IMD 4008 / Mobile User Interfaces

Tutorial 3 - Transaction Logger

Objectives

1. Become familiar with some new Views and Layouts.
2. Launch one Activity from another, engaging the device Camera
3. Develop a (somewhat) “realistic” (although not pretty!) looking app

Introduction

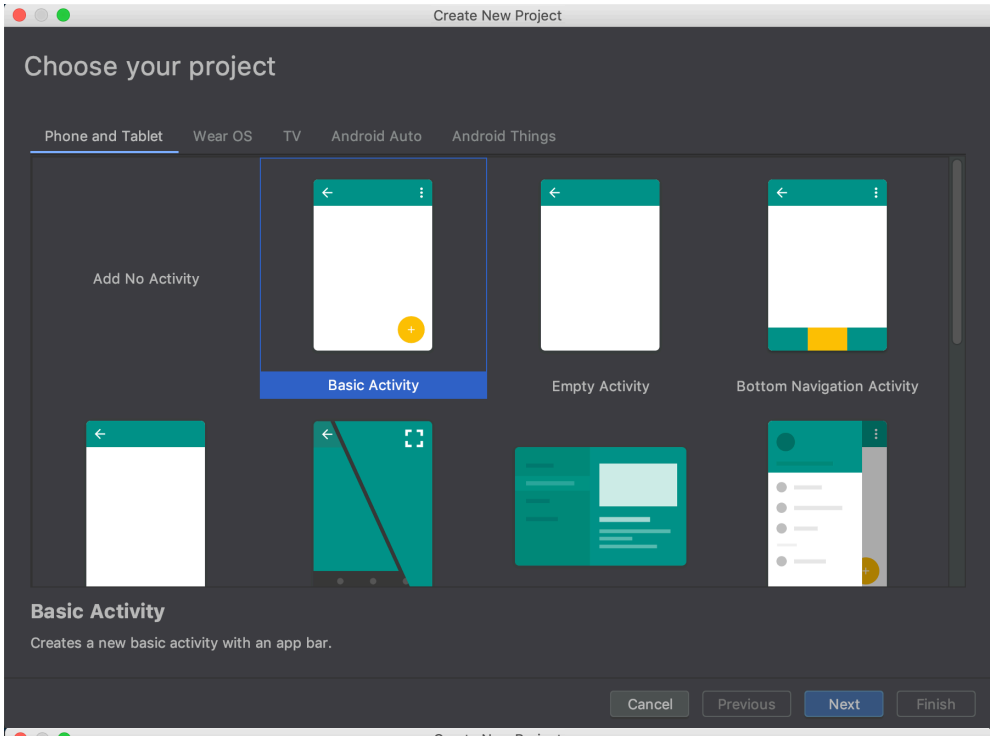
In this tutorial, we will develop a simple UI to log expense report transactions. This will be the first “realistic” app we develop (i.e., an app with a purpose other than educational). The transactions logged by the app will consist of a description, amount of transaction, date of transaction, category, and an option to add an image of the transaction receipt.

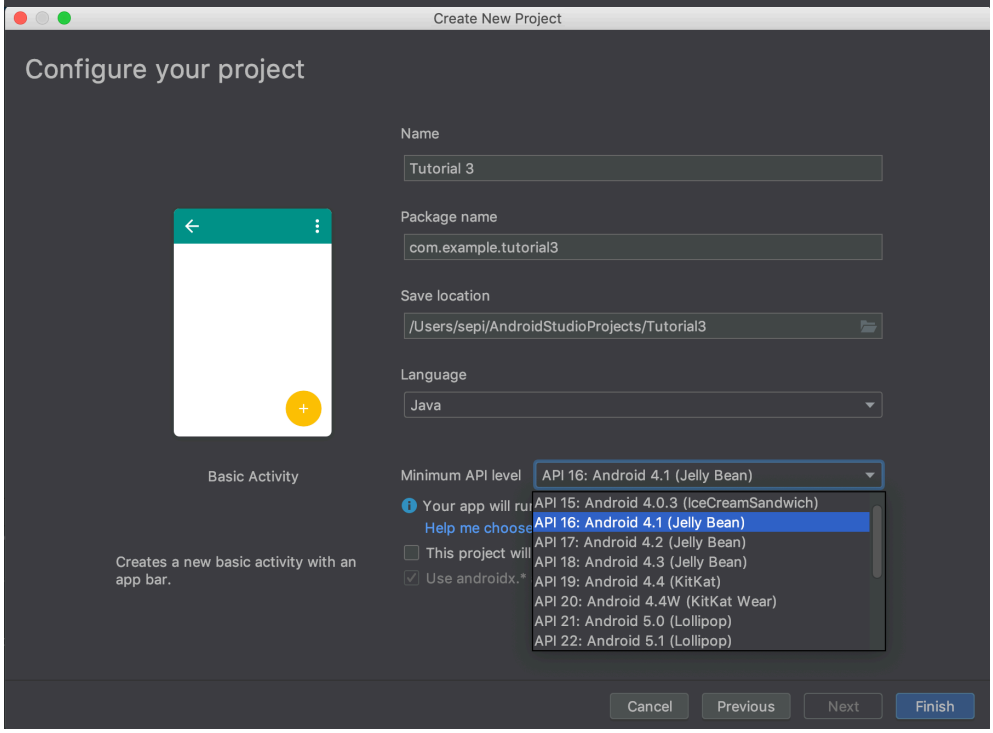
We will be diving deeper into setting up the layout of the UI, as well as using new components to get more comfortable using everything that Android Studio has to offer.

At the end of the tutorial, you should have created a UI that looks and behaves like this video.

1. Setting Up the Layout

Create a new project with a *basic Activity* to start and choose API 16 (Jelly Bean).



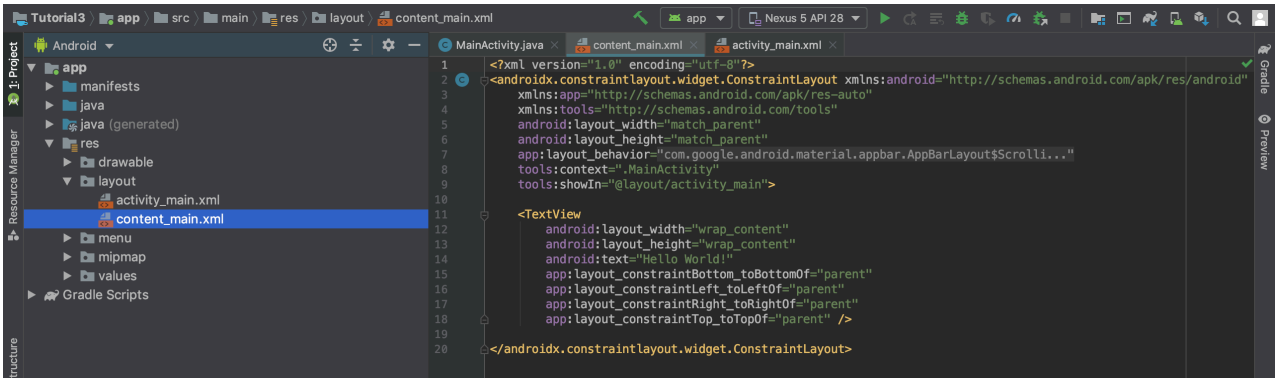


Once your project is created, notice how you now have two different XML files under the app > res > layout folder. One is called *content_main.xml*, the other is called *activity_main.xml*. The activity_main.xml file can be seen as the outer part of the layout, which includes the floating action button (bottom right of the screen), the toolbars, and so on. You can think of this as the “stock” components provided when you choose a basic Activity (rather than the empty Activity we’ve been using up to this point). Look carefully through the activity_main.xml; note the line

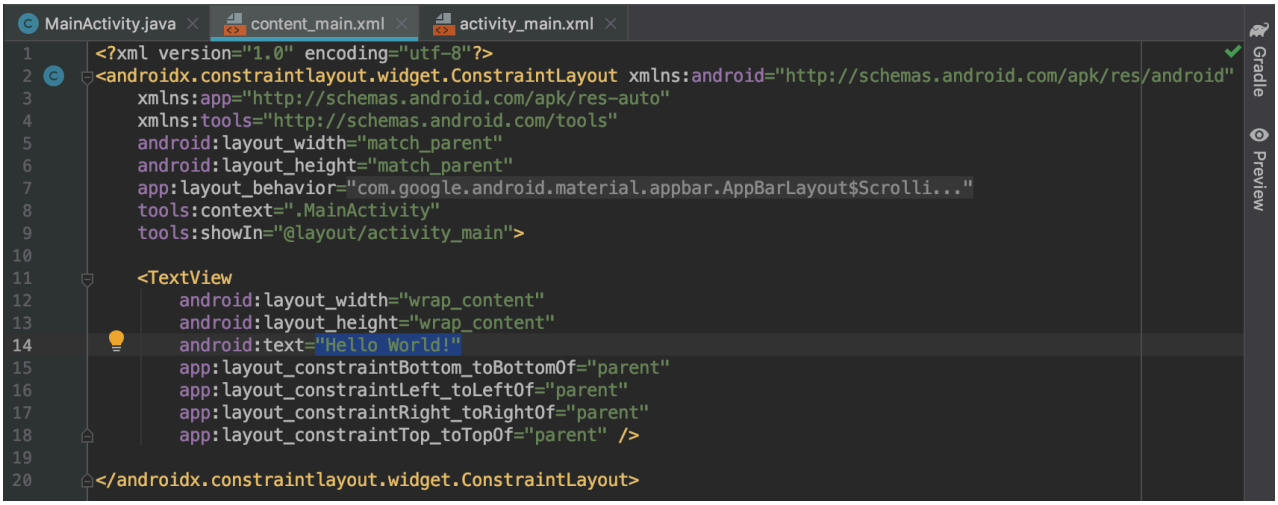
```
<include layayout="@layout/content_main"/>
```

From your past programming experience, you can probably guess that this include statement takes the layout from the content_main.xml file and effectively pastes it into the overall layout. Follow the link to read more about how the include statement works.

The content_main.xml file is where you define how your layout looks in the blank space available. For now, we will be working with this XML file, rather than activity_main.xml.



Open the text view of your content_main.xml file and take a look at the XML code that has been auto-generated for you. Note that the only View contained is the default “HelloWorld” TextView. All the other Views seen in the Design tab are provided by the “wrapper” layout from activity_main.xml.



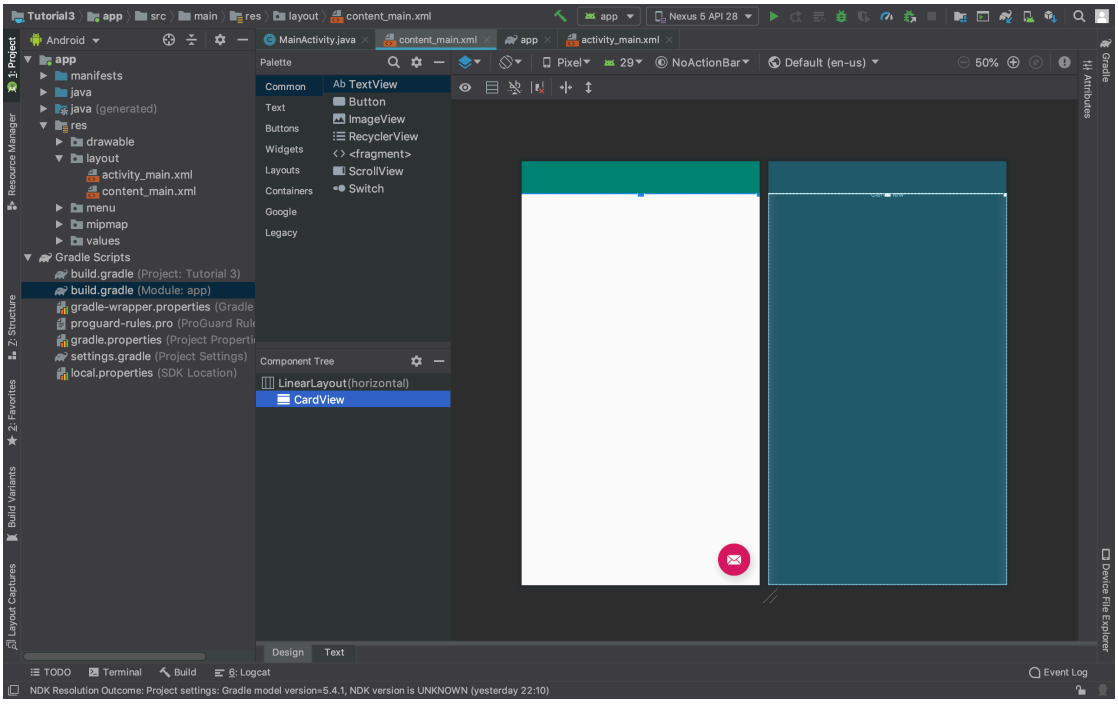
Notice on the second line, the layout is defined as “<androidx.constraintlayout.widget.ConstraintLayout”. For this tutorial, we want to use a different Layout, the “LinearLayout”. Replace the layout type with LinearLayout (leaving xmlns:android as-is). Don’t forget to update the closing tag to match! (The IDE may do this automatically for you though). Finally, add the *android:orientation* attribute inside the LinearLayout open tag, as shown below:

```
MainActivity.java | content_main.xml | activity_main.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="com.google.android.material.appbar.AppBarLayout$Scrolli..."
    tools:context=".MainActivity"
    tools:showIn="@layout/activity_main">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</LinearLayout>
```

Next, we will also get rid of the “Hello World” TextView leaving us with a blank Vertical LinearLayout to start working on.

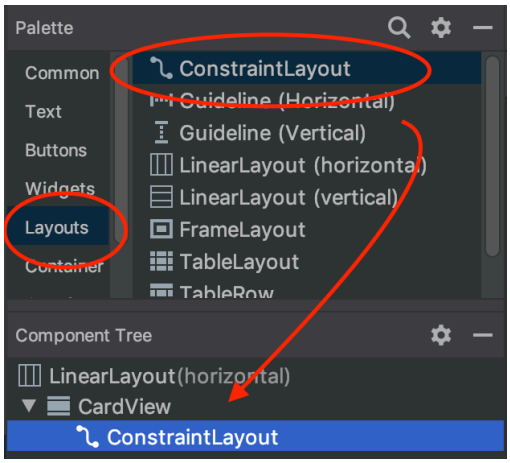
```
MainActivity.java | content_main.xml | activity_main.xml

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layout_behavior="com.google.android.material.appbar.AppBarLayout$Scrolli..."
    tools:context=".MainActivity"
    tools:showIn="@layout/activity_main">

</LinearLayout>
```

Now, check your Component Tree to ensure that the newly added CardView is a child of the Vertical LinearLayout. Read the reference on the CardView to get a better idea of how they work.

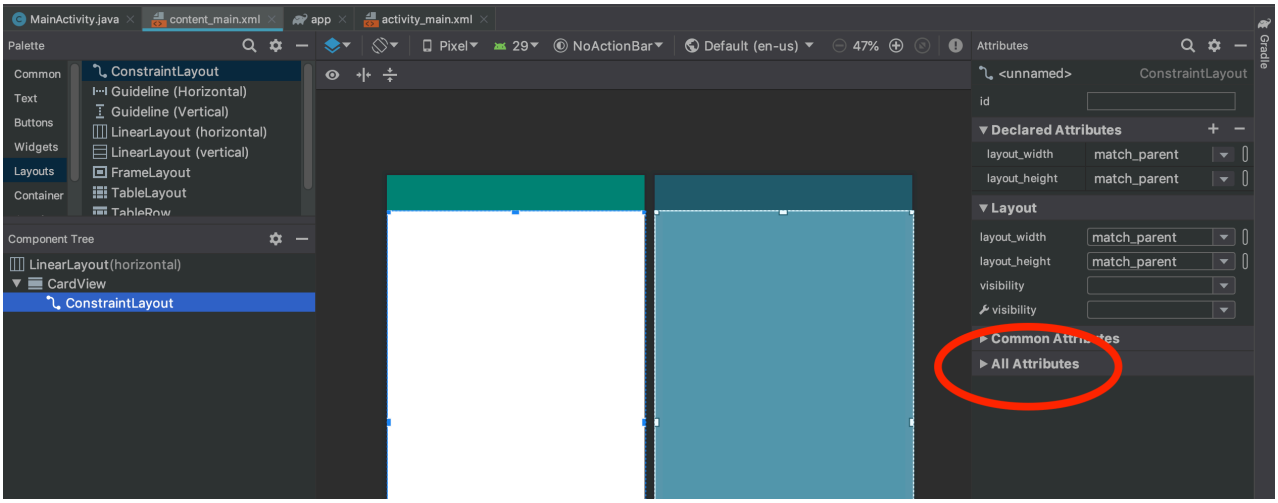


We will be using the CardView to group related information together, in a further nested Layout. This is commonly employed in Android applications (the reference above depicts an example). In our case we will be using the CardView to group UI fields together so that the layout makes visual sense. By default, CardViews stack widgets the contain on top of one another. You can observe this behaviour by trying to add a couple of Views (e.g., a TextView and a Button) to the CardView - in the Design window, they will overlap; if you try this step, delete this Views before moving on.

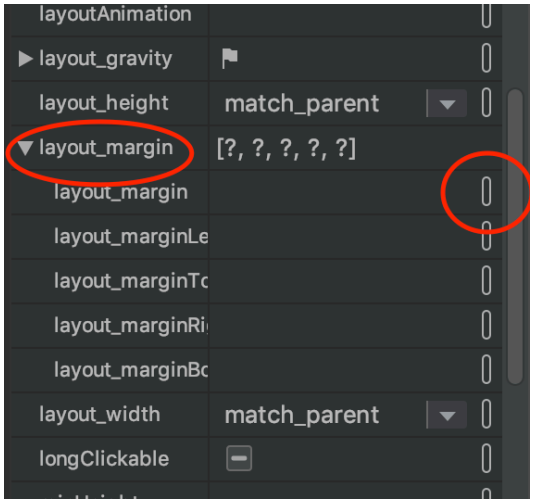
This is clearly undesired behaviour, so we need to add a Layout nested in the CardView to control how the Views are laid out within the CardView. In our case, we will add a ConstraintLayout as a child of CardView by dragging it from the palette to the Component Tree onto the CardView:



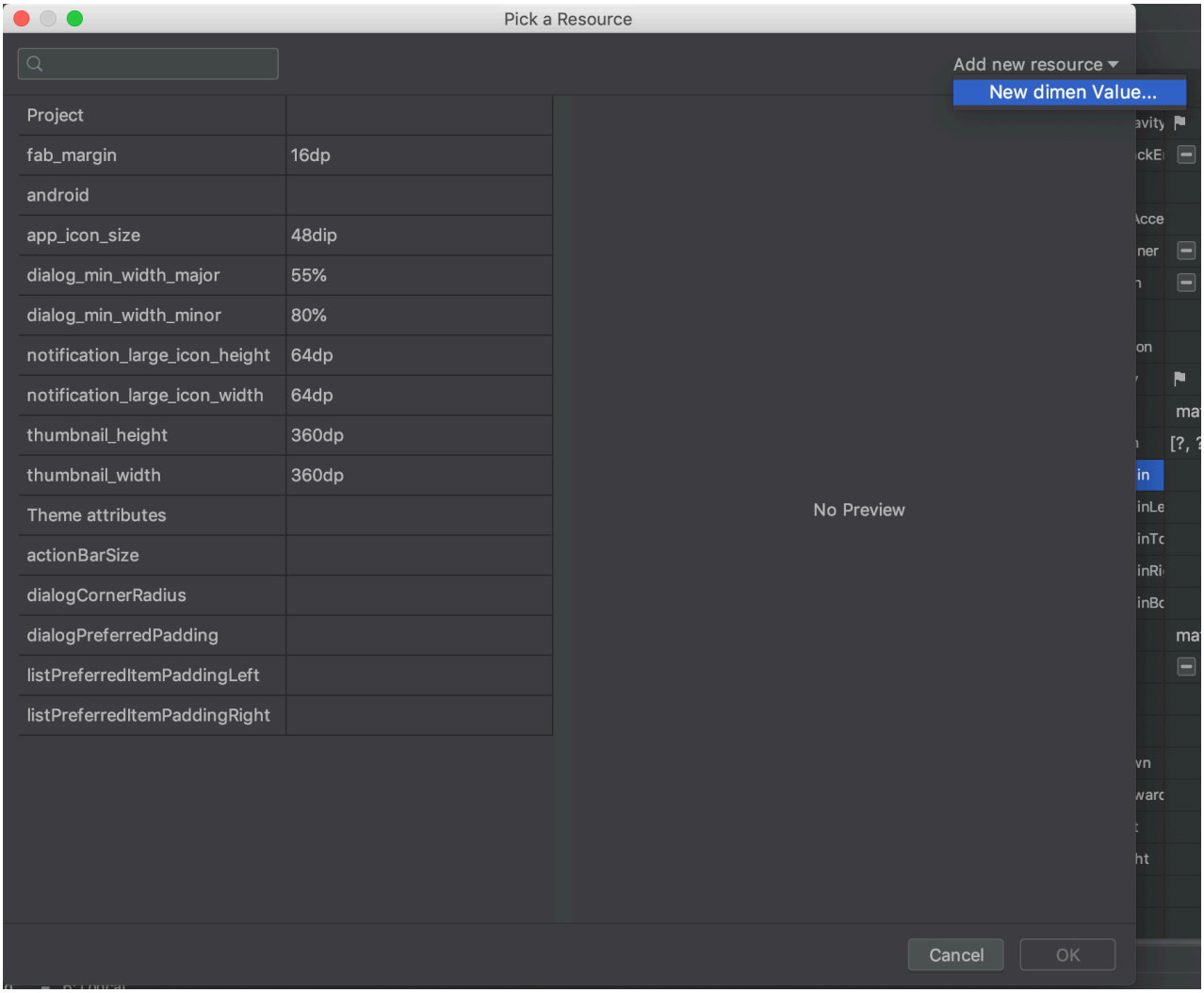
Now we will modify some attributes of the ConstraintLayout to improve its visual appearance. First, let's add a margin to our ConstraintLayout. Select ConstraintLayout in the Component Tree and in the attributes panel over to the left select All Attributes.



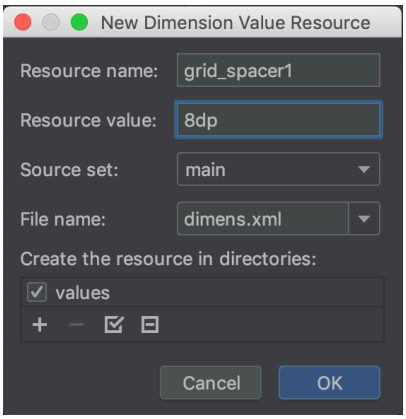
Open the section titled Layout_Margin, and in the option for all we will create a new resource by clicking on the button that is shown below:



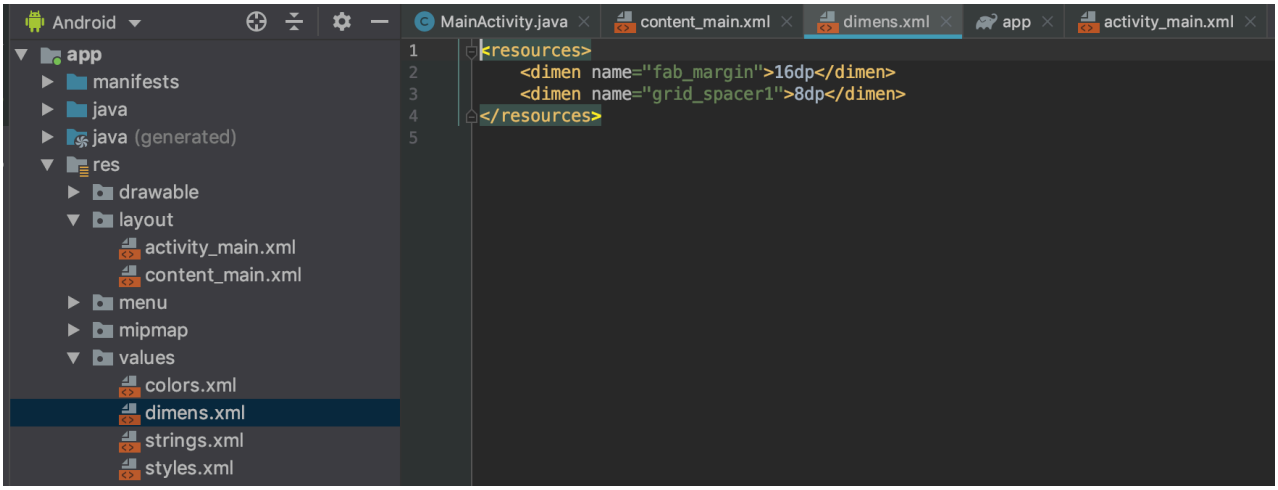
Select Add new resource/New dimen value from the popup menu:



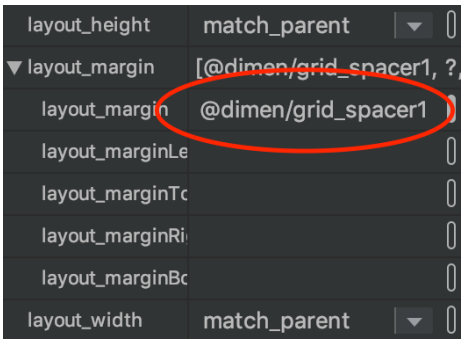
In the menu that shows up, we will create a resource named “grid_spacer1” with a value of 8dp.



Leave the rest as default. What this does, it creates a new resource in our dimens.xml file that we can reuse throughout our project. Try opening the dimens.xml file under res/values/dimens.xml and notice that a new XML component has been added for the dimension resource we created through this process.



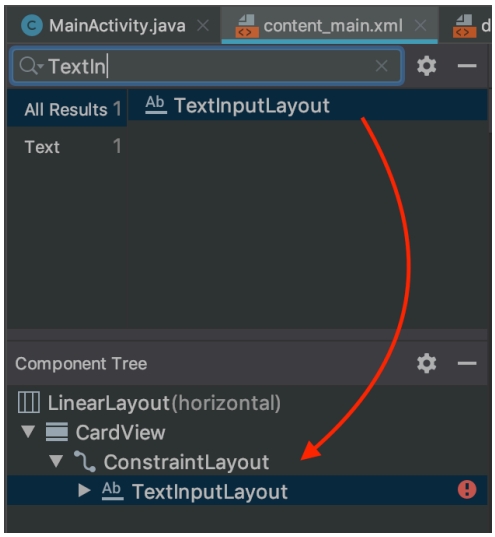
Also notice that in our attributes for our ConstraintLayout, the value for **Layout_margin** has been updated with a reference to this resource, and that our layout now has a margin on all sides of 8dp:



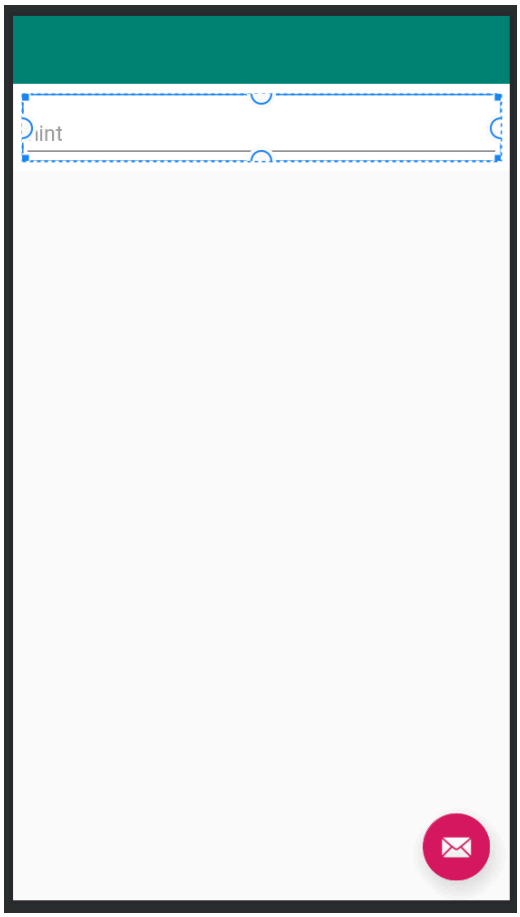
2. Adding content

Up to this point, we have been using the “TextView” widget to add text components to our UI’s. This time around, we will introduce the “TextInputLayout”. This is a Layout which wraps an EditText View with a floating grayed out hint label. The EditText allows user input and can be customized for the type of text format it allows, such as dates, all caps, email, etc.

We will create an initial TextInputLayout for a field called “Description” in our app (which will eventually be used for a Description of an expense being logged). Go to the palette window and search for TextInputLayout, once found, click and drag it as a child of our ConstraintLayout:

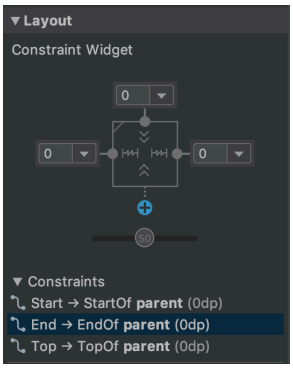


Now you should see the component show up in the design window as your first widget on the screen:

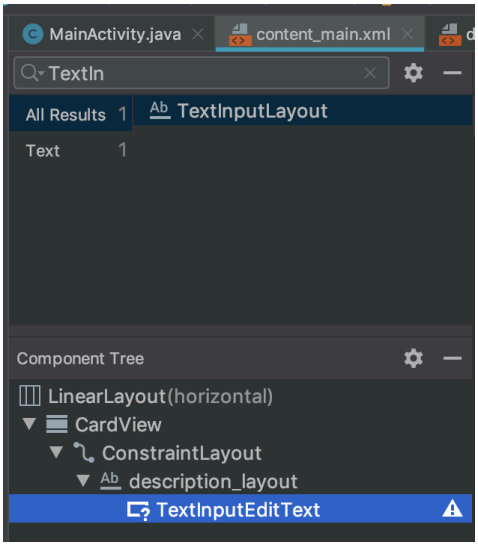


Set the ID attribute of the newly created TextInputLayout to *description_layout*.

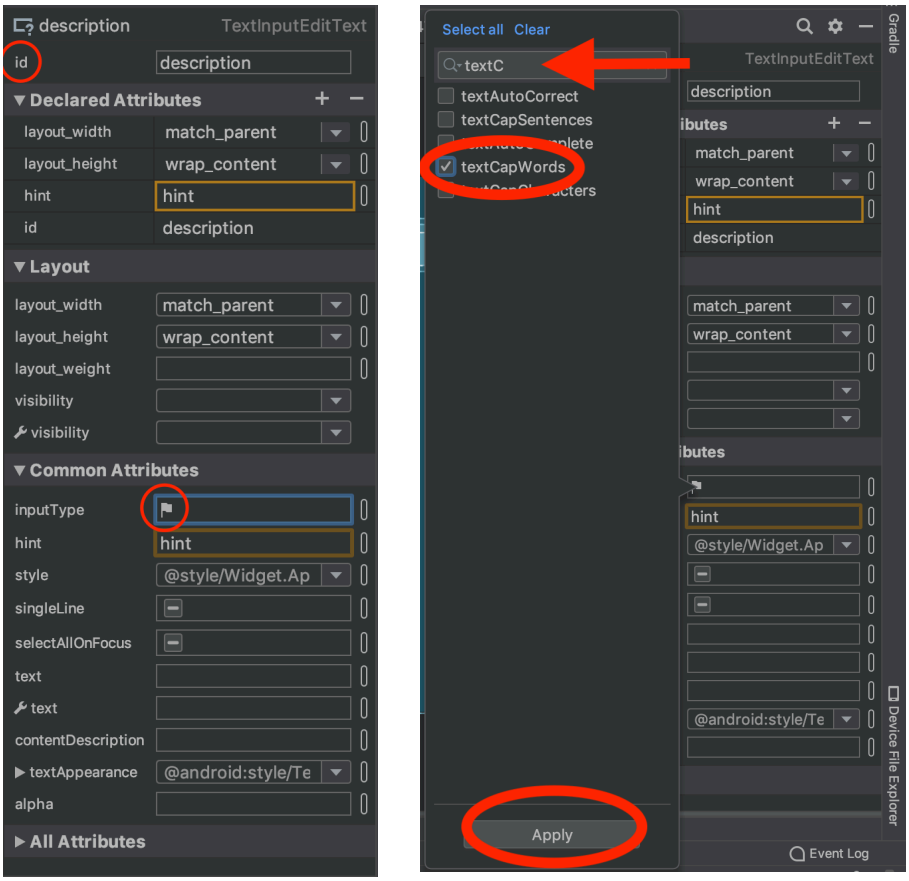
Select description_layout in the Component Tree (note that you do not accidentally select the EditTExt contained in it! Be sure you are selecting the entire TextInputLayout). Using the blueprint view, set constraints by dragging the nodes at the edges of description_layout to the edges of the containing CardView. Then, set the left and top margin constraints to 0 by clicking on the + button in the diagram as shown below:



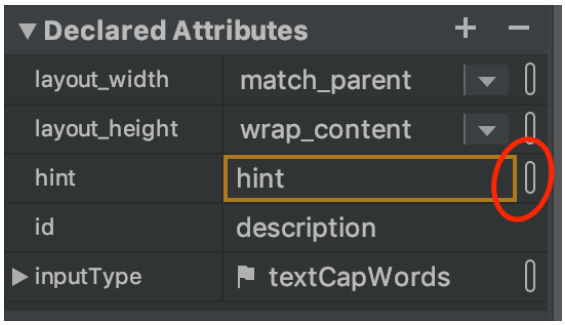
This will snap your TextInputLayout to the top left corner of its parent. Now using the Component Tree select the TextInputEditText inside the *description_layout* TextInputLayout.

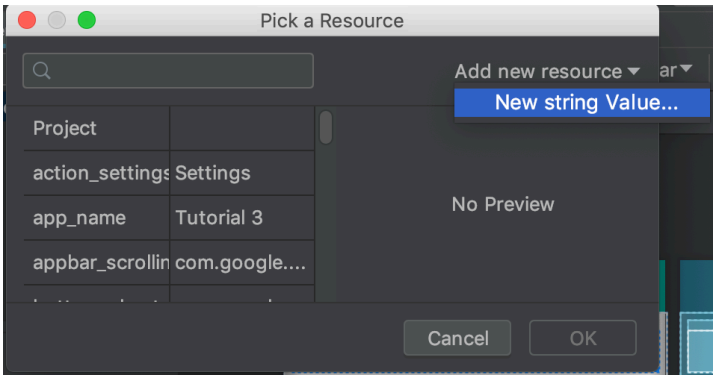


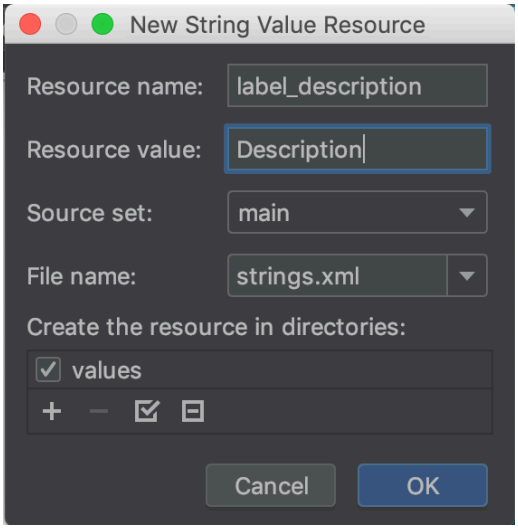
Change its ID to *description*, as well as changing the inputType to *textCapWords* as shown below. As usual, if you don't see the attribute you are looking for, you can search for it or find it in the "All Attributes" window.



Now we will change the *hint* attribute. Right now, it is hardcoded with the string *hint.* We will create a new resource for it and name it *label_description* with the value *Description* as detailed below:

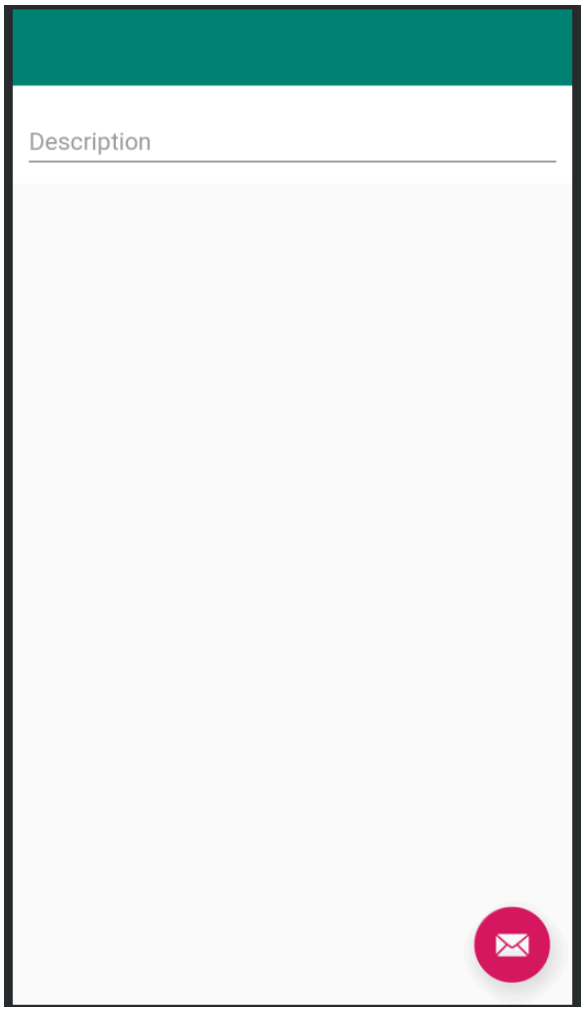



Pick a Resource
Add new resource
New string Value...
Project
action_settings
Settings
app_name
Tutorial 3
appbar_scrollin
com.google....
No Preview
Cancel
OK


New String Value Resource
Resource name:
label_description
Resource value:
Description
Source set:
main
File name:
strings.xml
Create the resource in directories:
values
Cancel
OK

As before, this process adds a new resource to our project. In this case, it adds a new entry to the strings.xml file (under app > res > values > strings.xml). You can also do this through the XML method seen in previous tutorials, if you are more comfortable using that approach. Either produces the same result.

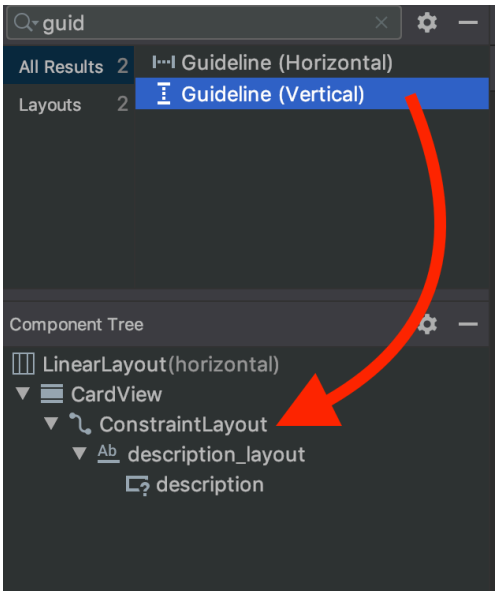
Now our UI should look something like that shown to the right.

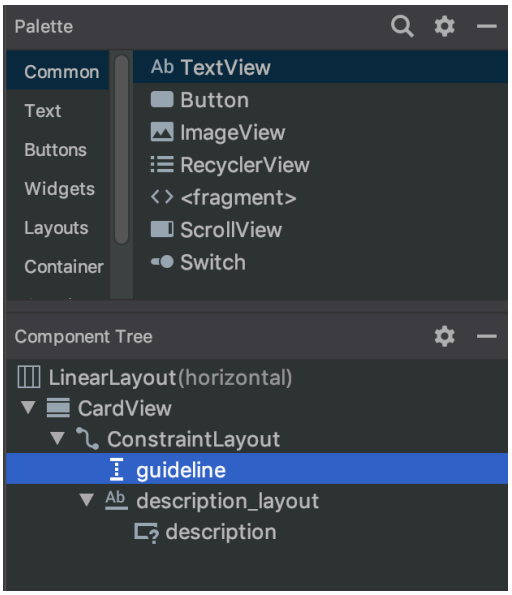


At this point, we want to resize our Description text to fit only 40% of the horizontal space available.

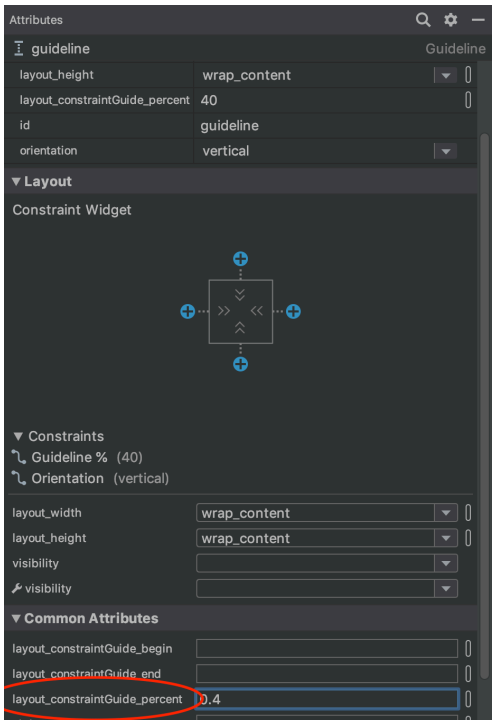
To do this, there is a component called a Guideline (Horizontal or Vertical) available in the palette window. Guidelines are a “helper” component. By default, they have no visual appearance but can be used to help control your layout.

Search for Guideline (vertical) in the palette window, select it, then drag the Guideline to the ConstraintLayout in the Component Tree (i.e., add the Guideline as a child of the ConstraintLayout) as shown below:

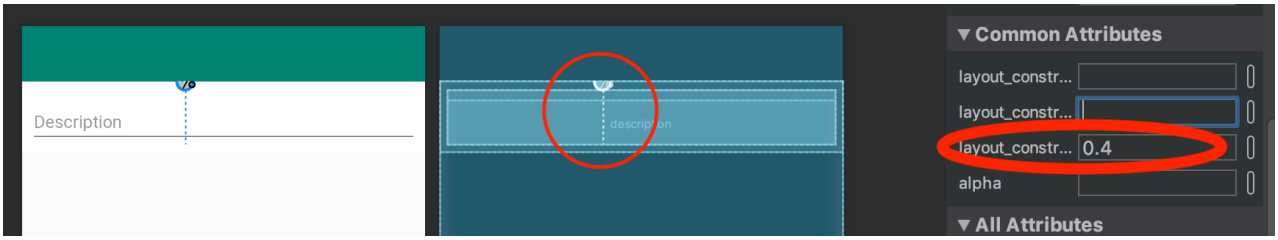




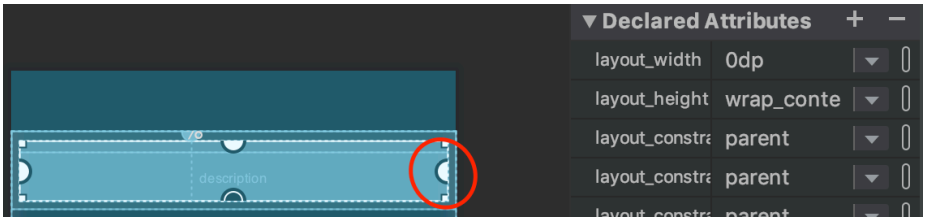
Select the newly added guideline and in the attributes panel change the *layout_constraintGuide_percent* attribute to 0.4 (40%).

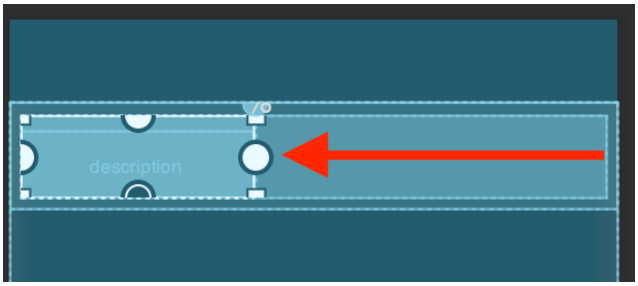


Now check your UI design and you'll notice there is a vertical line placed 40% from the left in your constraintLayout:

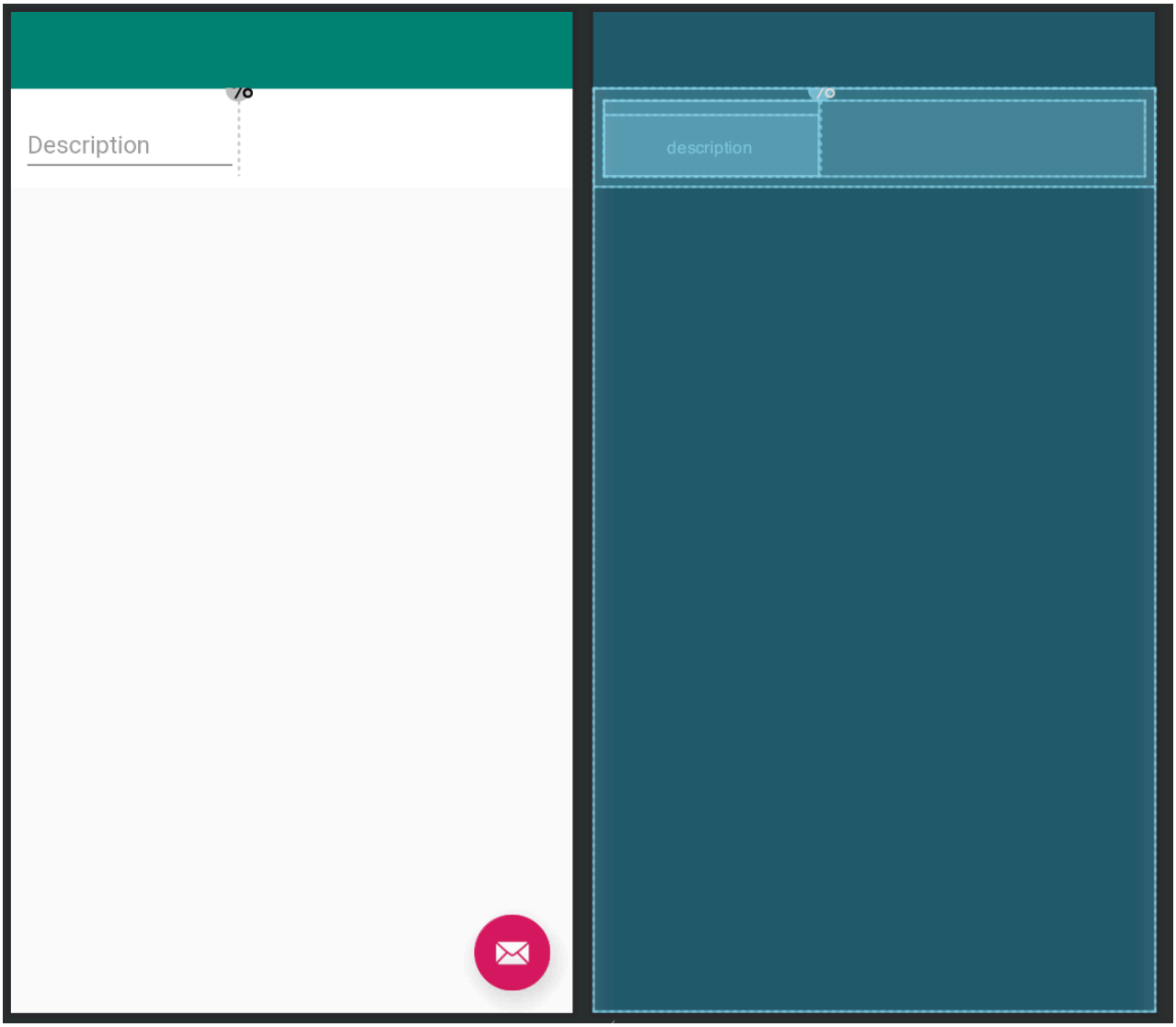


Under the Component Tree, select the description_layout, in the parameters panel, set it's layout_width to 0dp. Now check the design window, click and drag the constraint node on the right side of the widget and align it with our guideline component, this will set it's right side to be constrained with the guideline, which in turn is placed 40% from the left:

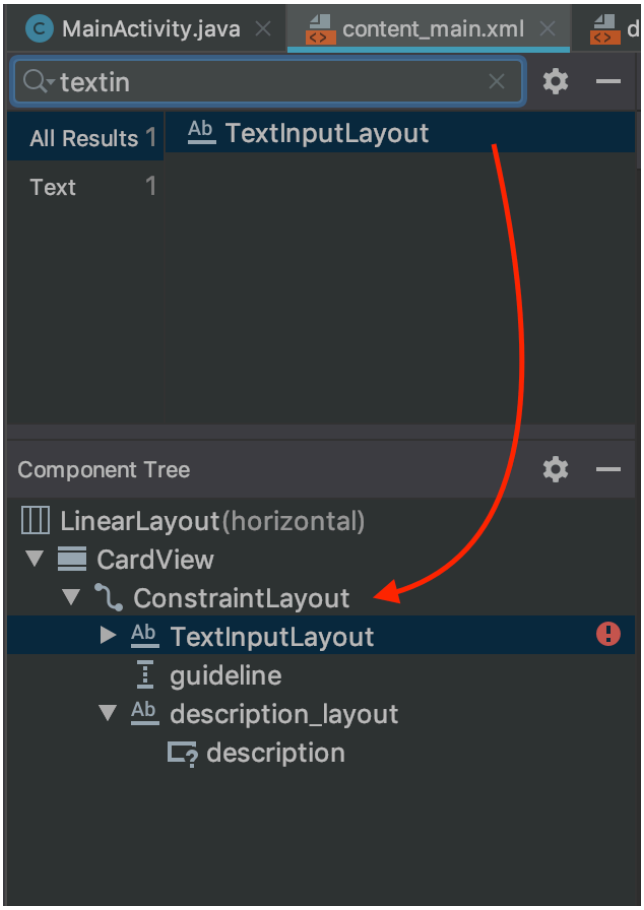




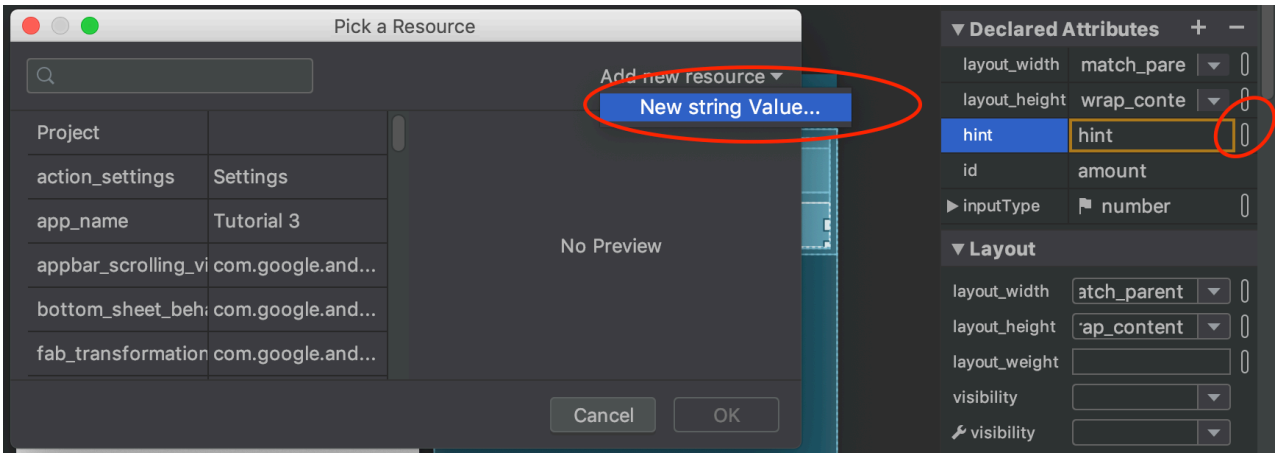
The result should look like the following image. Your Description TextInputLayout should now only span 40% of the available space.

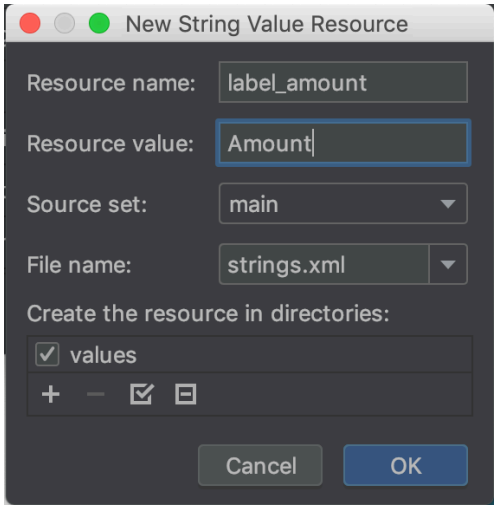


Now we will add a second TextInputLayout for the amount input (i.e., the value of the transaction being logged). As before, find the TextInputLayout in the palette window and add it as a child of ConstraintLayout in the Component Tree panel:

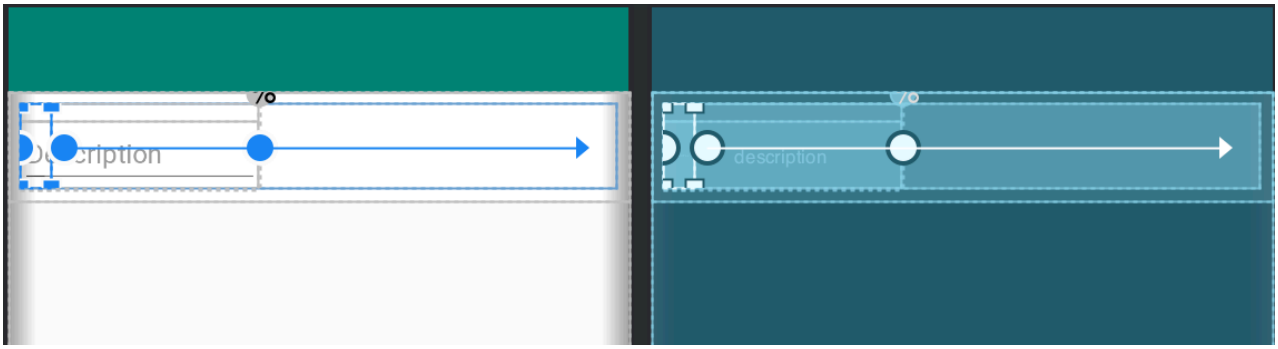


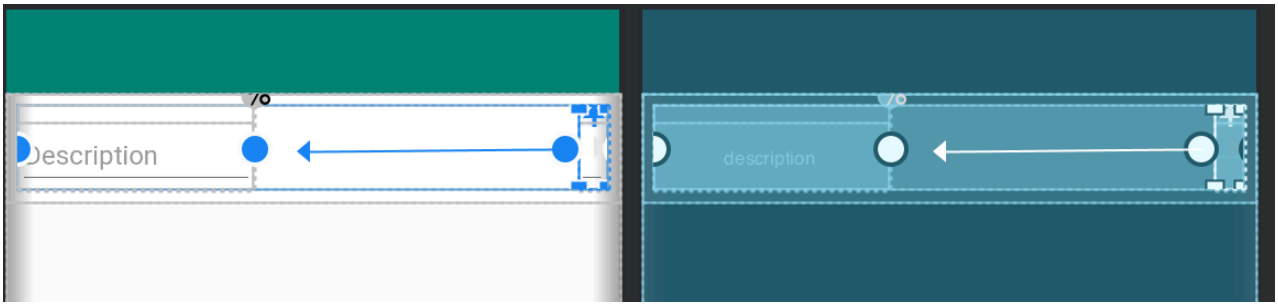
As before, name the TextInputLayout, setting its ID to *amount_layout*. Now select the child TextInputEditText, change the ID to *amount*, and change the inputType to number. For the hint attribute, open the resource editor to create a new string resource, name the resource *label_amount* and give it as a value *Amount*.





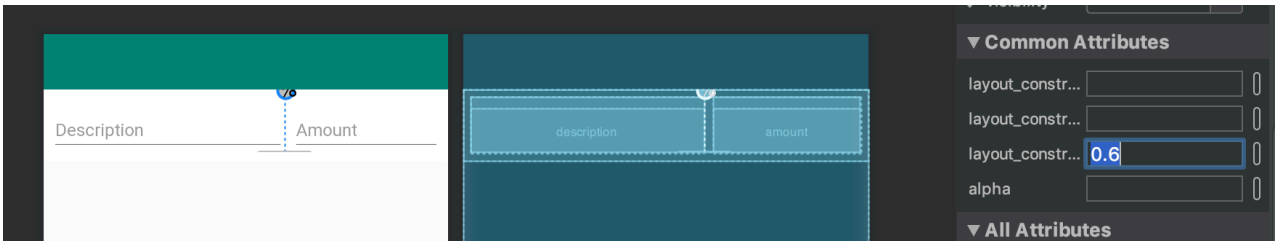
This new TextInputLayout has no constraints yet. We want it to be positioned beside the Description TextInputLayout. We can again use the guideline we previously set up to help with this. In the attributes, change it's layout_width to 0dp - this makes it easier to select its constraint handle. Now, select and drag the constraint handle to constrain the TextInputLayout's right side to the right edge of our CardView, the top side to the top edge of CardView, and the left side to the guideline component:





We have to properly align the margins. In the attributes panel, change the right and top margins to 0 and left to 8 to align it the same as the Description widget:

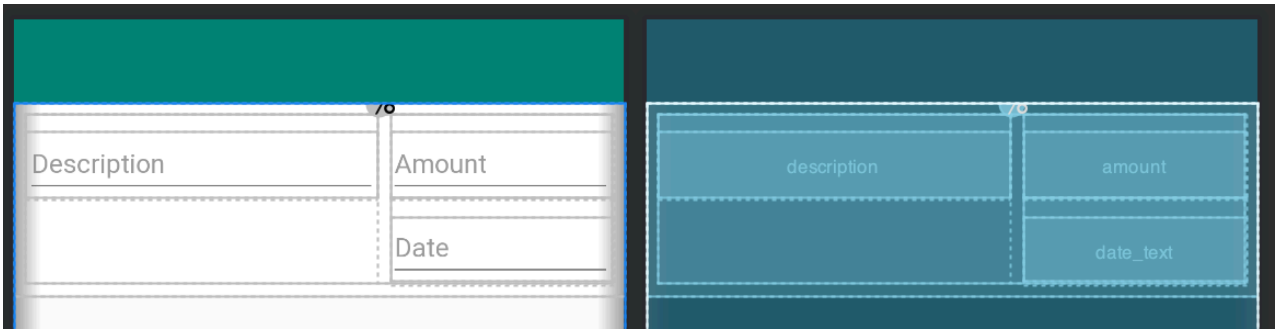
At this point, feel free to experiment with the size of the guideline by adjusting it's layout_constraintGuide_percent property. Changing it to 0.6 will cause the Description TextInputLayout to take up 60% of the space, while the Amount TextInputLayout will take up the remaining 40%. Experiment with this until it looks the way you like it.



Next step is to add the transaction date input. Similarly to how you did the first two TextInputLayouts, add a new TextInputLayout View as a child of the ConstraintLayout. Give it the ID *date_layout* and its child TextInputEditText should be called *date_text*. Create a new resource for the hint named *label_date* with a value of *Date*. Set the inputType of the child date_text to be date. If you forget how to do any of these steps, look back to the earlier instructions.

Position date_layout underneath the Amount widget making sure to adjust the appropriate constraints and margins. Like before, set the layout_width to 0dp, then constrain the top edge of the Date to the bottom edge of Amount, similarly, constrain the right edge to the right of the screen, and the left to the Guideline, after which, you should zero out the values in the constraints for Date.

At this point, your UI should look something like:

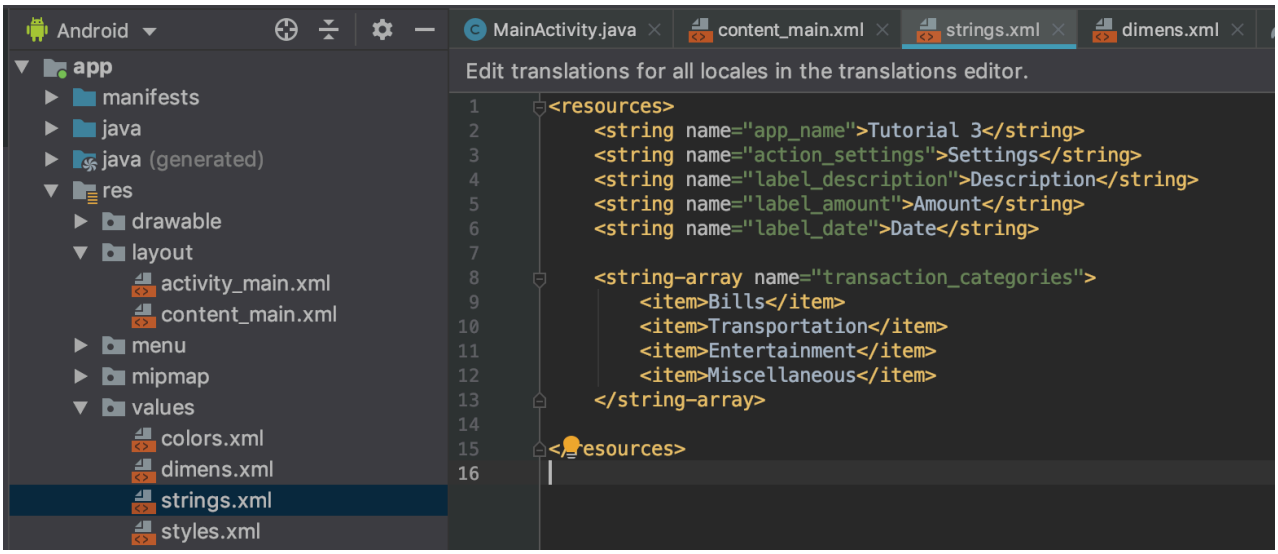


In this next part, we will create a new CardView for a second portion of our UI. Drag a new CardView into the Component Tree, making sure to add it as a child of the LinearLayout. Make sure it comes *after* the previous CardView in the Component Tree (otherwise, they will swap positions in the layout; if your layout suddenly seems to go blank, this is likely the problem). Like before, add a ConstraintLayout as a child of the new CardView, the same way we did for the first block. For the newly created ConstraintLayout, add the grid_spacer we created for the first one in the extended attributes for the *all* margin.

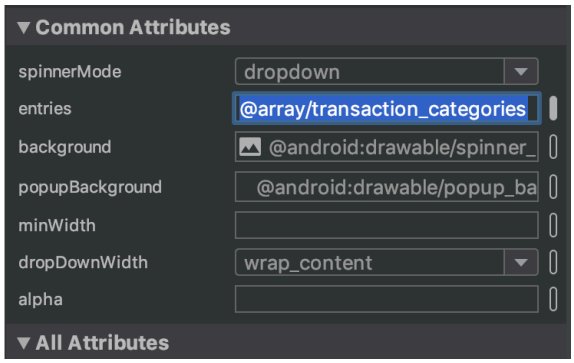
In this new block, we will add the option to add what category this transaction should go into (Transportation, Entertainment, Bills, Miscellaneous). For this, we will use the Spinner widget, which acts similarly to a drop-down menu. Search for the spinner widget in the Palette panel and add it as a child of our new ConstraintLayout. Constrain the top, left, and right edges to the top, left, and right edges of our ConstraintLayout, setting the margins to something different than the default number if you wish.

To add entries to our Spinner widget, open the strings.xml in the res/values folder. Create a new string-array resource as follows:

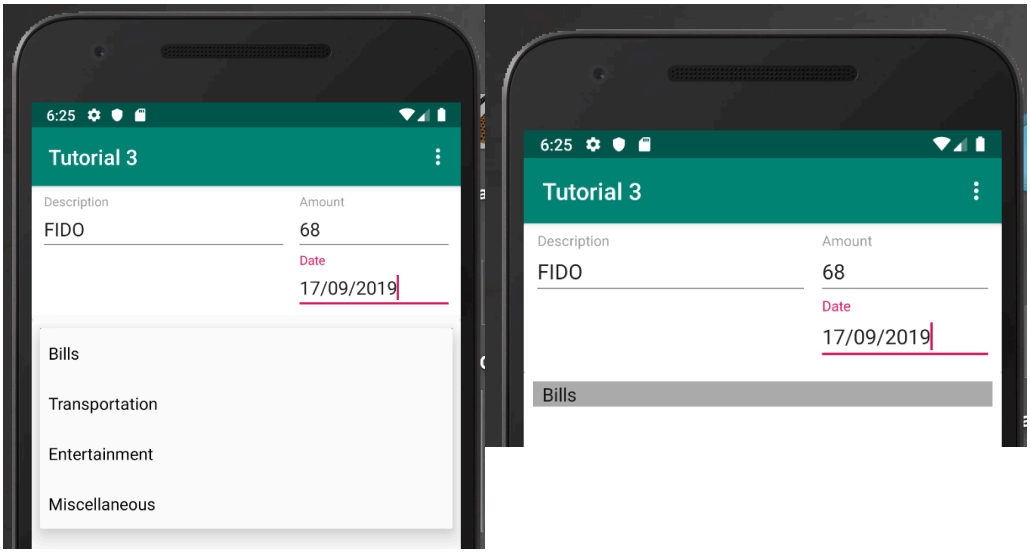
```
<string-array name="transaction_categories">
    <item>Bills</item>
    <item>Transportation</item>
    <item>Entertainment</item>
    <item>Miscellaneous</item>
</string-array>
```



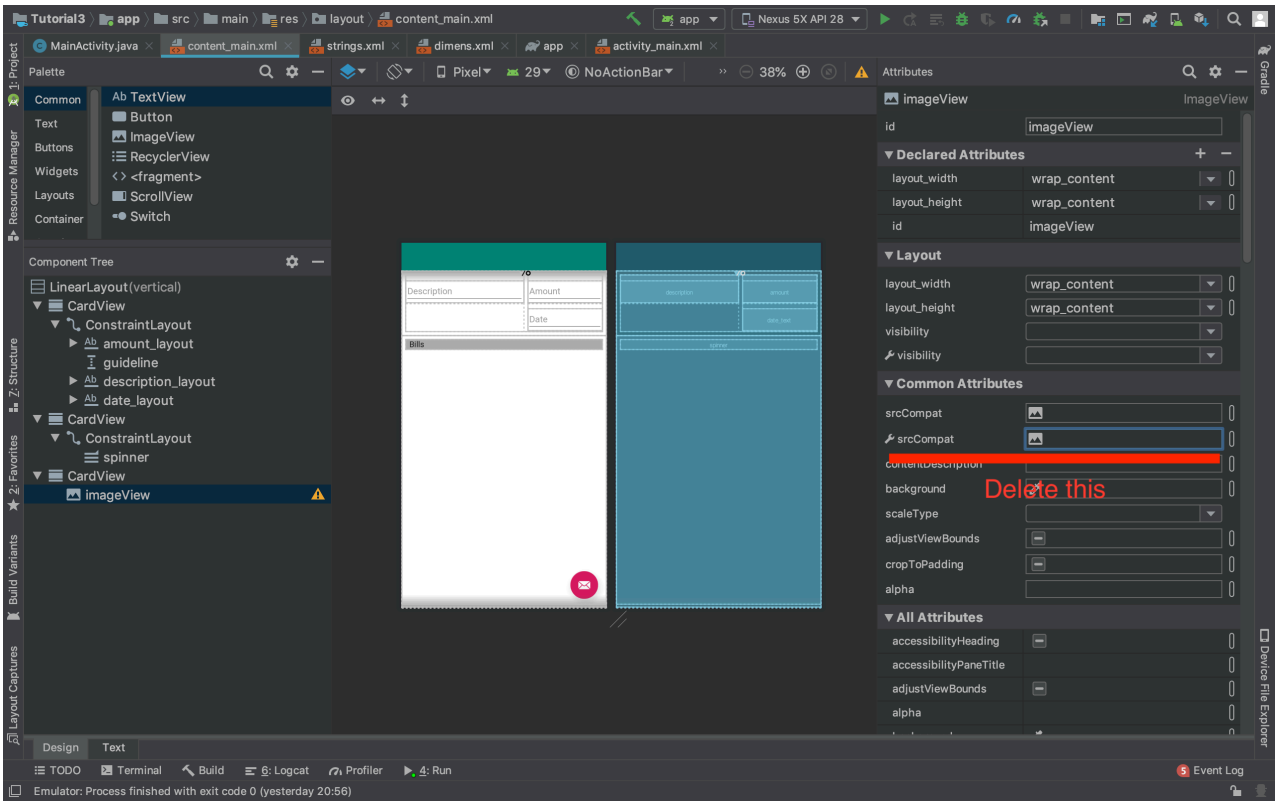
Now switch back to content_main.xml and select the spinner, and update the entries in the attributes panel with @array/transaction_categories. You can also find this by using the Resource Viewer (click the button, switch to Array, and find your transaction_categories array).



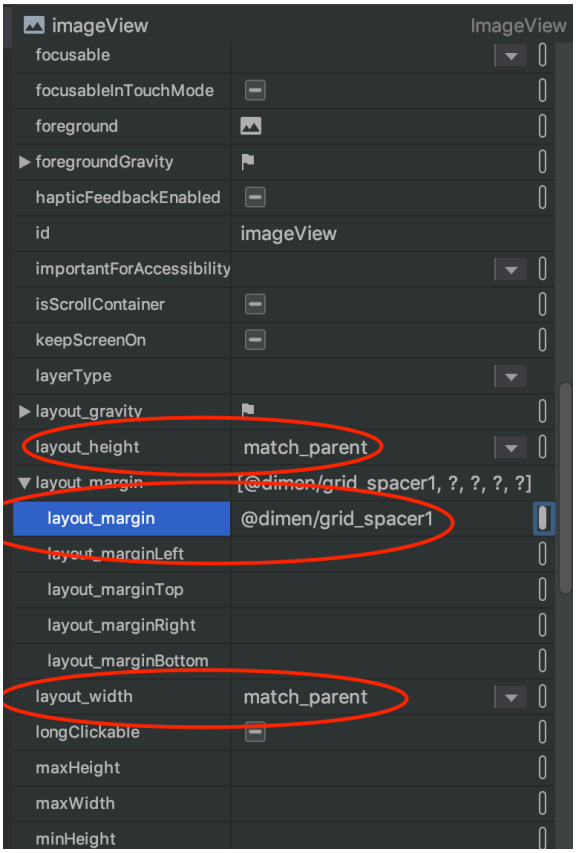
At this point, try changing the background color for the spinner to a light gray. Your UI should now look something like:



We will now add the final component of our UI, the preview attachments section. The idea of this component is that the user can take a picture of a receipt relating to the transaction they are logging with the app. The widget we will use is called an ImageView. ImageViews, as you might guess from the name, are used to display images in your application. Before we add it though, add a third CardView to your Component tree in the same way we did for the first two. After this, find the ImageView widget in your palette panel, and add it as a child of your third CardView. Change the ImageView ID to *image_view*. *Note*: You may be prompted to pick a resource for the ImageView. If so, pick any image that shows up, then delete this from the srcCompat attribute to make the ImageView empty.



At this point, you might not be able to see exactly what is happening with the ImageView. Change its layout_width and layout_height to match_parent, and add our previously created *grid_spacer1* resource to the *All* margin option in the attributes panel:

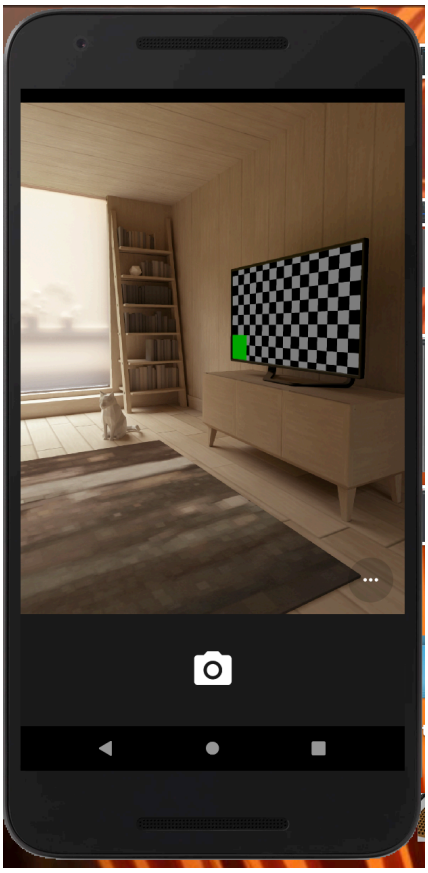


Change to text mode and edit a few attributes of the ImageView to match the following:

```
<ImageView
        android:id="@+id/imageView"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_alignParentTop="true"
        android:layout_centerHorizontal="true"
        android:background="#8c8c8c"
        android:scaleType="fitCenter"
        android:layout_margin="@dimen/grid_spacer1" />
```

Look up these attributes in the ImageView reference to understand what they are doing.

Now that we have our ImageView set up, the next step will be to switch to our MainActivity.java file to add functionality the FloatingActionButton. We will add the capability to open up the Camera app on the device, and take a picture. Note that if you're using the emulator, the Camera app is still available (hint: you can find it in all apps from the main Android desktop). Try it out if you like:



Note that if you are using your own device, you can use the camera in the same fashion, but for taking “real” pictures vs. the phoney ones provided by the emulator.

3. Adding Functionality

Now that we have at least one picture in our emulator gallery, it’s time to code the functionality for our activity button. In MainActivity.java, first check to see if you have all the necessary libraries that we will need:

MainActivity.java | activity_main.xml | content_main.xml | styles.xml

```
1   package com.example.tutorial3;
2
3   import android.content.Intent;
4   import android.graphics.Bitmap;
5   import android.os.Bundle;
6   import android.provider.MediaStore;
7   import android.view.Menu;
8   import android.view.MenuItem;
9   import android.view.View;
10  import android.widget.ImageView;
11  import androidx.appcompat.app.AppCompatActivity;
12  import androidx.appcompat.widget.Toolbar;
13  import com.google.android.material.floatingactionbutton.FloatingActionButton;
```

As usual, these can be added on the fly as you declare the items contained inside these. The IDE will prompt you to import these as you go. Now we'll move on to making a reference to our Image_view. Add the following 2 lines in the location shown:

```
15
16  public class MainActivity extends AppCompatActivity {
17
18      ImageView iv;
19
20      public static final int IMAGE_RESULT = 1;
21
22      @Override
23      protected void onCreate(Bundle savedInstanceState) {
24          super.onCreate(savedInstanceState);
25          setContentView(R.layout.activity_main);
26          Toolbar toolbar = findViewById(R.id.toolbar);
27          setSupportActionBar(toolbar);
28
29          iv = findViewById(R.id.image_view);
30
31          FloatingActionButton fab = findViewById(R.id.fab);
32          fab.setOnClickListener(new View.OnClickListener() {
33              @Override
```

Also notice on line 20, add this line as well as we will use this flag later on.

We have just created an ImageView object and in the method onCreate, instantiated as a reference to our *image_view* component.

Note the following next few lines of code for the FloatingActionButton *fab* (the pink button at the button at the bottom right corner of our UI). This code was automatically generated when we selected a Basic Activity (back at the start). It is similar to previous onClickListeners we have seen before.
Change the default code inside the onClick method to the following:

```
22      @Override
23      protected void onCreate(Bundle savedInstanceState) {
24          super.onCreate(savedInstanceState);
25          setContentView(R.layout.activity_main);
26          Toolbar toolbar = findViewById(R.id.toolbar);
27          setSupportActionBar(toolbar);
28
29          iv = findViewById(R.id.image_view);
30
31          FloatingActionButton fab = findViewById(R.id.fab);
32          fab.setOnClickListener(new View.OnClickListener() {
33              @Override
34              public void onClick(View view) {
35                  Intent cameraIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
36                  startActivityForResult(cameraIntent, IMAGE_RESULT);
37              }
38          });
39      }
40
41      @Override
```

Check out the Intent API on the Android Developer website to read up on what it does and how it works. In short, an Intent in Android is an “intention” to perform an actions. Used in conjunction with startActivity allows you to start a new Activity, such as another application on your device, or another part of your own application (which you will do later in the Homework part). In this case, the MediaStore class deals with all media on the device, including the camera. The flag *ACTION_IMAGE_CAPTURE* indicates that our intention is to start the built-in camera app to take a picture. The method startActivityForResult takes this intention, and actually starts the camera app. This version of startActivity (i.e., startActivityForResult) provides two-way communication between your current Activity and the one being called. This indicates that it will return some result; the standard version startActivity is effectively one-way, allowing you to start a new Activity, but not get any data back from it. Note that it is not as simple as a return value from most methods to get the image data back. To get the result back involves the next step below.

To get the image back from the Camera app Activity requires adding a new event handler. Upon returning from the other Activity, the system calls your app’s onActivityResult method. You must now implement this method. The system will return a new Intent back from the returning Activity, which includes a reference to the image data. Implement the onActivityResult method as seen below:

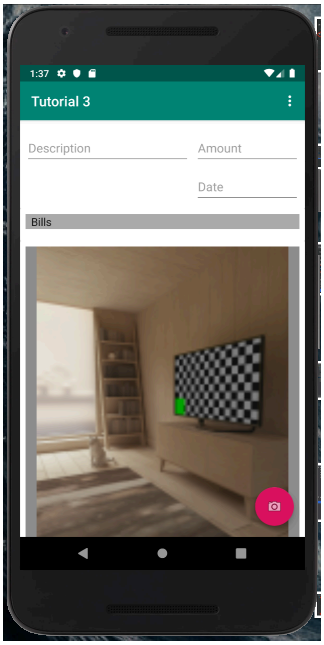
```
41      @Override
42      protected void onActivityResult(int reqCode, int resultCode, Intent data)
43      {
44          super.onActivityResult(reqCode,resultCode,data);
45
46          if(reqCode == IMAGE_RESULT && resultCode == RESULT_OK && data != null)
47          {
48              Bitmap bitmap = (Bitmap) data.getExtras().get("data");
49              iv.setImageBitmap(bitmap);
50          }
51
52      }
53
54      @Override
```

Note also that this is where the IMAGE_RESULT flag set earlier becomes useful. Recall that in the startActivityForResult call, we provided the IMAGE_RESULT flag as an argument. We again use the IMAGE_RESULT flag here for the argument reqCode. This is used in the onActivityResult method to identify who called the method, and what to do in that case. In an application using multiple Activities, you can use different flags like this to identify each other Activity launched and returned from, as necessary.
Note also that the method declares a Bitmap, which then gets the image data from the returning Intent. Finally, we set the ImageView to contain that bitmap data. After this, try running your app now and testing out the feature. You should be able to click on the FloatingActionButton on the bottom right. This should launch the camera, let you take a picture, and will set that picture to the ImageView in your UI.

Lastly, we will change the icon image for the action button. This can be easily done using one of the default image resources included in your project. Change to the activity_main.xml file in text mode, and scroll down to the definition of the *fab* FloatingActionButton component. Change the app:srcCompat line to:

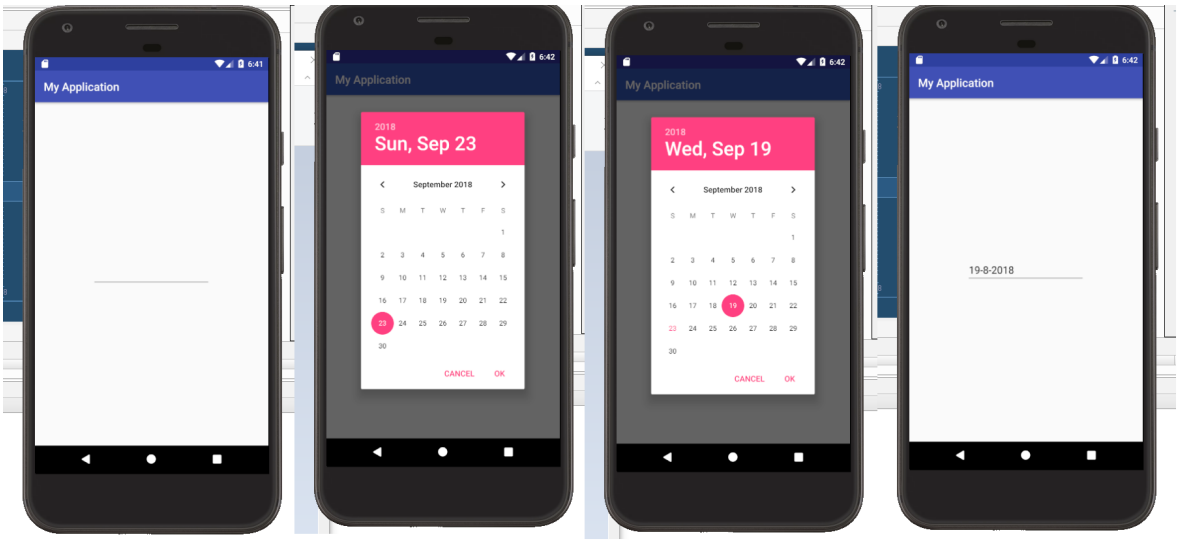
```
25      <com.google.android.material.floatingactionbutton.FloatingActionButton
26          android:id="@+id/fab"
27          android:layout_width="wrap_content"
28          android:layout_height="wrap_content"
29          android:layout_gravity="bottom|end"
30          android:layout_margin="16dp"
31          app:srcCompat="@android:drawable/ic_menu_camera" />
```

After doing this change back to design mode, and voila, your floating action button icon has been replaced with a camera icon:



Homework

1. Modify the *date* TextInputEditText View (i.e., the *child* of the TextInputLayout *date_layout*, but not the TextInputLayout itself) so that it opens a DatePickerDialog upon being clicked. Selecting the date via the DatePickerDialog should then update the text in the EditText View to the date selected by the user. The behaviour should look like this:



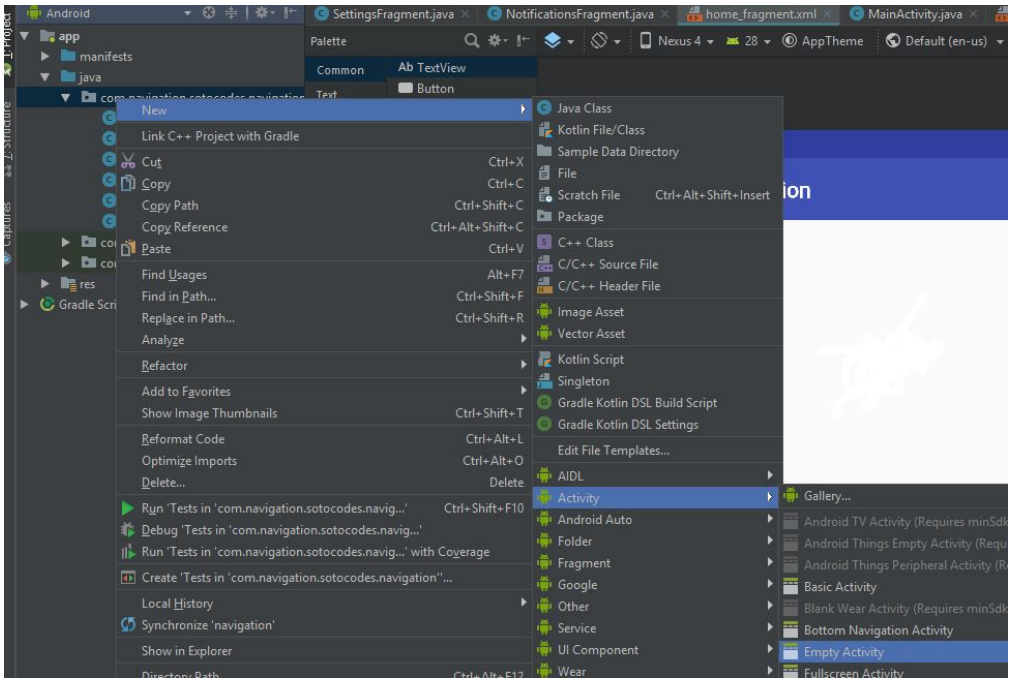
On clicking the EditText → Current Date Selected → Picking a new Date → EditText updated

Hint: Look up the EditText's focusable attribute if you cannot get it to behave properly when clicked (i.e., if it sets focus in the field or does not call the onClick method). See also the DatePickerDialog's OnDateSetListener. See also the Java Calendar class, paying special attention to the methods *getInstance* and *get*.

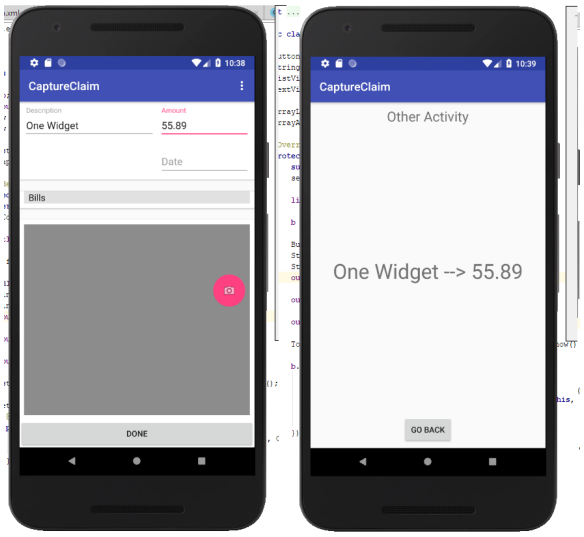
2. Create a new button at the end, change the text to "Done". Create a new blank activity in your java folder (check the image on the next page). This is done similarly to how you can add a new Java file (right click the package, go to new > Activity > Empty Activity). See the figure below for an example. Call the new Activity *NewActivity* (or whatever name makes sense to you). In CaptureClaimActivity.java add a new button event handler (onClick method) for your "Done" button.

   Pressing the "Done" Button should change to the other Activity, taking the user from CaptureClaimActivity to your ListActivity. This is somewhat similar to the approach used to launch the camera seen earlier, and involves setting up an Intent.

   Note that the newly generated activity also comes with its own activity_list.xml layout file! Modify the empty activity to add a TextView indicating you're in the ListActivity, as well as a button in the middle that says "Go Back". Pressing this "Go Back" Button should switch back to the MainActivity you created in this tutorial. *Hint*: This isn't as hard as it sounds, but is the first step toward multi-screen applications. Look up how to declare an Intent, and look up the startActivity method. Ask for help if you are stuck.



3. Take a look at different methods for transferring data between Activities. The following link has several solutions. Pay special attention to the use of Bundles, as it is likely the most straightforward method to complete the following. In your new empty activity, add a TextView. In your MainActivity, when you press the “Done” button, your activity should save the description and amount (you can expand it to include more elements as desired, but for our purposes, those two will be enough for this exercise). It should then start the new Activity (as in #2 above). It should pass over the values from the Description and Amount, then display these on the new empty Activity in the TextView. The final behaviour should be similar to the following:



Starting Activity →press “Done” → New Activity