

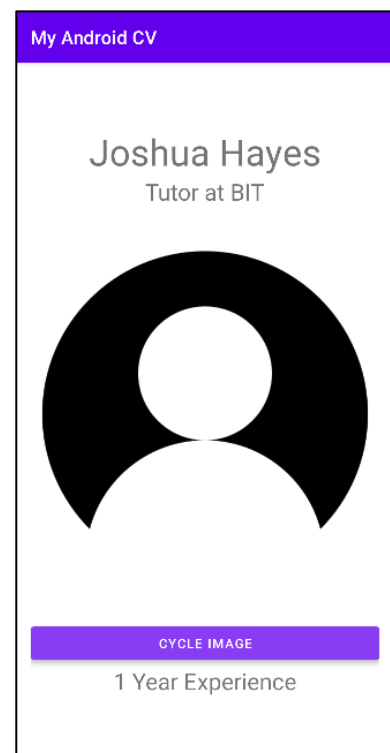
## Lab 2 – My Android CV

### Introduction

Welcome back to Mobile App Development. Now that we've got our copy of Android Studio all set up and we've taken a tour around the IDE, it's time to create an app a bit more useful than a simple Hello World message.

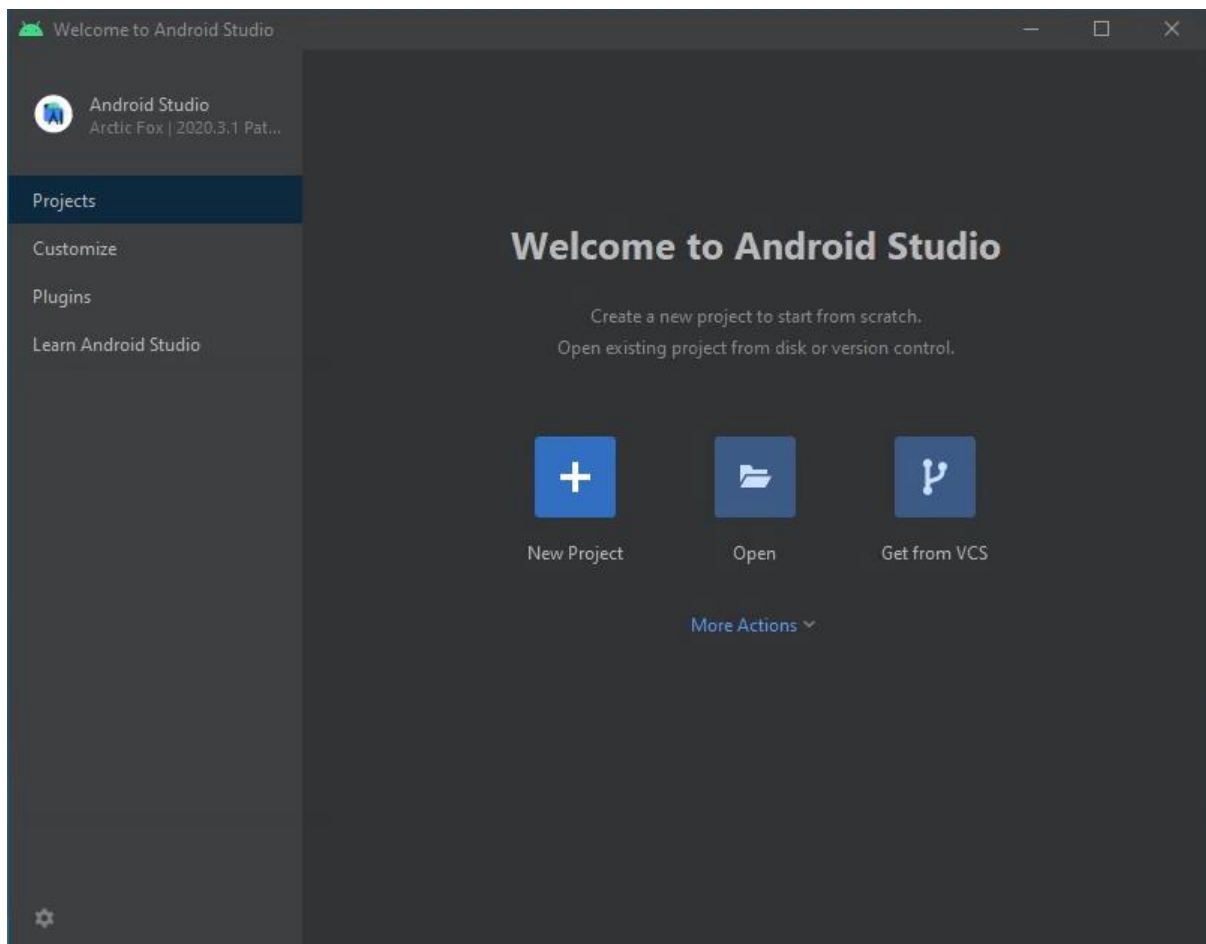
In this lab, we're going to be creating a CV Android app that will show off your skills to potential employers in the industry. It will contain your name, a bunch of text and an image. To get the most out of this session, I'd encourage you to experiment a bit and try and add some visual flare to your app, or try alternatives to the code suggestions I'm giving. The best way to learn is to experiment until you feel comfortable creating apps on your own. By the end, our app should be looking something like the picture on the right:

As always, you're more than welcome to ask any questions during this session or future sessions. Please feel free to call me over to help or send me [an email](#). I'm always happy to answer any queries you have.

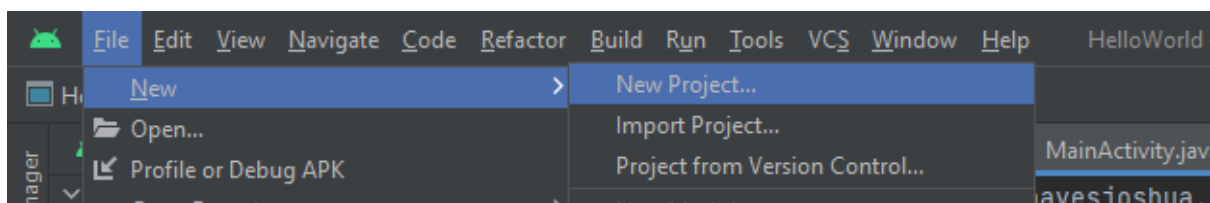


## Setting up our app

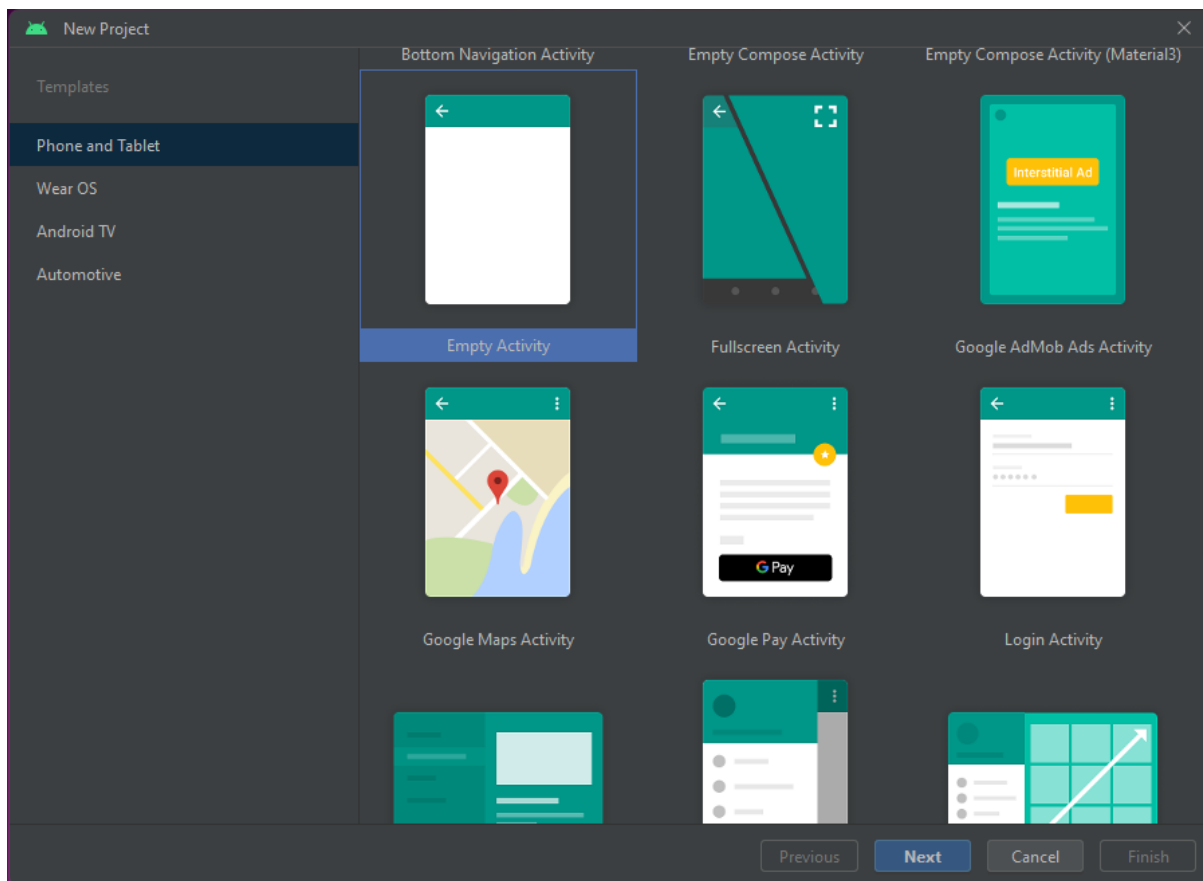
You should be greeted by this home screen when opening Android Studio back up.



If not, head to **File > New > New Project...** to get started.



We're going with the same template as last time. Ensure that you're in the Phone and Tablet tab, and select **Empty Activity**.



Call your project **MyAndroidCV**.



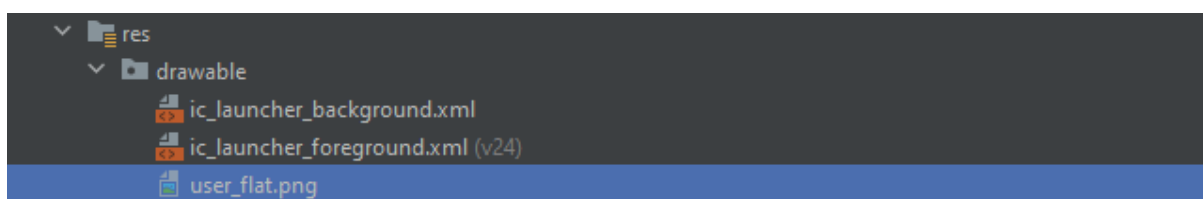
The **Package Name** should remember your details from our last session and have automatically updated – if not however, ensure that your app is **com.[yourname].myandroidcv**.



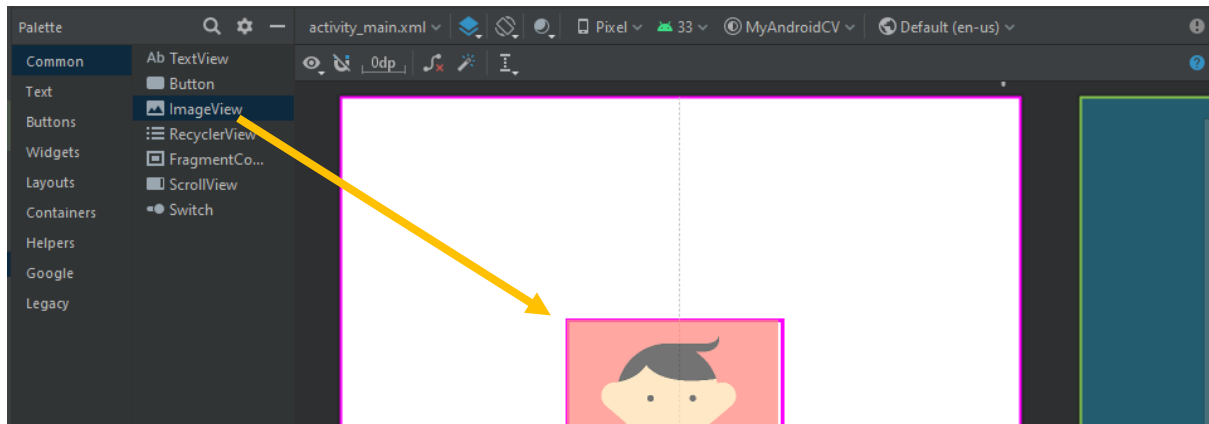
Now just ensure you are using Java targeting **API 26: Android 8.0 (Oreo)**, then hit **Finish**.

Inside **activity\_main.xml**, we should be able to use the designer view to see how our app looks. Right now, it's pretty basic, so let's add an image.

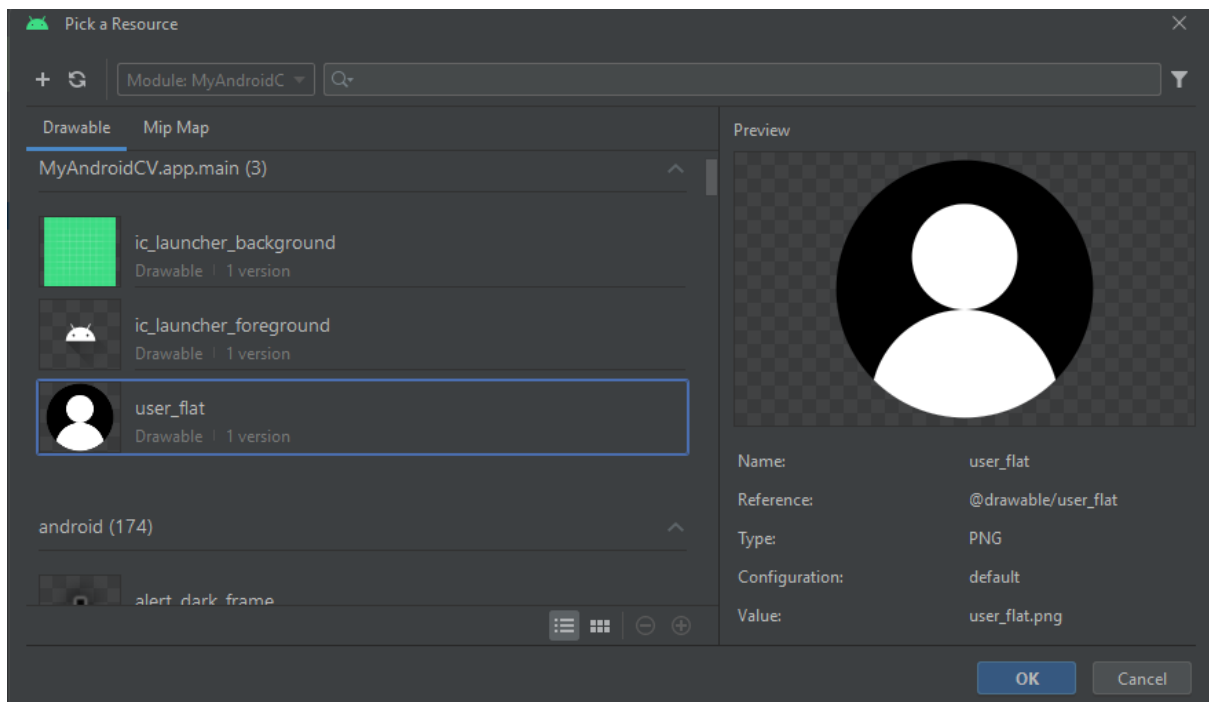
You can either grab your own image (preferably of yourself if you have one, it is a CV after all!) or you can [use this one here](#), and drag it into the **res > drawable** folder of your app.



In order to use our image, we need to use an **ImageView** component. While you're learning, I'd suggest dragging it into our app from the **Design** tab so it can automatically generate the XML code for you.



In the dialog that pops up, we can quickly pick a resource we'd like to use. Select the image you'd like and click **OK**.



If we pop over to our XML code, you should now see our newly created **ImageView** XML tag.

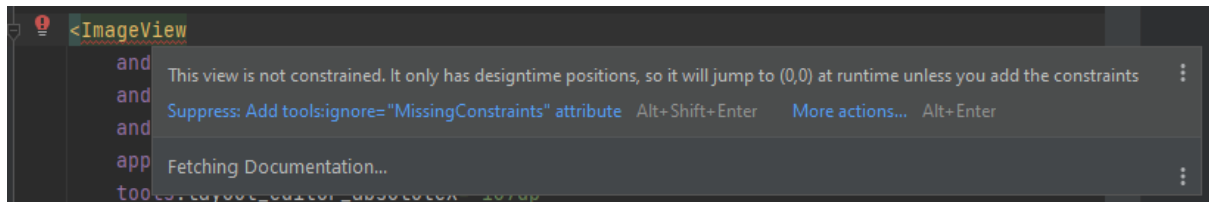
- The **ID** attribute is a unique identifier so we can find this element in our scripts later  

```
android:id="@+id/imageView"
```
- Your image source is defined by either a **srcCompat** or an **src** attribute. **srcCompat** is typically used for Vector images, but can also be used with **.png** image types. The string on the right always starts with an **@** symbol, followed by the path where the resource is saved. Do note that you do not need to define a file extension here.

```
app:srcCompat="@drawable/user_flat"
```

- The **layout\_width** and **layout\_height** defines how the image should be displayed relative to constraints on it. These default to **wrap\_content**, which should automatically fill the container it is placed in – but can be set to **dp** values as well.

You'll probably have noticed that the new **ImageView** tag is underlined in red as we haven't set up any constraints. Without constraints, our image may look a bit strange on a variety of different device sizes.



Let's fix this by changing our **ConstraintLayout** to a **LinearLayout**, the same as we did in Lab 1.

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

Remember from last lab, we have to configure our orientation to determine the direction our items are arranged, as well as some padding.

```
<LinearLayout
    . . .
    android:orientation="vertical"
    android:padding="16dp"
    . . .
</LinearLayout>
```

Once you've added this XML, our items should be displaying a bit nicer in our Design preview.

Using the **gravity** attribute, we can change the alignment of items within our **LinearLayout**. Go ahead and use **gravity** on the **LinearLayout** to set our alignment to **center**.

```
android:gravity="center"
```



Gravity is used to define how items should be displayed on the X and Y axis.

It's coming along nicely! Now that we're using a `LinearLayout` to handle our positioning, we can remove the **`absoluteX`** and **`absoluteY`** on our **`ImageView`** to keep our code tidy. Our code is now looking a bit like this so far:

```
<?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"
    android:orientation="vertical"
    android:gravity="center"
    android:padding="16dp"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

```

<ImageView
    android:id="@+id/imageView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:srcCompat="@drawable/user_flat" />

</LinearLayout>

```

Right now, our image is taking up a lot of space so there isn't much room for a CV. You can adjust the height by changing the **layout\_height** attribute on the **ImageView** to any **dp** value you'd like, or by dragging the bottom handle in the **Design** view with the image selected. I'm going with **370dp**.

```
android:layout_height="370dp"
```

I'm also going to add a **marginTop** to add **20dp** of space between my image and the top text.

```
android:layout_marginTop="20dp"
```

See if you can add a margin of **20dp below the image** as well.

Now it's time to set up our text on our app. Try and set up a few different text views yourself, replacing the text below for your own.

1. Joshua Hayes
2. Tutor for BIT
3. 1 Year Experience

Add as many text fields as you like, make sure it's relevant to you!

You can also have a play with the **textColor** and **textSize** attributes to improve the app's look.

Joshua Hayes  
Tutor at BIT



1 Year Experience

## Fixing warnings

You might notice after adding your text fields and changing the text size, that Android Studio is highlighting portions of your code

```

android:text="Joshua Hayes"
android:textSize="40dp"

```

Android Studio's default behaviour is to always highlight portions of code when there are warnings – these warnings won't necessarily stop the app from running, but are highlighted as they do not necessarily meet Google's design rules for apps.

The first one we'll fix is our **textSize**. Earlier I described **dp** as being density-independent pixels that can calculate based on the screen height and width. This way, our app should look relatively similar even when using a device with a much higher resolution. While **dp** is recommend for most elements, when dealing with text we should always instead use **scalable pixels (sp)**. These serve the same function as density-independent pixels, but scale depending on a user's font settings. This way, users that have increased their font sizes for accessibility reasons will have the font in your app increased as well.

```
android:textSize="40sp"
```

If you'd like to read up a bit more about Pixel density, you can do so by [clicking here](#).

Our text is also highlighted, because Android has been designed with the international market in mind and expects us to use a feature called **string resources**. This lets you easily add translations to your app at a later point.

You can find the strings under **res > values > strings.xml**.

```
<resources>
    <string name="app_name">MyAndroidCV</string>
</resources>
```

Right now we've only got one string resourced called **app\_name**. This is the same text that is displayed on our appbar at the top of our app. Feel free to change this if you wish.

Now add a **string** tag for each of the strings you added in your app earlier, and be sure to give each one a descriptive name that you'll recognise later.

```
<resources>
    <string name="app_name">My Android CV</string>
    <string name="my_name">Joshua Hayes</string>
    <string name="my_title">Tutor for BIT</string>
    <string name="my_experience">1 Year Experience</string>
</resources>
```

Now back in my **activity\_main.xml** file, I can access these strings by using **@string/[string-resource-name]**. Here's an example:

```
<TextView
    android:text="@string/my_name"
    . . .
```

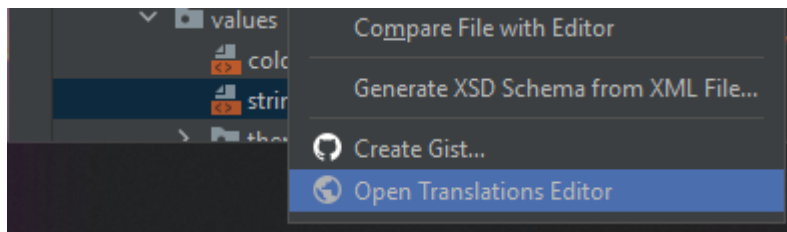



Our text should have stayed the same in our Design view, because it's now getting the value from our string resource. You can also do the same with custom colours if you wish, by adding a colour into **res > values > colors.xml** and referencing it with **@color/[color-resource-name]** in your activity\_main.xml file.

Keeping your colours and string resources in these files will mean that you only ever need to update them in one place down the road. Now that we've fixed our errors, let's add a translation!


## Translations

In order to add alternative translations to our app, **right-click** our **strings.xml** file and select **Open Translations Editor**.



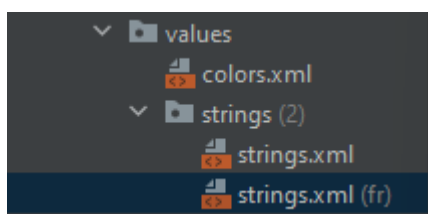
To add a new language to translate your app in, click the  icon in the top toolbar and select a language you'd like to configure. I'm going with French.

I've used Google Translate to help me fill in these fields – but if you speak another language and feel comfortable, feel free to add your own translations!

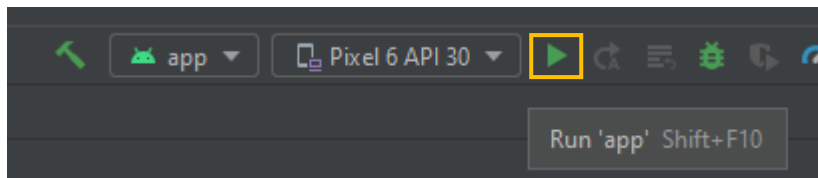


Key	Resource Folder	Untranslatable	Default Value	French (fr)
app_name	app/src/main/res	<input type="checkbox"/>	My Android CV	Mon CV Android
my_name	app/src/main/res	<input type="checkbox"/>	Joshua Hayes	Josué Hayes
my_title	app/src/main/res	<input type="checkbox"/>	Tutor for BIT	Professeur d'informatique
my_experience	app/src/main/res	<input type="checkbox"/>	1 Year Experience	1 an d'expérience

Now if you close the translation editor, you'll notice that our strings are split into two files for each translation.

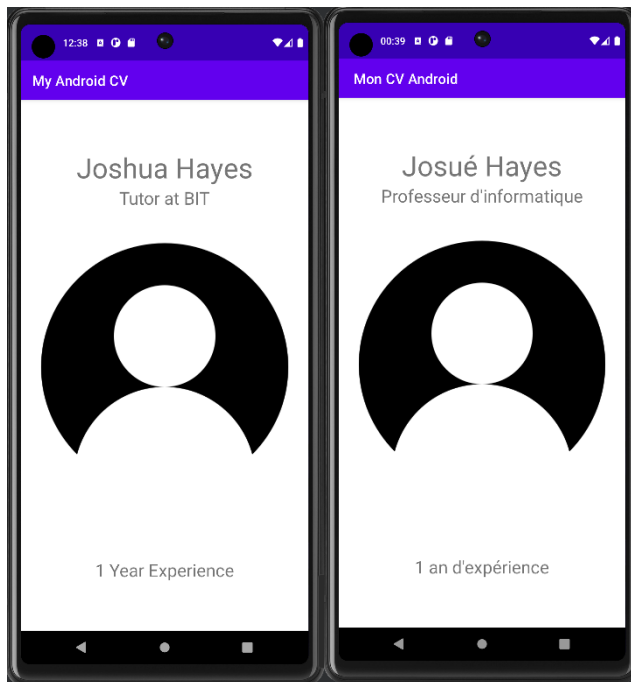


Let's run our app and see how it looks so far by clicking the play icon in the toolbar at the top.



If this is the first time you've used the emulator today it might take a while to open.

If we head into our device settings on our Virtual Device and change our language while our app is running, we should see the translation apply immediately!



## Challenge

Now that our app is pretty much complete, I'd like you to add a button that will change our image to display something different. When you click it again, it will go back to the original image.

Remember:

- You can use the Design view to quickly add a button without remembering the tags to do it in base XML
- Create your own **onClick** event in the `activity_main.xml` file
- Use the **setImageResource()** method to set your image. Don't forget to use **R.drawable.[image-name]** to find your image!
- Don't forget to add your button text to the `strings.xml` file and a translation!

If you are completely stuck, there are some code examples on the next page – but try and do it yourself before taking a look!

## Challenge Help

Add an `onClick` attribute to your Button in **activity\_main.xml**. The name can be whatever you want, as long as the method matches the name inside your java file later.

```
android:onClick="changeImage"
```

In **MainActivity.java**, add a new method that receives a **View** object.

```
public void changeImage(View view) {  
}
```

In this method, get a reference to the image view ID from our **activity\_main.xml** file.

```
ImageView imageView = findViewById(R.id.imageView);
```

You'll need to add a reference to **android.widget.ImageView** which you can do by using the quick actions when right-clicking on the code we just added, or by adding the following code above our methods.

```
import android.widget.ImageView;
```

Now use the **setImageResource** method to change our image to a new image that you've added to your project. We can reference the image by using **R.drawable.[image-name]**.

```
imageView.setImageResource(R.drawable.img2);
```

## Submission

When submitting your lab file, ensure that you have put the entire contents of your app's folder inside a **.zip** file. This will be in the location that you specified at the beginning of today's lab. It should be called **MyAndroidCV**.

Now rename your **.zip** file to

**[YourStudentID]\_IT617\_Lab2.zip**

We'll be using this naming convention for all of our labs moving forward.

Finally, upload this **.zip** to Blackboard.

*If you have any issues getting a lab completed or have been away sick for a length of time, please ensure that you send me an email to explain the situation before the due date. If you do not email me, you may receive a 0.*