CP3406_CP5307 Codelab 3.1: Create XML designs for Android

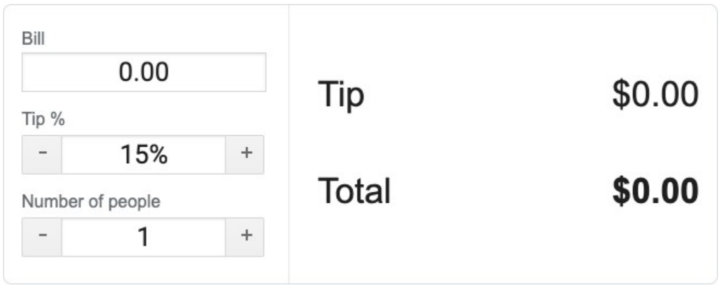
Contents

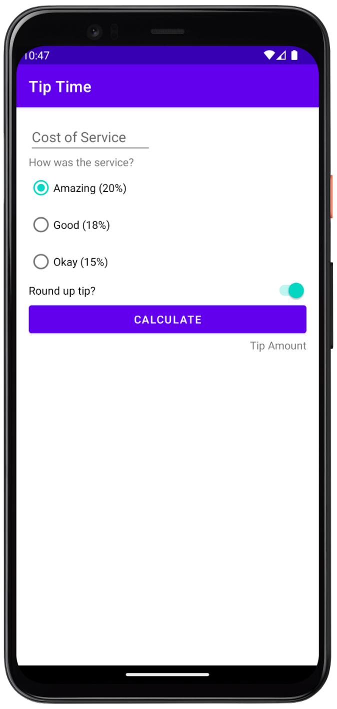
1. Before you start

In this codelab, you'll compile the design for a basic tip calculator app. At the end of the codelab, you will have a nice UI for the app, but it will not yet calculate the tip. In the next codelabs, we'll implement the functionality and make the app look more professional.

2. Start the project

Spend a minute or two searching online for a tip calculator tool, like the following. Note how the user can enter in the cost of their bill, the tip percentage, and number of people.





In this week's series of codelabs, you'll compile a simple version of a tip calculator as an Android app.

Developers often work this way: they get a simple version of the app ready and partially running (even if itdoesn't look very good) and then improve it so that it works completely and look visually elegant.

At the end of this codelab, your tip calculator app will look something like this:

You will use these elements of the UI that Android provides:

- `EditText`: Used to enter and edit text.
- `TextView`: This item is used to display text, such as the question about the service and the amount of the tip.
- `RadioButton`: A selection button to choose a tip option.
- `RadioGroup`: Used to group the selection button options.
- `Switch`: This is a boolean button to choose whether to round the tip or not.

Create an empty views activity project

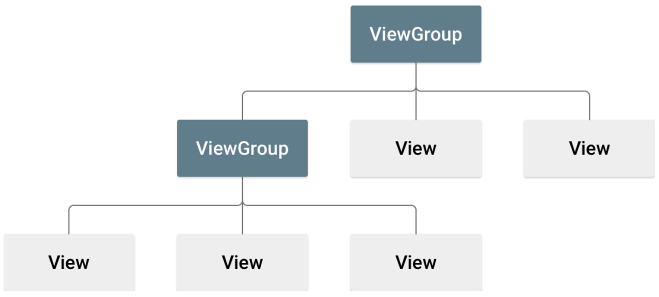
1. For starters, create a new Kotlin project on Android Studio with the **Empty Views Activity** template.
2. Use the name "Tip Time" for the app, which must have a minimum level of API 24. The name of the package is **au.edu.jcu.tiptime**.
3. Click **Finish** to create the app.

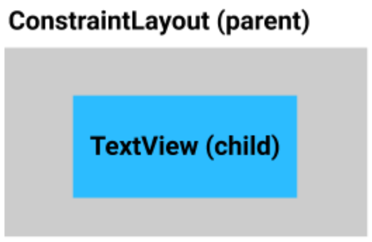
3. Read and understand XML

Instead of using the **design editor** you already know, this week you'll modify the XML that describes the UI to compile the design of your application. Learning to understand and modify UI designs using XML is important as an Android developer.

XML means *extendable markup language*, which is a way of describing data using a text-based document. Because the XMLformat is extendable and very flexible, it is used for several different things, including the definition of the UI design of Android apps. You may remember from previous codelabs that other resources used in your app, such as strings, are also defined in an XML file called `strings.xml`.

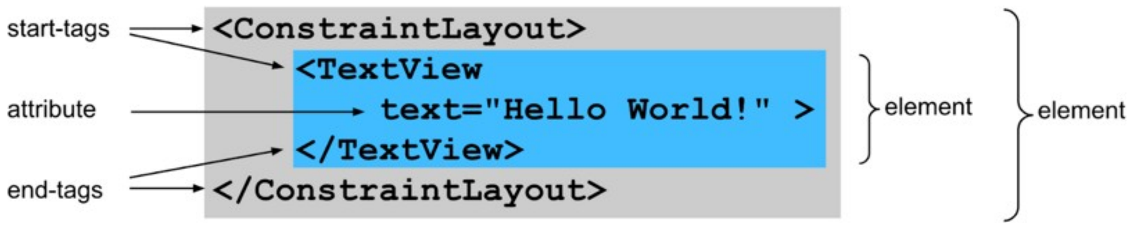
The UI of an Android app is compiled as a hierarchy of components (widgets) and the on-screen representations of those components. For example, a `ConstraintLayout` (the parent element) may contain `Buttons`, `TextViews`, `ImageViews` or other views (the child elements). Remember that `ConstraintLayout` is a subclass of `ViewGroup`. It will allow you to position child views flexibly.





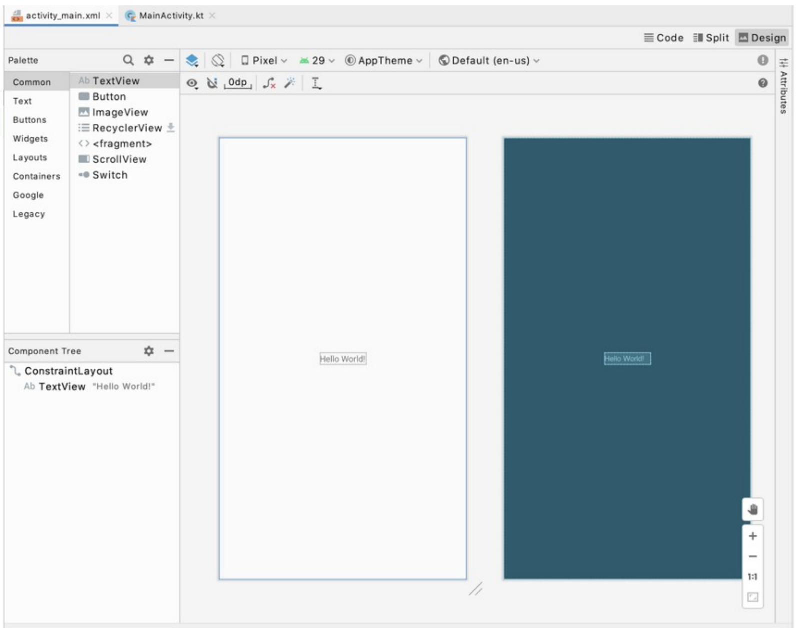
Each UI element is represented by an XML *element* in the XML file in XML format. Each item starts and ends with a label, and each label starts with a < and ends with a >. Just as you can set attributes on the elements of the UI through the **design editor (design view)**, the XML elements can also have *attributes*. Simplified, the XML of the UI elements mentioned above could be something like this:

```
<ConstraintLayout>
    <TextView
        text="Hello World!">
    </TextView>
</ConstraintLayout>
```

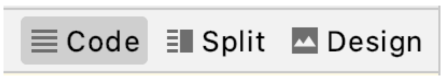


Let's see a real example.

4. Open up. `activity_main.xml`.
5. You'll notice that the app shows a `TextView` which says "Hello World" inside a `ConstraintLayout` as you saw in previous projects created from this template.



6. Find the options for **the Code**, **Split** and **Design** views at the top right of the design editor.
7. Select the **Code** view.



Here's how the XML looks in `activity_main.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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=".MainActivity">
    <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" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Android Studio does some things to help make the XML easier to read, just as it does with your Kotlin code.

8. Notice the indentation. Android Studio does this automatically to show you the hierarchy of elements.The `TextView` is indented because it is contained in the `ConstraintLayout`. The `ConstraintLayout` is the upper element, and the `TextView` is the secondary. The attributes of each element are shown with an indentation for the purpose of indicating that they are part of that element.
9. Also take note of the colour coding (in the Android Studio XML, not this document!): some things are in blue, others are in green, etc. The similar parts of the file are drawn in the same colour to help you to distinguish. In particular, notice that Android Studio draws the beginning and end of the element tags with the same colour.

XML labels, elements and attributes

Below is a simplified version of the element `TextView` so you can see some of its important parts:

```
<TextView
    android:text="Hello World!"
/>
```

The line with `<TextView` is the beginning of the tag and the line with `/>` is the end of it. The line with `android:text="Hello World!"` is an attribute that represents the text that will show in the `TextView`. XML elements without an explicitly separate closing tag are called an *empty tag*. It would be the same if you wrote it with a separate *start* and end *tag*, as shown below:

```
<TextView
    android:text="Hello World!"
></TextView>
```

It is also common to write an empty element tag on as few lines as possible and combine the end of the label with the pre-line. Therefore, you may see an empty element tag on two lines (or even on a line if it has no attributes):

```
<!-- with attributes, two lines -->
<TextView
    android:text="Hello World!" />
```

The element `ConstraintLayout` is written with separate start and end tags, as it must be able to contain other elements within it. This is a simplified version of the element `ConstraintLayout` containing the element `TextView`:

```
<androidx.constraintlayout.widget.ConstraintLayout>
    <TextView
        android:text="Hello World!" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

If you wanted to add another `View` as a secondary element of `ConstraintLayout`, like a `Button` underneath the `TextView`, it would be written after the end of the `/>` of the `TextView` andbefore the end tag of the `ConstraintLayout` as shown below:

```
<androidx.constraintlayout.widget.ConstraintLayout>
    <TextView
        android:text="Hello World!" />
    <Button
        android:text="Calculate" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Learn more about XML for designs

1. Look at the label `ConstraintLayout` and notes that it says `androidx.constraintlayout.widget.ConstraintLayout` instead of just `ConstraintLayout`, like the `TextView`. This is because `ConstraintLayout` is part of Android Jetpack, which contains code libraries that offer additional features in addition to Android's main platform. Jetpack has useful features that you can take advantage of in order to facilitate app compilation. You will recognize that this component of the UI is part of Jetpack because it starts with "androidx."
2. You may have noticed the lines that start with `xmlns:`, followed by `android`, `app` and `tools`.

```
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"
```

The `xmlns` refers to the XML namespace, and each line defines a *schema* or vocabulary for attributes related to those words. The Namespace `android:`, for example, marks attributes defined by the Android system. All the XML attributes of the design begin with one of those namespaces.

3. The blank space between XML elements does not change the meaning for the computer, but can facilitate the reading of the XML.

Android Studio will automatically add blank and indented spaces to facilitate reading.

You can add comments to the XML, just as you would with the Kotlin code. Start with `<!--` and ends with `-->`.

```
<!-- this is a comment in XML -->
<!-- this is a
multi-line
Comment.
And another
Multi-line comment -->
```

4. Look at the first line of the file:

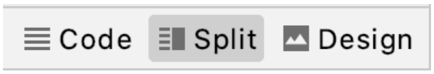
```
<?xml version="1.0" encoding="utf-8"?>
```

This indicates that this file is in XML format, but not all XML files include it.

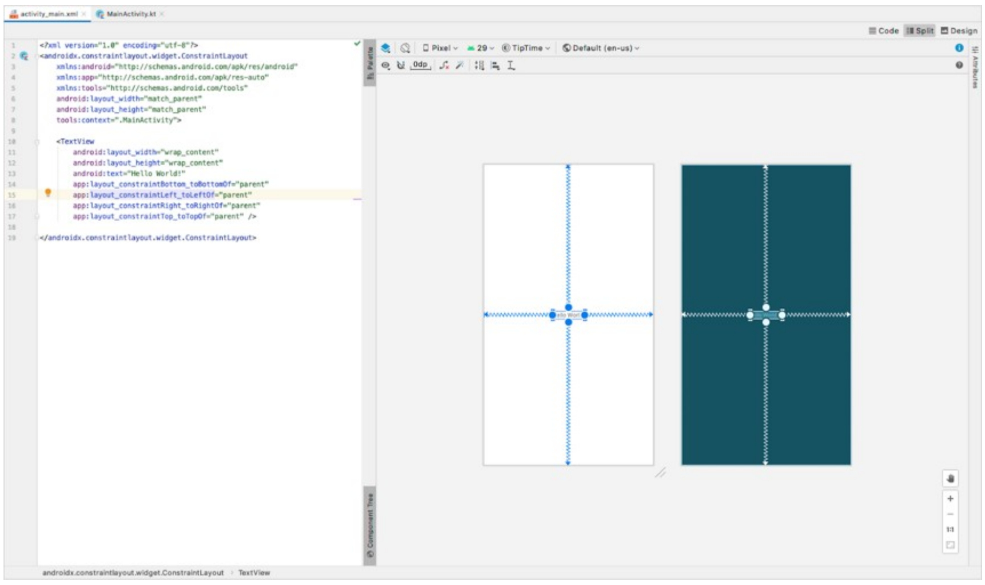
Note: In case there is a problem with the XML for your app, Android Studio will highlight the text in red. If you move the mouse over the red text, Android Studio will show you more information about the problem. If the problem is not obvious, look at the indenting and color encoding, which can give you a clue about why something doesn't work.

4. Compile the design in XML

1. Still in `activity_main.xml`, change to the **Split** view of the screen to see the XML next to the **design editor**. The **design editor** lets you get a preview of the UI design as you modify XML.



2. You can use whatever view you want. However, for this codelab, you must use **the Split** one to view the XML you are editing and the corresponding changes in the **design editor**.
3. Try clicking on different lines in XML (one below the `ConstraintLayout` and then another one under the `TextView`) and note that the relevant view shall be selected from the **design editor**. The reverse also works; for example, if you click on the `TextView`, in the **design editor**, the corresponding XMLwill be highlighted.



```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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=".MainActivity">

    <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" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Delete the TextView element

1. You don't need the `TextView` anymore, so remove it. Make sure you erase everything from the opening tag `<TextView` until the `/>` closure.

```
<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" />
```

All that's left in the file is the `ConstraintLayout`:

```
<androidx.constraintlayout.widget.ConstraintLayout
    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=".MainActivity">
</androidx.constraintlayout.widget.ConstraintLayout>
```

2. Add a padding of 16dp to `ConstraintLayout` so that the UI is not stuck to the edge of the screen.

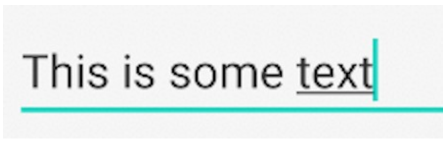
The padding is similar to margins, but adds internal space to `ConstraintLayout.` Instead of adding external space.

```
<androidx.constraintlayout.widget.ConstraintLayout
    ...
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    tools:context=".MainActivity">
```

Note: In order to shorten some of the code displayed in this codelab, not all elements are displayed. The code that has not changed or is not relevant to the current step will be represented with suspensive points (3 consecutive points `...`) so you can focus on the most important parts of the code.

Add a text field for the cost of the service

In this step, you will add the UI element so that the user can enter the cost of the service in the app. You'll use an element `EditText`, which will allow the user to enter or modify text in an app.



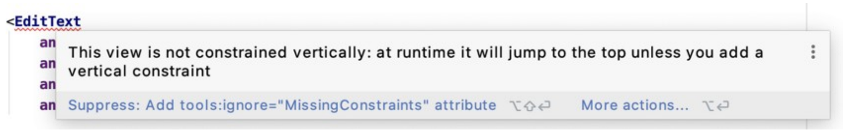
1. See the documentation of `EditText` and check the sample XML.
2. Look for a blank space between the start and end tags of `ConstraintLayout`.
3. Copy and paste the XML of documentation in that space of your design on Android Studio.

The design file should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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:padding="16dp"
    tools:context=".MainActivity">
    <EditText
        android:id="@+id/plain_text_input"
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:inputType="text"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

You may not understand all this yet, but we will explain it in the next steps.

4. Notice that `EditText` appears in red.
5. Place the cursor on it and you will see the error "view is not constrained" (the view is not restricted), which should be familiar to you from previous codelabs. Remember that the secondary elements of a `ConstraintLayout` need constraints so that the design knows how to organize them.



```
<EditText
  an
  an
  an
  an
```

This view is not constrained vertically: at runtime it will jump to the top unless you add a vertical constraint

Suppress: Add tools:ignore="MissingConstraints" attribute ⌥⇧↩ More actions... ⌥↩

6. Add these restrictions to `EditText` to anchor it in the upper left corner of the upper element.

```
app:layout_constraintStart_toStartOf="parent"
app:layout_constraintTop_toTopOf="parent"
```

If you are writing in English or in another language that is written from left to right (LTR), the initial edge will be the left. However, some languages, such as Arabic, are written from right to left (RTL), so the initial edge will be the right. Therefore, the restriction uses "start" (start) so that it can work with both LTR and RTL languages. Similarly, the restrictions use "end" (end) instead of the right.

Note: The name of the constraints follows the format `layout_constraint<Source>_to<Target>Of`.

After adding the constraints, the element `EditText` should now look as follows:

```
<EditText
    android:id="@+id/plain_text_input"
    android:layout_height="wrap_content"
    android:layout_width="match_parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    android:inputType="text"/>
```

Check EditText attributes

Let’s check some other attributes of `EditText`.

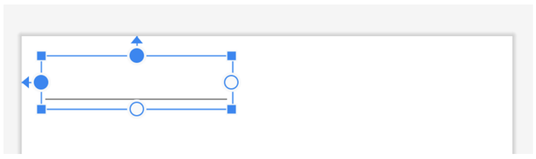
1. Find the attribute `id`, which is established in `@+id/plain_text_input`.
2. Change the attribute `id` to a more appropriate name: `@+id/cost_of_service`.

Note: When you add a `View` or any other resource with the **design editor**, Android Studio automatically assigns them a resource ID. When you write the XML manually, you must explicitly declare the Resource ID. New view IDs in XML format should be defined with the prefix `@+id`, which tells Android Studio to add that ID as a new resource ID.

Choose descriptive names for resources so that you know which elements they refer to. However, note that theymust be written in lowercase letters, and the different words must be separated with an underscore.

When you refer to resource ID in your app code, use `R.<type>.<name>`. For example, `R.string.roll`. In the case of `View`, the `<type>` is `id`, for example, `R.id.button`.

3. Observe the attribute `layout_height`. It is set to `wrap_content`, which means thatheight will be equal to that of its content. That's fine, as there will only be 1 line of text.
4. Observe the attribute `layout_width`. It is set to `match_parent`, but you can't set up `match_parent` in a secondary element of `ConstraintLayout`. Besides, the text field doesn't needto be that wide. Set it to a fixed width of `160dp`, which is enough space for the user to enter a service cost.



5. Observe the attribute `inputType` which is something new we haven’t encountered yet. The value of the attribute is `"text"`, which means that the user can enter any type of text characters in the screen field (alphanumeric characters, symbols, etc.).

```
android:inputType="text"
```

However, you'll want them to only write numbers on the `EditText` since the field represents a monetary value.

6. Delete the word `text` but leave the quotation marks.
7. Start writing `number` in place. After writing "n," Android Studio shows a list of possible endings thatinclude "n."

8. Choose `numberDecimal`, which limits the input to numbers with a decimal point.

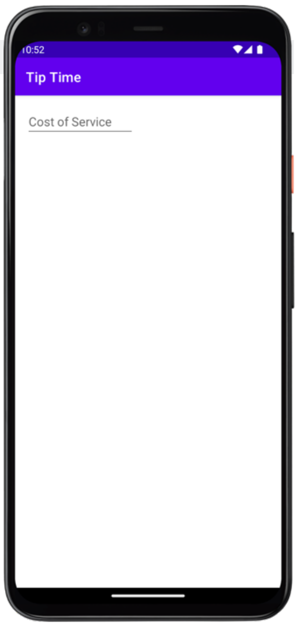
```
android:inputType="numberDecimal"
```

To view other input type options, see Specifying the type of input method in the developer documentation.

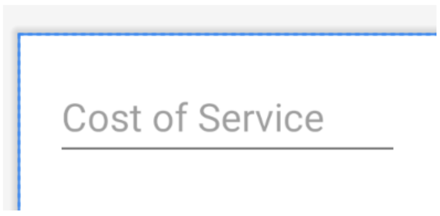
There is only one more change to be made. It would be helpful to show a suggestion about what the user should enter this field.

9. Add an attribute `hint` to the `EditText` to describe what the user must enter in this field.

```
android:hint="Cost of Service"
```



You'll also see this update in the **design editor**.



10. Run your app in the emulator. It should look like this:

Good job. It's not much yet, but this is a good start, and you managed to edit some XML. The XML of your design should look similar to the one shown below:

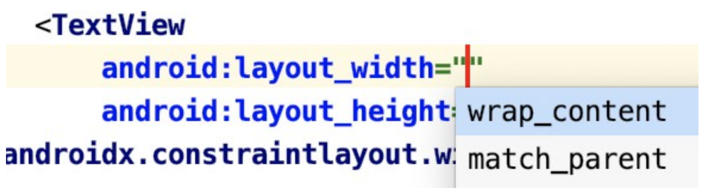
```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp"
    tools:context=".MainActivity">
    <EditText
        android:id="@+id/cost_of_service"
        android:layout_width="160dp"
        android:layout_height="wrap_content"
        android:hint="Cost of Service"
        android:inputType="numberDecimal"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Add a question about the service

In this step, you'll add a `TextView` to show this question: "How was the service?" Try writing this without copying and pasting. Android Studio suggestions should help you.

1. After the label close `EditText`, `/>`, add a new line and starts writing `<TextView`.
2. Select `TextView` from the suggestions, and Android Studio will automatically add the attributes `layout_width` and `layout_height` to the `TextView`.
3. Select `wrap_content` for both layout_width and layout_height since you just need the `TextView` to

be only as big as the text content it displays.

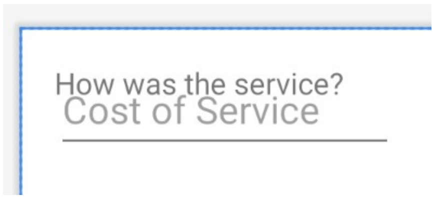


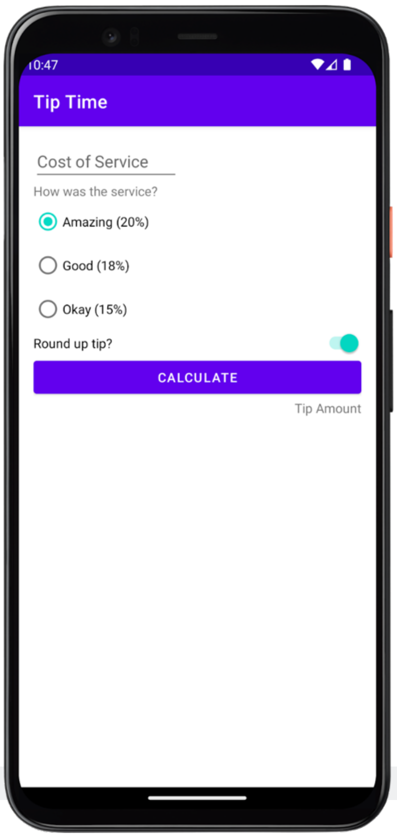
```
<TextView
    android:layout_width=""
    android:layout_height
androidx.constraintlayout.w
```

4. Add "`How was the service?`" to the attribute `text`.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="How was the service?"
```

5. Close the label with `/>`.
6. In the **design editor**, note that the `TextView` overlaps with the `EditText`.





That doesn't look good, so you'll have to add restrictions on the `TextView`. Think about the restrictions you need. What horizontal and vertical position should you place the `TextView`? The app's screenshot can help you.

With regard to the vertical position, you will want to `TextView` to be below the text field for the cost of the service. In terms of the horizontal position, you'll want the `TextView` to be aligned with the initial edge of the parent view.

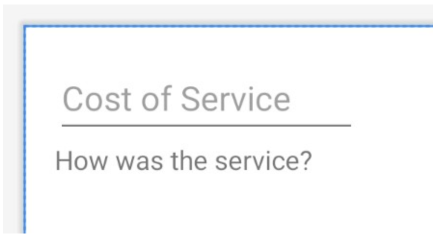
7. Add a horizontal constraint to the `TextView` to restrict its initial edge to that of the parent element.

```
app:layout_constraintStart_toStartOf="parent"
```

8. Add a vertical restriction to the `TextView` in order to restrict the upper edge of the `TextView` to the bottom of the `View` with the cost of the service.

```
app:layout_constraintTop_toBottomOf="@id/cost_of_service"
```

Notice that `@id/cost_of_service` does not contain a + because the ID is already defined.



The look still quite basic, but don't worry about it at the moment. You just have to make sure that all the necessary parts are on the screen and that the functionality has no problems. You can improve the look in the following codelabs.

9. Adds a Resource ID in the `TextView`. You should check this view later as you add more views and apply restrictions between them.

```
android:id="@+id/service_question"
```

At this point, your XML should be as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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:padding="16dp"
    tools:context=".MainActivity">
    <EditText
        android:id="@+id/cost_of_service"
        android:hint="Cost of Service"
        android:layout_height="wrap_content"
        android:layout_width="160dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:inputType="numberDecimal"/>
    <TextView
        android:id="@+id/service_question"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="How was the service?"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@id/cost_of_service"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

5. Add tip options

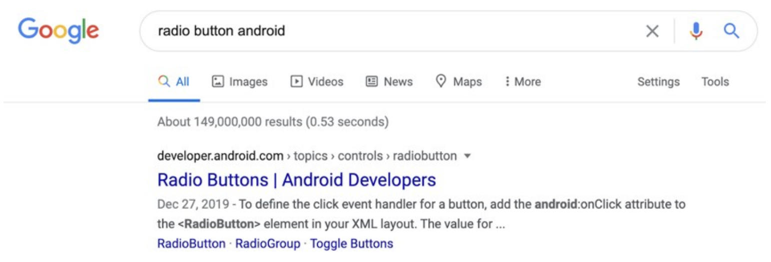
Now you will add selection buttons for the different tip options that the user can choose.

There should be three options:

- Amazing (20%)
- Good (18%)
- Acceptable (15%)

If you don't know how to do this, you can do a Google search. This is an excellent tool that developers use when they can't move forward.

1. Conduct a Google search for `radio button android`. The first result is a site guide for Android developers explaining how to use the selection buttons. Great.



Google
radio button android

All | Images | Videos | News | Maps | More | Settings | Tools

About 149,000,000 results (0.53 seconds)

developer.android.com › topics › controls › radiobutton

Radio Buttons | Android Developers

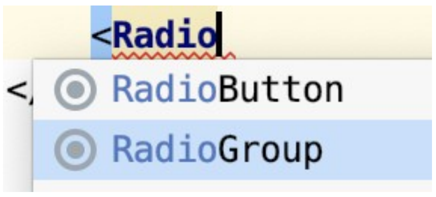
Dec 27, 2019 - To define the click event handler for a button, add the **android**:onClick attribute to the **<RadioButton>** element in your XML layout. The value for ...

RadioButton · RadioGroup · Toggle Buttons

2. Quickly read the selection button guide.

When reading the description, you can confirm that you can use a `RadioButton` as an element of the UI in your design for every selection button you need. In addition, you must group the selection buttons within a `RadioGroup`, since only one option should be selected at a time.

3. Go back to your design on Android Studio and add the `RadioGroup` and the `RadioButton` to your app.
4. After the element `TextView` but still inside the `ConstraintLayout`, start writing `<RadioGroup`. Android Studio will provide you with helpful suggestions to help you complete your XML.



```
<Radio
RadioButton
RadioGroup
```

5. Set the `layout_width` and the `layout_height` of the `RadioGroup` to `wrap_content`.
6. Add a resource ID configured as `@+id/tip_options`.
7. Close the start label with `>`.
8. Android Studio should automatically add `</RadioGroup>`. Like the `ConstraintLayout`, the element `RadioGroup` will have other elements within it, so you might want to move it to a new line.
9. Restrict the `RadioGroup` below the question about the service (vertically) and at the beginning of the parent element (horizontally).
10. Sets the attribute `android:orientation` to `vertical`. If you want the `RadioButtons` in a row, you would set the orientation as `horizontal`.

The XML of the `RadioGroup` should look like this:

```
<RadioGroup
    android:id="@+id/tip_options"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/service_question">
</RadioGroup>
```

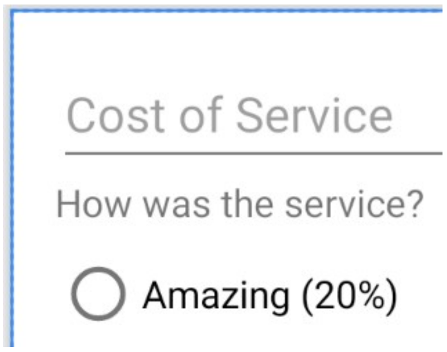
Add RadioButtons

1. After the last attribute of the `RadioGroup`, but before the end label `</RadioGroup>`, add a `RadioButton`.

```
<RadioGroup
    android:id="@+id/tip_options"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/service_question">
    <!-- add RadioButtons here -->
</RadioGroup>
```

2. Sets the `layout_width` and the `layout_height` to `wrap_content`.
3. Assign an ID `@+id/option_twenty_percent` to the `RadioButton`.
4. Configure the text as “`Amazing (20%)`”.
5. Close with `/>`.

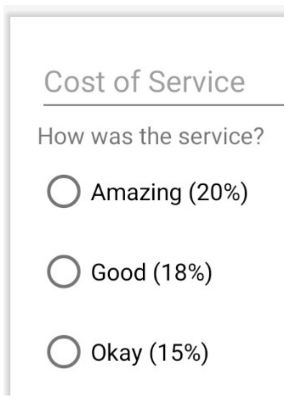
```
<RadioGroup
   android:id="@+id/tip_options"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   app:layout_constraintTop_toBottomOf="@id/service_question"
   app:layout_constraintStart_toStartOf="parent"
   android:orientation="vertical">
   <RadioButton
       android:id="@+id/option_twenty_percent"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Amazing (20%)" />
</RadioGroup>
```



Now you've added a `RadioButton`, can you modify the XML to add 2 additional selection buttons for the options `Good (18%)` and `Okay (15%)`?

This is the completed XML for the `RadioGroup` and the `RadioButtons`:

```
<RadioGroup
   android:id="@+id/tip_options"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   app:layout_constraintTop_toBottomOf="@id/service_question"
   app:layout_constraintStart_toStartOf="parent"
   android:orientation="vertical">
   <RadioButton
       android:id="@+id/option_twenty_percent"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Amazing (20%)" />
   <RadioButton
       android:id="@+id/option_eighteen_percent"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Good (18%)" />
   <RadioButton
       android:id="@+id/option_fifteen_percent"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Okay (15%)" />
</RadioGroup>
```



Add a default selection

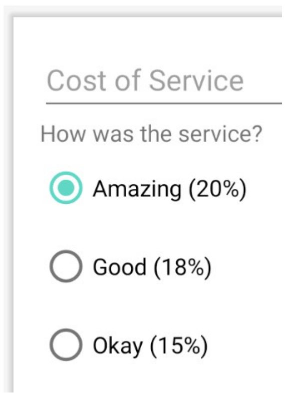
None of the tip options are currently selected. It would be good to select one of the selection button options by default.

There's an attribute in the `RadioGroup` in which you can specify the button that must be marked in the beginning. It's called. `checkedButton`. Set it to the Resource ID of the selection button you want to be selected.

1. In the `RadioGroup` set attribute `android:checkedButton` as `@id/option_twenty_percent`.

```
<RadioGroup
   android:id="@+id/tip_options"
   android:checkedButton="@id/option_twenty_percent"
   ...
```

In the **design editor**, note that the design is updated. The 20% tip option is selected by default. Good. It's starting to resemble a tip calculator.

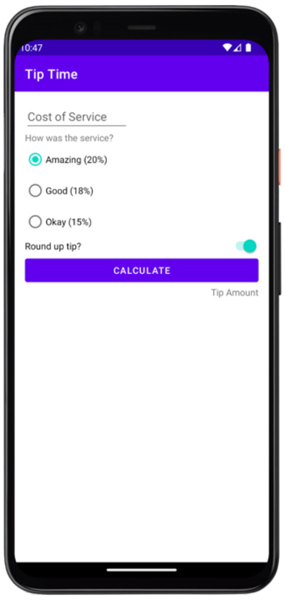


Here's how the XML looks so far:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    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:padding="16dp"
    tools:context=".MainActivity">
    <EditText
        android:id="@+id/cost_of_service"
        android:hint="Cost of Service"

        android:layout_height="wrap_content"
        android:layout_width="160dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:inputType="numberDecimal"/>
    <TextView
        android:id="@+id/service_question"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="How was the service?"
        app:layout_constraintTop_toBottomOf="@id/cost_of_service"
        app:layout_constraintStart_toStartOf="parent" />
    <RadioGroup
        android:id="@+id/tip_options"
        android:checkedButton="@id/option_twenty_percent"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/service_question"
        app:layout_constraintStart_toStartOf="parent"
        android:orientation="vertical">
        <RadioButton
            android:id="@+id/option_twenty_percent"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Amazing (20%)" />
        <RadioButton
            android:id="@+id/option_eighteen_percent"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Good (18%)" />
        <RadioButton
            android:id="@+id/option_fifteen_percent"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Okay (15%)" />
    </RadioGroup>
</androidx.constraintlayout.widget.ConstraintLayout>
```

6. Complete the rest of the design



You've reached the last part of the design. You'll add a `Switch`, a `Button` and one `TextView` to show the calculated amount of the tip.

Add a Switch to round the tip

You'll use a widget of `Switch` in order to allow the user to select Yes or No for the purpose ofrounding the tip.

You'll want him to `Switch` to be as wide as the parent element. Therefore, you may think that the width should be set to `match_parent`. As noted above, you cannot use `match_parent` for the elements of a UI within a `ConstraintLayout`. Instead, you will have to constrain the start and end edges of it to the parent view, and set the width to `0dp`. If you set the width in `0dp`, you will tell the system not to calculate the width, but only to try to match the restrictions present in the view.

1. Adds an element `Switch` after the XML for the `RadioGroup`.
2. As indicated above, set its `layout_width` to `0dp`.
3. Set the `layout_height` to `wrap_content`. This will make the `Switch` as tall as the content inside it.
4. Sets the attribute `id` to `@+id/round_up_switch`.
5. Sets the attribute `text` to `Round up tip?`. It will be used as a label of the `Switch`.

6. Restrict the starting edge of the `Switch` to that of the `tip_options` and the end edge at the end of the parent element.
7. Restrict the top of the `Switch` to the lower of the `tip_options`.
8. Close the label with `/>`.

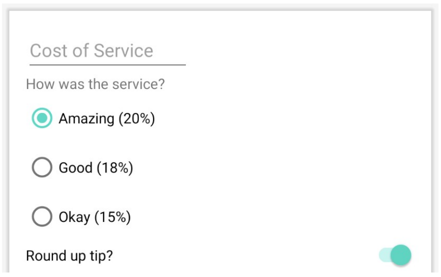
It would be good if the switch is active by default, and there is an attribute for it, `android:checked`, whose possible values are `true` (activated) and `false` (deactivated).

9. Sets the attribute `android:checked`in `true`.

With all this, the XML of the element `Switch` will look like this:

```
<Switch
    android:id="@+id/round_up_switch"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:checked="true"
    android:text="Round up tip?"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="@id/tip_options"
    app:layout_constraintTop_toBottomOf="@id/tip_options" />
```

Note: If you place the cursor on `Switch` in your XML design, you may see a suggestion from Android Studio that says "Use `SwitchCompat`From AppCompat or `SwitchMaterial` from Material library." You will implement this suggestion in a codelab after the compilation of the tip calculator app, so for the time being you can ignore the warning.



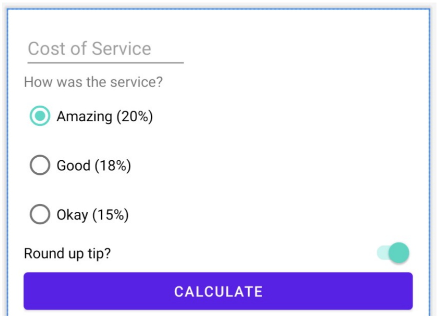
Add the Calculate button

Now you'll add a `Button` so that the user can request that the tip be calculated. You will want the button to be as wide as the parent element, so the horizontal and vertical restrictions will be the same as those of the `Switch`.

1. Add a `Button` after the `Switch`.
2. Sets the width in `0dp` like you did for the `Switch`.
3. Set the height to `wrap_content`.
4. Set the ID to `@+id/calculate_button`, with the text `"Calculate"`.
5. Restrict the upper edge of `Button` to the lower edge of the `Switch`.
6. Restrict the starting and end edges to those of the parent element.
7. Close the label with `/>`.

Below, we show you what the XML looks like for the newly added `Button`:

```
<Button
   android:id="@+id/calculate_button"
   android:layout_width="0dp"
   android:layout_height="wrap_content"
   android:text="Calculate"
   app:layout_constraintTop_toBottomOf="@id/round_up_switch"
   app:layout_constraintStart_toStartOf="parent"
   app:layout_constraintEnd_toEndOf="parent" />
```

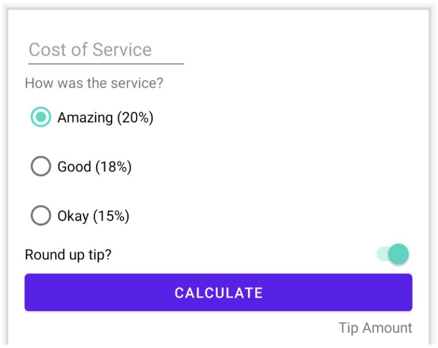


Add the resulting tip

You're almost done with the design. In this step, you'll add an element `TextView` for the tip result, located under the **Calculate** button.

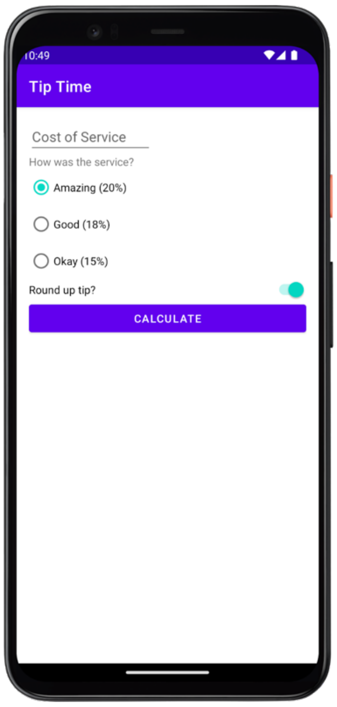
1. Add a `TextView` with a resource ID of `tip_result` and text `Tip Amount`.
2. Restrict the right edge of the `TextView` to that of the parent element.
3. Restrict the top edge to the bottom of the **Calculate** button.

```
<TextView
    android:id="@+id/tip_result"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/calculate_button"
    android:text="Tip Amount" />
```



4. Run the app. It should look very similar to this screenshot.

Excellent work, especially if it's the first time you've worked with XML.



Note that the app may not look exactly the same as screenshot, as templates may have changed in a later version of Android Studio. The **Calculate** button is not yet working, but you can type the cost, select the percentage of the tip and turn on or off the tip rounding option. In the next codelab, you'll make the **Calculate** button work.

7. Implement coding best practices

Extract the strings

You may have noticed the warnings about hard-coded strings. Based on prior code labs, remember that extracting strings into a resource file is best practice as it helps in reusing strings and translating your app. Check `activity_main.xml` and extracts all string resources.

1. Click on a string. Place the cursor over the yellow light bulb icon that appears click on it. Select **Extract String Resource**. The default names the Android Studio suggests for the string resources are fine. If you want, for tip options, you can use `amazing_service`, `good_service` and `ok_service` so that the names are more descriptive.

Now, check the string resources you just added.

2. If the **Project/Android** window is not displayed, click on the **Project** tab on the left side of the window.
3. Open **strings.xml** in res/values.

```
<resources>
    <string name="app_name">Tip Time</string>
    <string name="cost_of_service">Cost of Service</string>
    <string name="how_was_the_service">How was the service?</string>
    <string name="amazing_service">Amazing (20%)</string>
    <string name="good_service">Good (18%)</string>
    <string name="ok_service">Okay (15%)</string>
    <string name="round_up_tip">Round up tip?</string>
    <string name="calculate">Calculate</string>
    <string name="tip_amount">Tip Amount</string>
</resources>
```

Automatically format the XML

Android Studio provides several tools to sort your code and ensure that it complies with recommended programming conventions.

1. In `activity_main.xml`, choose **Edit > Select All**.
2. Select **Code > Reformat Code**.

This will ensure that the indenting is consistent and may reorganize some of the XML of the UI elements in order to better group things together, for example by uniting all attributes `android:` of an element.

8. Summary

- XML (extensible markup language) is a way of organizing the text, consisting of tags, elements and attributes.
- Use XML to define the design of an Android app.
- Use `EditText` in order to allow the user to enter or edit text.
- An `EditText` should have a suggestion that tells the user what is expected to enter into that field.
- Specify the attribute `android:inputType` to limit the type of text that the user can enter in a field `EditText`.
- Make a list of exclusive options with `RadioButtons`, grouped with a `RadioGroup`.
- A `RadioGroup` can be vertical or horizontal, and you can specify what `RadioButton` must be selected initially.
- Use a `Switch` in order to allow the user to activate or deactivate two options.
- You can add a label to a `Switch` without using a `TextView`.
- Each child element of a `ConstraintLayout` must have vertical and horizontal restrictions.