

### Build a simple user interface

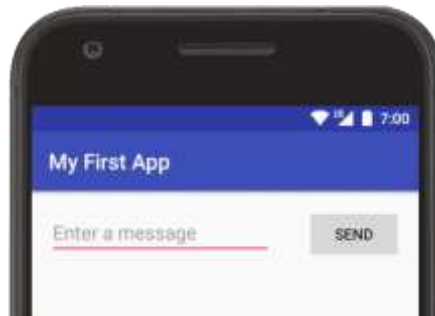


Figure 01:

The user interface for an Android app is built using a hierarchy of *layouts* ([ViewGroup](#) objects) and *widgets* ([View](#) objects). Layouts are containers that control how their child views are positioned on the screen. Widgets are UI components such as buttons and text boxes.

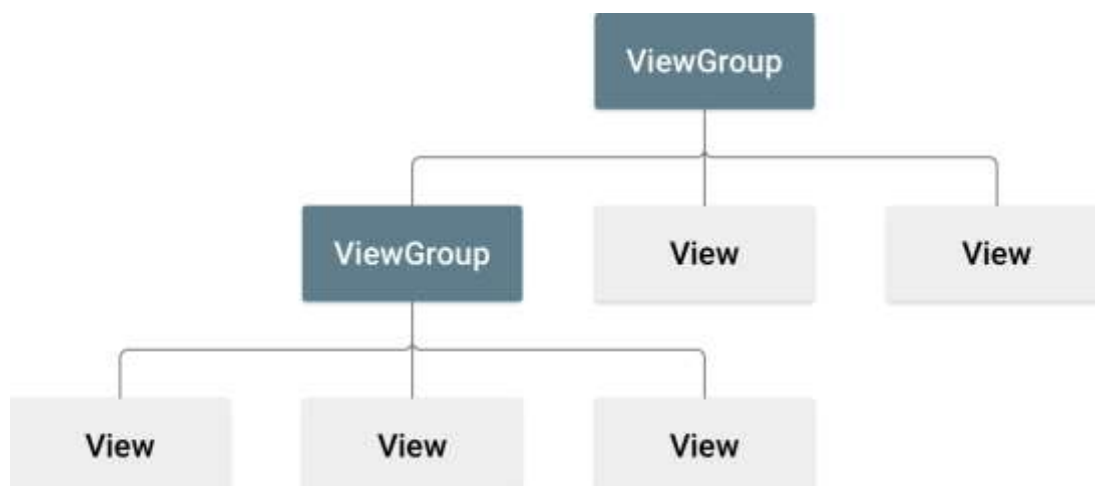








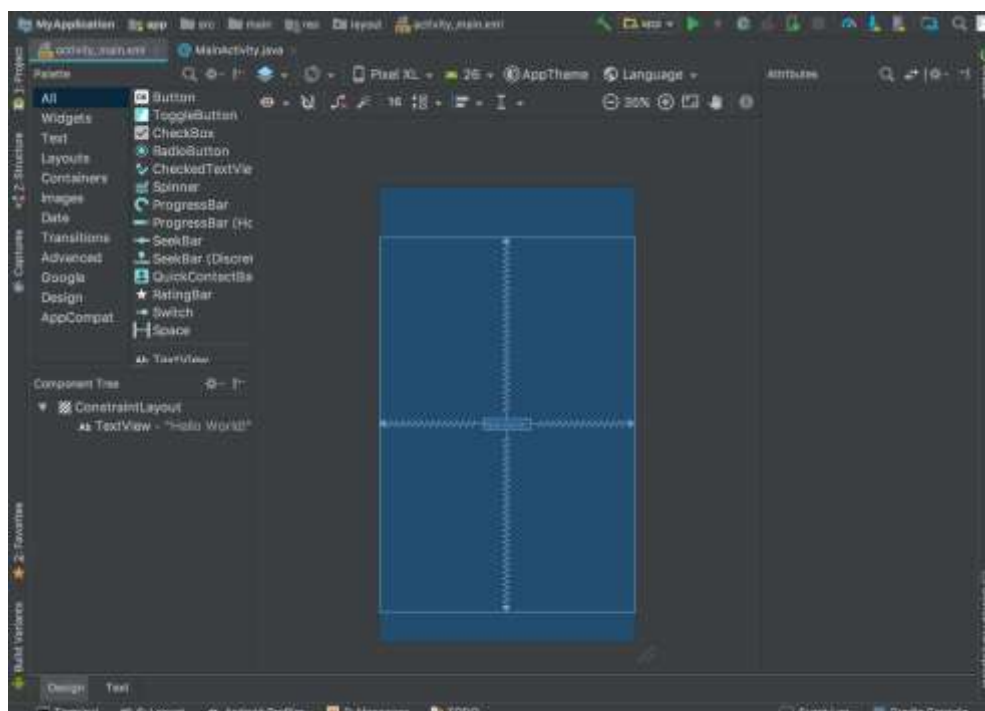
Figure 02: Illustration of how ViewGroup objects form branches in the layout and contain View objects

### Open the Layout Editor

To get started, set up your workspace as follows:

1. In Android Studio's Project window, open **app > res > layout > activity\_main.xml**.
2. To make more room for the Layout Editor, hide the **Project** window by selecting **View > Tool Windows > Project** (or click **Project**  on the left side of Android Studio).
3. If your editor shows the XML source, click the **Design** tab at the bottom of the window.
4. Click **Select Design Surface**  and select **Blueprint**.
5. Click **Show**  in the Layout Editor toolbar and make sure **Show Constraints** is checked.
6. Make sure Autoconnect is off. The tooltip in the toolbar should read **Turn On Autoconnect**  (because it's now off).
7. Click **Default Margins**  in the toolbar and select **16** (you can still adjust the margin for each view later).
8. Click **Device in Editor**  in the toolbar and select **5.5, 1440 × 2560, 560dpi (Pixel XL)**.

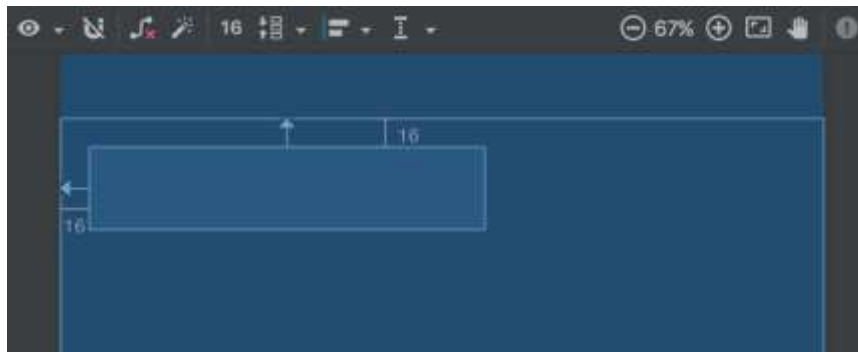
Your editor should now look as shown in below figure.



The **Component Tree** window on the bottom-left side shows the layout's hierarchy of views. In this case, the root view is a **ConstraintLayout**, containing just one **TextView** object.

**ConstraintLayout** is a layout that defines the position for each view based on constraints to sibling views and the parent layout. In this way, you can create both simple and complex layouts with a flat view hierarchy. That is, it avoids the need for nested layouts (a layout inside a layout, as shown in figure 2), which can increase the time required to draw the UI.

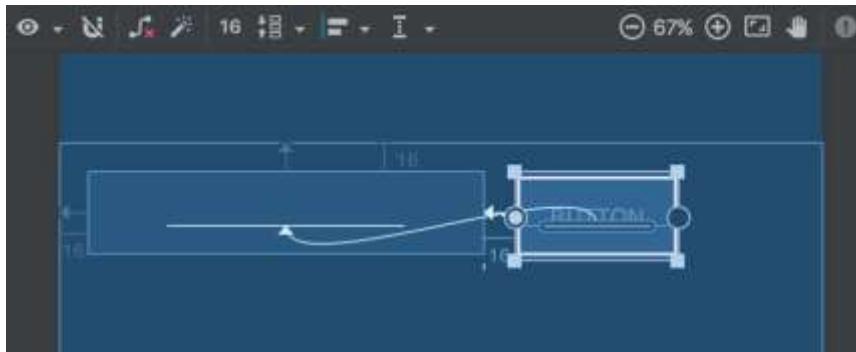
### Add a text box




**Figure 5.** The text box is constrained to the top and left of the parent layout

1. First, you need to remove what's already in the layout. So click **TextView** in the **Component Tree** window, and then press Delete.
2. In the **Palette**, click **Text** to show the available text controls.
3. Drag **Plain Text** into the design editor and drop it near the top of the layout. This is an **EditText** widget that accepts plain text input.
4. Click the view in the design editor. You can now see the resizing handles on each corner (squares), and the constraint anchors on each side (circles).  
For better control, you might want to zoom in on the editor using the buttons in the Layout Editor toolbar.
5. Click-and-hold the anchor on the top side, and then drag it up until it snaps to the top of the layout and release. That's a constraint—it specifies the view should be 16dp from the top of the layout (because you set the default margins to 16dp).
6. Similarly, create a constraint from the left side of the view to the left side of the layout.


### Add a button




1. In the **Palette**, click **Buttons**.
2. Drag **Button** into the design editor and drop it near the right side.
3. Create a constraint from the left side of the button to the right side of the text box.
4. To constrain the views in a horizontal alignment, you need to create a constraint between the text

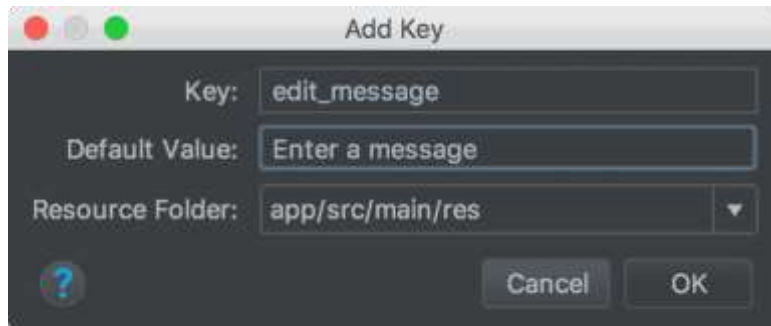
baselines. So click the button, and then click **Edit Baseline** , which appears in the design editor directly below the selected view. The baseline anchor appears inside the button. Click-and-hold on this anchor and then drag it to the baseline anchor that appears in the text box.

### Change the UI strings

To preview the UI, click **Select Design Surface**  in the toolbar and select **Design**. Notice that the text input is pre-filled with "Name" and the button is labeled "Button." So now you'll change these strings.

1. Open the **Project** window and then open **app > res > values > strings.xml**.  
This is a [string resources](#) file where you should specify all your UI strings. Doing so allows you to manage all UI strings in a single location, which makes it easier to find, update, and localize (compared to hard-coding strings in your layout or app code).
2. Click **Open editor** at the top of the editor window. This opens the [Translations Editor](#), which provides a simple interface for adding and editing your default strings, and helps keep all your translated strings organized.




3. Click **Add Key**  to create a new string as the "hint text" for the text box.



**Figure 7.** The dialog to add a new string

- In the **Add Key** dialog, enter "edit\_message" for the key name.
  - Enter "Enter a message" for the value.
  - Click **OK**.
4. Add another key named "button\_send" with a value of "Send".

Now you can set these strings for each view. So return to the layout file by clicking **activity\_main.xml** in the tab bar, and add the strings as follows:

- Click the text box in the layout and, if the **Attributes** window isn't already visible on the right, click **Attributes**  on the right sidebar.
- Locate the **text** property (currently set to "Name") and delete the value.
- Locate the **hint** property and then click **Pick a Resource**  to the right of the text box. In the dialog that appears, double-click on **edit\_message** from the list.
- Now click the button in the layout, locate the **text** property (currently set to "Button"), click **Pick a Resource** , and then select **button\_send**.

### Make the text box size flexible

To create a layout that's responsive to different screen sizes, you'll now make the text box stretch to fill all remaining horizontal space (after accounting for the button and margins).

Before you continue, click **Show**  in the toolbar and select **Blueprint**.

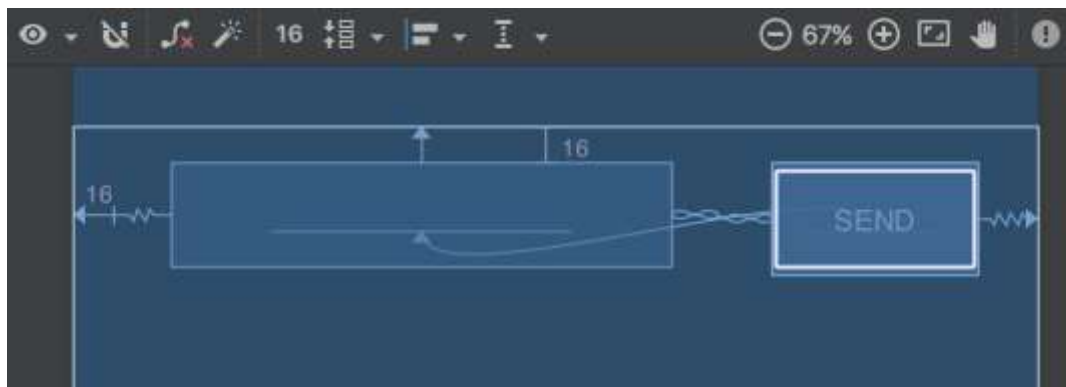


Figure 8. The result of choosing **Create Horizontal Chain**

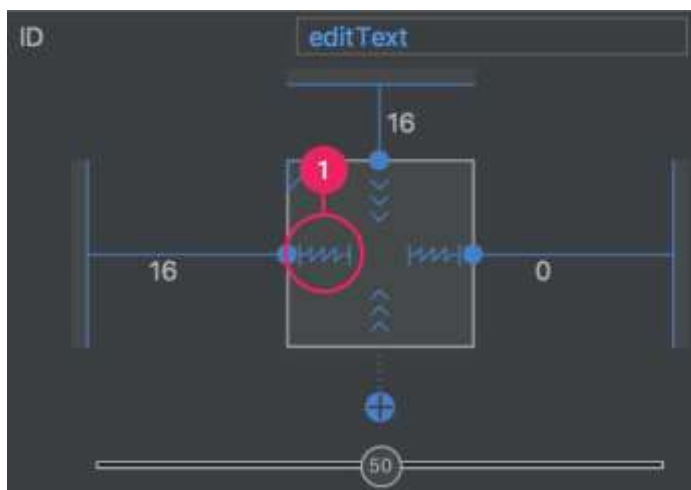


Figure 9. Click to change the width to **Match Constraints**



**Figure 10.** The text box now stretches to fill the remaining space

1. Select both views (click one, hold Shift, and click the other), and then right-click either view and select **Chain > Create Horizontal Chain**. The layout should appear as shown in figure 8.  
A [chain](#) is a bidirectional constraint between two or more views that allows you to lay out the chained views in unison.
2. Select the button and open the **Attributes** window. Using the view inspector at the top of the **Attributes** window, set the right margin to 16.
3. Now click the text box to view its attributes. Click the width indicator twice so that it is set to **Match Constraints**, as indicated by callout in figure 9.

"Match constraints" means that the width expands to meet the definition of the horizontal constraints and margins. Therefore, the text box stretches to fill the horizontal space (after accounting for the button and all margins).

Now the layout is done and should appear as shown in figure 10.

## Start another activity

### Respond to the send button

Add a method to the `MainActivity` class that's called by the button as follows:

1. In the file `app > java > com.example.myfirstapp > MainActivity`, add the `sendMessage()` method stub as shown below:

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    /** Called when the user taps the Send button */
    public void sendMessage(View view) {
        // Do something in response to button
    }
}
```

2. You may see an error because Android Studio cannot resolve the `View` class used as the method argument. So click to place your cursor on the `View` declaration, and then perform a Quick Fix by pressing `Alt + Enter` (If a menu appears, select **Import class.**)
3. Now return to the `activity_main.xml` file to call this method from the button:
  1. Click to select the button in the Layout Editor.
  2. In the **Attributes** window, locate the **onClick** property and select **sendMessage [MainActivity]** from the drop-down list.



### Build an Intent

An [Intent](#) is an object that provides runtime binding between separate components, such as two activities. The [Intent](#) represents an app's "intent to do something." You can use intents for a wide variety of tasks, but in this lesson, your intent starts another activity.

In `MainActivity`, add the `EXTRA_MESSAGE` constant and the `sendMessage()` code, as shown here:

```
public class MainActivity extends AppCompatActivity {  
    public static final String EXTRA_MESSAGE =  
    "com.example.myfirstapp.MESSAGE";  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.activity_main);  
    }  
  
    /** Called when the user taps the Send button */  
    public void sendMessage(View view) {  
        Intent intent = new Intent(this, DisplayMessageActivity.class);  
        EditText editText = (EditText) findViewById(R.id.editText);  
        String message = editText.getText().toString();  
        intent.putExtra(EXTRA_MESSAGE, message);  
        startActivity(intent);  
    }  
}
```

Here's what's going on in `sendMessage()`:

- The [Intent](#) constructor takes two parameters:
- A [Context](#) as its first parameter (this is used because the [Activity](#) class is a subclass of [Context](#))
- The [Class](#) of the app component to which the system should deliver the [Intent](#) (in this case, the activity that should be started).
- The `putExtra()` method adds the `EditText`'s value to the intent. An [Intent](#) can carry data types as key-value pairs called *extras*. Your key is a public constant `EXTRA_MESSAGE` because the next activity uses the key to retrieve the text value. It's a good practice to define keys for

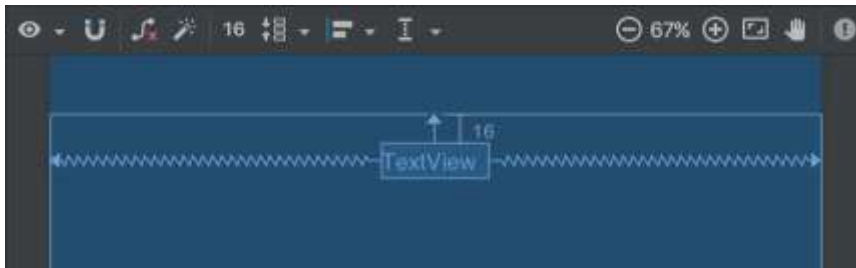
intent extras using your app's package name as a prefix. This ensures the keys are unique, in case your app interacts with other apps.

- The `startActivity()` method starts an instance of the `DisplayMessageActivity` specified by the `Intent`. Now you need to create that class.

### Create the second activity


1. In the **Project** window, right-click the **app** folder and select **New > Activity > Empty Activity**.
2. In the **Configure Activity** window, enter "DisplayMessageActivity" for **Activity Name** and click **Finish** (leave all other properties set to the defaults).

### Add a text view



**Figure 1.** The text view centered at the top of the layout

The new activity includes a blank layout file, so now you'll add a text view where the message will appear.

1. Open the file **app > res > layout > activity\_display\_message.xml**.
2. Click **Turn On Autoconnect**  in the toolbar (it should then be enabled, as shown in figure 1).
3. In the **Palette** window, click **Text** and then drag a **TextView** into the layout—drop it near the top of the layout, near the center so it snaps to the vertical line that appears. Autoconnect adds left and right constraints to place the view in the horizontal center.

---

### Display the message

---

Now you will modify the second activity to display the message that was passed by the first activity.

1. In `DisplayMessageActivity`, add the following code to the `onCreate()` method:

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

    // Get the Intent that started this activity and extract the
    string
    Intent intent = getIntent();
    String message =
intent.getStringExtra(MainActivity.EXTRA_MESSAGE);

    // Capture the layout's TextView and set the string as its text
    TextView textView = findViewById(R.id.textview);
    textView.setText(message);
}
```

### Add up navigation

---

Each screen in your app that is not the main entry point (all screens that are not the "home" screen) should provide navigation so the user can return to the logical parent screen in the app hierarchy by tapping the Up button in the [app bar](#).

All you need to do is declare which activity is the logical parent in the `AndroidManifest.xml` file. So open the file at `app > manifests > AndroidManifest.xml`, locate the `<activity>` tag for `DisplayMessageActivity` and replace it with the following:

```
<activity android:name=".DisplayMessageActivity"
    android:parentActivityName=".MainActivity">
    <!-- The meta-data tag is required if you support API level 15 and
    lower -->
    <meta-data
```

```
android:name="android.support.PARENT_ACTIVITY"  
android:value=".MainActivity" />  
</activity>
```

#### Reference

<https://developer.android.com/training/basics/firstapp/creating-project>

<https://developer.android.com/guide/topics/resources/providing-resources>

<https://developer.android.com/training/basics/firstapp/building-ui>

<https://developer.android.com/training/basics/firstapp/starting-activity#java>

<https://developer.android.com/guide/topics/ui/declaring-layout>