Workshop Schedule

Day 1: Android Studio basics, basic app creation, interactivity

Day 2: Debugging, Intents, Layouts, Lists, Introduction to Firebase

Day 3: Firebase Integration, Android App Development Process, alternatives, expanding knowledge

Workshop resources

https://github.com/slimechips/AndroidIAP

All resources for this workshop can be found here

Notes:

The text in orange are meant for advanced users

1. Debugging

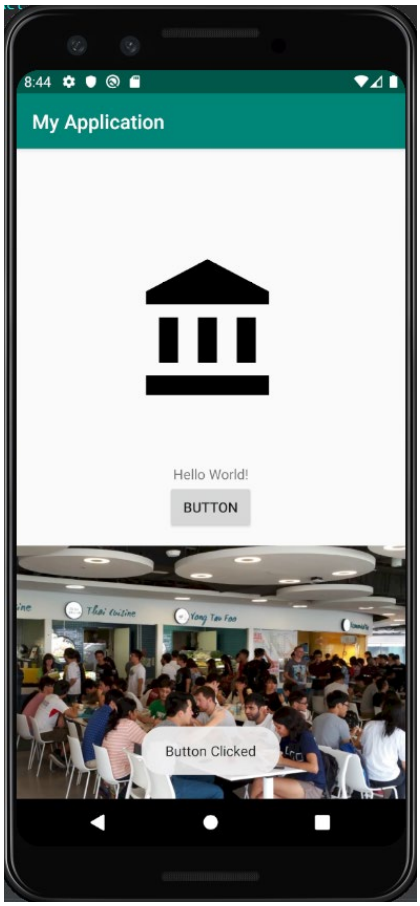
Debugging is the process of trying to identify and fix bugs that are causing your program to run slowly, incorrectly, or crash.

Often, the hardest part is trying to identify the bug. You may have a lot of code and do not know which part of the code is causing issues.

One way to resolve this is to add **indicators** to verbosely let us know when the app has successfully passed a certain part of the code (i.e. the state of the app).

Toast Debugging

One way is to use **Toast** to let us see on the app which code checkpoints we have gone past. For example, from last session's project, we used a Toast to check whether a button's **click listener** was indeed working.



Toasts have the advantage of simplicity and being easily visible, but comes with many other disadvantages (when used for debugging).

1. Toasts can only contain so much text. Often, if you're trying to print out some objects or variables in our programming code, it may be very long and won't fit into the Toast.

2. Toasts display sequentially, meaning a new Toast must wait for an old Toast to complete before it will be shown. This can be very troublesome if we have multiple Toasts.

Instead, we can use Log Debugging.

Log Debugging

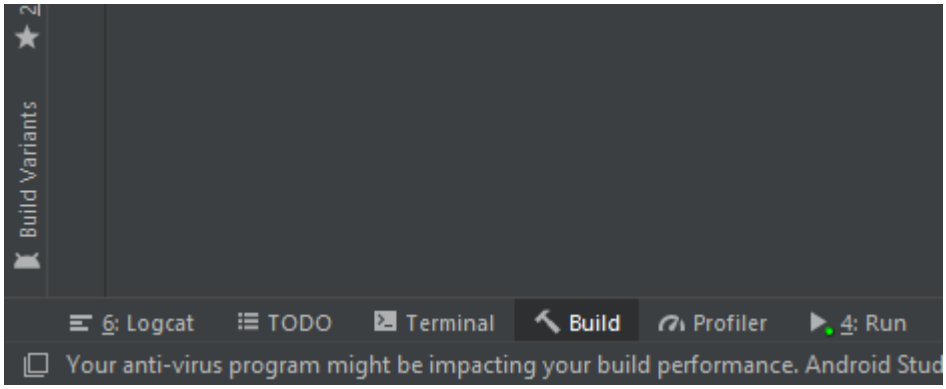
Log debugging allows you to

- Debug instantaneously
- View from the logcat
- Debug as much information as you need, at as many parts as you need

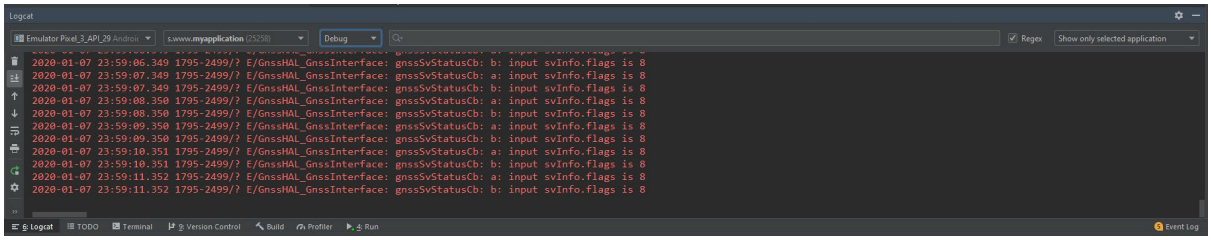
The disadvantage with log debugging from Toast debugging is you must look at the logcat to debug, whereas Toast debugging is simple, you just look at the app screen. Nevertheless, log debugging is way more useful.

Opening Logcat

You will find the Logcat at the bottom left of your screen. Click to open it.



You will start to see a lot of messages popping out (if you emulator is on/your phone is connected). Switch to **Debug** view to see only the messages that matter.



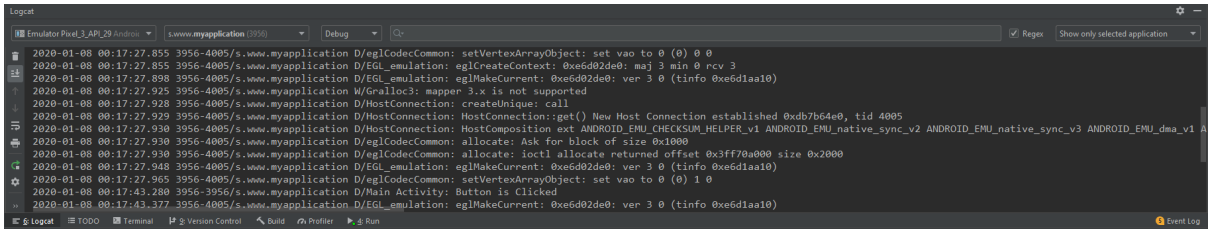
Right now we have not logged out any messages of our own yet, let's get down to it.

Logging messages onto Logcat

You may insert the following right underneath the Toast.makeText line.

```
Log.d("Main Activity", "Button is Clicked");
```

Try clicking on the button! What this does is the logger will log out the message **“Button is Clicked”** with a tag of **“Main Activity”**, every time the button is clicked.



Activity: Create a message that logs out when the activity starts. It should have a tag of “Main Activity” and a message of “Activity has started”.

Advanced Debugging

Instead of having to manually add a Log command every time you want to test something out (and end up forgetting to remove them later which makes your code and application very messy), we can instead attach a debugger to your android device.

Take a look back at your action panel. You will notice a bug icon (3 icons right of the play icon).

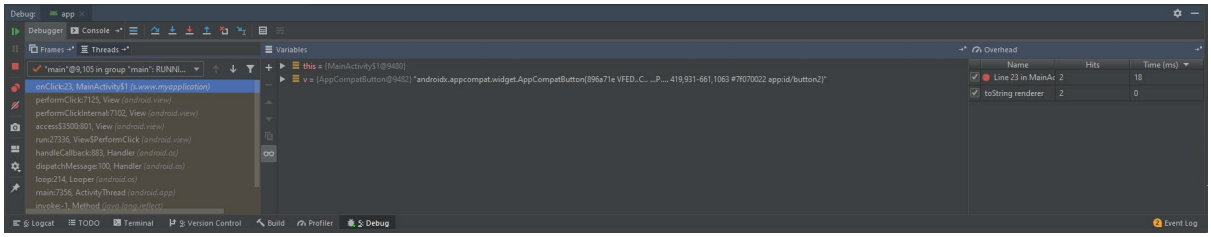
Click on the bug icon. You will launch the app in your emulator/phone as usual. But this time, you can add **breakpoints**.

Breakpoints are specific points in your code that you can add on the fly without having to change your code. When the app reaches those breakpoints, the breakpoint gets activated, the app is paused there, and you can inspect the status of the program and check on the values of variables etc.

To add a breakpoint to a part of the program (in the java file), simply click on the blank space to the right of the line number you want. It will show up as a red dot if registered successfully. **Add a breakpoint** to the onClickListener that we have.

```
19
20 o↑        myButton.setOnClickListener((v) → {    // The event to be executed
23 ●                Toast.makeText(getApplicationContext(),  text: "Button Click
24          });
26      }
27  }
```

Now try clicking on the button. Notice that the Toast does not show up, and the **debug menu** shows up in your Android Studio.



Click on the **green play button** on the top left of the menu, and you will realise that the app will now “resume” as per normal, and the Toast finally shows up.

We will now see how this mode of debugging is extremely convenient if we want to check variables. Let us add a counter in our program to check how many times the user has clicked the button. Add the following code **right before our onCreate function.**

```
int clickCount = 0;
```

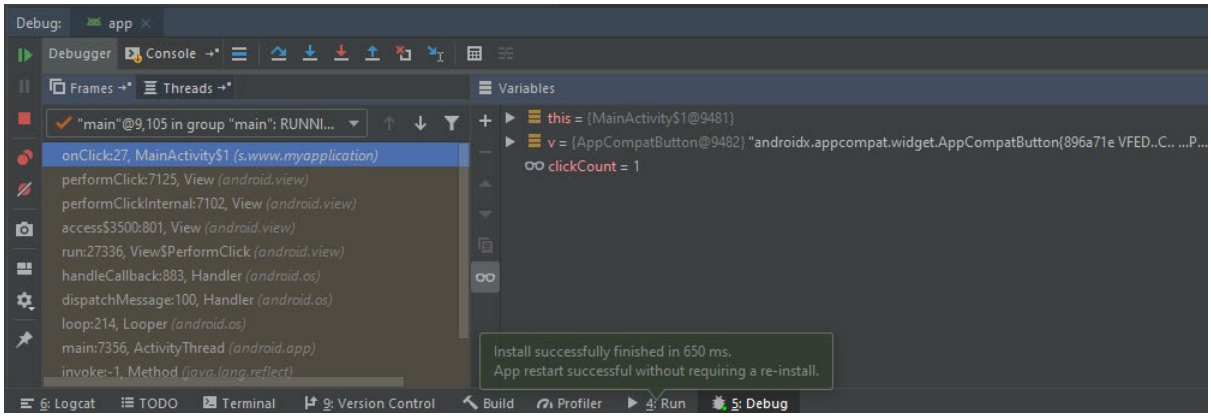
What this does is we have initialised a variable(integer) called *clickCount* to be 0.

Now, we want clickCount to increase by 1, every time we click the button. To do that, add the following code to the **onClick** function, before the Toast.

```
clickCount = clickCount + 1;
```

Now clickCount increments itself by 1 every time the button is clicked! But apart from directly showing the value of clickCount as text on the app itself, we can use the debugger to check that clickCount is working as intended.

Simply **add a breakpoint** on the line right after the code we just added. Then make sure the app is running on debug mode, and click on the button. Look at the debug menu.



You realise that we can actually see the variable clickCount and its associated value! Resume the app and try clicking the button again. You will see that the value of clickCount has increased from 1 to 2, and we can confirm that clickCount is indeed working as intended!

2. Intents (Creating a new Activity)

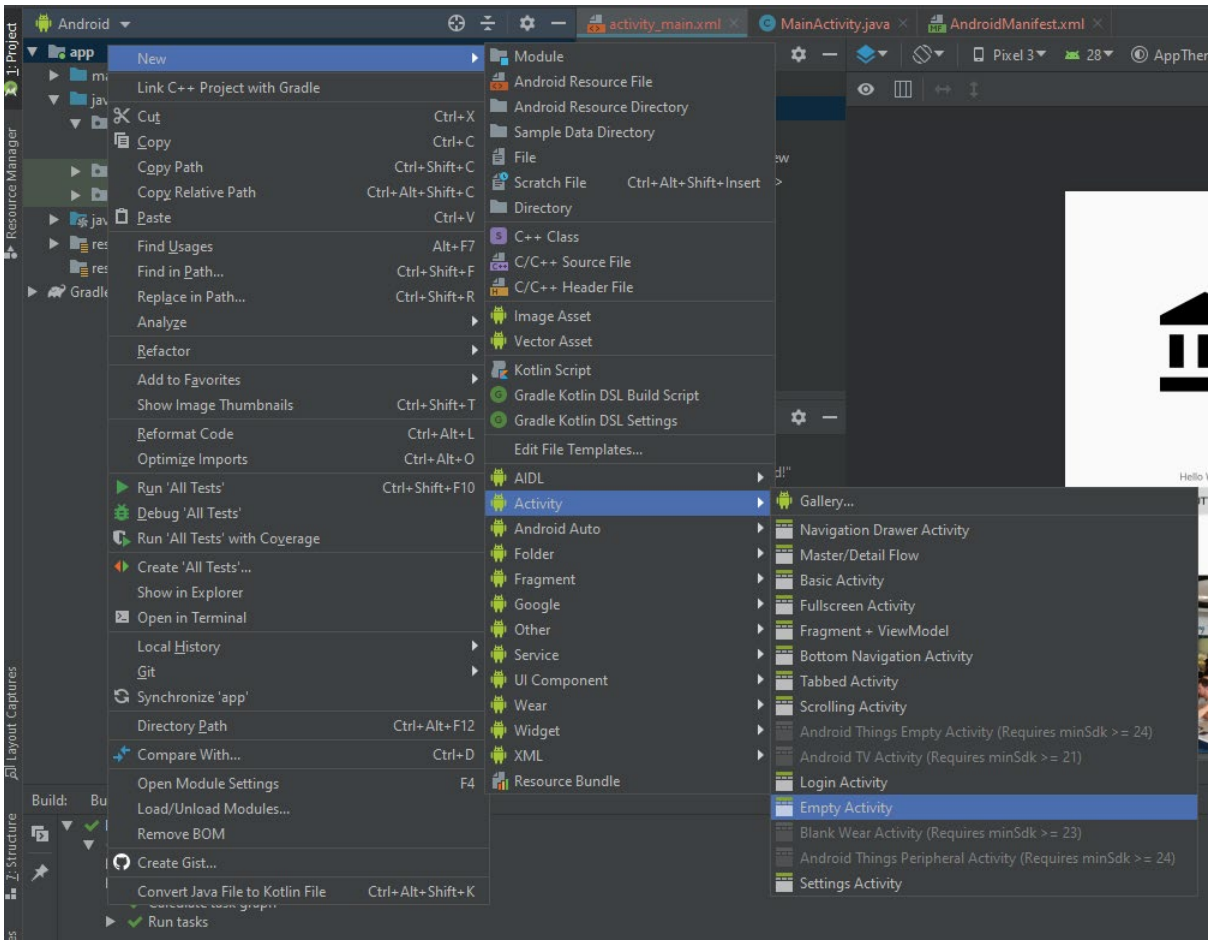
Now let us start work on our app. We want our app to be something like a ratings app, where users can rate things (e.g. food, categorised based on the canteen stall).

Let us create a new page for our app. This page will probably be a page with the different canteen stalls. A good way to introduce a new page is to create a new page from the button when it is clicked.

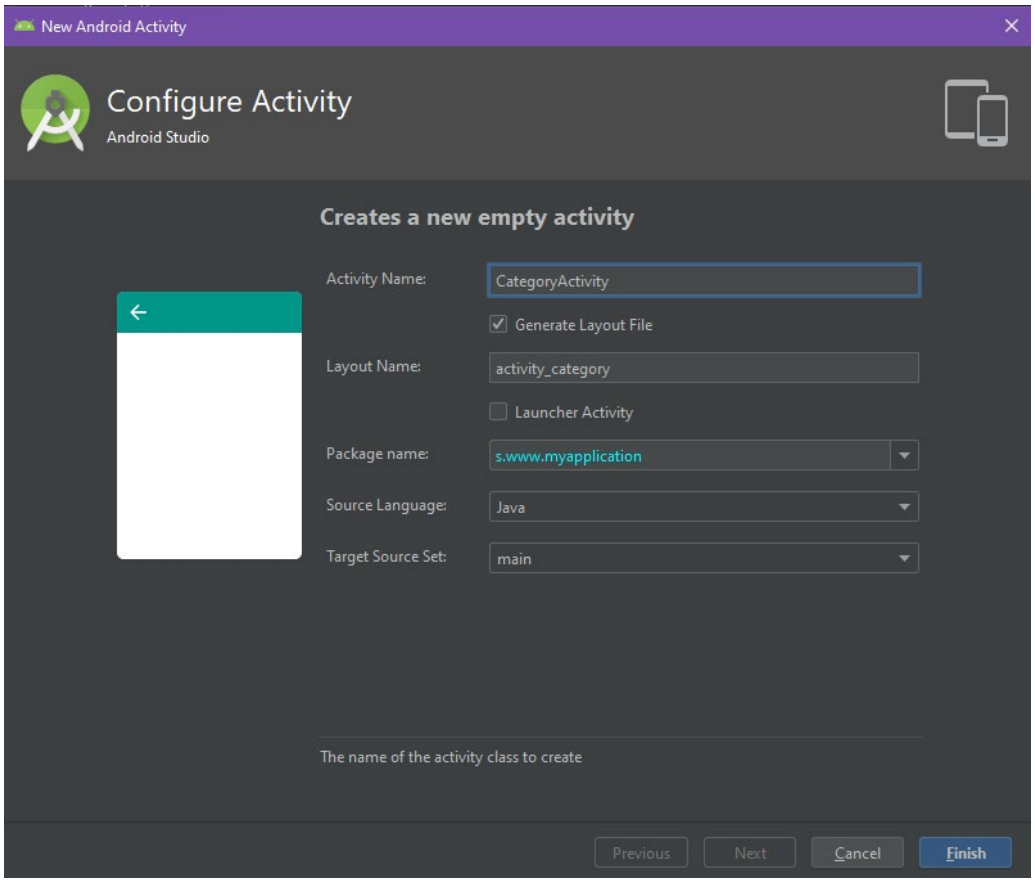
Activity: Change the button text to something like “Start Rating”.

Now, we will create a new page. To do that, we can create a new **Activity**. So let us create a new Activity. We can call this **CategoryActivity** (or any name really).

To do so, just right click on the **app** root folder to create a new empty activity.



As mentioned previously, I will call this **CategoryActivity** (but you can call it anything really).



Note that Android Studio has automatically created a new Java file and a new layout file for you to use (and the new Java file automatically displays the new layout file).

This is great, but the Activity currently just exists by itself, with no one to launch it. Recall that our **MainActivity** is specified to be the activity that launches when the app starts, but our **CategoryActivity** is not specified to be so. Therefore we need to find a way to launch **CategoryActivity** from **MainActivity**. In other words, we need to provide the **Intent** to launch **CategoryActivity**.

An **Intent** is simply a description of an operation that you want to perform, and is most commonly used as the glue to connect activities. Let us explore it a bit.

There are two main steps to an **Intent**.

1. Create the Intent
2. Execute the Intent

Creating a new Intent

To create a new intent, you simply need to create a new **Intent** object. Recall Day 1, where we created a **Button** object, which was a reference to a button in the XML file.

We will create the intent using a somewhat similar method. Here is the code (you should place this code after your Toast.makeText):

```
Intent intent = new Intent(MainActivity.this,
CategoryActivity.class);
```

Let's analyse this line of code. First off, we create an **Intent** object, using the **new** keyword. This new Intent requires two parameters that we will need to provide. The first parameter is the **instance of the current activity**. In this case, the current activity is given by **MainActivity.this**. The second parameter is the **type of the new Activity we want to create**. This is given by **CategoryActivity.class**. We give this new intent that we created the name of *intent*.

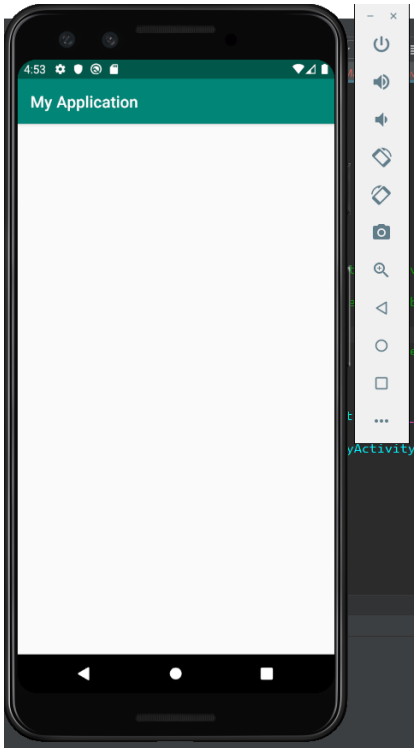
Execute the Intent

We just now have to execute the intent. This is very simple.

```
startActivity(intent);
```

All we simply did was to use a function called **startActivity()**, and use **the intent that we created previously** as a parameter.

Now, try clicking your button and test it out! After clicking the button, you should get a blank page (which is your **activity_category.xml**).



3. Useful Layouts and Views (LinearLayout, ScrollView, CardView)

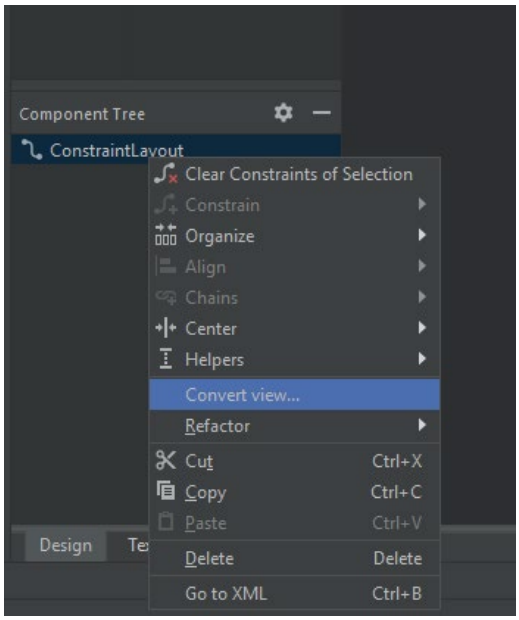
We want to create a sort of list of items, where we can click the items and bring us to another page. You can customise your case, but for my case, I will be creating a list of canteen stalls. Now there are a few steps to be done.

1. Make a scrollable layout to accommodate for multiple items.
2. Create a LinearLayout to contain the items
3. Each item should contain a picture and some text.
4. Each item should be clickable (you can do just one/two for learning purposes) and bring us to another page/activity.

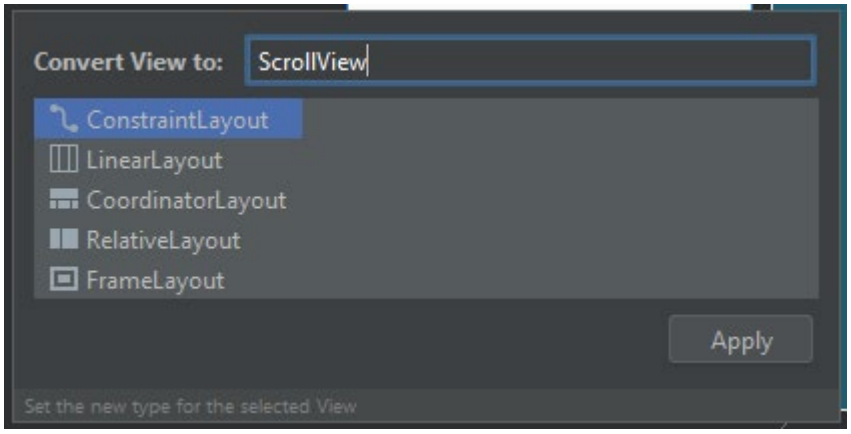
Let's get down to it.

a. Creating a ScrollView

Go to the new layout file that you created for your new activity. Go to the component tree, right click on the ConstraintLayout, and select ***Convert View***.



You will now see a popup to select which view to convert to. Type in **ScrollView**.

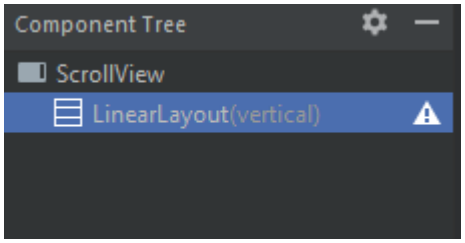


A **ScrollView** simply enables scrolling capability of the **layout that is embedded within it**. Anyway, your Component tree should now just have a ScrollView.

b. Create a LinearLayout to contain items

A **LinearLayout** is simply a layout that arranges other views either horizontally in a single column or vertically in a single row.

Simply go to the **Palette**, go to **Layouts** and drag in **LinearLayout(vertical)**. You should now see something like that in the Component Tree.



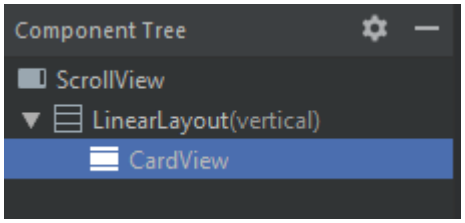
We have now created our LinearLayout!

c. Creating items to display

While you could simply put pictures and texts a individual objects in the LinearLayout, it may be smarter to group each related text and picture together, put them within a **CardView.**

A CardView makes each item have a rounded rectangle, with some sort of shadow at the back, making it look like … a card.

Simply drag a **CardView** (Found under **Containers**) into the LinearLayout. You should now see something like this:



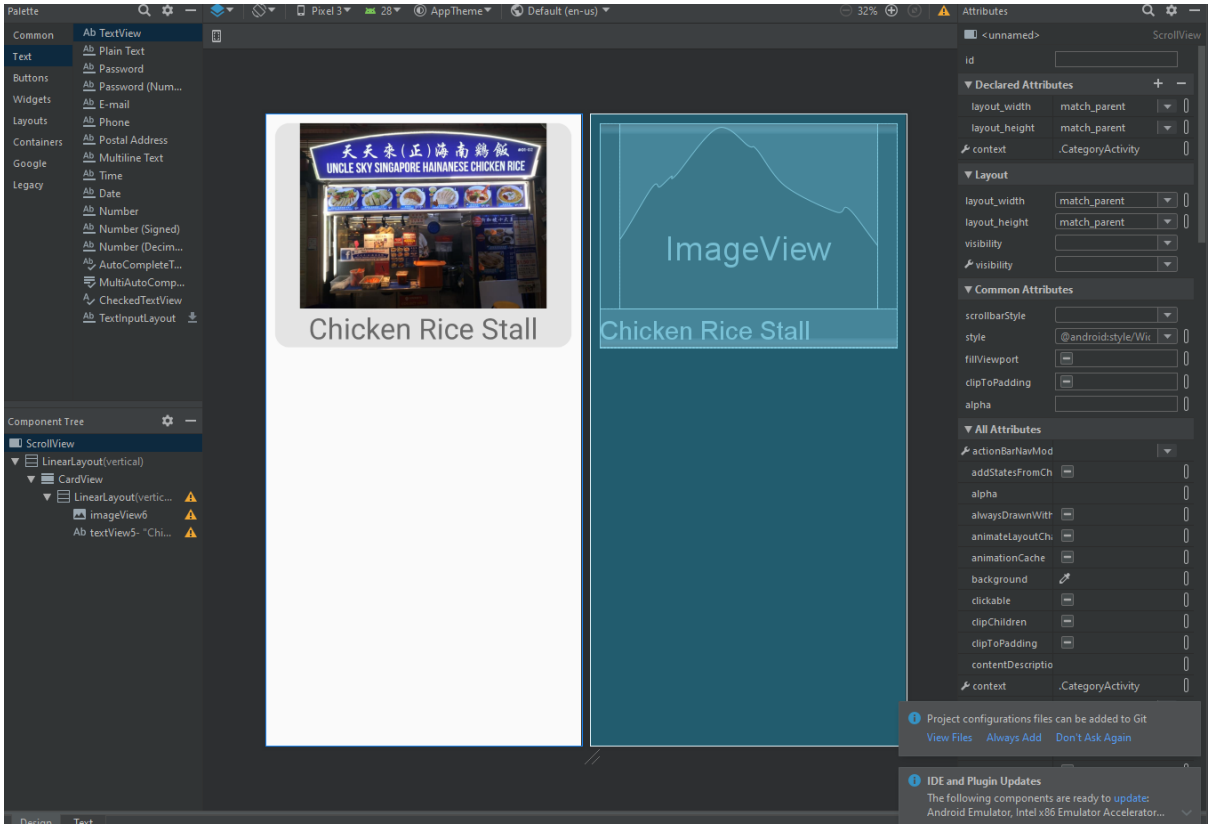
However, a CardView by itself does not have a set layout. We want our picture to be above our text within the CardView. What other better layout to use than LinearLayout? Drag a LinearLayout into the CardView. Also, set the CardView's **layout_height** to be "wrap content".

Once that is done, we want to look for a picture that we wish to put into our CardView. For me, I am just going to put a picture of a Hainanese Chicken Rice stall. You can put whatever picture you want. Recall the steps of importing an Image into your project, and placing the Imageview inside the CardView.

Next, you will want to put a **TextView** below the **ImageView**. Again, the text is up to you. I am just going to put "Chicken Rice Stall". I am also going to increase the font size to 34sp, and centralise the text by setting **gravity** to "center". I will also centralise my picture by setting its **layout_gravity** to "center". For my CardView, I will set my **cardCornerRadius** to 18dp and **cardElevation** to 4dp. I will also set the CardView's **cardBackground** to a very light gray.

I also want the CardViews to have a margin from the page so it looks less squeezed. So I set the **layout_margin** of my CardView to 12dp.

The end result should look something like this:

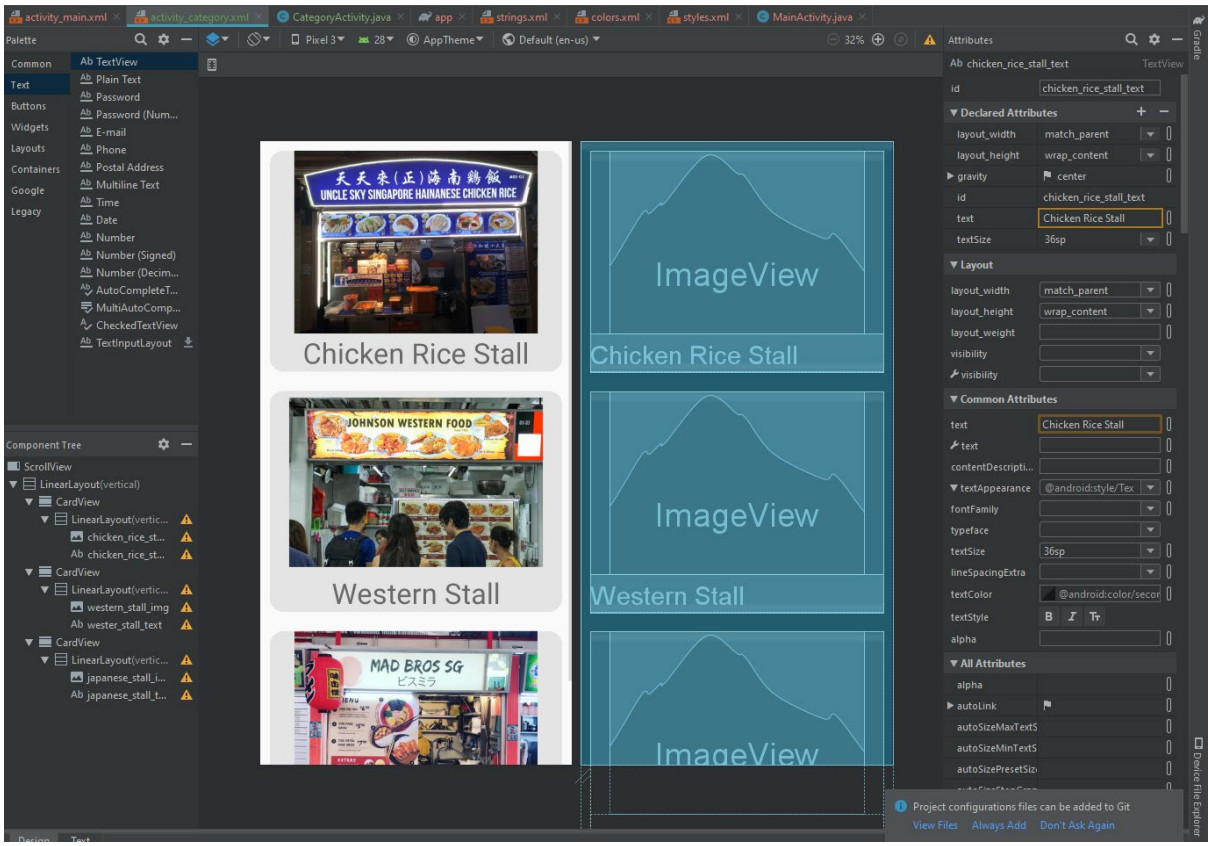


Now, all I have to do is copy and paste a few more CardViews for more stores! Again I import more pictures and change the image source of the ImageViews, and the text of the TextViews accordingly.

However, we have a problem. When we performed the copy and paste, the ids of the TextViews and ImageViews stayed the same! As a result, we have multiple objects with the same Id, which is not allowed. We have to change the ids of each of these objects manually. (Change them to something memonic!)

NOTE: I encountered a bug while trying to change the ids of the ImageViews and TextViews where all the ids changed together. A workaround is to simply change the Ids from the XML code itself.

The end result should look something like this:



(Bonus) Creating a ScrollView using the XML Method

As some of you might know, using the XML method to edit our layout file is the more ideal way to do things. So, I will show you the XML method to edit the layout file. If you need help understanding how to edit the XML file, please refer to the Day 1 notes, where there is a section dedicated to explaining this.

Undo your changes from before. Now, change the line

```
<androidx.constraintlayout.widget.ConstraintLayout
```

to just LinearLayout. Ensure the closing XML tag is changed as well. It should now look something like that:

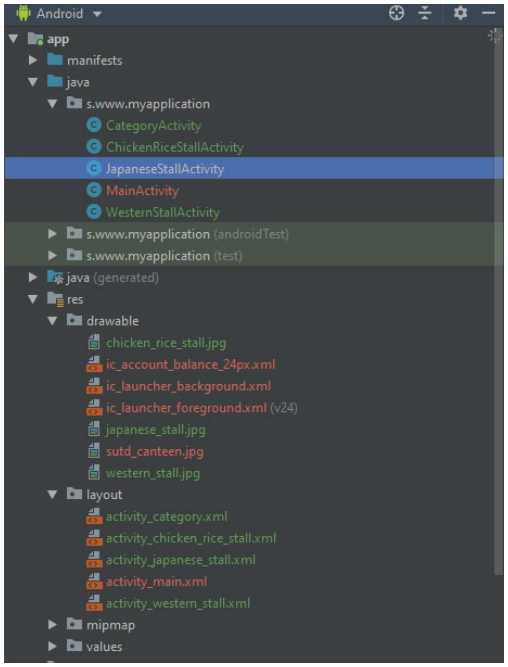
```
<?xml version="1.0" encoding="utf-8"?>
<ScrollView
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"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

</ScrollView>
```

You can switch back to design view to check that your change was indeed applied correctly!

d. Making the items clickable

Before you do the following steps, you might want to create a new empty activity for every CardView that you had. For example, I will create 3 new activities, named appropriately.



First, we will need to assign Ids to the CardViews. Go ahead and give them some logical ids, and take note of them.

Now open your new Activity's java file. For me, that would be **CategoryActivity.java**. Now, we need to repeat the steps that we did in day 1, to make the CardView do something when clicked. Though in this case, we will be getting a reference to the CardView instead of Button object. Let's take a look at the steps again.

1. Create the CardViews to be clicked (Done)
2. Get references to the CardViews in the XML file, from the Java file
3. Set onClickListeners on the CardViews (onClickListeners can be placed on objects other than buttons!)
4. Implement the action to be executed (Create a new Intent/Activity).

I recommend doing it for just one CardView first. You can implement the rest later on if you have more time.

The end result in your Java file will look something like this:

```java
package s.www.myapplication;

import androidx.appcompat.app.AppCompatActivity;
import androidx.cardview.widget.CardView;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class CategoryActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_category);

        CardView chickenRiceStallCard = findViewById(R.id.chicken_rice_stall_card);
        CardView westernStallCard = findViewById(R.id.western_stall_card);
        CardView japaneseStallCard = findViewById(R.id.japanese_stall_card);

        chickenRiceStallCard.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(CategoryActivity.this, ChickenRiceStallActivity.class);
                startActivity(intent);  // Go to ChickenRiceStallActivity when button clicked
            }
        });

        westernStallCard.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(CategoryActivity.this, WesternStallActivity.class);
                startActivity(intent);  // Go to WesternStallActivity when button clicked
            }
        });

        japaneseStallCard.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                Intent intent = new Intent(CategoryActivity.this, JapaneseStallActivity.class);
                startActivity(intent);  // Go to JapaneseStallActivity when button clicked
            }
        });
    }
}
```

Test out your app! You should find that every card you click will bring you to a new page!

4. Useful Layouts (TableLayout)

Let us try to create some content for the new activities that we have created. Something we could do is do some sort of a rating system. In this case, since I have a food stall (chicken rice stall), I could do ratings for each of the food items sold by the chicken rice stall! (e.g. White Chicken Rice, Roasted Chicken Rice, Duck Rice).

What I want to show is a table like the following:

Food item	**Good Votes**	**Vote Button**
<pic of white chicken rice>	5	<Button to vote good>
<pic of duck rice>	9	<Button to vote good>

Eventually, we want to integrate this rating system with a online database, but for now, we will hardcode the existing ratings of the food.

The steps we need to take are the following

1. Create a ScrollView for our TableLayout
2. Create a TableLayout with pictures, text, buttons
3. Set up references to Rating TextViews and voting buttons.
4. Perform mathematical operations to set the ratings

a. Create ScrollView

As usual, convert the ConstraintLayout in your new activity, to a ScrollView.

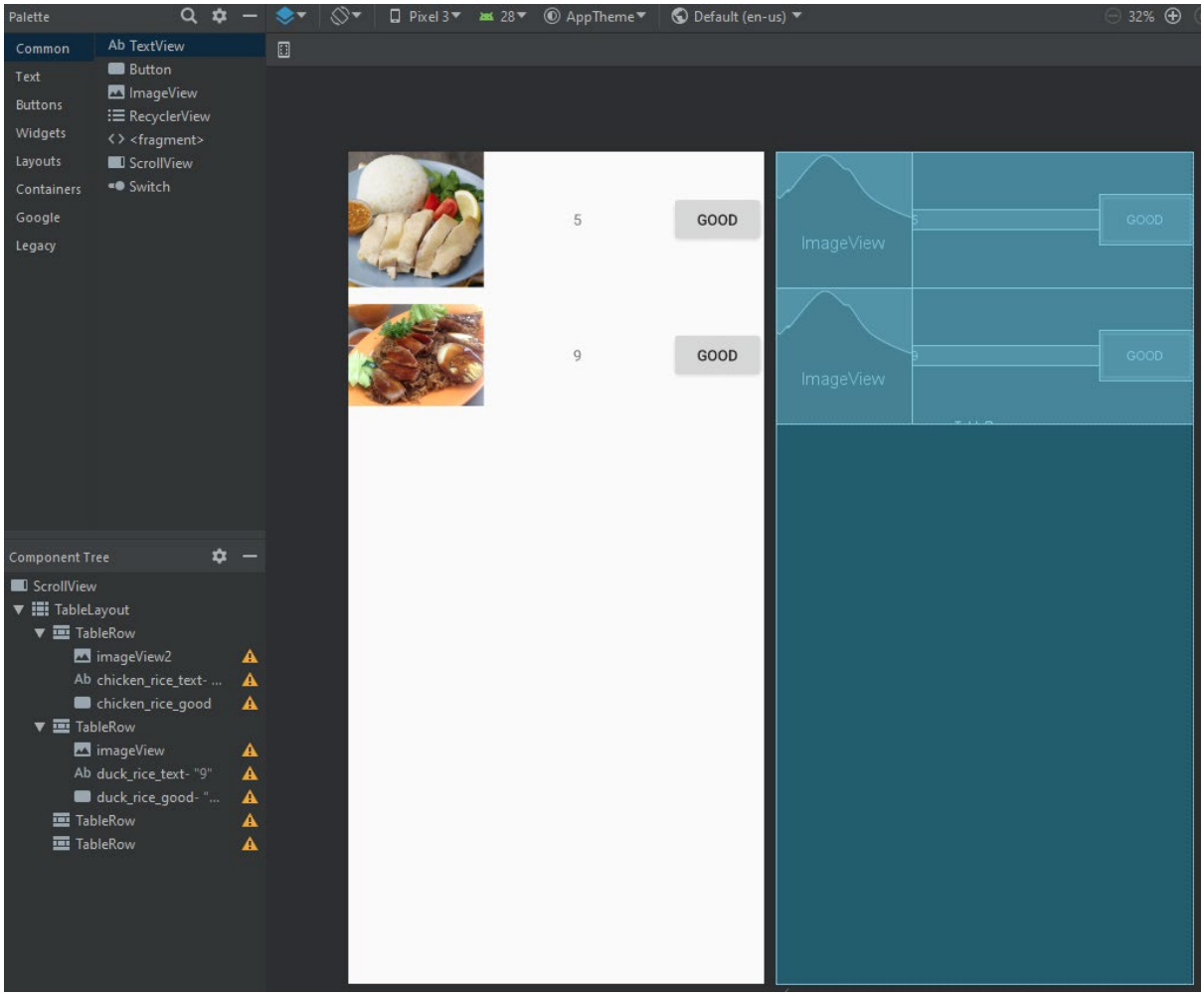
b. Create TableLayout

Drag in a **TableLayout** from the palette, under **Layouts**. Notice the TableLayout comes with 4 TableRows. Place all the items you need in the row (ImageView, TextView, Button).

You can customise the TableRow accordingly to your liking. I recommend

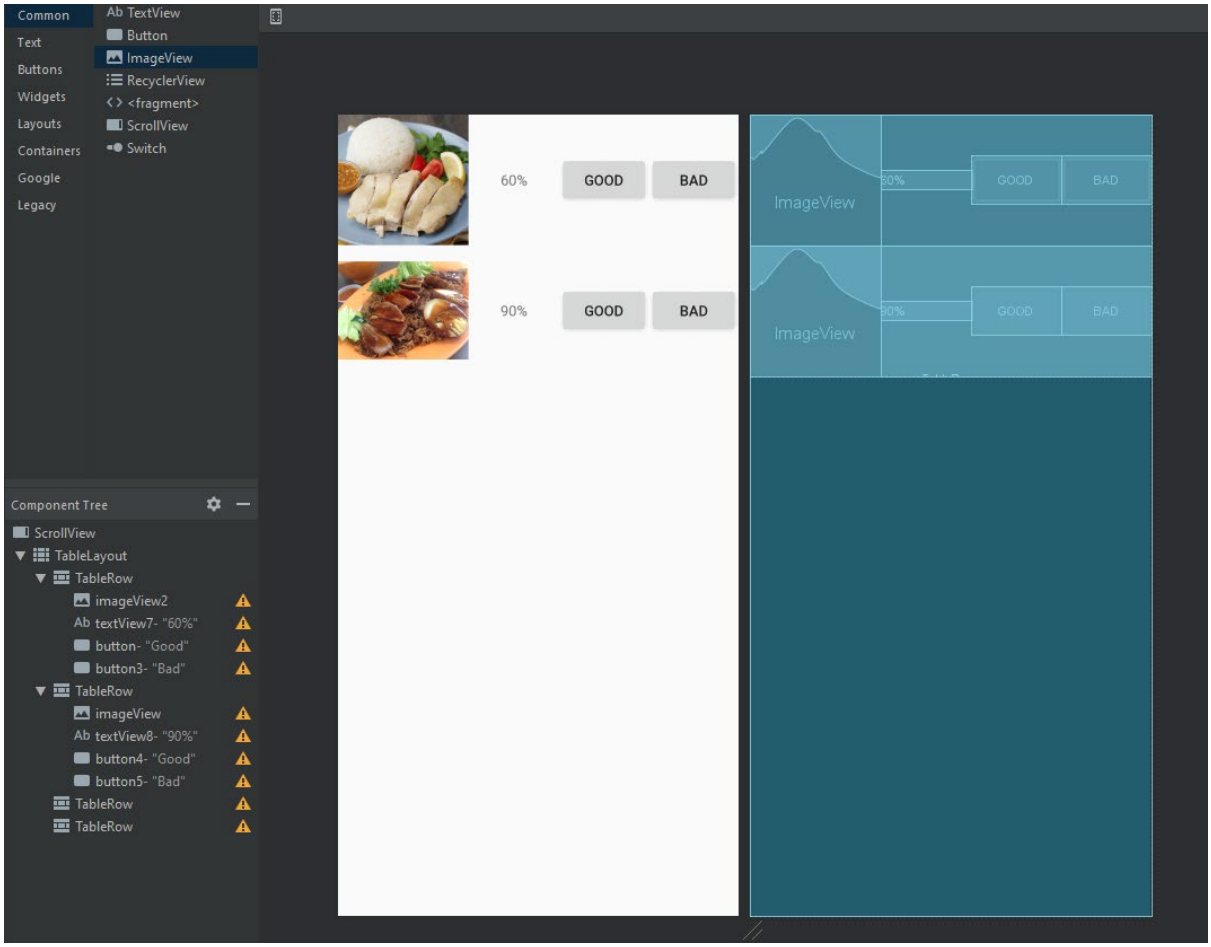
- Set the ImageView **layout_width** and **layout_height** to 128dp.
- Set the **gravity** of the TableRow to **center_vertical**.
- Set the **layout_weight** of the TextViews to 1. This allows the TextViews to stretch to fill the available width of the table.

The end result should look something like this:



(Optional: Bad Votes, Percentage Rating)

If you find yourself with more than enough time, you can add a button for bad votes, and the text displayed will be the percentage of people who voted good for the item. See below for an example:



c. Set up References to TextViews and Buttons

Assign appropriate ids to your TextViews and Buttons. Then, go to the Java file of this activity, and perform the findViewByIds. Remember, this time we have **TextViews** objects to get reference to. **This time, declare the TextViews and Button variables you want to use before the onCreate function**. The code will end up looking something like this:

```
package s.www.myapplication;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;

public class ChickenRiceStallActivity extends AppCompatActivity {

    TextView chickenRiceText;
    TextView duckRiceText;
```

```
    Button chickenRiceGoodBtn;
    Button duckRiceGoodBtn;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_chicken_rice_stall);

        chickenRiceText = findViewById(R.id.chicken_rice_text);
        chickenRiceGoodBtn = findViewById(R.id.chicken_rice_good);

        duckRiceText = findViewById(R.id.duck_rice_text);
        duckRiceGoodBtn = findViewById(R.id.duck_rice_good);
    }
}
```

d. Interactivity

We will first hardcode the initial number of votes and calculate the initial rating. Then we will update the rating whenever the buttons are pressed.

First, initialise the votes before the onCreate function.

```
int chickenRiceVoteGood = 5;
int duckRiceVoteGood = 9;
```

Now, you want to set the initial text of the TextView, right after setting the references. Use the **setText(Text)** function, together with an **Integer.toString(Text)** conversion function, which changes your variable from an integer to a text that can be displayed.

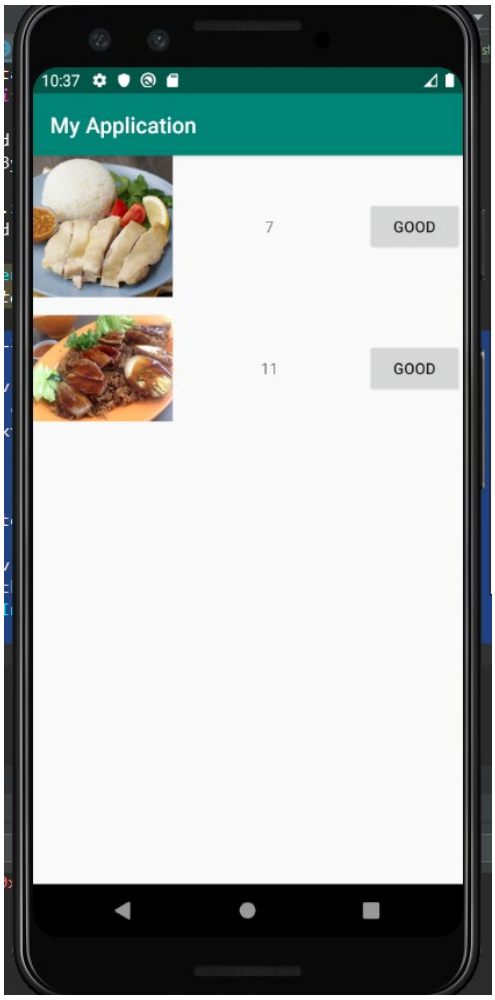
```
chickenRiceText.setText(Integer.toString(chickenRiceVoteGood));
duckRiceText.setText(Integer.toString(duckRiceVoteGood));
```

Now, you want to set the onClickListeners. The Listeners will increment the good vote count by 1, then set the text again.

```
chickenRiceGoodBtn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        chickenRiceVoteGood = chickenRiceVoteGood + 1;
        chickenRiceText.setText(Integer.toString(chickenRiceVoteGood));
    }
});

duckRiceGoodBtn.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        duckRiceVoteGood = duckRiceVoteGood + 1;
        duckRiceText.setText(Integer.toString(duckRiceVoteGood));
    }
});
```

Now try out, the app! The votes should increment every time you click the button.



(Optional: Bad Votes, Percentage Rating)

Again, you can implement the logic for the rating percentage in your java code, if you have enough time.

5. Firebase

You might have realised that so far, we have created a relatively functional app. However, there are many problems with this app that make it impractical in real life. For example:

1. There is no login system
2. There is no internet database. The votes you have for each item are purely local, and get reset every time you reset your app.

One way to resolve the issue of values getting reset everytime is to use **SharedPreferences**, which save your session. However, this does not solve the issue of the votes being only local.

Firebase is a useful platform that can solve both these issues. With Firebase, setting up a database is easy, and it also simplifies your login and authentication system. It even has a Machine Learning Kit that you can use.

Firebase is generally regarded to be a very simplified solution to things that a mobile app needs. In reality, you app requires a dedicated backend server to handle all internet requests, but for the purpose of many proof-of-concepts and small projects, Firebase is sufficient. Integrating Firebase into your App is also very easy.

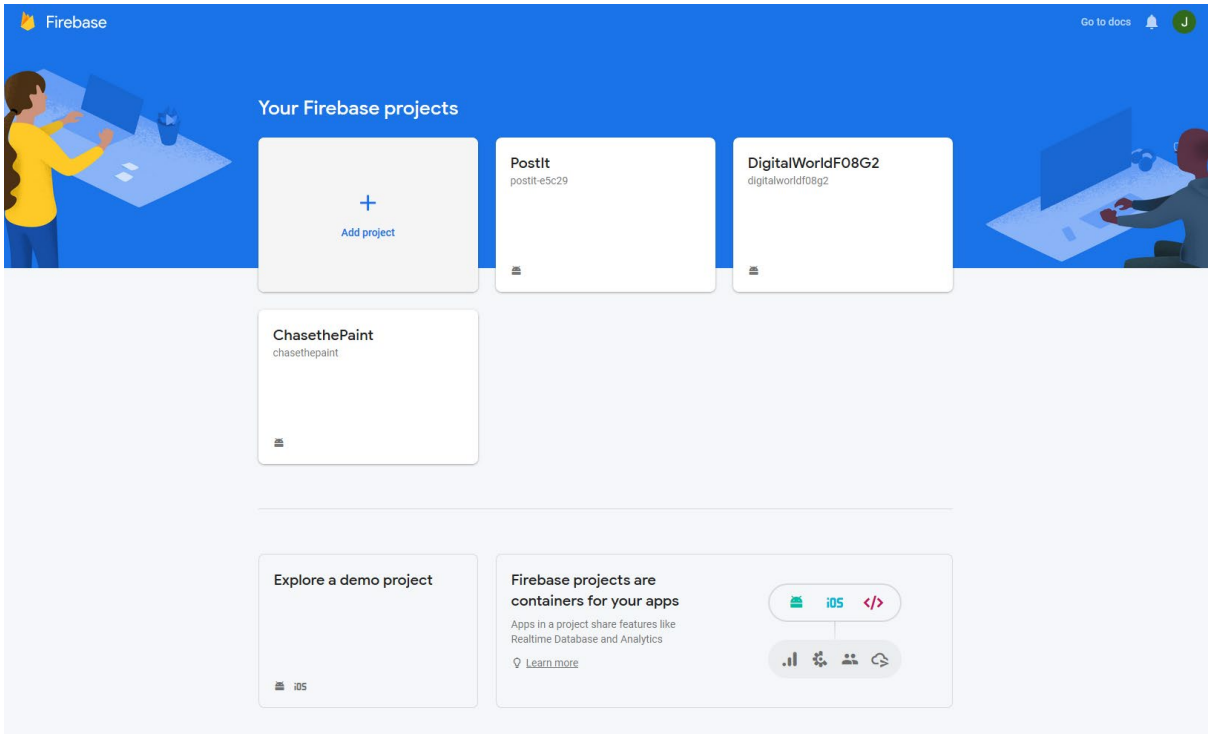
For today, we will just focus on setting up a Firebase account.

Set up your Firebase Account

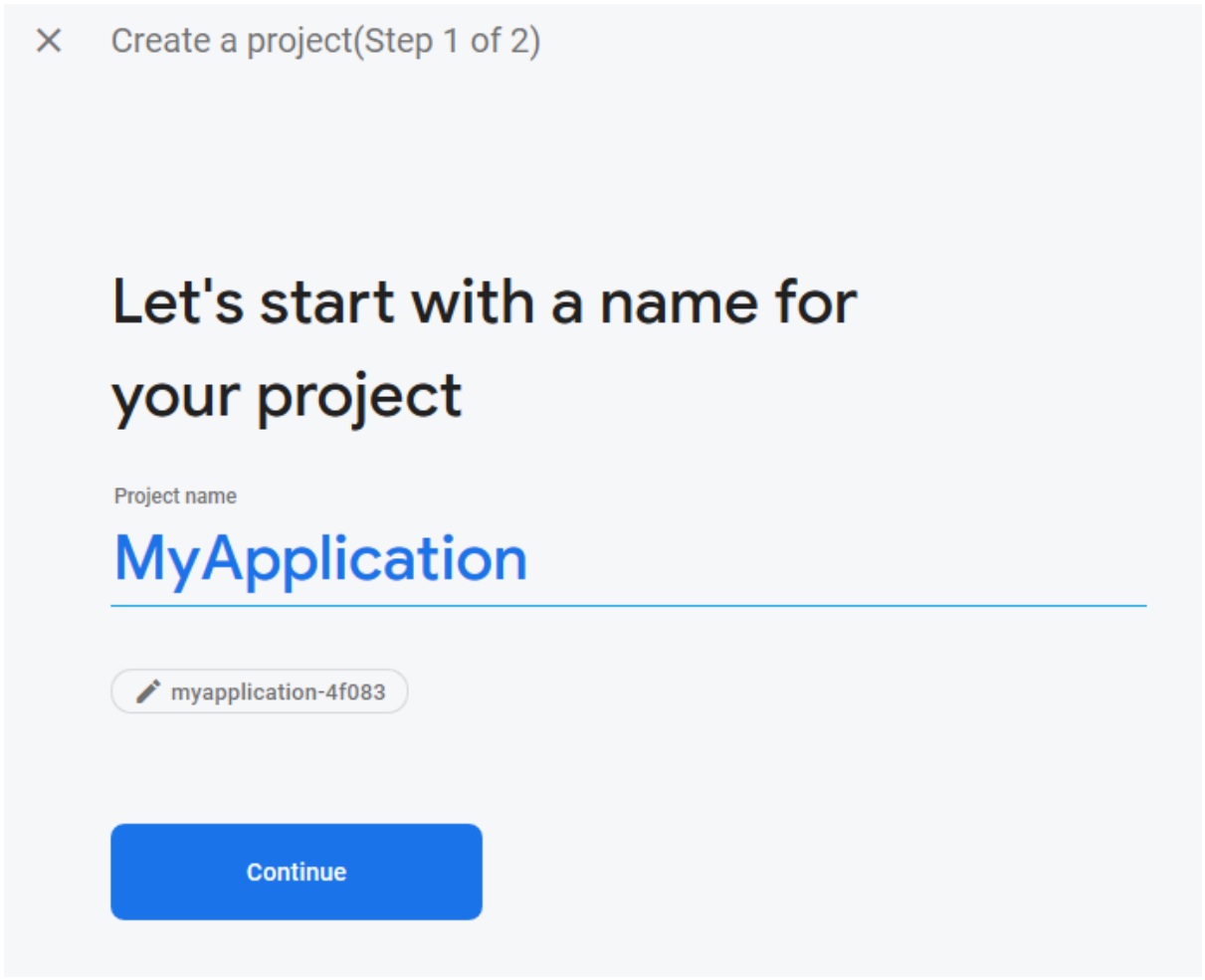
Go to https://firebase.google.com, and sign in with a google account.

Create a Firebase Project

Go to the firebase console at https://console.firebase.google.com/.

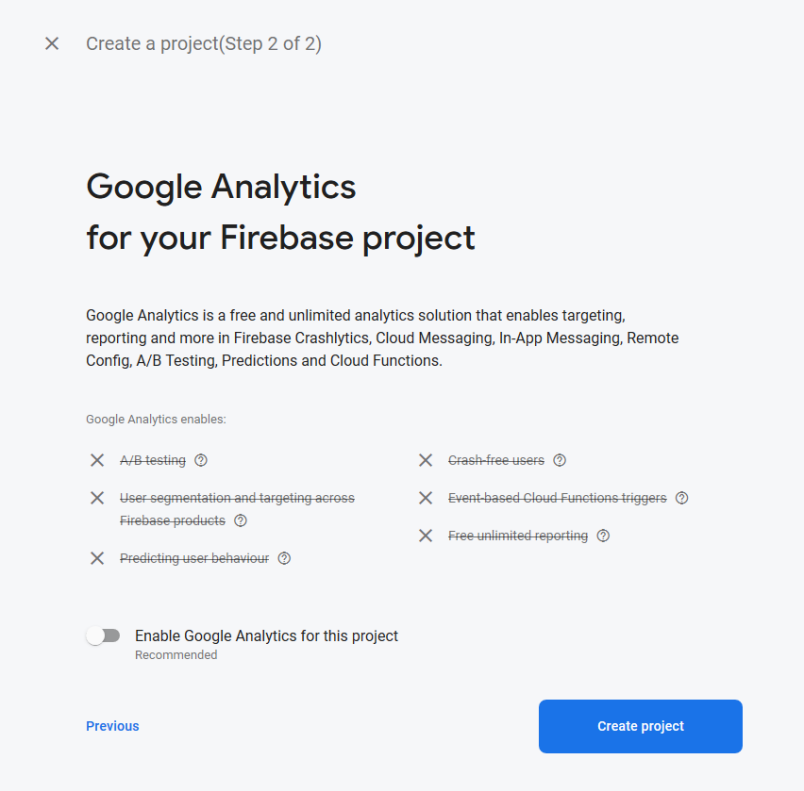


Click on the big **Add Project** button.

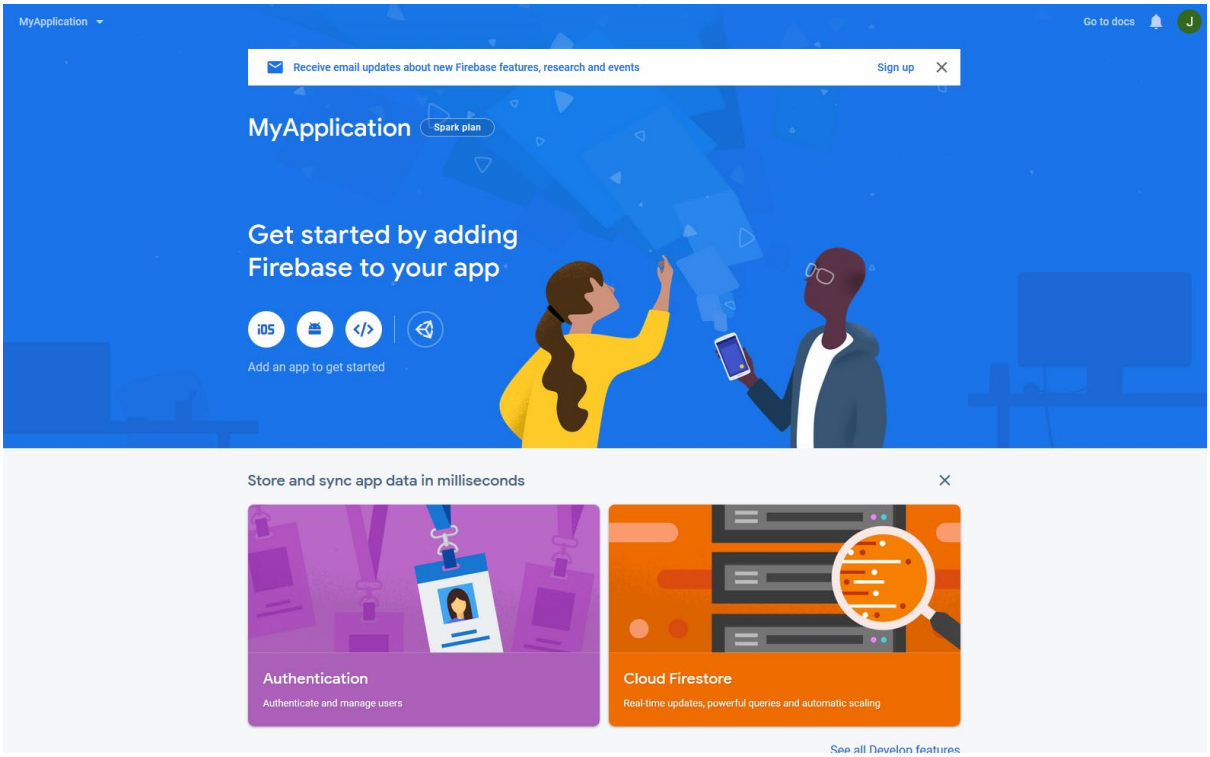


Enter your Application name.

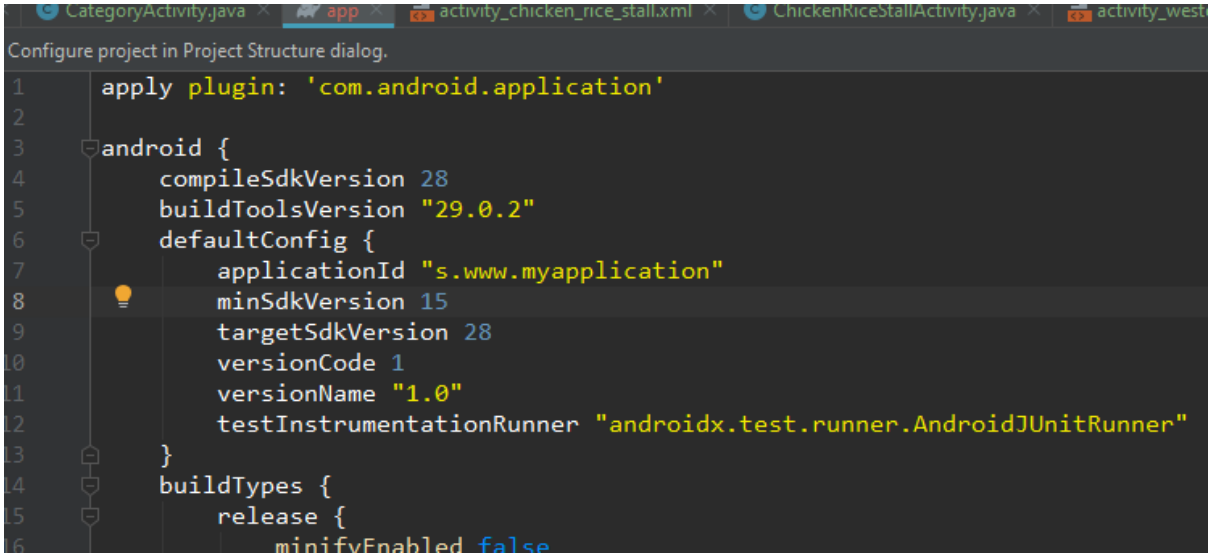
Disable Google Analytics and create your project.



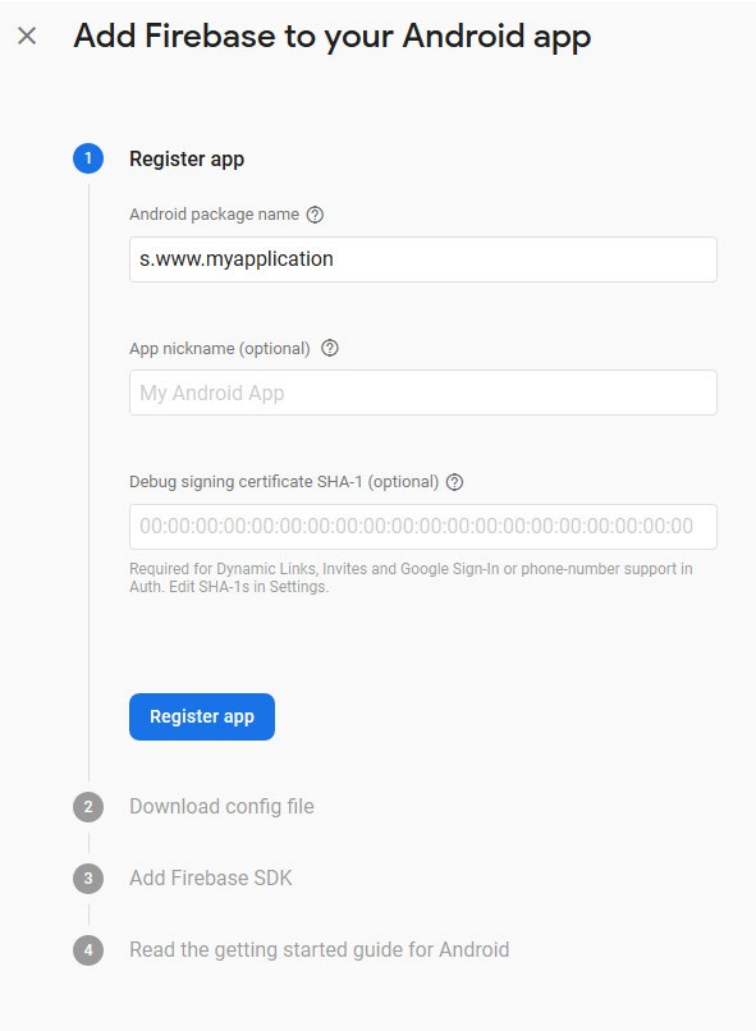
After it is done, click on the Android Button on the main page.



Find your application name in the **build.gradle(Module:app)** file in your project. You can find the build.gradle file under **Gradle Scripts.** Look for the **applicationId.**



Copy the applicationId into the Android Package Name on Firebase.



Follow the rest of the instructions on Firebase.