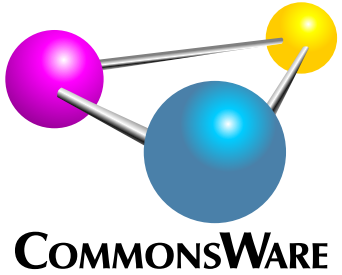
Version 0.4

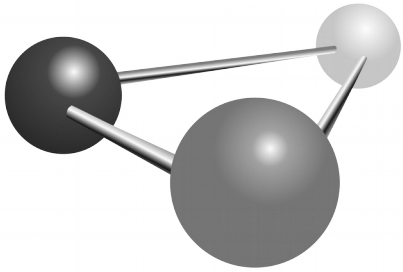
Exploring Android

Mark L. Murphy



Exploring Android

by Mark L. Murphy



COMMONSWARE

Exploring Android
by Mark L. Murphy

Printed in the United States of America.

Printing History:
February 2019: Version 0.4

Table of Contents

Headings formatted in ***bold-italic*** have changed since the last version.

Preface

Thanks!

First, thanks for your interest in Android app development! Android is the world's most popular operating system, but its value comes from apps written by developers like you.

Also, thanks for your interest in this book! Hopefully, it can help "spin you up" on how to create Android applications that meet your needs and those of your users.

And thanks for your interest in CommonsWare! The Warescription program makes this book and others available, to help developers like you craft the apps that your users need.

How the Book Is Structured

Many books — such as *Elements of Android Jetpack*, — present programming topics, showing you how to use different APIs, tools, and so on.

This book is different.

This book has you build an app from the beginning. Whereas traditional programming guides are focused on breadth and depth, this book is focused on "hands-on", guiding you through the steps to build the app. It provides a bit of details on the underlying concepts, but it relies on other resources — such as *Elements of Android Jetpack* — for the full explanation of those details. Instead, this book provides step-by-step instructions for building the app.

If you are the sort of person who "learns by doing", then this book is for you!

Second-Generation Book

Android app development can be divided into two generations:

- First-generation app development uses Java as the programming language and leverages the Android Support Library and the `android.arch` edition of the Architecture Components
- Second-generation app development more often uses Kotlin as the programming language and leverages AndroidX and the rest of Jetpack (which includes an AndroidX edition of the Architecture Components)

This book is a second-generation book. It will show you step-by-step how to build a Kotlin-based Android app, using AndroidX libraries.

Prerequisites

This book is targeted at developers starting out with Android app development.

You will want another educational resource to go along with this book. The book will cross-reference *Elements of Android Jetpack*, but you can use other programming guides as well. This book shows you each step for building an app, but you will need to turn to other resources for answers to questions like "why do we need to do X?" or "what other options do we have than Y?".

The app that you will build will be written in Kotlin, so you will need to have a bit of familiarity with that language. *Elements of Kotlin* covers this language and will be cross-referenced in a few places in this book.

Also, the app that you will create in this book works on Android 5.0+ devices and emulators. You will either need a suitable device or be in position to use the Android SDK emulator in order to build and run the app.

About the Updates

This book will be updated a few times per year, to reflect new advances with Android, the libraries used by the sample app, and the development tools.

If you obtained this book through the Warescription, you will be able to download updates as they become available, for the duration of your subscription period.

If you obtained this book through other channels... um, well, it's still a really nice book!

Each release has notations to show what is new or changed compared with the immediately preceding release:

- The Table of Contents shows sections with changes in bold-italic font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

And, there is the "What's New" section, just below this paragraph.

What's New in Version 0.4?

The book was rewritten to use Kotlin and the Jetpack! The app is the same app as before, just written using newer technologies.

Right now, this is a partial rewrite, roughly mirroring Version 0.1. The rest of the book will be rewritten and released in the upcoming months.

Along the way, the book was updated for Android Studio 3.3.1.

Note that due to the scope of the changes, changebars like you see here were suppressed in the rest of the book... except for those related to a bug in the original release of this book update.

Warescription

If you purchased the Warescription, read on! If you obtained this book from other channels, feel free to jump ahead.

The Warescription entitles you, for the duration of your subscription, to digital editions of this book and its updates, in PDF, EPUB, and Kindle (MOBI/KF8) formats. You also have access to other books that CommonsWare publishes during that subscription period, such as the aforementioned *Elements of Android Jetpack*. You also get access to first-generation Android books, such as the legendary *The Busy Coder's Guide to Android Development*.

Each subscriber gets personalized editions of all editions of each book. That way, your books are never out of date for long, and you can take advantage of new

material as it is made available.

However, you can only download the books while you have an active Warescription. There is a grace period after your Warescription ends: you can still download the book until the next book update comes out after your Warescription ends. After that, you can no longer download the book. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The CommonsBlog
2. The CommonsWare Twitter feed
3. The Warescription newsletter, which you can subscribe to off of your Warescription page
4. Just check back on the Warescription site every month or two

Subscribers also have access to other benefits, including:

- "Office hours" — online chats to help you get answers to your Android application development questions. You will find a calendar for these on your Warescription page.
- A Stack Overflow "bump" service, to get additional attention for a question that you have posted there that does not have an adequate answer.
- A discussion board for asking arbitrary questions about Android app development

Book Bug Bounty

Find a problem in the book? Let CommonsWare know!

Be the first to report a unique concrete problem in the current digital edition, and CommonsWare will extend your Warescription by six months as a bounty for helping CommonsWare deliver a better product.

By "concrete" problem, we mean things like:

1. Typographical errors
2. Sample applications that do not work as advertised, in the environment described in the book
3. Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Be sure to check the book's errata page, though, to see if your issue has already been reported. One coupon is given per email containing valid bug reports.

We appreciate hearing about "softer" issues as well, such as:

1. Places where you think we are in error, but where we feel our interpretation is reasonable
2. Places where you think we could add sample applications, or expand upon the existing material
3. Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code and Its License

The source code samples shown in this book are available for download from the book's GitHub repository. All of the Android projects are licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Creative Commons and the Four-to-Free (42F) Guarantee

Each CommonsWare book edition will be available for use under the Creative Commons Attribution-Noncommercial-ShareAlike 4.0 license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition,

starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 February 2023*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit the Creative Commons Web site

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

What We Are Building

By following the instructions in this book, you will build an Android app.

But first, let's see what the app is that you are building.

The Purpose

Everybody has stuff to do. Ever since we have had "digital assistants" — such as the venerable Palm line of PDAs – a common use has been for tracking tasks to be done. So-called "to-do lists" are a popular sort of app, whether on the Web, on the desktop, or on mobile devices.

The world has more than enough to-do list apps. Google themselves have published a long list of sample apps that use a to-do list as a way of exploring various GUI architectures.

So, let's build another one!

Ours is not a fork of Google's, but rather a "cleanroom" implementation of a to-do list with similar functionality.

The Core UI

There are three main screens that the user will spend time in: the roster of to-do items, a screen with details of a particular item, and a screen for either adding a new item or editing an existing one.

There is also an "about" screen for displaying information about the app.

The Roster

When initially launched, the app will show a roster of the recorded to-do items, if there are any. Hence, on the first run, it will show just an “empty view”, prompting the user to click the “add” action bar item to add a new item:

Figure 1: ToDo App, As Initially Launched, with No Items

Once there are some items in the database, the roster will show those items, in alphabetical order by description, with a checkbox indicating whether or not they have been completed:

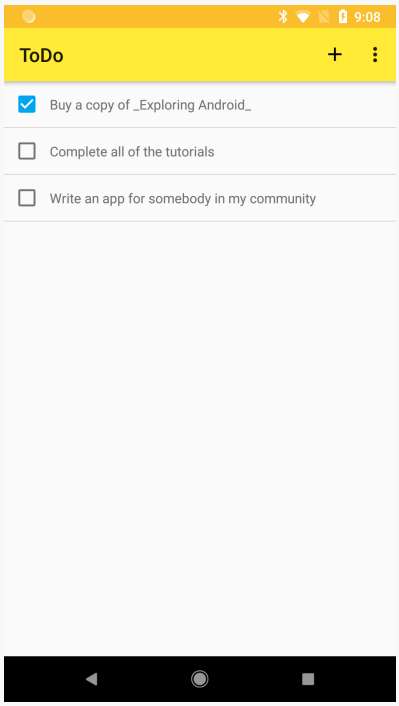


Figure 2: ToDo App, Showing Some Items

From here, the user can tap the checkbox to quickly mark an item as completed (or un-mark it if needed).

The Details

A simple tap on an item in the roster brings up the details screen:

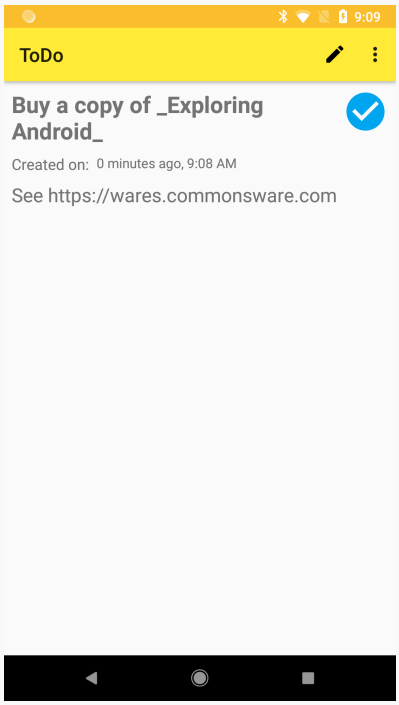


Figure 3: ToDo App, Showing a Completed Item

This just shows additional information about the item, including any notes the user entered to provide more detail than the simple description that gets shown in the roster. The checkmark icon will appear for completed items.

From here, the user can edit this item (via the “pencil” icon).

The Editor

The editor is a simple form, either to define a new to-do item or edit an existing one. If the user taps on the "add" action bar item from the roster, the editor will appear blank, and submitting the form will create a new to-do item. If the user taps on the "edit" (pencil) action bar item from the details screen, the editor will have the existing item's data, which can be altered and saved:

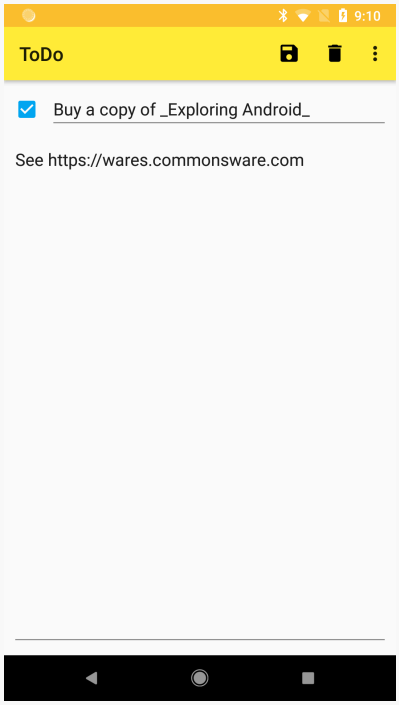


Figure 4: ToDo App, Editing a Completed Item

Clicking the "save" toolbar button will either add the new item or edit the item that the user requested to edit. For an edit, the "delete" toolbar button will be available and will allow the user to delete this specific item, after confirmation.

What We Are Missing

Completing all of the tutorials in this book will get you the screens that are shown above. It will also save those items in a database, so they will persist from run-to-run of the app. However, there will still be a lot of missing functionality:

- While the UI works well on phones, it is not optimized for larger-screen

devices, such as tablets
- You can only delete items one at a time; there is no multiple-selection option to delete them as a group
- There are no import or export options, or any way to use this information other than from within the app itself

Those features and more will be added in new tutorials added to upcoming versions of the book.

Phase One: Getting a GUI

Installing the Tools

First, let us get you set up with the pieces and parts necessary to build an Android app. Specifically, in this tutorial, we will set up Android Studio.

Step #1: Checking Your Hardware

Compiling and building an Android application, on its own, can be a hardware-intensive process, particularly for larger projects. Beyond that, your IDE and the Android emulator will stress your development machine further. Of the two, the emulator poses the bigger problem.

The more RAM you have, the better. 8GB or higher is a very good idea if you intend to use an IDE and the emulator together. If you can get an SSD for your data storage, instead of a conventional hard drive, that too can dramatically improve the IDE performance.

A faster CPU is also a good idea. The Android SDK emulator supports CPUs with multiple cores. However, other processes on your development machine will be competing with the emulator for CPU time, and so the faster your CPU is, the better off you will be. Ideally, your CPU has 2 to 4 cores, each 2.5GHz or faster at their base speed.

There are two types of emulator: x86 and ARM. These are the two major types of CPUs used for Android devices. You *really* want to be able to use the x86 emulator, as the ARM emulator is extremely slow. However, to do that, you need a CPU with certain features:

Development OS	CPU Manufacturer	CPU Requirements
mac OS	Intel	none — any modern Mac should work
Linux/ Windows	Intel	support for Intel VT-x, Intel EM64T (Intel 64), and Execute Disable (XD) Bit functionality
Linux	AMD	support for AMD Virtualization (AMD-V) and Supplemental Streaming SIMD Extensions 3 (SSSE3)
Windows 10 April 2018 or newer	AMD	support for Windows Hypervisor Platform (WHPX) functionality

If your CPU does not meet those requirements, you will want to have 1+ Android devices available to you, so that you can test on hardware.

Also, if you are running Windows or Linux, you need to ensure that your computer's BIOS is set up to support the Intel/AMD virtualization extensions. Unfortunately, many PC manufacturers disable this by default. The details of how to get into your BIOS settings will vary by PC, but usually it involves rebooting your computer and pressing some function key on the initial boot screen. In the BIOS settings, you are looking for references to "virtualization" (or perhaps "VT-x" for Intel). Enable them if they are not already enabled. macOS machines come with virtualization extensions pre-enabled, which is really nice of Apple.

Step #2: Install Android Studio

At the time of this writing, the current production version of Android Studio is 3.3.1, and this book covers that version. If you are reading this in the future, you may be on a newer version of Android Studio, and there may be some differences between what you have and what is presented here.

You have two major download options. You can get the latest shipping version of Android Studio from the Android Studio download page.

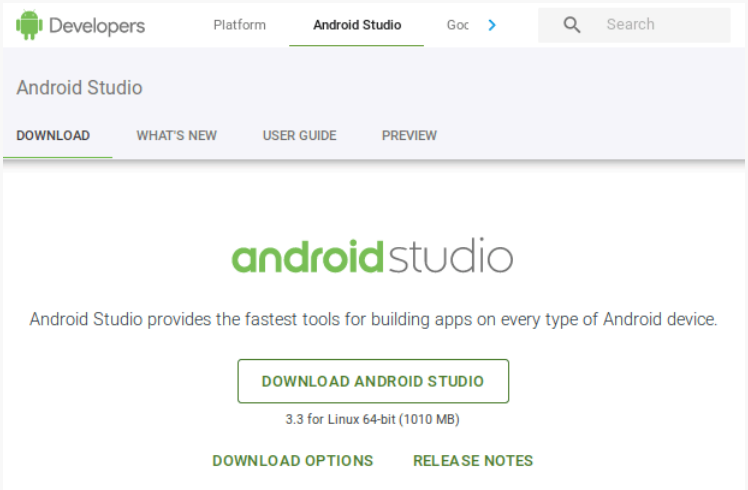


Figure 5: Android Studio Download Page

Or, you can download Android Studio 3.3.1 — the version used in this edition of this book — directly, for:

- Windows
- macOS
- Linux

Windows users can download a self-installing EXE, which will add suitable launch options for you to be able to start the IDE.

Mac users can download a DMG disk image and install it akin to other Mac software, dragging the Android Studio icon into the Applications folder.

Linux users (and power Windows users) can download a ZIP file, then unZIP it to some likely spot on your hard drive. Android Studio can then be run from the `studio` batch file or shell script from your Android Studio installation's `bin/` directory.

Step #3: Run Android Studio

When you first run Android Studio, you may be asked if you want to import settings from some other prior installation of Android Studio:

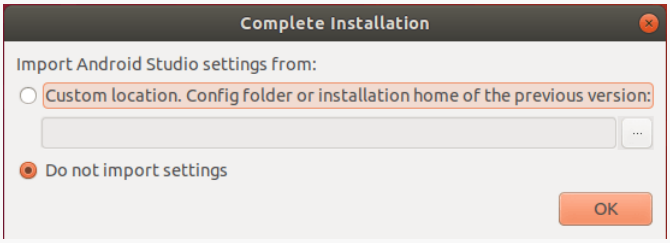


Figure 6: Android Studio First-Run Settings Migration Dialog

For most users, particularly those using Android Studio for the first time, the "Do not import settings" option is the correct choice to make.

Then, after a short splash screen, you may be presented with a "Data Sharing" dialog:

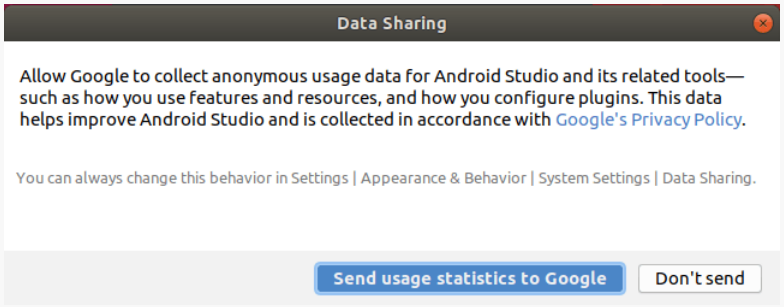


Figure 7: Android Studio Data Sharing Dialog

Click whichever button you wish.

Then, after a potentially long "Finding Available SDK Components" progress dialog, you will be taken to the Android Studio Setup Wizard:

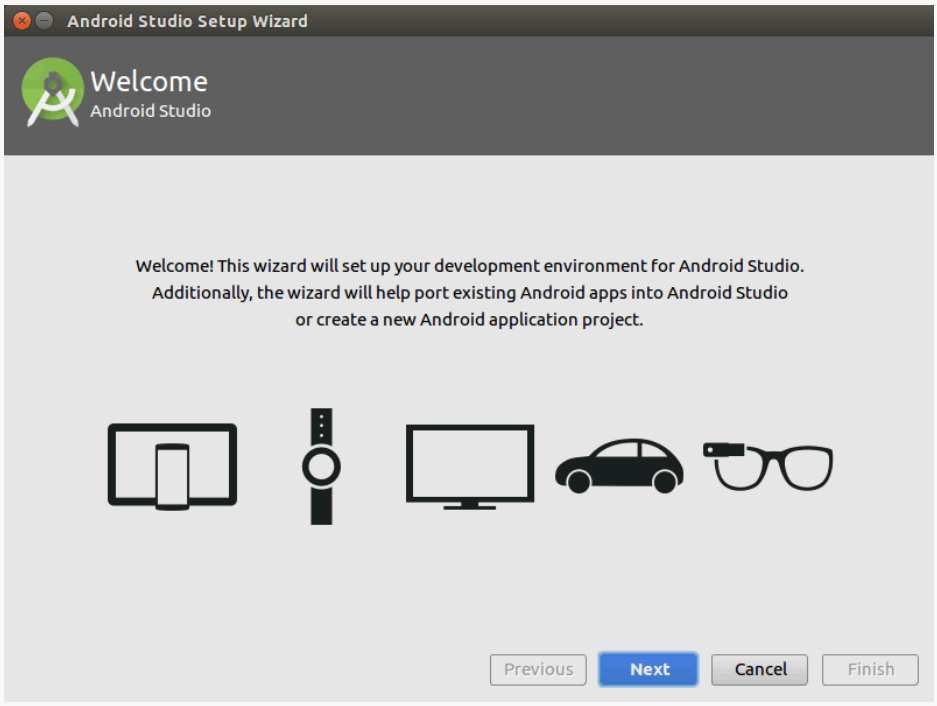


Figure 8: Android Studio Setup Wizard, First Page

Just click “Next” to advance to the second page of the wizard:

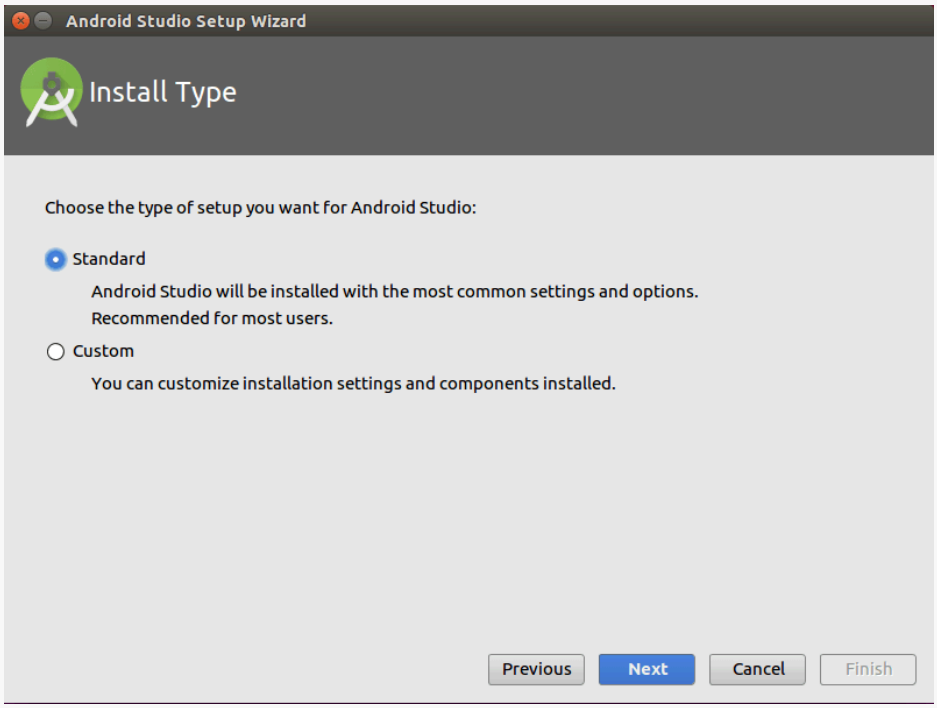


Figure 9: Android Studio Setup Wizard, Second Page

Here, you have a choice between “Standard” and “Custom” setup modes. Most likely, right now, the “Standard” route will be fine for your environment.

If you go the “Standard” route and click “Next”, you should be taken to a wizard page where you can choose your UI theme:

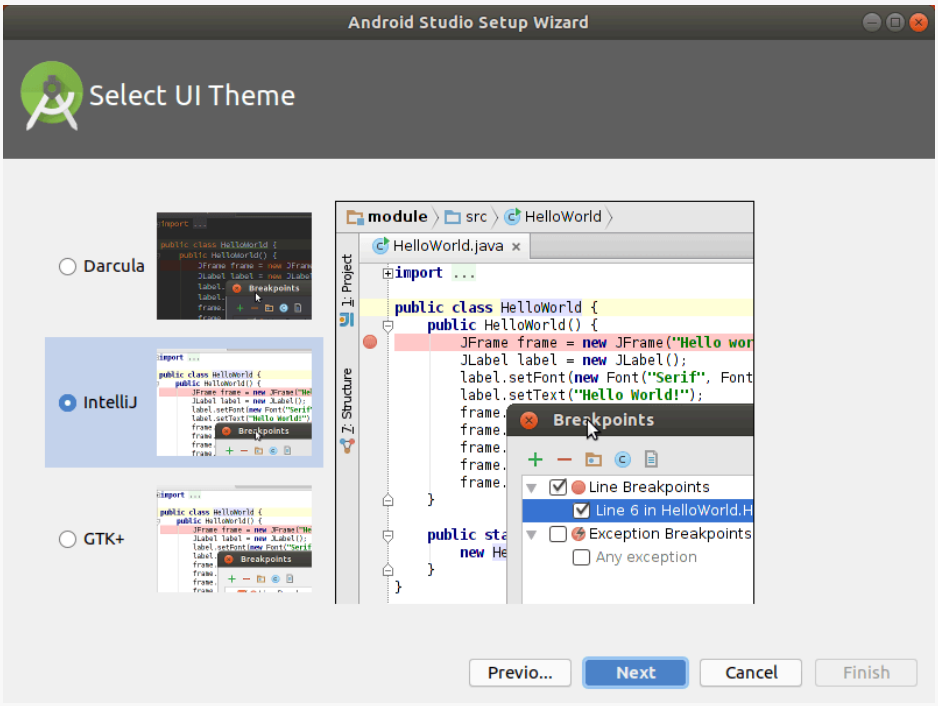


Figure 10: Android Studio Setup Wizard, UI Theme Page

Choose whichever you like, then click Next, to go to a wizard page to verify what will be downloaded and installed:

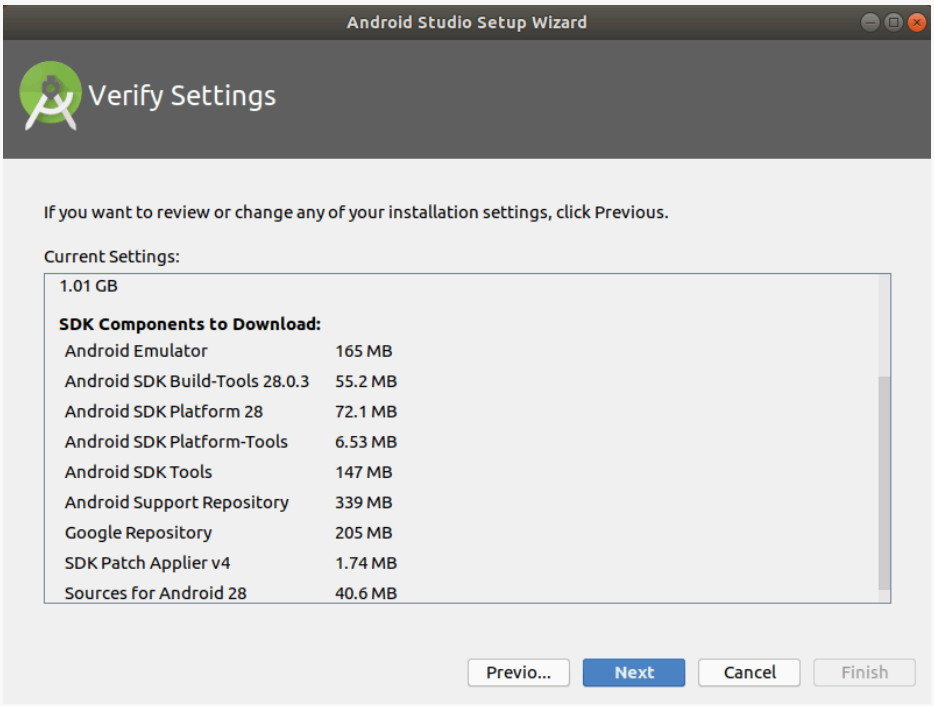


Figure 11: Android Studio Setup Wizard, Verify Settings Page

Clicking Next may take you to a wizard page explaining some information about the Android emulator:

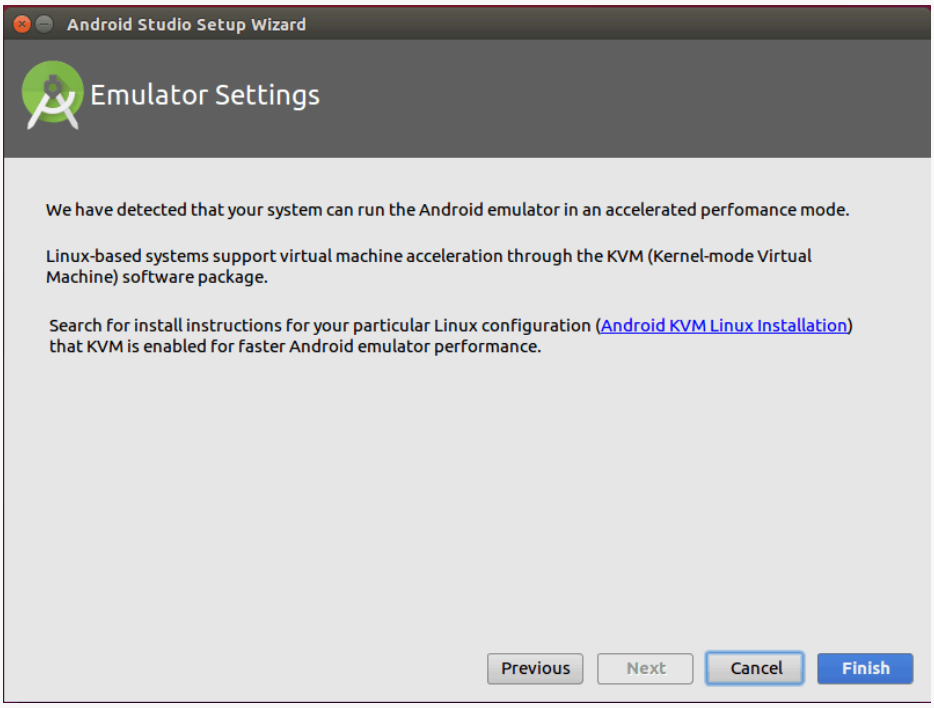


Figure 12: Android Studio Setup Wizard, Emulator Info Page

What is explained on this page may not make much sense to you. That is perfectly normal, and we will get into what this page is trying to say later in the book. Just click "Finish" to begin the setup process. This will include downloading a copy of the Android SDK and installing it into a directory adjacent to where Android Studio itself is installed.

When that is done, after clicking "Finish", Android Studio will busily start downloading stuff to your development machine.

Clicking “Finish” when that is done will then take you to the Android Studio Welcome dialog:

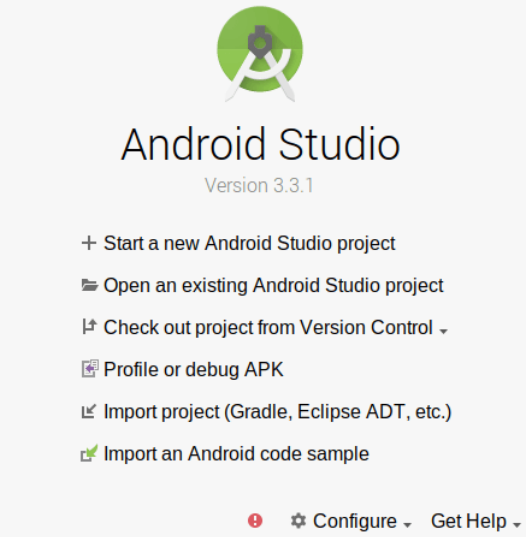


Figure 13: Android Studio Welcome Dialog

Creating a Starter Project

Creating an Android application first involves creating an Android "project". As with many other development environments, the project is where your source code and other assets (e.g., icons) reside. And, the project contains the instructions for your tools for how to convert that source code and other assets into an Android APK file for use with an emulator or device, where the APK is Android's executable file format.

Hence, in this tutorial, we kick off development of a sample Android application, to give you the opportunity to put some of what you are learning in this book in practice.

Step #1: Importing the Project

First, we need an Android project to work in.

Sometimes, you will create a new project yourself, using Android Studio's new-project wizard. However, frequently, you will start with an existing project that somebody else created. For example, if you are joining an Android development team, odds are that somebody else will create the project, or the project will already have been created by the time you join. In those cases, you will import an existing project, and that's what we will do here.

Download the starter project from CommonsWare's Web site. Then, UnZIP that project to some place on your development machine. It will unZIP into a `ToDo/` directory.

At that point, look at the contents of `gradle/wrapper/gradle-wrapper.properties`. It should look like this:

```
#Mon Jan 28 17:30:12 EST 2019
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-4.10.1-all.zip
```

(from To2-Project/ToDo/gradle/wrapper/gradle-wrapper.properties)

In particular, make sure that the `distributionUrl` points to a `services.gradle.org` URL. **Never** import a project into Android Studio without checking the `distributionUrl`, as a malicious person could have `distributionUrl` point to malware that Android Studio would load and execute.

Then, import the project. From the Android Studio welcome dialog — where we left off in the previous tutorial — that is handled by the "Import project (Gradle, Eclipse ADT, etc.)" option. From an existing open Android Studio IDE window, you would use File > New > Import Project... from the main menu.

Importing a project brings up a typical directory-picker dialog. Pick the `ToDo/` directory and click OK to begin the import process. This may take a while, depending on the speed of your development machine. A "Tip of the Day" dialog may appear, which you can dismiss.

At this point, the IDE window should be open on your starter project:

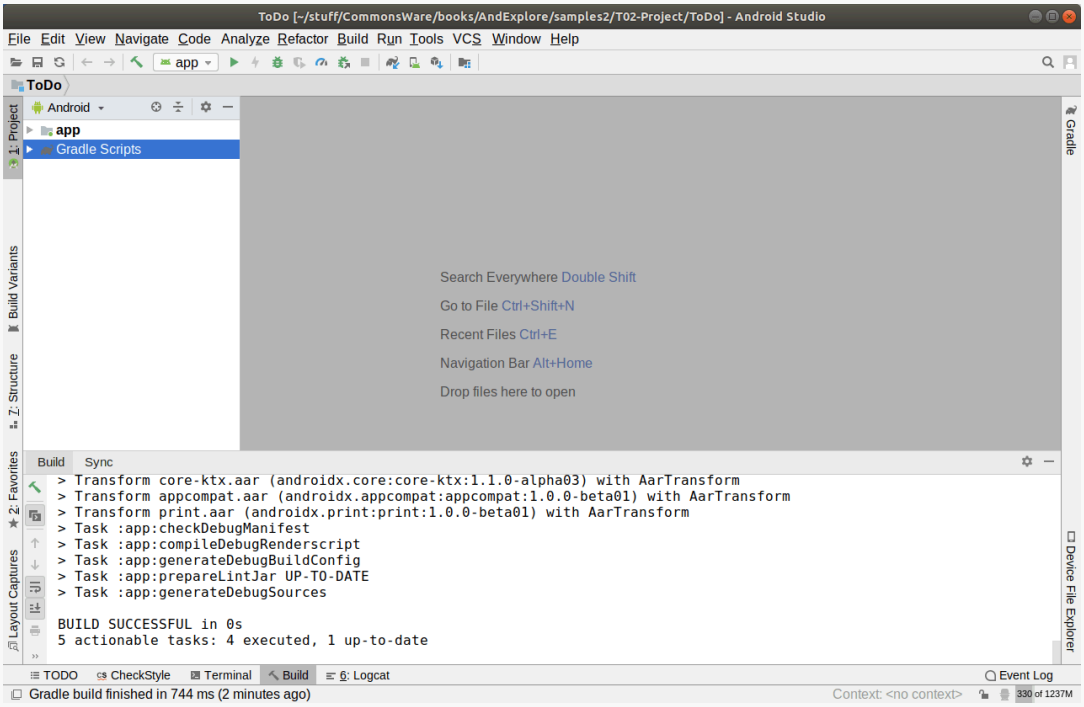


Figure 14: ToDo Project, As Initially Imported

The “Project” view — docked by default on the left side, towards the top — brings up a way for you to view what is in the project. Android Studio has several ways of viewing the contents of Android projects. The default one, that you are presented with when creating or importing the project, is known as the “Android view”:

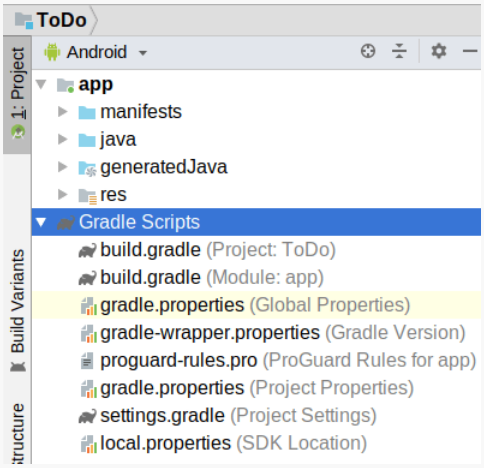


Figure 15: Android Studio, Project View, Android Content

While you are welcome to navigate your project using it, the tutorial chapters in this book, where they have screenshots of Android Studio, will show the “Project” contents in this view:

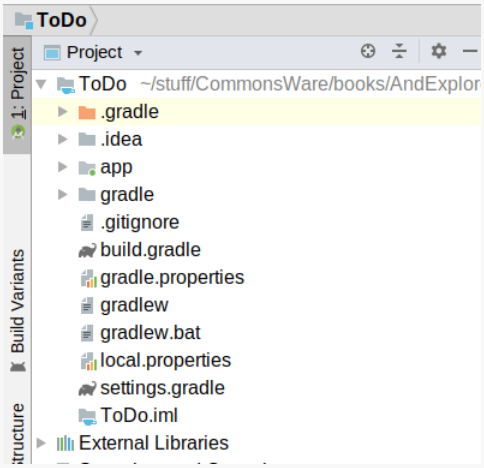


Figure 16: Android Studio, Project View, Project Content

To switch to this view — and therefore match what the tutorials will show you — click the Android drop-down above the tree and choose “Project” from the list.

Step #2: Setting Up the Emulator AVD

Your first decision to make is whether or not you want to bother setting up an emulator image right now. If you have an Android device, you may prefer to start testing your app on it, and come back to set up the emulator at a later point. In that case, skip to Step #3.

The Android emulator can emulate one or several Android devices. Each configuration you want is stored in an "Android virtual device", or AVD. The AVD Manager is where you create these AVDs.

To open the AVD Manager in Android Studio, choose Tools > AVD Manager from the main menu.

You should be taken to a "welcome"-type screen:

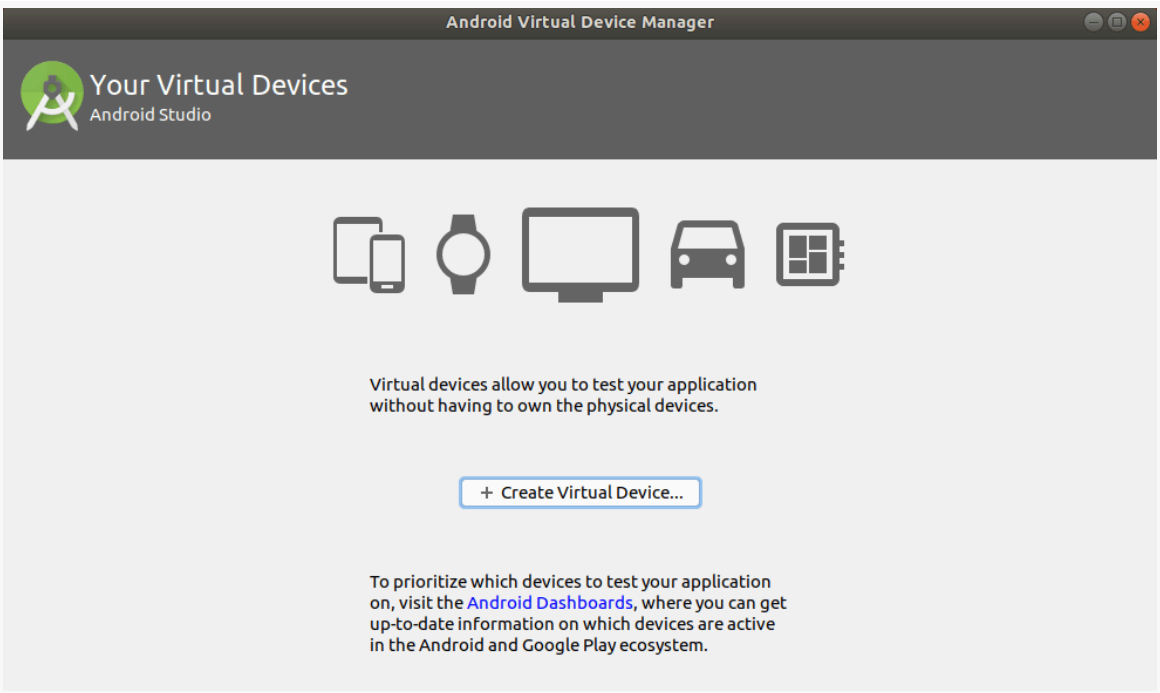


Figure 17: Android Studio AVD Manager, Welcome Screen

Click the “Create Virtual Device...” button, which brings up a “Virtual Device Configuration” wizard:

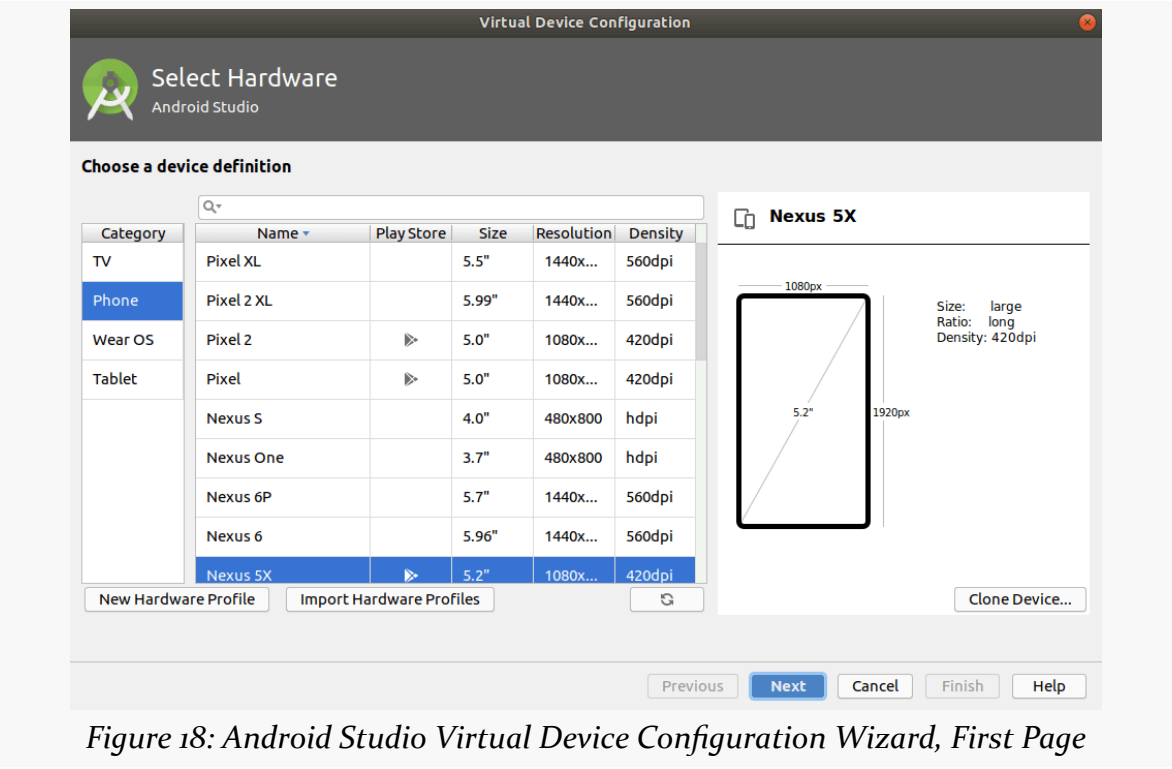


Figure 18: Android Studio Virtual Device Configuration Wizard, First Page

The first page of the wizard allows you to choose a device profile to use as a starting point for your AVD. The “New Hardware Profile” button allows you to define new profiles, if there is no existing profile that meets your needs.

Since emulator speeds are tied somewhat to the resolution of their (virtual) screens, you generally aim for a device profile that is on the low end but is not completely ridiculous. For example, a 1280x768 phone would be considered by many people to be fairly low-resolution. However, there are plenty of devices out there at that resolution (or lower), and it makes for a reasonable starting emulator.

If you want to create a new device profile based on an existing one — to change a few parameters but otherwise use what the original profile had – click the “Clone Device” button once you have selected your starter profile.

However, in general, at the outset, using an existing profile is perfectly fine. The Nexus 4 image is a likely choice to start with.

Clicking “Next” allows you to choose an emulator image to use:

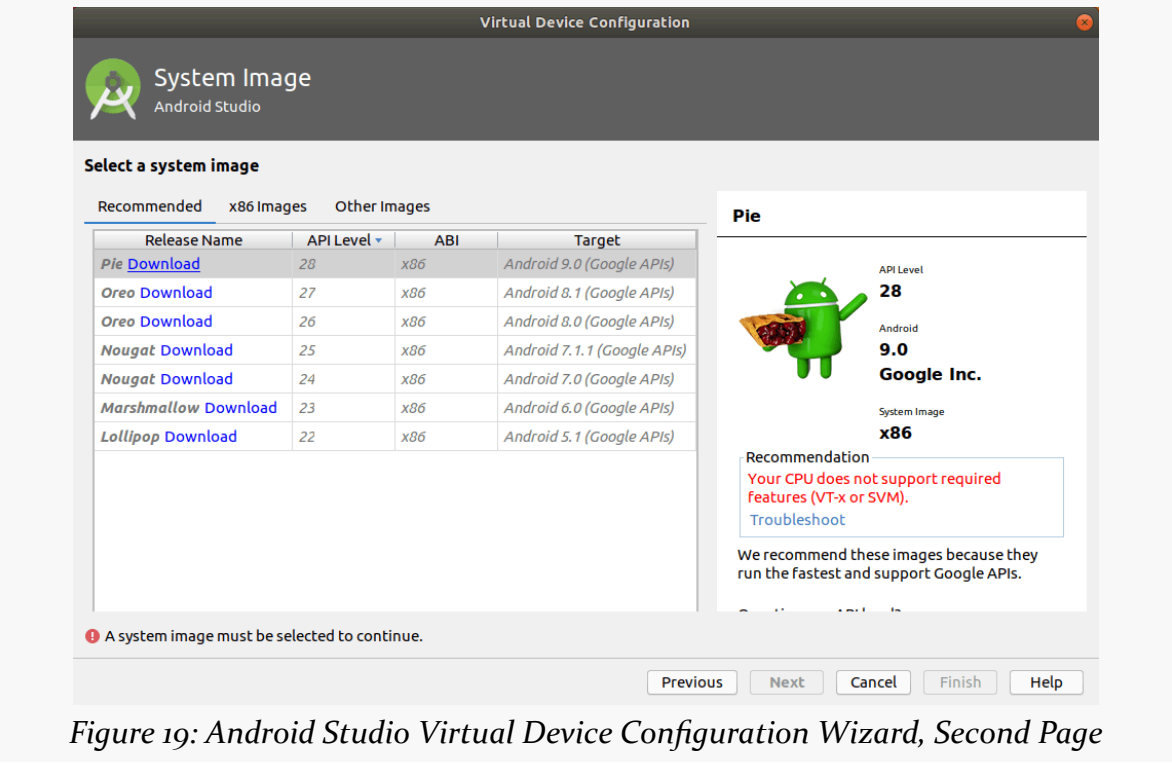


Figure 19: Android Studio Virtual Device Configuration Wizard, Second Page

The emulator images are spread across three tabs:

- “Recommended”
- “x86 Images”
- “Other Images”

For the purposes of the tutorials, you do not need an emulator image with the “Google APIs” — those are for emulators that have Google Play Services in them and related apps like Google Maps. However, in terms of API level, you can choose anything from API Level 21 (Android 5.0) on up.

If you click on the x86 Images tab, you should see some images with a “Download” link, and possibly others without it:

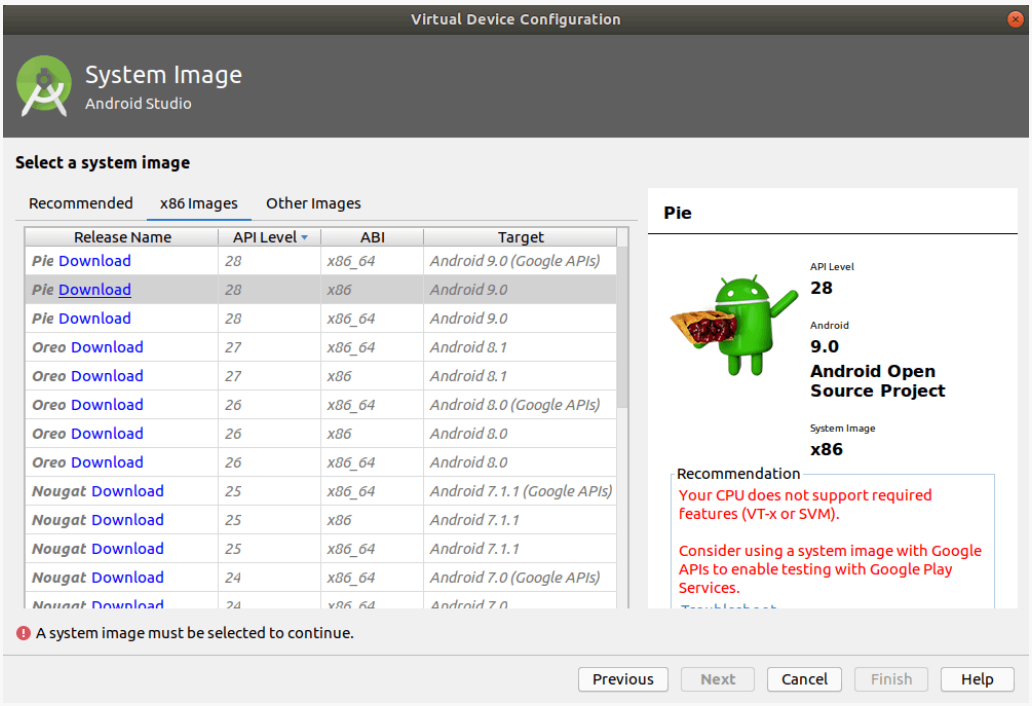


Figure 20: Android Studio Virtual Device Configuration Wizard, x86 Images

The emulator images with “Download” next to them will trigger a one-time download of the files necessary to create AVDs for that particular API level and CPU architecture combination, after a license dialog and progress dialog:

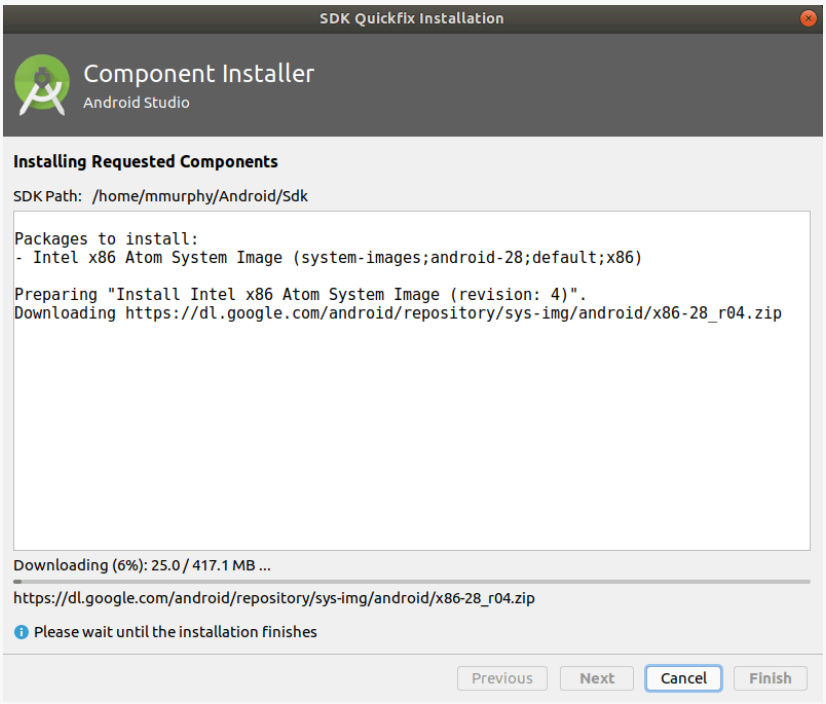


Figure 21: Android Studio Component Installer Dialog, Downloading API 28 x86 Image

Once you have identified the image(s) that you want — and have downloaded any that you did not already have — click on one of them in the wizard:

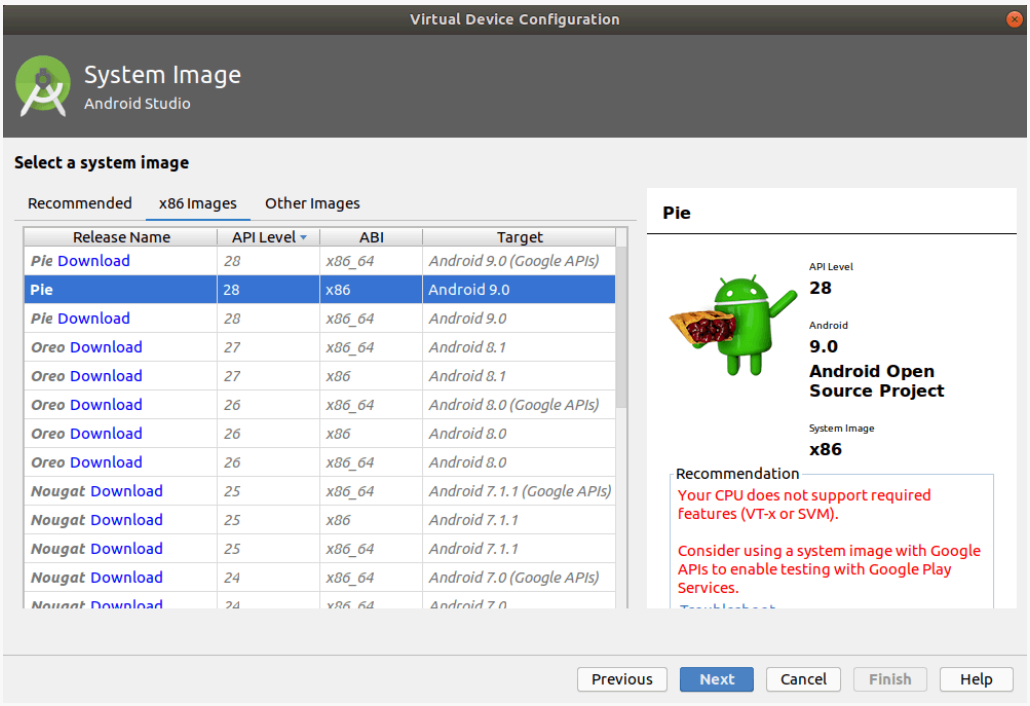


Figure 22: Android Studio Virtual Device Configuration Wizard, After Choosing Image

If you get the "Recommendation" box with the red "Your CPU does not support required features..." message — as is shown in the screenshot — your development machine is not set up to support this type of emulator image. For example, you may need to enable virtualization extensions in your PC's BIOS, as was noted in the previous tutorial.

Clicking “Next” allows you to finalize the configuration of your AVD:

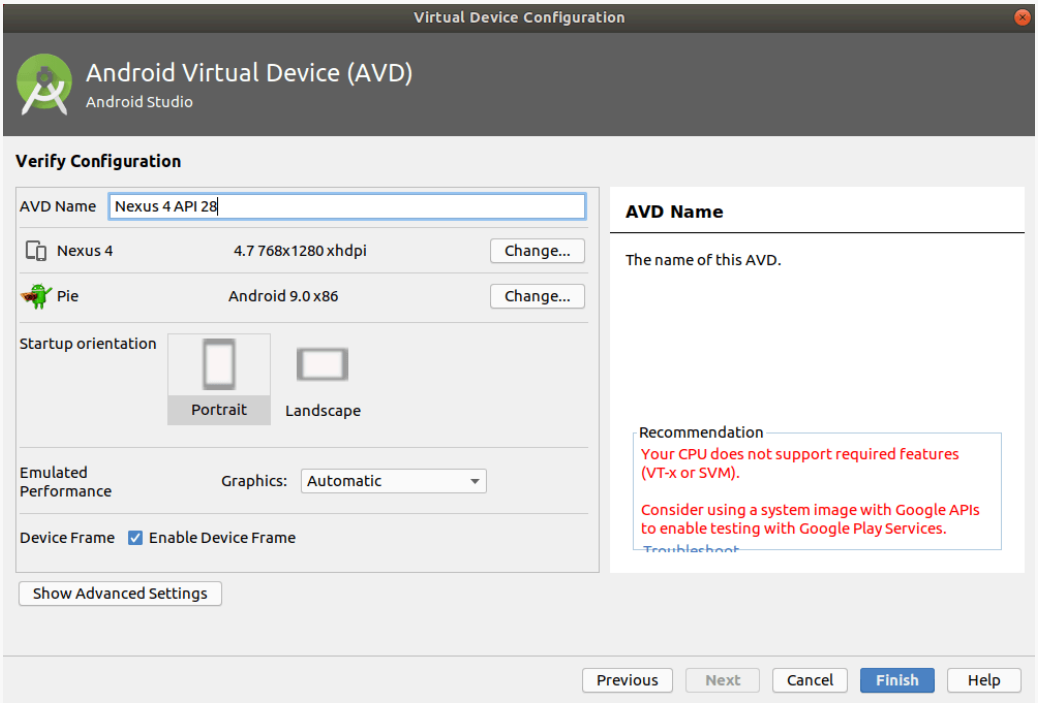


Figure 23: Android Studio Virtual Device Configuration Wizard, Third Page

A default name for the AVD is suggested, though you are welcome to replace this with your own value.

Change the AVD name, if necessary, to something valid: only letters, numbers, spaces, and select punctuation (e.g., `.`, `_`, `-`, `(`, `)`) are supported.

The rest of the default values should be fine for now.

Clicking “Finish” will return you to the main AVD Manager, showing your new AVD. You can then close the AVD Manager window.

Step #3: Setting Up the Device

You do not need an Android device to get started in Android application development. Having one is a good idea before you try to ship an application (e.g., upload it to the Play Store). And, perhaps you already have a device – maybe that is what is spurring your interest in developing for Android.

If you do not have an Android device that you wish to set up for development, skip this step.

The first thing to do to make your device ready for use with development is to go into the Settings application on the device. On Android 8.0+, go into System > About phone. On older devices, About is usually a top-level entry. In the About screen, tap on the build number seven times, then press BACK, and go into "Developer options" (which was formerly hidden)

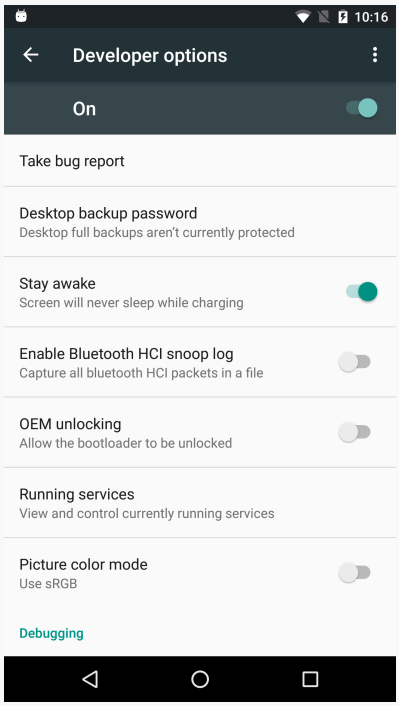


Figure 24: Developer Options, in Settings App

You may need to slide a switch in the upper-right corner of the screen to the "ON" position to modify the values on this screen.

Generally, you will want to scroll down and enable USB debugging, so you can use your device with the Android build tools:

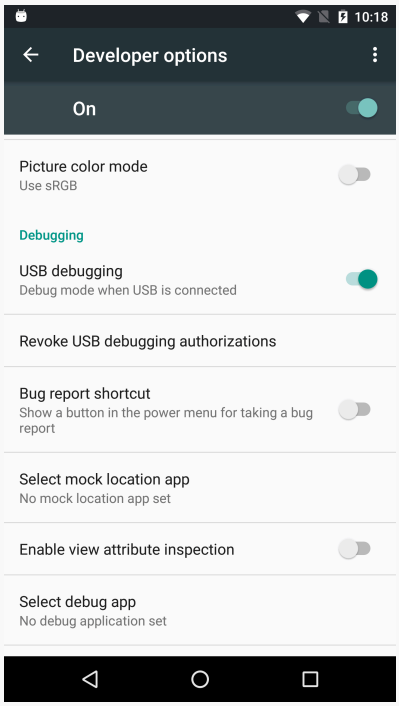


Figure 25: Debugging Options, in Settings App

You can leave the other settings alone for now if you wish, though you may find the "Stay awake" option to be handy, as it saves you from having to unlock your phone all of the time while it is plugged into USB.

Note that on Android 4.2.2 and higher devices, before you can actually use the setting you just toggled, you will be prompted to allow USB debugging with your *specific* development machine via a dialog box:

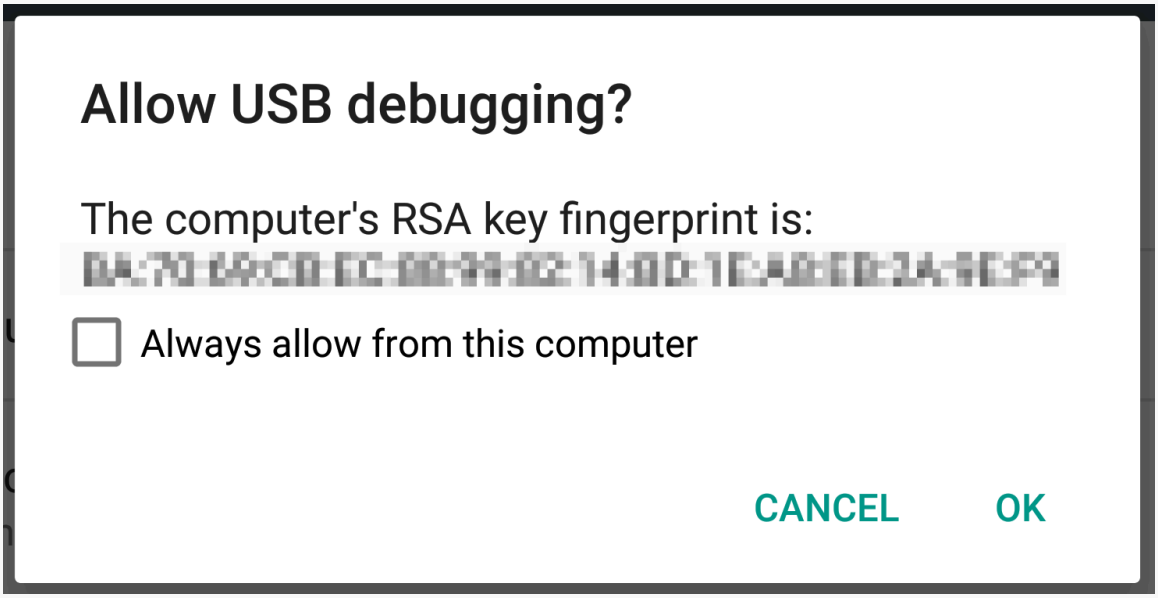


Figure 26: Allow USB Debugging Dialog

This occurs when you plug in the device via the USB cable and have the driver appropriately set up. That process varies by the operating system of your development machine, as is covered in the following sections.

Windows

When you first plug in your Android device, Windows will attempt to find a driver for it. It is possible that, by virtue of other software you have installed, that the driver is ready for use. If it finds a driver, you are probably ready to go.

If the driver is not found, here are some options for getting one.

Windows Update

Some versions of Windows (e.g., Vista) will prompt you to search Windows Update for drivers. This is certainly worth a shot, though not every device will have supplied its driver to Microsoft.

Standard Android Driver

In your Android SDK installation, if you chose to install the "Google USB Driver" package from the SDK Manager, you will find an `extras/google/usb_driver/` directory, containing a generic Windows driver for Android devices. You can try pointing the driver wizard at this directory to see if it thinks this driver is suitable for your device. This will often work for Nexus devices.

Manufacturer-Supplied Driver

If you still do not have a driver, the OEM USB Drivers in the developer documentation may help you find one for download from your device manufacturer. Note that you may need the model number for your device, instead of the model name used for marketing purposes (e.g., GT-P3113 instead of "Samsung Galaxy Tab 2 7.0").

macOS and Linux

Odds are decent that simply plugging in your device will "just work". You can see if Android recognizes your device via running `adb devices` in a shell (e.g., macOS Terminal), where `adb` is in your `platform-tools/` directory of your SDK. If you get output similar to the following, the build tools detected your device:

```
List of devices attached
HT9CPP809576  device
```

If you are running Ubuntu (or perhaps other Linux variants), and this command did not work, you may need to add some `udev` rules. For example, here is a `51-android.rules` file that will handle the devices from a handful of manufacturers:

```
SUBSYSTEM=="usb", SYSFS{idVendor}=="0bb4", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="22b8", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="18d1", MODE="0666"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="18d1", ATTRS{idProduct}=="0c01", MODE="0666",
OWNER="<me>"
SUBSYSTEM=="usb", SYSFS{idVendor}=="19d2", SYSFS{idProduct}=="1354", MODE="0666"
SUBSYSTEM=="usb", SYSFS{idVendor}=="04e8", SYSFS{idProduct}=="681c", MODE="0666"
```

Drop that in your `/etc/udev/rules.d` directory on Ubuntu, then either reboot the computer or otherwise reload the `udev` rules (e.g., `sudo service udev reload`). Then, unplug and re-plug in the device and see if it is detected.

Step #4: Running the Project

Now, we can confirm that our project is set up properly by running it on a device or emulator.

To do that in Android Studio, just press the Run toolbar button (usually depicted as a green rightward-pointing triangle):

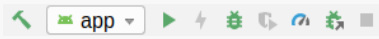


Figure 27: Android Studio Toolbar, Showing Run Button

You will then be presented with a dialog indicating where you want the app to run: on some existing device or emulator, or on some newly-launched emulator:

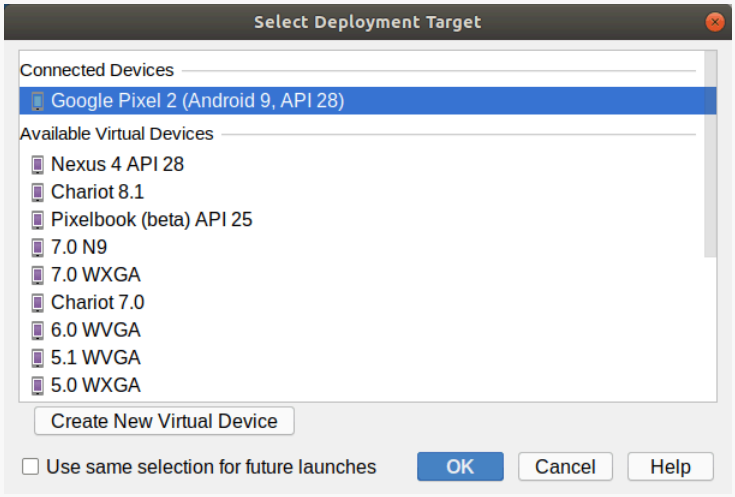


Figure 28: Android Studio Device Chooser Dialog

If you do not have an emulator running, choose one from the list, then click OK. Android Studio will launch your emulator for you.

And, whether you start a new emulator instance or reuse an existing one, your app should appear on it:

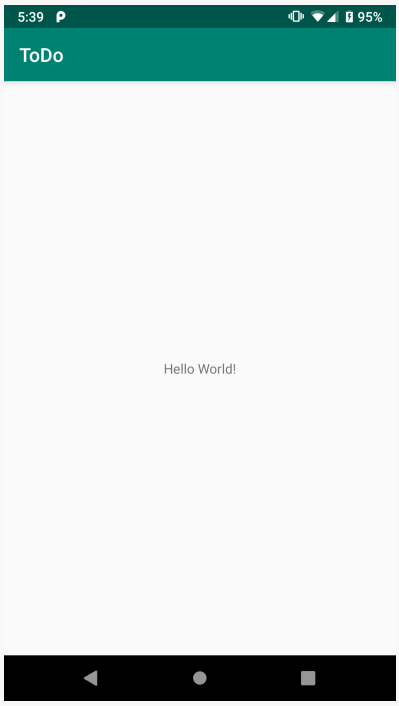


Figure 29: Android 8.1 Device, Showing ToDo App

Note that you may have to unlock your device or emulator to actually see the app running.

The first time you launch the emulator for a particular AVD, you may see this message:

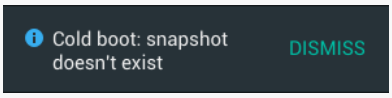


Figure 30: Android Emulator Cold-Boot Warning

The emulator now behaves a bit more like an Android device. Closing the emulator window used to be like completely powering off a phone, but now it is more like tapping the POWER button to turn off the screen. The next time you start that particular AVD, it will wake up to the state in which you left it, rather than booting from scratch ("cold boot"). This speeds up starting the emulator. Occasionally, though, you will have the need to start the emulator as if the device were powering

on. To do that, in the AVD Manager, in the drop-down menu in the Actions column, choose “Cold Boot Now”.

Modifying the Manifest

Now that we have our starter project, we need to start making changes, as we have a *lot* of work to do.

In this tutorial, we will start with the Android manifest, one of the core files in an app. Here, we will make a few changes, just to help get you familiar with editing this file. We will be returning to this file — and other core files, like Gradle build files – many times over the course of the rest of the book.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.

You can learn more about the contents of the manifest in the "Inspecting Your Manifest" chapter of *Elements of Android Jetpack*!

Some Notes About Relative Paths

In these tutorials, you will see references to relative paths, like `AndroidManifest.xml`, `res/layout/`, and so on.

You should interpret these paths as being relative to the `app/src/main/` directory within the project, except as otherwise noted. So, for example, Step #1 below will ask you to open `AndroidManifest.xml` — that file can be found in `app/src/main/AndroidManifest.xml` from the project root.

Step #1: Supporting Screens

Android devices come in a wide range of shapes and sizes. Our app can support them all. However, we should advise Android that we are indeed willing to support any screen size. To do this, we need to add a `<supports-screens>` element to the manifest.

To do this, double-click on `AndroidManifest.xml` in the project explorer:

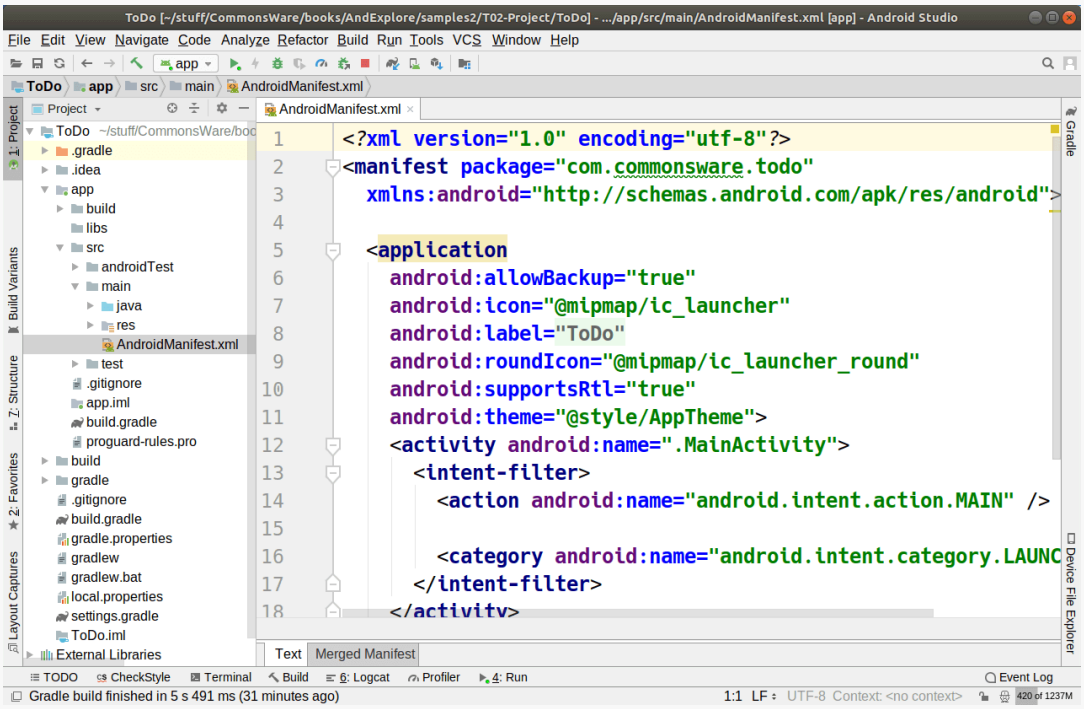


Figure 31: Android Studio, Showing Manifest Editor

As a child of the root `<manifest>` element, add a `<supports-screens>` element as follows:

```
<supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"
    android:xlargeScreens="true"/>
```

At this point, the manifest should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.todo"
  xmlns:android="http://schemas.android.com/apk/res/android">
  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"
    android:xlargeScreens="true"/>

  <application
    android:allowBackup="true"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

Step #2: Blocking Backups

If you look at the `<application>` element, you will see that it has a few attributes, including `android:allowBackup="true"`. This attribute indicates that `ToDo` should participate in Android's automatic backup system.

That is not a good idea, until you understand the technical and legal ramifications of that choice.

In the short term, change `android:allowBackup` to be `false`.

At this point, your manifest should look like:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 28
```

```
    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdkVersion 21
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.1'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
}
```

(from To3-Manifest/ToDo/app/build.gradle)

What We Changed

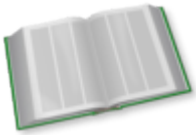
The book's GitHub repository contains the entire result of having completed this tutorial. In particular, it contains the changed files:

- `app/src/main/AndroidManifest.xml`

Changing Our Icon

Our `ToDo` project has some initial resources, such as our app's display name and its launcher icon. However, the defaults are not what we want for the long term. So, in addition to adding new resources in future tutorials, we will change the launcher icon in this tutorial.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.



You can learn more about Android's resource system in the "Exploring Your Resources" chapter of *Elements of Android Jetpack*!

You can learn more about launcher icons and the Image Asset Wizard in the "Icons" chapter of *Elements of Android Jetpack*!

Step #1: Getting the Replacement Artwork

First, we need something that visually represents a to-do list, particularly when shown as the size of an icon in a home screen launcher.

This public domain piece of clipart will serve this purpose:

Figure 32: Checklist Clipart from OpenClipArt.org

That clipart is available as an SVG, a PNG (in one of three resolutions), or a PDF. For the purposes of creating a launcher icon, the PNG is the simplest file format to use.

Download the medium-resolution PNG file to some location on your development machine *outside* of the project directory. You will only need it for a few minutes, so feel free to use a temporary location (e.g., `/tmp` on Linux) if desired.

Step #2: Changing the Icon

Android Studio includes an Image Asset Wizard that is adept at creating launcher icons. This is important, as while creating launcher icons used to be fairly simple, Android 8.0 made launcher icons a *lot* more complicated... but the Image Asset Wizard hides most of that complexity.

First, right-click over the `res/` directory in your `main` source set in the project explorer:

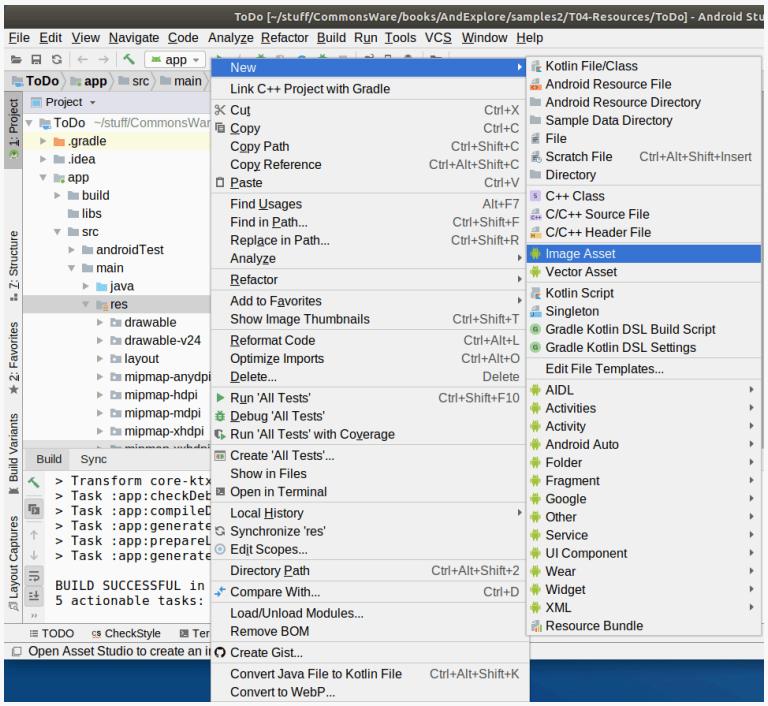


Figure 33: Android Studio Project Tree Context Menu

In that context menu, choose New > Image Asset from the context menu. That will bring up the Asset Studio wizard:

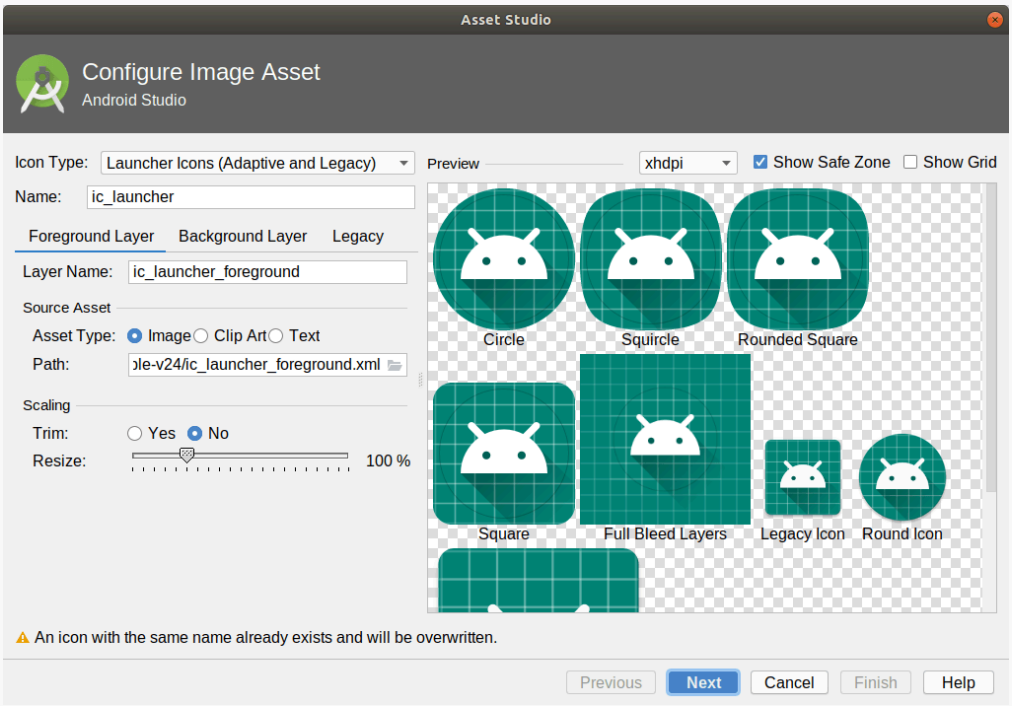


Figure 34: Android Studio Image Asset Wizard, As Initially Launched

In the “Icon Type” drop-down, make sure that “Launcher Icons (Adaptive and Legacy)” is chosen — this should be the default. Also, ensure that the “Name” field has `ic_launcher`, which also should be the default.

In the “Foreground Layer” tab, ensure that the “Layer Name” is `ic_launcher_foreground`. In the “Source Asset” group, ensure that the “Asset Type” is set to “Image”. Then, click the folder button next to the “Path” field, and find the clipart that you downloaded in Step #1 above.

When you load the image, it will be just a bit too big:

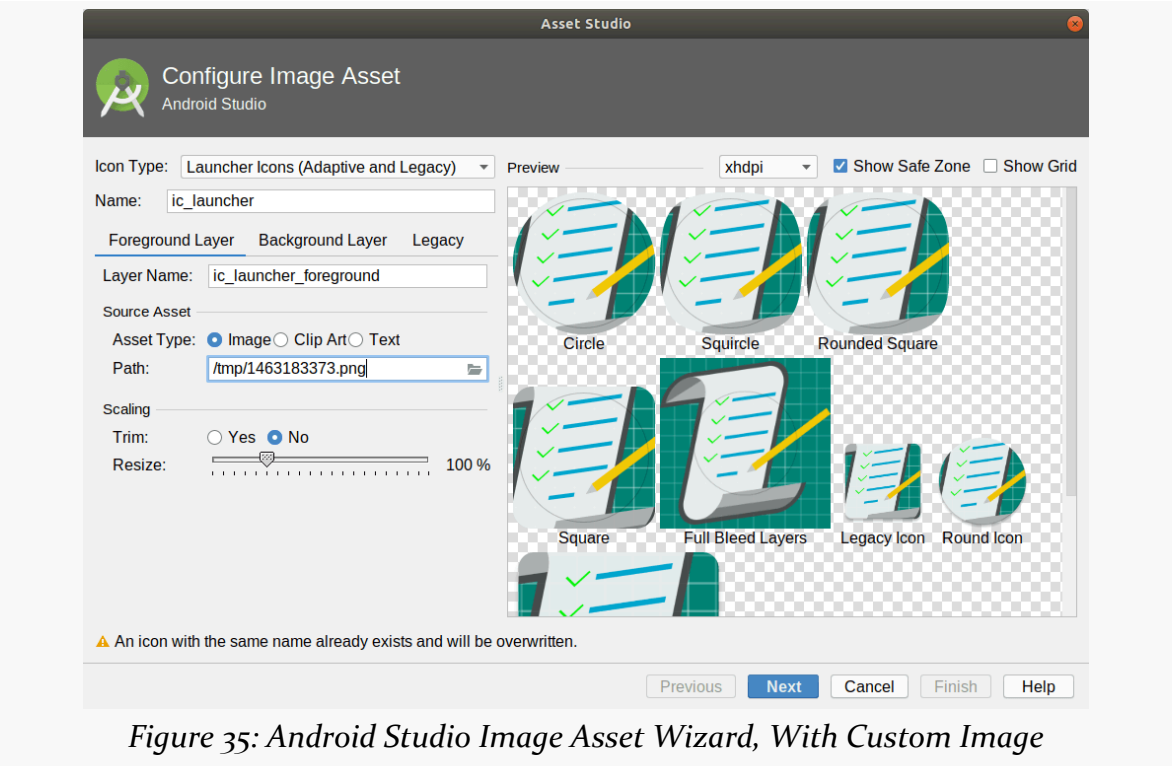


Figure 35: Android Studio Image Asset Wizard, With Custom Image

To fix this, in the "Scaling" group, select "Yes" for "Trim". Then, adjust the "Resize" slider until the clipart is inside the circular "safe zone" region in the previews. A "Resize" value of around 80% should work:

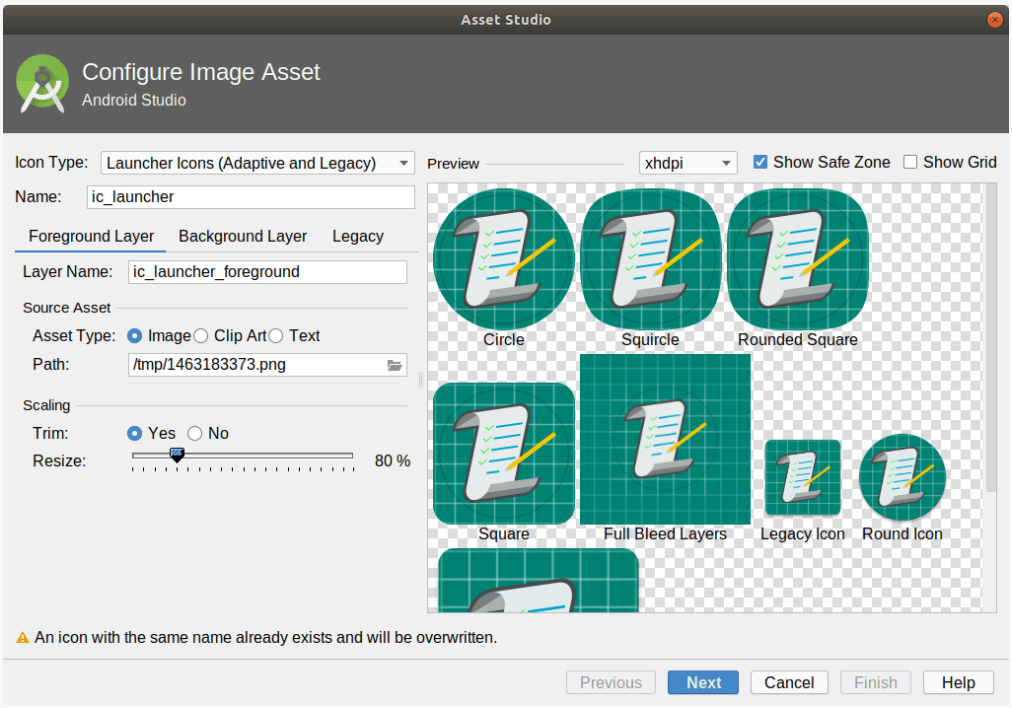


Figure 36: Android Studio Image Asset Wizard, With Scaled Custom Image

Switch to the “Background Layer” tab and ensure that the “Layer Name” is `ic_launcher_background`. Then, switch the “Asset Type” to “Color”:

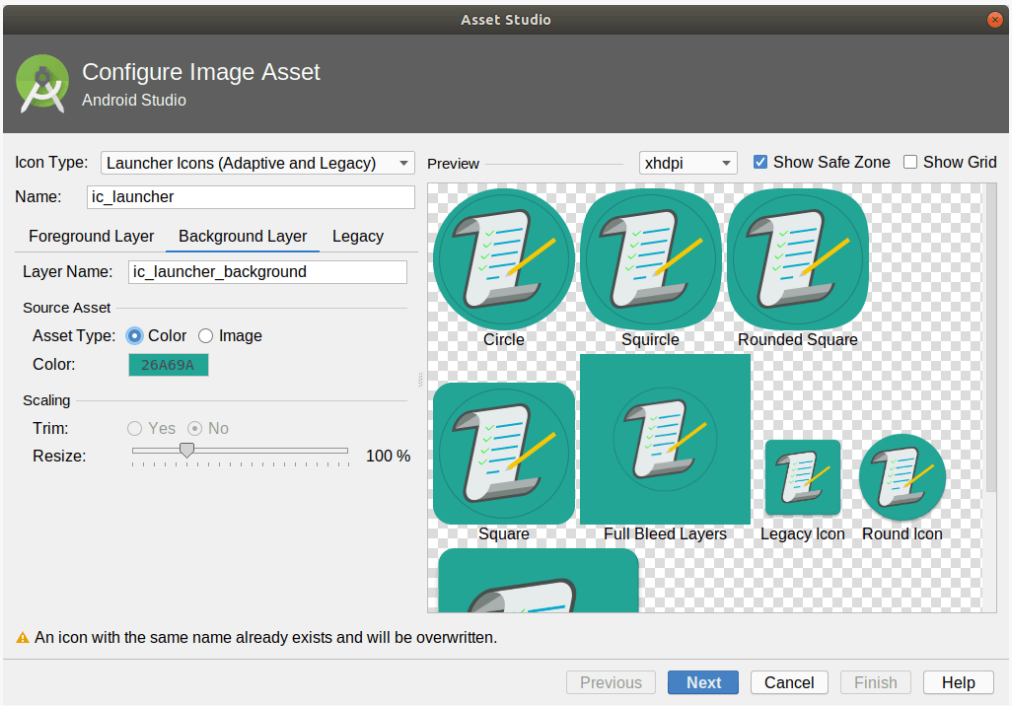


Figure 37: Android Studio Image Asset Wizard, Background Layer Tab, Using Default Color Background

If you do not like the default color, tap the hex color value to bring up a color picker:

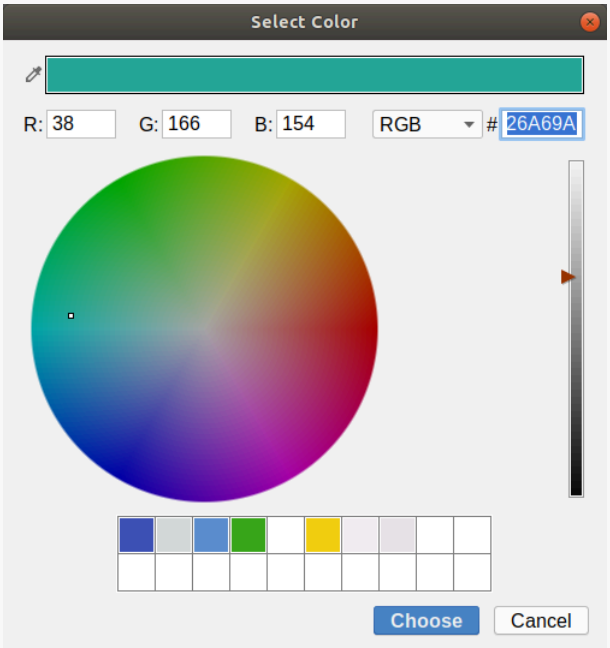


Figure 38: Android Studio Image Asset Wizard Color Picker Dialog

Pick some other color, then click “Choose” to apply that to the icon background:

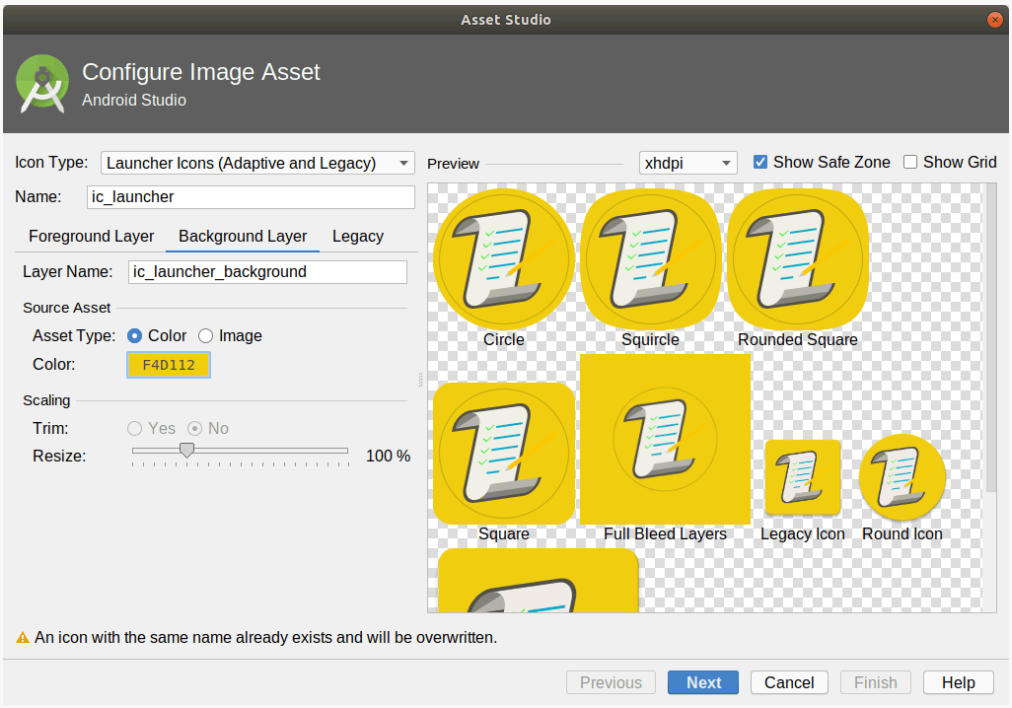


Figure 39: Android Studio Image Asset Wizard, Background Layer Tab, Using Custom Color Background

Then, switch to the “Legacy” tab. Ensure that the “Generate” value is “Yes” for both “Legacy Icon” and “Round Icon”, but set it to “No” for “Google Play Store Icon” (as this app will not be published on the Play Store). Also, switch the “Shape” value for the “Legacy Icon” to “Circle”:

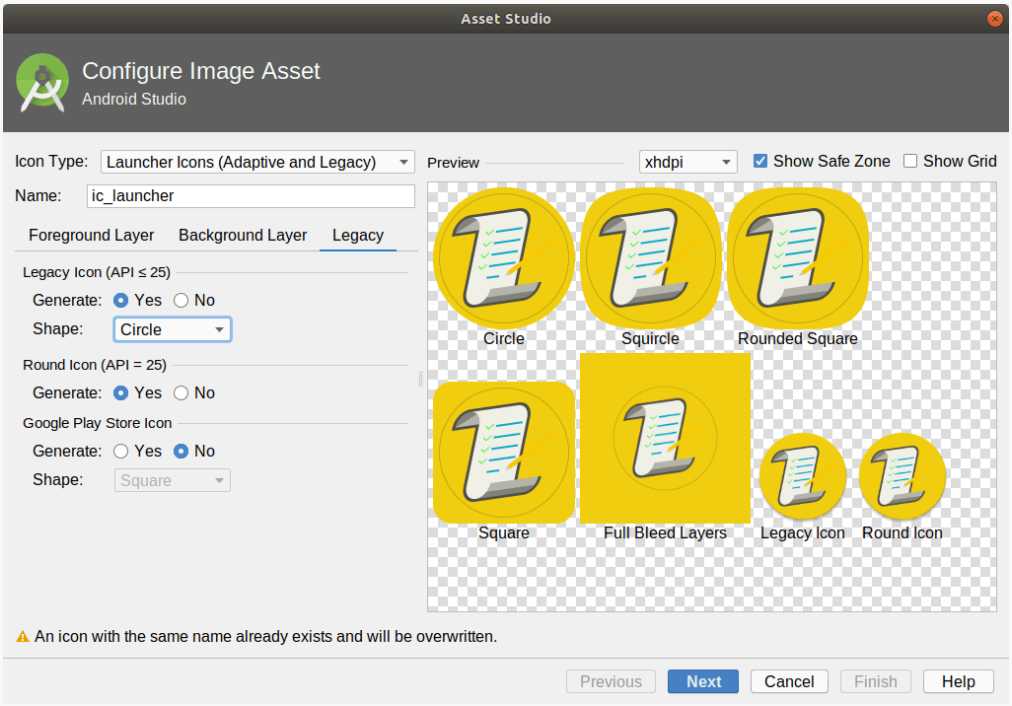


Figure 40: Android Studio Image Asset Wizard, Legacy Tab, Using Custom Settings

That way, our icon should be the same on most pre-Android 8.0 devices. On Android 8.0+ devices — and on a few third-party home screens on older devices – our icon will be our clipart on our chosen background color, but with a shape determined by the home screen implementation.

Click the “Next” button at the bottom of the wizard to advance to a confirmation screen:

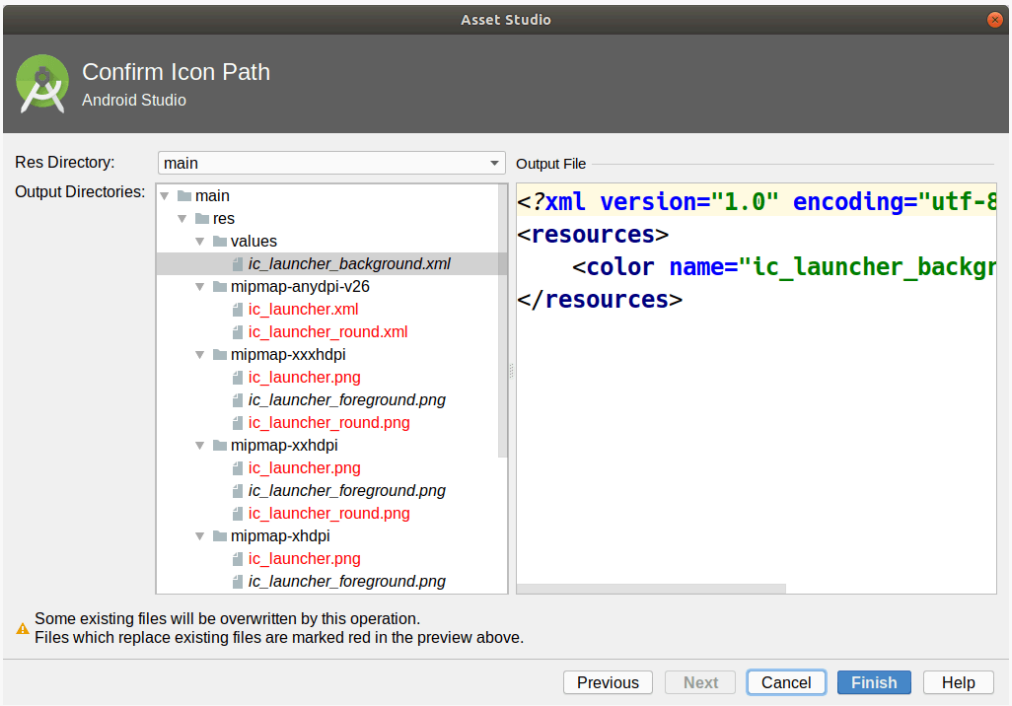


Figure 41: Android Studio Image Asset Wizard, Confirmation Page

There will be a warning that existing files will be overwritten. Since that is what we are intending to do, this is fine.

Click “Finish”, and Android Studio will generate your launcher icon.

Step #3: Running the Result

If you run the resulting app, then go back to the home screen launcher, you will see that it shows up with the new icon:

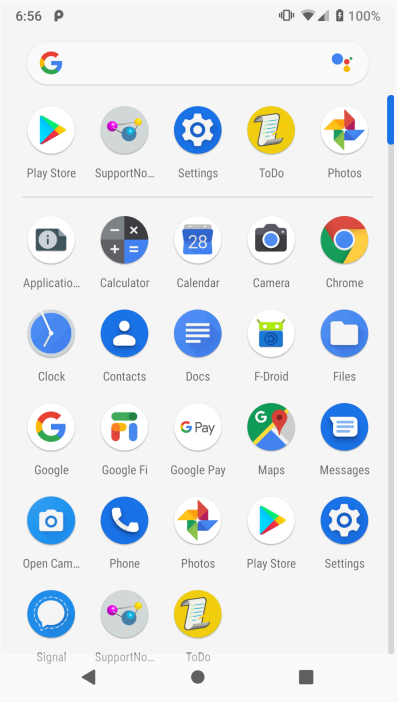


Figure 42: Android Home Screen Launcher, Showing App Icon (Lower Right)

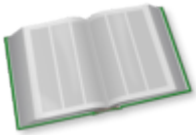
What We Changed

The book's GitHub repository contains the entire result of having completed this tutorial. A number of files were changed in `app/src/main/res/`, as creating launcher icons is annoyingly complicated.

Adding a Library

Most of an Android app comes from code that you did not write. It comes from code written by others, in the form of libraries. Even though we have not gotten very far with the ToDo app, we are already using some libraries, and in this chapter, we will update that roster.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.



You can learn more about Gradle in the "Reviewing Your Gradle Scripts" chapter of *Elements of Android Jetpack*!

Step #1: Removing Unnecessary Cruft

Open `app/build.gradle` in Android Studio. You will find that it contains a `dependencies` closure that looks like this:

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.1'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
}
```

A new Android Studio project will contain this sort of initial set of dependencies, though the details will vary a bit depending on Android Studio version and the particular choices you make when creating the project. The `implementation`, `testImplementation`, and `androidTestImplementation` lines indicate libraries that we want to use, where `implementation` is for our app and the others are for our tests.

Most of these dependencies are ones that we will use, either currently or in the future. The one that we will not is:

```
    implementation fileTree(dir: 'libs', include: ['*.jar'])
```

This says "hey, look in the `libs/` directory of this module, and pull in any JAR files you find in there". We will not have any such JARs. So, you can delete the `libs/` directory from the `app/` module directory, by right-clicking over `libs/` and choosing "Delete" from the context menu. Then, delete the `implementation` line shown above, so our build process does not try looking in that now-deleted directory.

At this point, you should get a banner at the top of the editor, offering you the chance to "Sync Now":

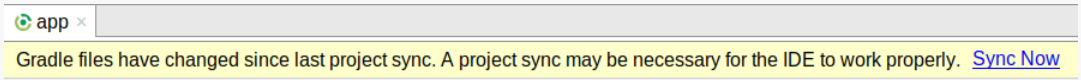


Figure 43: Android Studio "Sync Now" Banner

Since we have other changes to make to this file, you can hold off on clicking that link.

Step #2: Adding Support for RecyclerView

The idea is that the ToDo app will present a list of tasks to be done. That requires that we have something to display a list to the user. There are two typical solutions for that problem: `ListView` and `RecyclerView`. `RecyclerView` is more modern and more flexible, so it is a good choice for this problem.

However, `ListView` does have one advantage over `RecyclerView`: `ListView` is part of the framework portion of the Android SDK, and so it is always available to apps. `RecyclerView` requires us to add a dependency to the app.

Fortunately, we happen to be in a tutorial where we are working with the

dependencies in the app.

To that end, inside the `dependencies` closure, add the following line:

```
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
```

(from To5-Libraries/ToDo/app/build.gradle)

At this point, go ahead and click the "Sync Now" link in the banner at the top of the editor.

Your resulting `app/build.gradle` file should now resemble:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 28

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdkVersion 21
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}

dependencies {
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.1'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
}
```

(from To5-Libraries/ToDo/app/build.gradle)

What We Changed

The book's GitHub repository contains the entire result of having completed this

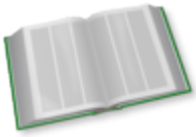
tutorial. In particular, it contains the changed files:

- `app/build.gradle`

Constructing a Layout

Our starter project has a layout resource: `res/layout/activity_main.xml` already. However, it is just a bit different from what we need. So, in this tutorial, we will modify that layout, using the Android Studio drag-and-drop GUI builder.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.



You can learn more about `ConstraintLayout` in the "Introducing ConstraintLayout" chapter of *Elements of Android Jetpack*!

Step #1: Examining What We Have And What We Want

The starter project has a single layout resource, in `res/layout/activity_main.xml`. If you open that up in the IDE and switch to the "Text" sub-tab, you will see XML 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"
  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>
```

We have a `ConstraintLayout` as our root container. `ConstraintLayout` comes from that `com.android.support.constraint:constraint-layout` artifact that we saw in our `dependencies` list in the preceding tutorial. `ConstraintLayout` is Google's recommended base container for most layout resources, as it is the most flexible option.

Inside, we have a `TextView`, with a simple "Hello World!" message.

As it turns out, we can use both of those in the UI that we are going to construct:

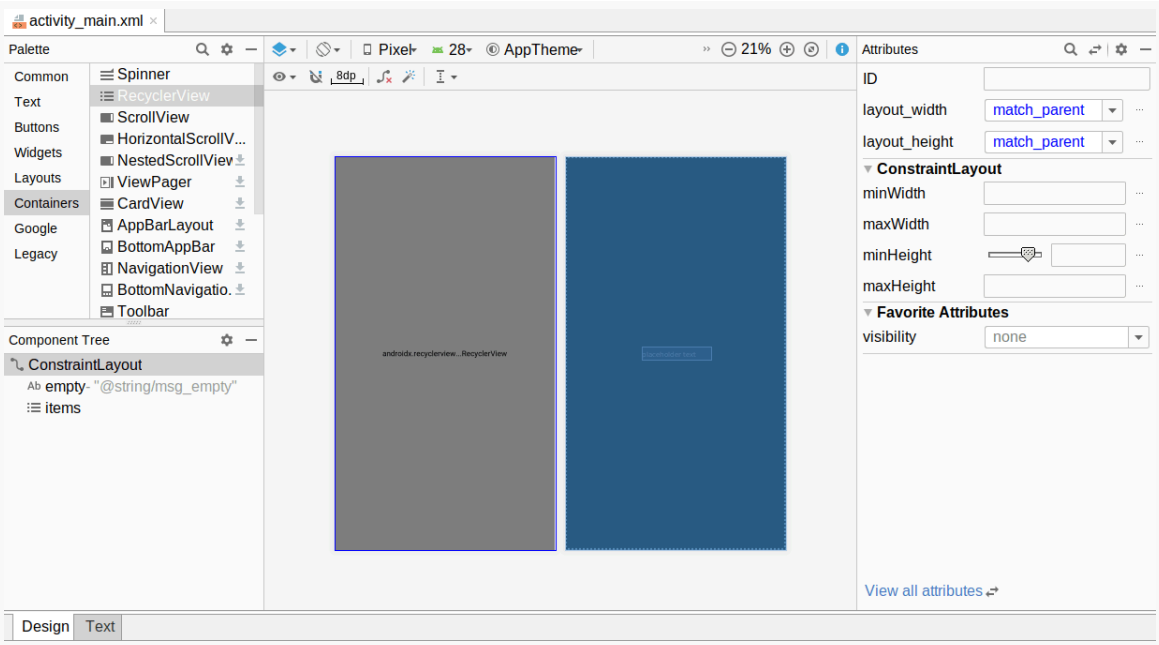


Figure 44: Android Studio Layout Designer, Showing End Result of This Tutorial

We want:

- a `RecyclerView`, to use for our list of to-do items
- a `TextView`, to show when the `RecyclerView` is empty

The `RecyclerView` and the `TextView` will go in the same space. In code, we will toggle the visibility of the `TextView`, so that it is visible when we have no to-do items to show in the `RecyclerView` and hidden when we have one or more to-do items to show.

Step #2: Adding a RecyclerView

In the GUI builder, in the "Palette" area, switch to "Containers" category:

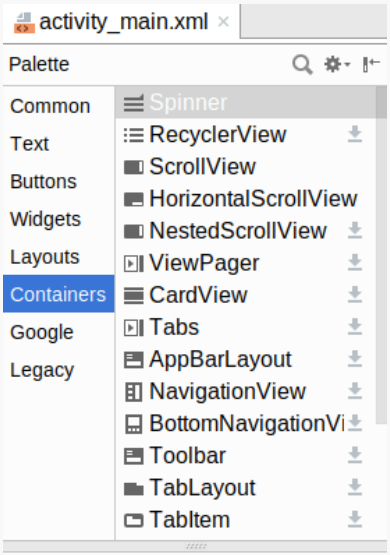


Figure 45: Android Studio Layout Designer Palette

Drag a `RecyclerView` out of the "Palette" and drop it roughly in the center of the preview area:

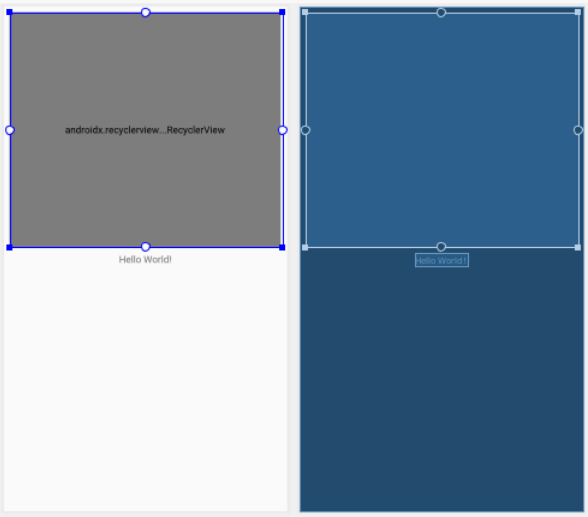


Figure 46: Android Studio Layout Designer, Showing Added RecyclerView

Unfortunately, the Android Studio Layout Designer has many issues, including making the `RecyclerView` too big to manipulate. Grab the upper-right corner of the `RecyclerView` and drag it inwards to shrink it a bit:

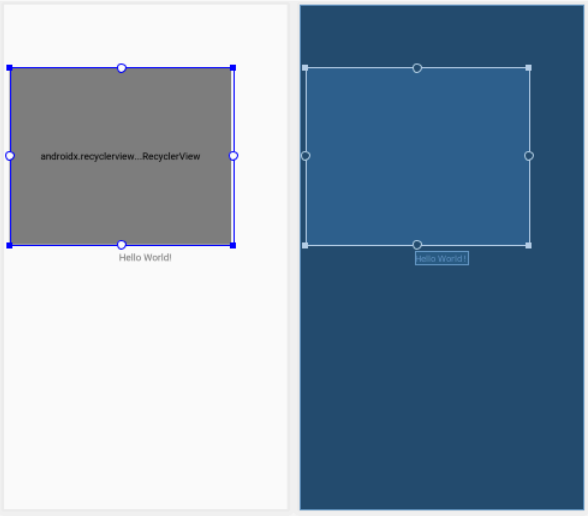


Figure 47: Android Studio Layout Designer, Showing Smaller RecyclerView

Then drag the RecyclerView away from the left edge a bit, to give you room to

maneuver:

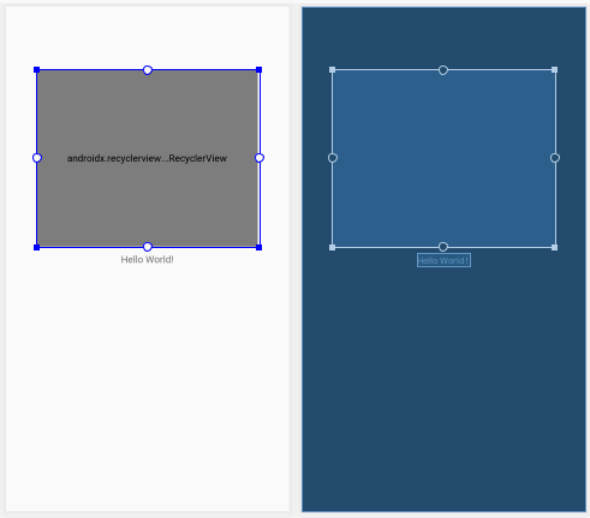


Figure 48: Android Studio Layout Designer, Showing Moved RecyclerView

Hover your mouse over the left edge of the `RecyclerView` preview rectangle, find the dot towards the center of the left edge, and drag it to connect with the left edge of the preview area, which will connect it to that side of the `ConstraintLayout`:

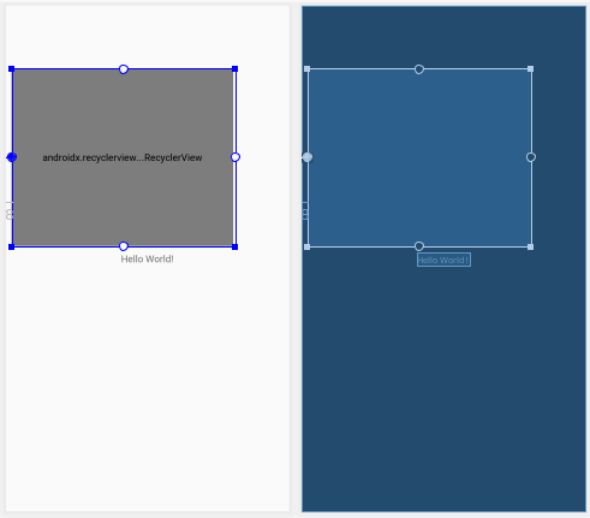


Figure 49: Android Studio Layout Designer, Showing RecyclerView Anchored on the Left

Repeat that process on the right side:

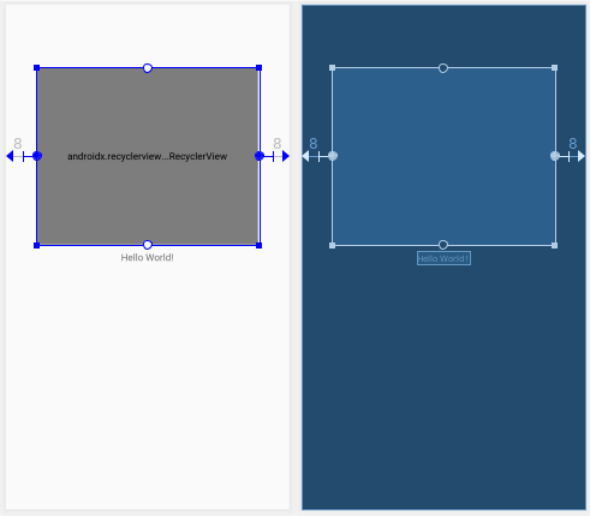


Figure 50: Android Studio Layout Designer, Showing RecyclerView Anchored on Both Sides

Repeat that process on the top side:

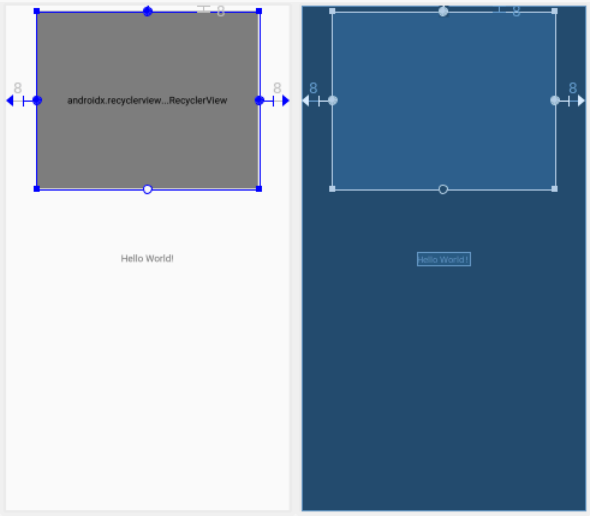


Figure 51: Android Studio Layout Designer, Showing RecyclerView Anchored on Both Sides and the Top

Repeat that process on the bottom side:

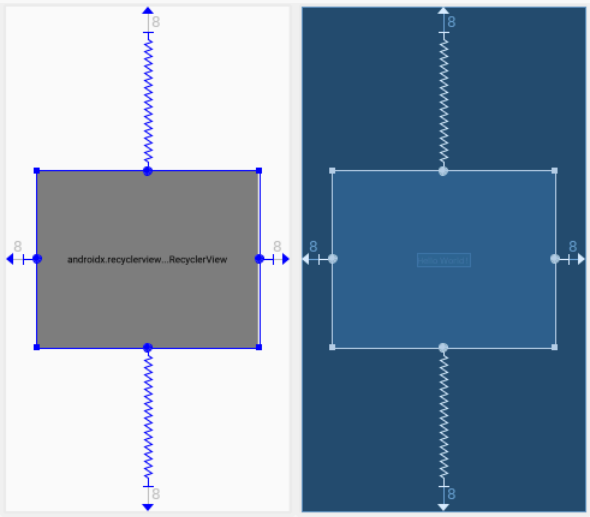


Figure 52: Android Studio Layout Designer, Showing RecyclerView Anchored on All Four Sides

In the "Attributes" pane on the right side of the Layout Designer, change the `layout_width` and `layout_height` values each to `match_constraint` (a.k.a., `0dp`):

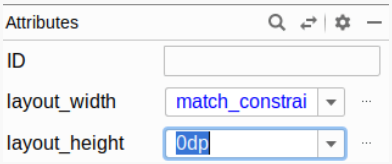


Figure 53: Android Studio Layout Designer, Attributes Pane, Showing New Sizes

Now, you should see our `RecyclerView` fill the entire space:

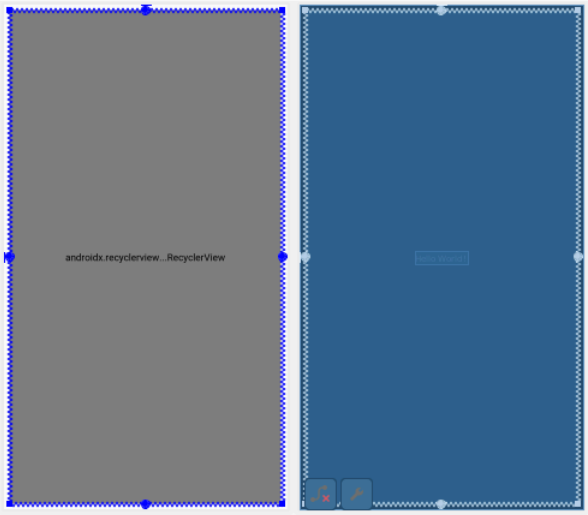


Figure 54: Android Studio Layout Designer, Showing RecyclerView Filling the Screen

Back in the "Attributes" pane, give the `RecyclerView` an ID of `items`, via the field at the top. In the diagram beneath the ID field, change the 8 values to 0, by clicking on the `8`, then choosing `0` from the drop-down list that appears:

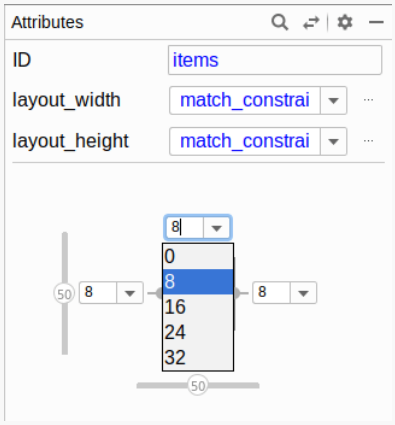


Figure 55: Android Studio Layout Designer, Attributes Pane, Showing Margin Drop-Down

The drag-and-drop process automatically sets up some margins, but we will be applying margins elsewhere (in the `RecyclerView` rows) and so we do not need it here, which is why we are setting them all to 0.

Step #3: Adjusting the TextView

We can reuse the `TextView` that came in the starter project, but we need to make a few changes to it. However, to change it, we need to select it first, and now it is covered by the `RecyclerView` that we just added. Instead, click on the `TextView` entry in the "Component Tree" pane of the Layout Designer:

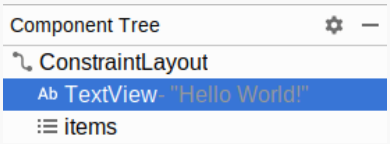


Figure 56: Android Studio Layout Designer, Component Tree Pane

Then, in the "Attributes" pane, fill in `empty` for the ID. Then, click on the "..." button to the side of the "text" field that has "Hello World!" as its current value:

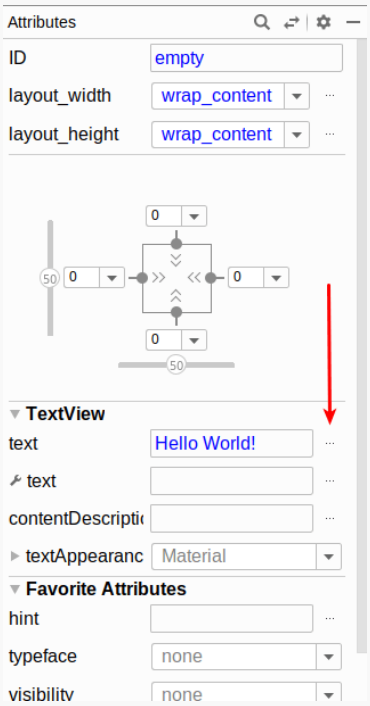


Figure 57: Android Studio Layout Designer, Attributes Pane, with Arrow Pointing to Button

This will bring up a dialog showing available string resources:

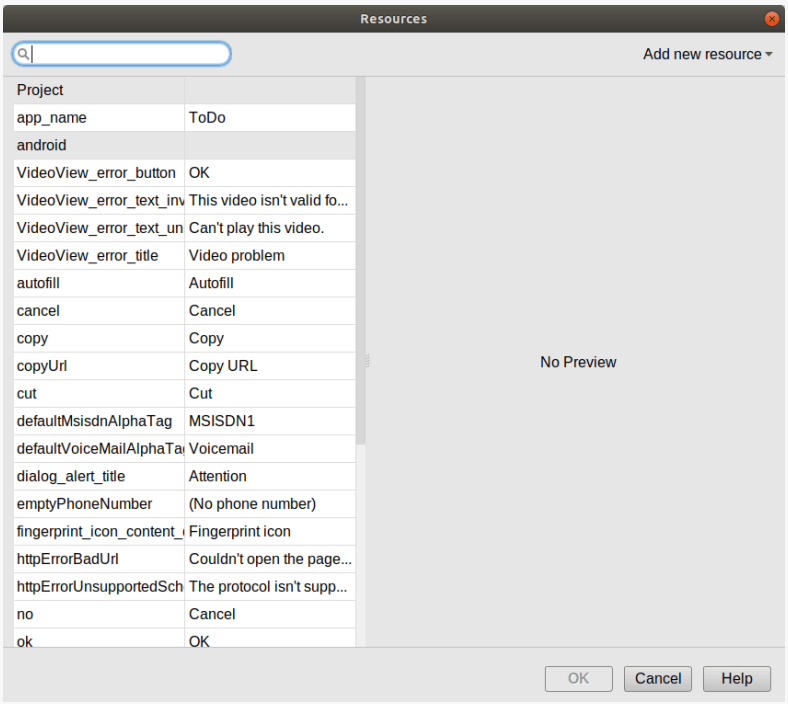


Figure 58: Android Studio String Resource Selector

Click the “Add new resource” drop-down towards the top, and in there choose “New string Value”. This brings up a dialog to define a new string resource:

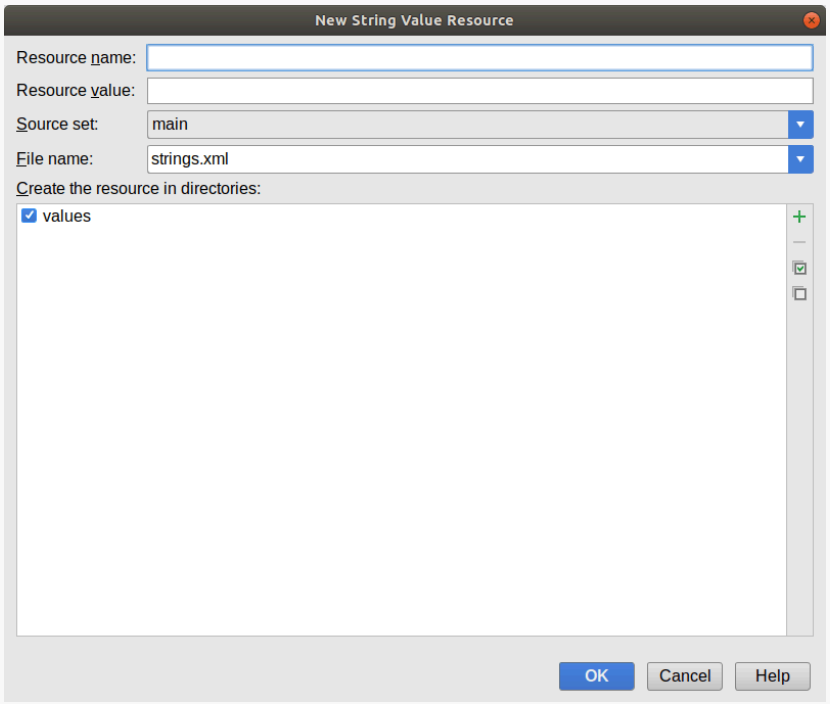


Figure 59: Android Studio New String Resource Dialog

For the “Resource name”, fill in `msg_empty`. For the “Resource value”, fill in “placeholder text”:

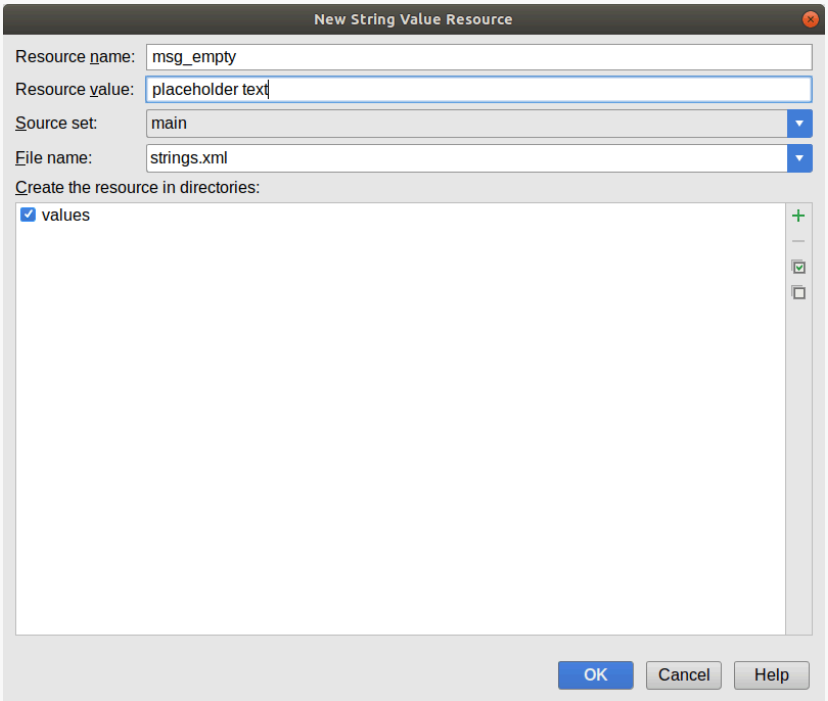


Figure 60: Android Studio New String Resource Dialog, with Values

As the text suggests, this is a placeholder for a better message that we will swap in later in this book.

Click OK to define the resource, and you should be taken back to the designer.

Switch back to the “Text” subtab of the editor, where you can see the XML of the layout. Add `android:textAppearance="?android:attr/textAppearanceMedium"` as an attribute to the `<TextView>` element:

```
<TextView
  android:id="@+id/empty"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="@string/msg_empty"
  android:textAppearance="?android:attr/textAppearanceMedium"
  app:layout_constraintBottom_toBottomOf="parent"
  app:layout_constraintLeft_toLeftOf="parent"
  app:layout_constraintRight_toRightOf="parent"
```

```
    app:layout_constraintTop_toTopOf="parent" />
```

(from To6-Layout/ToDo/app/src/main/res/layout/activity_main.xml)

This says "we want this text to be in the standard medium text size for whatever overall UI theme we happen to be using".

This, then, gives us what we were seeking from the outset: the `RecyclerView`, and the `TextView`, all properly configured and positioned:

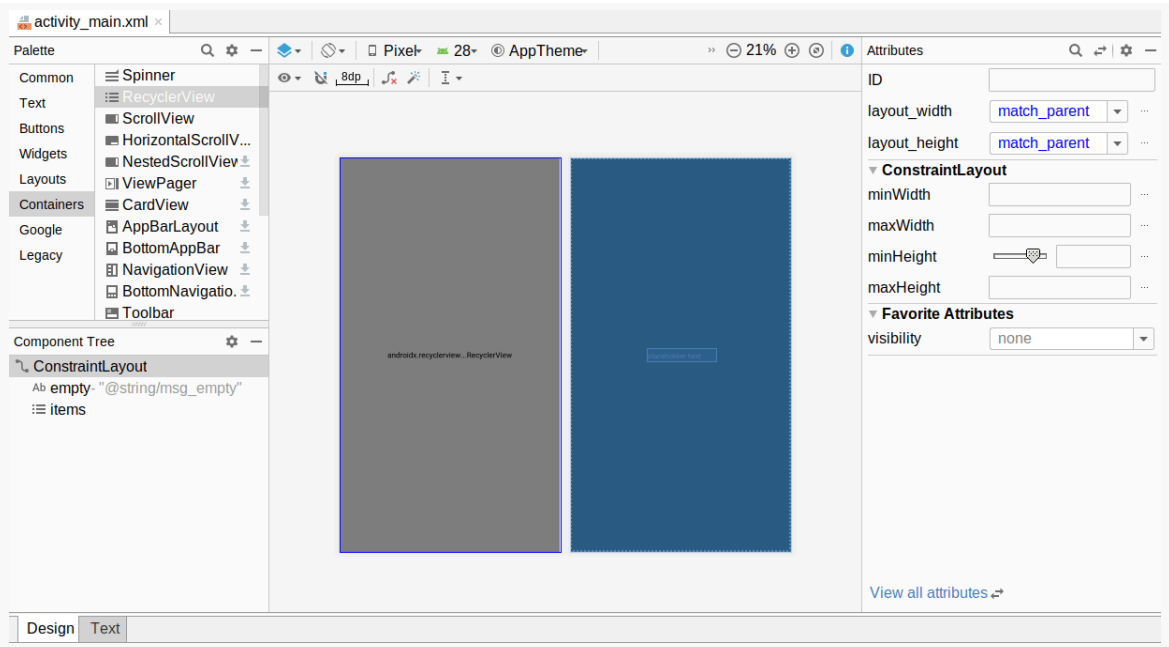


Figure 61: Android Studio Layout Designer, Showing End Result of This Tutorial

```
<?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:id="@+id/empty"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/msg_empty"
    android:textAppearance="?android:attr/textAppearanceMedium"
```

```
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <androidx.recyclerview.widget.RecyclerView
    android:id="@+id/items"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

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

(from To6-Layout/ToDo/app/src/main/res/layout/activity_main.xml)

If you run the app, since the `MainActivity` loads up this layout resource via `setContentView(R.layout.activity_main)`, you will see the "placeholder text" and nothing else:

Figure 62: ToDo App, As Currently Implemented

We have not put anything into the `RecyclerView`, so it has no content for us to see.

What We Changed

The book's GitHub repository contains the entire result of having completed this tutorial. In particular, it contains the changed files:

- `app/src/main/res/layout/activity_main.xml`

Setting Up the App Bar

Next up is to configure the app bar in our ToDo application. The app bar is that bar at the top of your activity UI, showing your app's title. It can also have toolbar-style buttons and an "overflow menu", each holding what are known as action items.

Google has made a bit of a mess of this app bar over the years, mixing the terms "app bar", "action bar", and "toolbar". This book will tend to use:

- `Toolbar`, in monospace, when referring to the actual `Toolbar` class
- "App bar", when referring to the concept of this bar
- "toolbar buttons", when referring to the icons that can appear in this bar that the user can tap on to perform actions

In this tutorial, we will add a `Toolbar` to our UI that will serve as our app bar. In that `Toolbar`, we will add an action item to the overflow menu to launch an "about" page, though we will not actually show that page until a later tutorial. And, along the way, we will update our app's theme with a new color scheme.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.

Starting in this tutorial, we will begin editing Kotlin source files. Some useful Android Studio shortcut key combinations are:

- Alt-Enter (Option-Return on macOS) for bringing up quick-fixes for the problem at the code where the cursor is.
- Ctrl-Alt-O (Command-Option-O on macOS) will organize your Java `import` statements, including removing unused imports.

- Ctrl-Alt-L (Command-Option-L on macOS) will reformat the Kotlin or XML in the current editing window, in accordance with either the default styles in Android Studio or whatever you have modified them to in Settings.

NOTE: Copying and pasting Kotlin code from this book may or may not work, depending on what you are using to read the book. For the PDF, some PDF viewers (e.g., Adobe Reader) should copy the code fairly well; others may do a much worse job. Reformatting the code with Ctrl-Alt-L (Command-Option-L on macOS) after pasting it in sometimes helps.

You can learn more about styles and themes in the "Defining and Using Styles" chapter of *Elements of Android Jetpack*!

You can learn more about `Toolbar` in the "Configuring the App Bar" chapter of *Elements of Android Jetpack*!

Step #1: Defining Some Colors

Just as Android has layout, drawable, and string resources, Android has color resources. We can define some colors in a resource file, then apply those colors elsewhere in our app.

By convention, colors are defined in a `colors.xml` file. Colors are considered “value” resources, like our strings, and so the file would go into `res/values/colors.xml`.

But, we need to choose some colors.

To that end, visit https://www.materialpalette.com/, which offers a very simple point-and-click way of setting up a color palette for use in an Android app:

Figure 63: Material Design Palette Site, As Initially Launched

For the purposes of this tutorial, click on “Yellow”, then “Light Blue”:

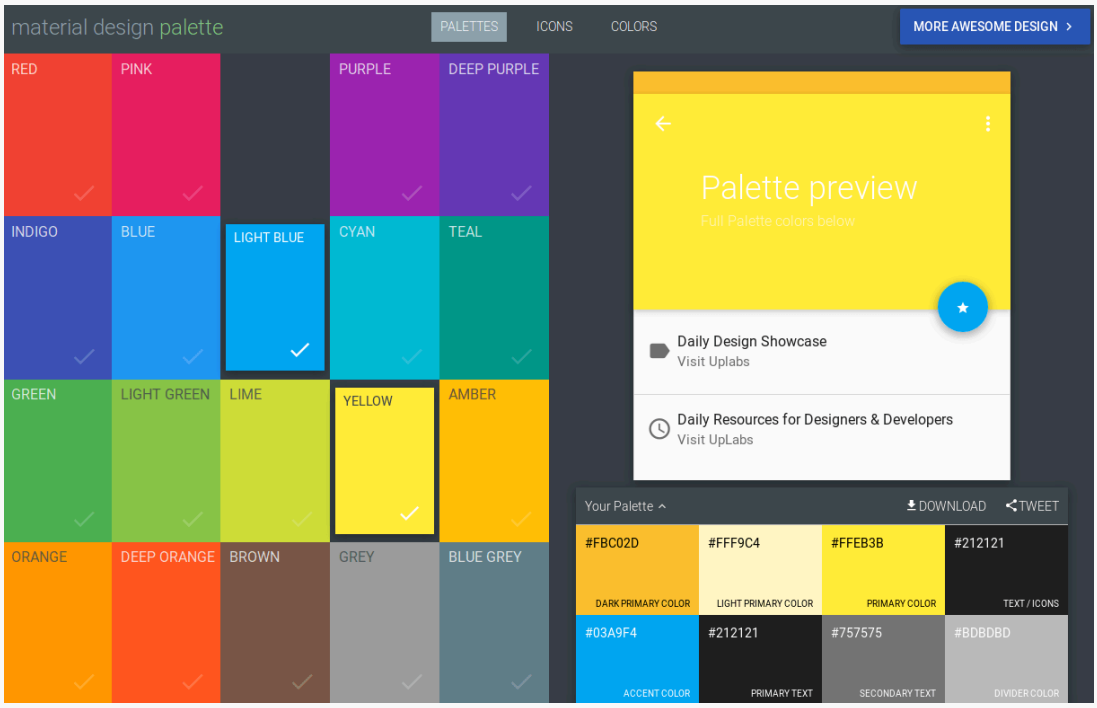


Figure 64: Material Design Palette Site, With Yellow/Light Blue Colors

The site assumes that the app bar will be dark with light text, but that is merely a limitation of the site. We will teach Android to use dark text, and so the white-on-yellow effect seen here is not going to be a problem.

Then, click the “Download” button in the “Your Palette” area, and choose “XML” as the type of file to download. This will trigger your browser to download a file named `colors_yellow_light_blue.xml`. Open in it your favorite text editor. You should see:

```
<!-- Palette generated by Material Palette - materialpalette.com/yellow/light-blue -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="primary">#FFEB3B</color>
  <color name="primary_dark">#FBC02D</color>
  <color name="primary_light">#FFF9C4</color>
  <color name="accent">#03A9F4</color>
  <color name="primary_text">#212121</color>
  <color name="secondary_text">#757575</color>
  <color name="icons">#212121</color>
  <color name="divider">#BDBDBD</color>
</resources>
```

Then, in Android Studio, open your existing `res/values/colors.xml` file, which will have three colors already defined:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <color name="colorPrimary">#FFEB3B</color>
  <color name="colorPrimaryDark">#FBC02D</color>
  <color name="colorAccent">#03A9F4</color>
</resources>
```

(from T07-Toolbar/ToDo/app/src/main/res/values/colors.xml)

The file from the Material Palette site has colors for the same roles as Android Studio uses, but with slightly different names (e.g., `primary` instead of `colorPrimary`). In the end, the names do not matter all that much. For the purposes of this tutorial, we will use Android Studio's names.

With that in mind, adjust `res/values/colors.xml` to use the colors from the Material Palette site:

- Change `colorPrimary` to `#FFEB3B`
- Change `colorPrimaryDark` to `#FBC02D`
- Change `colorAccent` to `#03A9F4`

You will see that the Android Studio color resource editor contains color swatches in the "gutter" area, adjacent to each of the color values:

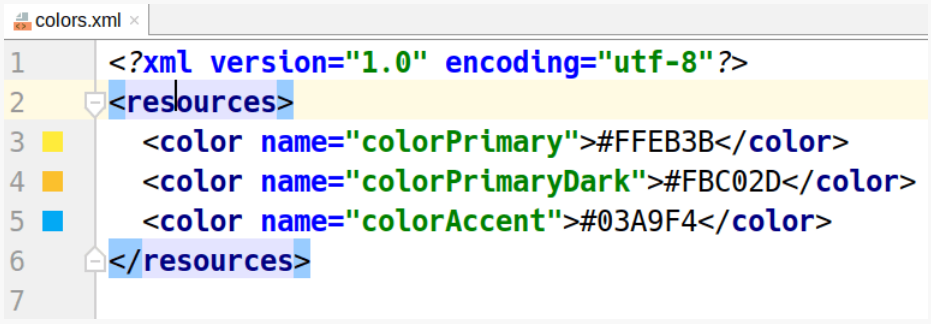


Figure 65: Android Studio Values Resource Editor, with Color Swatches

The color swatches are clickable and will bring up a color picker, if you wanted to change any of the colors a bit from what the site gave you:

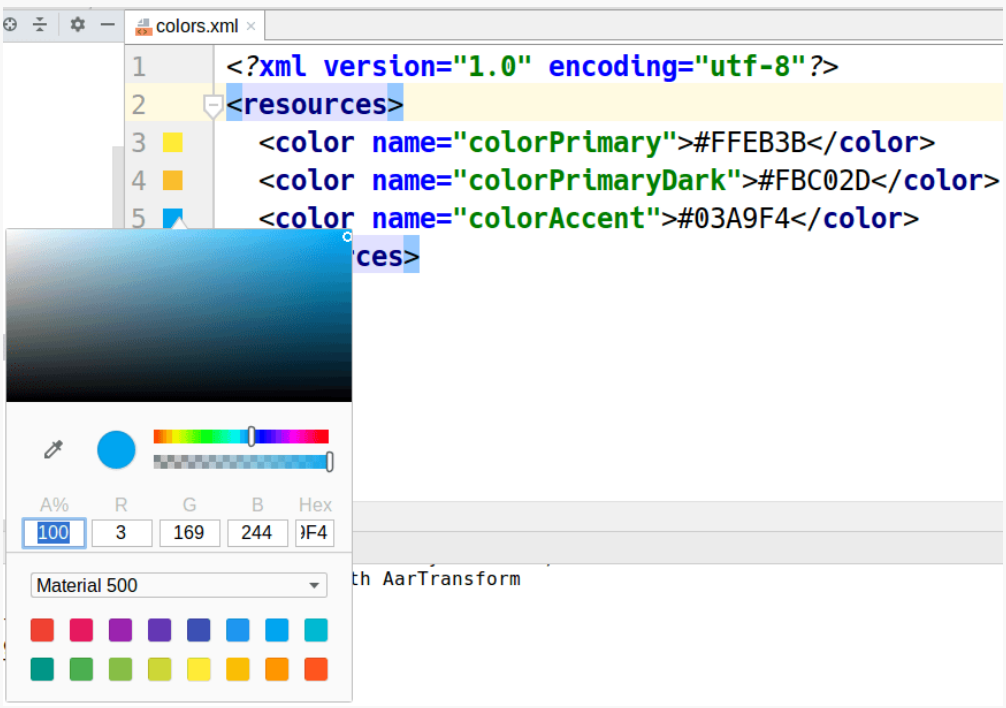


Figure 66: Android Studio Color Picker

Step #2: Adjusting Our Theme

The app bar color is one aspect of our app that is managed by a theme. A theme provides overall "look and feel" instructions for our activity, including the app bar color.

Your project already has a custom theme declared. If you look in your `res/values/` directory, you will see a `styles.xml` file — open that in Android Studio:

```
<resources>

  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.Light.DarkActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
  </style>
```

```
</resources>
```

Here, we see that we have a style resource named `AppTheme`. Style resources can be applied either to widgets (to tailor that particular widget) or as a theme to an activity or entire application. By convention, style resources with "Theme" in the name are themes. This particular theme inherits from `Theme.AppCompat.Light.DarkActionBar`, as indicated in the `parent` attribute. And, it associates our three colors with three roles in the theme:

- `colorPrimary` will be the dominant color and will be the background color of the app bar
- `colorPrimaryDark` mostly is used for coloring the status bar (the bar at the top of the screen that has the time, battery level, signal strength, etc.)
- `colorAccent` will be used for certain pieces of widgets, such as the text-selection cursor in `EditText` widgets

There are two problems with our default theme:

- We will be using a light-colored app bar, not a dark one
- We will be configuring the app bar ourselves with a `Toolbar`, so we do not need the theme to add one for us.

So, replace the `DarkActionBar` portion of that `parent` value with `NoActionBar`, leaving you with `Theme.AppCompat.Light.NoActionBar`.

The resulting resource should look like:

```
<resources>

  <!-- Base application theme. -->
  <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
    <!-- Customize your theme here. -->
    <item name="colorPrimary">@color/colorPrimary</item>
    <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
    <item name="colorAccent">@color/colorAccent</item>
  </style>

</resources>
```

(from To7-Toolbar/ToDo/app/src/main/res/values/styles.xml)

Note that the color swatches in the gutter in this Android Stuido are non-clickable, so you cannot edit the colors directly from the style resource.

Our `AndroidManifest.xml` file already ties in this custom theme, via the `android:theme` attribute in the `<application>` element:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="com.commonsware.todo"
  xmlns:android="http://schemas.android.com/apk/res/android">
  <supports-screens
    android:largeScreens="true"
    android:normalScreens="true"
    android:smallScreens="true"
    android:xlargeScreens="true"/>

  <application
    android:allowBackup="false"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>

</manifest>
```

(from To7-Toolbar/ToDo/app/src/main/AndroidManifest.xml)

Step #3: Adding a `Toolbar`

Our app bar will be in the form of a `Toolbar` widget. This is an ordinary widget that you can put in a layout wherever it needs to go. Traditionally, the app bar appears at the top of the activity, so we will place one there.

Open `res/layout/activity_main.xml` in Android Studio. Right now, this contains our `RecyclerView` for our to-do items and a `TextView` to use for the "empty state" (what to show when there are no to-do items in the list). Now, we want to modify the layout to have a `Toolbar` at the top.

In the Containers category of the "Palette", you should find a Toolbar option:

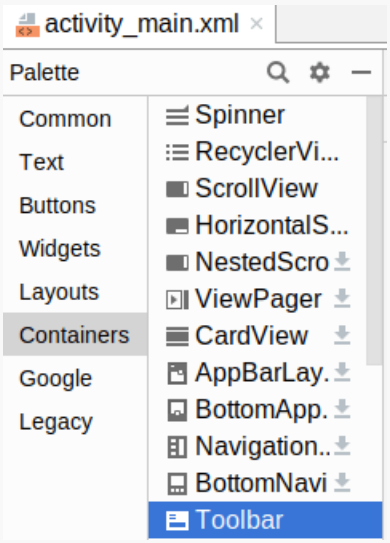


Figure 67: Android Studio Palette, Showing `Toolbar`

Drag one from the "Palette" over the `ConstraintLayout` in the "Component Tree" view to add it as a child widget:

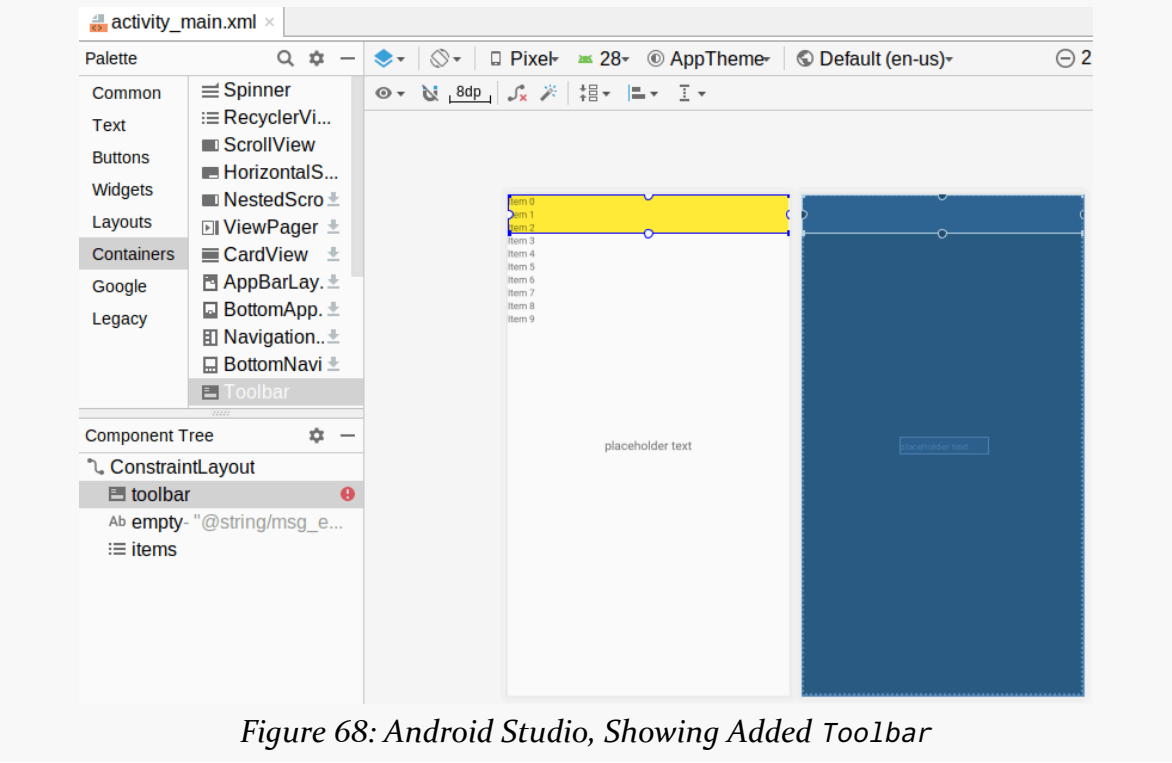


Figure 68: Android Studio, Showing Added `Toolbar`

Next, click on the `items` entry in the "Component Tree" to select the `RecyclerView`. You should see it be connected with the bounds of the `ConstraintLayout` on all four sides. Grab the top anchor and drag it down until it connects with the bottom of the `Toolbar`:

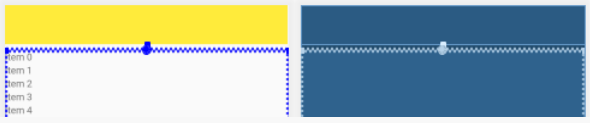


Figure 69: Android Studio, Showing `RecyclerView` Top Anchored to `Toolbar` Bottom

We want to do the same thing to the `empty` `TextView`. This time, we'll take a different approach to making the change. Click on the `empty` entry in the "Component Tree" to select the `TextView`, then click on the circle on the top of the `TextView`. This will eliminate the existing anchor rule on that side, causing the `TextView` to fall to the bottom of the screen:

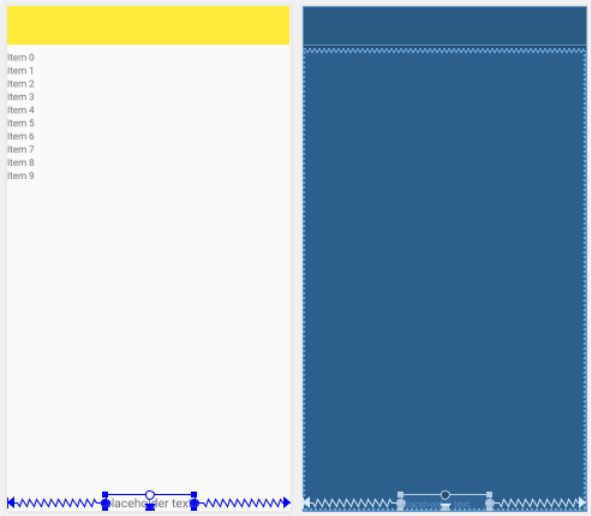


Figure 70: Android Studio, Showing Collapsed `TextView`

Then, you should be able to grab that circle and drag it to the bottom of the `Toolbar`... or perhaps to the top of the `RecyclerView`:

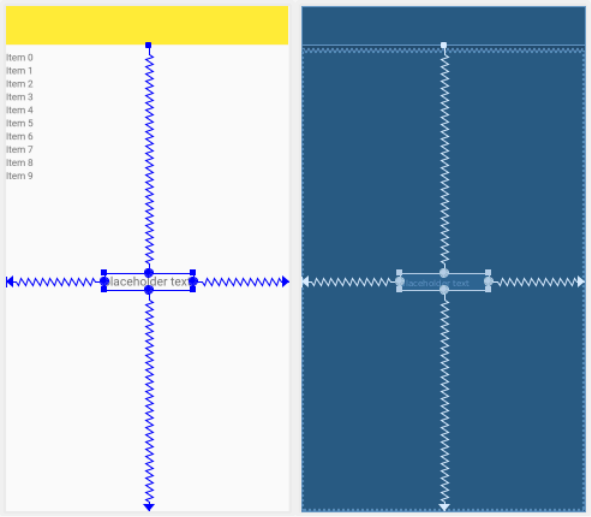


Figure 71: Android Studio, Showing Repositioned `TextView`

Whether you wind up tying the top of the `TextView` to the top of the `RecyclerView` or the bottom of the `Toolbar` does not matter.

Next, we need to set up the anchoring rules for the `Toolbar` itself. It *looks* like it is in the correct position but that is just the default behavior. We really should set up the rules properly. So, grab the circles on the start, top, and end sides of the `Toolbar` and connect them with the start, top, and end sides of the `ConstraintLayout`. Leave the bottom alone.

Then, in the "Attributes" pane:

- Ensure that the ID is set to `toolbar` (it should be by default)
- Set the `layout_width` to `match_constraint` (a.k.a., `0dp`)
- Clear the margins on the start, top, and end sides

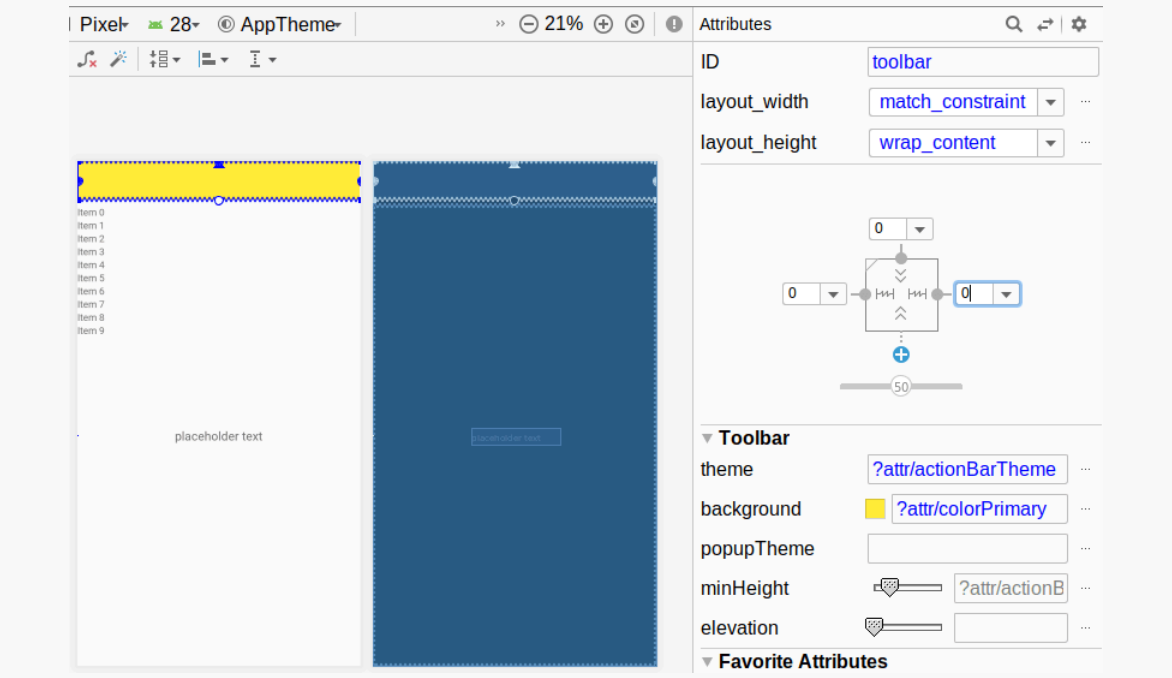


Figure 72: Android Studio, Showing Anchored and Configured `Toolbar`

In theory, your layout XML should now look like:

```
<?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">

  <androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    android:minHeight="?attr/actionBarSize"
    android:theme="?attr/actionBarTheme"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <TextView
    android:id="@+id/empty"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="8dp"
    android:text="@string/msg_empty"
    android:textAppearance="?android:attr/textAppearanceMedium"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/toolbar" />

  <androidx.recyclerview.widget.RecyclerView
    android:id="@+id/items"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginTop="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/toolbar"
    app:layout_constraintVertical_bias="1.0" />

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

(from T07-Toolbar/ToDo/app/src/main/res/layout/activity_main.xml)

If the `Toolbar` constraints (e.g., `app:layout_constraintEnd_toEndOf`) connect to `@id/items` instead of `parent`, that is OK, or you can change them manually here to match the code used in the book. This illustrates why the drag-and-drop approach is OK but far from perfect: sometimes, you cannot easily connect the anchor to what

you want just using the mouse.

Step #4: Adding an Icon

We are going to need a an icon for our app bar item. Nowadays, the preferred approach for doing this is to start with vector drawables, rather than bitmaps, to reduce the size of the app and maximize the quality of the icons when they are displayed.

Right-click over the `res/` directory and choose New > "Vector Asset" from the context menu. This brings up the first page of the vector asset wizard:

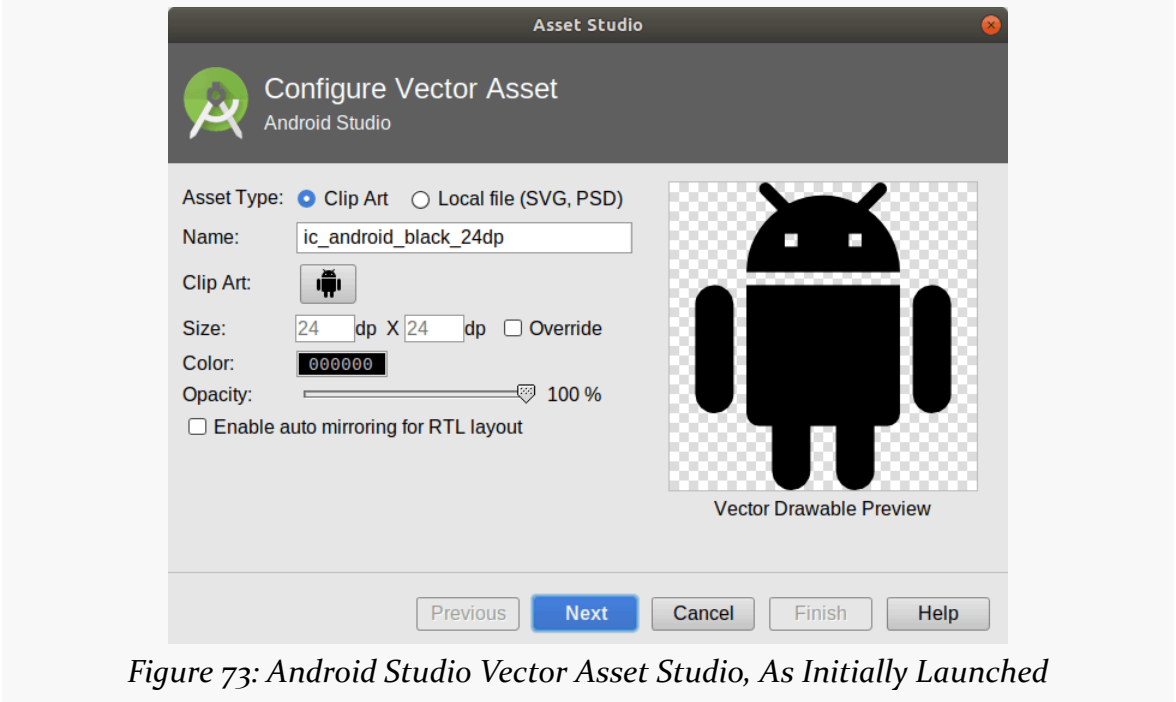


Figure 73: Android Studio Vector Asset Studio, As Initially Launched

Click on the "Clip Art" button, which by default has the image of the Android mascot ("bugdroid"). This will bring up an icon selector, with a bunch of icons from Google's "Material Design" art library. In the search field, type `info`, then click on the "info outline" icon:

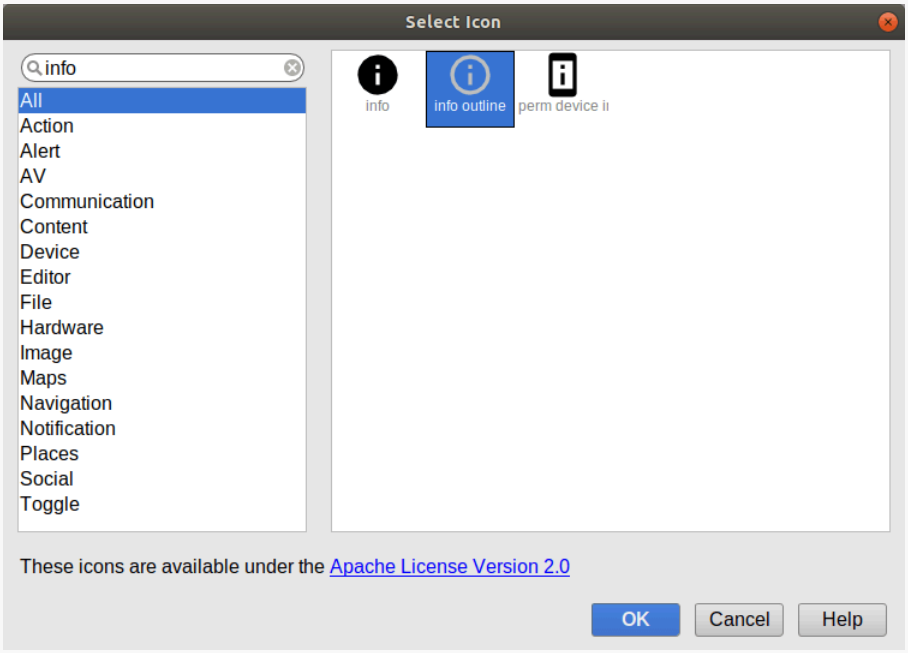


Figure 74: Android Studio Icon Selector, Showing "info outline" Icon

Click "OK". This will update the name of the asset to `ic_info_outline_black_24dp`.

Click Next, then Finish, to add that icon as an XML file in `res/drawable/`.

Step #5: Defining an Item

Next, we will add a low-priority action item, for an "about" screen.

Right click over the `res/` directory in your project, and choose New > "Android resource directory" from the context menu. This will bring up a dialog to let you create a new resource directory:

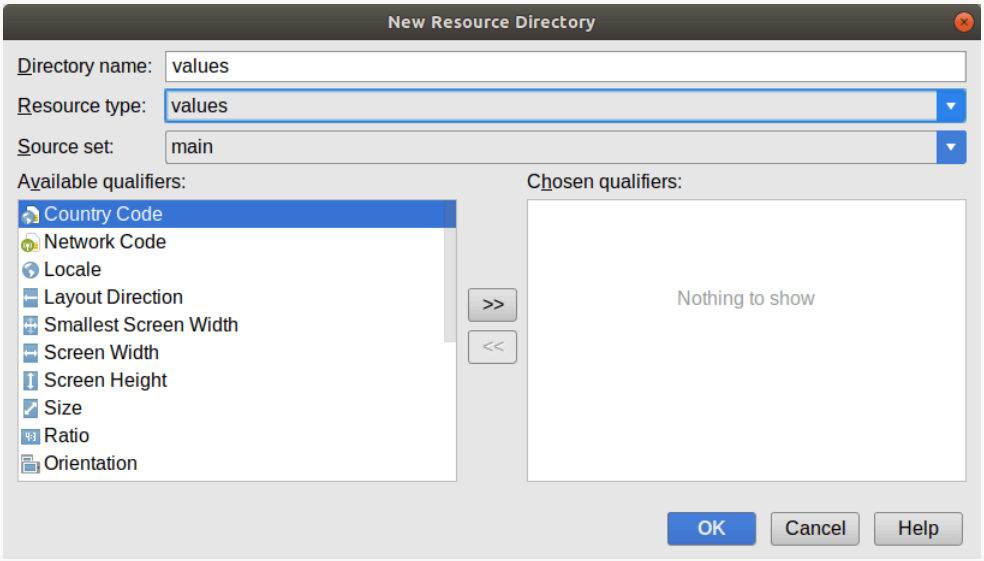


Figure 75: Android Studio New Resource Directory Dialog

Change the "Resource type" drop-down to be "menu", then click OK to create the directory.

Then, right-click over your new `res/menu/` directory and choose New > "Menu resource file" from the context menu. Fill in `actions.xml` in the "New Menu Resource File" dialog:

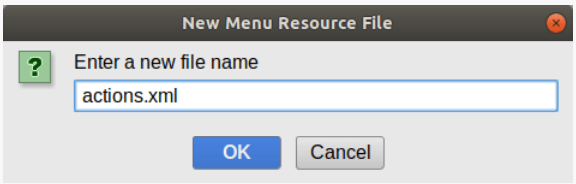


Figure 76: Android Studio New Menu Resource File Dialog

Then click OK to create the file. It will open up into a menu editor:

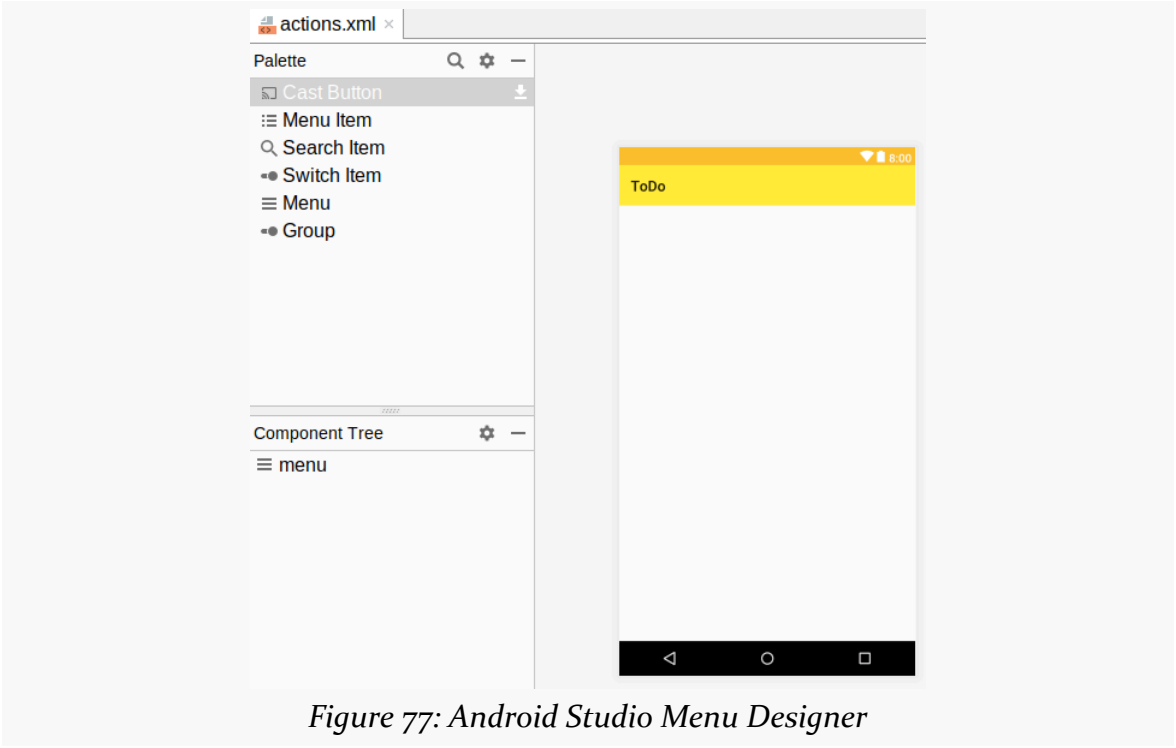


Figure 77: Android Studio Menu Designer

This editor looks and works a lot like the layout editor. The "Palette" contains things that can be dragged-and-dropped into the menu. The "Component Tree" shows the current contents of the menu. The preview area shows visually what this looks like, and the "Attributes" pane (not shown in the above screenshot) shows attributes of the selected item in the "Component Tree".

In the “Palette” view, drag a “Menu Item” into the preview area over the right end of the app bar. This will appear as an item in an overflow area:

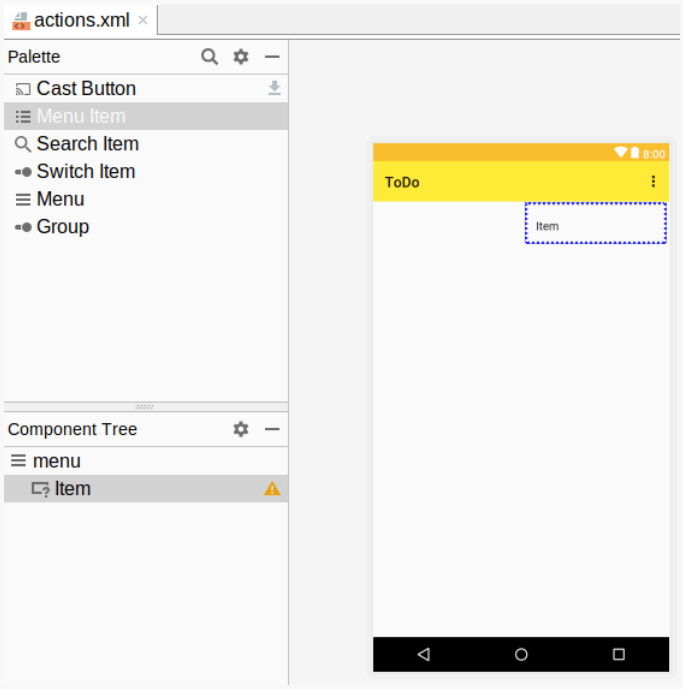


Figure 78: Android Studio Menu Designer, with a New Menu Item

In the Attributes pane, fill in:

- `about` for the “id”
- “never” for “showAsAction”

To set the “showAsAction” value, click the field in the “Attributes” pane, which will bring up a dialog with the available options:

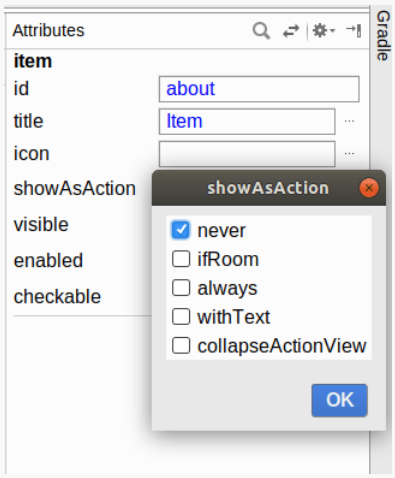


Figure 79: Android Studio Menu Designer, Attributes Pane, Showing “showAsAction” Options

Then, click on the "..." button next to the "icon" field. This will bring up a drawable resource selector:

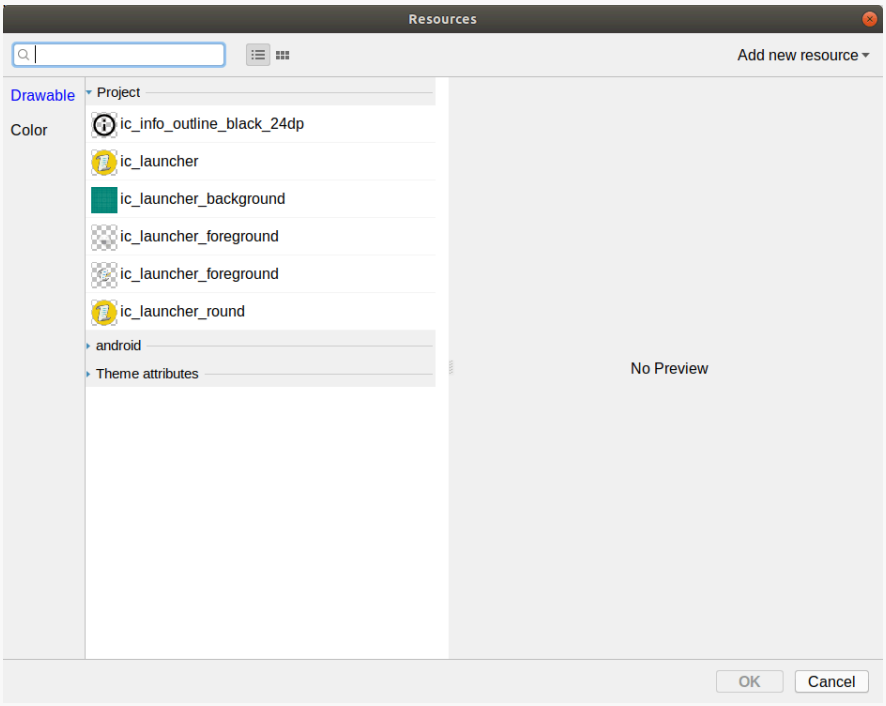


Figure 80: Android Studio Drawable Resource Selection Dialog

Click on `ic_info_outline_black_24dp` in the list of "Project" drawables, then click OK to accept that choice of icon. In truth, this is unnecessary, as our item should never show the icon. But, you never know when someday Google will decide to show icons for overflow menu items, so it is best to define one.

Then, click the "..." button next to the "title" field. As before, this brings up a string resource selector. Click on "Add new resource" > "New string Value" in the drop-down towards the top. In the dialog, fill in `menu_about` as the resource name and "About" as the resource value:

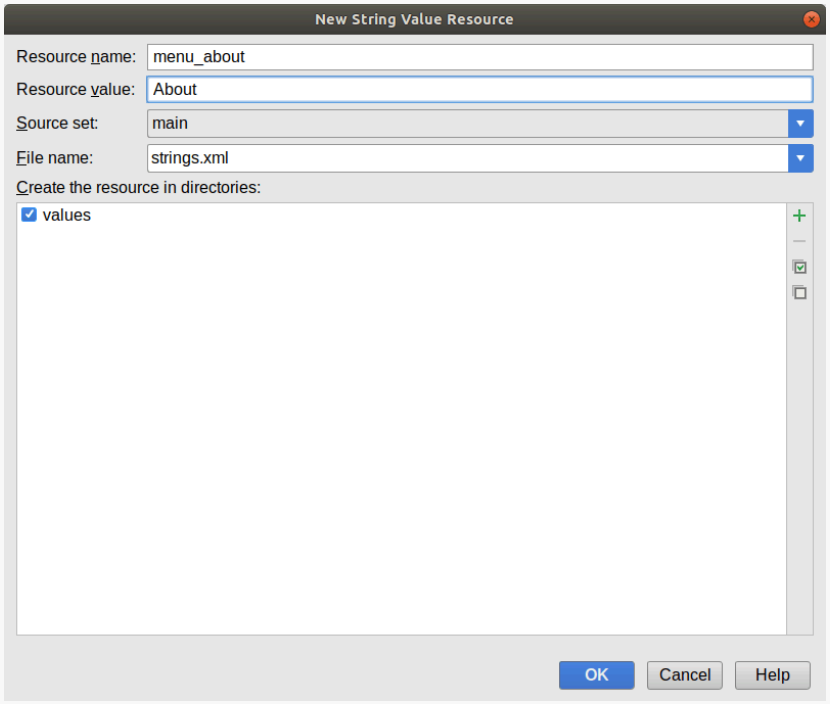


Figure 81: Android Studio New String Resource Dialog

Click OK to close the dialog, and you will see your new title appear in the menu editor:

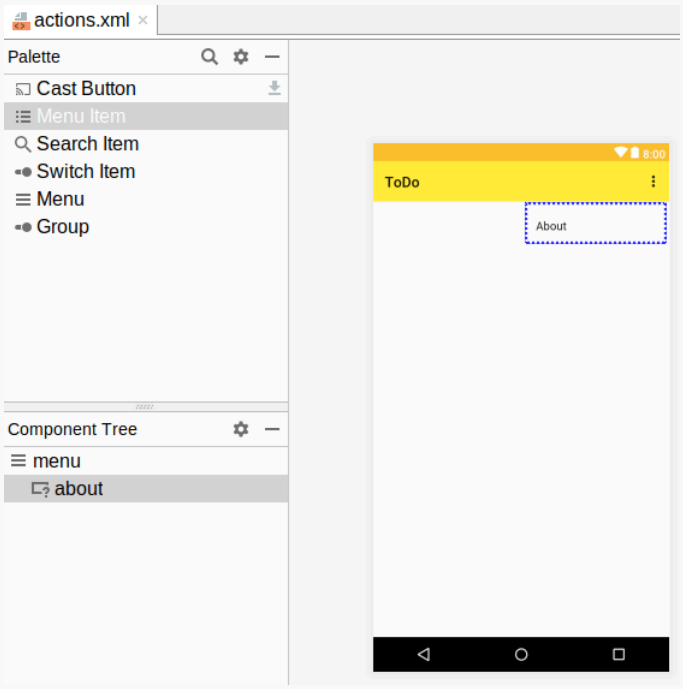


Figure 82: Android Studio Menu Designer, Showing About Item

Step #6: Enabling Kotlin Synthetic Attributes

We are going to need to work with our widgets from our Kotlin code. There are several approaches for doing that, such as using `findViewById()` calls to retrieve the widgets. However, for Kotlin-based Android app development, a popular approach is to use "Kotlin synthetic properties", where the Kotlin plugin for Android code-generates stuff to give us direct access to our widgets.

However, that is an experimental feature at this time, so we need to opt into using it.

To that end, add the following closure to the `android` closure of your `app/build.gradle` file:

```
    androidExtensions {
        experimental = true
    }
```

(from To7-Toolbar/ToDo/app/build.gradle)

In the end, your overall `app/build.gradle` file should resemble:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'

android {
    compileSdkVersion 28

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdkVersion 21
        targetSdkVersion 28
        versionCode 1
        versionName "1.0"
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }

    androidExtensions {
        experimental = true
    }
}

dependencies {
    implementation"org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.1'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
    testImplementation 'junit:junit:4.12'
    androidTestImplementation 'androidx.test:runner:1.1.1'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
}
```

(from To7-Toolbar/ToDo/app/build.gradle)

Step #7: Loading Our Options

Simply defining `res/menu/actions.xml` is insufficient. We need to actually tell Android to use what we defined in that file and show it in our `Toolbar`.

Go into the project tree, and drill down into the `java/` directory to find the `com.commonsware.todo` package and the `MainActivity` class inside of it:

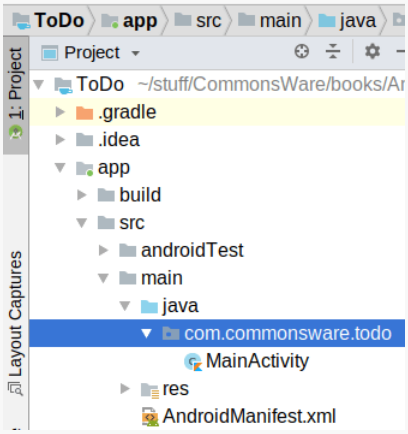


Figure 83: Android Studio Project Tree, Showing MainActivity

Double-click on `MainActivity` to open it in an editor. Modify it to contain the following code:

```
package com.commonsware.todo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    toolbar.title = getString(R.string.app_name)
    toolbar.inflateMenu(R.menu.actions)
  }
}
```

(from To7-Toolbar/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt)

This adds two lines to `onCreate()`. Both call functions on a `toolbar` object. `toolbar` is a reference to the widget with the `android:id` value of `toolbar` from our layout. The `toolbar` `import` comes from:

```
import kotlinx.android.synthetic.main.activity_main.*
```

(from To7-Toolbar/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt)

This is available to us courtesy of those Kotlin synthetic properties that we enabled in the previous step. Here:

- `kotlinx.android.synthetic` is the top-level package for all of these synthetic properties
- `main` refers to our `main` source set, where all our code resides
- `activity_main` refers to the layout resource that we are using

If you start typing `toolbar` into the `onCreate()` function, you should get an auto-complete option that will add the synthetic properties `import` statement for you.

Our two lines:

- Sets the title of our `Toolbar` to be the contents of the `app_name` string resource (whose value is retrieved by calling `getString()` on our activity)
- "Inflate" the menu resource that we created earlier in the project, causing the `Toolbar` to show the items that we have defined in it

Step 8: Trying It Out

If you run the app, you should see a "..." icon on the action bar:

Figure 84: ToDo App, Showing Overflow Menu Affordance

Pressing that brings up a menu showing our “About” item:

Figure 85: ToDo App, Showing Overflow Menu with About Item

Tapping that item has no effect — we will address that in an upcoming tutorial.

Step #9: Dealing with Crashes

Most likely, you will not need this step.

But, sometimes, when writing Android apps, you will make mistakes. Your code will compile, but then it will crash at runtime. A crash is signaled by a dialog indicating that there was a problem. The look of that dialog varies by Android version, but a typical one is:

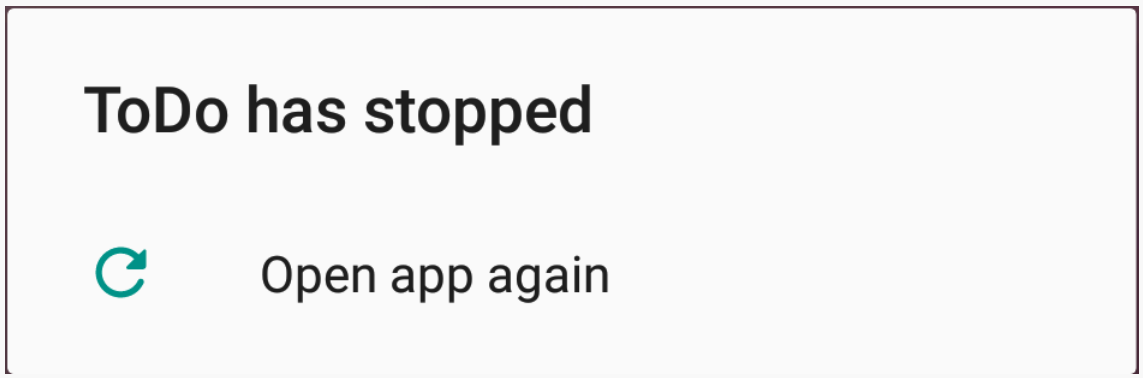


Figure 86: Crash Dialog, on Android 8.1

When that occurs, you can find out more about the crash by opening the LogCat tool in Android Studio. By default, this is docked along the lower edge. Opening it gives you access to all sorts of messages logged by apps and the operating system.

There will be *lots* of messages.

Ideally, Android Studio would help you narrow down the messages. It offers a couple of things for that:

- There is a message "severity" drop down (third from left in the screenshot below), showing options like "Verbose" and "Error" — crashes are logged at "Error" severity
- The end drop-down will default to "Show only selected application", which will then (theoretically) limit the output to only messages logged by your app, or by whatever app is shown in the second drop-down

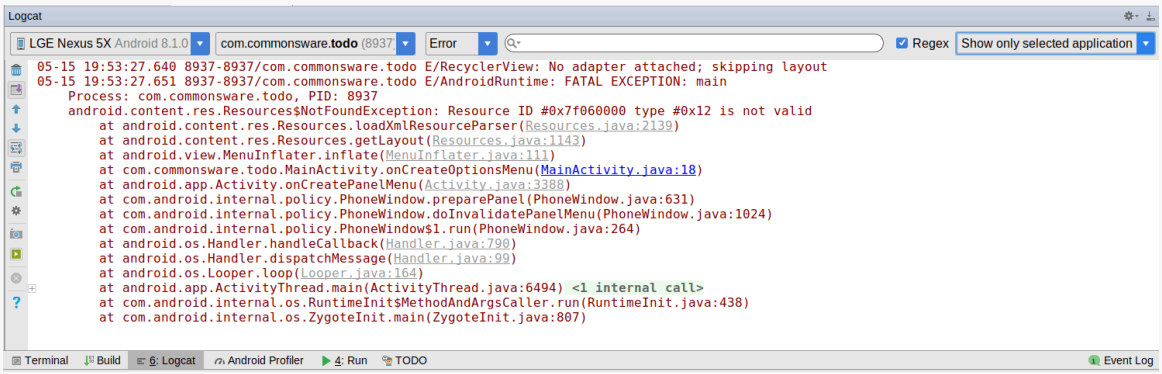


Figure 87: LogCat, Showing Stack Trace

When you crash, you will get a red Java stack trace showing what went wrong:

```
8937-8937/com.commonsware.todo E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.commonsware.todo, PID: 8937
    android.content.res.Resources$NotFoundException: Resource ID #0x7f060000 type
#0x12 is not valid
        at android.content.res.Resources.loadXmlResourceParser(Resources.java:2139)
        at android.content.res.Resources.getLayout(Resources.java:1143)
        at android.view.MenuInflater.inflate(MenuInflater.java:111)
        at com.commonsware.todo.MainActivity.onCreateOptionsMenu(MainActivity.kt:14)
        at android.app.Activity.onCreatePanelMenu(Activity.java:3388)
        at com.android.internal.policy.PhoneWindow.preparePanel(PhoneWindow.java:631)
        at
com.android.internal.policy.PhoneWindow.doInvalidatePanelMenu(PhoneWindow.java:1024)
        at com.android.internal.policy.PhoneWindow$1.run(PhoneWindow.java:264)
        at android.os.Handler.handleCallback(Handler.java:790)
        at android.os.Handler.dispatchMessage(Handler.java:99)
        at android.os.Looper.loop(Looper.java:164)
        at android.app.ActivityThread.main(ActivityThread.java:6494)
        at java.lang.reflect.Method.invoke(Native Method)
        at
com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:438)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:807)
```

In this case, this comes from a modified version of this sample app, hacked to

introduce a crash. Typically, you look for the top-most line that refers to your code. In this case, that is:

```
        at com.commonsware.todo.MainActivity.onCreateOptionsMenu(MainActivity.kt:14)
```

The location (`MainActivity.kt:14`) will be a link that you can click to jump to that particular line of code. That, plus the error message, will hopefully help you diagnose exactly what went wrong.

What We Changed

The book's GitHub repository contains the entire result of having completed this tutorial. In particular, it contains the changed files:

- app/src/main/res/menu/actions.xml
- app/src/main/res/values/colors_yellow_light_blue.xml
- app/src/main/res/values/styles.xml
- app/src/main/java/com/commonsware/todo/MainActivity.kt

Setting Up an Activity

Of course, it would be nice if that “About” menu item that we added in a previous tutorial actually did something.

In this tutorial, we will define another activity class, one that will be responsible for the “about” details. And, we will arrange to start up that activity when that menu item is selected.

This is a continuation of the work we did in the previous tutorial. The book’s GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.

You can learn more about having multiple activities in the "Implementing Multiple Activities" chapter of *Elements of Android Jetpack*!

Step #1: Creating the Stub Activity Class and Manifest Entry

First, we need to define the Kotlin class for our new activity, `AboutActivity`.

Right-click on your `main/` source set directory in the project explorer, and choose New > Activity > Empty Activity from the context menu. This will bring up a new-activity wizard:

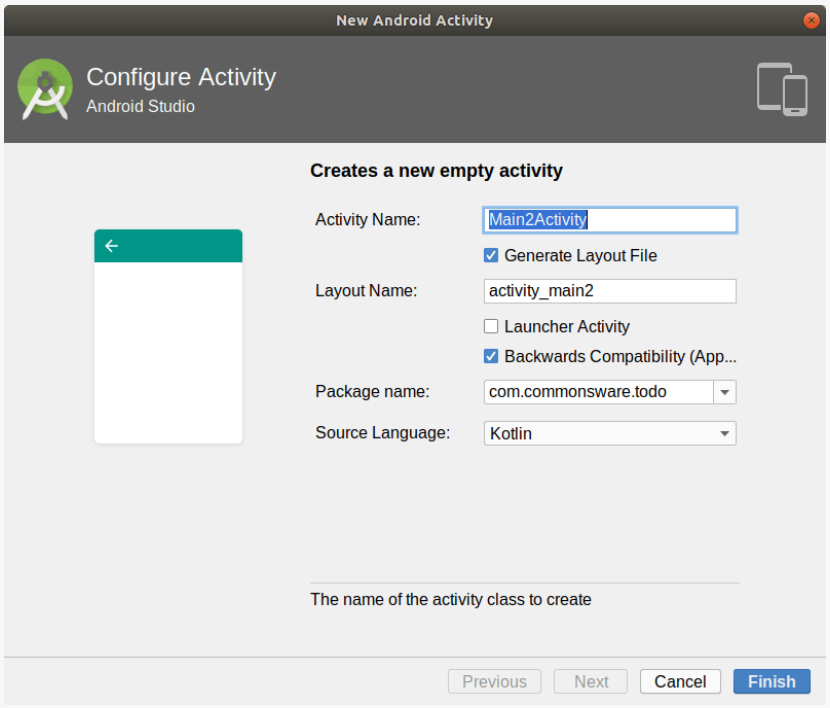


Figure 88: Android Studio New-Activity Wizard, As Initially Launched

Fill in `AboutActivity` in the "Activity Name" field. Leave "Launcher Activity" unchecked. If the package name drop-down is showing the app's package name (`com.commonsware.todo`), leave it alone. On the other hand, if the package name drop-down is empty, click on it and choose the app's package name. Leave the source language drop-down set to Kotlin.

This should give you a dialog like:

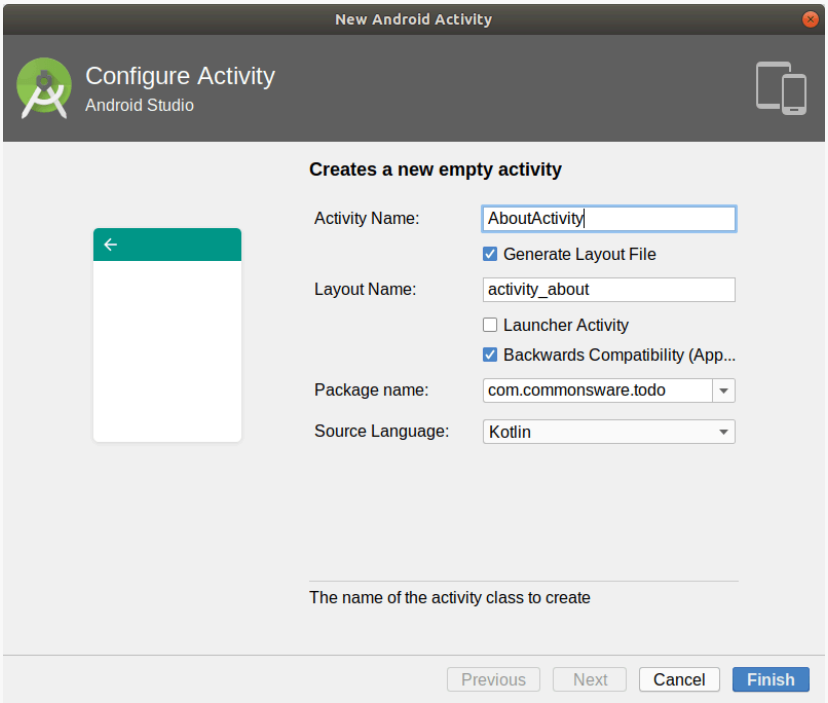


Figure 89: Android Studio New-Activity Wizard, Filled In

If you click on Finish, Android Studio will create your `AboutActivity` class and open it in the editor. The source code should look like:

```
package com.commonsware.todo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle

class AboutActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_about)
  }
}
```

The new-activity wizard also added a manifest entry for us:

```
    <activity android:name=".AboutActivity"></activity>
```

(from To8-Activities/ToDo/app/src/main/AndroidManifest.xml)

Step #2: Adding a `Toolbar` and a `WebView`

In addition to a new `AboutActivity` Java class and manifest entry, the new-activity wizard created an `activity_about` layout resource for us, alongside the existing `activity_main` layout. Open `activity_about` into the graphical layout editor.

As we did in the previous tutorial, in the "Palette", choose the "Containers" category, and drag a `Toolbar` into the preview area:

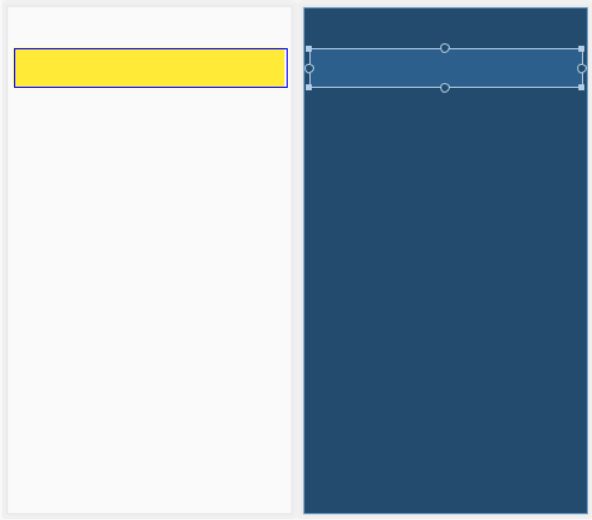


Figure 90: Android Studio Layout Designer, Showing `Toolbar`

Use the grab handles on the start, top, and end sides and connect them to the start, top, and end sides of the `ConstraintLayout` that is the root of our layout:

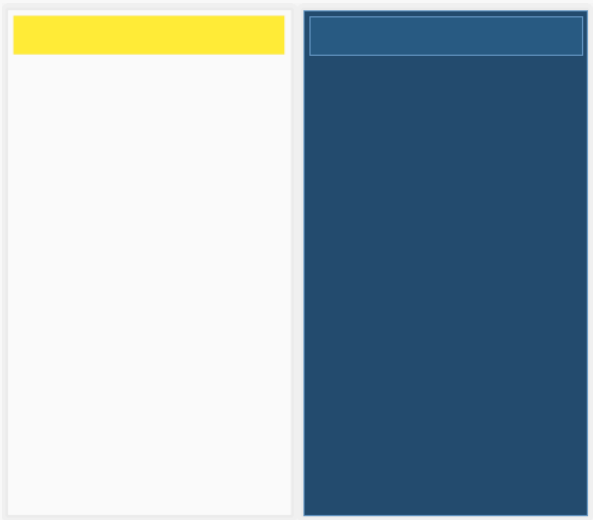


Figure 91: Android Studio Layout Designer, Showing Anchored `Toolbar`

Then, in the "Attributes" pane, set the `layout_width` to be `match_constraint` (a.k.a., `0dp`), and set the start, top, and end margins to `0dp`:

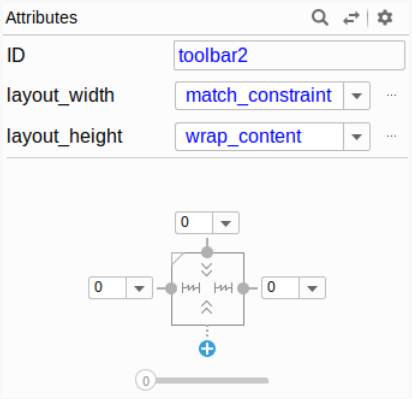


Figure 92: Android Studio Layout Designer, Showing `Toolbar` Attributes

Next, in the "Palette", choose the "Widgets" category, and drag a `WebView` into the preview area:

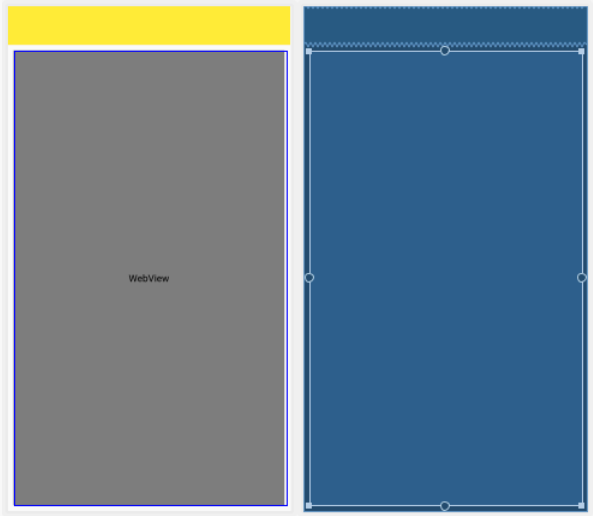


Figure 93: Android Studio Layout Designer, Showing `Toolbar` and `WebView`

However, while the `WebView` might *seem* like it is set to fill all of the available space, the design tool probably just assigned it some hard-coded values, ones that make it difficult to work with. So, in the "Attributes" pane, temporarily assign `wrap_content` to both "layout_width" and "layout_height", to give you a smaller `WebView` to work with:

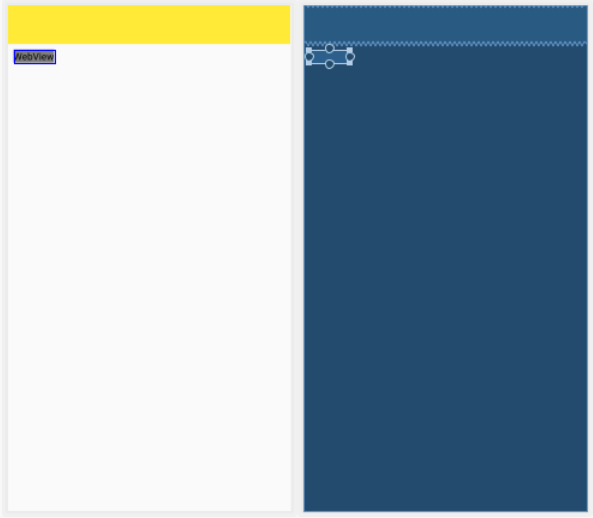


Figure 94: Android Studio Layout Designer, Showing Smaller `WebView`

Then, drag the grab handles from the start, bottom, and end of the `WebView` and attach them to the corresponding sides of the `ConstraintLayout`. Also, drag the grab handle from the top of the `WebView` and connect it to the bottom of the `Toolbar`:

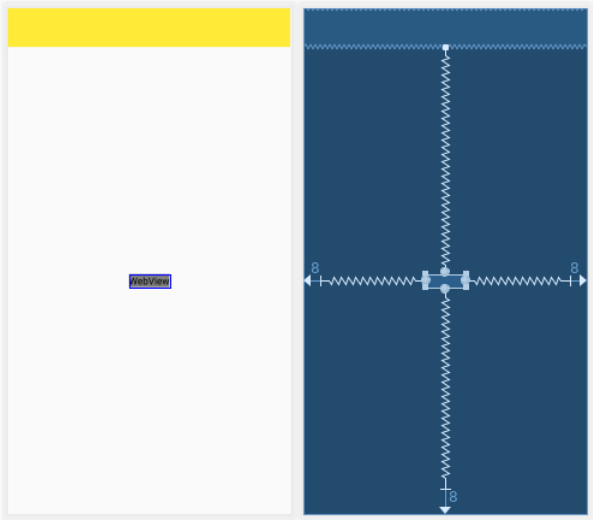


Figure 95: Android Studio Layout Designer, Showing Constrained `WebView`

Then, go back to the "Attributes" pane and set the "layout_width" and "layout_height" each to `match_constraint` (a.k.a., `0dp`), to have the `WebView` fill all of the available space, and remove the `8dp` of margin from all four sides:

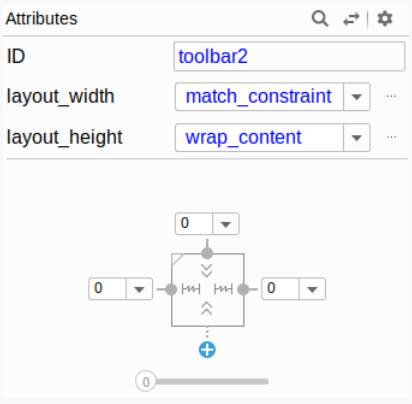


Figure 96: Android Studio Layout Designer, Showing `WebView` Attributes

Also, back in the "Attributes" pane, give the `WebView` an "ID" of `about` and the `Toolbar` an "ID" of `toolbar`.

At this point, the layout XML should resemble:

```
<?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=".AboutActivity">

  <androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    android:minHeight="?attr/actionBarSize"
    android:theme="?attr/actionBarTheme"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

  <WebView
    android:id="@+id/about"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/toolbar" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from To8-Activities/ToDo/app/src/main/res/layout/activity_about.xml)

Step #3: Launching Our Activity

Now that we have declared that the activity exists and can be used, we can start using it.

Go into `MainActivity` and modify `onCreate()` to start `AboutActivity` if the user chooses the `about` menu item:

```
package com.commonsware.todo

import android.content.Intent
import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_main.*
```

```
class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    toolbar.title = getString(R.string.app_name)
    toolbar.inflateMenu(R.menu.actions)

    toolbar.setOnMenuItemClickListener { item ->
      when (item.itemId) {
        R.id.about -> startActivity(Intent(this, AboutActivity::class.java))
        else -> return@setOnMenuItemClickListener false
      }

      true
    }
  }
}
```

(from To8-Activities/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt)

We call `setOnMenuClickListener()` on the `Toolbar` to find out when the user clicks on a menu item. Right now, we only have one menu item, but we will add more later, so we use `when` to perform different actions based on the menu item.

For the `R.id.about` menu item, we create an `Intent`, pointing at our new `AboutActivity`. Then, we call `startActivity()` on that `Intent`. You will need to add an import for `android.content.Intent` to get this to compile.

The lambda expression that we use for `setOnMenuItemClickListener()` will be converte into the `onMenuItemClick()` method of a `Toolbar.OnMenuItemClickListener` by the compiler. That Java method is supposed to return a `boolean` value, `true` meaning that we handled the event, `false` otherwise. So, we:

- Have `true` as the last value of the lambda expression, for the normal case
- In the `else` of the `when`, we explicitly return `false` from the lambda expression

If you run this app in a device or emulator, and you choose the About overflow item, the `AboutActivity` should appear, but empty, as we have not given the `Toolbar` or `WebView` any content yet.

Step #4: Defining Some About Text

We need some HTML to put into the `WebView`. We could load some from the Internet. However, then the user can only view the about text when they are online, which seems like a silly requirement. Instead, we can package some HTML as an asset inside of our app, then display that HTML in the `WebView`.

To that end, right-click over the `main` source set directory and choose "New" > "Directory" from the context menu. That will pop up a dialog, asking for the name of the directory to create:

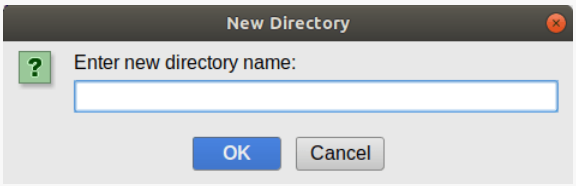


Figure 97: Android Studio New Directory Dialog

Fill in `assets` and click "OK" to create this directory.

Then, right-click over your new `assets/` directory and choose "New" > "File" from the context menu. Once again, you will get a dialog, this time to provide the filename. Fill in `about.html` and click "OK" to create this file. It should also open up an editor tab on that file, which will be empty.

There, fill in some HTML. For example, you could use:

```
<h1>About This App</h1>

<p>This app is cool!</p>

<p>No, really &mdash; this app is awesome!</p>

<div>
    .
    <br/>
    .
    <br/>
    .
    <br/>
    .
</div>
```

```
<p>OK, this app isn't all that much. But, hey, it's mine!</p>
```

(from To8-Activities/ToDo/app/src/main/assets/about.html)

Step #5: Populating the `Toolbar` and `WebView`

Open up `AboutActivity` into the editor, and change it to:

```
package com.commonsware.todo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import kotlinx.android.synthetic.main.activity_about.*

class AboutActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_about)

    toolbar.title = getString(R.string.app_name)
    about.loadUrl("file:///android_asset/about.html")
  }
}
```

(from To8-Activities/ToDo/app/src/main/java/com/commonsware/todo/AboutActivity.kt)

Here, we retrieve the `about` `WebView` from our inflated layout, then call `loadUrl()` on it to tell it what to display. `loadUrl()` normally takes an `https` URL, but in this case, we use the special `file:///android_asset/` notation to indicate that we want to load an asset out of `assets/`. `file:///android_asset/` points to the root of `assets/`, so `file:///android_asset/about.html` points to `assets/about.html`.

(yes, `file:///android_asset/` is singular, and `assets/` is plural – eventually, you just get used to this...)

We also set the title of the `Toolbar`, much as we did in `MainActivity`.

If you now run the app, and choose “About” from the overflow, you will see your about text:

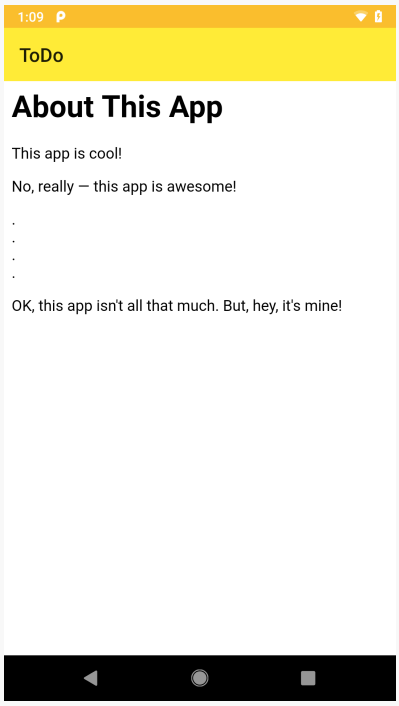


Figure 98: ToDo About Activity

What We Changed

The book’s GitHub repository contains the entire result of having completed this tutorial. In particular, it contains the changed files:

- app/src/main/AndroidManifest.xml
- app/src/main/res/layout/activity_about.xml
- app/src/main/assets/about.html
- app/src/main/java/com/commonsware/todo/MainActivity.kt
- app/src/main/java/com/commonsware/todo/AboutActivity.kt

Integrating Fragments

As we saw at the outset, there will be three main elements of the user interface when we are done:

- a list of to-do items
- a place to edit an item, whether that is a new one being added to the list or modifying an existing one
- a place to view details of a single item

We could implement all of those as activities, if we wanted to. However, that will make it difficult to implement a good UI on a tablet-sized device. Each one of those three elements is much too small to be worth taking up an entire 10" tablet screen, for example. So, while we will show one of those elements at a time on smaller screens, on larger screens we could show two at a time:

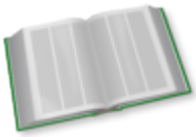
- the list plus the details, or
- the list plus the editor

We cannot do this with three independent activities. This is where fragments come into play. We can define each of the three elements as a fragment, then arrange to show either one or two fragments at a time, based upon screen size.

In this chapter, we will convert our existing list into a fragment and have our activity display that fragment. This will have no immediate impact upon the user experience — the app will be unchanged visibly as a result of these changes. But, we will be setting ourselves up for creating the other two elements — a details fragment and an edit fragment — in later tutorials.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of

completing the work in this tutorial.



You can learn more about fragments in the "Adopting Fragments" chapter of *Elements of Android Jetpack*!

Step #1: Creating a Fragment

First, we need to set up a fragment. While Android Studio offers a new-fragment wizard, its results are poor, so we will create one as a normal Java class.

Right-click over the `com.commonsware.todo` package in the `java/` directory and choose "New" > "Kotlin File/Class" from the context menu. This will bring up a dialog where we can define a new Kotlin class:

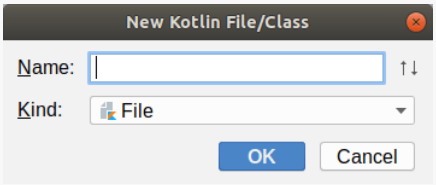


Figure 99: Android Studio Create Kotlin Class Dialog

For the name, fill in `RosterListFragment`, as this fragment is showing a list of our roster of to-do items. Choose "Class" in the "Kind" drop-down. Then, click OK to create the class. That will give you a `RosterListFragment` that looks like:

```
package com.commonsware.todo

class RosterListFragment {
}
```

Modify it to extend `androidx.fragment.app.Fragment`:

```
package com.commonsware.todo

import androidx.fragment.app.Fragment

class RosterListFragment : Fragment() {
}
```

However, this fragment does not do anything, and we need it to display our user interface. So, with your cursor inside the `{ }` of the class, press Ctrl-O (or Command-O on macOS), to bring up a list of methods that we could override:

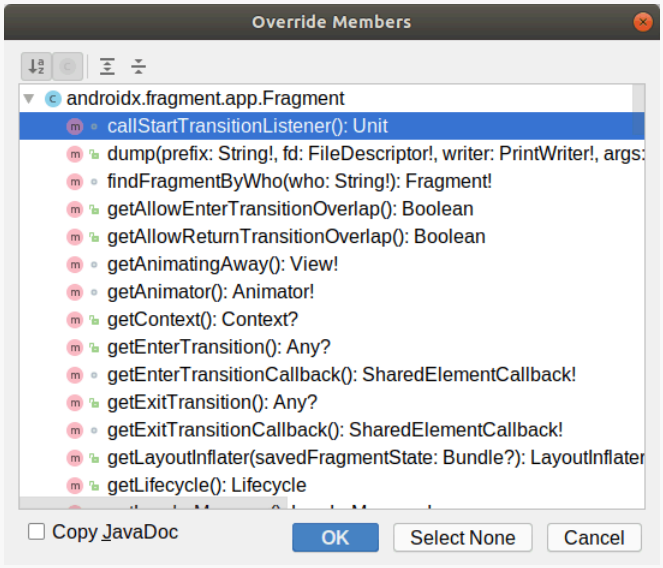


Figure 100: Android Studio Method Override Dialog

If you start typing with that dialog on the screen, what you type in works as a search mechanism, jumping you to the first method that resembles what you typed in. So, start typing in `onCreateView`, until that becomes the selected method:

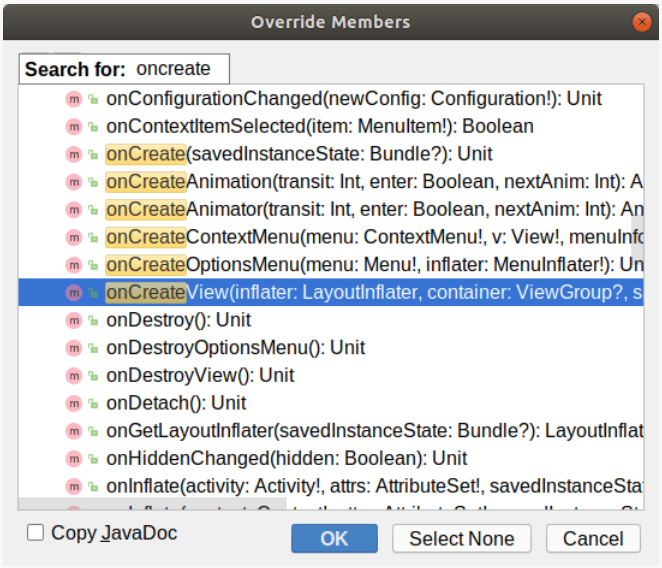


Figure 101: Android Studio Method Override Dialog, During Search

Then, click OK to add a stub implementation of that method to your `RosterListFragment`:

```
package com.commonsware.todo

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment

class RosterListFragment : Fragment() {
  override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View? {
    return super.onCreateView(inflater, container, savedInstanceState)
  }
}
```

The `override` keyword means that we are overriding an existing function that we are

inheriting from `Fragment`.

The job of `onCreateView()` of a fragment is to set up the UI for that fragment. In `MainActivity`, right now, we are doing that by calling `setContentView(R.layout.activity_main)`. We want to use that layout file here instead. To do that, modify `onCreateView()` to look like:

```
package com.commonsware.todo

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment

class RosterListFragment : Fragment() {
  override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View? = inflater.inflate(R.layout.activity_main, container, false)
}
```

Here, we use the supplied `LayoutInflater`. To "inflate" in Android means "convert an XML resource into a corresponding tree of Java objects". `LayoutInflater` inflates layout resources, via its family of `inflate()` methods. We are specifically saying:

- Inflate `R.layout.activity_main`
- Its widgets will eventually go into the `container` supplied to `onCreateView()`
- Do *not* put those widgets in that container right now, as the fragment system will handle that for us at an appropriate time

In practice, you could skip the return type of the function. However, Android Studio will complain about that, as Kotlin cannot tell whether `onCreateView()` is allowed to return `null` or not. So, to eliminate the Android Studio warning, we have `onCreateView()` return `View?` specifically.

Step #2: Updating the Toolbar

Our layout resource contains our `Toolbar`. Previously, we had our activity fill in the `Toolbar`, but now that needs to be managed by the fragment.

So, with your cursor inside the body of `RosterListFragment`, press Ctrl-O (or Command-O on macOS), to once again bring up a list of methods that we could override. Search for `onViewCreated()` and add it:

```
package com.commonsware.todo

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment

class RosterListFragment : Fragment() {
  override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View? = inflater.inflate(R.layout.activity_main, container, false)

  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)
  }
}
```

Add `android.content.Intent` and `kotlinx.android.synthetic.main.activity_main.*` to your list of imports.

Then, modify `onViewCreated()` to be:

```
  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    toolbar.title = getString(R.string.app_name)
    toolbar.inflateMenu(R.menu.actions)

    toolbar.setOnMenuItemClickListener { item ->
      when (item.itemId) {
        R.id.about -> startActivity(Intent(activity, AboutActivity::class.java))
        else -> return@setOnMenuItemClickListener false
      }

      true
    }
  }
```

(from Too-Fragments/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt)

This just adds the same `Toolbar` code that we were using in the activity. `onViewCreated()` is the traditional fragment lifecycle method for configuring our

inflated UI.

At this point, `RosterListFragment` should look like:

```
package com.commonsware.todo

import android.content.Intent
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import kotlinx.android.synthetic.main.activity_main.*

class RosterListFragment : Fragment() {
  override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View? = inflater.inflate(R.layout.activity_main, container, false)

  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    toolbar.title = getString(R.string.app_name)
    toolbar.inflateMenu(R.menu.actions)

    toolbar.setOnMenuItemClickListener { item ->
      when (item.itemId) {
        R.id.about -> startActivity(Intent(activity, AboutActivity::class.kt))
        else -> return@setOnMenuItemClickListener false
      }

      true
    }
  }
}
```

Step #3: Add the KTX Dependency

There are some Kotlin utility extension functions available for fragments that we could use. So, back in `app/build.gradle`, add in a dependency on `androidx.fragment:fragment-ktx` to our list of dependencies:

```
dependencies {
```

```
  implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
  implementation 'androidx.appcompat:appcompat:1.0.2'
  implementation 'androidx.core:core-ktx:1.0.1'
  implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
  implementation 'androidx.recyclerview:recyclerview:1.0.0'
  implementation 'androidx.fragment:fragment-ktx:1.0.0'
  testImplementation 'junit:junit:4.12'
  androidTestImplementation 'androidx.test:runner:1.1.1'
  androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
}
```

(from To9-Fragments/ToDo/app/build.gradle)

When the yellow banner appears about your Gradle file changes, click the "Sync Now" link.

Step #4: Displaying the Fragment

Just because we have a fragment class does not mean that it will be displayed anywhere. We have to arrange to have that happen.

There are two ways to show a fragment on the screen:

- Use a `<fragment>` element in a layout resource ("static fragments")
- Use a `FragmentTransaction` to show one from our Kotlin code ("dynamic fragments")

We will use a static fragment later in the book, so here, let's use a dynamic fragment.

To that end, modify `MainActivity` to look like this:

```
package com.commonsware.todo

import android.os.Bundle
import androidx.appcompat.app.AppCompatActivity
import androidx.fragment.app.transaction

class MainActivity : AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    if (supportFragmentManager.findFragmentById(android.R.id.content) == null) {
      supportFragmentManager.transaction {
        add(android.R.id.content, RosterListFragment())
      }
    }
  }
```

```
}
```

(from To9-Fragments/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt)

This removes most of our previous logic, including the `setContentView()` call and the `Toolbar` code. All of that we moved into `RosterListFragment`.

Instead, we are using a `FragmentManager`, obtained by referencing `supportFragmentManager`. The name `supportFragmentManager` is because we are using a library-supplied implementation of fragments — `fragmentManager` is for the deprecated framework implementation of fragments.

We are asking the `FragmentManager` to give us the fragment located in a container identified as `android.R.id.content`, by calling `findFragmentById()` and passing in that container ID. This container is provided to every activity by the framework. When we called `setContentView()` before, we were telling the activity to take the contents of that layout resource and put it into this container. However, we can also use the container with fragments.

If the `FragmentManager` says that we do not have such a fragment (`findFragmentById()` returns `null`), then we call `transaction()` on the `FragmentManager`. This extension function comes from `androidx.fragment:fragments-ktx`. It:

- Opens a `FragmentTransaction`
- Executes our supplied lambda expression, setting it up such that `this` points to the `FragmentTransaction` (akin to how the Kotlin `apply()` scope function works)
- Commits the `FragmentTransaction`

Inside of our transaction, we call `add()`. As the name suggests, this adds a fragment to our activity. Specifically, we are adding an instance of `RosterListFragment` that we are creating here, putting it into the `android.R.id.content` container.

So now we have migrated our minimal UI from being managed by the activity to being managed by a fragment, which in turn is managed by the activity.

The point behind the find-then-add approach is that when Android performs a configuration change, it will destroy and recreate our activity and fragments. `onCreate()` is called both when the activity is initially created and when a new instance is created after a configuration change. In the configuration change scenario, we already have a `RosterListFragment` and do not need two. So, to handle

that, we check for the fragment first, and then add the fragment if it does not already exist.

Step #5: Renaming Our Layout Resource

However, our layout resource now has a silly name. It is called `activity_main`, and it is not being displayed (directly) by an activity. Moreover, eventually, our `MainActivity` will be using three fragments, each with its own layout resource.

So, let's rename this layout to `todo_roster` instead.

To do that, right-click over `res/layout/activity_main.xml` in the project tree, then choose "Refactor" > "Rename" from the context menu. This will bring up a dialog for you to provide the replacement name:

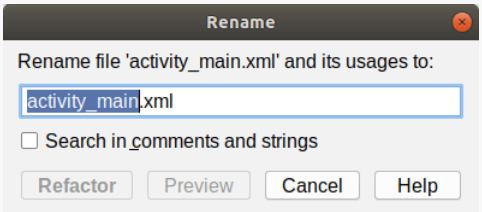


Figure 102: Android Studio Rename Dialog

Change that to be `todo_roster.xml`, then click "Refactor". This may display a "Refactoring Preview" view towards the bottom of the IDE:

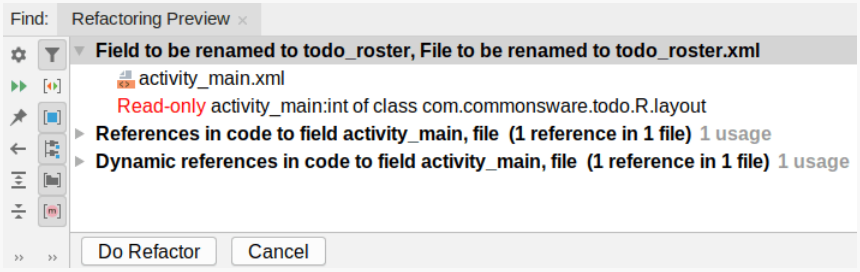


Figure 103: Android Studio Refactoring Preview view

This will not appear for everything that you rename, but it will show up from time to time, particularly when Android Studio wants confirmation that you really want to rename all of these things.

Click the "Do Refactor" button towards the bottom of the "Refactoring Preview"

view. Not only will this change the name of the file, but if you look at `RosterListFragment`, you will see that Android Studio *also* fixed up our `inflate()` call to use the new resource name *and* our `kotlinx` import as well:

```
package com.commonsware.todo

import android.content.Intent
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import kotlinx.android.synthetic.main.todo_roster.*

class RosterListFragment : Fragment() {
  override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
  ): View? = inflater.inflate(R.layout.todo_roster, container, false)

  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    toolbar.title = getString(R.string.app_name)
    toolbar.inflateMenu(R.menu.actions)

    toolbar.setOnMenuItemClickListener { item ->
      when (item.itemId) {
        R.id.about -> startActivity(Intent(activity, AboutActivity::class.java))
        else -> return@setOnMenuItemClickListener false
      }

      true
    }
  }
}
```

(from To9-Fragments/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt)

If you run the app, everything looks as it did before, even though now our UI is managed by the fragment.

What We Changed

The book's GitHub repository contains the entire result of having completed this tutorial. In particular, it contains the changed files:

- app/build.gradle
- app/src/main/res/layout/todo_roster.xml
- app/src/main/java/com/commonsware/todo/MainActivity.kt

- app/src/main/java/com/commonsware/todo/AboutActivity.kt
- app/src/main/java/com/commonsware/todo/RosterListFragment.kt

Defining a Model

If we are going to show to-do items in this list, it would help to have some to-do items. That, in turn, means that we need a Java class that represents a to-do item. Such a class is often referred to as a "model" class, so in this chapter, we will create a `ToDoModel`, where each `ToDoModel` instance represents one to-do item.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.

Step #1: Adding a Stub POJO

First, let's create the base `ToDoModel` class. To do this, right-click over the `com.commonsware.todo` package in the project tree in Android Studio, and choose "New" > "Kotlin File/Class" from the context menu. As before, this brings up a dialog where we can define a new Kotlin class, by default into the same Java package that we right-clicked over. Fill in `ToDoModel` in the "Name" field and choose "Class" in the "Kind" drop-down list. Then click "OK" to create this class. `ToDoModel` should show up in an editor, with an implementation like this:

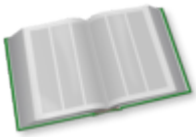
```
package com.commonsware.todo

class ToDoModel {
}
```

Step #2: Switching to a `data` Class

A typical pattern for model objects in Kotlin is for them to be `data` classes. This makes them immutable: you do not change a model, but instead replace it with a

new instance that has the new values.



You can learn more about `data` classes in the "Data Class" chapter of *Elements of Kotlin*!

So, add the `data` keyword before `class`, giving you:

```
package com.commonsware.todo

data class ToDoModel {
}
```

This will immediately show a red undersquiggle, indicating that Android Studio is unhappy about something:

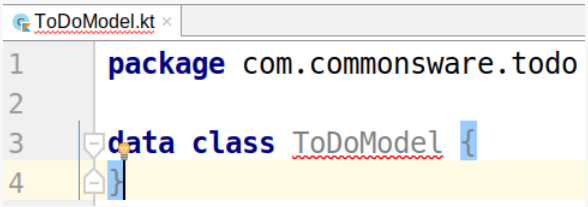


Figure 104: Android Studio, Yelling

That is because a `data` class must have a constructor with 1+ parameters. We will add that constructor in the next section.

Step #3: Adding the Constructor

Let's add 5 properties to `ToDoModel`, as constructor `val` parameters:

- A unique ID
- A flag to indicate if the task is completed or not
- A description, which will appear in the list
- Some notes, in case there is more information
- The date/time that the model was created on

To that end, modify `ToDoModel` to look like:

```
package com.commonsware.todo
```

```
import java.util.*

data class ToDoModel(
  val description: String,
  val id: String = UUID.randomUUID().toString(),
  val isCompleted: Boolean = false,
  val notes: String = "",
  val createdOn: Calendar = Calendar.getInstance()
) {
}
```

(from T10-Model/ToDo/app/src/main/java/com/commonsware/todo/ToDoModel.kt)

Here, we have added the five constructor parameters. Four of them — all but `description` – provide default values, so we can supply values or not as we see fit when we create instances.

Of particular note:

- We use `UUID` to generate a unique identifier for our to-do item, held in the `id` property
- We use `Calendar` for tracking the created-on time for this to-do item, held in the `createdOn` property

What We Changed

The book's GitHub repository contains the entire result of having completed this tutorial. In particular, it contains the changed files:

- `app/src/main/java/com/commonsware/todo/ToDoModel.kt`

Setting Up a Repository

So, now we have a `ToDoModel`. Wonderful!

But, this raises the question: where do `ToDoModel` instances come from?

In the long term, we will be storing our to-do items in a database. For the moment, to get our UI going, we can just cache them in memory. We could, if desired, have a server somewhere that is the "system of record" for our to-do items, with the local database serving as a persistent cache.

Ideally, our UI code does not have to care about any of that. And, ideally, our code that does have to deal with all of the storage work does not care about how our UI is written.

One pattern for enforcing that sort of separation is to use a repository. The repository handles all of the data storage and retrieval work. Exactly *how* it does that is up to the repository itself. It offers a fairly generic API that does not "get into the weeds" of the particular storage techniques that it uses. The UI layer works with the repository to get data, create new data, update or delete existing data, and so on, and the repository does the actual work.

A repository is usually a singleton. Later we will set up that singleton using a technique called "dependency injection". For now, though, we will use a simple Kotlin `object`.

You can learn more about Kotlin and `object` in the "The object Keyword" chapter of *Elements of Kotlin*!

So, in this tutorial, we will set up a simple repository. Right now, that will just be an in-memory cache, but in later tutorials we will move that data to a database.

This is a continuation of the work we did in the previous tutorial. The book's GitHub repository contains the results of the previous tutorial as well as the results of completing the work in this tutorial.

NOTE: Starting with this tutorial, we will stop reminding you about adding `import` statements for newly-referenced classes. By now, you should be used to the pattern of adding `import` statements. If you see a class name show up in red, and Android Studio says that it does not know about that class, most likely you need to add an `import` statement for it.

Step #1: Adding the Object

Once again, we need some more Kotlin code. This time, at least for now, it will be a Kotlin `object`, rather than a class.

Right-click over the `com.commonsware.todo` package in the project tree in Android Studio, and choose "New" > "Kotlin File/Class" from the context menu. As before, this brings up a dialog where we can define a new Kotlin source file, by default into the same Java package that we right-clicked over. Fill in `ToDoRepository` in the "Name" field, and choose "Object" from the "Kind" drop-down. Then click "OK" to create this file. `ToDoRepository` should show up in an editor, with an implementation like this:

```
package com.commonsware.todo

object ToDoRepository {
}
```

Step #2: Creating Some Fake Data

At the moment, our repository has no data. We need to fix this, so that we have some to-do items to show in our UI. But, right now, our UI can only show to-do items; we have not built any forms to allow the user to create new to-do items. So, for the time being, we can have our repository create some fake data, which we can then replace with user-supplied data later on.

To that end, replace the stub `ToDoRepository` that Android Studio gave us with:

```
package com.commonsware.todo

object ToDoRepository {
  val items = listOf(
    ToDoModel(
      description = "Buy a copy of _Exploring Android_",
      isCompleted = true,
      notes = "See https://wares.commonsware.com"
    ),
    ToDoModel(
      description = "Complete all of the tutorials"
    ),
    ToDoModel(
      description = "Write an app for somebody in my community",
      notes = "Talk to some people at non-profit organizations to see what they need!"
    )
  )
}
```

(from T11-Repository/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt)

This just adds an `items` property that is a simple immutable list of three `ToDoModel` objects. We provide a `description` for all three models, but we use the default constructor options for some of the other properties.

Later, this is going to need to get a *lot* more complicated:

- We will need to get our data from a database
- We will need to update the database with new, changed, or deleted models
- All of that is slow, so we will need to do that work on a background thread

But, for the moment, this will suffice. In an upcoming tutorial, we will have our `RosterListFragment` get its data from this `ToDoRepository` singleton.

What We Changed

The book's GitHub repository contains the entire result of having completed this tutorial. In particular, it contains the changed files:

- `app/src/main/java/com/commonsware/todo/ToDoRepository.kt`