPageLv();
23.       liveData.observe(getViewLifecycleOwner(), new
   Observer<PagingData<ImageData>>() {
24.           @Override
25.           public void onChanged(PagingData<ImageData>
   objectPagingData) {
26.
   listAdapter.submitData(getViewLifecycleOwner().getLifecycle(),objectP
   agingData);
27.           }
28.       });
29.
30.       RecyclerView rvGallery = (RecyclerView)
   view.findViewById(R.id.rvList);
31.       rvGallery.setAdapter(listAdapter);
32.       rvGallery.setLayoutManager(new
   LinearLayoutManager(getContext()));
33.   }
34.}
```

O código acima mostra o código referente a **ListViewFragment**. Por padrão, quando um **Fragment** é criado usando o **Wizard** do **Android Studio**, ele já vem acompanhado de uma implementação básica composta pelos seguintes métodos:

- **newInstance** (linha 6) -> cria uma instância do Fragmento e retorna;
- **onCreateView** (linha 11) -> esse método é chamado automaticamente pelo sistema toda vez que o fragmento é setado em uma **Activity**. Ele é responsável por criar a interface de usuário do fragmento.

Além dessa implementação básica, nós reimplementamos o seguinte método também:

- **onViewCreated** (linha 18) -> esse método é chamado logo após onCreateView. Quando esse método é chamado, é garantido de que todos os elementos da interface do fragmento já foram criados. É dentro deste método que iremos configurar o **RecycleView**.

Dentro do método **onViewCreated**, iniciamos obtendo uma referência para MainViewModel (linha 20). Em seguida criamos uma instância de **ListAdapter** e passamos uma instância de **ImageDataComparator** para **ListAdapter** (linha 21).

Dentro de **MainViewModel**, a biblioteca Paging 3 já foi configurada para buscar e gerar novas páginas de dados. Essas páginas de dados são disponibilizadas através do containter **LiveData**. Na linha 22 nós obtemos o **LiveData** de **MainViewModel** e começamos a observá-lo na linha 23. Toda vez que uma nova página de dados estiver disponível, o **LiveData** chamará o método **onChanged** (linha 25) entregando a nova página de dados. Na linha 26 nós submetemos essa página de dados em **ListAdapter**, que será responsável por atualizar o **RecycleView** com esse novo bloco de dados.

Por fim, é realizada a configuração padrão do **RecycleView** (linhas 30 a 32).

- O código acima mostra somente a implementação de **ListViewFragment**. Faça o mesmo para **GridViewFragment**. A implementação é quase a mesma, sendo somente diferente o uso de **GridViewAdapter** e **GridLayoutManager**.

Passo 10 - Teste

Teste sua aplicação, veja se ela possui problemas e os corrija antes de enviar para o professor.

Passo 11 - Envio da tarefa

Realize o commit e o push da sua app para o GitHub. **Envie no AVA somente o link para o GitHub!**