|  |  |  |
| --- | --- | --- |
|  | **2, LEBUH ACHEH, GEORGE TOWN**  **10300 GEORGE TOWN**  **PULAU PINANG**  **INFORMATION SHEET** | |
| **PROGRAM’S CODE & NAME** | J620-002-4:2020 FRONT-END SOFTWARE DEVELOPMENT | |
| **LEVEL** | FOUR (4) | |
| **COMPETENCY UNIT NO. AND TITLE** | J620-002-4:2020-C04 MOBILE APPLICATION WITH THIRD PARTY API DEVELOPMENT | |
| **WORK ACTIVITIES NO. AND STATEMENT** | 1. CREATE MOBILE APP DESIGN MOCK-UP ELEMENTS. 2. PLAN MOBILE APP DESIGN STRUCTURE. 3. TRANSFORM MOCK-UP TO MOBILE APP. 4. INTEGRATE MOBILE APP WITH DATA SOURCE. 5. VERIFY SUCCESSFUL API INTEGRATION 6. **VERIFY DEVELOPED MOBILE APP.** 7. VERIFY MOBILE APP ACCESSIBLE GLOBALLY. | |
| **CODE NO.** | J620-002-4:2020-C04/IS(11/15) | Page: 1 of 29 |

**TITLE**:

**CONNECTING TO SIMULATOR AND DEVICE**

**PURPOSE**:

This information sheet is intended to provide insight and knowledge to trainees with regards to the fundamentals of simulator and device.

**INFORMATION:**

This information sheet provides useful notes and explanations on fundamental concepts of simulator and device.

# **BUILDING YOUR FIRST APP**

This section describes how to build a simple Android app. First, you learn how to create a "Hello, World!" project with Android Studio and run it. Then, you create a new interface for the app that takes user input and switches to a new screen in the app to display it.

Before you start, there are two fundamental concepts that you need to understand about Android apps: how they provide multiple entry points, and how they adapt to different devices.

## Apps provide multiple entry points

Android apps are built as a combination of components that can be invoked individually. For example, an activity is a type of app component that provides a user interface (UI).

The "main" activity starts when the user taps your app's icon. You can also direct the user to an activity from elsewhere, such as from a notification or even from a different app.

Other components, such as WorkManager, allow your app to perform background tasks without a UI.

After you build your first app, you can learn more about the other app components at Application fundamentals.

## Apps adapt to different devices

Android allows you to provide different resources for different devices. For example, you can create different layouts for different screen sizes. The system determines which layout to use based on the screen size of the current device.

If any of your app's features need specific hardware, such as a camera, you can query at runtime whether the device has that hardware or not, and then disable the corresponding features if it doesn't. You can specify that your app requires certain hardware so that Google Play won't allow the app to be installed on devices without them.

After you build your first app, learn more about device configurations at Device compatibility overview.

Where to go from here

With these two basic concepts in mind, you have two options. If you prefer staying in the main documentation, which makes it easy to branch off to other topics to learn more about specific aspects of building an app, you can proceed to the next lesson to build your first app. However, if you like to follow step-by-step tutorials that explain every step from beginning to end, then consider the Android Basics in Kotlin course.

## Creating an android project

This lesson shows you how to create a new Android project with Android Studio, and it describes some of the files in the project.

To create your new Android project, follow these steps:

Install the latest version of Android Studio.

In the Welcome to Android Studio window, click Create New Project.

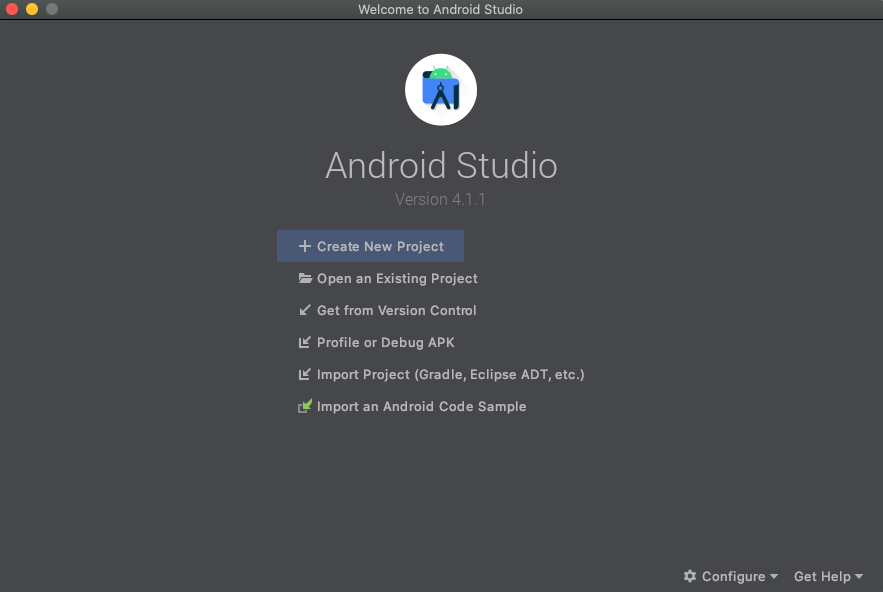


Figure 1: Android Studio Welcome Screen

If you have a project already opened, select File > New > New Project.

In the Select a Project Template window, select Empty Activity and click Next.

In the Configure your project window, complete the following:

1. Enter "My First App" in the Name field.
2. Enter "com.example.myfirstapp" in the Package name field.
3. If you'd like to place the project in a different folder, change its Save location.
4. Select either Java or Kotlin from the Language drop-down menu.
5. Select the lowest version of Android you want your app to support in the Minimum SDK field.
6. If your app will require legacy library support, mark the Use legacy android.support libraries checkbox.
7. Leave the other options as they are.

Click Finish.

After some processing time, the Android Studio main window appears.

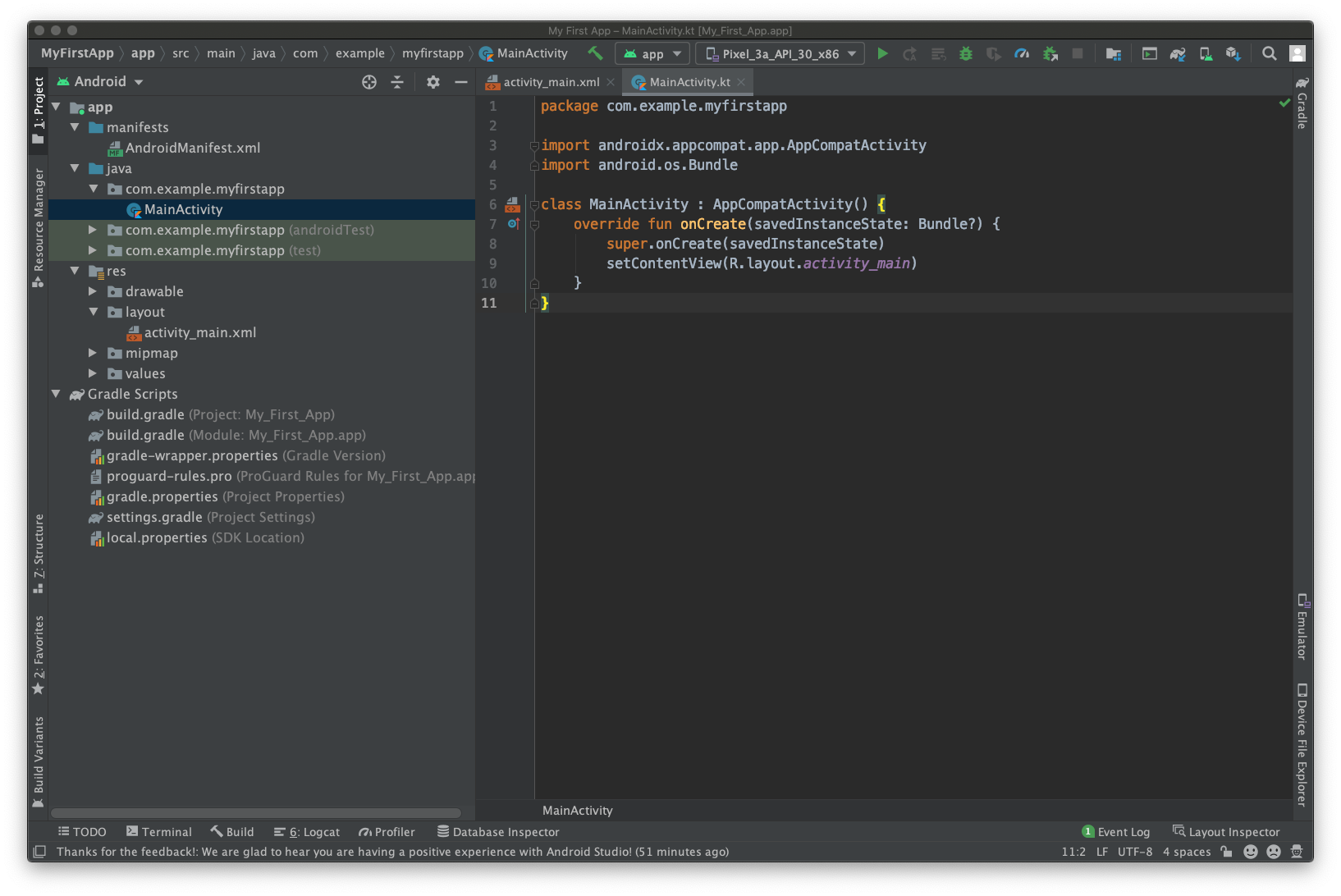


Figure 2: Android Studio Main Window

Now take a moment to review the most important files.

First, be sure the Project window is open (select View > Tool Windows > Project) and the Android view is selected from the drop-down list at the top of that window. You can then see the following files:

**app > java > com.example.myfirstapp > MainActivity**

This is the main activity. It's the entry point for your app. When you build and run your app, the system launches an instance of this Activity and loads its layout.

**app > res > layout > activity\_main.xml**

This XML file defines the layout for the activity's user interface (UI). It contains a TextView element with the text "Hello, World!"

**app > manifests > AndroidManifest.xml**

The manifest file describes the fundamental characteristics of the app and defines each of its components.

**Gradle Scripts > build.gradle**

There are two files with this name: one for the project, "Project: My\_First\_App," and one for the app module, "Module: My\_First\_App.app." Each module has its own build.gradle file, but this project currently has just one module. Use each module's build.gradle file to control how the Gradle plugin builds your app. For more information about this file, see Configure your build.

# **RUNNING YOUR APP**

## Running on a real device

Set up your device as follows:

1. Connect your device to your development machine with a USB cable. If you developed on Windows, you might need to install the appropriate USB driver for your device.
2. Perform the following steps to enable USB debugging in the Developer options window:
3. Open the Settings app.
4. If your device uses Android v8.0 or higher, select System. Otherwise, proceed to the next step.
5. Scroll to the bottom and select About phone.
6. Scroll to the bottom and tap Build number seven times.
7. Return to the previous screen, scroll to the bottom, and tap Developer options.
8. In the Developer options window, scroll down to find and enable USB debugging.

Run the app on your device as follows:

1. In Android Studio, select your app from the run/debug configurations drop-down menu in the toolbar.
2. In the toolbar, select the device that you want to run your app on from the target device drop-down menu.

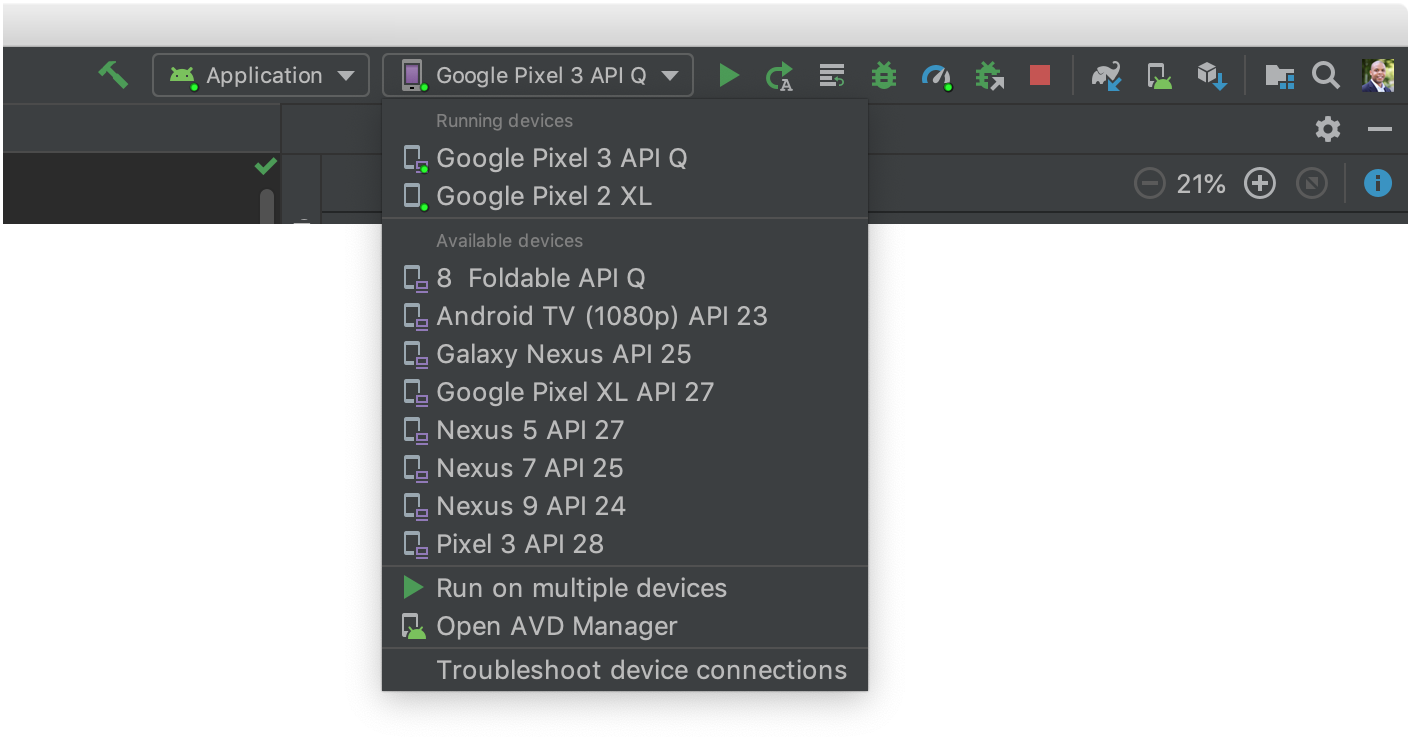


Figure 3: Target Device Drop-Down Menu

1. Click Run.

Android Studio installs your app on your connected device and starts it. You now see "Hello, World!" displayed in the app on your device.

## Running on an emulator

Run the app on an emulator as follows:

1. In Android Studio, create an Android Virtual Device (AVD) that the emulator can use to install and run your app.
2. In the toolbar, select your app from the run/debug configurations drop-down menu.
3. From the target device drop-down menu, select the AVD that you want to run your app on.

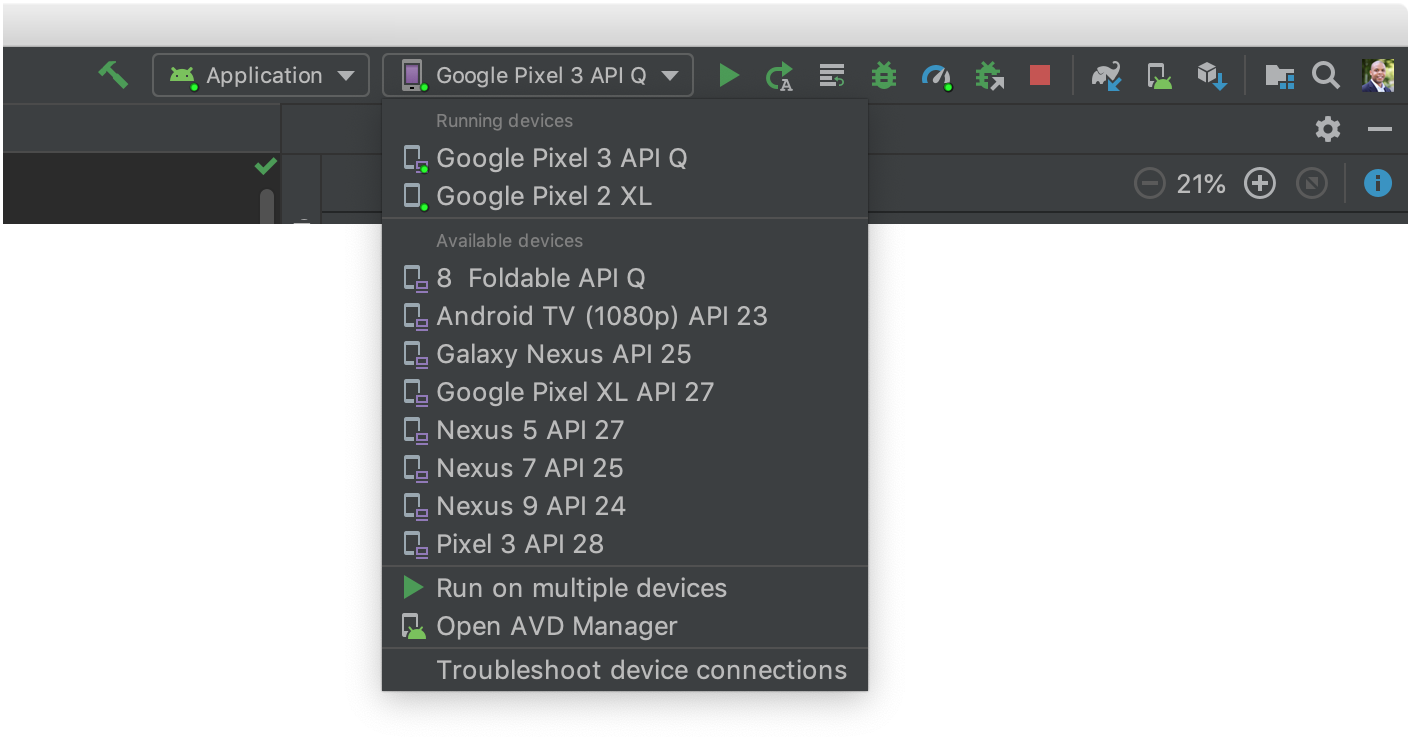


Figure 4: Target Device Drop-Down Menu

1. Click Run.

Android Studio installs the app on the AVD and starts the emulator. You now see "Hello, World!" displayed in the app.

## Building a simple user interface

In this lesson, you learn how to use the Android Studio Layout Editor to create layout that includes a text box and a button. This sets up the next lesson, where you learn how to make the app send the content of the text box to another activity when the button is tapped.

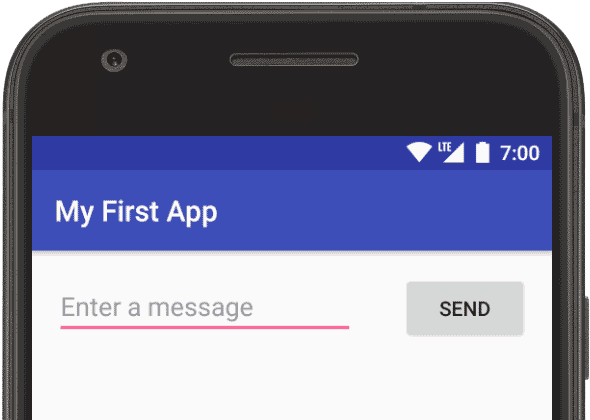
****

Figure 5: Screenshot of The Final Layout

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

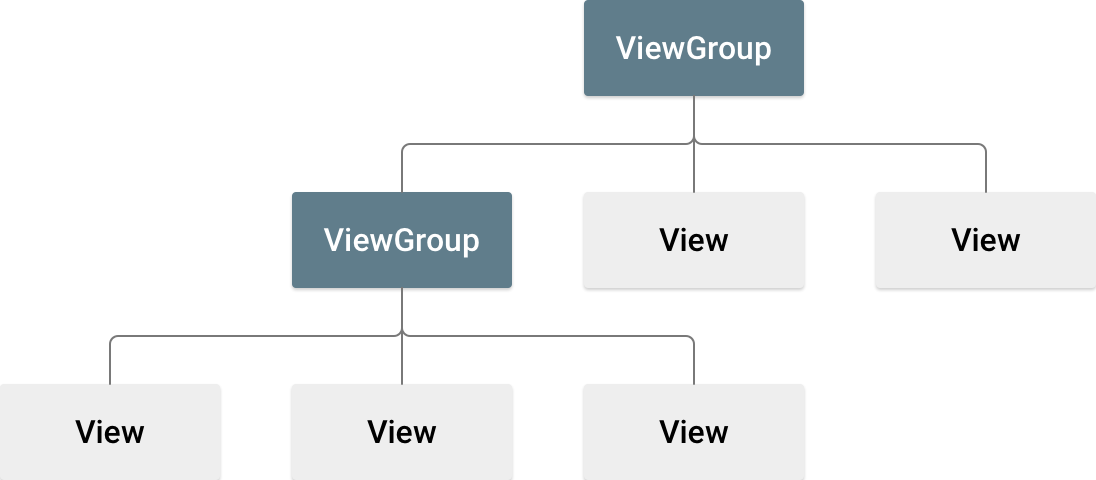


Figure 6: Illustration of how ViewGroup objects form branches in the layout and contain View objects

Android provides an XML vocabulary for ViewGroup and View classes, so most of your UI is defined in XML files. However, rather than teach you to write XML, this lesson shows you how to create a layout using Android Studio's Layout Editor. The Layout Editor writes the XML for you as you drag and drop views to build your layout.

1. Open the Layout Editor

To get started, set up your workspace as follows:

1. In the Project window, open app > res > layout > activity\_main.xml.
2. To make room for the Layout Editor, hide the Project window. To do so, select View > Tool Windows > Project, or just click Project on the left side of the Android Studio screen.
3. If your editor shows the XML source, click the Design tab at the top right of the window.
4. Click (Select Design Surface) and select Blueprint.
5. Click (View Options) in the Layout Editor toolbar and make sure that Show All Constraints is checked.
6. Make sure Autoconnect is off. A tooltip in the toolbar displays (Enable Autoconnection to Parent) when Autoconnect is off.
7. Click (Default Margins) in the toolbar and select 16. If needed, you can adjust the margins for each view later.
8. Click (Device for Preview) in the toolbar and select 5.5, 1440 × 2560, 560 dpi (Pixel XL).

Your Layout Editor now looks as shown in figure 3.

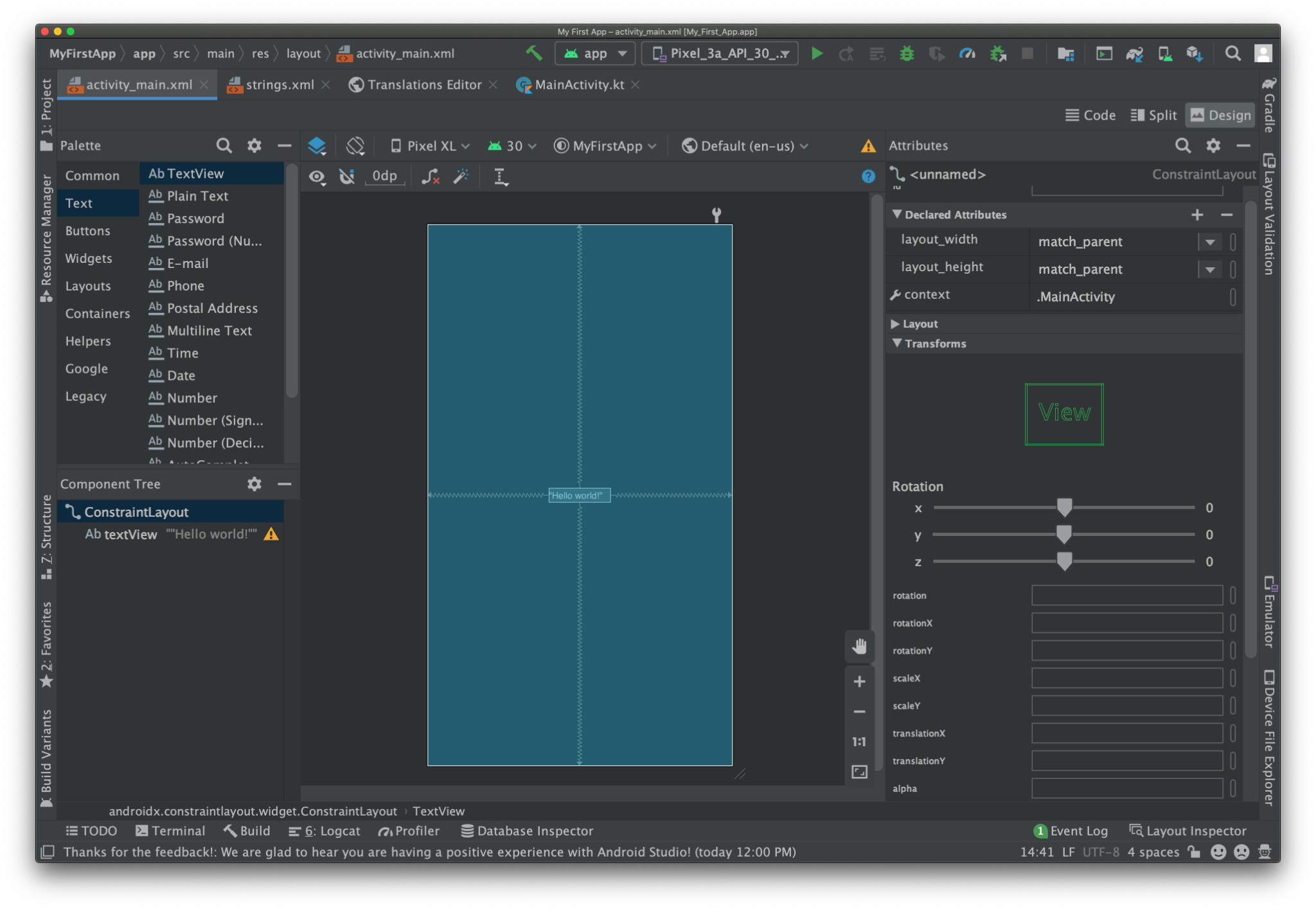


Figure 7: The Layout Editor showing activity\_main.xml

For additional information, see Introduction to the Layout Editor.

The Component Tree panel on the bottom left shows the layout's hierarchy of views. In this case, the root view is a ConstraintLayout, which contains 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. This type of layout avoids the need for nested layouts. A nested layout, which is a layout inside a layout, as shown in figure 2, can increase the time required to draw the UI.

For example, you can declare the following layout, which is shown in figure 8:

1. View A appears 16 dp from the top of the parent layout.
2. View A appears 16 dp from the left of the parent layout.
3. View B appears 16 dp to the right of view A.
4. View B is aligned to the top of view A.

In the following sections, you'll build a layout similar to the layout in figure 8.

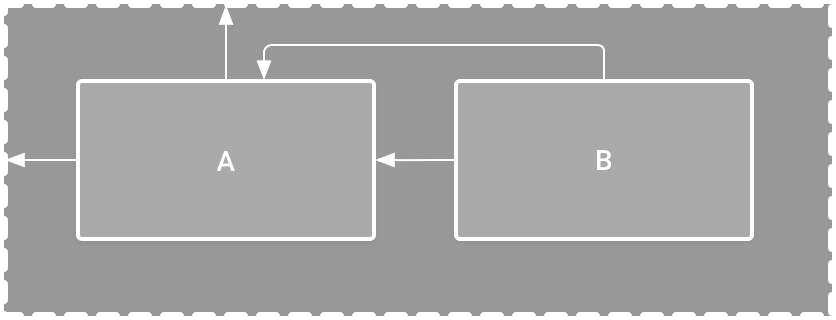


Figure 8: Illustration of two views positioned inside ConstraintLayout

1. Add a text box

Follow these steps to add a text box:

1. First, you need to remove what's already in the layout. Click TextView in the Component Tree panel and then press the Delete key.
2. In the Palette panel, click Text to show the available text controls.
3. Drag the 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 square handles to resize the view on each corner, and the circular constraint anchors on each side. For better control, you might want to zoom in on the editor. To do so, use the Zoom buttons in the Layout Editor toolbar.
5. Click and hold the anchor on the top side, drag it up until it snaps to the top of the layout, and then release it. That's a constraint: it constrains the view within the default margin that was set. In this case, you set it to 16 dp from the top of the layout.
6. Use the same process to create a constraint from the left side of the view to the left side of the layout.

The result should look as shown in figure 5.

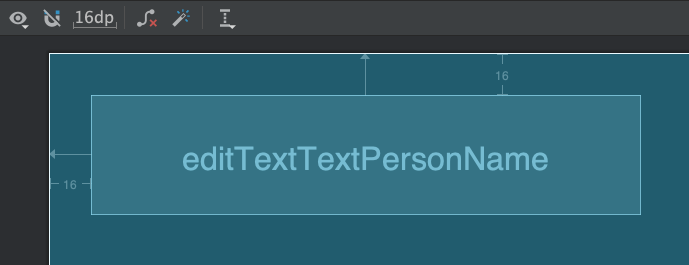


Figure 9: The text box is constrained to the top and left of the parent layout

1. Add a button
2. In the Palette panel, click Buttons.
3. Drag the Button widget into the design editor and drop it near the right side.
4. Create a constraint from the left side of the button to the right side of the text box.
5. To constrain the views in a horizontal alignment, create a constraint between the text baselines. To do so, right-click the button and then select Show Baseline Show Baseline action in Layout Editor. The baseline anchor appears inside the button. Click and hold this anchor, and then drag it to the baseline anchor that appears in the adjacent text box.

The result should look as shown in figure 6.

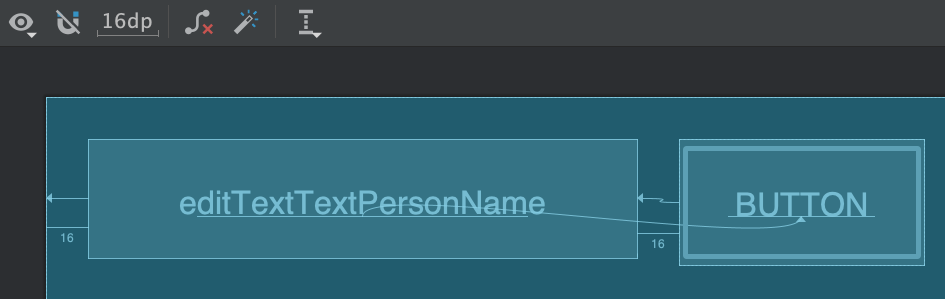


Figure 10: The button is constrained to the right side of the text box and its baseline

1. Change the UI strings

To preview the UI, click (Select Design Surface) in the toolbar and select Design. Notice that the text input and button label are set to default values.

Follow these steps to change the UI strings:

1. Open the Project window and then open app > res > values > strings.xml.

This is a string resources file, where you can specify all of your UI strings. It allows you to manage all of your UI strings in a single location, which makes them easier to find, update, and localize.

1. Click Open editor at the top of the window. This opens the Translations Editor, which provides a simple interface to add and edit your default strings. It also helps you keep all of your translated strings organized.
2. Click (Add Key) to create a new string as the "hint text" for the text box. At this point, the window shown in figure 7 opens.

In the Add Key dialog box, complete the following steps:

* Enter "edit\_message" in the Key field.
* Enter "Enter a message" in the Default Value field.
* Click OK.

1. Add another key named "button\_send" with a value of "Send".

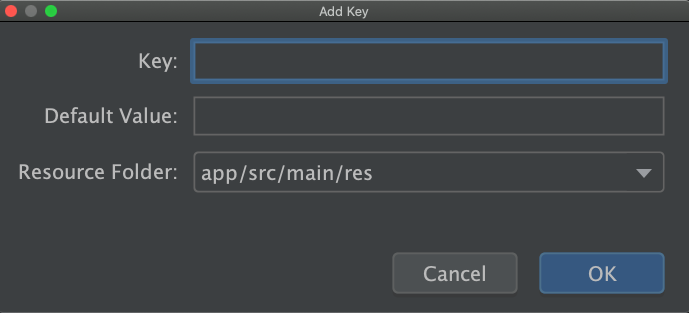


Figure 11: The dialog to add a new string

Now you can set these strings for each view. To return to the layout file, click activity\_main.xml in the tab bar. Then, add the strings as follows:

1. Click the text box in the layout. If the Attributes window isn't already visible on the right, click Attributes on the right sidebar.
2. Locate the text property, which is currently set to "Name," and delete the value.
3. Locate the hint property and then click (Pick a Resource), which is to the right of the text box. In the dialog that appears, double-click edit\_message from the list.
4. Click the button in the layout and locate its text property, which is currently set to "Button." Then, click (Pick a Resource) and select button\_send.
5. Make the text box size flexible

To create a layout that's responsive to different screen sizes, you need to make the text box stretch to fill all the horizontal space that remains after the button and margins are accounted for.

Before you continue, click (Select Design Surface) in the toolbar and select Blueprint.

To make the text box flexible, follow these steps:

Select both views. To do so, click one, hold Shift, then click the other, and then right-click either one and select Chains > Create Horizontal Chain. The layout then appears 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.

1. Select the button and open the Attributes window. Then, use the Constraint Widget to set the right margin to 16 dp.
2. Click the text box to view its attributes. Then, click the width indicator twice so it's set to a jagged line (Match Constraints), as indicated by callout 1 in figure 9.
3. 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 that remains after the button and all the margins are accounted for.

If your layout didn't turn out as expected, click See the final layout XML below to see what your XML should look like. Compare it to what you see in the Code tab. If your attributes appear in a different order, that's okay.

Now the layout is done.

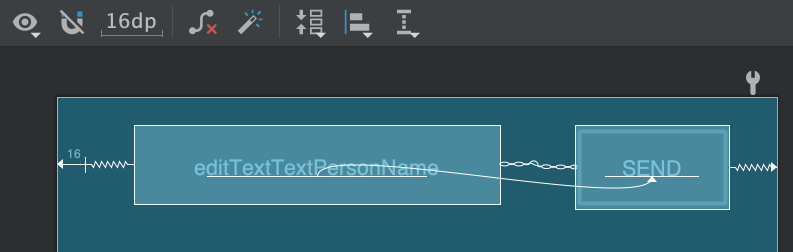


Figure 12: The result of choosing Create Horizontal Chain

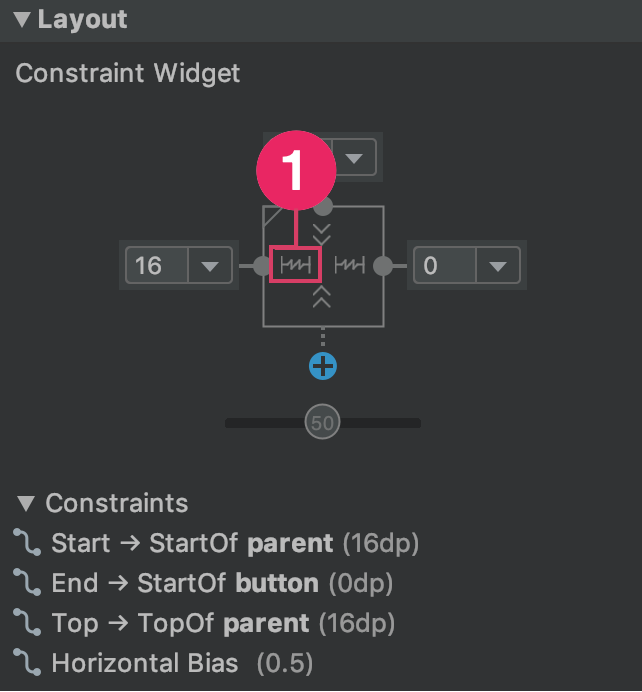


Figure 13: The Width to Match Constraints

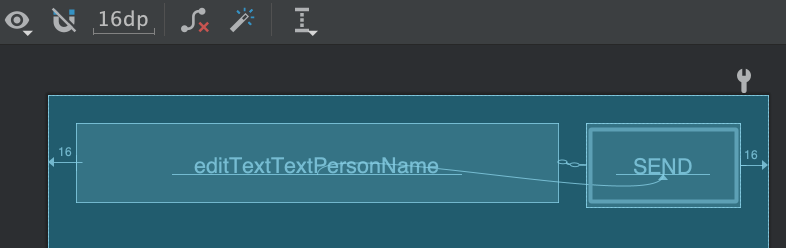


Figure 14: The text box now stretches to fill the remaining space

1. Run the app

If your app is already installed on the device from the previous lesson, simply click (Apply Changes) in the toolbar to update the app with the new layout. Or click Run 'app' to install and run the app.

## Add on activities

1. Respond to the Send button

Follow these steps to add a method to the MainActivity class that's called when the Send button is tapped:

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

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

}

/\*\* Called when the user taps the Send button \*/

fun sendMessage(view: View) {

// Do something in response to button

}

}

You might see an error because Android Studio cannot resolve the View class used as the method argument. To clear the error, click the View declaration, place your cursor on it, and then press Alt+Enter, or Option+Enter on a Mac, to perform a Quick Fix. If a menu appears, select Import class.

1. Return to the activity\_main.xml file to call the method from the button:

* Select the button in the Layout Editor.
* In the Attributes window, locate the onClick property and select sendMessage [MainActivity] from its drop-down list.

Now when the button is tapped, the system calls the sendMessage() method.

Take note of the details in this method. They're required for the system to recognize the method as compatible with the android:onClick attribute. Specifically, the method has the following characteristics:

* Public access.
* A void or, in Kotlin, an implicit unit return value.
* A View as the only parameter. This is the View object you clicked at the end of Step 1.

Next, fill in this method to read the contents of the text field and deliver that text to another activity.

1. 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:

const val EXTRA\_MESSAGE = "com.example.myfirstapp.MESSAGE"

class MainActivity : AppCompatActivity() {

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_main)

}

/\*\* Called when the user taps the Send button \*/

fun sendMessage(view: View) {

val editText = findViewById<EditText>(R.id.editText)

val message = editText.text.toString()

val intent = Intent(this, DisplayMessageActivity::class.java).apply {

putExtra(EXTRA\_MESSAGE, message)

}

startActivity(intent)

}

}

Expect Android Studio to encounter Cannot resolve symbol errors again. To clear the errors, press Alt+Enter, or Option+Return on a Mac. You should end up with the following imports:

import androidx.appcompat.app.AppCompatActivity

import android.content.Intent

import android.os.Bundle

import android.view.View

import android.widget.EditText

An error still remains for DisplayMessageActivity, but that's okay. You fix it in the next section.

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

1. The Intent constructor takes two parameters, a Context and a Class.
2. The Context parameter is used first because the Activity class is a subclass of Context.
3. The Class parameter of the app component, to which the system delivers the Intent, is, in this case, the activity to start.
4. The putExtra() method adds the value of EditText to the intent. An Intent can carry data types as key-value pairs called extras.
5. 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 with your app's package name as a prefix. This ensures that the keys are unique, in case your app interacts with other apps.
6. The startActivity() method starts an instance of the DisplayMessageActivity that's specified by the Intent. Next, you need to create that class.
7. Create the second activity

To create the second activity, follow these steps:

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. Leave all other properties set to their defaults and click Finish.

Android Studio automatically does three things:

1. Creates the DisplayMessageActivity file.
2. Creates the layout file activity\_display\_message.xml, which corresponds with the DisplayMessageActivity file.
3. Adds the required <activity> element in AndroidManifest.xml.

If you run the app and tap the button on the first activity, the second activity starts but is empty. This is because the second activity uses the empty layout provided by the template.

1. Add a text view

The new activity includes a blank layout file. Follow these steps to add a text view to where the message appears:

1. Open the file app > res > layout > activity\_display\_message.xml.
2. Click Enable Autoconnection to Parent in the toolbar. This enables Autoconnect. See figure 1.
3. In the Palette panel, click Text, drag a TextView into the layout, and drop it near the top-center of the layout so that it snaps to the vertical line that appears. Autoconnect adds left and right constraints in order to place the view in the horizontal center.
4. Create one more constraint from the top of the text view to the top of the layout, so that it appears as shown in figure 1.

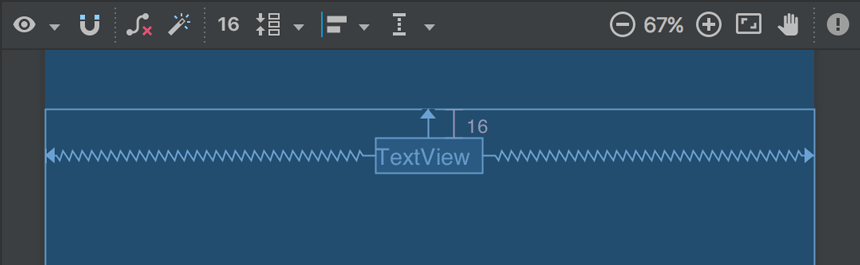


Figure 15: The text view centered at the top of the layout

Optionally, you can make some adjustments to the text style if you expand textAppearance in the Common Attributes panel of the Attributes window, and change attributes such as textSize and textColor

1. Display the message

In this step, you 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 fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity\_display\_message)

// Get the Intent that started this activity and extract the string

val message = intent.getStringExtra(EXTRA\_MESSAGE)

// Capture the layout's TextView and set the string as its text

val textView = findViewById<TextView>(R.id.textView).apply {

text = message

}

}

1. Press Alt+Enter, or Option+Return on a Mac, to import these other needed classes:

import androidx.appcompat.app.AppCompatActivity

import android.content.Intent

import android.os.Bundle

import android.widget.TextView

1. Add upward navigation

Each screen in your app that's not the main entry point, which are all the screens that aren't the home screen, must provide navigation that directs the user to the logical parent screen in the app's hierarchy. To do this, add an Up button in the app bar.

To add an Up button, you need to declare which activity is the logical parent in the AndroidManifest.xml file. 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>

The Android system now automatically adds the Up button to the app bar.

1. Run the app

Click Apply Changes in the toolbar to run the app. When it opens, type a message in the text field and tap Send to see the message appear in the second activity.



Figure 16: App opened, with text entered on the left screen and displayed on the right

.

That's it, you've built your first Android app!

One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app.

Each Android app lives in its own security sandbox, protected by the following Android security features:

1. The Android operating system is a multi-user Linux system in which each app is a different user.
2. By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.
3. Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.
4. By default, every app runs in its own Linux process. The Android system starts the process when any of the app's components need to be executed, and then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

The Android system implements the *principle of least privilege*. That is, each app, by default, has access only to the components that it requires to do its work and no more. This creates a very secure environment in which an app cannot access parts of the system for which it is not given permission. However, there are ways for an app to share data with other apps and for an app to access system services:

1. It's possible to arrange for two apps to share the same Linux user ID, in which case they are able to access each other's files. To conserve system resources, apps with the same user ID can also arrange to run in the same Linux process and share the same VM. The apps must also be signed with the same certificate.
2. An app can request permission to access device data such as the device's location, camera, and Bluetooth connection. The user has to explicitly grant these permissions. For more information, see [Working with System Permissions](https://developer.android.com/training/permissions).

# **APP COMPONENTS**

App components are the essential building blocks of an Android app. Each component is an entry point **through** which the system or a user can enter your app. Some components depend on others.

There are four different types of app components:

* Activities
* Services
* Broadcast receivers
* Content providers

Each type serves a distinct purpose and has a distinct lifecycle that defines how the component is **created** and destroyed. The following sections describe the four types of app components.

**Activities**

An activity is the entry point for interacting with the user. It represents a single screen with a user interface. For example, an email app might have one activity that shows a list of new emails, another activity to compose an email, and another activity for reading emails. Although the activities work together to form a cohesive user experience in the email app, each one is independent of the others. As such, a different app can start any one of these activities if the email app allows it. For example, a camera app can start the activity in the email app that composes new mail to allow the user to share a picture. An activity facilitates the following key interactions between system and app:

* Keeping track of what the user currently cares about (what is on screen) to ensure that the system keeps running the process that is hosting the activity.
* Knowing that previously used processes contain things the user may return to (stopped activities), and thus more highly prioritize keeping those processes around.
* Helping the app handle having its process killed so the user can return to activities with their previous state restored.
* Providing a way for apps to implement user flows between each other, and for the system to coordinate these flows. (The most classic example here being share.)

You implement an activity as a subclass of the Activity class. For more information about the Activity class, see the Activities developer guide.

# **THE MANIFEST FILE**

Before the Android system can start an app component, the system must know that the component exists by reading the app's *manifest file*, AndroidManifest.xml. Your app must declare all its components in this file, which must be at the root of the app project directory.

The manifest does a number of things in addition to declaring the app's components, such as the following:

* Identifies any user permissions the app requires, such as Internet access or read-access to the user's contacts.
* Declares the minimum [API Level](https://developer.android.com/guide/topics/manifest/uses-sdk-element#ApiLevels) required by the app, based on which APIs the app uses.
* Declares hardware and software features used or required by the app, such as a camera, bluetooth services, or a multitouch screen.
* Declares API libraries the app needs to be linked against (other than the Android framework APIs), such as the [Google Maps library](http://code.google.com/android/add-ons/google-apis/maps-overview.html).

**QUESTIONS:**

1. App components are the essential building blocks of an Android app. List the Four (4) essential components of an Android application.

**Answer:**

* **Activities**
* **Services**
* **Broadcast receivers**
* **Content providers**

(4 Marks)

**REFERENCE:**

1. https://developer.android.com/training/basics/firstapp
2. https://developer.android.com/training/basics/firstapp/creating-project
3. https://developer.android.com/training/basics/firstapp/running-app#Emulator
4. https://developer.android.com/training/basics/firstapp/building-ui
5. https://developer.android.com/training/basics/firstapp/starting-activity