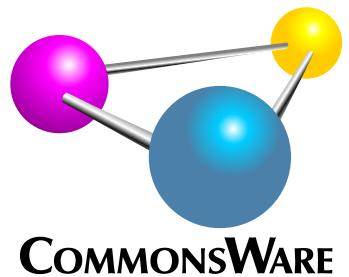


FINAL Version

Exploring Android



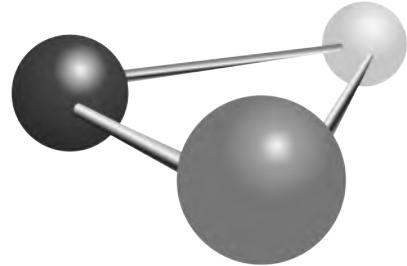
Mark L. Murphy



COMMONSWARE

Exploring Android

by Mark L. Murphy



COMMONSWARE

Exploring Android
by Mark L. Murphy

Copyright © 2017-2021 CommonsWare, LLC. All Rights Reserved.
Printed in the United States of America.

Printing History:
December 2021: FINAL Version

The CommonsWare name and logo, "Busy Coder's Guide", and related trade dress are trademarks of CommonsWare, LLC.

All other trademarks referenced in this book are trademarks of their respective firms.

The publisher and author(s) assume no responsibility for errors or omissions or for damages resulting from the use of the information contained herein.

Table of Contents

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

• <u>Preface</u>	◦ How the Book Is Structured	ix
	◦ Second-Generation Book	x
	◦ Prerequisites	x
	◦ Copying Code From This Book	x
	◦ Source Code and Its License	xi
• <u>What We Are Building</u>	◦ The Purpose	1
	◦ The Core UI	1
• <u>Installing the Tools</u>	◦ Step #1: Checking Your Hardware	9
	◦ Step #2: Install Android Studio	10
	◦ Step #3: Run Android Studio	11
• <u>Creating a Starter Project</u>	◦ Step #1: Importing the Project	19
	◦ Step #2: Setting Up the Emulator AVD	23
	◦ Step #3: Setting Up the Device	29
	◦ Step #4: Running the Project	32
• <u>Modifying the Manifest</u>	◦ Some Notes About Relative Paths	35
	◦ Step #1: Supporting Screens	36
	◦ Step #2: Blocking Backups	37
	◦ Final Results	37
	◦ <i>What We Changed</i>	38
• <u>Changing Our Icon</u>	◦ Step #1: Getting the Replacement Artwork	39
	◦ Step #2: Changing the Icon	40
	◦ Step #3: Running the Result	50
	◦ What We Changed	50
• <u>Adding a Library</u>	◦ Step #1: Examining What We Have	51
	◦ Step #2: Adding Support for RecyclerView	52
	◦ Final Results	52
	◦ <i>What We Changed</i>	54
• <u>Constructing a Layout</u>		

◦ Step #1: Examining What We Have And What We Want	55
◦ Step #2: Adding a RecyclerView	58
◦ Step #3: Adjusting the TextView	66
◦ Final Results	72
◦ <i>What We Changed</i>	73
• <u>Integrating Fragments</u>	
◦ But First, Some Notes About Working with Kotlin	75
◦ Step #1: Creating a Fragment	77
◦ Step #2: Renaming Our Layout	79
◦ <i>Step #3: Inflating Our Layout</i>	81
◦ Step #4: Dealing with Crashes	84
◦ Final Results	86
◦ <i>What We Changed</i>	86
• <u>Wiring In Navigation</u>	
◦ Step #1: Defining the Version	87
◦ Step #2: Adding the Plugin Dependency	88
◦ <i>Step #3: Requesting the Plugins</i>	89
◦ <i>Step #4: Augmenting Our Dependencies</i>	90
◦ Step #5: Defining Our Navigation Graph	90
◦ Step #6: Setting Up a New Activity Layout Resource	93
◦ Step #7: Wiring in the Navigation	95
◦ Final Results	96
◦ <i>What We Changed</i>	99
• <u>Setting Up the App Bar</u>	
◦ Step #1: Defining Some Colors	102
◦ Step #2: Adjusting Our Theme	105
◦ Step #3: Adding a Toolbar	108
◦ Step #4: Adding an Icon	112
◦ Step #5: Defining an Item	114
◦ Step #6: Enabling View Binding	124
◦ Step #7: Using View Binding in Our Activity	125
◦ Step #8: Loading Our Options	126
◦ Step #9: Trying It Out	127
◦ Final Results	128
◦ <i>What We Changed</i>	131
• <u>Setting Up an Activity</u>	
◦ Step #1: Creating the Stub Activity Class and Manifest Entry	133
◦ <i>Step #2: Adding a Toolbar and a WebView</i>	136
◦ Step #3: Launching Our Activity	141
◦ Step #4: Defining Some About Text	142
◦ Step #5: Populating the Toolbar and WebView	144

◦ Final Results	145
◦ <i>What We Changed</i>	147
• <u>Defining a Model</u>	
◦ Step #1: Adding a Stub POJO	149
◦ Step #2: Switching to a data Class	149
◦ Step #3: Adding the Constructor	150
◦ Step #4: Supporting Instant on Older Devices	151
◦ Final Results	152
◦ <i>What We Changed</i>	154
• <u>Setting Up a Repository</u>	
◦ Step #1: Adding the Repository Class	156
◦ Step #2: Creating Some Fake Data	156
◦ Final Results	157
◦ <i>What We Changed</i>	157
• <u>Inverting Our Dependencies</u>	
◦ Step #1: Adding the Dependencies	160
◦ Step #2: Creating a Custom Application	161
◦ Step #3: Defining Our Module	162
◦ Final Results	164
◦ <i>What We Changed</i>	167
• <u>Incorporating a ViewModel</u>	
◦ Step #1: Creating a Stub ViewModel	170
◦ Step #2: Getting and Using Our Repository	170
◦ Step #3: Depositing a Koin	171
◦ Step #4: Injecting the Motor	171
◦ Final Results	172
◦ <i>What We Changed</i>	175
• <u>Populating Our RecyclerView</u>	
◦ Step #1: Defining a Row Layout	177
◦ Step #2: Adding a Stub ViewHolder	184
◦ Step #3: Creating a Stub Adapter	184
◦ Step #4: Comparing Our Models	187
◦ Step #5: Completing the Adapter and ViewHolder	189
◦ Step #6: Wiring Up the RecyclerView	191
◦ Final Results	194
◦ <i>What We Changed</i>	197
• <u>Tracking the Completion Status</u>	
◦ Step #1: Registering for Events	199
◦ Step #2: Passing the Event Up the Chain	200
◦ Step #3: Saving the Change	202
◦ Final Results	203

◦ <i>What We Changed</i>	207
• <u>Displaying an Item</u>	
◦ Step #1: Creating the Fragment	209
◦ Step #2: Updating the Navigation Graph	210
◦ Step #3: Responding to List Clicks	214
◦ Step #4: Teaching Navigation About the App Bar	217
◦ Step #5: Creating an Empty Layout	219
◦ Step #6: Adding the Completed Icon	219
◦ Step #7: Displaying the Description	226
◦ Step #8: Showing the Created-On Date	228
◦ Step #9: Adding the Notes	231
◦ Step #10: Adding Navigation Arguments	234
◦ Step #11: Displaying the Layout	237
◦ Step #12: Making Another Motor	238
◦ Step #13: Populating the Layout	240
◦ Final Results	241
◦ <i>What We Changed</i>	251
• <u>Editing an Item</u>	
◦ Step #1: Creating the Fragment	254
◦ Step #2: Setting Up the Navigation	254
◦ Step #3: Setting Up a Menu Resource	256
◦ Step #4: Showing the App Bar Item	260
◦ Step #5: Displaying the (Empty) Fragment	262
◦ Step #6: Creating an Empty Layout	263
◦ Step #7: Adding the CheckBox	263
◦ Step #8: Creating the Description Field	264
◦ Step #9: Adding the Notes Field	268
◦ Step #10: Populating the Layout	270
◦ Final Results	272
◦ <i>What We Changed</i>	277
• <u>Saving an Item</u>	
◦ Step #1: Adding the App Bar Item	279
◦ Step #2: Improving the Motor	282
◦ Step #3: Replacing the Item	283
◦ Step #4: Returning to the Display Fragment	284
◦ Step #5: Getting Updated Items	286
◦ Final Results	287
◦ <i>What We Changed</i>	291
• <u>Adding and Deleting Items</u>	
◦ Step #1: Trimming Our Repository	293
◦ Step #2: Showing an Empty View	294

◦ Step #3: Adding an Add App Bar Item	296
◦ Step #4: Launching the EditFragment for Adds	299
◦ Step #5: Hiding the Empty View	305
◦ Step #6: Adding a Delete App Bar Item	306
◦ Step #7: Deleting the Item	308
◦ Step #8: Fixing the Delete-on-Add Problem	309
◦ Final Results	310
◦ <i>What We Changed</i>	317
• <u>Interlude: So, What's Wrong?</u>	
◦ Issues With What We Have	319
◦ We Can Do Better	320
• <u>Refactoring Our Code</u>	
◦ Step #1: Creating Some Packages	325
◦ Step #2: Moving Our Classes	326
◦ What We Changed	329
• <u>Getting a Room (And Some Coroutines)</u>	
◦ Step #1: Requesting More Dependencies	332
◦ Step #2: Defining an Entity	333
◦ Step #3: Crafting a DAO	334
◦ Step #4: Adding a Database	336
◦ Step #5: Creating a Transmogrifier	338
◦ Step #6: Add Our Database to Koin	340
◦ <i>Step #7: Adding a Store to the Repository</i>	341
◦ Step #8: Fixing the Repository	342
◦ Final Results	344
◦ <i>What We Changed</i>	348
• <u>Completing the Reactive Architecture</u>	
◦ Step #1: Defining a Roster View State	352
◦ Step #2: Emitting View States	353
◦ <i>Step #3: Consuming Roster View States</i>	354
◦ Step #4: Wrapping the suspend Functions	356
◦ Step #5: Updating SingleModelMotor	357
◦ Step #6: Adapting DisplayFragment	358
◦ Step #7: Adapting EditFragment	359
◦ Final Results	361
◦ <i>What We Changed</i>	371
• <u>Testing a Motor</u>	
◦ Step #1: Examine Our Existing Tests	375
◦ Step #2: Decide on Instrumented Tests vs. Unit Tests	377
◦ Step #3: Adding Some Unit Test Dependencies	378
◦ Step #4: Renaming Our Unit Test	379

◦ Step #5: Running the Stub Unit Test	381
◦ Step #6: Adding a MainDispatcherRule	384
◦ Step #7: Setting Up a Mock Repository	387
◦ Step #8: Adding a Test Function	389
◦ Step #9: Adding Another Test Function	390
◦ Final Results	391
◦ <i>What We Changed</i>	394
• <u>Testing the Repository</u>	
◦ Step #1: Renaming Our Instrumented Test	396
◦ Step #2: Adding Some Instrumented Test Dependencies	396
◦ Step #3: Supporting a Test Database	397
◦ Step #4: Testing Adds	398
◦ Step #5: Writing and Running More Tests	402
◦ Final Results	403
◦ <i>What We Changed</i>	407
• <u>Testing a UI</u>	
◦ Step #1: Adding a New Test Class	409
◦ Step #2: Initializing Our Repository	410
◦ Step #3: Testing Our List	411
◦ Final Results	413
◦ <i>What We Changed</i>	414
• <u>Tracking Our Load Status</u>	
◦ Step #1: Adjusting Our Layout	417
◦ <i>Step #2: Reporting our Loaded Status</i>	420
◦ <i>Step #3: Reacting to the Loaded Status</i>	421
◦ <i>Final Results</i>	422
◦ <i>What We Changed</i>	426
• <u>Filtering Our Items</u>	
◦ Step #1: Adding a Query	427
◦ Step #2: Defining a FileMode	428
◦ Step #3: Consuming a FileMode	428
◦ <i>Step #4: Augmenting Our Motor</i>	429
◦ Step #5: Adding a Checkable Submenu	432
◦ Step #6: Getting Control on Filter Choices	437
◦ Step #7: Fixing the Empty Text	439
◦ Step #8: Addressing the Menu Problem	443
◦ Final Results	444
◦ <i>What We Changed</i>	453
• <u>Generating a Report</u>	
◦ Step #1: Adding a Save App Bar Item	455
◦ Step #2: Making a Save	457

◦ Step #3: Adding Some Handlebars	462
◦ Step #4: Creating the Report	463
◦ Step #5: Writing Where the User Asked	465
◦ Step #6: Saving the Report	466
◦ Step #7: Viewing the Report	467
◦ Final Results	470
◦ <i>What We Changed</i>	481
• <u>Sharing the Report</u>	
◦ Step #1: Adding a Share App Bar Item	483
◦ Step #2: Adding FileProvider	485
◦ Step #3: Caching the Report	490
◦ Step #4: Sharing the Report	492
◦ Final Results	493
◦ <i>What We Changed</i>	503
• <u>Collecting a Preference</u>	
◦ Step #1: Adding a Dependency	505
◦ Step #2: Defining a Preference Screen	506
◦ Step #3: Displaying Our Preference Screen	508
◦ Step #4: Adding PrefsFragment to Our Navigation Graph	509
◦ Step #5: Navigating to Our Preference Screen	512
◦ Final Results	517
◦ <i>What We Changed</i>	523
• <u>Contacting a Web Service</u>	
◦ Step #1: Adding Some Dependencies	525
◦ Step #2: Requesting a Permission	526
◦ Step #3: Defining Our Response	527
◦ Step #4: Retrieving the Items	529
◦ Step #5: Updating the Local Items	531
◦ Step #6: Fixing the Existing Tests	534
◦ Step #7: Retrieving Our Preference	535
◦ Step #8: Offering the Download Option	536
◦ Final Results	538
◦ <i>What We Changed</i>	559
• <u>Showing a Dialog</u>	
◦ Step #1: Adding a Stub Fragment	562
◦ Step #2: Updating the Navigation Graph	562
◦ Step #3: Defining the Dialog Content	564
◦ Step #4: Emitting Errors From the Motor	565
◦ Step #5: Reacting to Errors	566
◦ Step #6: Responding to Input	567
◦ Step #7: Trying It Out	570

◦ Final Results	571
◦ <i>What We Changed</i>	582
• <u>Scheduling Work</u>	
◦ Step #1: Defining a SwitchPreference	583
◦ Step #2: Observing Preference Changes	585
◦ Step #3: Adding the Dependency	586
◦ Step #4: Creating a Stub Worker	586
◦ Step #5: Injecting Into the Worker	587
◦ Step #6: Doing the Work	588
◦ Step #7: Scheduling the Work	588
◦ Step #8: Trying It Out	591
◦ Final Results	592
◦ <i>What We Changed</i>	599

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 being “hands-on”, guiding you through the steps to build the app. It provides some 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.

Copying Code From This Book

You are welcome to copy the code as you see in the book itself, as part of working through the tutorials.

However, copying from the PDF version of the book can be troublesome, depending on your PDF viewer. Some PDF viewers do not handle the syntax highlighting used

in this book very well.

Recommended PDF viewers include:

- Adobe Reader (Windows, macOS)
- Foxit Reader (Windows, macOS, Linux)
- Google Chrome (Windows, macOS)
- Google Chromium (Linux)

Also, once we start modifying files, you will find “Final Results” sections towards the end of each chapter. Those will contain the full listings of the source files that were modified in that chapter’s tutorial. And, these listings will *not* have syntax highlighting, making them suitable copy sources for a wider range of PDF viewers.

Source Code and Its License

The source code samples shown in this book are available for download from the [book’s GitLab 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.

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.

WHAT WE ARE BUILDING

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” app bar item to add a new item:

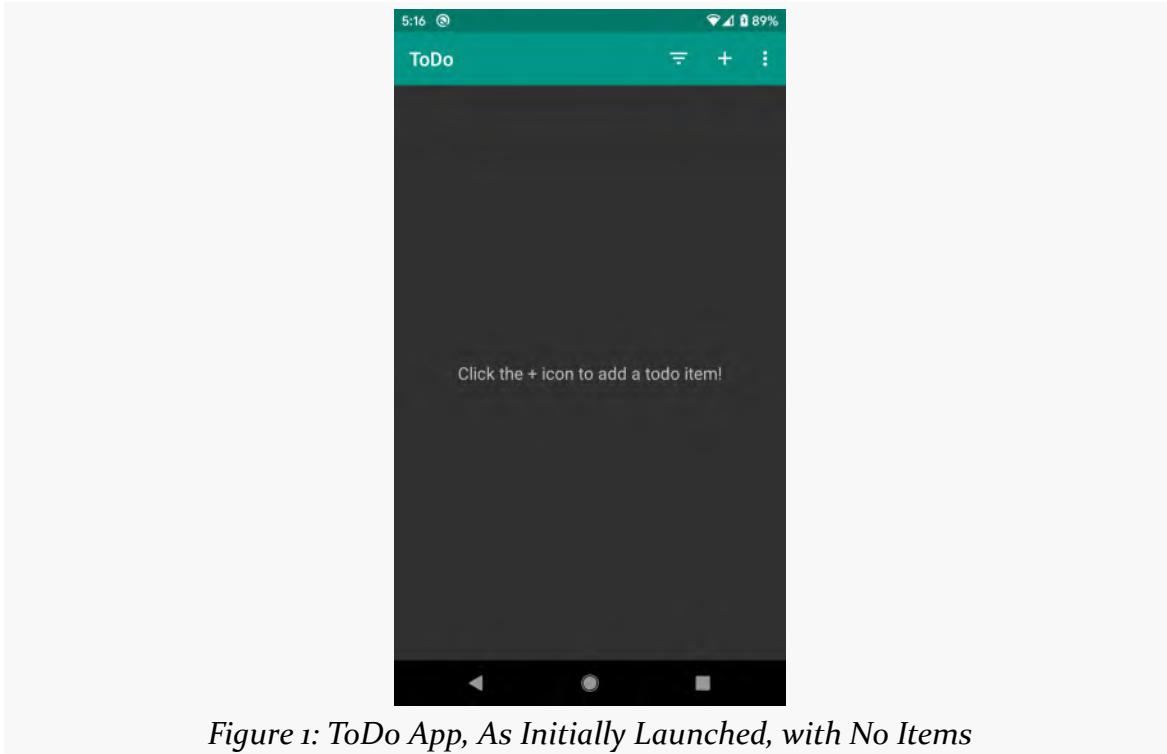


Figure 1: ToDo App, As Initially Launched, with No Items

WHAT WE ARE BUILDING

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:

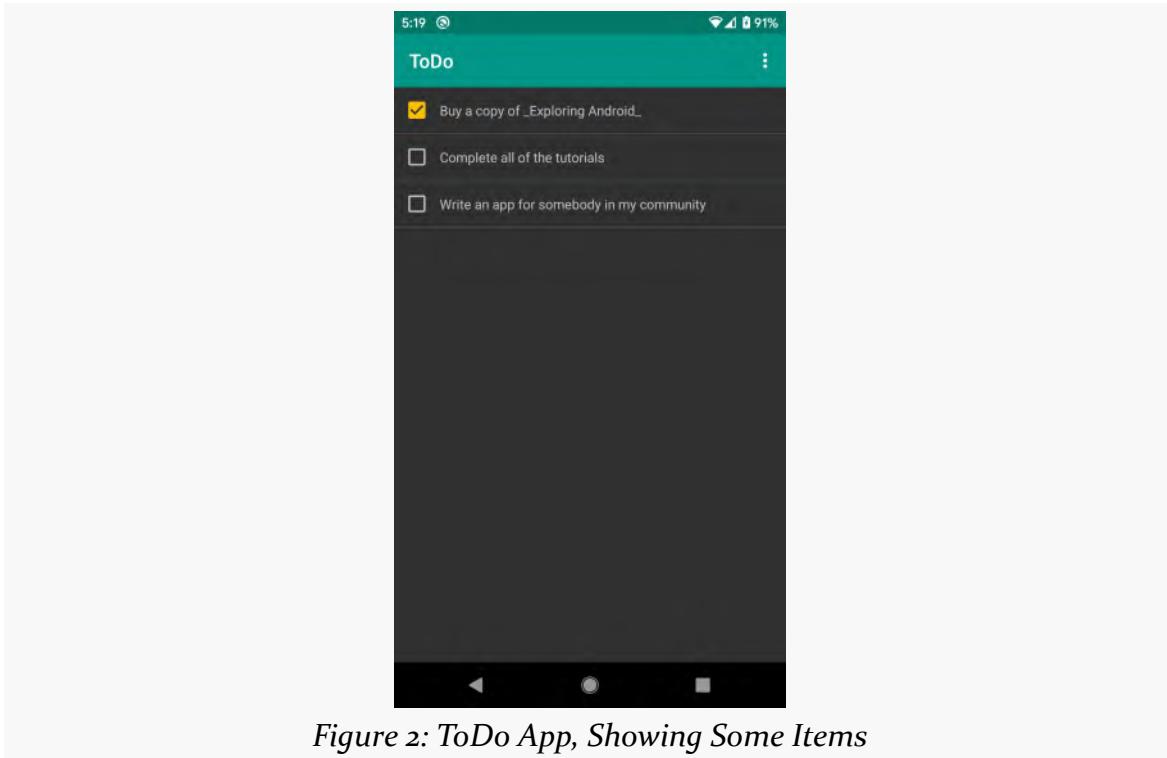


Figure 2: ToDo App, Showing Some Items

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:

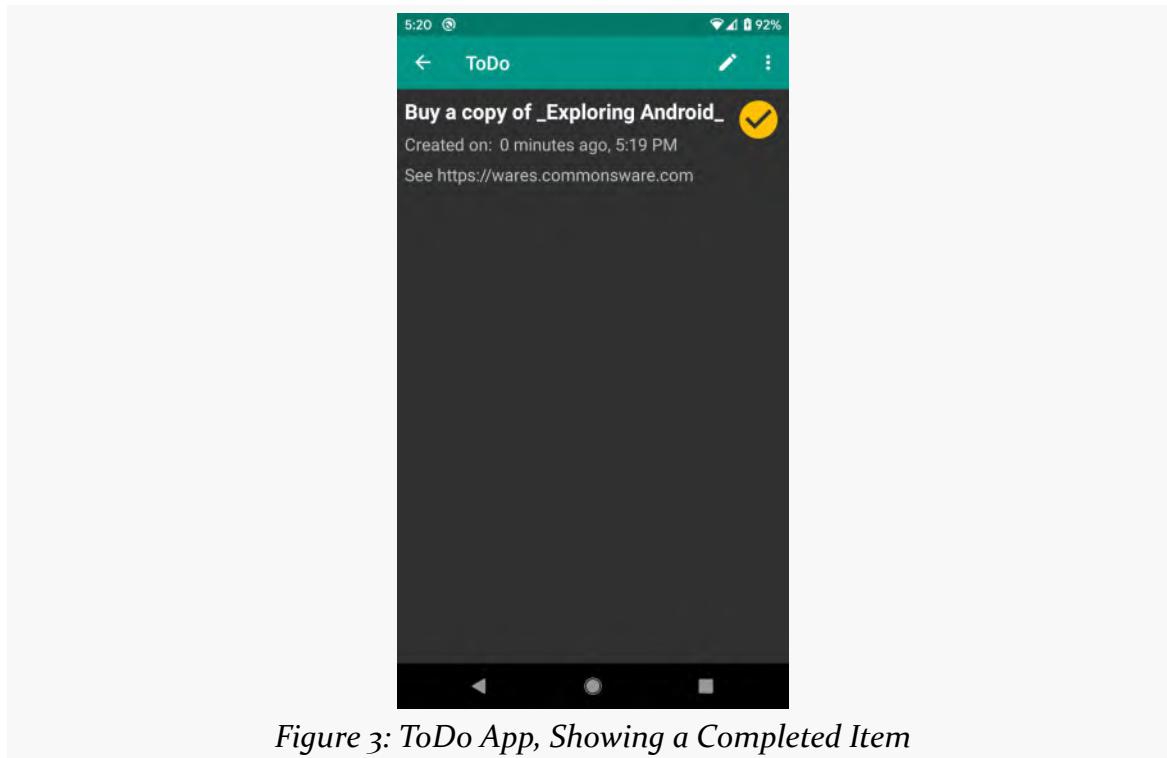


Figure 3: ToDo App, Showing a Completed Item

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” app 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) app bar item from the details screen, the editor will have the existing item’s data, which can be altered and saved:

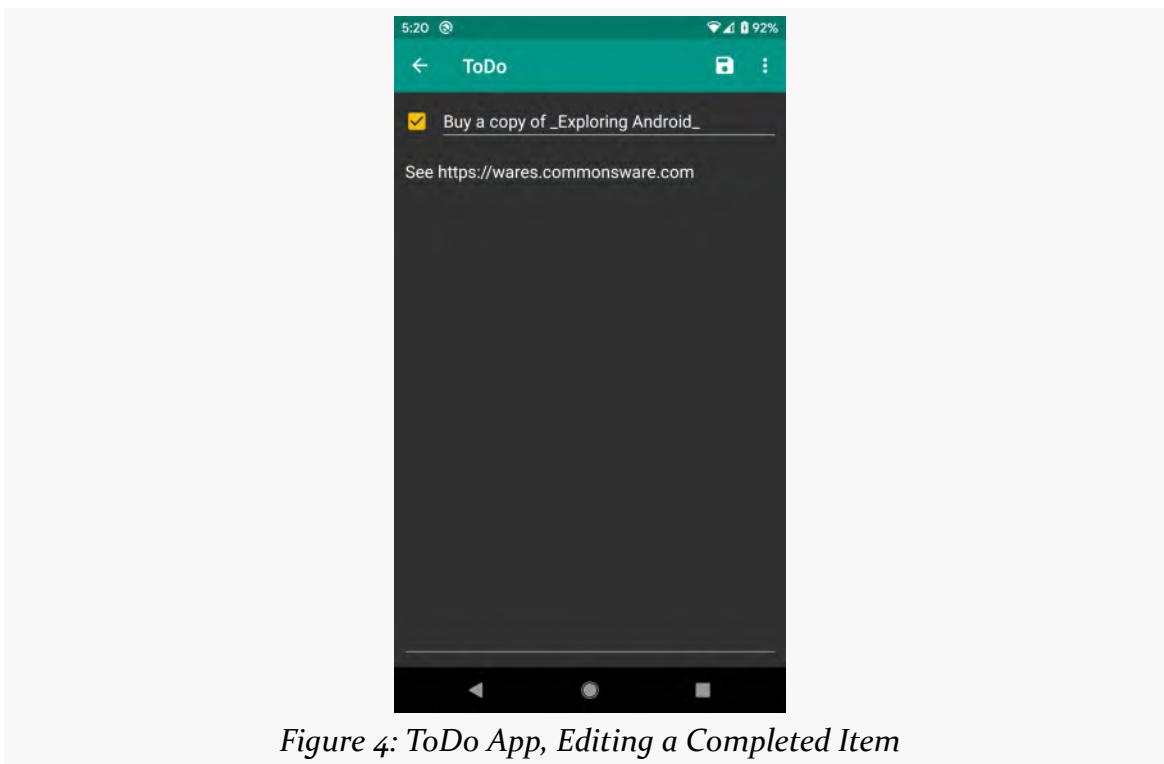


Figure 4: ToDo App, Editing a Completed Item

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 4 or more 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:

INSTALLING THE TOOLS

Development OS	CPU Manufacturer	CPU Requirements
macOS	Intel	any modern Mac should work
macOS	Apple M1	unclear
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 one or more 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.

Note that Apple M1 chip support is still a work in progress. While Android Studio Arctic Fox appears to have M1 support, it also appears that this support is a bit rough in spots. Hopefully, this will smooth out with future versions of Android Studio.

Step #2: Install Android Studio

At the time of this writing, the current production version of Android Studio is 2020.3.1 Arctic Fox and this book covers that version. Android Studio gets updated often, and so you may be on a newer version — there may be some differences between what you have and what is presented here.

INSTALLING THE TOOLS

You have two major download options. You can get the latest shipping version of Android Studio from [the Android Studio download page](#). Or, you can download Android Studio Arctic Fox directly, for:

- [Windows](#)
- [macOS x86](#)
- [macOS M1](#)
- [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 x86 users can download a DMG disk image and install it akin to other Mac software, dragging the Android Studio icon into the Applications folder. M1 users get a ZIP file instead.

Linux 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 in 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:

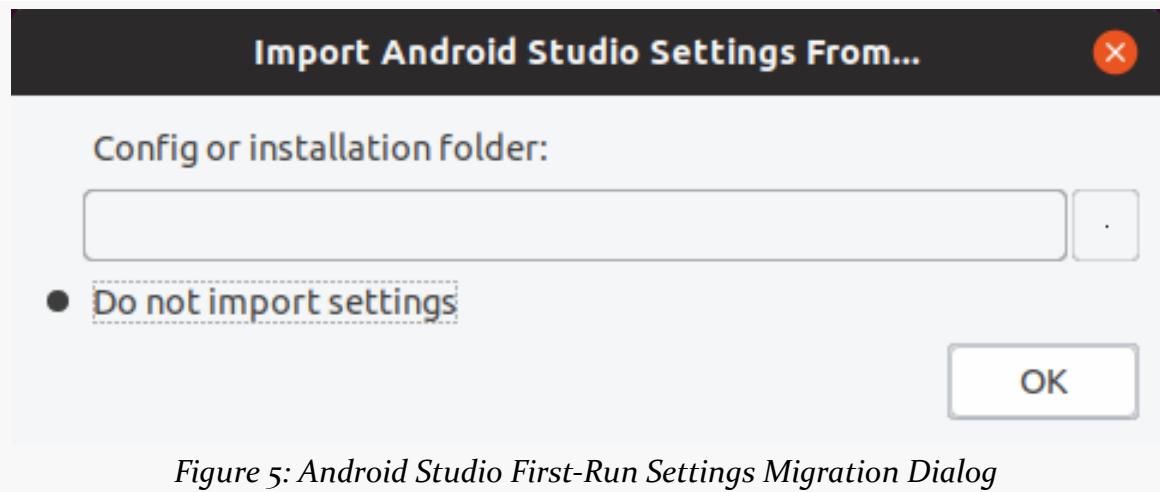


Figure 5: Android Studio First-Run Settings Migration Dialog

If you are using Android Studio for the first time, the “Do not import settings”

INSTALLING THE TOOLS

option is the correct choice to make.

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

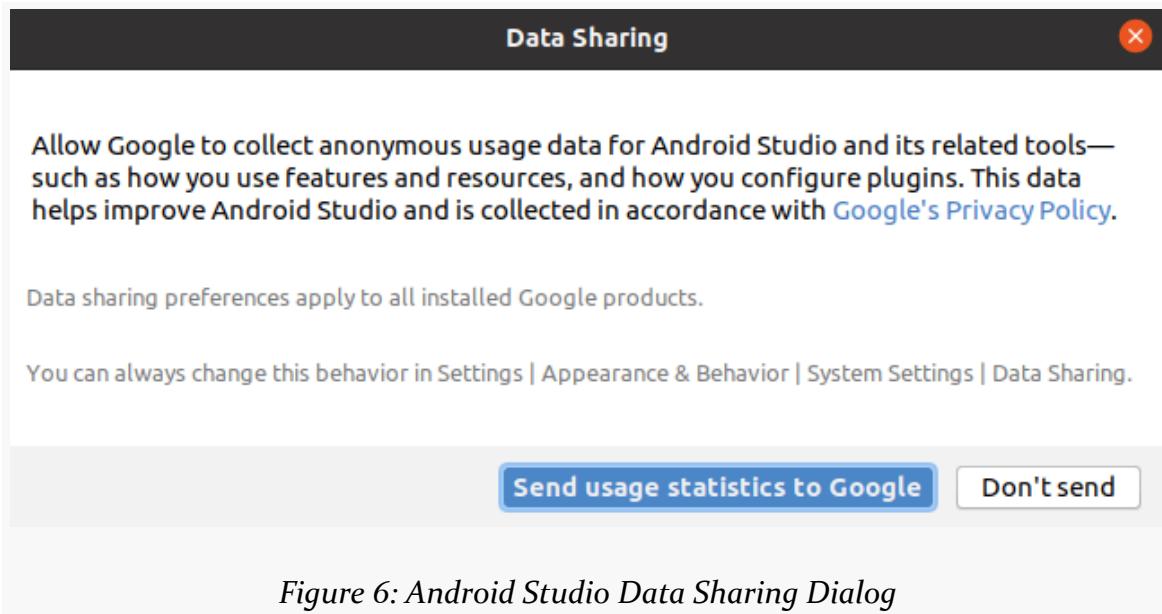


Figure 6: Android Studio Data Sharing Dialog

Click whichever button you wish.

INSTALLING THE TOOLS

Eventually, you will be taken to the Android Studio Setup Wizard:

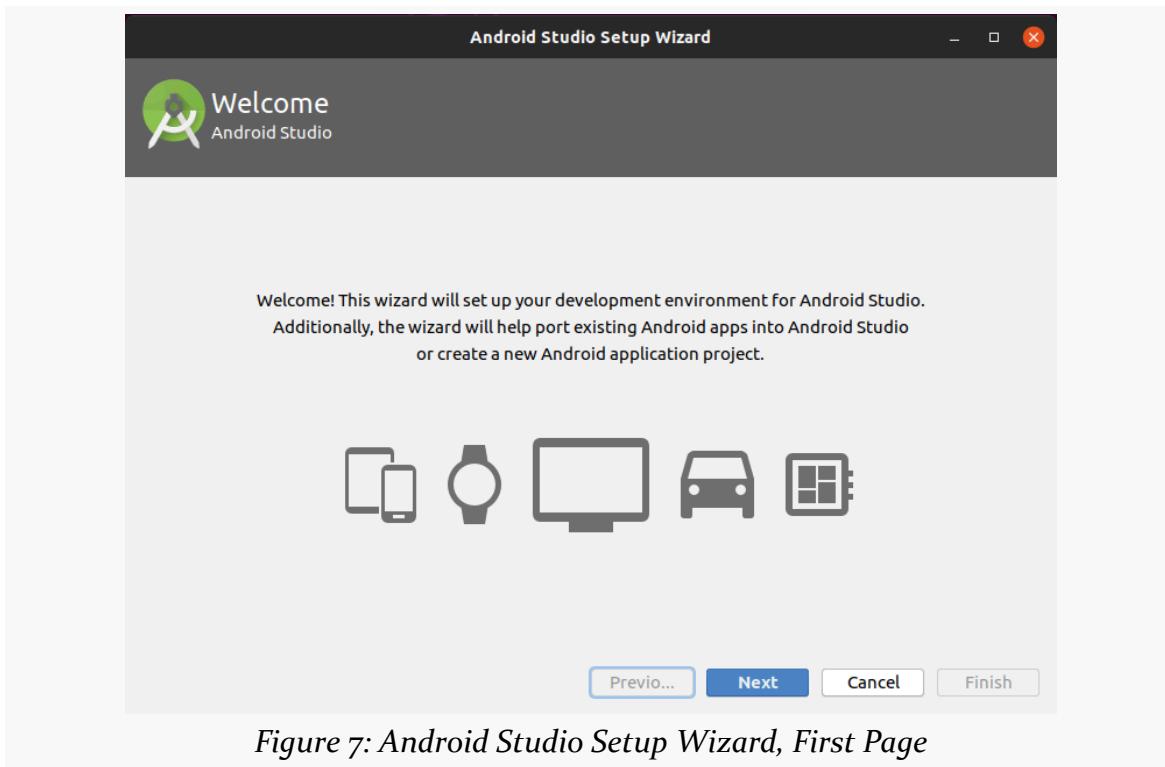


Figure 7: Android Studio Setup Wizard, First Page

INSTALLING THE TOOLS

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

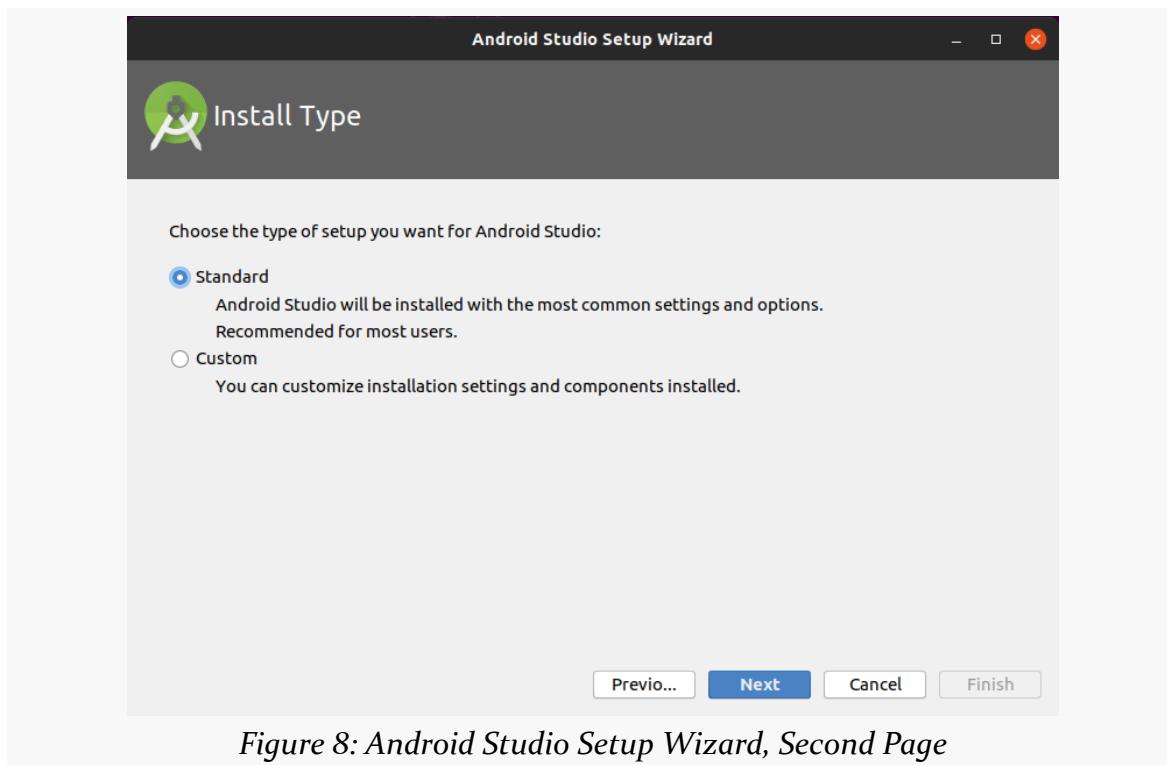


Figure 8: Android Studio Setup Wizard, Second Page

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

INSTALLING THE TOOLS

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

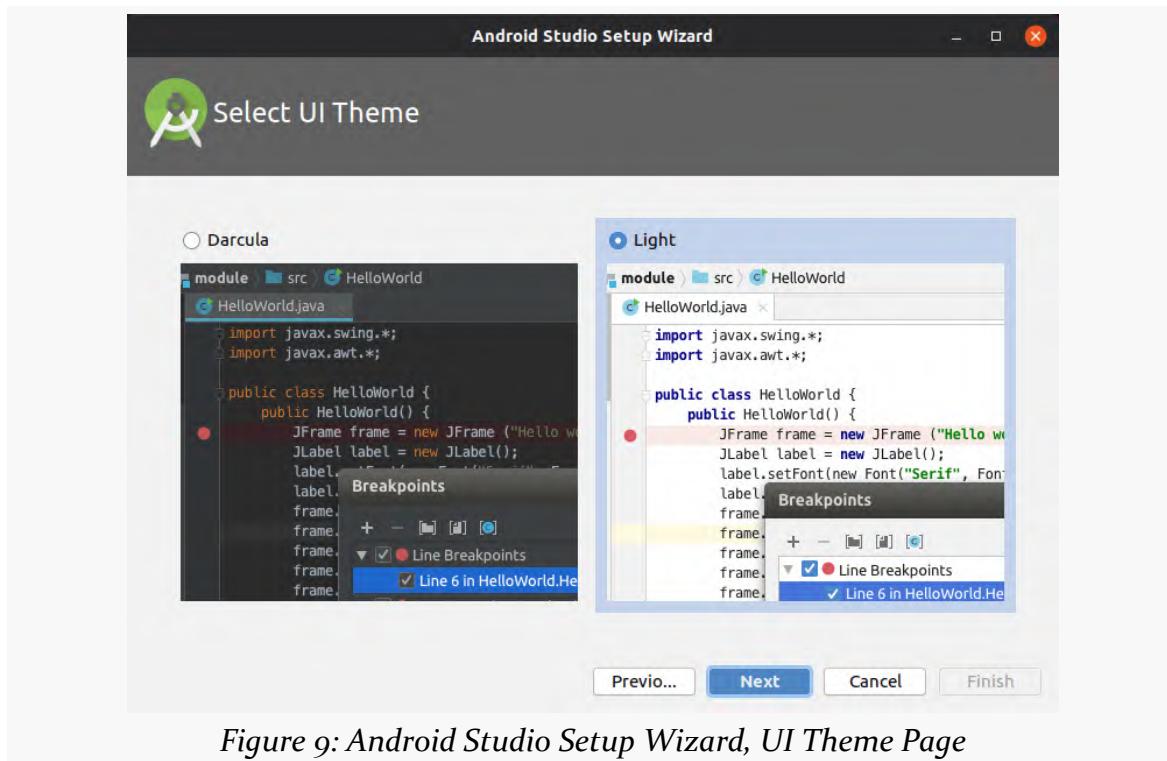


Figure 9: Android Studio Setup Wizard, UI Theme Page

INSTALLING THE TOOLS

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

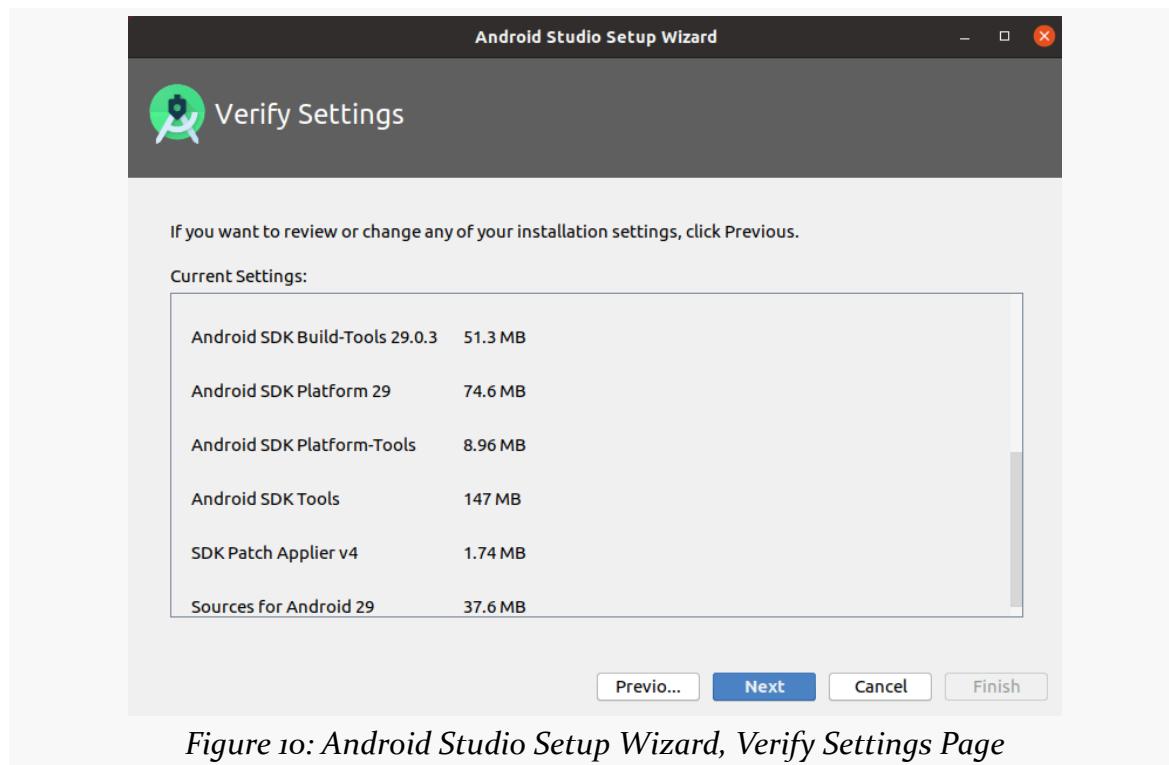


Figure 10: Android Studio Setup Wizard, Verify Settings Page

INSTALLING THE TOOLS

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

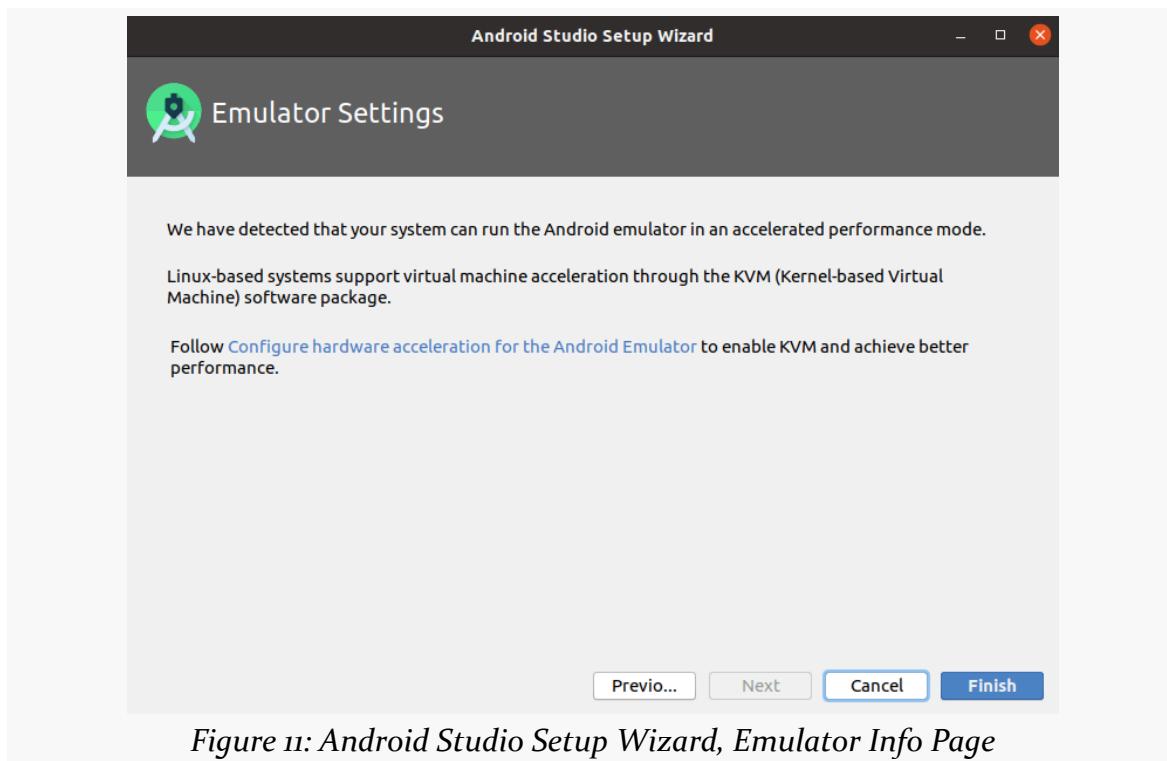


Figure 11: Android Studio Setup Wizard, Emulator Info Page

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, Android Studio will busily start downloading stuff to your development machine.

INSTALLING THE TOOLS

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

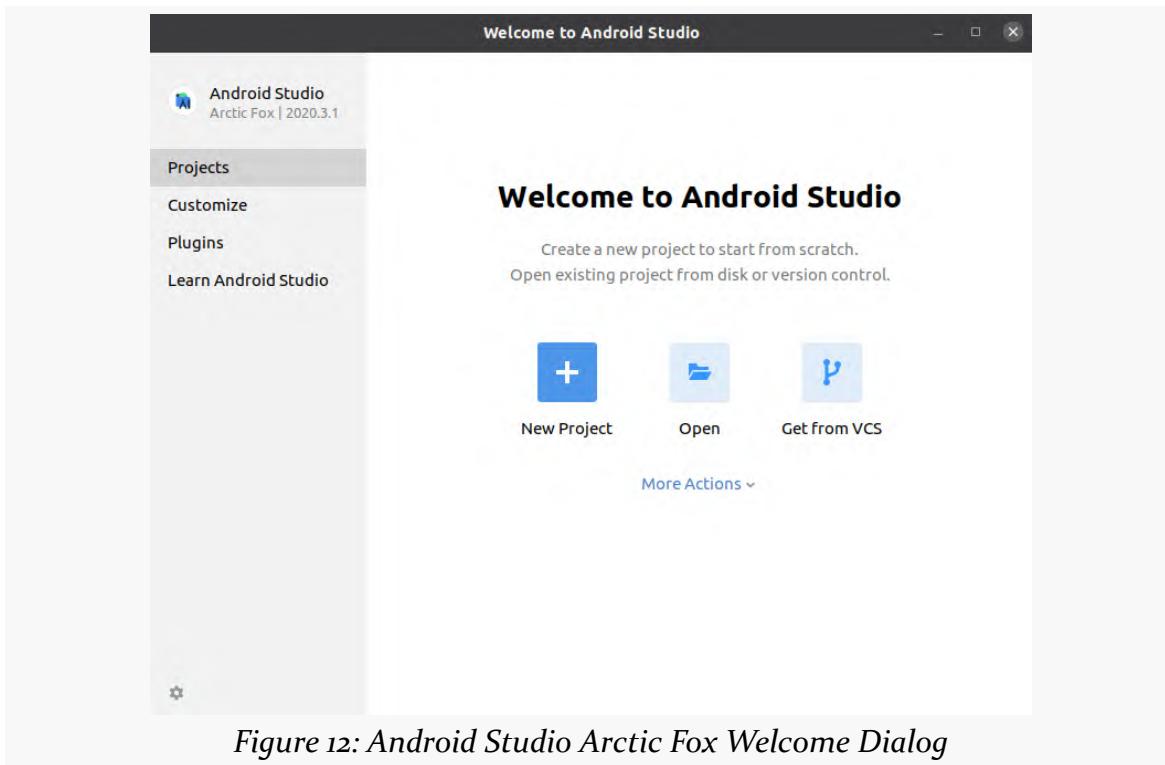


Figure 12: Android Studio Arctic Fox Welcome Dialog

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:

CREATING A STARTER PROJECT

```
#Sun Aug 15 13:58:44 EDT 2021
distributionBase=GRADLE_USER_HOME
distributionUrl=https\://services.gradle.org/distributions/gradle-7.0.2-bin.zip
distributionPath=wrapper/dists
zipStorePath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
```

(from [ToDo-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.

CREATING A STARTER PROJECT

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

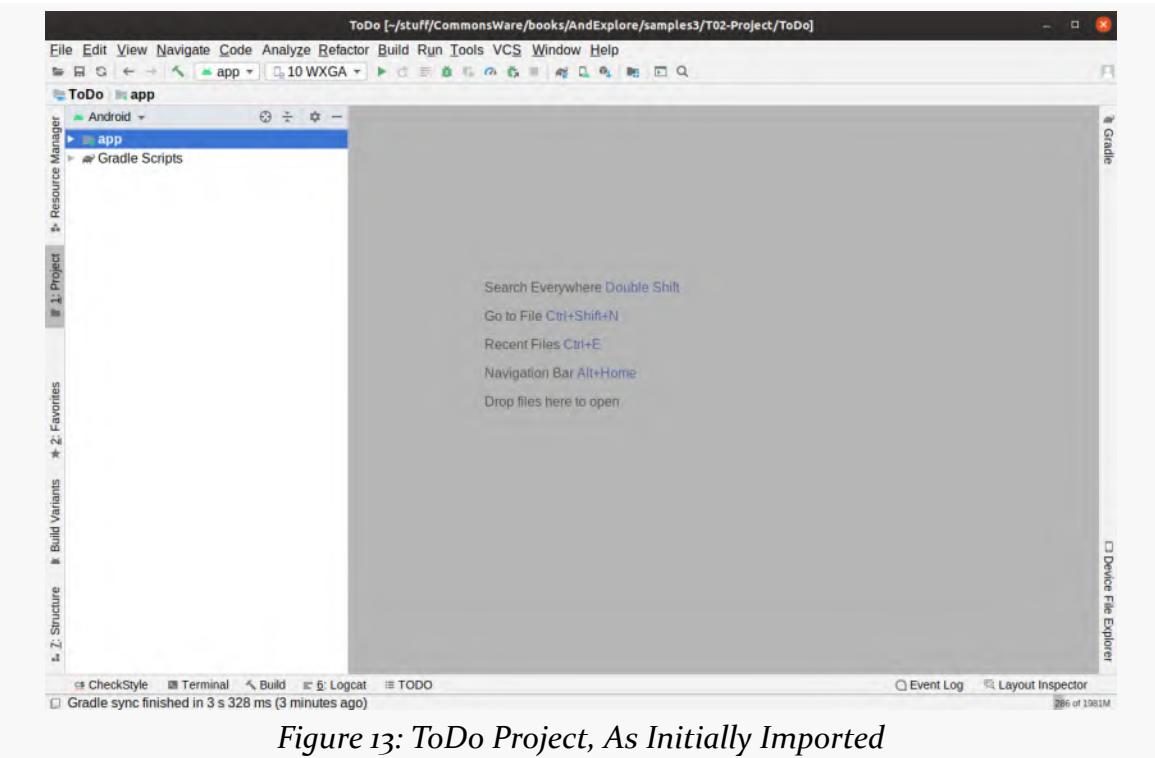


Figure 13: ToDo Project, As Initially Imported

CREATING A STARTER PROJECT

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”:

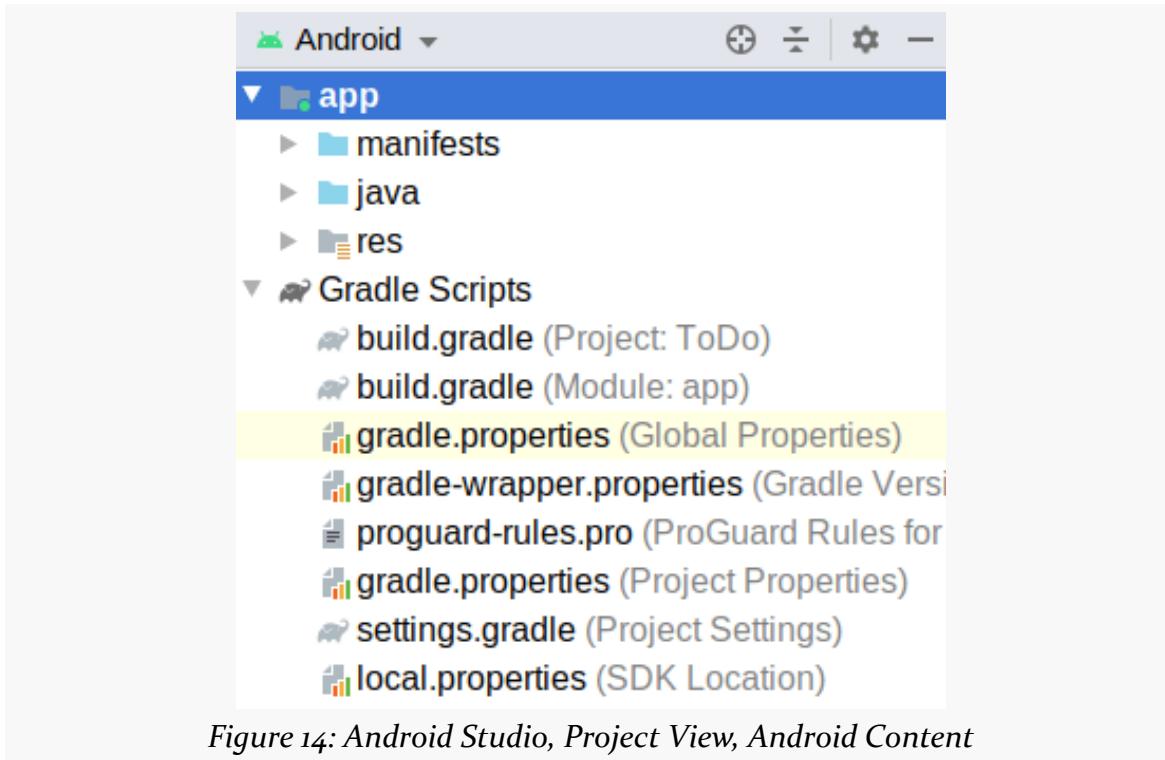


Figure 14: Android Studio, Project View, Android Content

CREATING A STARTER PROJECT

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:

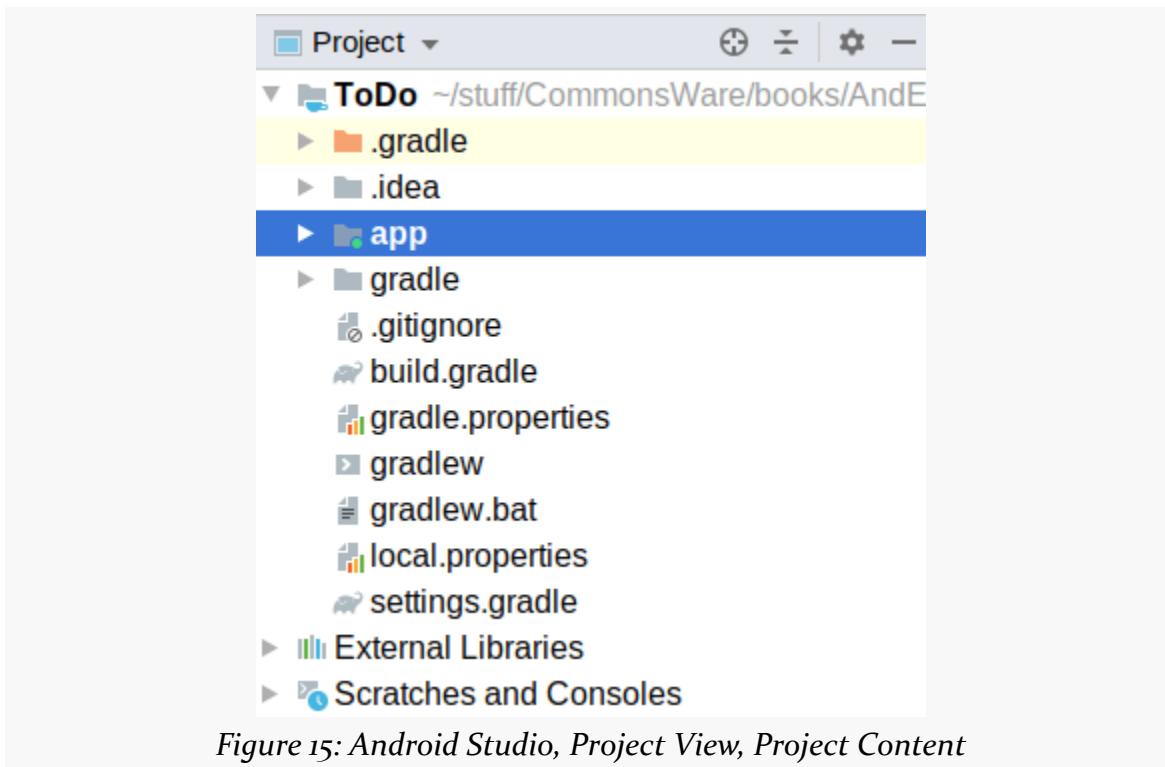


Figure 15: Android Studio, Project View, Project Content

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.

CREATING A STARTER PROJECT

By default, you have no AVDs to use:

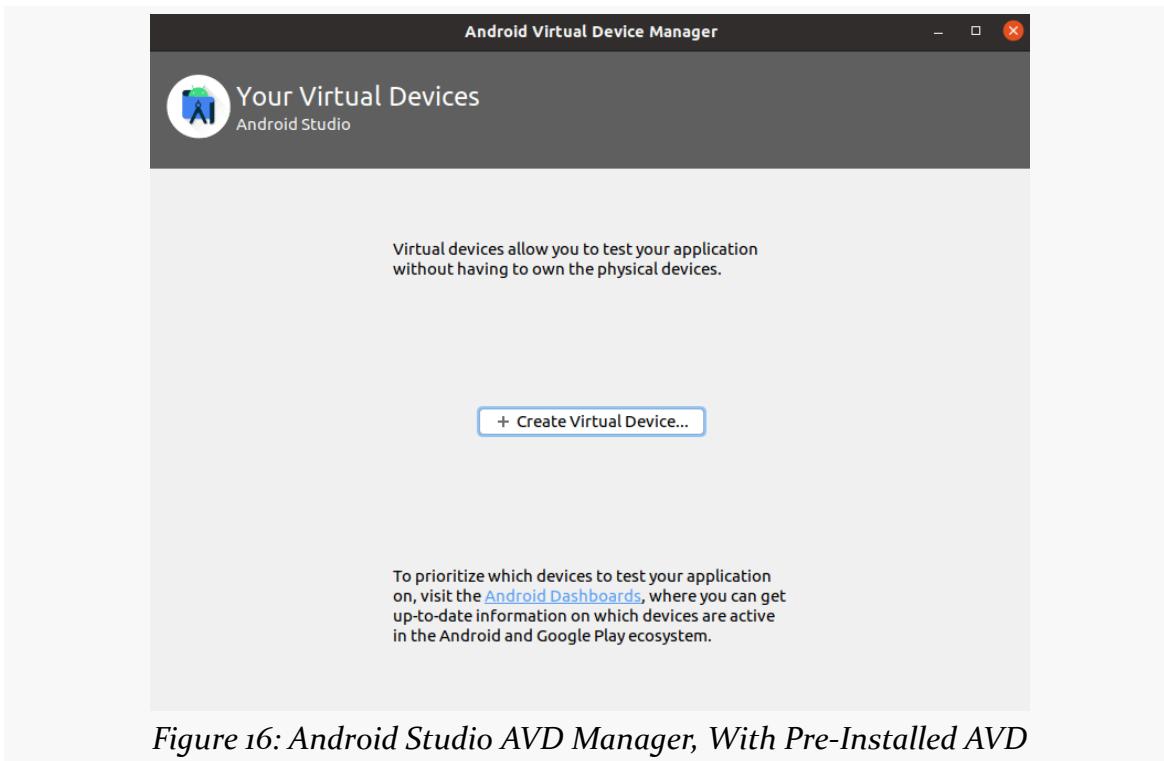


Figure 16: Android Studio AVD Manager, With Pre-Installed AVD

CREATING A STARTER PROJECT

To create an AVD, click the “Create Virtual Device” button, which brings up a “Virtual Device Configuration” wizard:

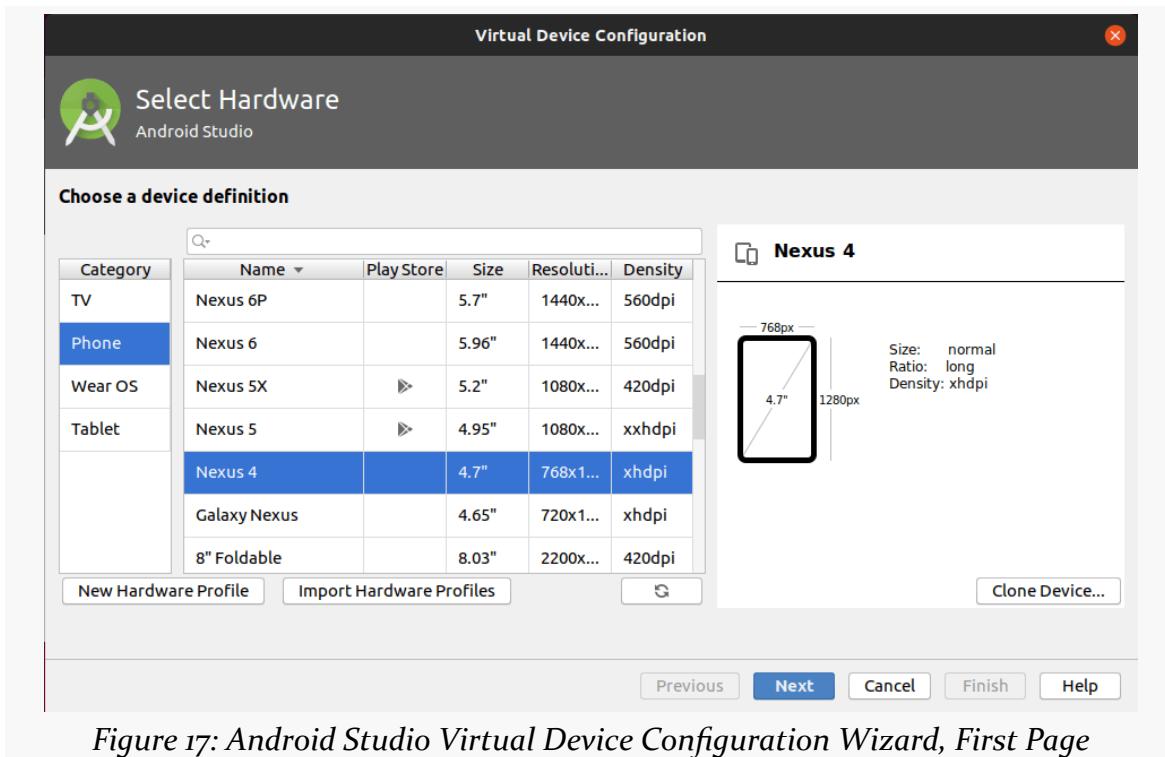


Figure 17: Android Studio Virtual Device Configuration Wizard, First Page

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 or 1280x720 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.

CREATING A STARTER PROJECT

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

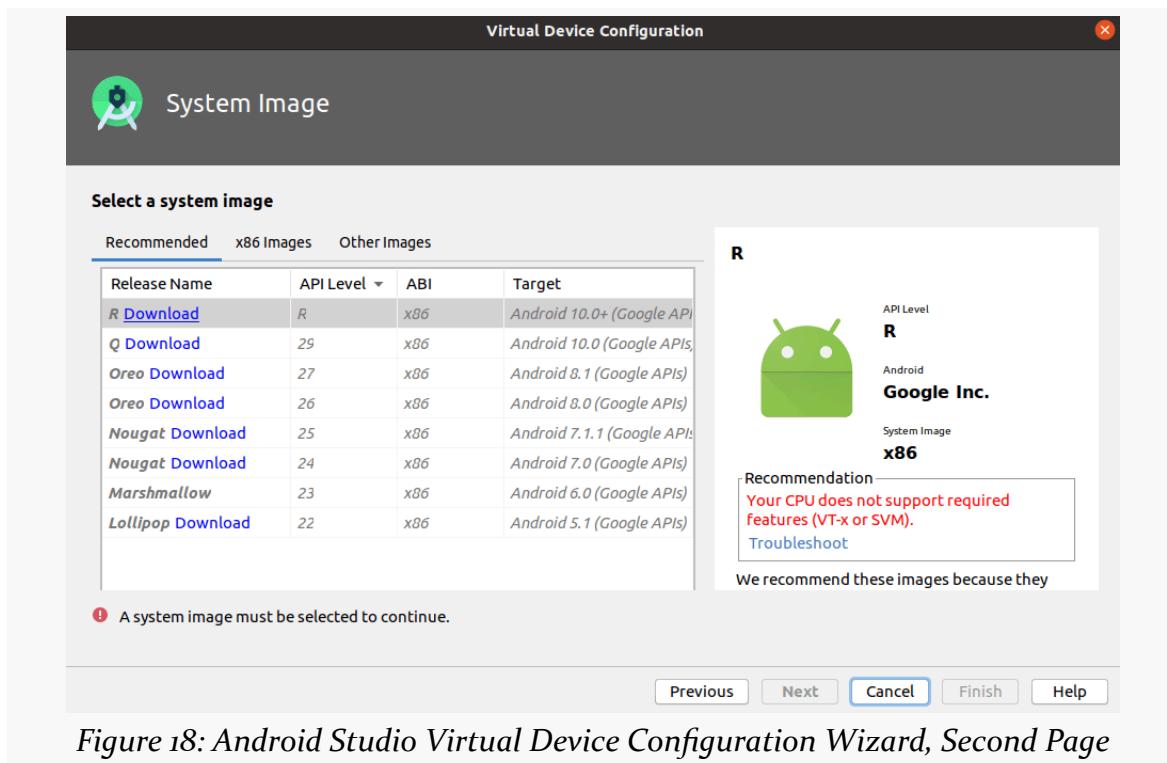


Figure 18: Android Studio Virtual Device Configuration Wizard, Second Page

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.

CREATING A STARTER PROJECT

It is best to use one of the x86 images for the best emulator performance. On the “x86 Images” tab, you should see some entries with a “Download” link, and you might see others without it. 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 another license dialog and progress dialog:

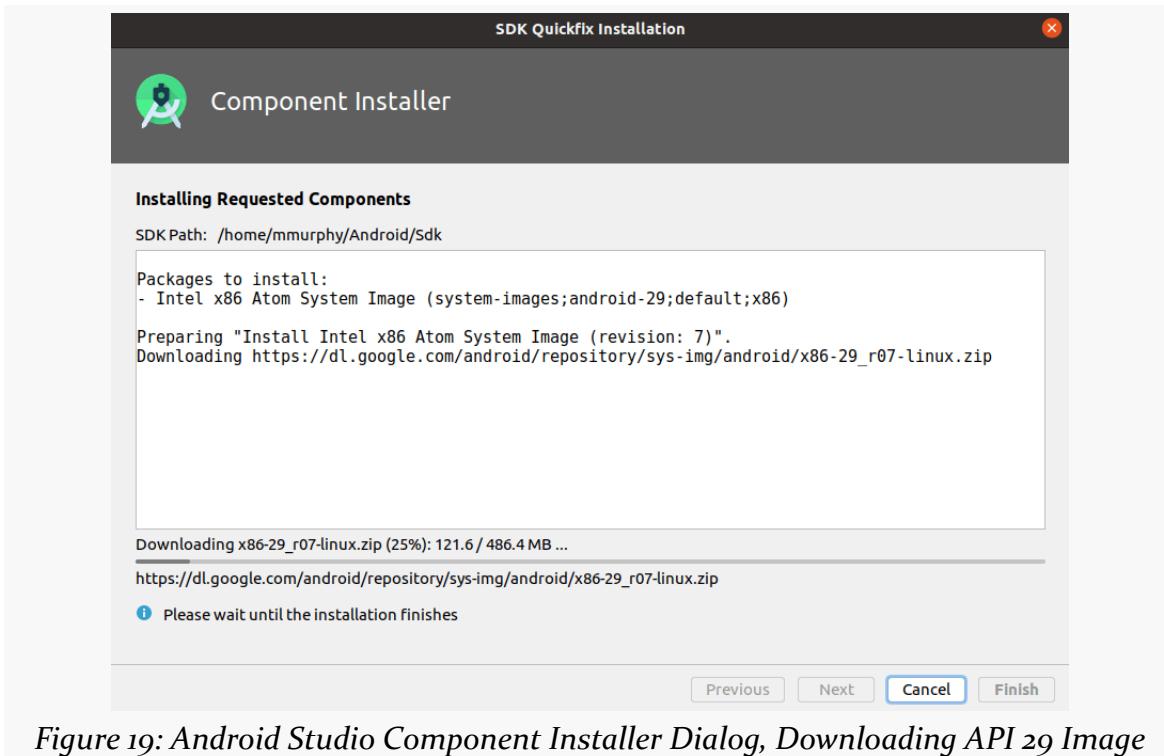


Figure 19: *Android Studio Component Installer Dialog, Downloading API 29 Image*

CREATING A STARTER PROJECT

Once you have identified the image that you want — and have downloaded it if needed — click on one of them in the wizard. Clicking “Next” allows you to finalize the configuration of your AVD:

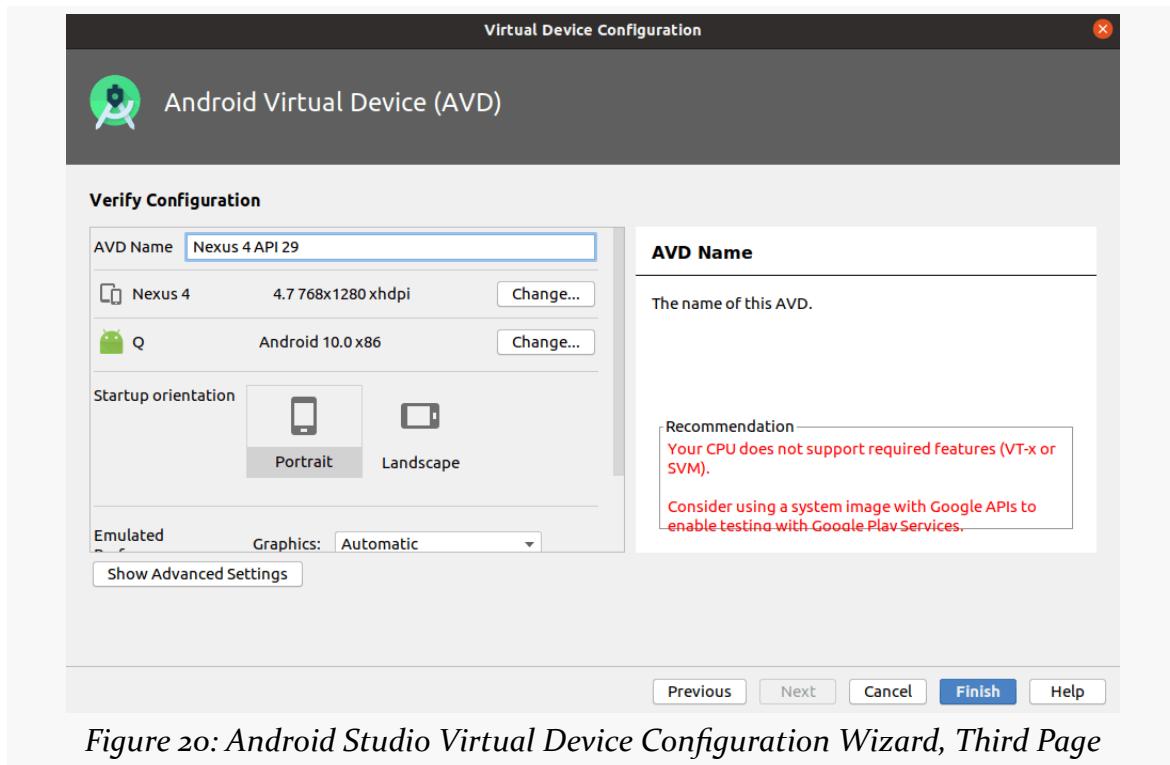


Figure 20: Android Studio Virtual Device Configuration Wizard, Third Page

If you get the “Recommendation” box with the red “Your CPU does not support required features...” message, 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.

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

If you also have a physical device that you want to use for testing, continue with Step #3. Otherwise, feel free to skip to Step #4.

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)

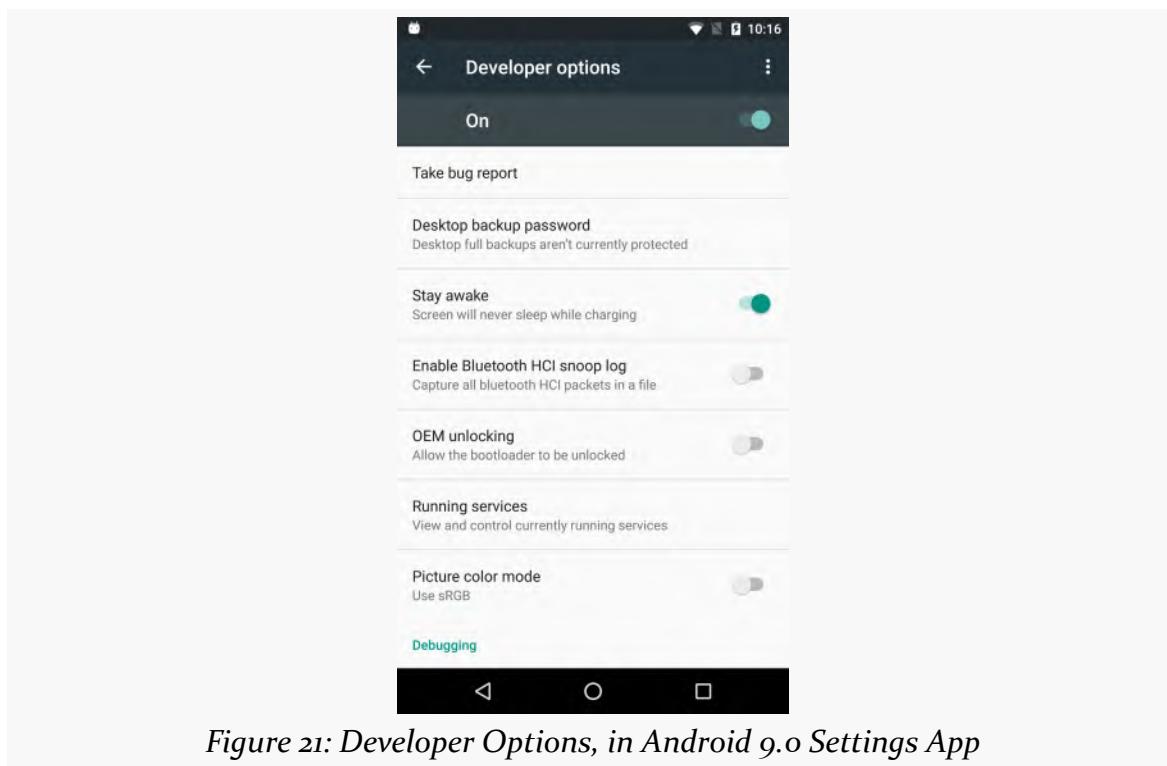


Figure 21: Developer Options, in Android 9.0 Settings App

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.

CREATING A STARTER PROJECT

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

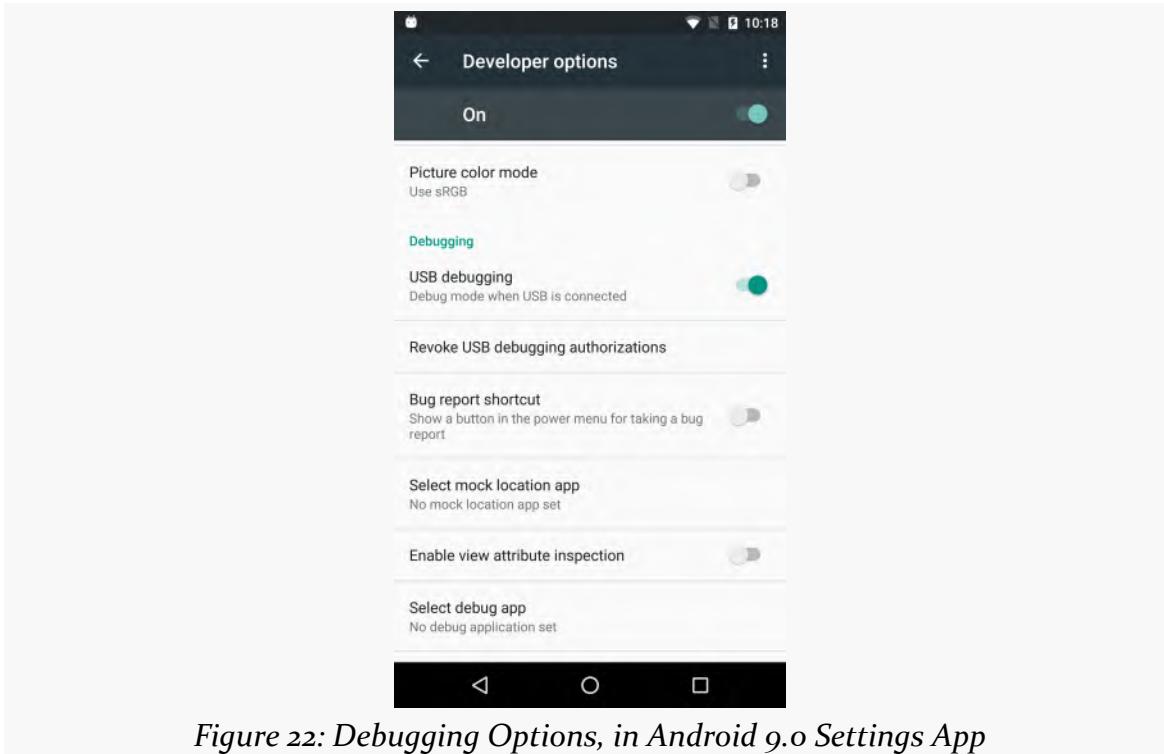


Figure 22: Debugging Options, in Android 9.0 Settings App

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.

CREATING A STARTER PROJECT

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:

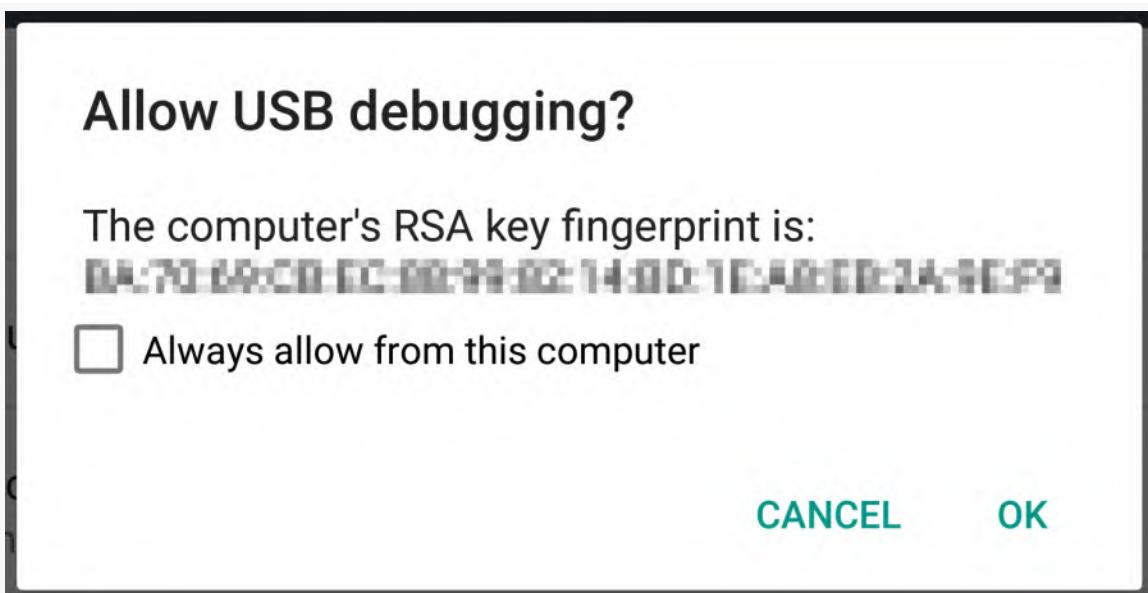


Figure 23: Allow USB Debugging Dialog

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

It is likely that simply plugging in your device will “just work”.

If you are running Ubuntu (or perhaps other Linux variants), and when you later try running your app it appears that Android Studio does not “see” your device, you may need to add some udev rules. [This GitHub repository](#) contains some instructions and a large file showing the rules for devices from a variety of manufacturers, and [this blog post](#) provides more details of how to work with udev rules for Android devices.

Step #4: Running the Project

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

Android Studio has a toolbar just below the main menu. In that toolbar, you will find two drop-down lists, followed by the Run toolbar button (usually depicted as a green rightward-pointing triangle):



Figure 24: Android Studio Toolbar Segment

CREATING A STARTER PROJECT

The first drop-down says “this is what I want to run”. Right now, your only viable option is “app”, referring to the app that this project builds.

The second drop-down says “this is where I want to run it”. Here, you will find a list of devices and emulators that are available to you.

To run the app, choose your desired device or emulator in the second drop-down, then click the Run toolbar button. If you choose an emulator, and the emulator is not already running, Android Studio will start it up. Then, after a short wait, your app should appear on it:

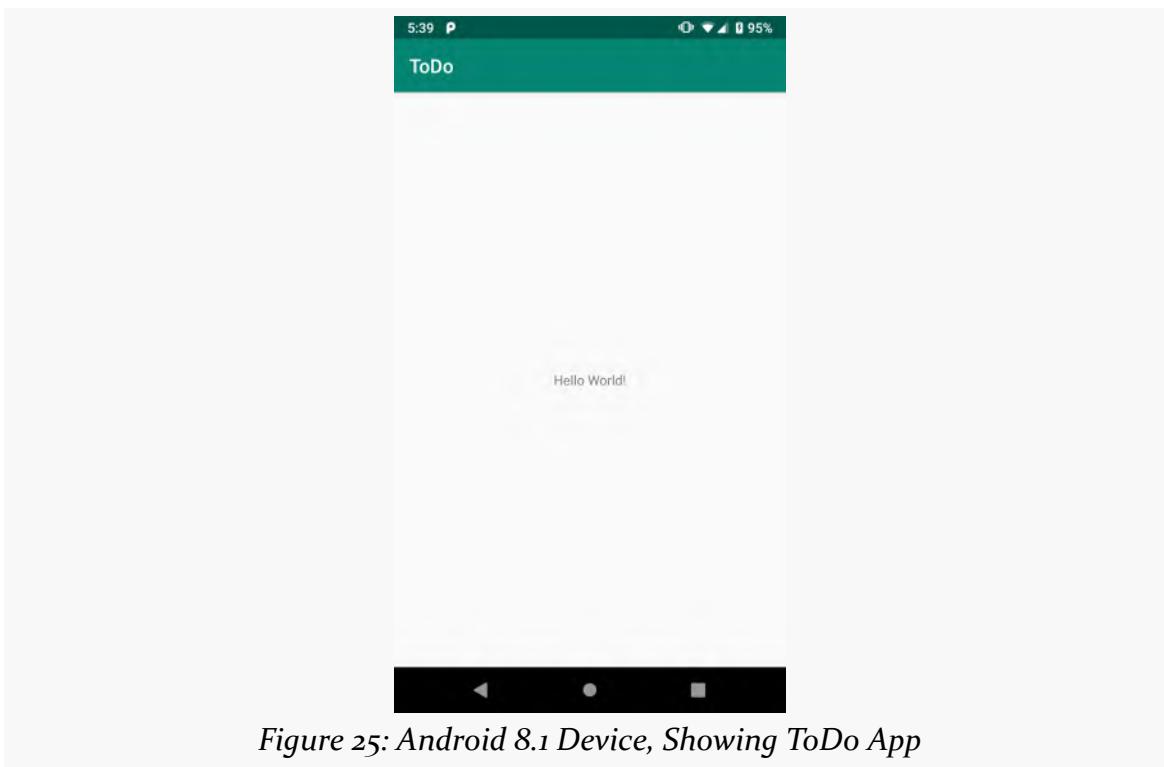


Figure 25: Android 8.1 Device, Showing ToDo App

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

CREATING A STARTER PROJECT

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



Figure 26: Android Emulator Cold-Boot Warning

The emulator behaves a bit more like an Android device. Closing the emulator window is 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".

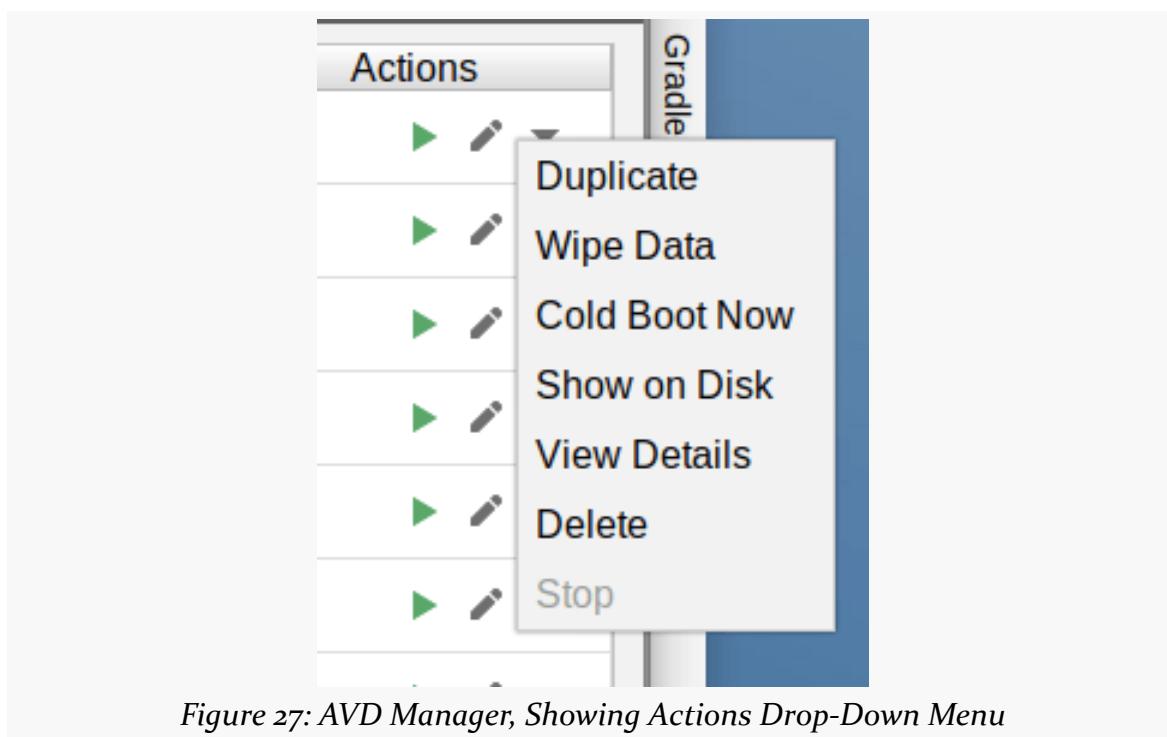


Figure 27: AVD Manager, Showing Actions Drop-Down Menu

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 GitLab 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.

MODIFYING THE MANIFEST

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:

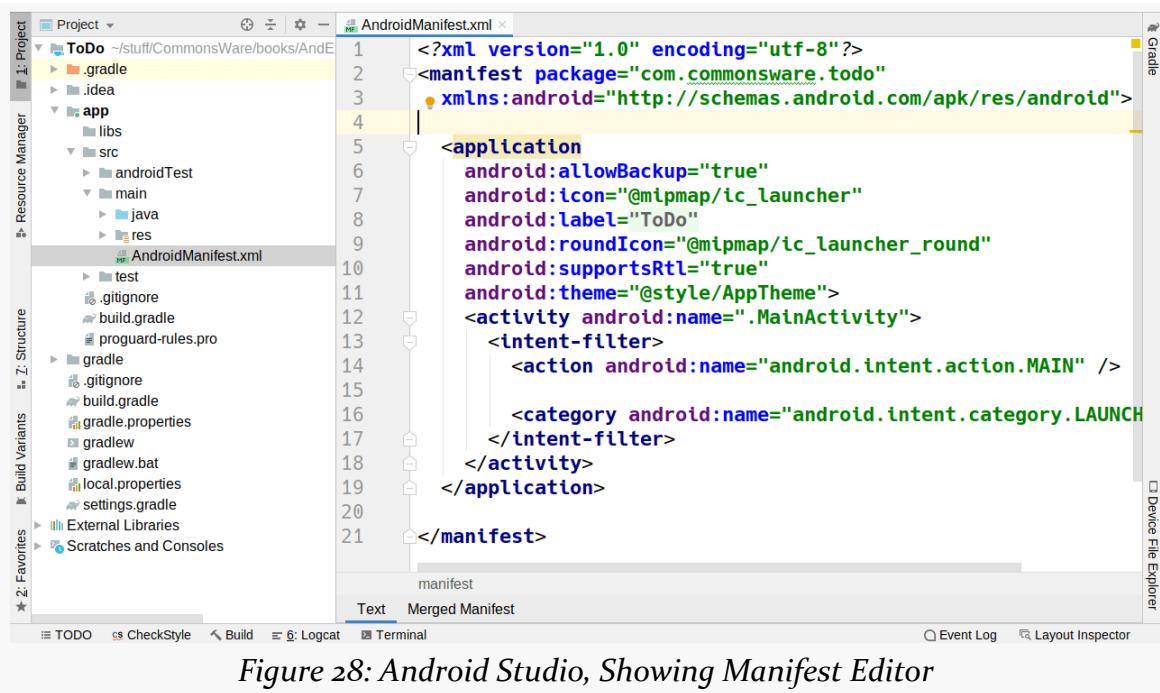


Figure 28: Android Studio, Showing Manifest Editor

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:

MODIFYING THE MANIFEST

```
<?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/Theme.ToDo">
    <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`.

Final Results

At this point, your manifest should look like:

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

MODIFYING THE MANIFEST

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.todo">

    <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/Theme.ToDo">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

(from [To3-Manifest/ToDo/app/src/main/AndroidManifest.xml](#))

What We Changed

The book's GitLab 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 GitLab 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 launcher.

CHANGING OUR ICON

This piece of [clipart](#), originally from OpenClipArt.org, will serve this purpose:

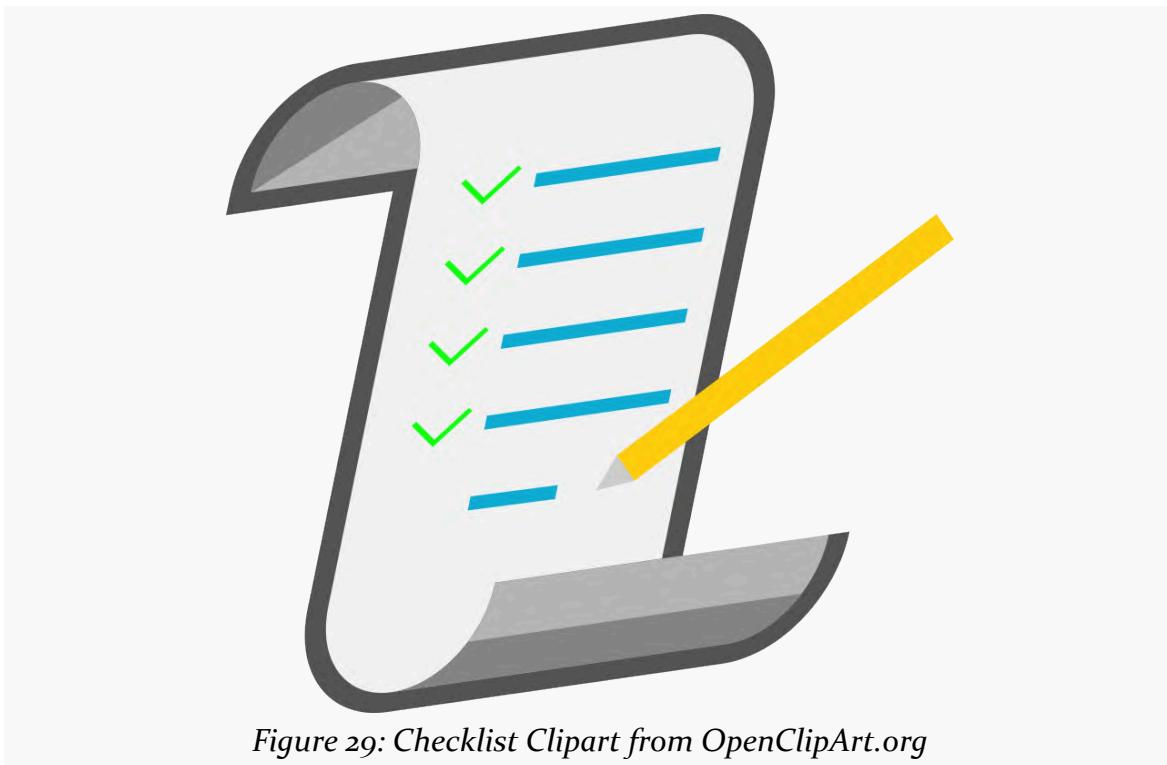


Figure 29: Checklist Clipart from OpenClipArt.org

Download [the 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 great 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.

CHANGING OUR ICON

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

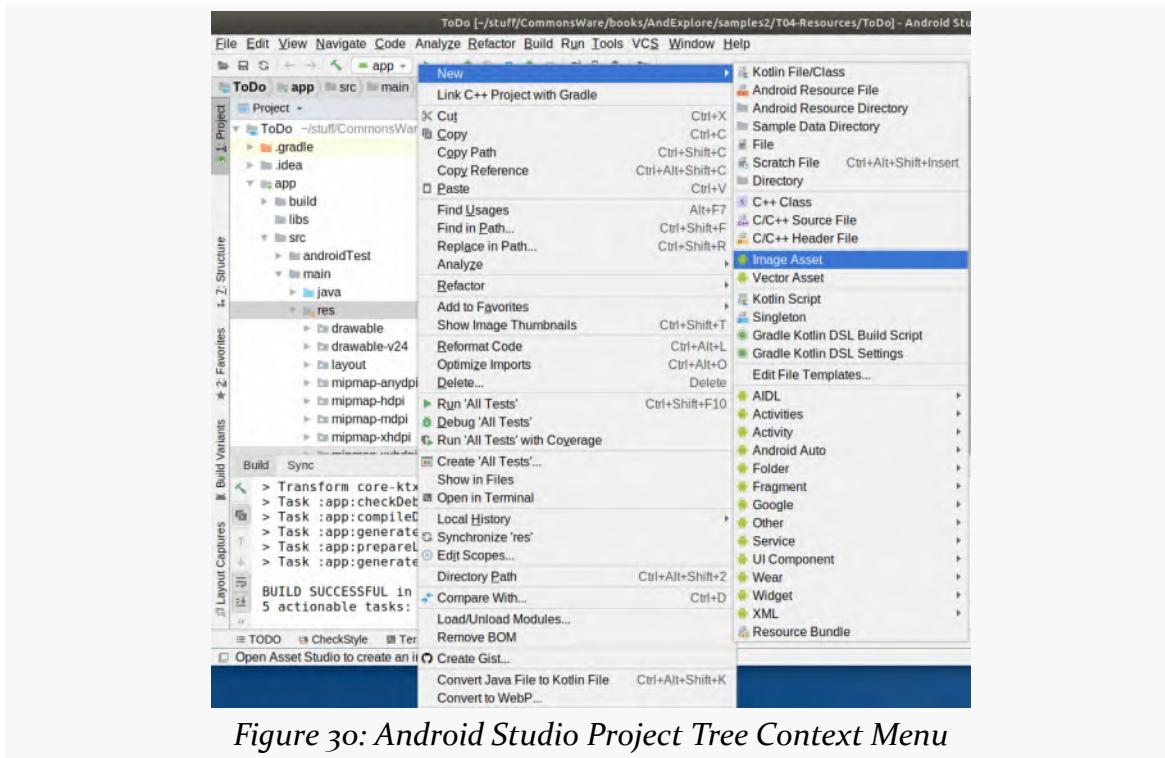


Figure 30: Android Studio Project Tree Context Menu

CHANGING OUR ICON

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

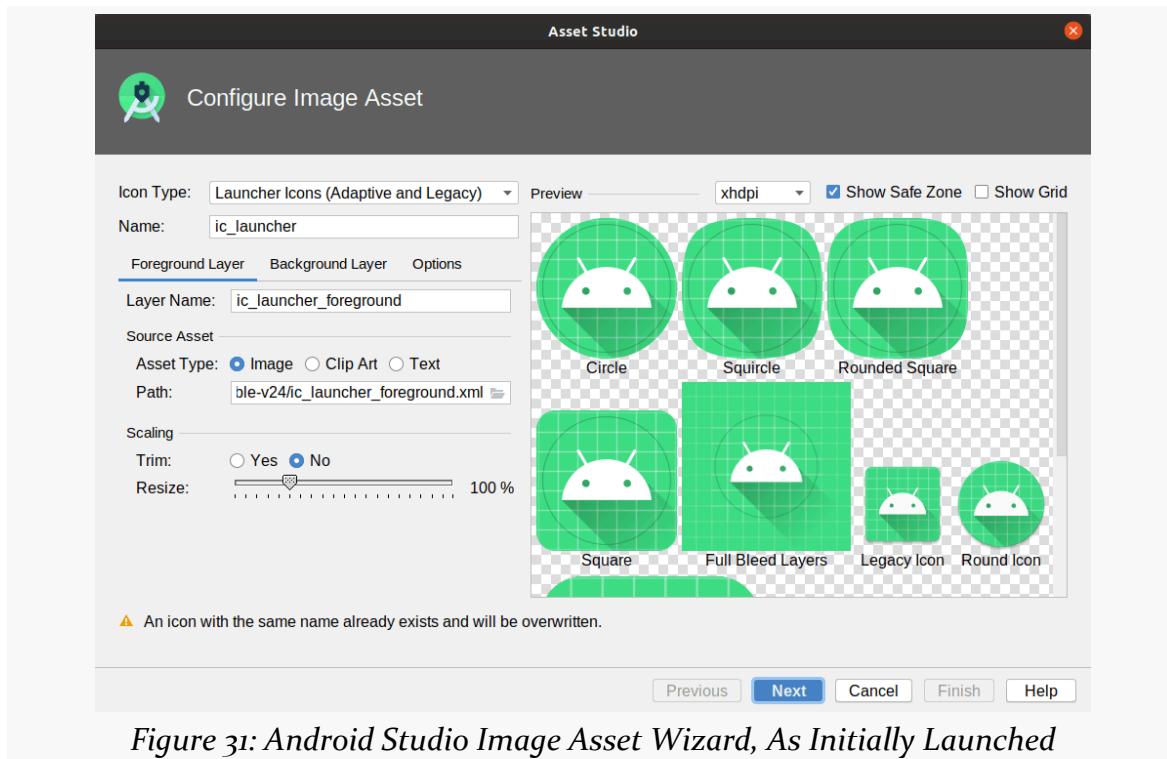


Figure 31: Android Studio Image Asset Wizard, As Initially Launched

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.

CHANGING OUR ICON

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

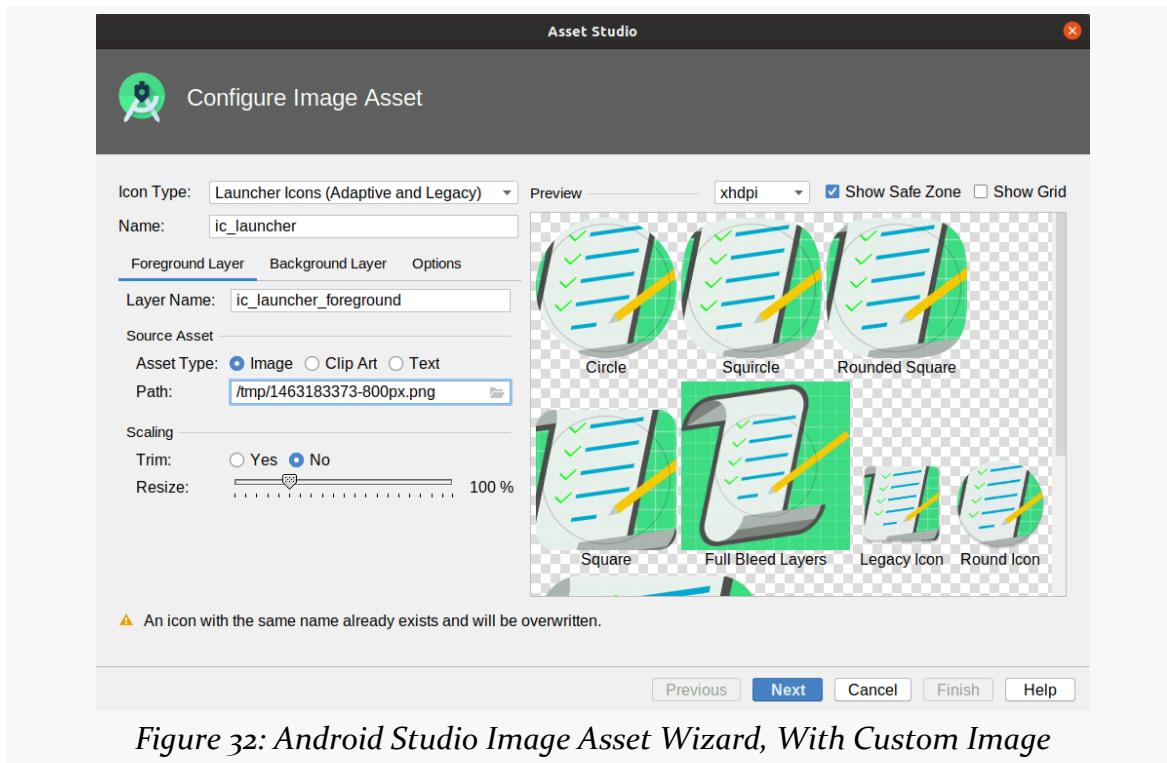


Figure 32: *Android Studio Image Asset Wizard, With Custom Image*

CHANGING OUR ICON

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:

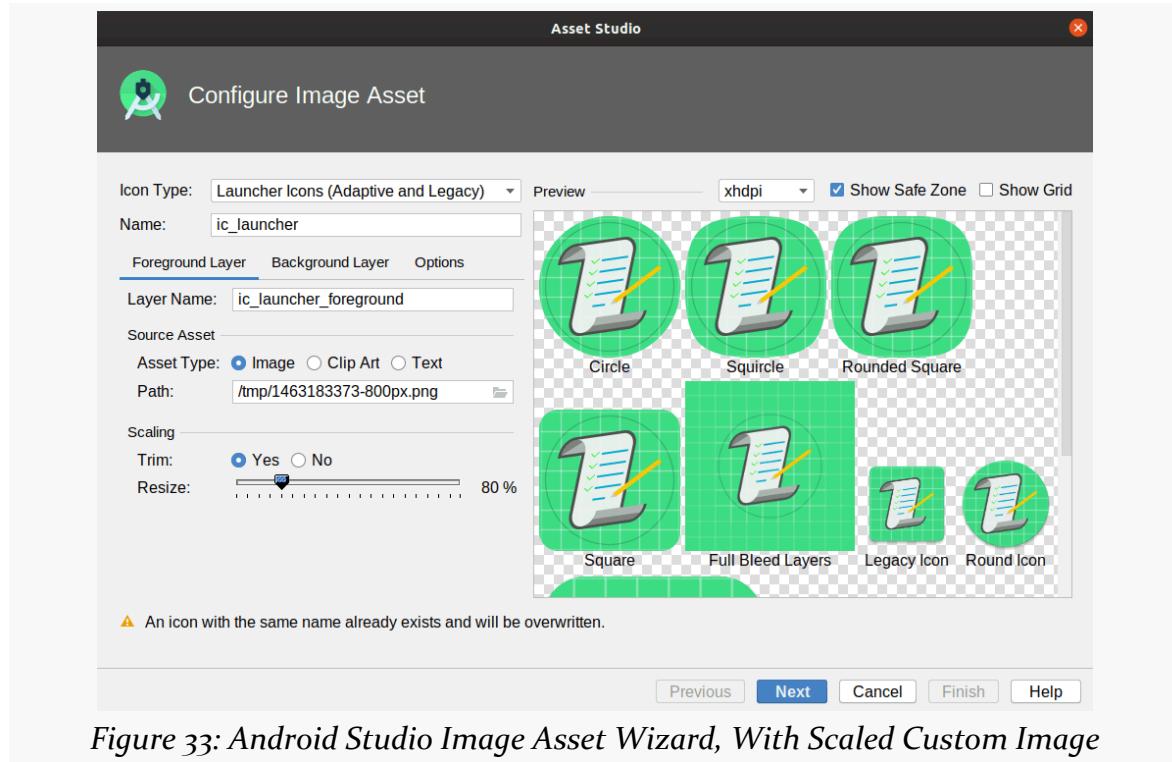


Figure 33: Android Studio Image Asset Wizard, With Scaled Custom Image

CHANGING OUR ICON

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

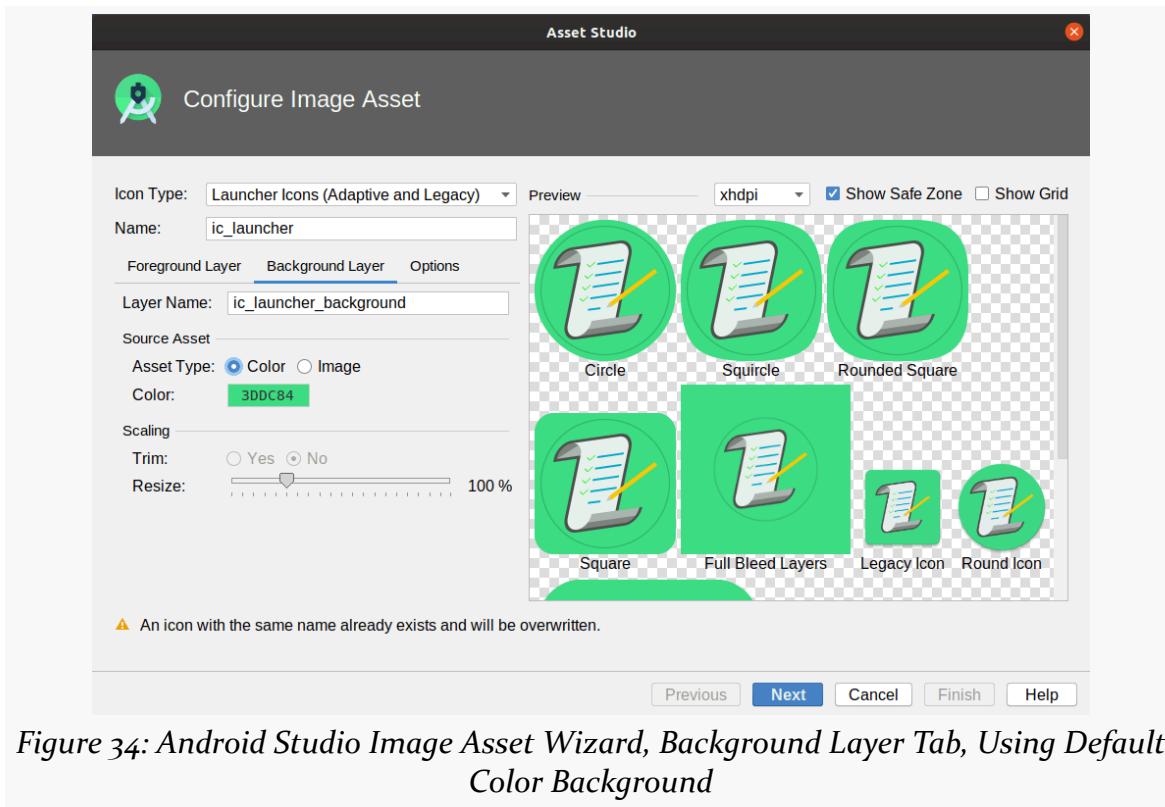


Figure 34: Android Studio Image Asset Wizard, Background Layer Tab, Using Default Color Background

CHANGING OUR ICON

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

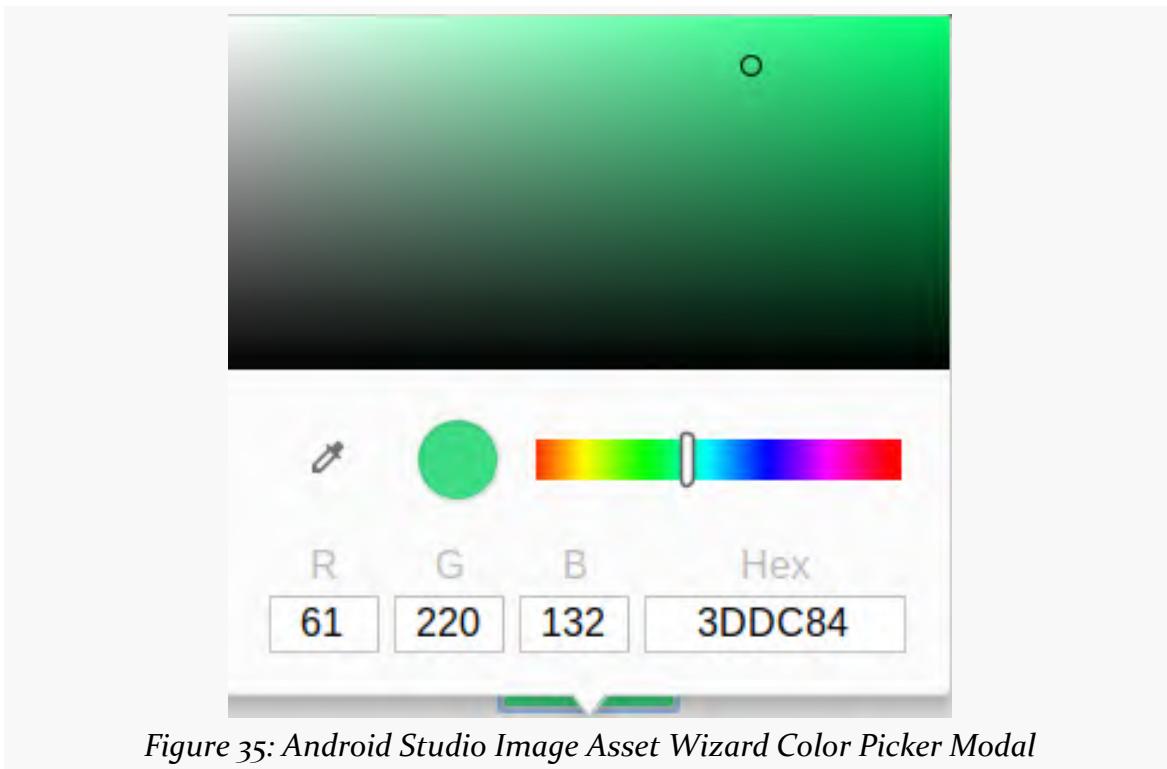


Figure 35: Android Studio Image Asset Wizard Color Picker Modal

CHANGING OUR ICON

Pick some other color (such as #006144) to apply it to the icon background, then click anywhere outside the modal to dismiss it:

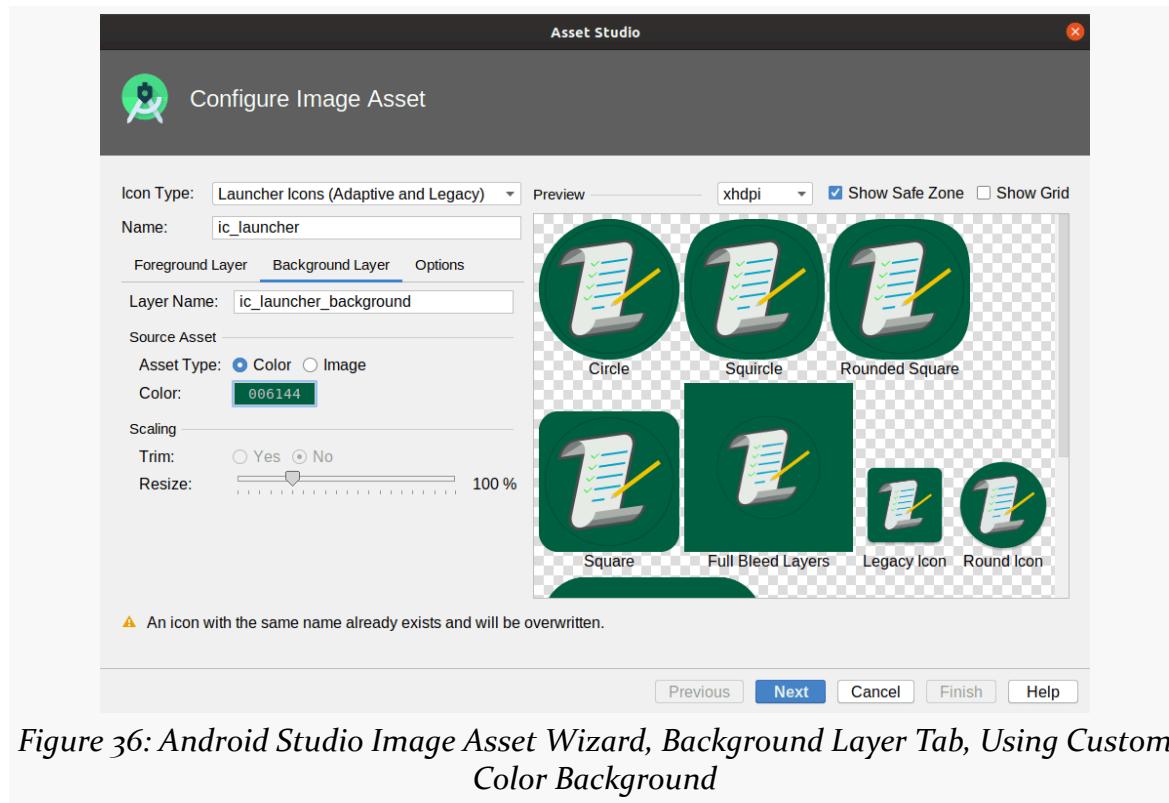


Figure 36: Android Studio Image Asset Wizard, Background Layer Tab, Using Custom Color Background

CHANGING OUR ICON

Then, switch to the “Options” 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”:

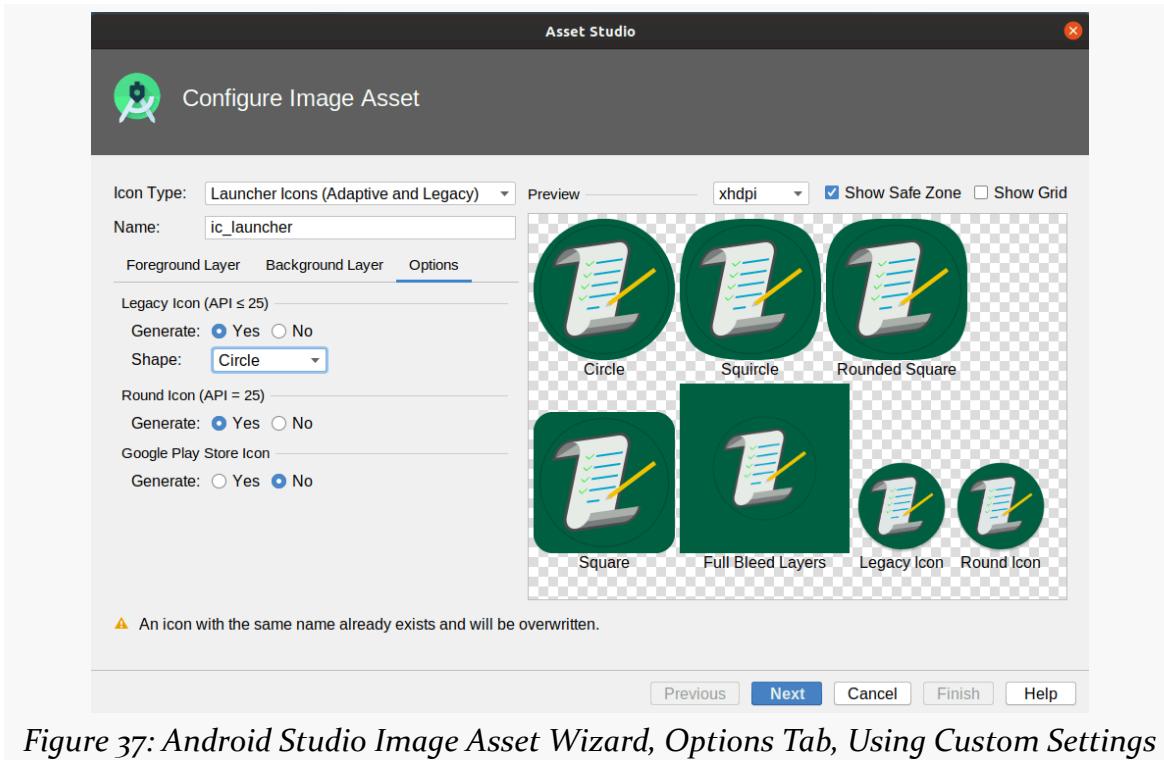


Figure 37: Android Studio Image Asset Wizard, Options Tab, Using Custom Settings

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 launchers on older devices — our icon will be our clipart on our chosen background color, but with a shape determined by the launcher implementation.

CHANGING OUR ICON

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

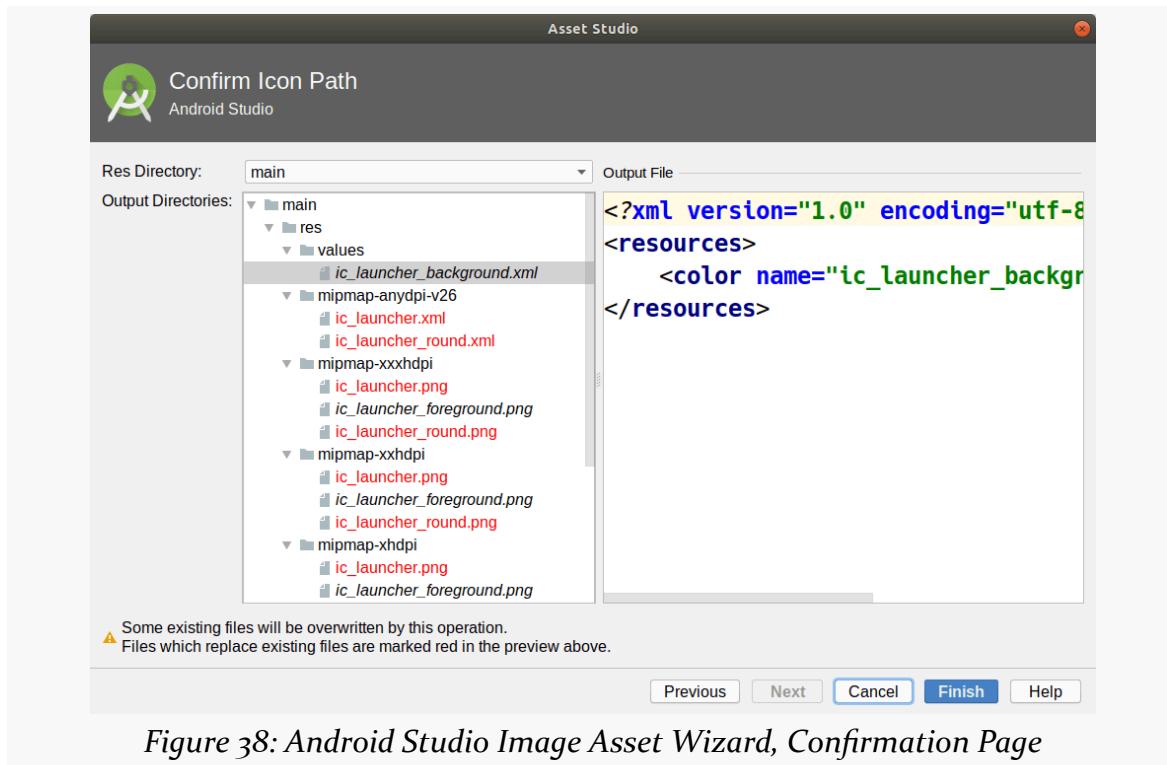


Figure 38: Android Studio Image Asset Wizard, Confirmation Page

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 launcher, you will see that it shows up with the new icon:

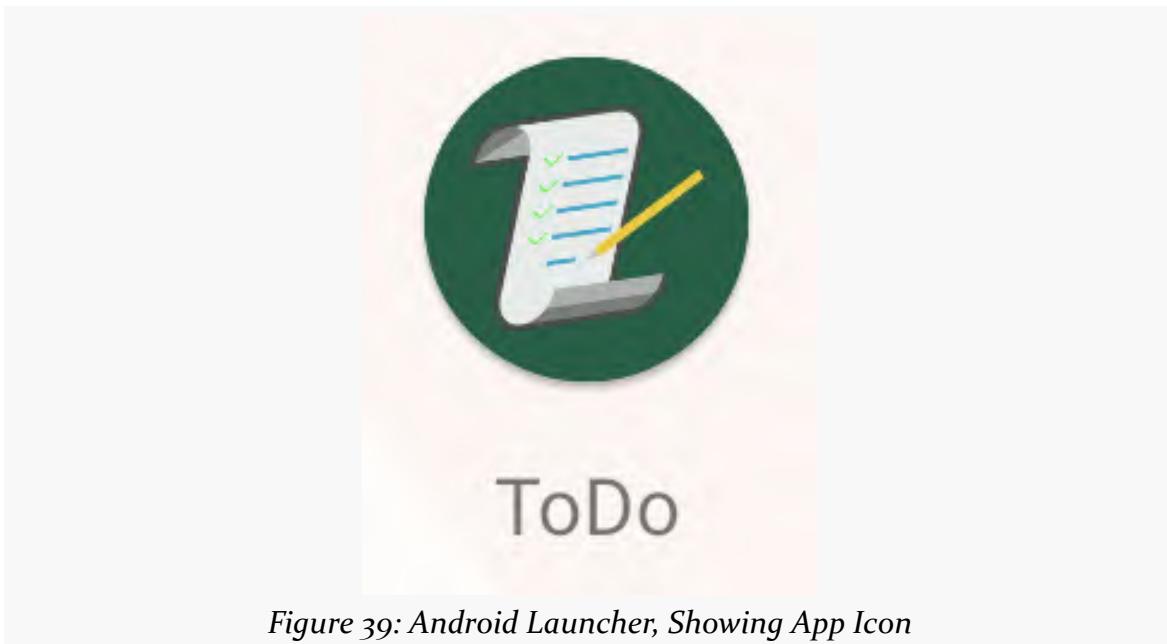


Figure 39: Android Launcher, Showing App Icon

What We Changed

The book's GitLab 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 GitLab 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: Examining What We Have

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

```
dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

(from [To4-Resources/ToDo/app/build.gradle](#))

ADDING A LIBRARY

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.

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:1.2.1"
```

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

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

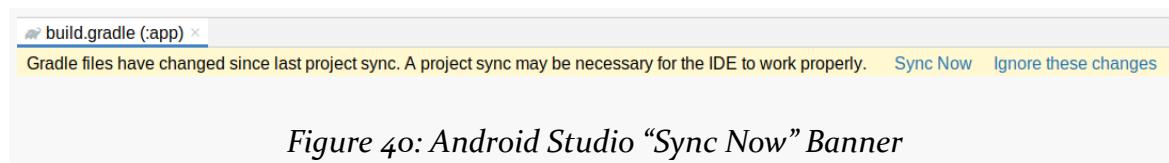


Figure 40: Android Studio “Sync Now” Banner

Go ahead and click the “Sync Now” link in the banner at the top of the editor.

Final Results

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

ADDING A LIBRARY

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

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

    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    kotlinOptions {
        jvmTarget = '1.8'
    }
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

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

What We Changed

The book's GitLab 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 GitLab 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`. Open that in the IDE.

If it does not open up showing you XML, look towards the upper-right corner of the editor for a small toolbar of icons:



Code Split Design

Figure 41: Android Studio Layout Editor Toolbar

Click the “Code” button to switch to viewing the XML of the layout.

CONSTRUCTING A LAYOUT

That XML should 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">

    <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>
```

(from [T05-Libraries/ToDo/app/src/main/res/layout/activity_main.xml](#))

We have a `ConstraintLayout` as our root container. `ConstraintLayout` comes from that `androidx.constraintlayout:constraintlayout` 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.

CONSTRUCTING A LAYOUT

As it turns out, we can use both the ConstraintLayout and the TextView in the UI that we are going to construct:

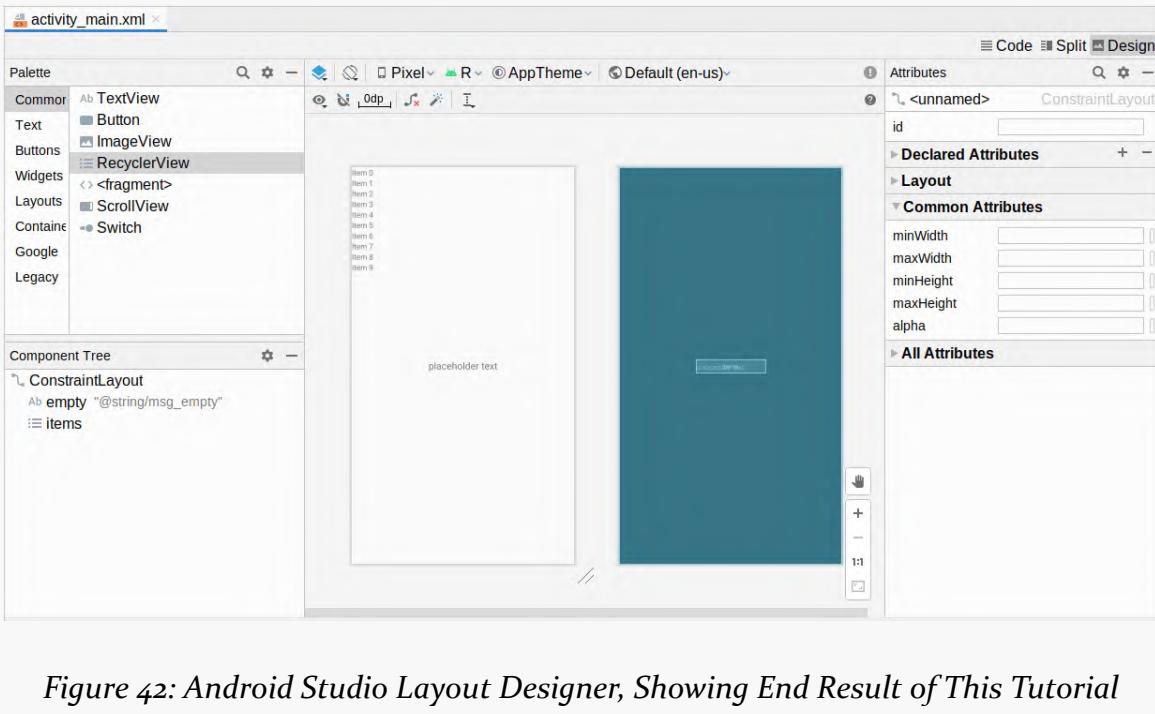


Figure 42: Android Studio Layout Designer, Showing End Result of This Tutorial

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 that toolbar, click the “Design” button, to switch to the design view. Then, in the “Palette” area, switch to “Common” category:

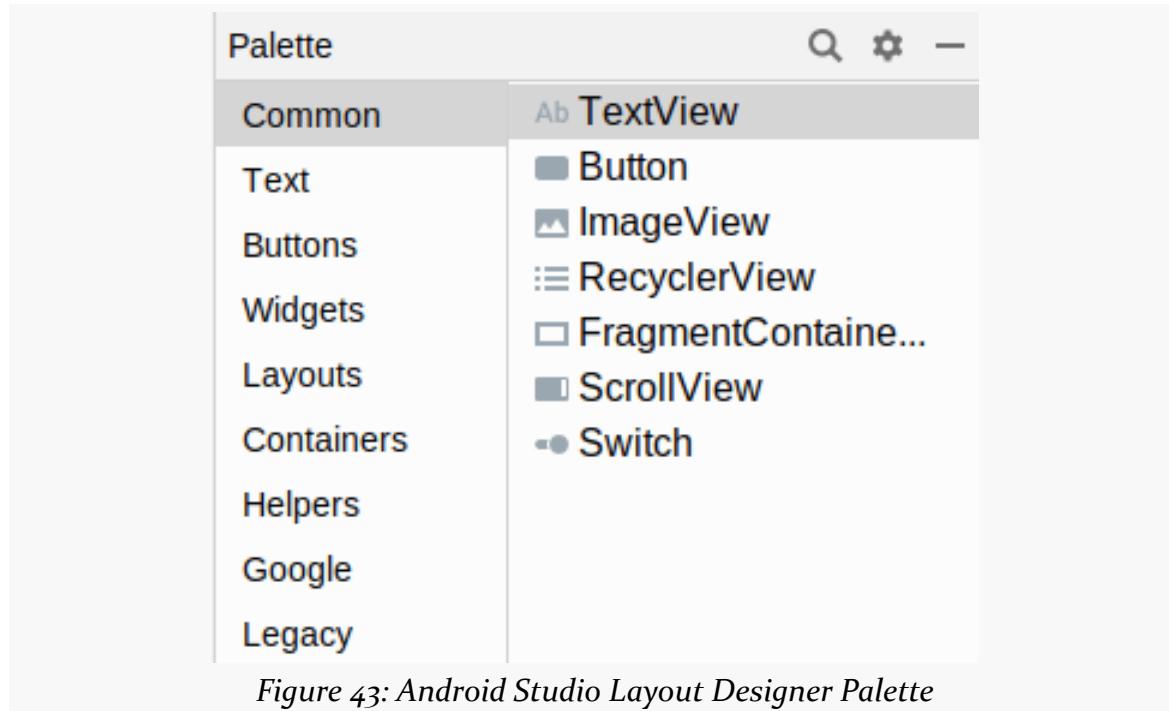


Figure 43: Android Studio Layout Designer Palette

CONSTRUCTING A LAYOUT

Drag a RecyclerView out of the “Palette” and drop it in the preview area:

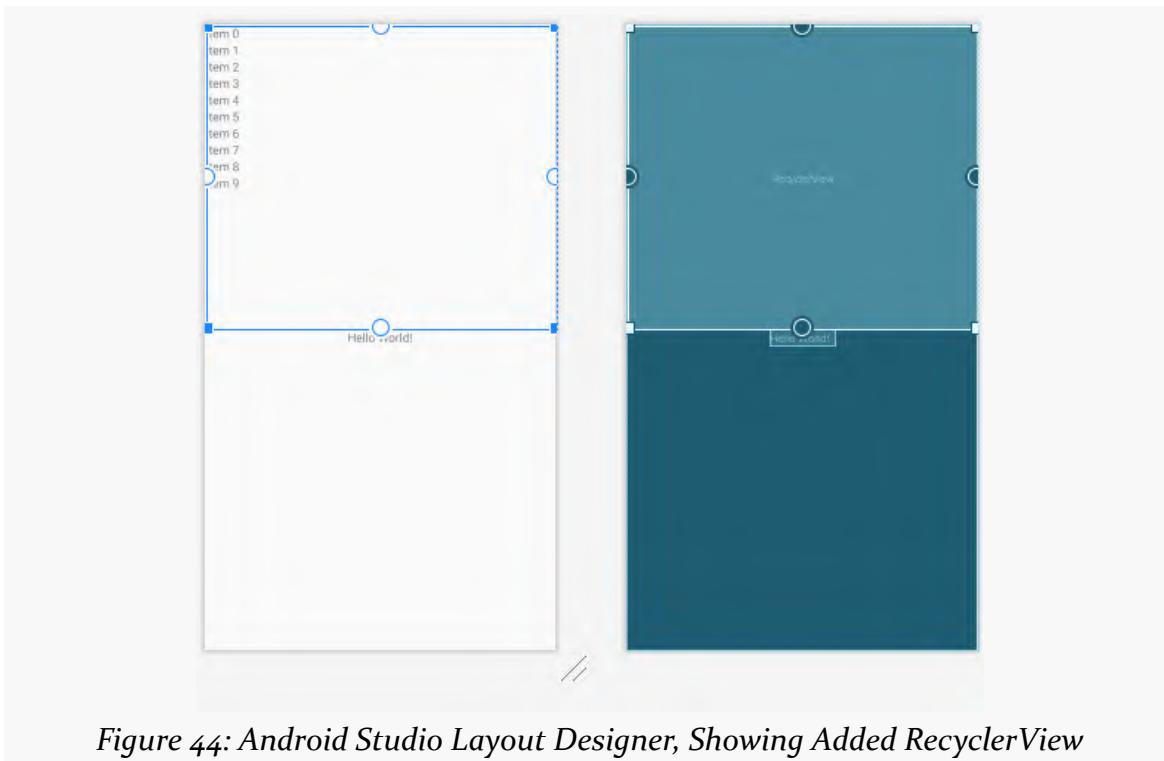


Figure 44: Android Studio Layout Designer, Showing Added RecyclerView

This will take up the top or bottom half of the layout, or possibly the full layout, or possibly just the middle of the layout, depending on where you drop it.

CONSTRUCTING A LAYOUT

Unfortunately, the Android Studio layout editor has many issues, including making the RecyclerView too big to manipulate. Grab a corner of the RecyclerView and drag it inwards to shrink it a bit. Then, drag the RecyclerView away from the edge a bit, to give you room to maneuver on all four sides:

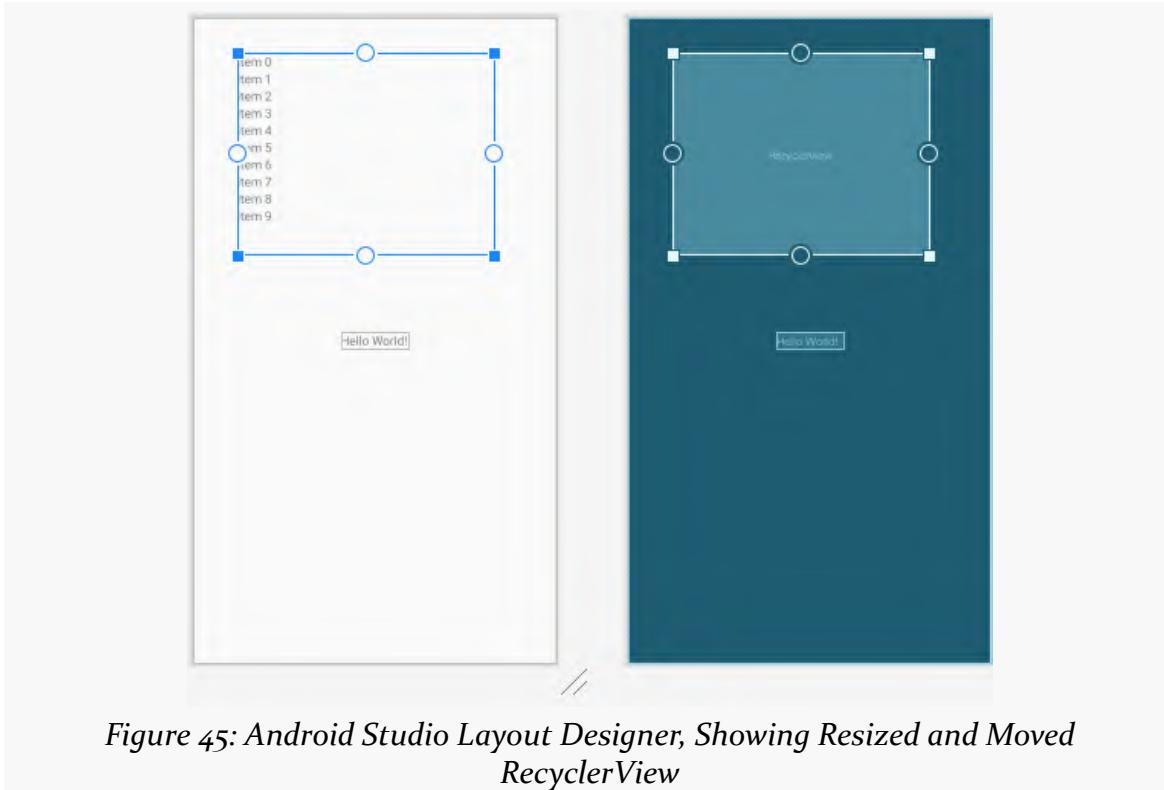


Figure 45: Android Studio Layout Designer, Showing Resized and Moved RecyclerView

CONSTRUCTING A LAYOUT

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:

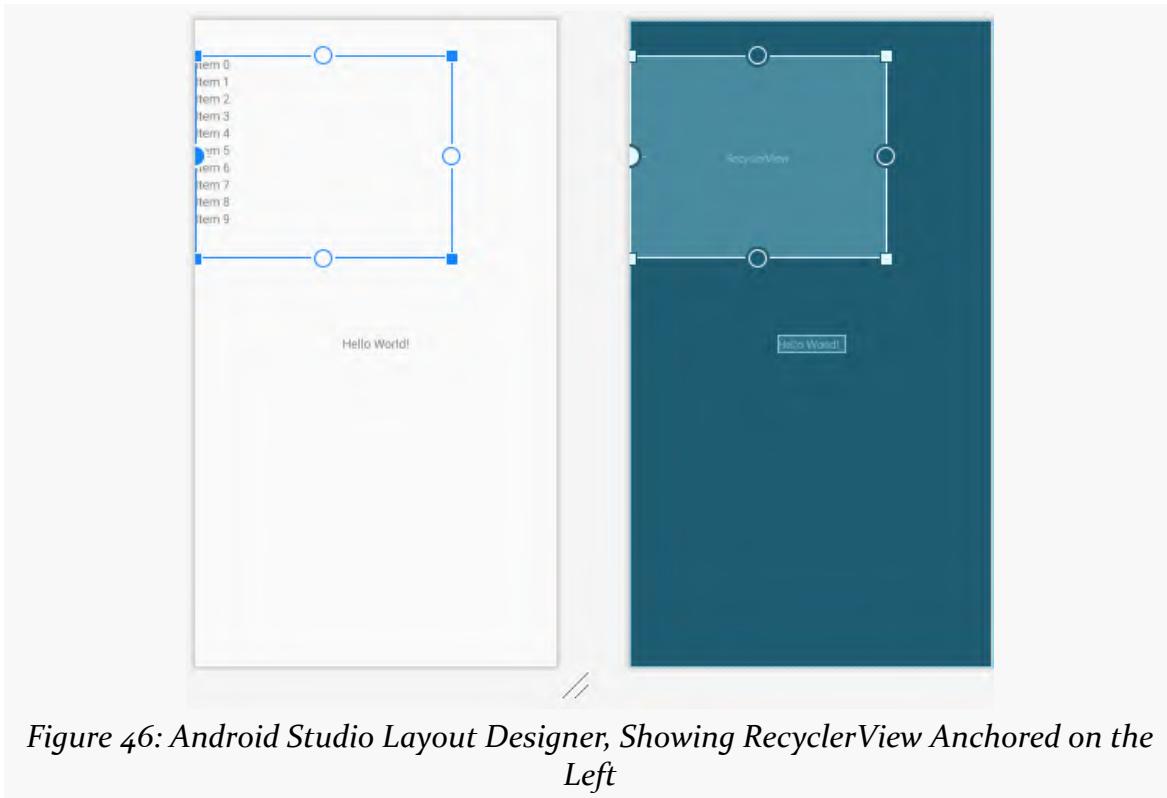


Figure 46: Android Studio Layout Designer, Showing RecyclerView Anchored on the Left

CONSTRUCTING A LAYOUT

Repeat that process on the right side:

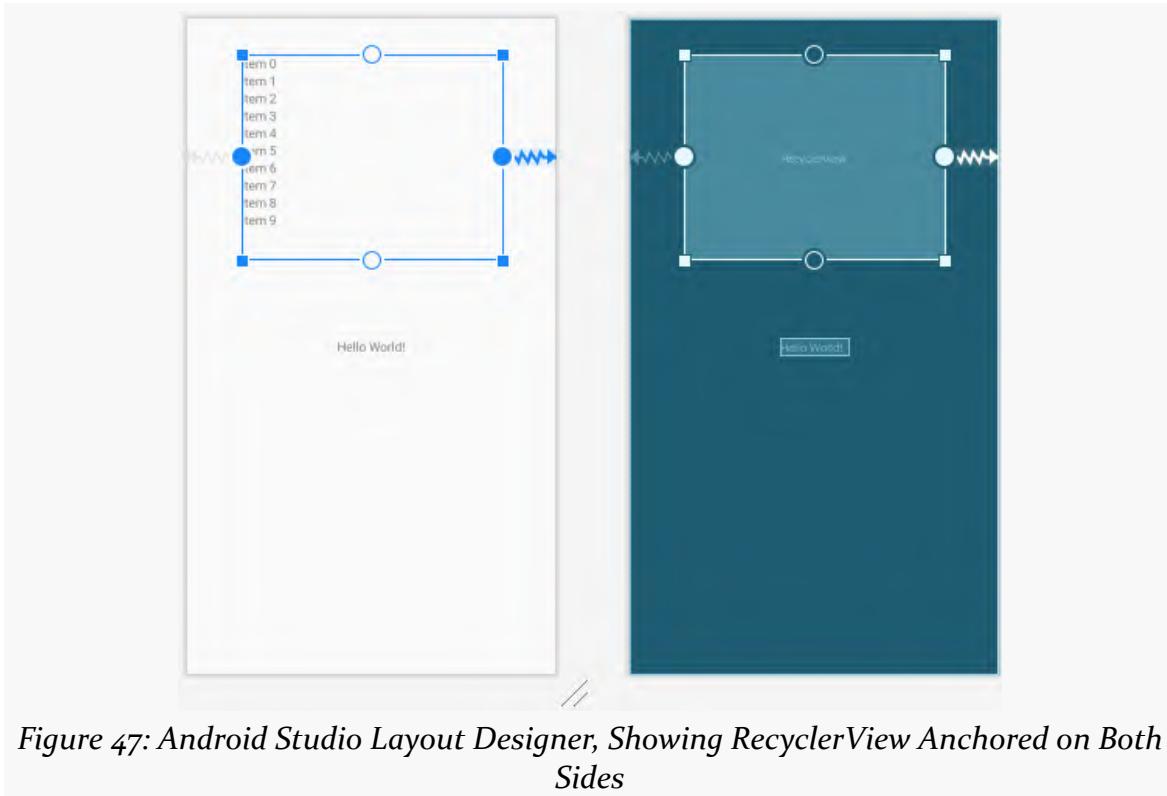


Figure 47: Android Studio Layout Designer, Showing RecyclerView Anchored on Both Sides

CONSTRUCTING A LAYOUT

Repeat that process on the top side:

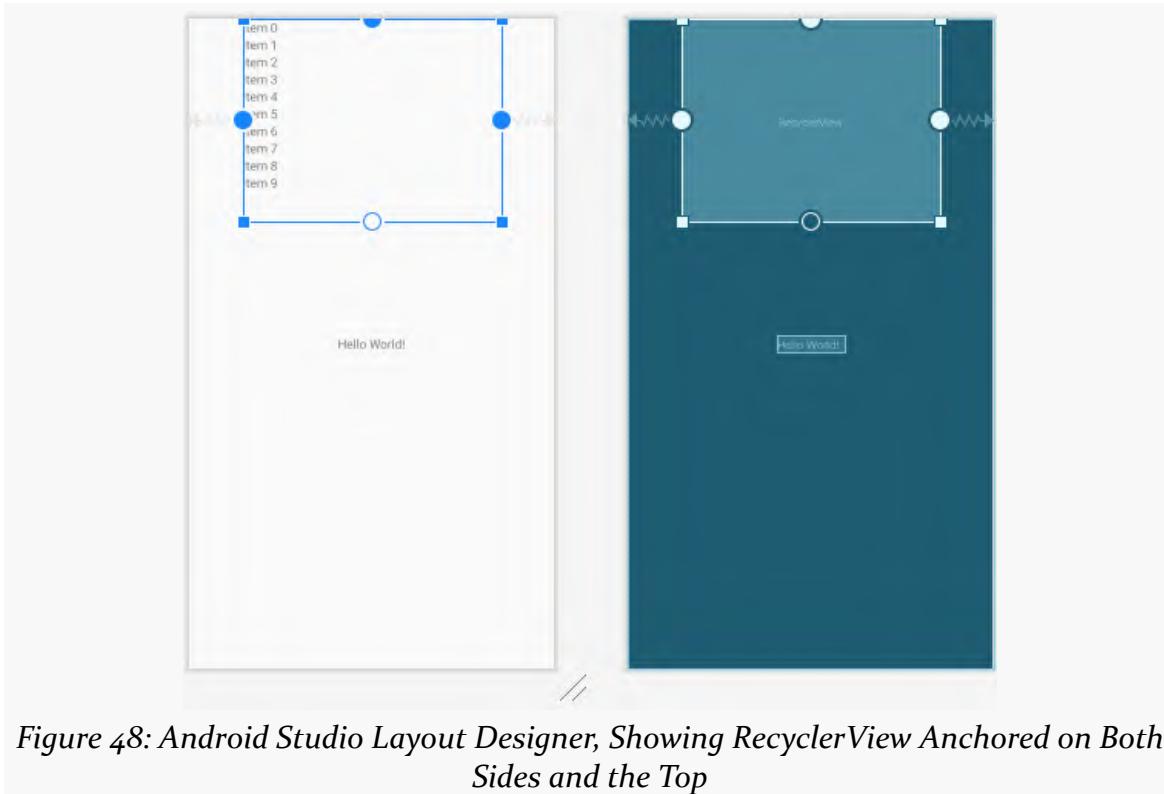
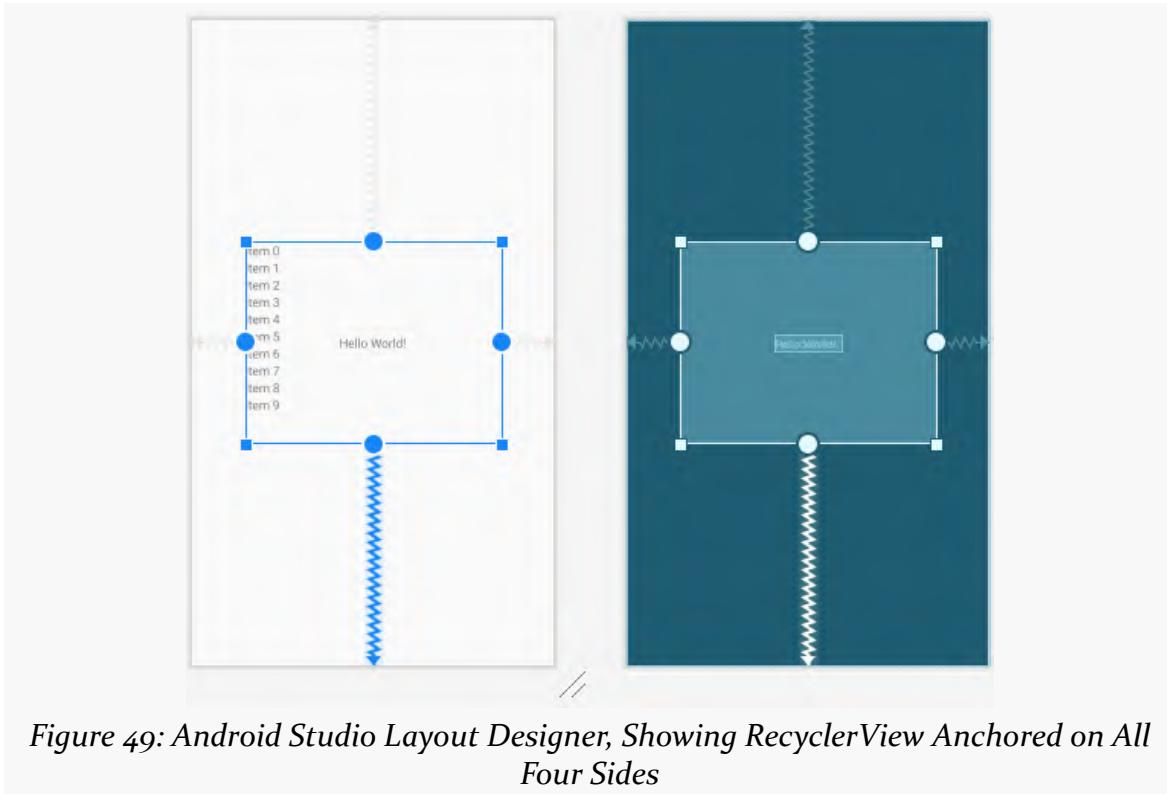


Figure 48: Android Studio Layout Designer, Showing RecyclerView Anchored on Both Sides and the Top

CONSTRUCTING A LAYOUT

Repeat that process on the bottom side:



CONSTRUCTING A LAYOUT

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`) from their current fixed values:

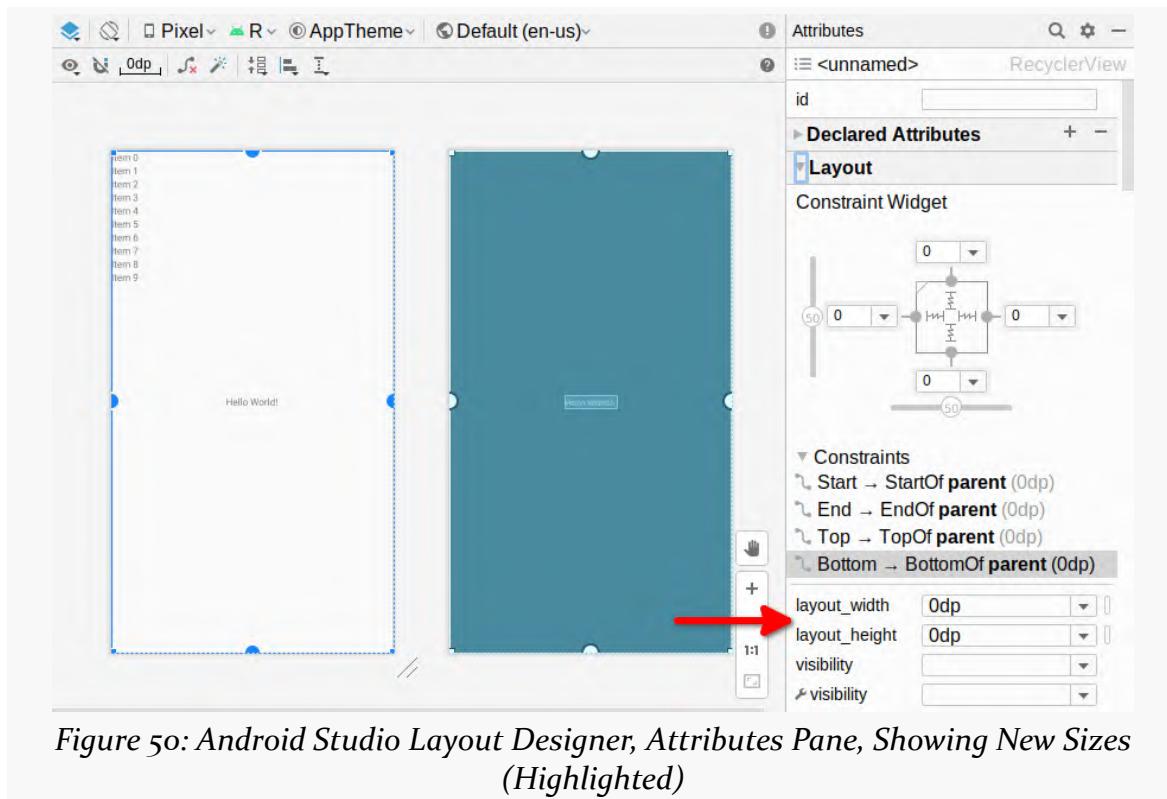


Figure 50: Android Studio Layout Designer, Attributes Pane, Showing New Sizes (Highlighted)

Now, you should see our `RecyclerView` fill the entire space. More importantly, we taught the `RecyclerView` to fill the entire space, no matter what the screen size is. Before, the `RecyclerView` would have some fixed size, regardless of whether the screen is larger or smaller than that size.

CONSTRUCTING A LAYOUT

Back in the “Attributes” pane, give the RecyclerView an ID of items, via the field at the top:

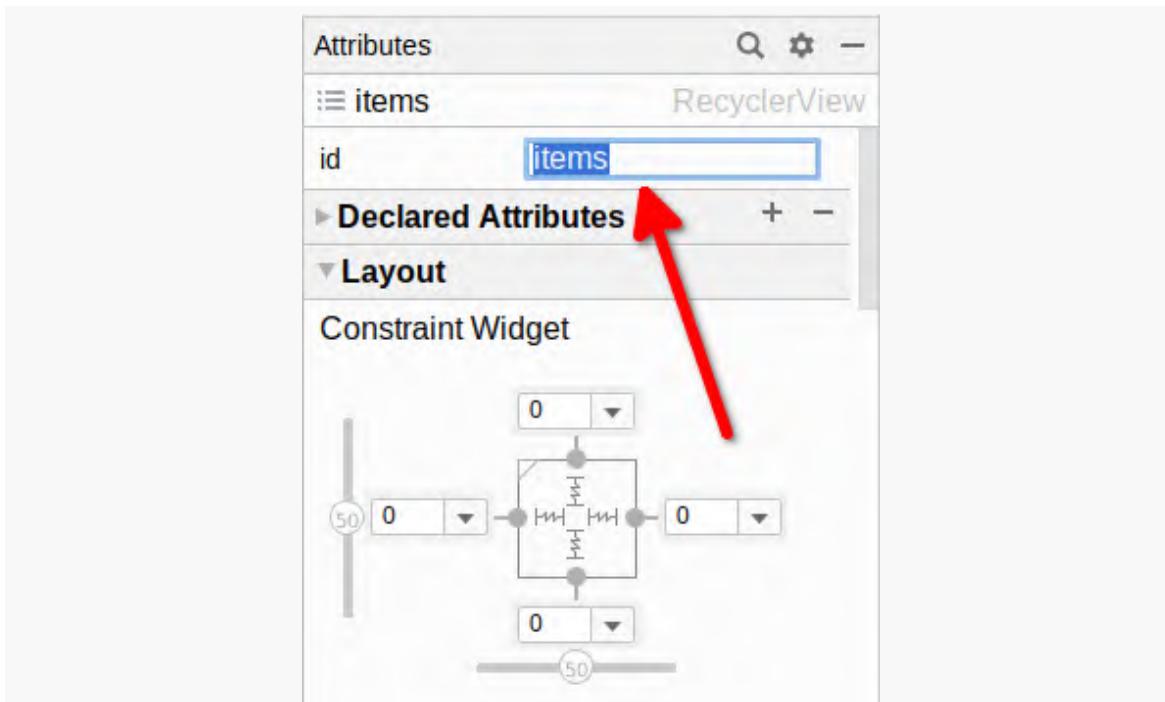


Figure 51: Android Studio Layout Designer, Attributes Pane, ID Highlighted

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:

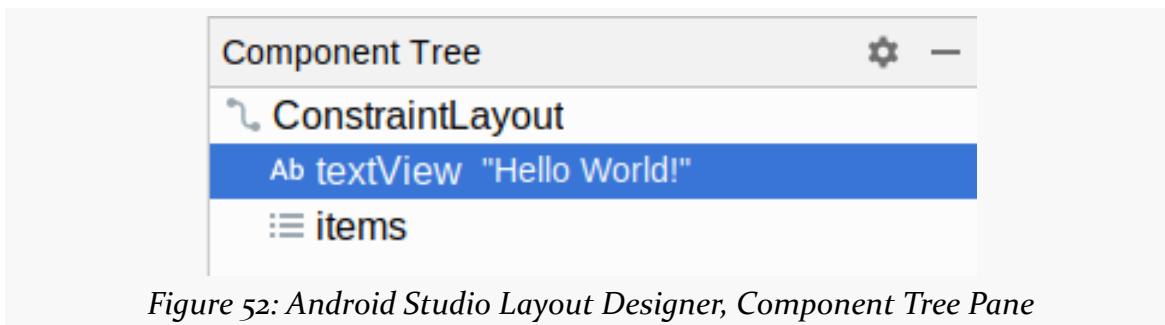


Figure 52: Android Studio Layout Designer, Component Tree Pane

CONSTRUCTING A LAYOUT

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

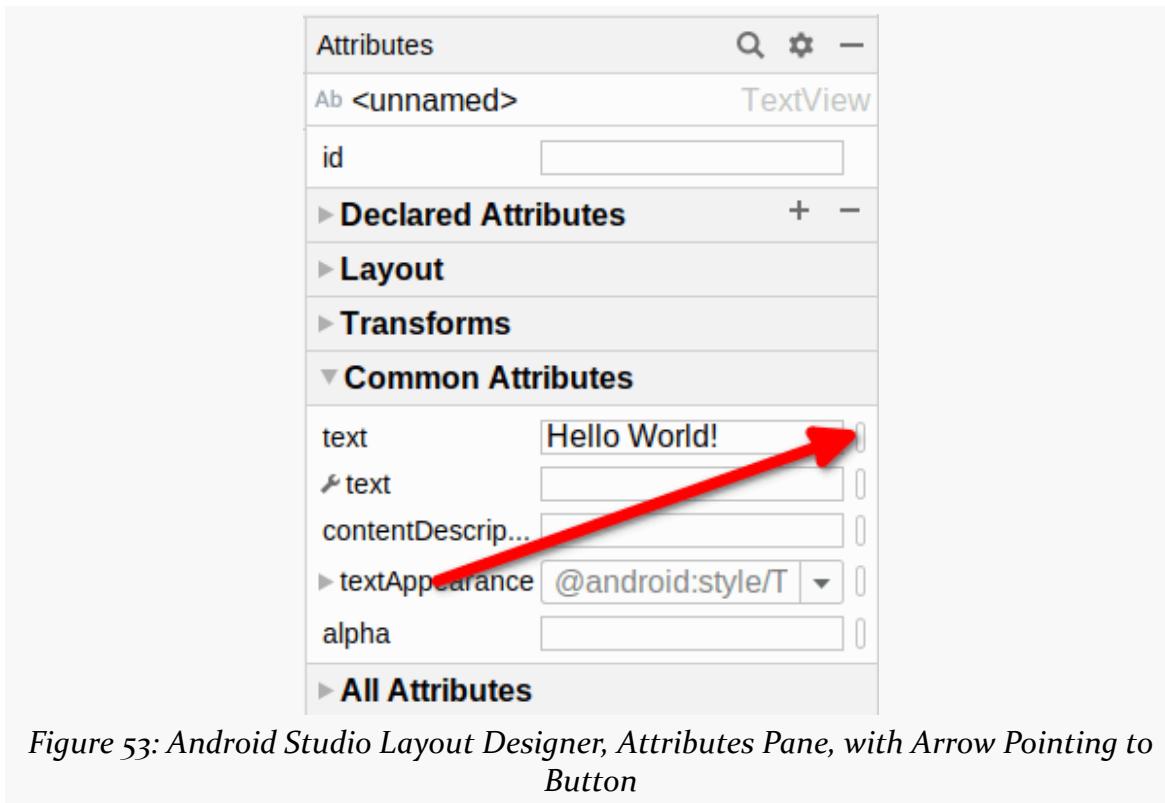


Figure 53: Android Studio Layout Designer, Attributes Pane, with Arrow Pointing to Button

CONSTRUCTING A LAYOUT

This will bring up a dialog showing available string resources:

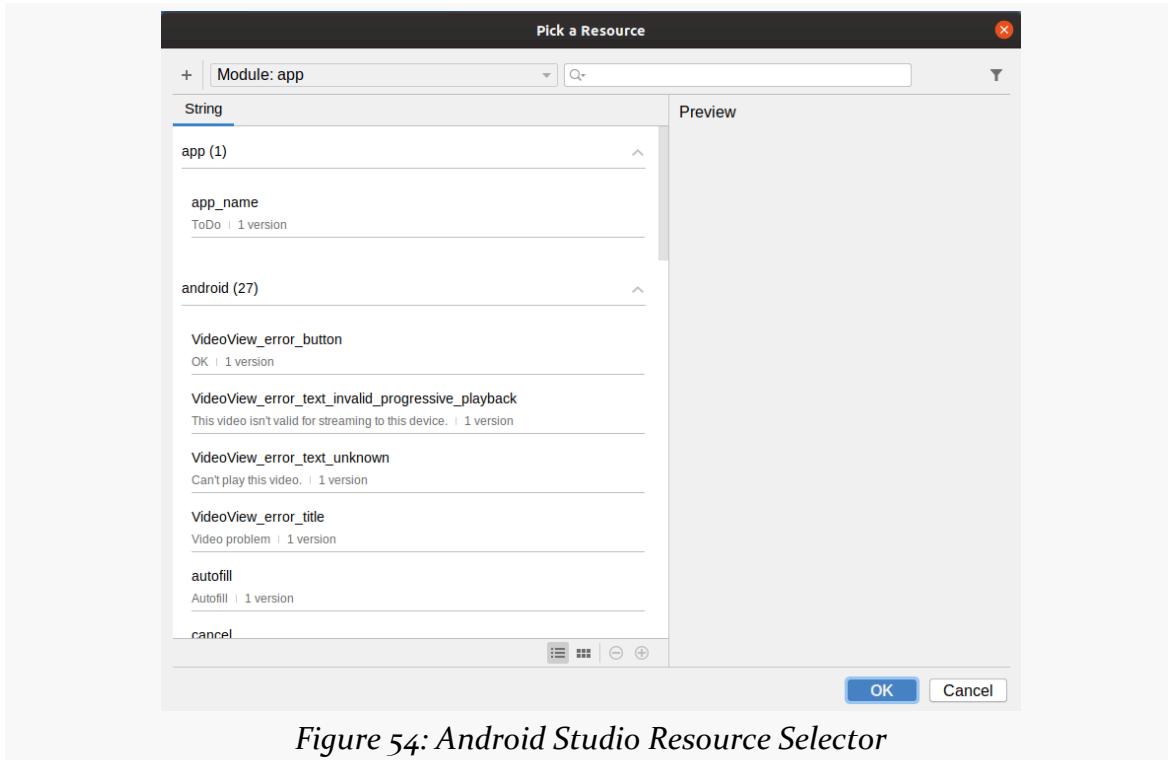


Figure 54: Android Studio Resource Selector

CONSTRUCTING A LAYOUT

Click the “+” icon to fold open a drop-down menu, and in there choose “String Value”. This brings up a dialog to define a new string resource:

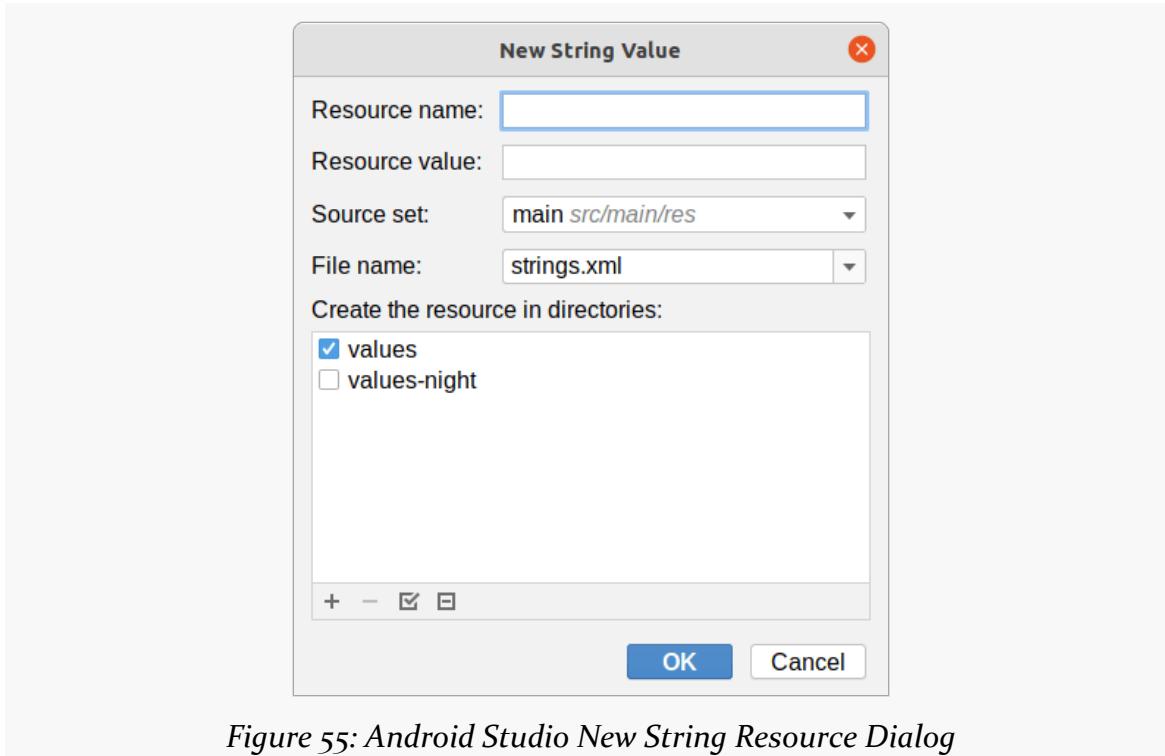


Figure 55: Android Studio New String Resource Dialog

CONSTRUCTING A LAYOUT

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

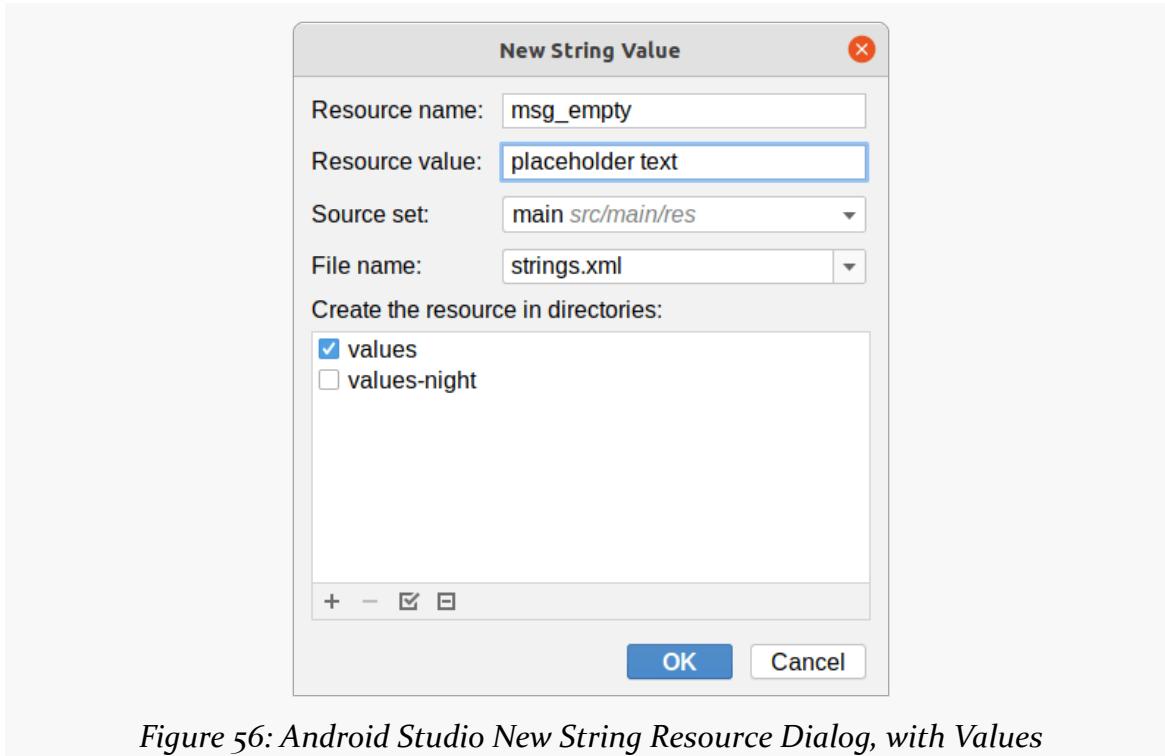


Figure 56: Android Studio New String Resource Dialog, with Values

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, then click “OK” to close the resource selector, and you should be taken back to the designer.

Then, in the “Common Attributes” section, fill in the following value for the “textAppearance”:

```
?android:attr/textAppearanceMedium
```

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

CONSTRUCTING A LAYOUT

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

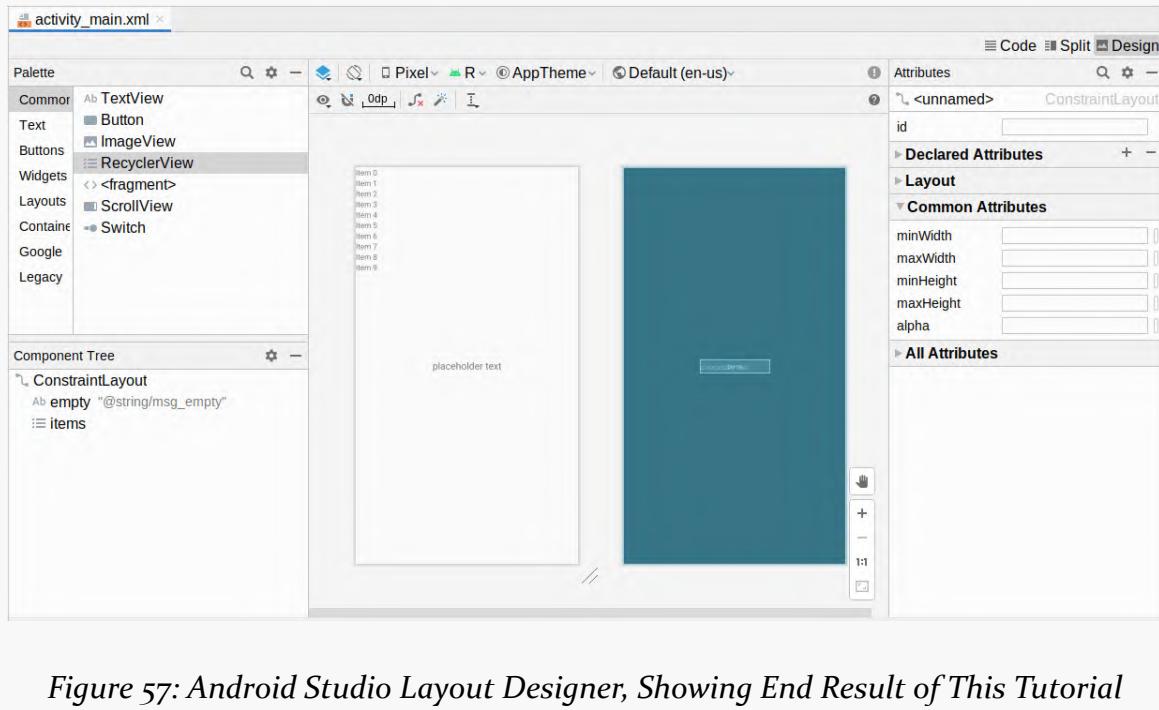


Figure 57: Android Studio Layout Designer, Showing End Result of This Tutorial

CONSTRUCTING A LAYOUT

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:

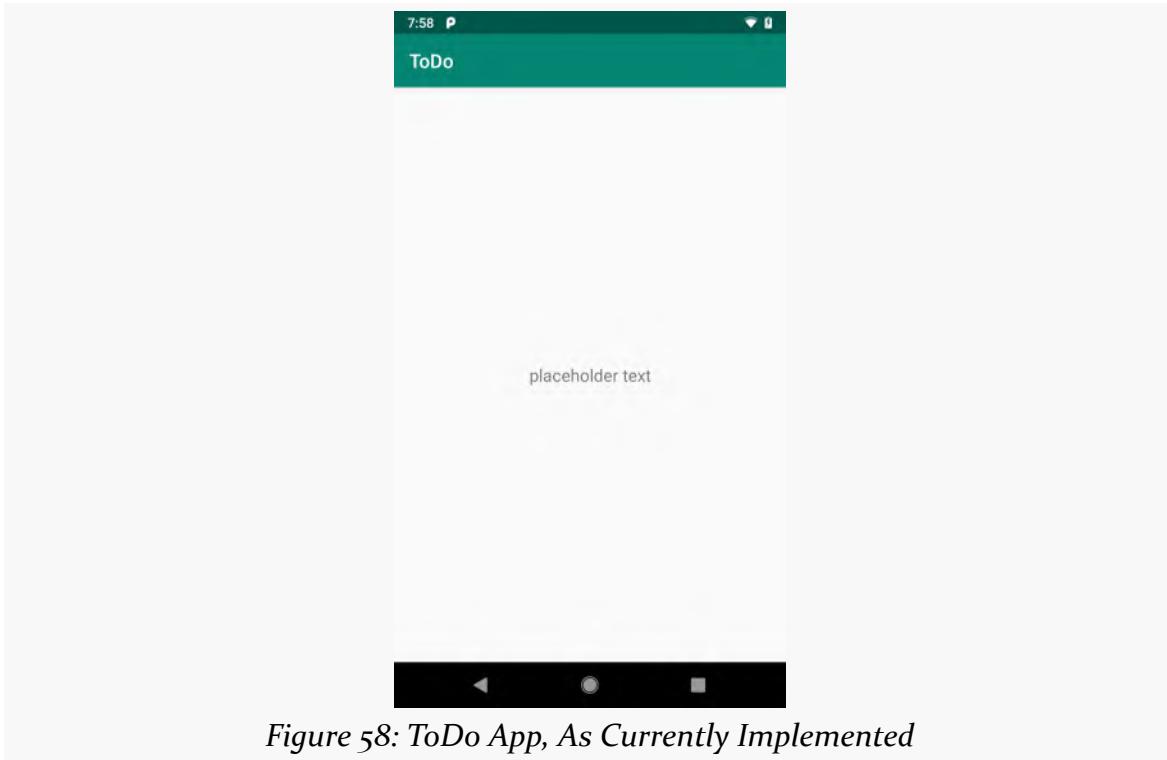


Figure 58: ToDo App, As Currently Implemented

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

Final Results

At this point, your `activity_main` layout resource XML should look something 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">

    <TextView
        android:id="@+id/empty"
```

CONSTRUCTING A LAYOUT

```
        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](#))

What We Changed

The book's GitLab 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](#)

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 will use fragments to implement each of those. This lines up with current recommended practices in Android development, and it gives us the flexibility to rearrange those bits of UI in varying situations (e.g., show both the list and one of the other fragments at the same time on larger-screen devices). In this chapter, we start setting up the first of these fragments, to show the list of to-do items.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab 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!*](#)

But First, Some Notes About Working with Kotlin

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

INTEGRATING FRAGMENTS

- **Alt-Enter** (**Option-Return** on macOS) will bring up context-aware “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.

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. The book’s preface has [a section with recommended PDF viewers](#). Reformatting the code with **Ctrl-Alt-L** (**Command-Option-L** on macOS) after pasting it in sometimes helps.

Also, you may find it useful to have the IDE supply “hints” about the types it thinks that variables and function return values resolve to. Kotlin does not require you to enter all of the type information, as the compiler can infer types in many places. However, sometimes that makes it difficult to identify where things are going wrong. Enabling type hints allows the IDE to tell you the inferred types, without you having to enter those types yourself:

```
val binding : ActivityMainBinding = ActivityMainBinding.inflate(layoutInflater)
```

Figure 59: Android Studio Kotlin Editor, Showing Variable Type Hint

INTEGRATING FRAGMENTS

To toggle this on, go into the Settings dialog (“File” > “Settings” in Linux and Windows, and in “Android Studio” > “Preferences...” on macOS). Drill down into “Editor” > “Inlay Hints” > “Kotlin” in the category tree on the left, then check the hints that you want in the various lists:

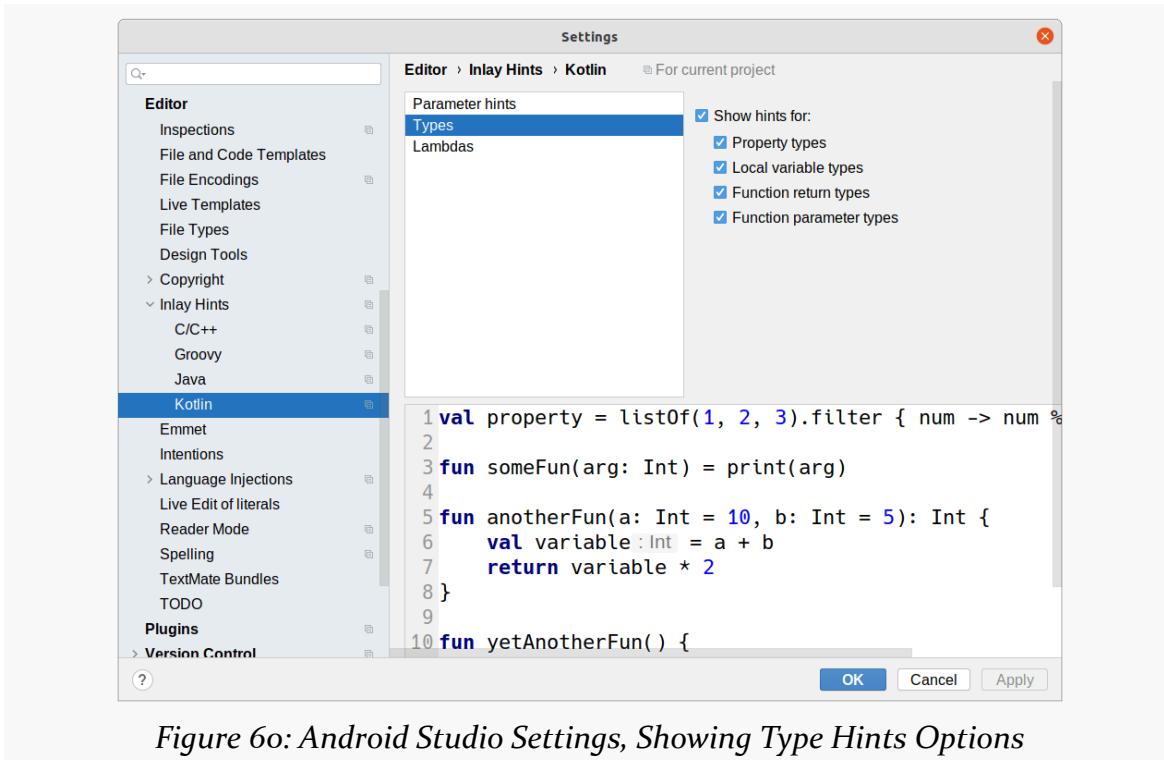


Figure 6o: Android Studio Settings, Showing Type Hints Options

And if you see [a “Code Vision” checkbox with no caption](#), that is a bug.

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 Kotlin class.

INTEGRATING FRAGMENTS

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

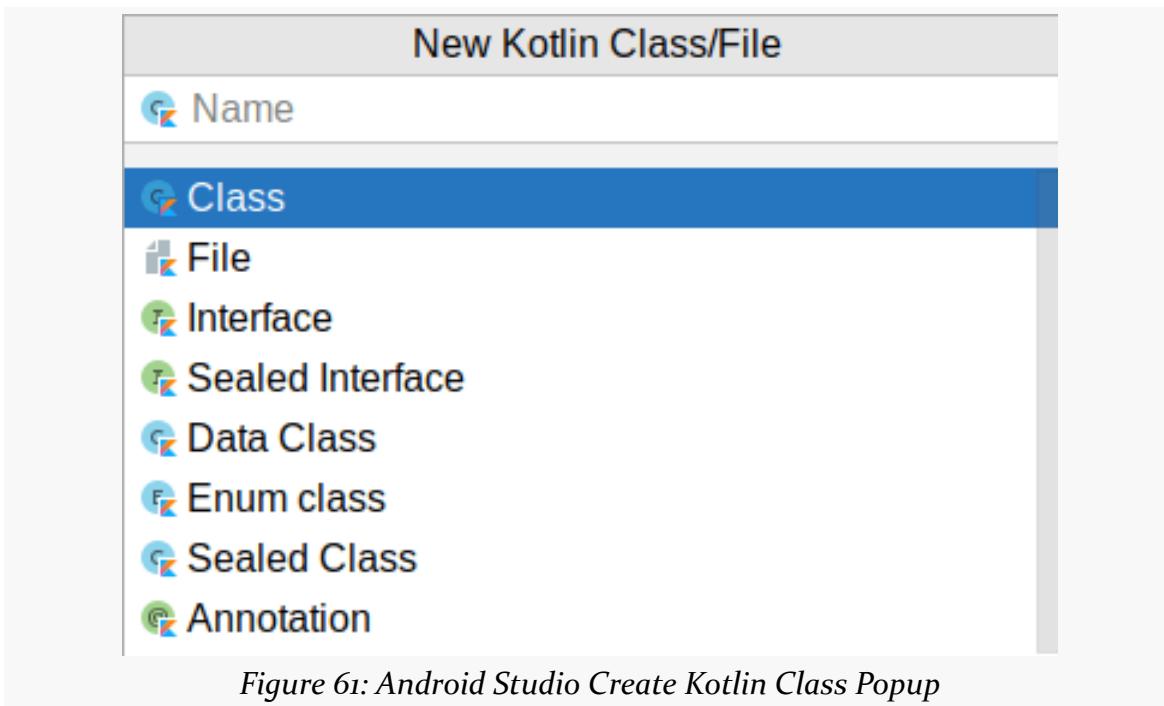


Figure 61: Android Studio Create Kotlin Class Popup

For the name, fill in `RosterListFragment`, as this fragment is showing a list of our to-do items. Choose “Class” in the list of Kotlin structures below the field. Then, press `Enter` or `Return` to create the class. That will give you a `RosterListFragment` that looks like:

```
package com.commonware.todo

class RosterListFragment {
```

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

```
package com.commonware.todo

import androidx.fragment.app.Fragment

class RosterListFragment : Fragment() {
```

Step #2: Renaming Our Layout

We want to show the layout resource that we tweaked in [the preceding chapter](#). However, our layout resource is called `activity_main`, and we want to use it from a fragment, not an activity.

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:

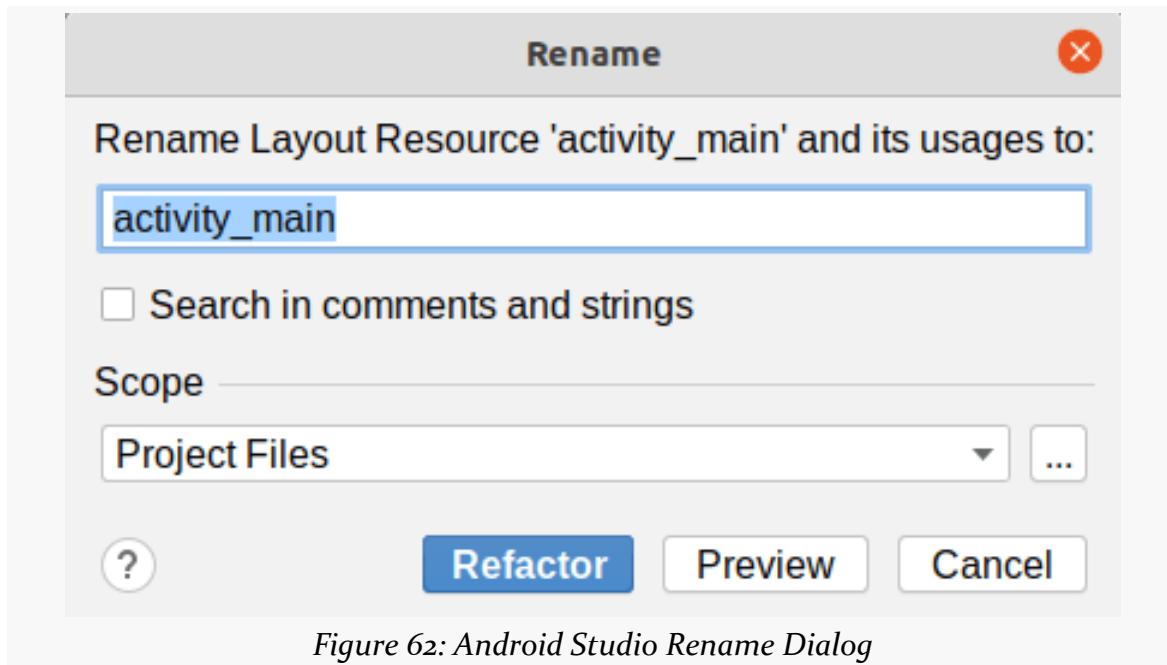


Figure 62: Android Studio Rename Dialog

INTEGRATING FRAGMENTS

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



Figure 63: 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. If it does show up, click the “Do Refactor” button towards the bottom of the “Refactoring Preview” view.

Step #3: Inflating Our Layout

Right now, 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` to bring up a list of methods that we could override:

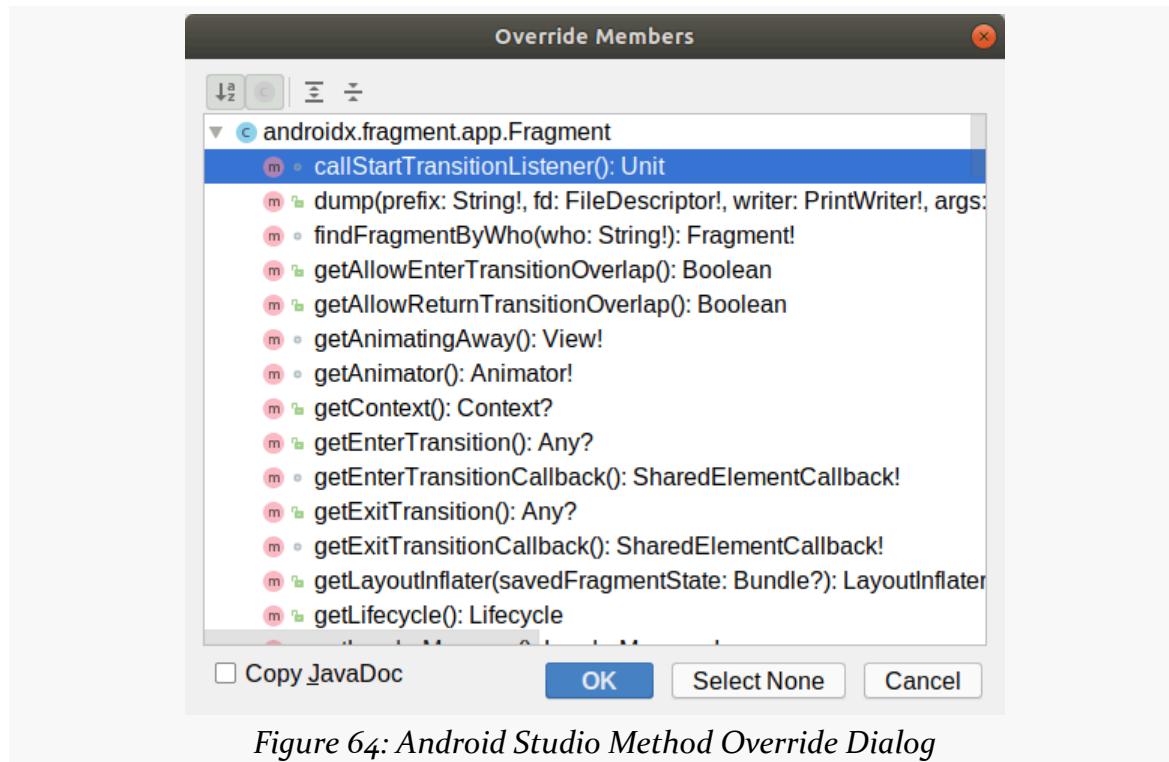


Figure 64: Android Studio Method Override Dialog

INTEGRATING FRAGMENTS

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:

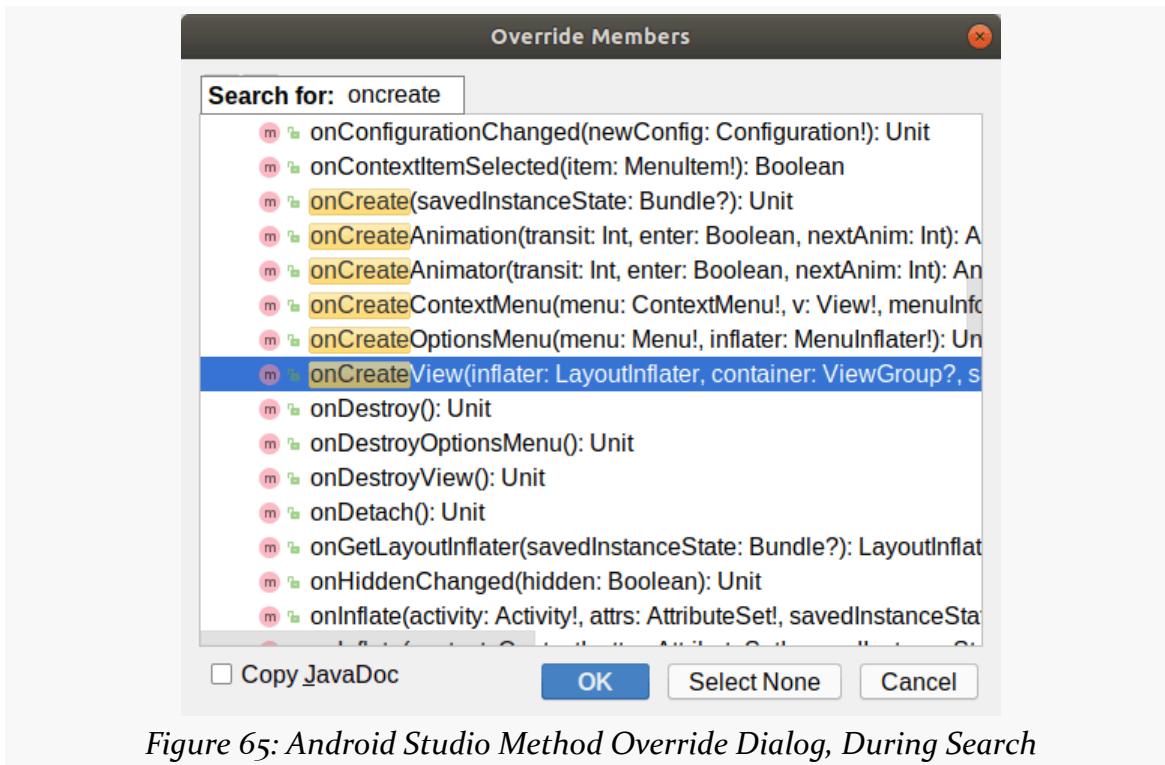


Figure 65: Android Studio Method Override Dialog, During Search

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? {
```

INTEGRATING FRAGMENTS

```
    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? {
        return inflater.inflate(R.layout.todo_roster, 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.todo_roster`
- 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.

Note that at this point you cannot see this fragment. We need to take some steps to have `MainActivity` display it, which we will handle in [the next tutorial](#).

Step #4: 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 66: 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:

INTEGRATING FRAGMENTS

- 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

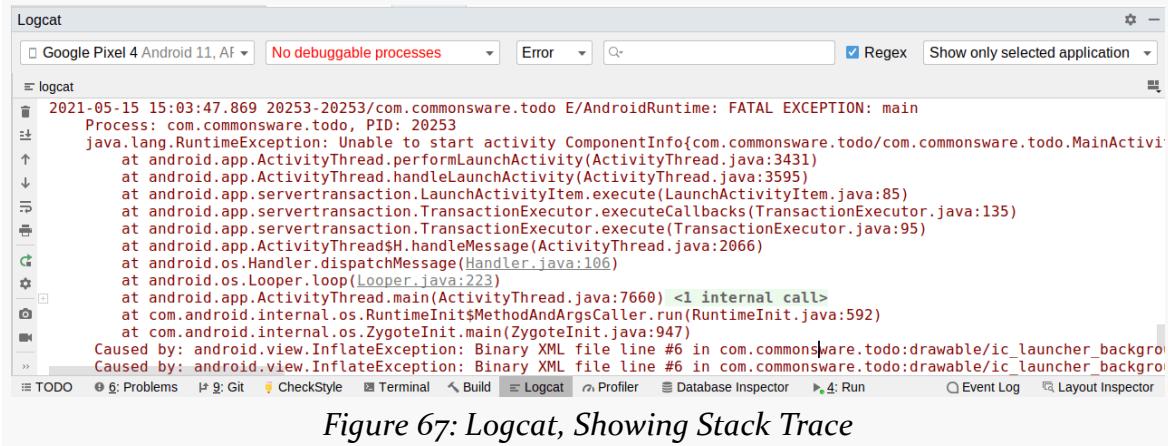


Figure 67: Logcat, Showing Stack Trace

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.LayoutInflater.inflate(LayoutInflater.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)
```

INTEGRATING FRAGMENTS

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.

Final Results

Your `RosterListFragment` should 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? {
        return inflater.inflate(R.layout.todo_roster, container, false)
    }
}
```

(from [ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/res/layout/todo_roster.xml](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)

Wiring In Navigation

In [the last tutorial](#), we created a fragment, but we did not display it. There are three main ways we have of displaying a fragment:

- Use a <fragment> element in a layout resource. This is for fragments that we will be showing all the time.
- Use a FragmentTransaction to tell a FragmentManager to display a fragment in a specified container.
- Use the Navigation component to abstract away requests to navigate to a particular screen from the implementation of that screen.

In this tutorial, we will look at the third of those options.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab 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 Navigation component in the "Navigating Your App" chapter of [Elements of Android Jetpack](#)!

Step #1: Defining the Version

We are going to have several dependencies entries tied to the Navigation component. These will have synchronized version numbers, and we will want to use the same version number for each dependency. So, it is best to define the version number as a constant, so we can refer to that constant everywhere we need the version number. Then, when the version number changes, we can change it in one

WIRING IN NAVIGATION

place and have it update all the necessary lines automatically.

If you open up the top-level build.gradle file — the one in the root of your project — it should resemble this:

```
buildscript {  
  
    repositories {  
        google()  
        mavenCentral()  
    }  
  
    dependencies {  
        classpath "com.android.tools.build:gradle:7.0.2"  
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"  
    }  
}  
  
task clean(type: Delete) {  
    delete rootProject.buildDir  
}
```

(from [To7-Fragments/ToDo/build.gradle](#))

Just after the opening buildscript { line, add:

```
ext.nav_version = '2.3.5'
```

(from [To8-Nav/ToDo/build.gradle](#))

This sets up a constant that we can use in our Gradle builds files. Specifically, we are going to use a particular version of the Navigation component, and this line sets up that version number.

After making this change, you should get a banner suggesting that you “Sync Now” due to your Gradle changes. Ignore it for now, as we have more changes to make.

Step #2: Adding the Plugin Dependency

In that same buildscript closure, you will see a list of dependencies:

```
dependencies {  
    classpath "com.android.tools.build:gradle:7.0.2"  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"  
}
```

WIRING IN NAVIGATION

(from [To7-Fragments/ToDo/build.gradle](#))

These represent sources of Gradle plugins, for helping us do more interesting things when we build our app.

Part of the Navigation component is a plugin, so we need to add another dependency to the buildscript roster. So, add this line to that dependencies closure:

```
classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
```

(from [To8-Nav/ToDo/build.gradle](#))

This pulls in the androidx.navigation:navigation-safe-args-gradle-plugin artifact, for the version number that we specified. We use string interpolation to add our nav_version value into the dependency, which is why this string uses double-quotes; in Gradle (and the Groovy language it is built upon), a single-quoted string cannot use string interpolation.

The banner should still be there, asking you to “Sync Now”. Continue to hold off, as we need to make changes to another Gradle file.

Step #3: Requesting the Plugins

Just because we added a plugin artifact does not mean that we actually use the plugin. We need an additional line to say where we want that plugin to take effect.

That line goes in the app/build.gradle file, representing build instructions for the app module. We already have a plugins closure listing a pair of plugins:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'
```

(from [To7-Fragments/ToDo/app/build.gradle](#))

Add one more to that list:

```
    id 'androidx.navigation.safeargs.kotlin'
```

(from [To8-Nav/ToDo/app/build.gradle](#))

The androidx.navigation.safeargs.kotlin plugin is for “Safe Args”, a feature of the Navigation component that helps us pass data from one screen to another.

WIRING IN NAVIGATION

You will be tempted by the banner asking you to “Sync Now”. Do not give into the temptation.

(or, go ahead and click “Sync Now” if you *really* want to, though we have more changes to make)

Step #4: Augmenting Our Dependencies

The line we added to dependencies in the top-level `build.gradle` file defined an artifact that contributes compile-time code, in the form of this Gradle plugin. We also need to add dependencies for runtime code, just as we have for things like `RecyclerView`.

So, in `app/build.gradle`, in its `dependencies` closure, add these lines:

```
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

(from [To8-Nav/ToDo/app/build.gradle](#))

The `androidx.navigation:navigation-fragment-ktx` artifact contains the core code for using the Navigation component to navigate between fragments. The `androidx.navigation:navigation-ui-ktx` contains a bit of additional code for integrating navigation with the Toolbar.

You may now go ahead and click the “Sync Now” link in the banner. Conversely, if for some reason that banner did not appear, choose “File” > “Sync Project with Gradle Files” in the Android Studio main menu.

Step #5: Defining Our Navigation Graph

The Navigation component uses navigation resources to define navigation graphs. A navigation graph simply states what screens there are, how they are implemented (e.g., as fragments), and how they are connected. A navigation graph is stored in a XML file as a navigation resource, in a `res/navigation/` directory in your module.

We need to create a stub navigation graph for our app to begin using the Navigation component.

With that in mind, right-click over the `res/` directory in your module and choose “New” > “Android resource file” from the context menu. This will bring up a dialog

WIRING IN NAVIGATION

that allows you to define a resource type and file in one shot:

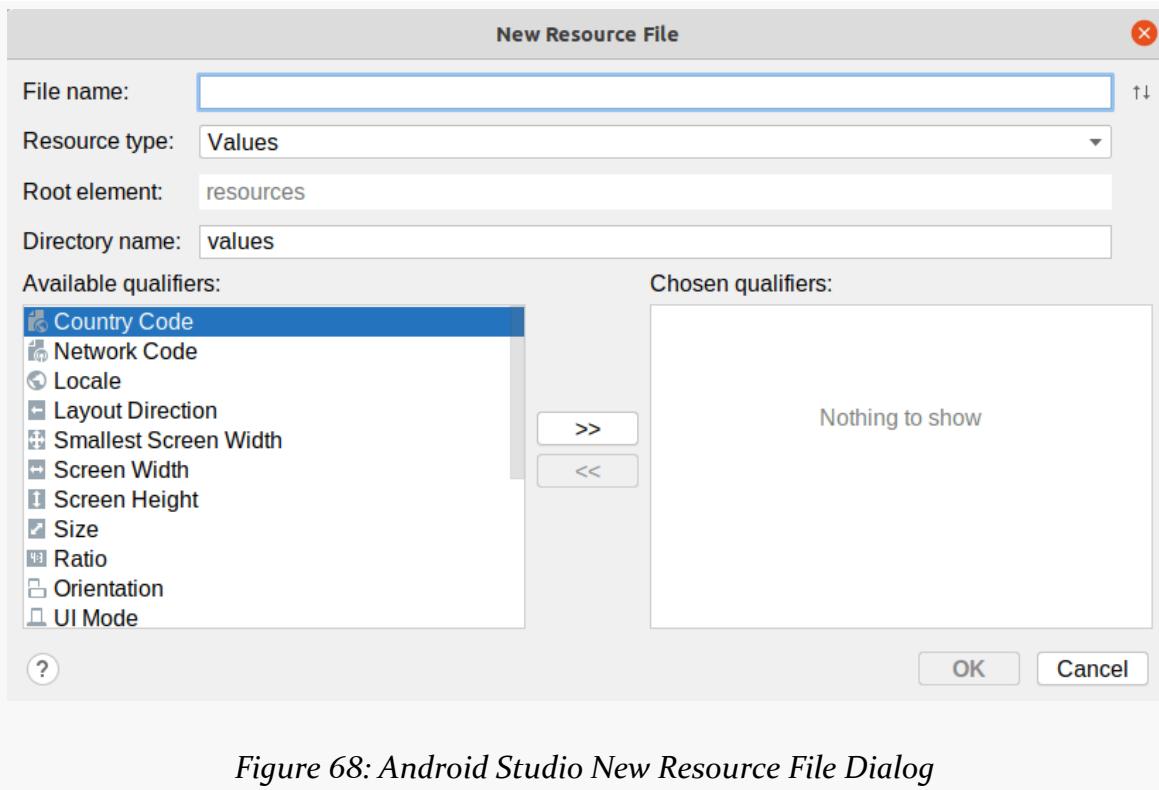


Figure 68: Android Studio New Resource File Dialog

Fill in `nav_graph.xml` for the filename. Choose “Navigation” for the “Resource Type”. Then, click OK to create a `res/navigation/` directory and a `nav_graph.xml` file in it.

As with layout and menu resources, the editor for navigation resources contains multiple views, controlled by toolbar buttons. The two main views are the one that shows the raw XML (“Code”) and the one that shows a graphical navigation designer (“Design”).

If you click on “Code” to view the raw XML, you will see that our XML is pretty empty:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph">

</navigation>
```

WIRING IN NAVIGATION

To add RosterListFragment as our one-and-only screen, click the “Design” button to switch to the graphical designer view. There, click the toolbar button that looks like a document with a green + sign in the lower-right corner:



Figure 69: Android Studio Navigation Resource Editor Toolbar

This will drop down a list of possible “destinations”, and RosterListFragment will be among them:

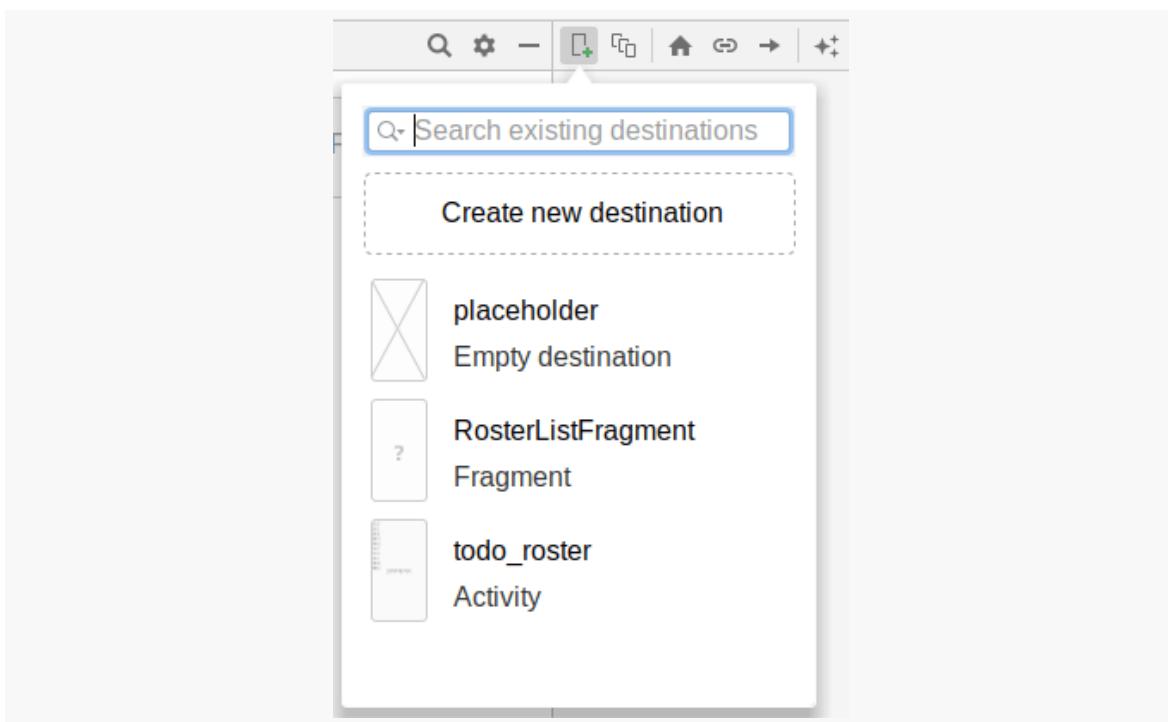


Figure 70: Android Studio Navigation Resource Editor Candidate Destinations

WIRING IN NAVIGATION

Click on RosterListFragment in the drop-down list. That will add it as a destination to our navigation graph:

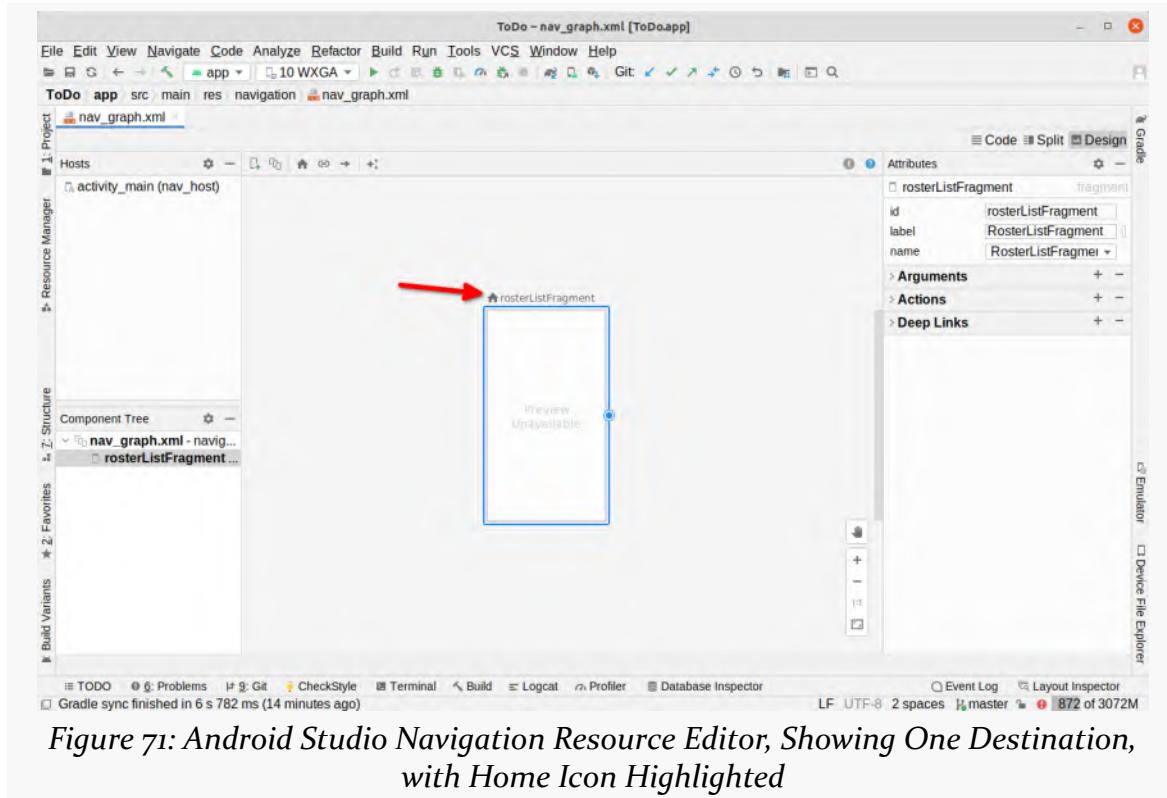


Figure 71: Android Studio Navigation Resource Editor, Showing One Destination, with Home Icon Highlighted

The little house icon above the preview rectangle marks this destination as the “home”. It is where this navigation graph will start, when we begin using it to navigate screens in our app.

We will have several changes to make to this in later tutorials, but this will suffice for now.

Step #6: Setting Up a New Activity Layout Resource

To navigate between fragments, the Navigation component uses one fragment as the “host”. That fragment, in turn, will hold the fragments representing individual screens.

Earlier in the book, we had an `activity_main` layout resource, but we renamed it to

WIRING IN NAVIGATION

todo_roster when we converted the app to use fragments. Now, we need a layout resource for our `MainActivity` again, where we can set up the host fragment.

So, right-click over the `res/layout/` directory and choose “New” > “Layout resource file” from the context menu. Fill in `activity_main` for the filename and use `ConstraintLayout` for the “Root element” (if you start typing in that name, it will show up in a selection list for you to choose from). Click “OK” to create the mostly-empty layout resource.

We want to add a `<FragmentContainerView>` element to the layout resource. As the name suggests, this is a container for a fragment and will show that fragment wherever we size and position it. An `android:name` attribute will indicate what fragment class we want.

While the drag-and-drop GUI builder offers `FragmentContainerView`, not all widgets and containers that we want to use will be available for drag-and-drop. So, in this case, we will add this element by hand, directly in the XML.

Click on the “Code” button to switch to the XML editor view. There, add in this XML element as a child of the `ConstraintLayout` element:

```
<androidx.fragment.app.FragmentContainerView  
    android:id="@+id/nav_host"  
    android:name="androidx.navigation.fragment.NavHostFragment"  
    android:layout_width="0dp"  
    android:layout_height="0dp"  
    app:defaultNavHost="true"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    app:navGraph="@navigation/nav_graph" />
```

(from [To8-Nav/ToDo/app/src/main/res/layout/activity_main.xml](#))

This fragment will be part of our UI for as long as we are using this layout. The actual fragment implementation is `androidx.navigation.fragment.NavHostFragment`, which is a fragment from the Navigation component that knows how to switch between screens defined in a navigation resource. That navigation resource is identified via the `app:navGraph` attribute, in this case pointing to our `nav_graph` that we defined. The fragment also has `app:defaultNavHost="true"`, which tells the Navigation component that this fragment is the one responsible for that navigation graph.

WIRING IN NAVIGATION

You may find that the app namespace shows up in red:

```
<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:navGraph="@navigation/nav_graph" />
```

Figure 72: Android Studio Layout XML Editor, Yelling About app Namespace

app is used as a namespace prefix for a lot of attributes used by widgets and containers that we get from libraries. To add the definition of this namespace, with the text cursor in one of those app prefixes, press **Alt-Enter** (**Option-Return** on macOS) and choose “Create namespace declaration” from the quick-fix menu.

Step #7: Wiring in the Navigation

We need to switch `MainActivity` to use this re-created `activity_main` layout resource. So, change the `onCreate()` function in `MainActivity` to be:

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

(from [To8-Nav/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

WIRING IN NAVIGATION

If you now run the app, it should give you the same result as before we added the fragment:

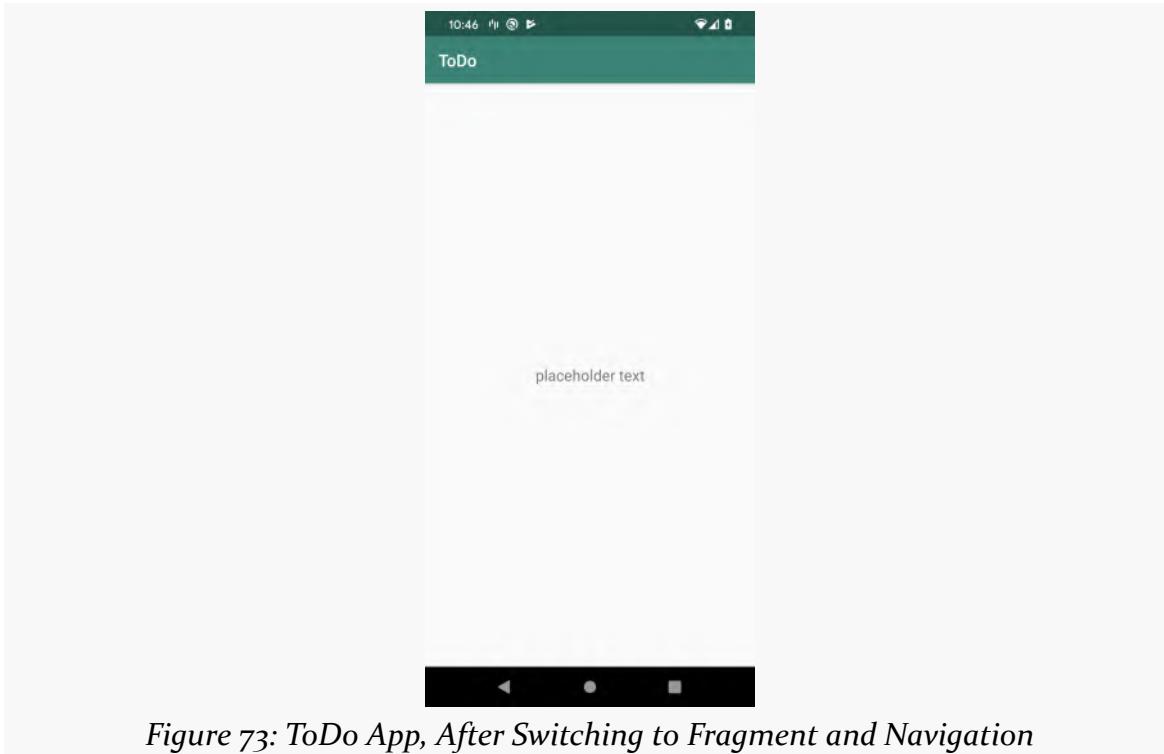


Figure 73: ToDo App, After Switching to Fragment and Navigation

When we add new screens in upcoming tutorials, we will:

- Create fragments for those screens
- Add those as destinations in our navigation graph, connecting them with previous screens to indicate how we move from one to the next
- Add some Kotlin code to say “let’s navigate from where we are to this destination”

And the rest will be taken care of by the Navigation component.

Final Results

Your overall top-level build.gradle should now resemble:

```
buildscript {  
    ext.nav_version = '2.3.5'
```

WIRING IN NAVIGATION

```
repositories {  
    google()  
    mavenCentral()  
}  
  
dependencies {  
    classpath "com.android.tools.build:gradle:7.0.2"  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"  
    classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"  
}  
}  
  
task clean(type: Delete) {  
    delete rootProject.buildDir  
}
```

(from [To8-Nav/ToDo/build.gradle](#))

The overall app/build.gradle file should now resemble:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
    id 'androidx.navigation.safeargs.kotlin'  
}  
  
android {  
    compileSdk 31  
  
    defaultConfig {  
        applicationId "com.commonsware.todo"  
        minSdk 21  
        targetSdk 31  
        versionCode 1  
        versionName "1.0"  
  
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    }  
  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),  
            'proguard-rules.pro'  
        }  
    }  
  
    compileOptions {
```

WIRING IN NAVIGATION

```
sourceCompatibility JavaVersion.VERSION_1_8
targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

(from [To8-Nav/ToDo/app/build.gradle](#))

The navigation resource XML (`res/navigation/nav_graph.xml`) now should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph.xml"
    app:startDestination="@+id/rosterListFragment">

    <fragment
        android:id="@+id/rosterListFragment"
        android:name="com.commonsware.todo.RosterListFragment"
        android:label="RosterListFragment" />
</navigation>
```

(from [To8-Nav/ToDo/app/src/main/res/navigation/nav_graph.xml](#))

The `res/layout/activity_main.xml` resource should 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"
    android:layout_width="match_parent"
```

WIRING IN NAVIGATION

```
        android:layout_height="match_parent">>

<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:navGraph="@navigation/nav_graph" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [To8-Nav/ToDo/app/src/main/res/layout/activity_main.xml](#))

And `MainActivity.kt` should look like:

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

(from [To8-Nav/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [build.gradle](#)
- [app/build.gradle](#)
- [app/src/main/res/navigation/nav_graph.xml](#)
- [app/src/main/res/layout/activity_main.xml](#)
- [app/src/main/java/com/commonsware/todo/MainActivity.kt](#)

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 GitLab 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 styles and themes in the “Defining and Using Styles” chapter of [Elements of Android Jetpack](#)!

SETTING UP THE APP BAR



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:

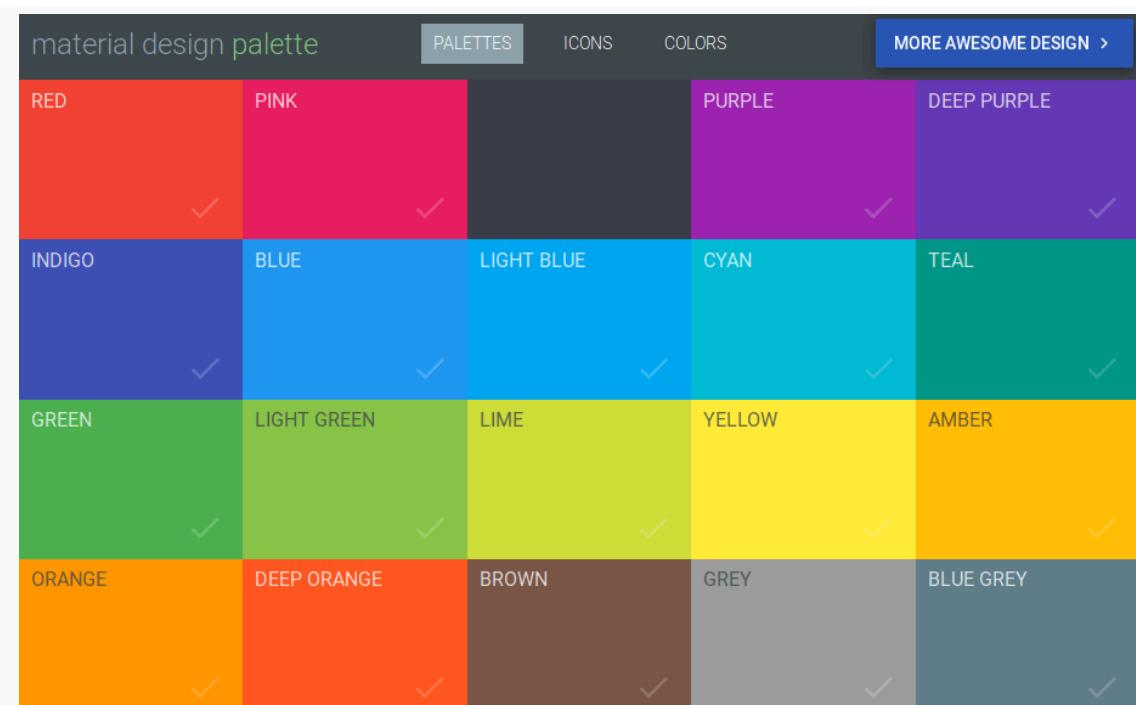


Figure 74: Material Design “Palette” Site, As Initially Launched

SETTING UP THE APP BAR

For the purposes of this tutorial, click on “Teal”, then “Amber”:

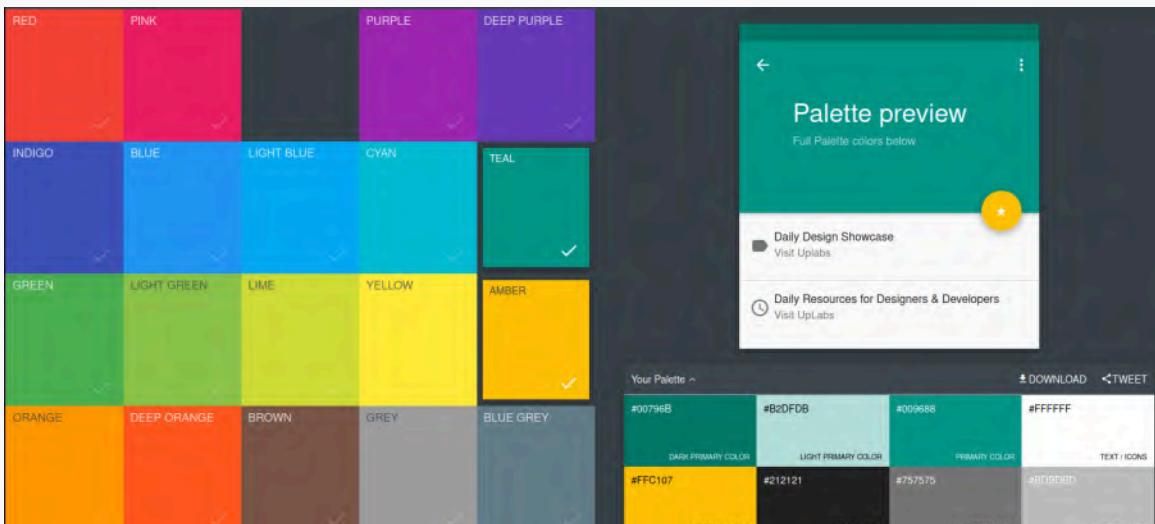


Figure 75: Material Design “Palette” Site, With Teal/Amber Colors

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_teal_amber.xml`. Open in it your favorite text editor. You should see something like:

```
<!-- Palette generated by Material Palette - materialpalette.com/teal/amber -->
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="primary">#009688</color>
    <color name="primary_dark">#00796B</color>
    <color name="primary_light">#B2DFDB</color>
    <color name="accent">#FFC107</color>
    <color name="primary_text">#212121</color>
    <color name="secondary_text">#757575</color>
    <color name="icons">#FFFFFF</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">#008577</color>
```

SETTING UP THE APP BAR

```
<color name="colorPrimaryDark">#00574B</color>
<color name="colorAccent">#D81B60</color>
</resources>
```

(from [To8-Nav/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 #009688
- Change colorPrimaryDark to #00796B
- Change colorAccent to #FFC107

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

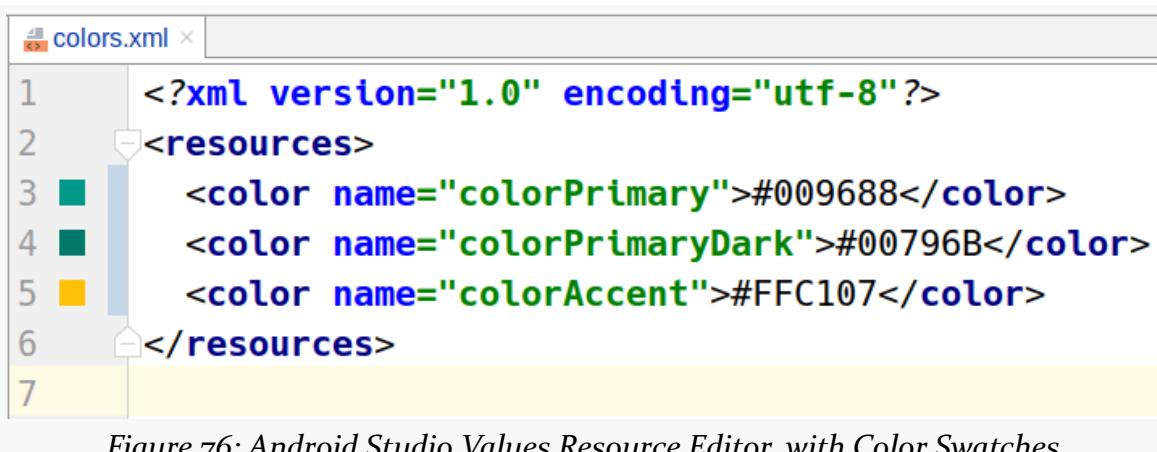


Figure 76: Android Studio Values Resource Editor, with Color Swatches

SETTING UP THE APP BAR

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:

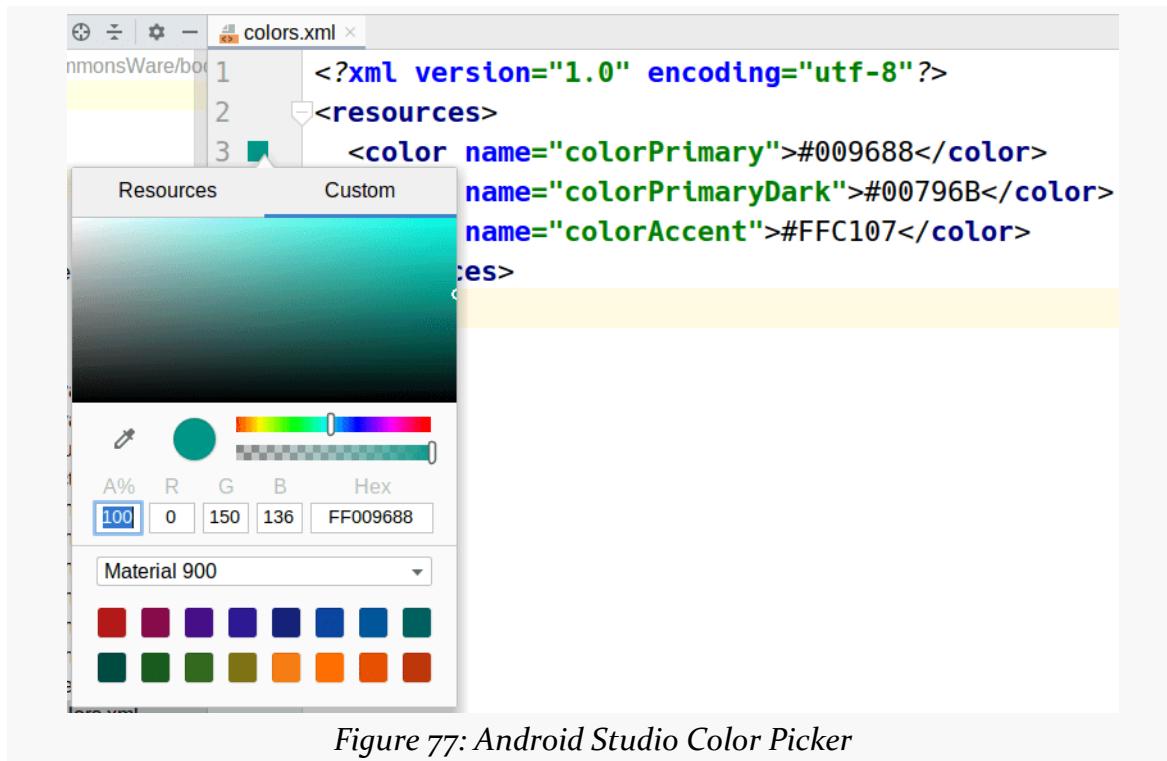


Figure 77: Android Studio Color Picker

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="Theme.ToDo" 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>
```

SETTING UP THE APP BAR

```
</style>  
  
</resources>
```

(from [To8-Nav/ToDo/app/src/main/res/values/styles.xml](#))

Here, we see that we have a style resource named `Theme . ToDo`. 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

However, we will be configuring the app bar ourselves with a `Toolbar`. By default, a `DarkActionBar` theme will add an app bar for us, which we do not need.

Another consideration is whether the overall color scheme will be “light” or “dark”. Historically, Google would steer developers towards a “light” theme, with dark text on a mostly-white background. This is not great for people using apps in dark places or at night, though. In Android 10, Google is starting to steer developers towards having *two* themes: a light one for normal use and a “dark mode” one. This means that you, the developer, have four main courses of action:

1. Ignore Google and stick with a light theme
2. Use two themes
3. Use a single “day-night” theme with two sets of colors
4. Use a dark theme all the time

The latter approach is simplest, in that it accommodates “dark mode” scenarios and does not require that you deal with two themes or two sets of colors. So, that is what we will use in this tutorial.

With that in mind, replace `Theme . AppCompat . Light . DarkActionBar` in `res/value/styles.xml` with `Theme . AppCompat . NoActionBar`. Removing the `Light` portion gives us a dark theme, and replacing `DarkActionBar` with `NoActionBar` removes the

SETTING UP THE APP BAR

automatically-added app bar.

The resulting resource should look like:

```
<resources>

    <!-- Base application theme. -->
    <style name="Theme.ToDo" parent="Theme.AppCompat.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 [Tog-Toolbar/ToDo/app/src/main/res/values/styles.xml](#))

Note that the color swatches in the gutter in this Android Studio are clickable as well, bringing up the same editor as before, this time defaulting to the “Resources” tab:

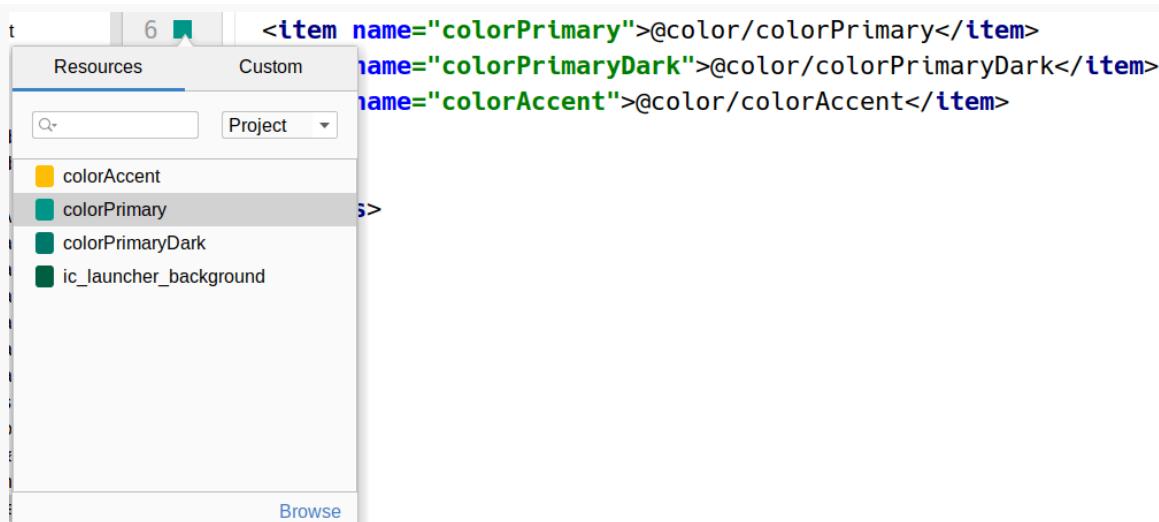


Figure 78: Style Resource Editor, Showing Pop-Up Color Picker

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

SETTING UP THE APP BAR

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.todo">

    <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/Theme.ToDo">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

(from [ToG-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 `FragmentContainerView`. Now, we want to modify the layout to have a Toolbar at the top.

SETTING UP THE APP BAR

Switch to the “Design” view if you are not already there. In the “Containers” category of the “Palette”, you should find a Toolbar option:

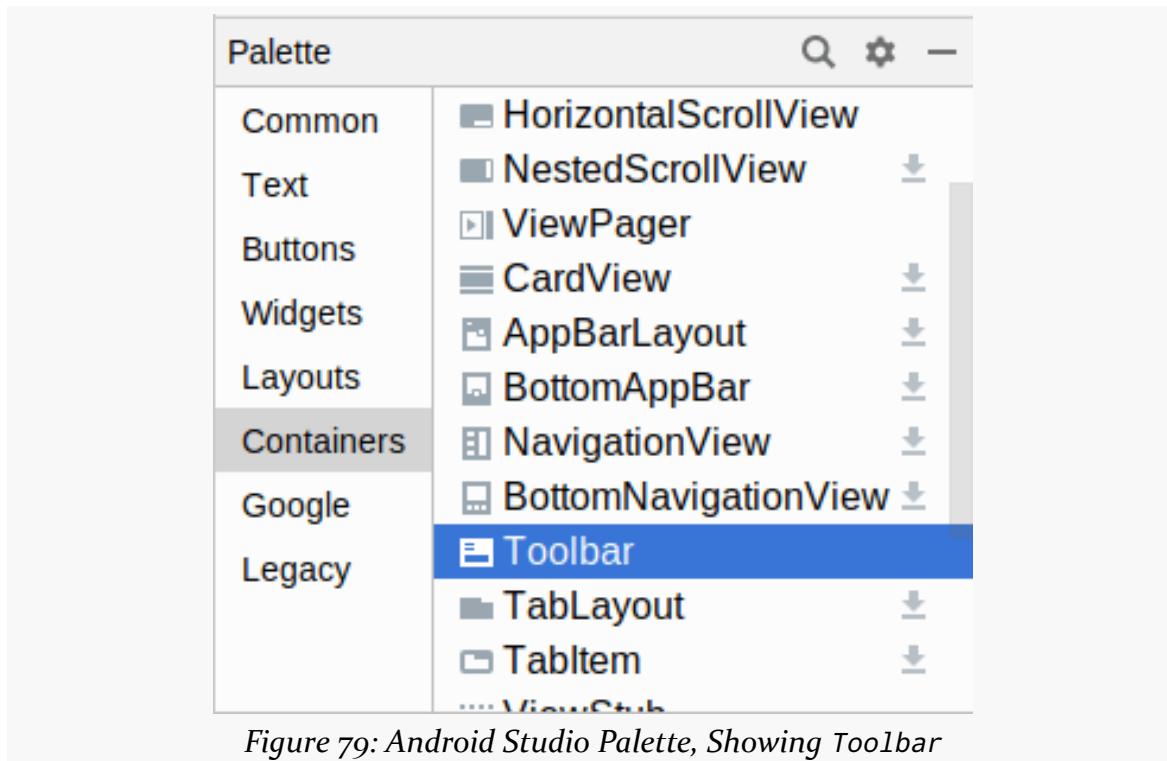


Figure 79: Android Studio Palette, Showing Toolbar

SETTING UP THE APP BAR

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

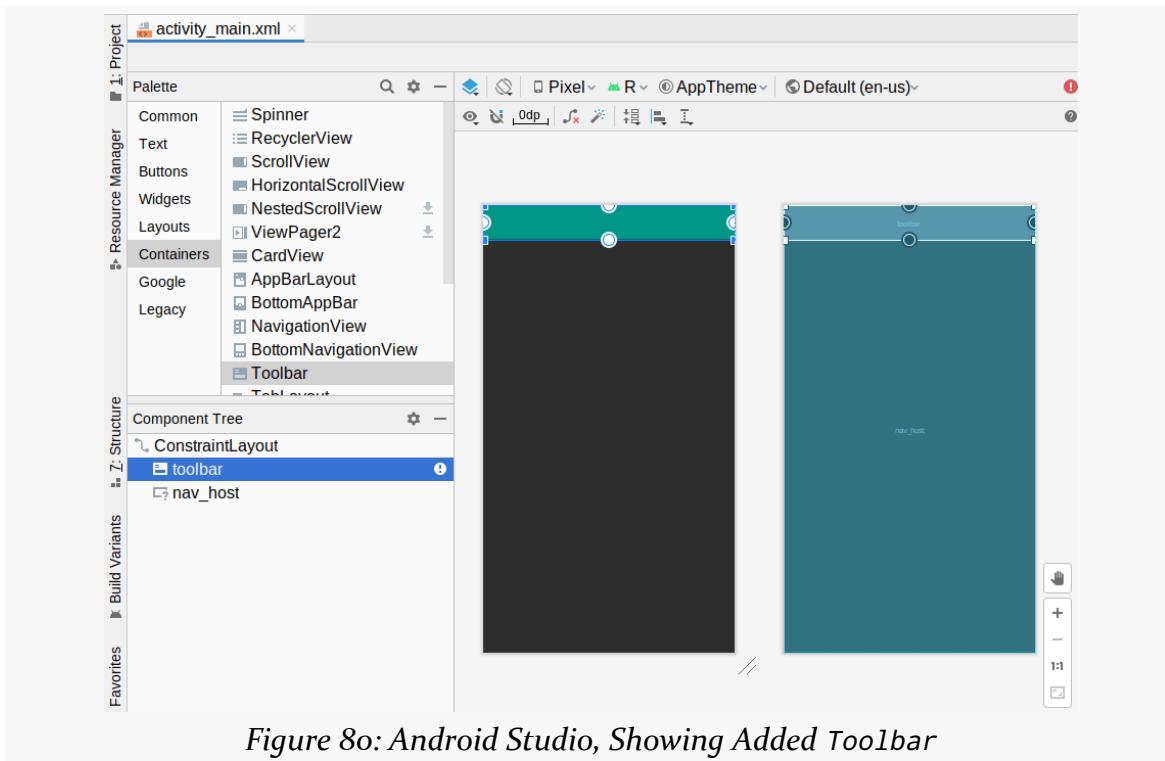


Figure 8o: Android Studio, Showing Added Toolbar

Next, we need to set up the anchoring rules for the Toolbar. 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. And, since the drag-and-drop editor makes this difficult, you could elect to modify the XML instead and add

```
app:layout_constraintStart_toStartOf="parent",
app:layout_constraintEnd_toEndOf="parent", and
app:layout_constraintTop_toTopOf="parent" to the Toolbar.
```

SETTING UP THE APP BAR

Next, click on the `nav_host` entry in the “Component Tree” to select the `FragmentContainerView`. 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`:

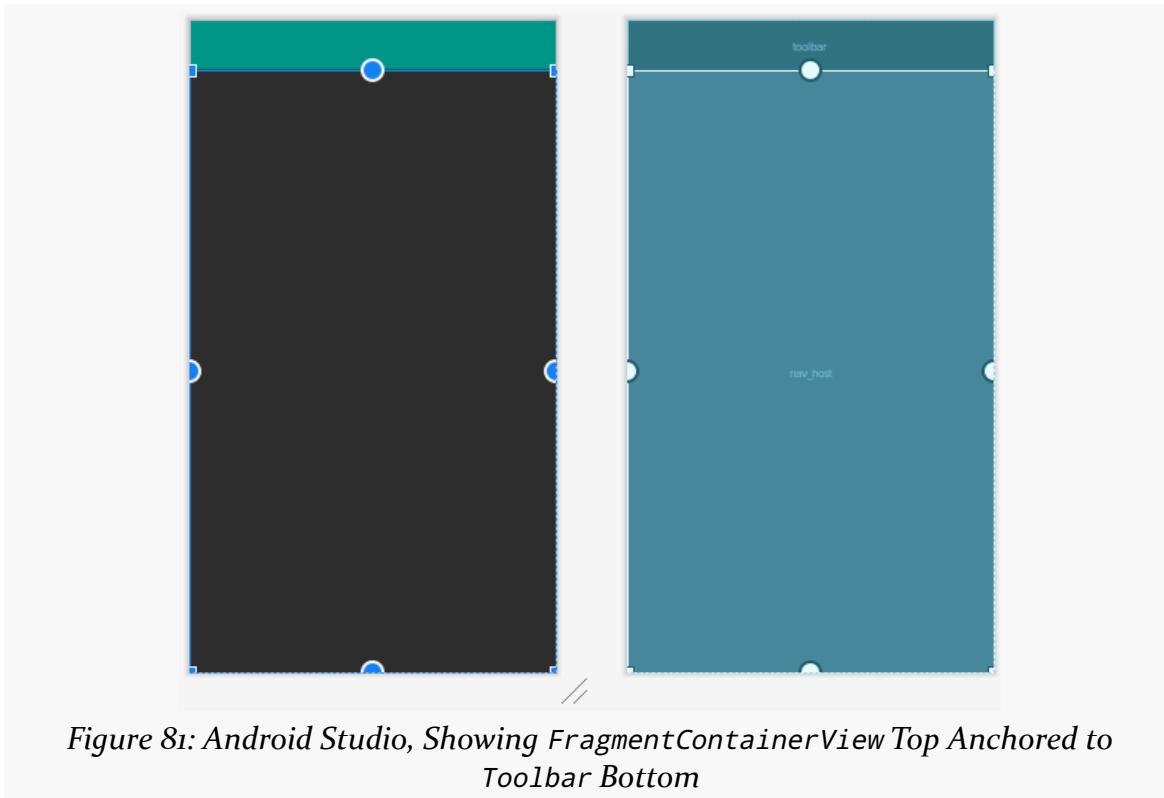


Figure 81: Android Studio, Showing `FragmentContainerView` Top Anchored to `Toolbar` Bottom

Again, the drag-and-drop editor makes this difficult. If you prefer, switch to the XML and replace `app:layout_constraintTop_toTopOf="parent"` to `app:layout_constraintTop_toBottomOf="@+id/toolbar"` on the `FragmentContainerView`.

Then, select the `Toolbar` widget, and in the “Attributes” pane:

SETTING UP THE APP BAR

- Ensure that the ID is set to toolbar (it should be by default)
- Set the layout_width to match_constraint (a.k.a., 0dp)

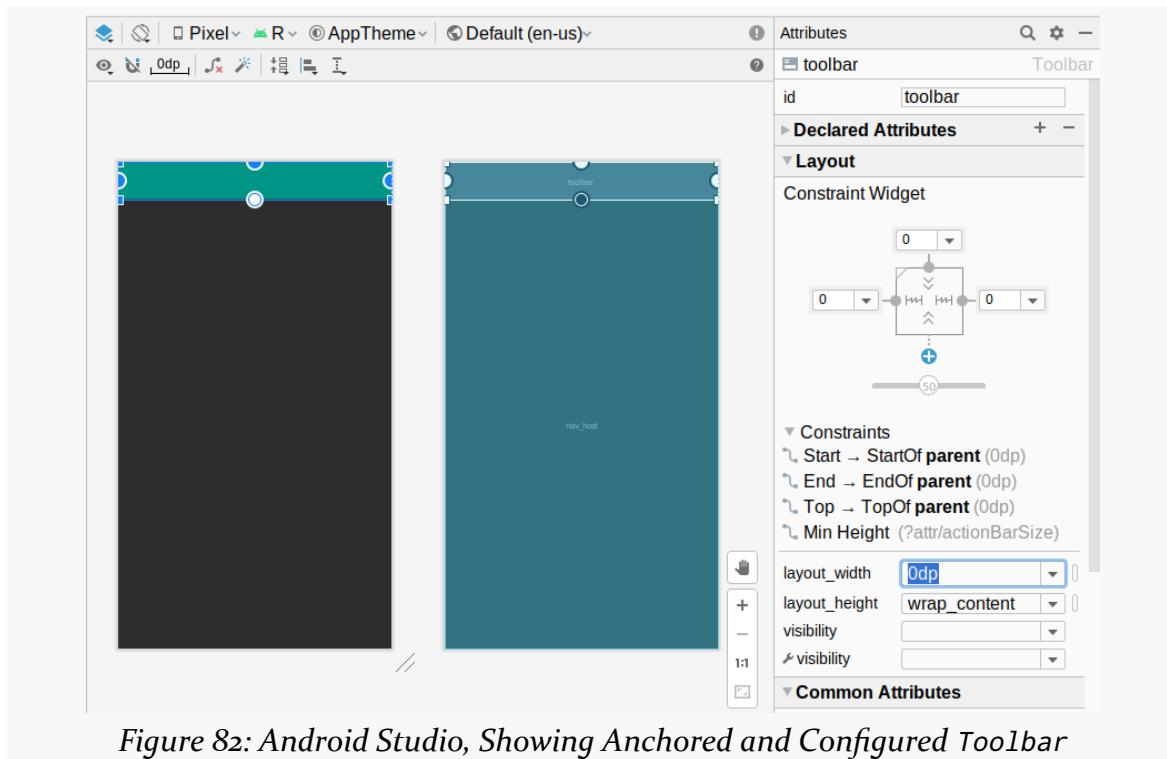


Figure 82: Android Studio, Showing Anchored and Configured Toolbar

Step #4: Adding an Icon

We are going to need a 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.

SETTING UP THE APP BAR

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:

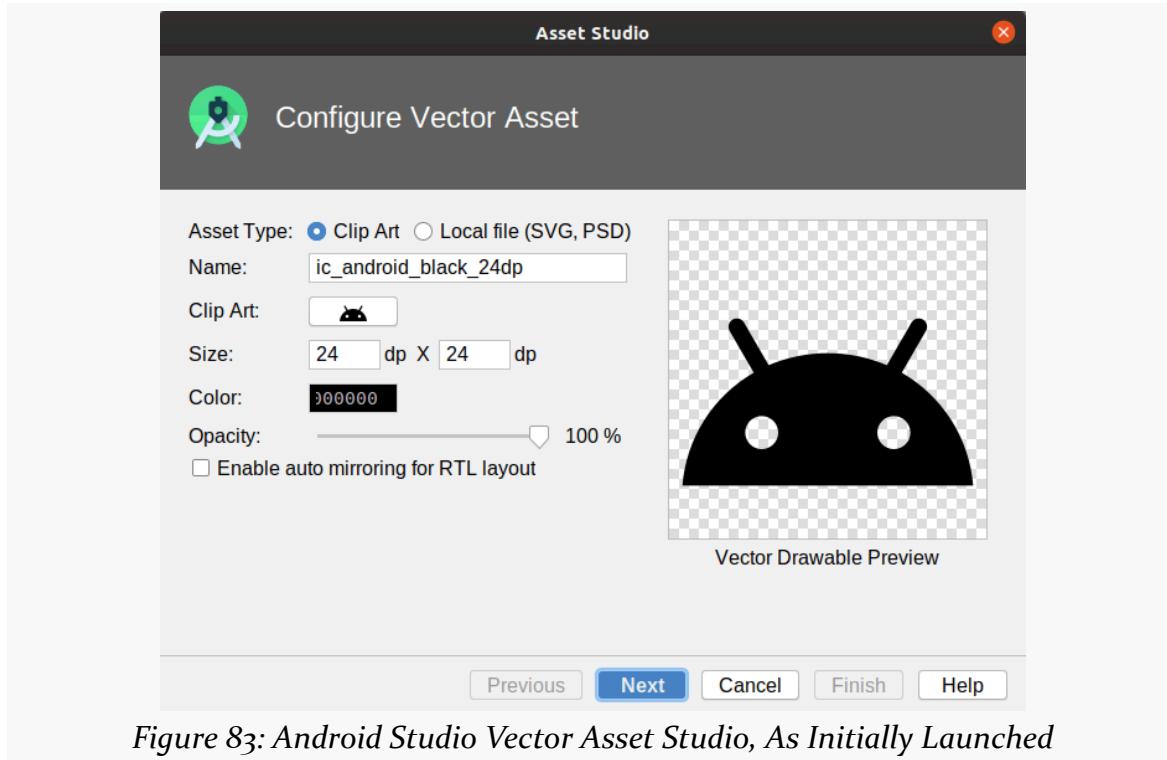


Figure 83: Android Studio Vector Asset Studio, As Initially Launched

SETTING UP THE APP BAR

Click on the “Clip Art” button, which by default has the image of the head of the Android mascot (“bugdroid”). This is supposed to 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” icon:



Figure 84: Android Studio Icon Selector, Showing “info” Icon

Click “OK”. This will update the name of the asset to `ic_baseline_info_24`. Change the name in the “Name” field to `ic_about` (`ic` is a prefix representing icons).

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

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

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

SETTING UP THE APP BAR

Resource Directory” from the context menu. This will bring up a dialog to let you create a new resource directory:

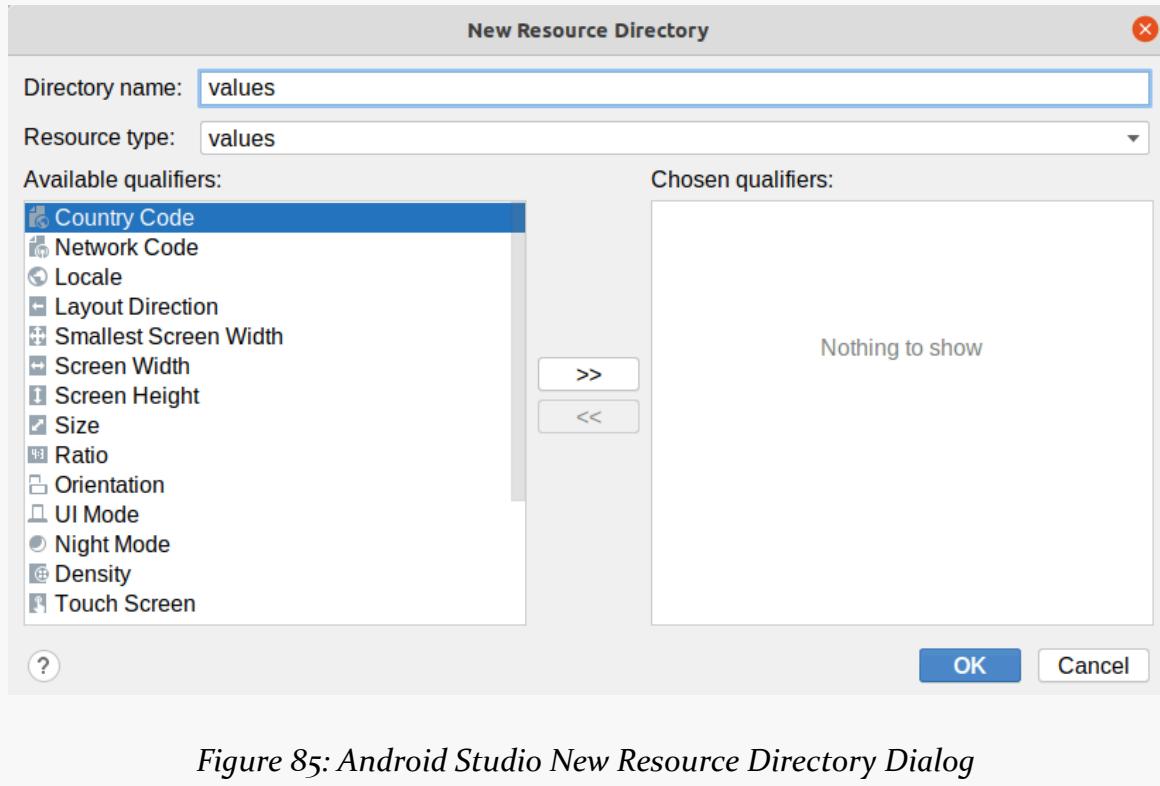


Figure 85: Android Studio New Resource Directory Dialog

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

SETTING UP THE APP BAR

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:

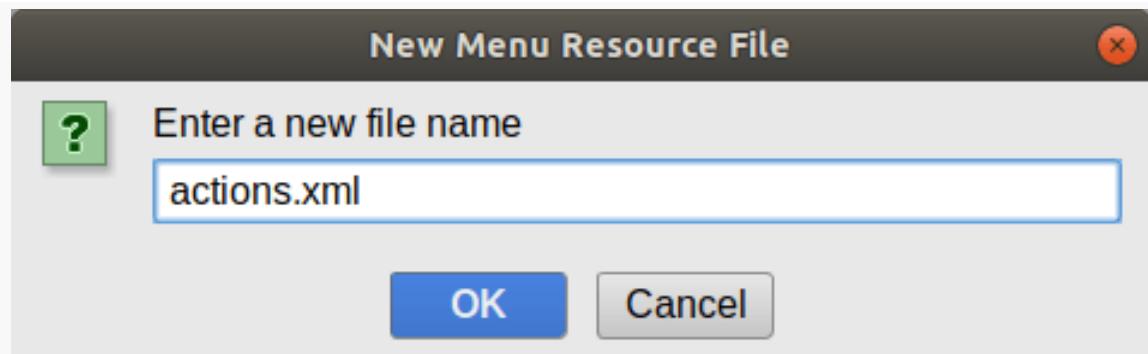


Figure 86: Android Studio New Menu Resource File Dialog

Then click “OK” to create the file. It will open up into a menu editor. As with the layout editor, there are modes that you can toggle via icons in the tab itself:

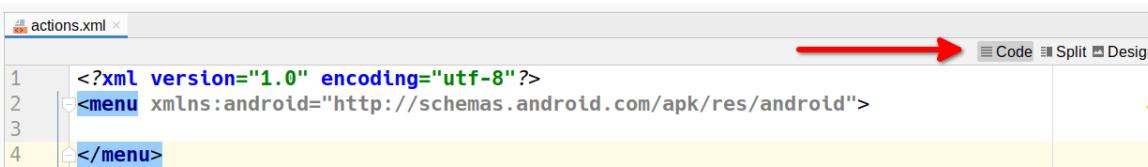


Figure 87: Android Studio Menu Editor, with Toolbar Buttons Highlighted

SETTING UP THE APP BAR

As with the layout designer, the “Code” button shows you the XML of this resource, while the “Design” button gives you a graphical menu designer:

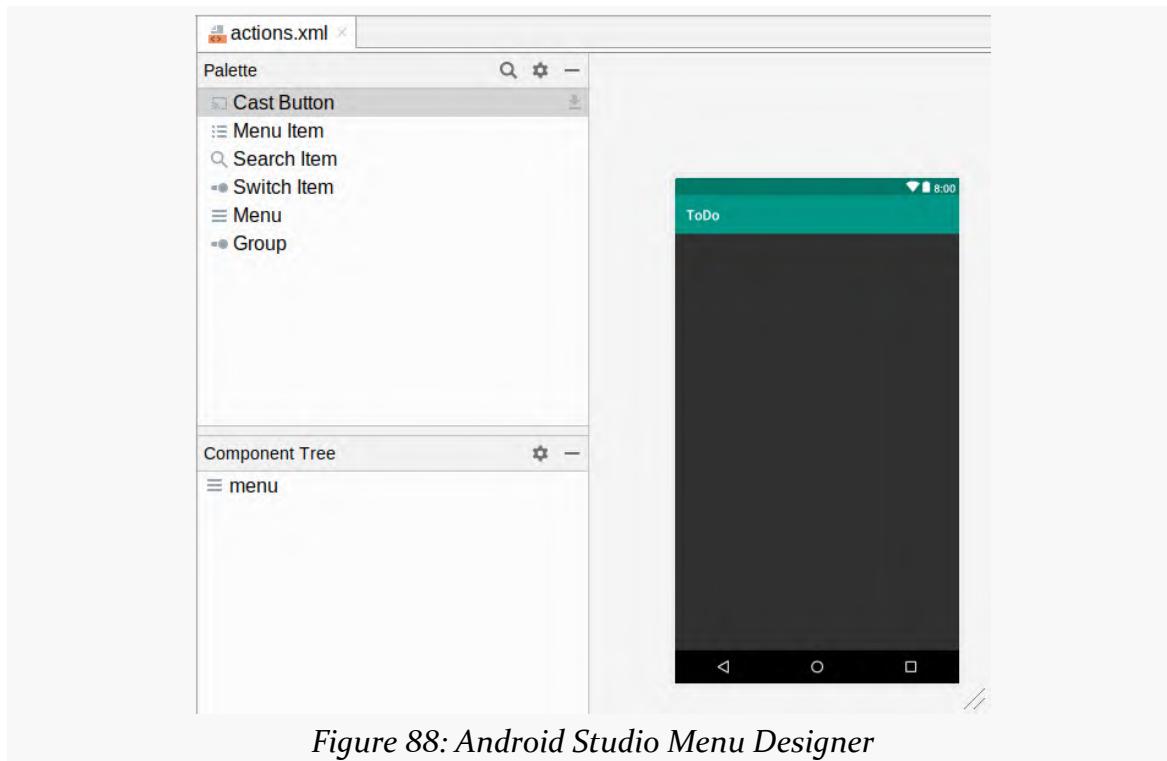


Figure 88: Android Studio Menu Designer

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”.

SETTING UP THE APP BAR

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:

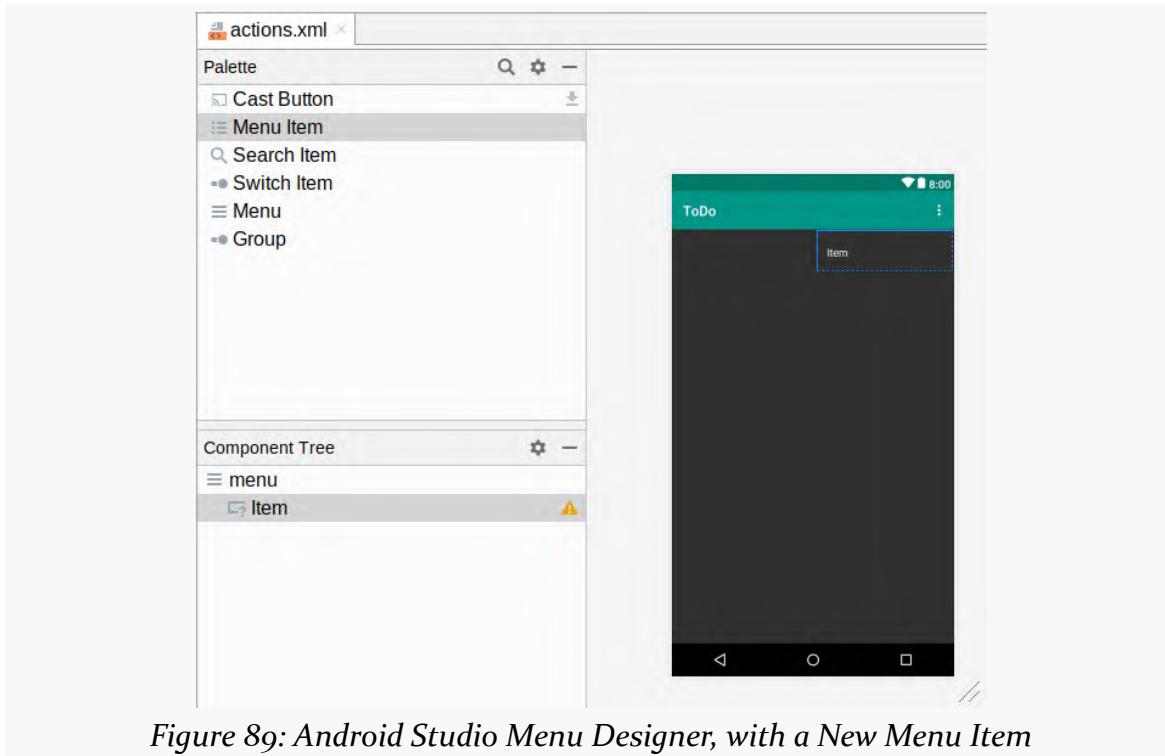


Figure 89: Android Studio Menu Designer, with a New Menu Item

In the Attributes pane, fill in about for the “id”.

SETTING UP THE APP BAR

Next, we want to set the “showAsAction” value to never. To do this, click the little flag icon in the “showAsAction” field:

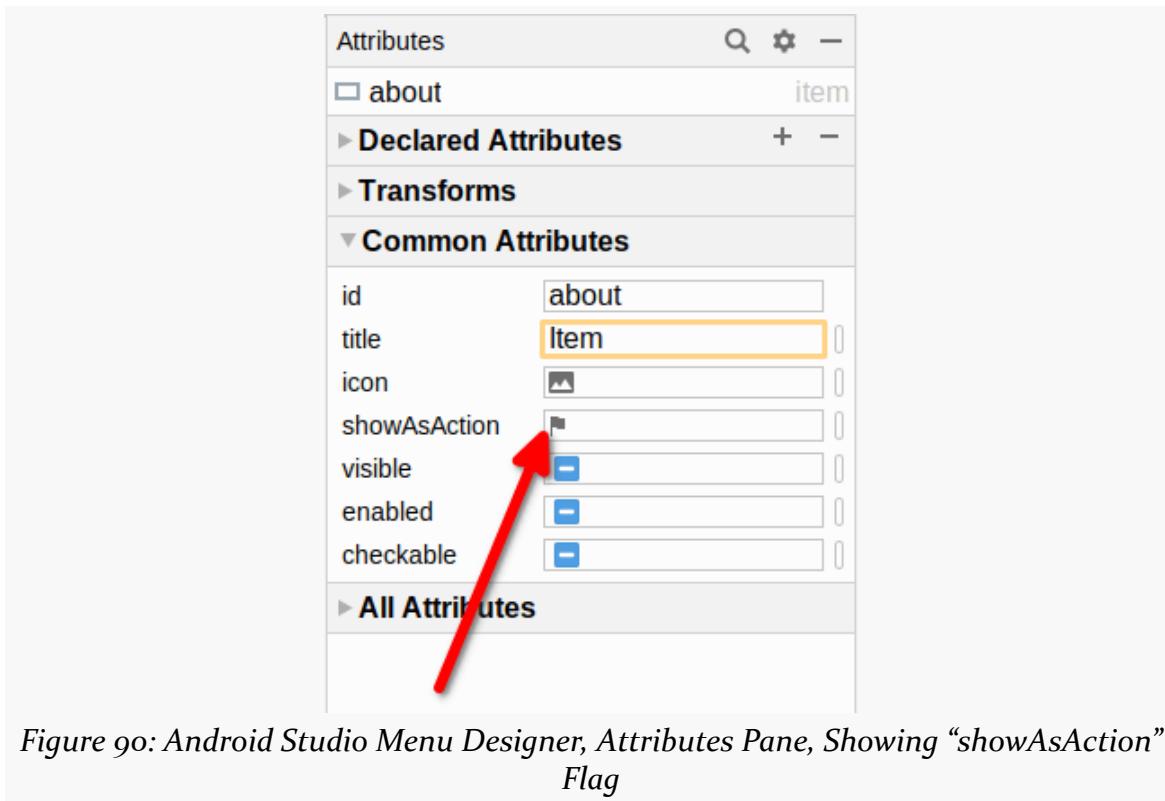


Figure 90: Android Studio Menu Designer, Attributes Pane, Showing “showAsAction” Flag

SETTING UP THE APP BAR

That will fold open a list of available choices:

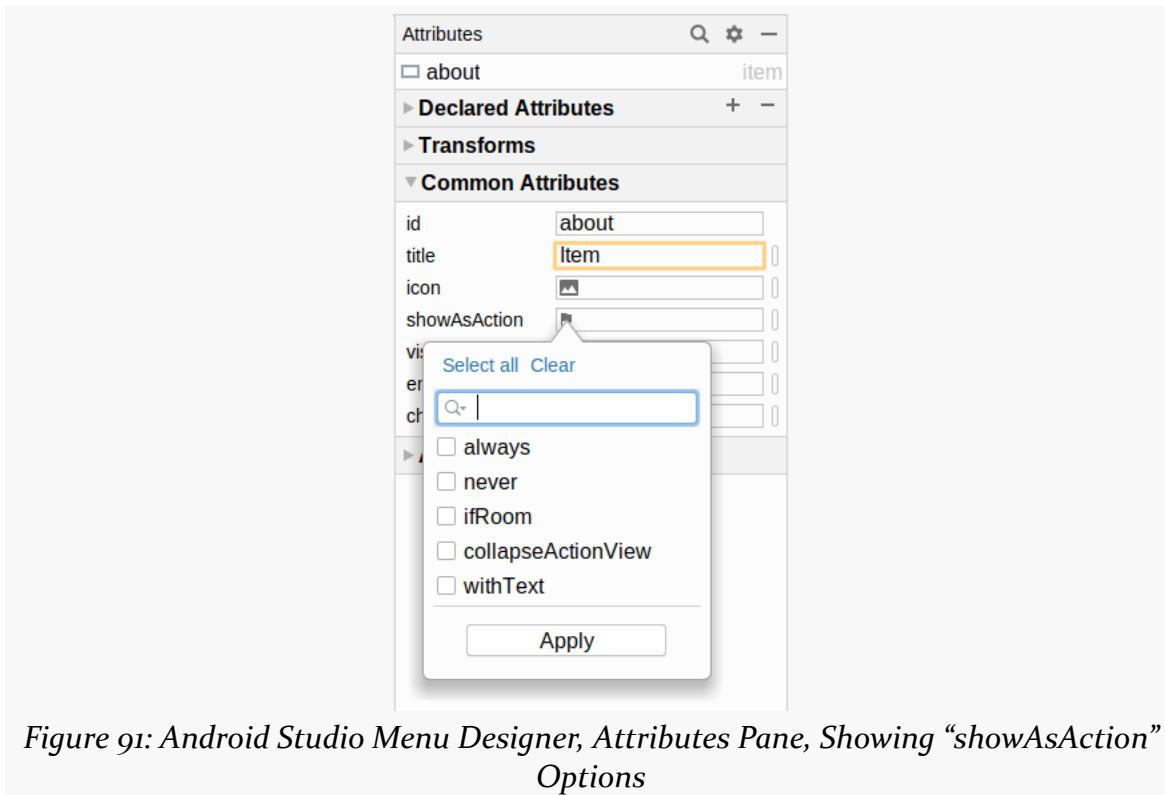


Figure 91: Android Studio Menu Designer, Attributes Pane, Showing “showAsAction” Options

Check the “never” checkbox in the list, then click the “Apply” button in the drop-down to close it and set “showAsAction” to never.

SETTING UP THE APP BAR

Then, click on the “O” button next to the “icon” field:

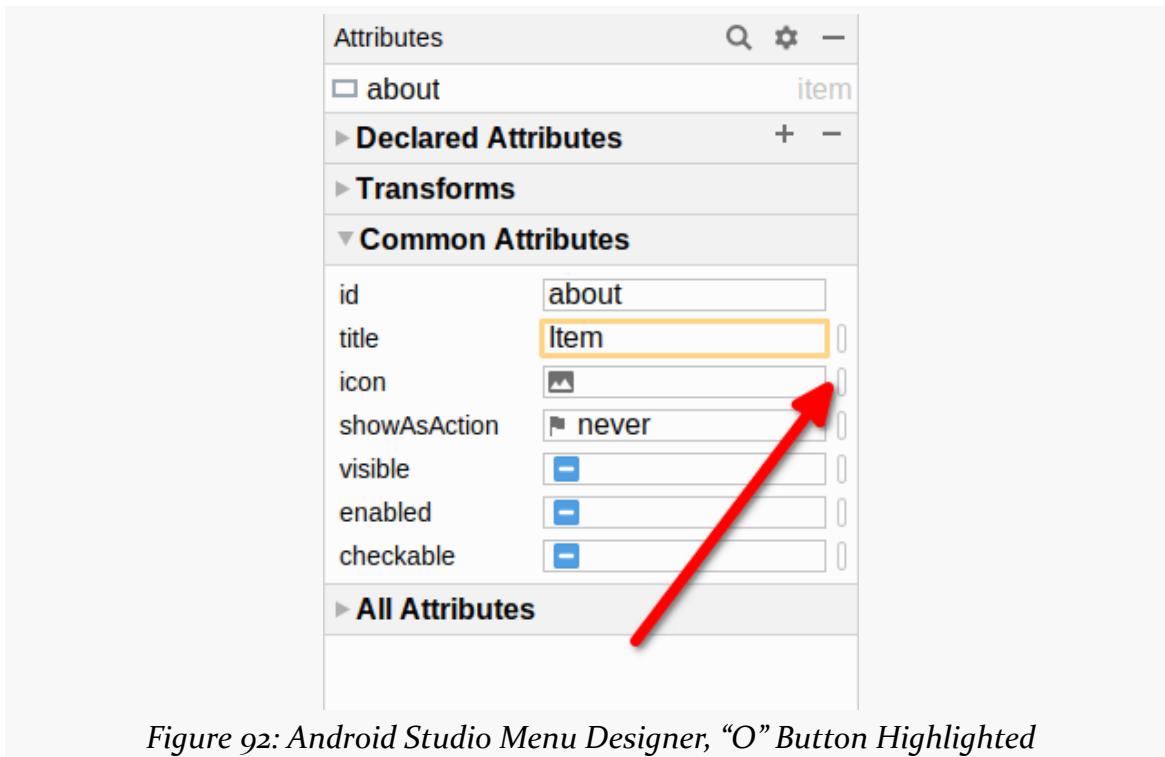


Figure 92: Android Studio Menu Designer, “O” Button Highlighted

SETTING UP THE APP BAR

This will bring up a drawable resource selector:

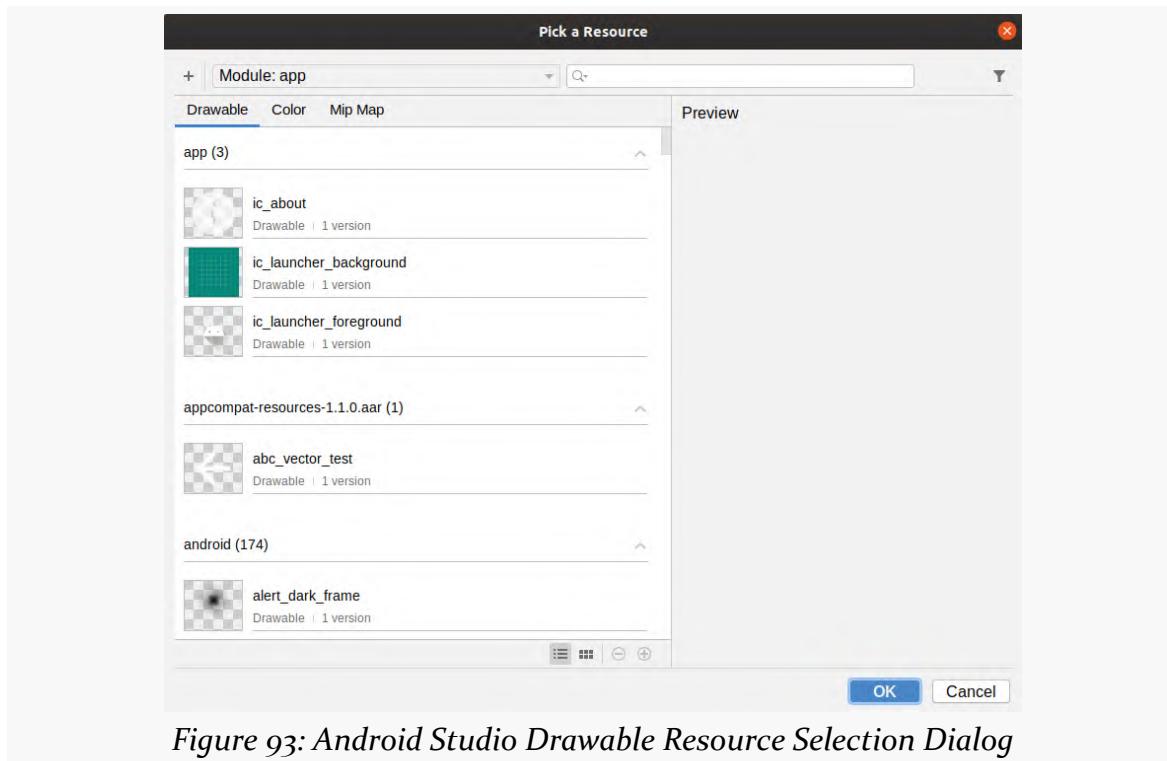


Figure 93: Android Studio Drawable Resource Selection Dialog

Click on `ic_about` in the list of 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.

SETTING UP THE APP BAR

Then, click the “O” button next to the “title” field. As before, this brings up a string resource selector. Click on the “+” icon, followed by “String Value” in the resulting drop-down list. This will bring up a new string resource dialog. In the dialog, fill in menu_about as the resource name and “About” as the resource value:

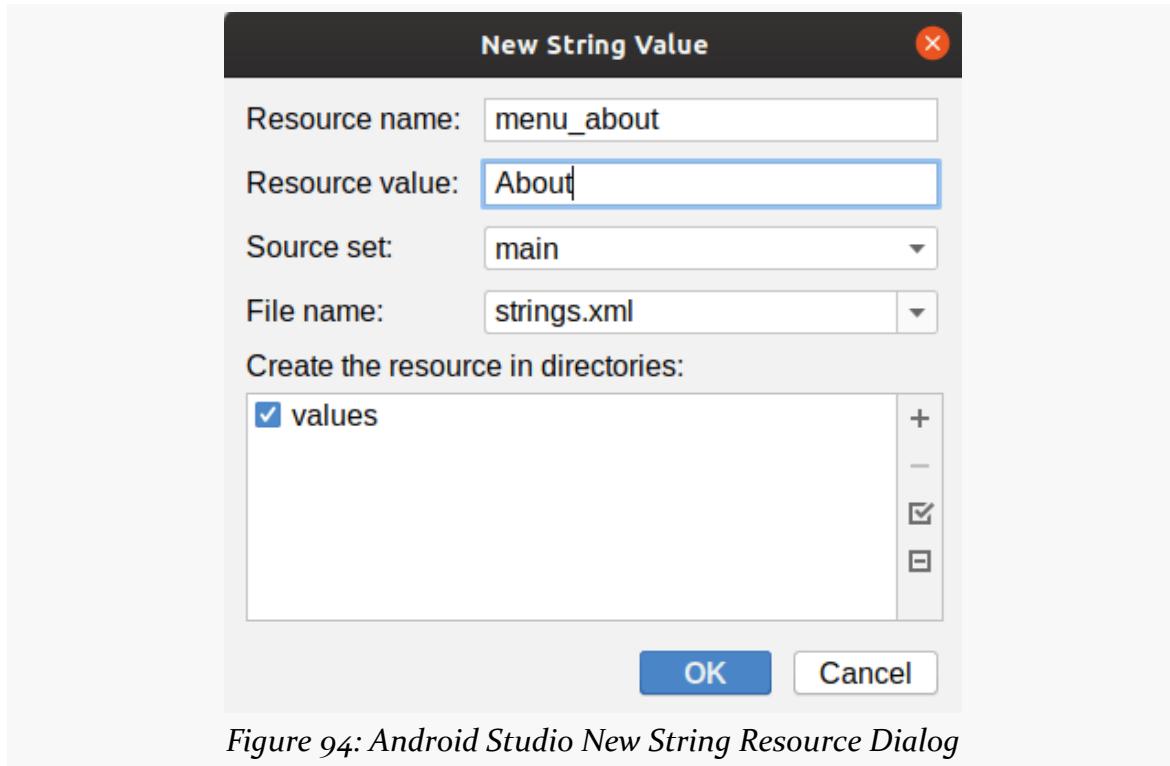


Figure 94: Android Studio New String Resource Dialog

SETTING UP THE APP BAR

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

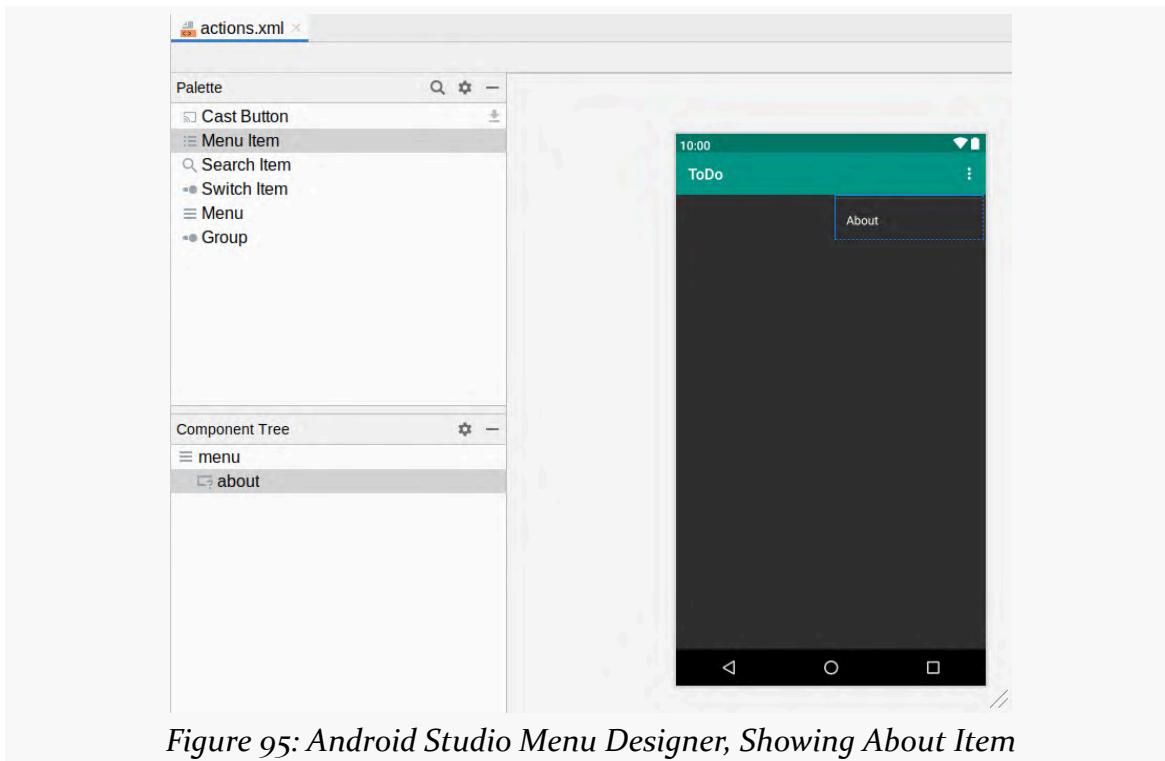


Figure 95: Android Studio Menu Designer, Showing About Item

Step #6: Enabling View Binding

We are going to need to start working with our widgets from Kotlin. There are a variety of options for doing this. The one that we will use in this book is called “view binding”. By enabling view binding, the build tools will code-generate a class that helps us work with our widgets in a type-safe fashion.

To turn this on, add these lines to the android closure in your `app/build.gradle` file:

```
buildFeatures {  
    viewBinding true  
}
```

(from [ToG-Toolbar/ToDo/app/build.gradle](#))

SETTING UP THE APP BAR



You can learn more about view binding in the "Binding Your Data" chapter of [Elements of Android Jetpack!](#)

When the editor suggests that you sync your Gradle files with the project, go ahead and do that.

NOTE: You will see these lines written elsewhere as:

```
buildFeatures {  
    viewBinding = true  
}
```

That was the correct syntax for a while, but Arctic Fox (and its edition of the Android Gradle Plugin) changed the syntax.

Step #7: Using View Binding in Our Activity

Next, modify `onCreate()` of `MainActivity` to look like this:

```
override fun onCreate(savedInstanceState: Bundle?) {  
    super.onCreate(savedInstanceState)  
  
    val binding = ActivityMainBinding.inflate(layoutInflater)  
  
    setContentView(binding.root)  
}
```

The class code-generated by view binding is based off of the name of the layout resource. The `lower_snake_case` portion of the name is converted into `UpperCamelCase`, then gets `Binding` appended to it. So, `activity_main` becomes `ActivityMainBinding`.

Originally, `onCreate()` used `setContentView(R.layout.activity_main)`. “Under the covers”, this would use a `LayoutInflater` to “inflate” `activity_main`, creating a tree of widgets and containers based on what is in the layout resource file. `setContentView()` would then take the root of that hierarchy and use it for rendering the UI.

Our two replacement lines do the same thing, in the end. The `inflate()` function

SETTING UP THE APP BAR

on the `ActivityMainBinding` class inflates the `activity_main` layout using a `LayoutInflater`. It gets that `LayoutInflater` from us as a parameter to `inflate()`, and we get one via `layoutInflater` from the activity. The `ActivityMainBinding` object that we get back from `inflate()` has a `root` property, and we pass that to `setContentView()` to display our view hierarchy. The only effective difference between what we had and what we now have is that we have the `binding` object, and we can use that to reference the widgets inside of the layout, such as using `binding.toolbar` to get to our Toolbar.

Step #8: 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.

Once again, there are a few ways of doing this. For this book, we are going to use our Toolbar as the action bar. This is the simplest way to have multiple fragments all contribute to the Toolbar. In particular, it is the simplest way to have those fragments' contributions come and go as the fragments themselves come and go.

To do that, add a `setSupportActionBar()` call to the bottom of `onCreate()` of `MainActivity`:

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

    val binding = ActivityMainBinding.inflate(layoutInflater)

    setContentView(binding.root)
    setSupportActionBar(binding.toolbar)
}
```

(from [ToG-Toolbar/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

This tells `AppCompatActivity` that we want to use our Toolbar in the role of the activity's action bar. `binding.toolbar` is a reference to the Toolbar widget from our layout, courtesy of view binding.

Then, add this function to `MainActivity`

SETTING UP THE APP BAR

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.actions, menu)

    return super.onCreateOptionsMenu(menu)
}
```

(from [ToDo/Toolbar/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

This is how you contribute toolbar buttons, overflow menu items, and other things to the activity's action bar. The "options menu" name is a reference to the original Android UI (from Android 1.0).

Here, just as we used a LayoutInflator with ActivityMainBinding to inflate a layout resource, we use a MenuInflater to inflate a menu resource, pouring its contents into the supplied Menu object. We then chain to the superclass, just in case some superclass also wants to put things in the action bar.

Step #9: Trying It Out

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

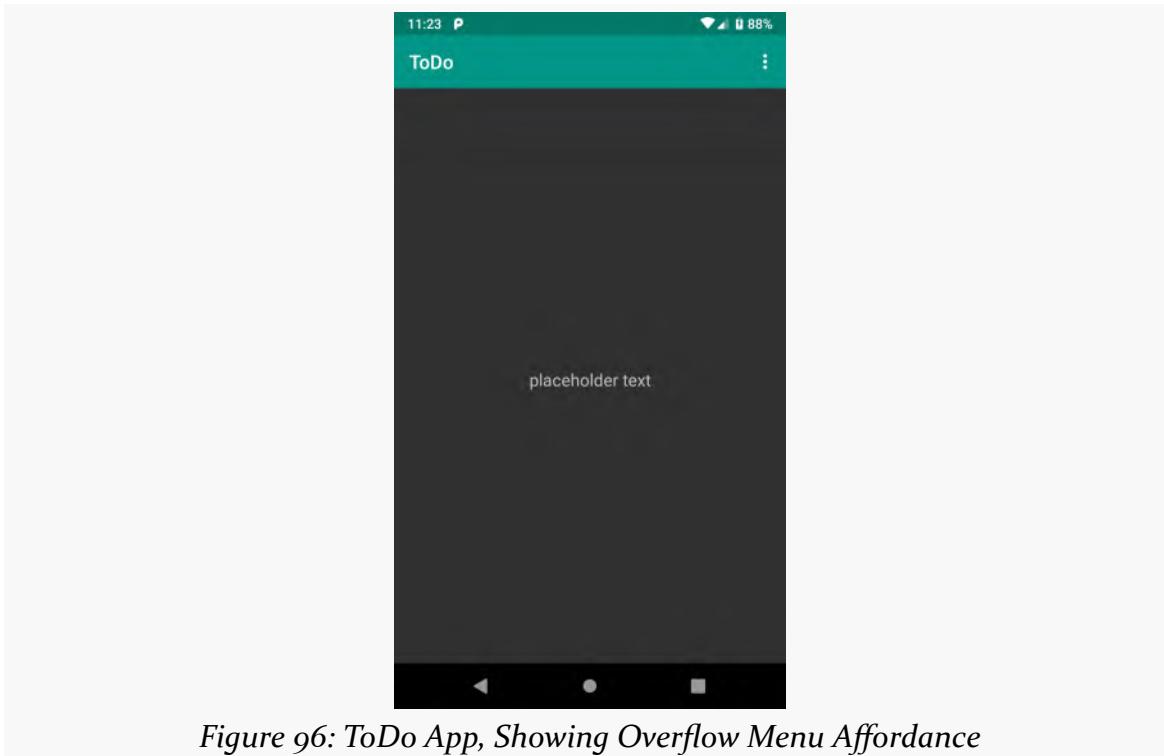


Figure 96: ToDo App, Showing Overflow Menu Affordance

SETTING UP THE APP BAR

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

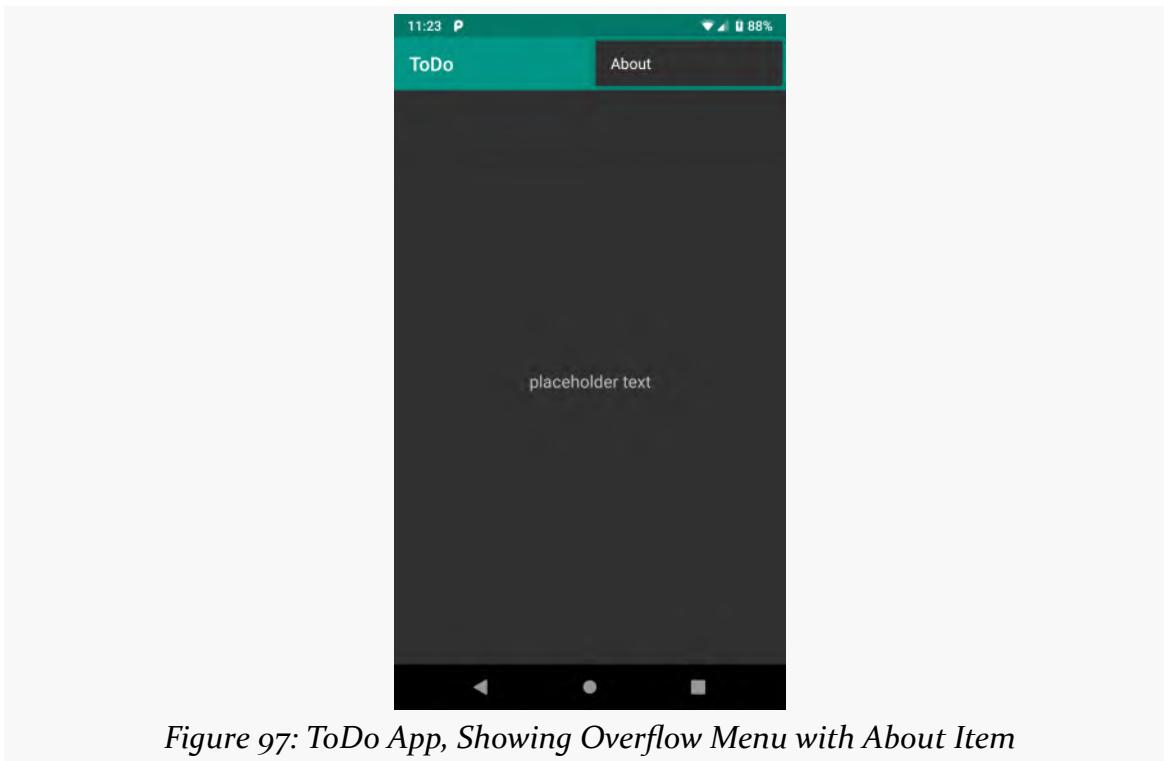


Figure 97: ToDo App, Showing Overflow Menu with About Item

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

Final Results

In theory, your `res/layout/activity_main.xml` resource 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"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <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"
```

SETTING UP THE APP BAR

```
    android:theme="?attr/actionBarTheme"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

<androidx.fragment.app.FragmentContainerView
    android:id="@+id/nav_host"
    android:name="androidx.navigation.fragment.NavHostFragment"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:defaultNavHost="true"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/toolbar"
    app:navGraph="@navigation/nav_graph">

</androidx.fragment.app.FragmentContainerView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [Tog-Toolbar/ToDo/app/src/main/res/layout/activity_main.xml](#))

It might vary a from this, given that the drag-and-drop GUI editor is not very precise.

Your app/build.gradle file should look like:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'androidx.navigation.safeargs.kotlin'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

    buildTypes {
```

SETTING UP THE APP BAR

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

buildFeatures {
    viewBinding true
}

compileOptions {
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

(from [ToG-Toolbar/ToDo/app/build.gradle](#))

And `MainActivity` should resemble:

```
package com.commonsware.todo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.view.Menu
import com.commonsware.todo.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

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

SETTING UP THE APP BAR

```
val binding = ActivityMainBinding.inflate(layoutInflater)

setContentView(binding.root)
setSupportActionBar(binding.toolbar)
}

override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.actions, menu)

    return super.onCreateOptionsMenu(menu)
}
}
```

(from [To9-Toolbar/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

What We Changed

The book's GitLab 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.xml](#)
- [app/src/main/res/values/styles.xml](#)
- [app/src/main/res/layout/activity_main.xml](#)
- [app/build.gradle](#)
- [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.

We could set up another fragment, and have that be displayed when the user clicked “About”. However, we have a few other fragments to set up, so we will have plenty of opportunities to learn about fragments. Besides, we do not want you to get bored.

So, 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. While in modern Android app development you would not need a full activity to display an “about” screen, there may be times when you really do need another activity, so this will show you how to set one up.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab 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`. We could just create a new empty class, but the Android Studio new-activity wizard is

SETTING UP AN ACTIVITY

not bad, so we will use it.

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:

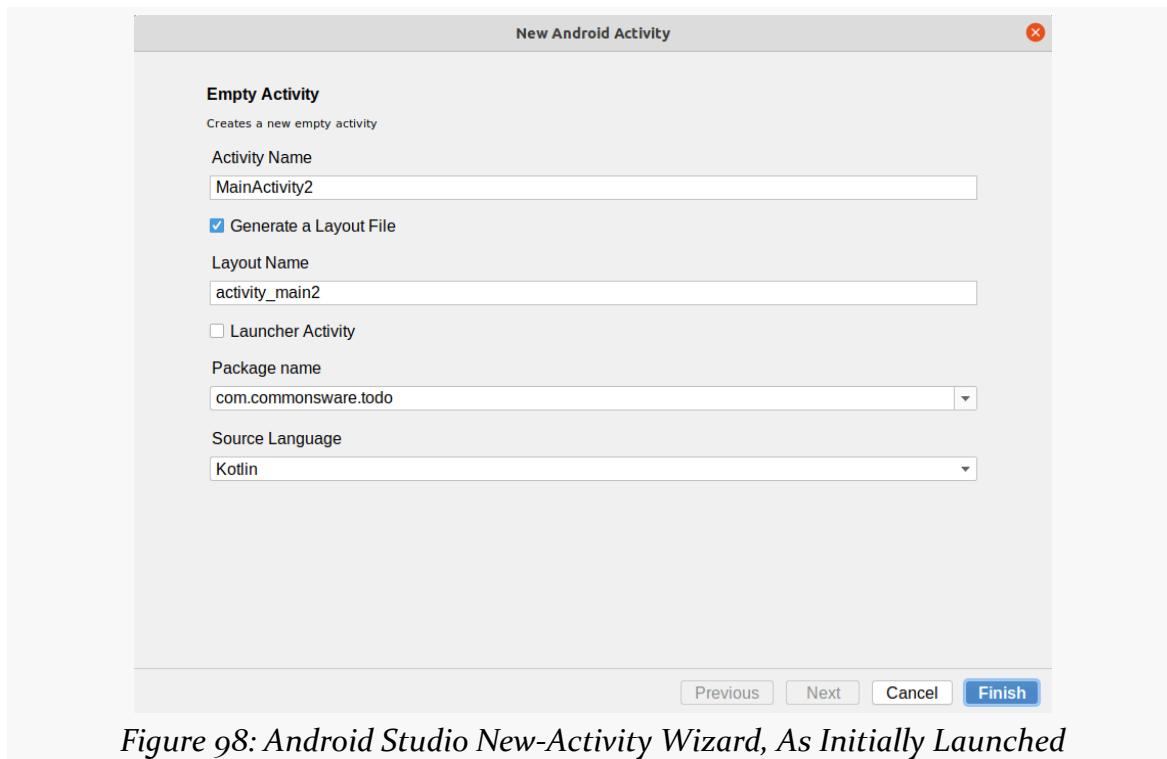


Figure 98: Android Studio New-Activity Wizard, As Initially Launched

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.commonware.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.

SETTING UP AN ACTIVITY

This should give you a dialog like:

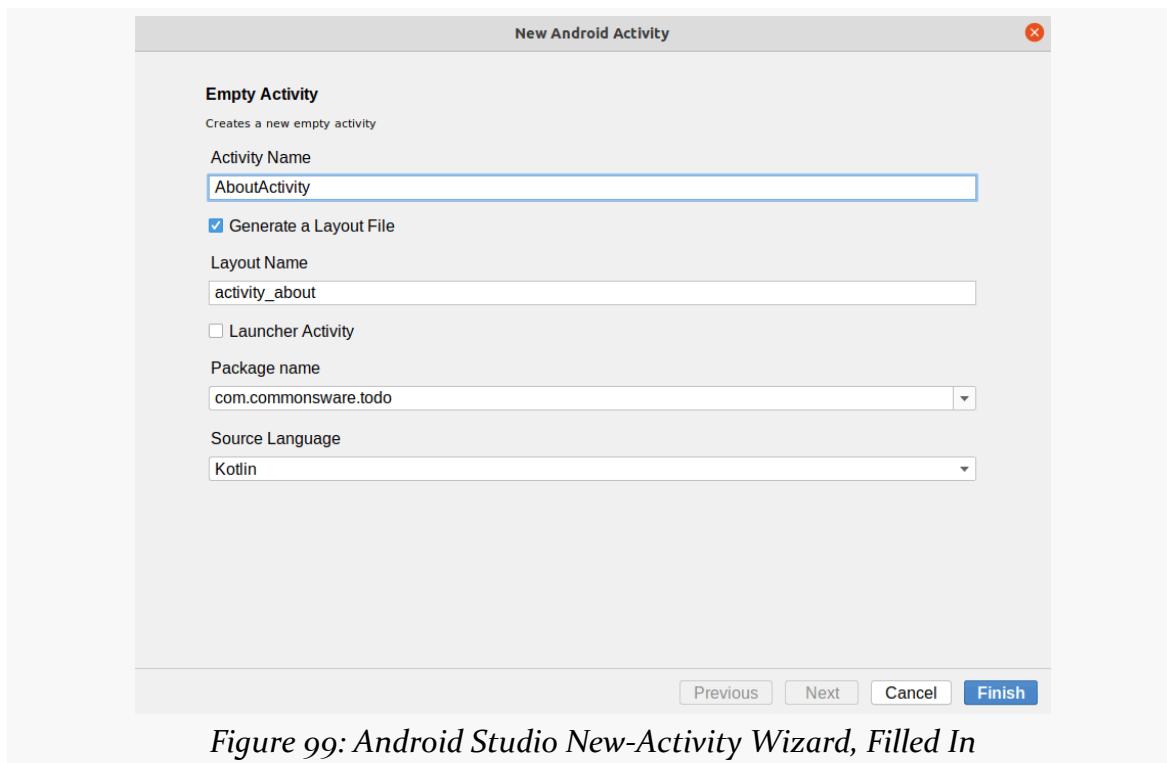


Figure 99: Android Studio New-Activity Wizard, Filled In

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.commonware.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"
    android:exported="true" />
```

Step #2: Adding a Toolbar and a WebView

In addition to a new `AboutActivity` Kotlin 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:

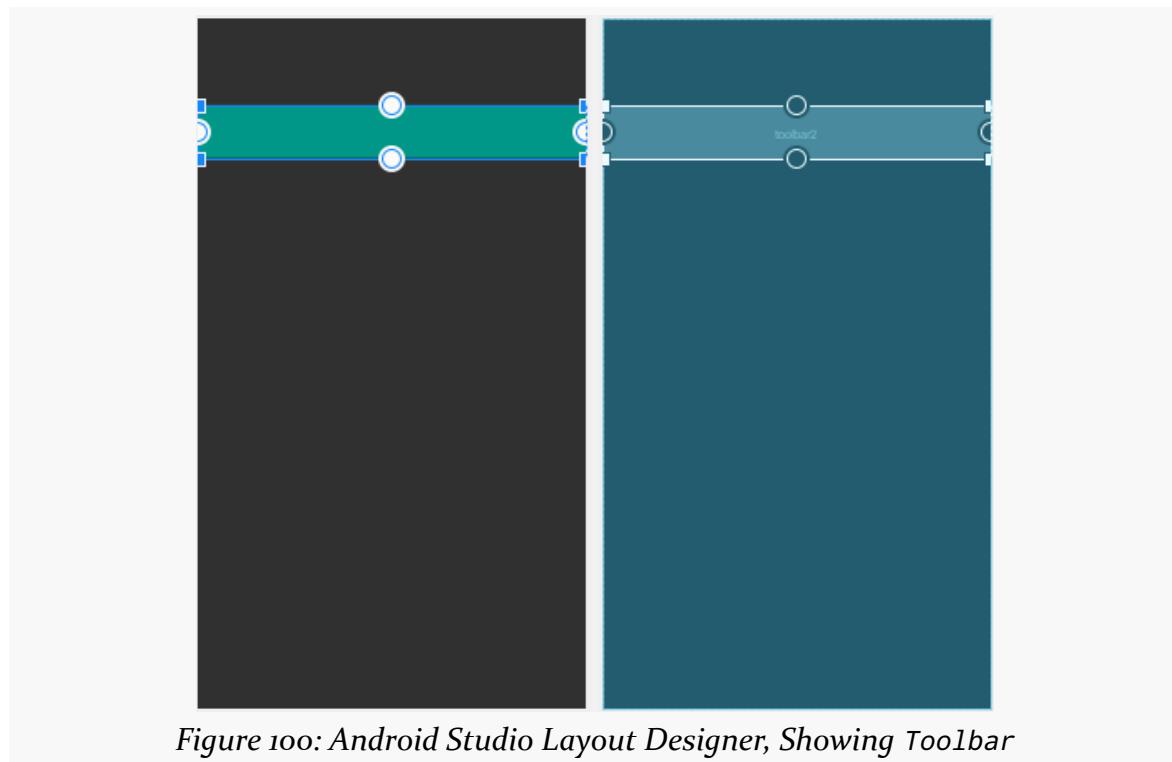


Figure 100: Android Studio Layout Designer, Showing Toolbar

SETTING UP AN ACTIVITY

Using the corner (square) handles, drag the ends of the Toolbar in from the left and right edges of the layout, so you have room to maneuver. Then, 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:

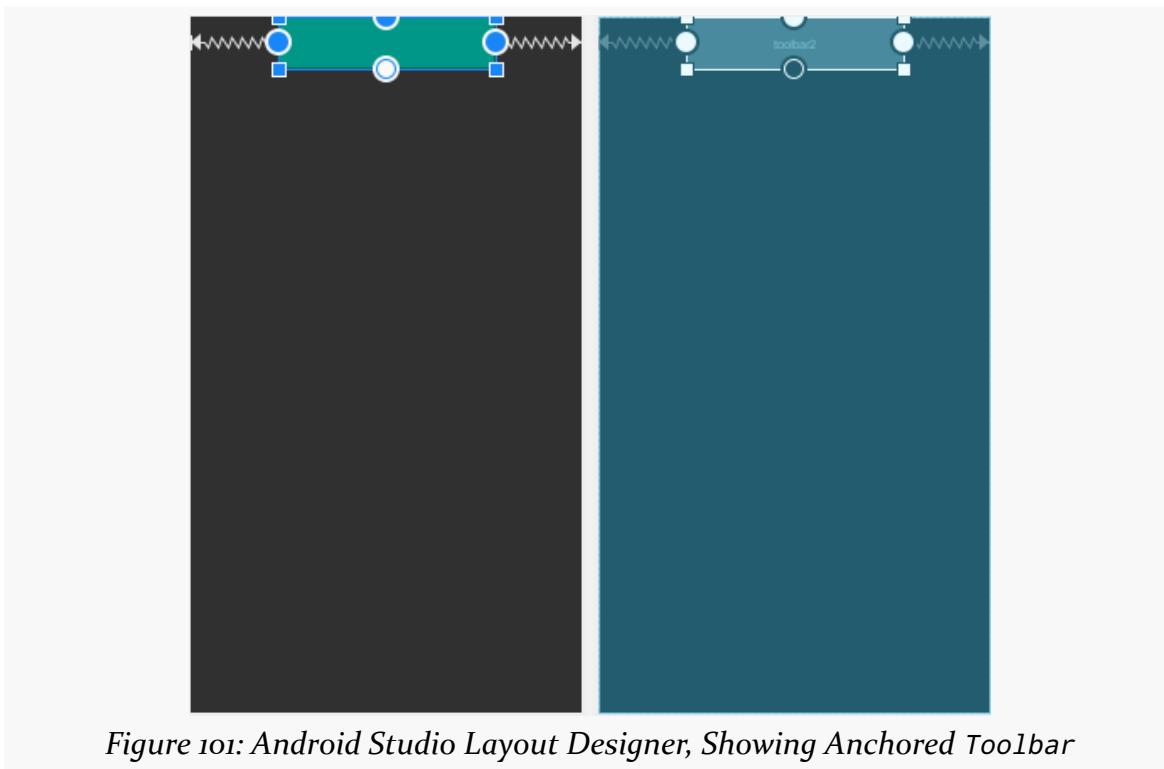


Figure 101: Android Studio Layout Designer, Showing Anchored Toolbar

Then, in the “Attributes” pane, set the `layout_width` to be `match_constraint` (a.k.a., `0dp`) and the `layout_height` to be `wrap_content`.

We want the `id` to be `toolbar`. If it shows up as `toolbar2`, change it to be `toolbar`. |

SETTING UP AN ACTIVITY

Next, switch back to the “Design” view. In the “Palette”, choose the “Widgets” category, and drag a `WebView` into the preview area:

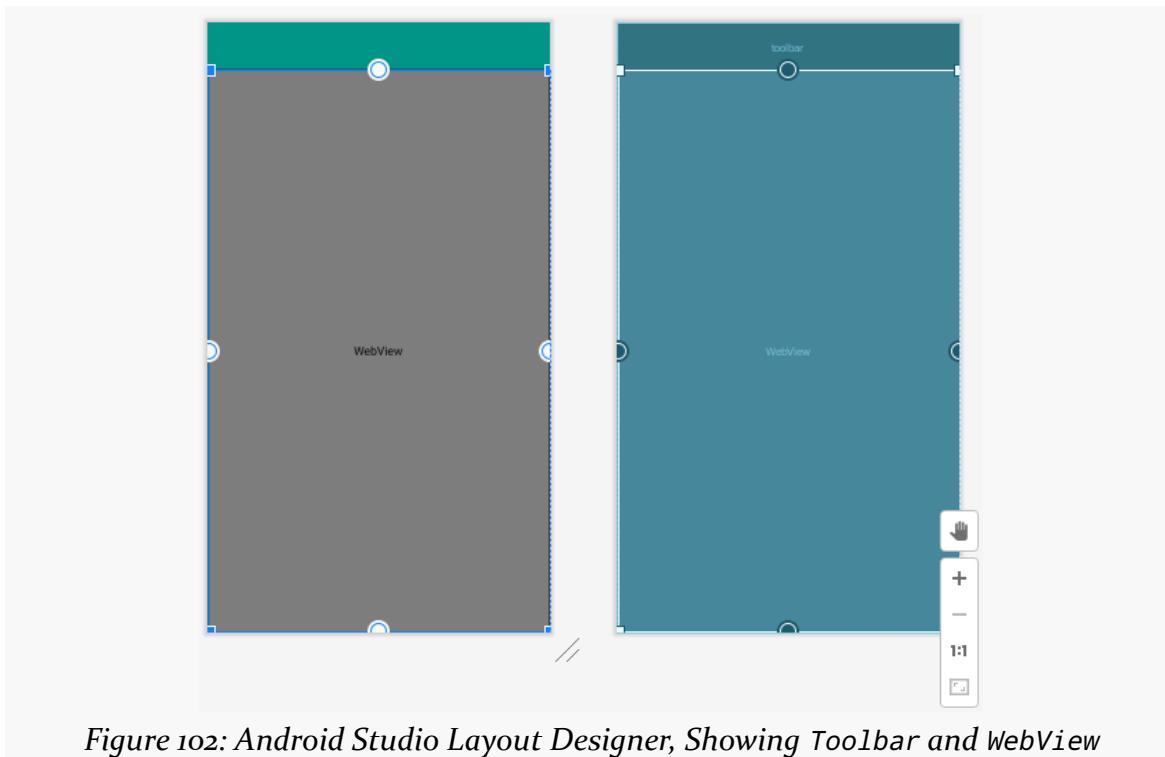


Figure 102: Android Studio Layout Designer, Showing Toolbar and WebView

SETTING UP AN ACTIVITY

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, once again, use the corner handles to resize the `WebView` to be a bit smaller, and drag it away from the edges:

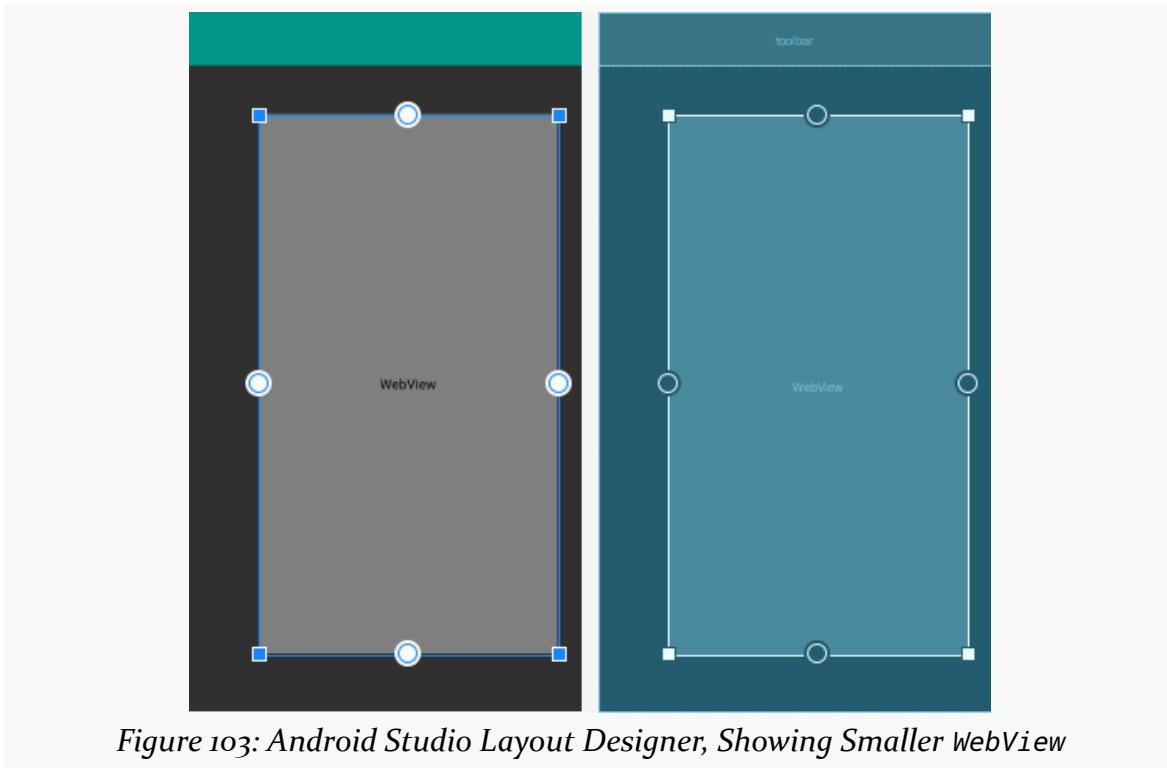


Figure 103: Android Studio Layout Designer, Showing Smaller `WebView`

SETTING UP AN ACTIVITY

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:

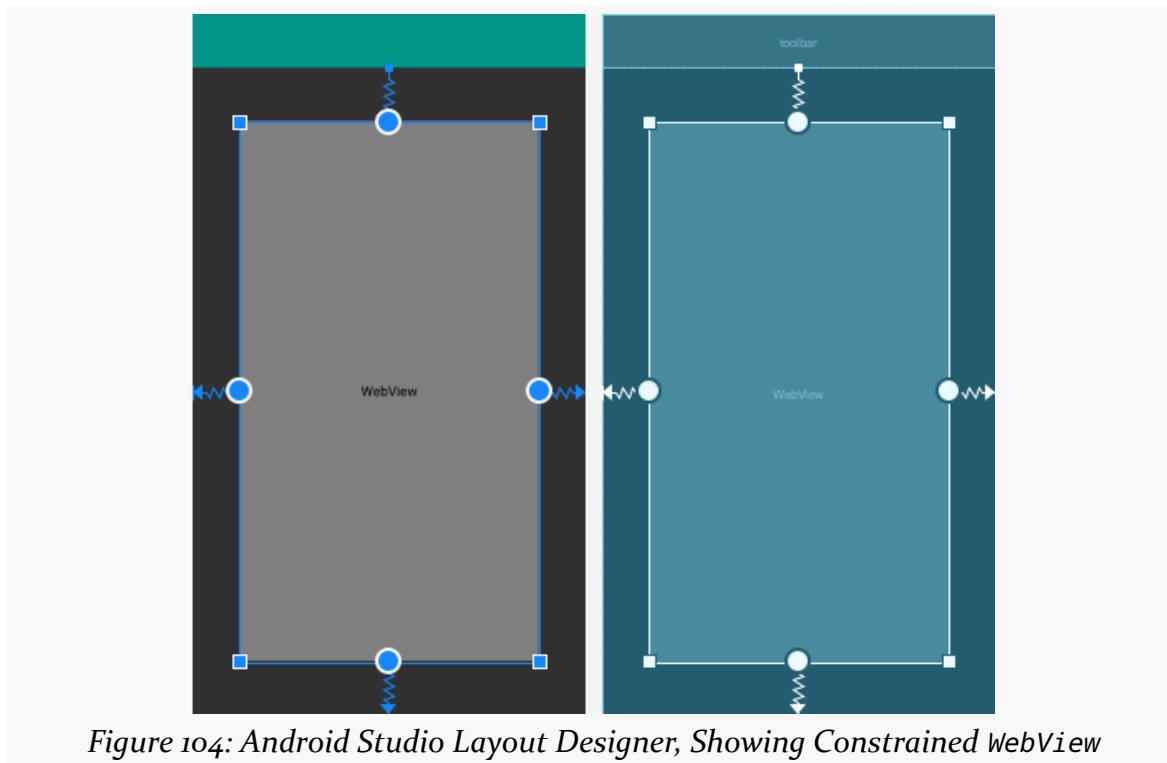


Figure 104: Android Studio Layout Designer, Showing Constrained WebView

SETTING UP AN ACTIVITY

Then, go back to the “Attributes” pane and set the ID to about, the “layout_width” and “layout_height” each to match_constraint (a.k.a., 0dp), to have the WebView fill all of the available space:

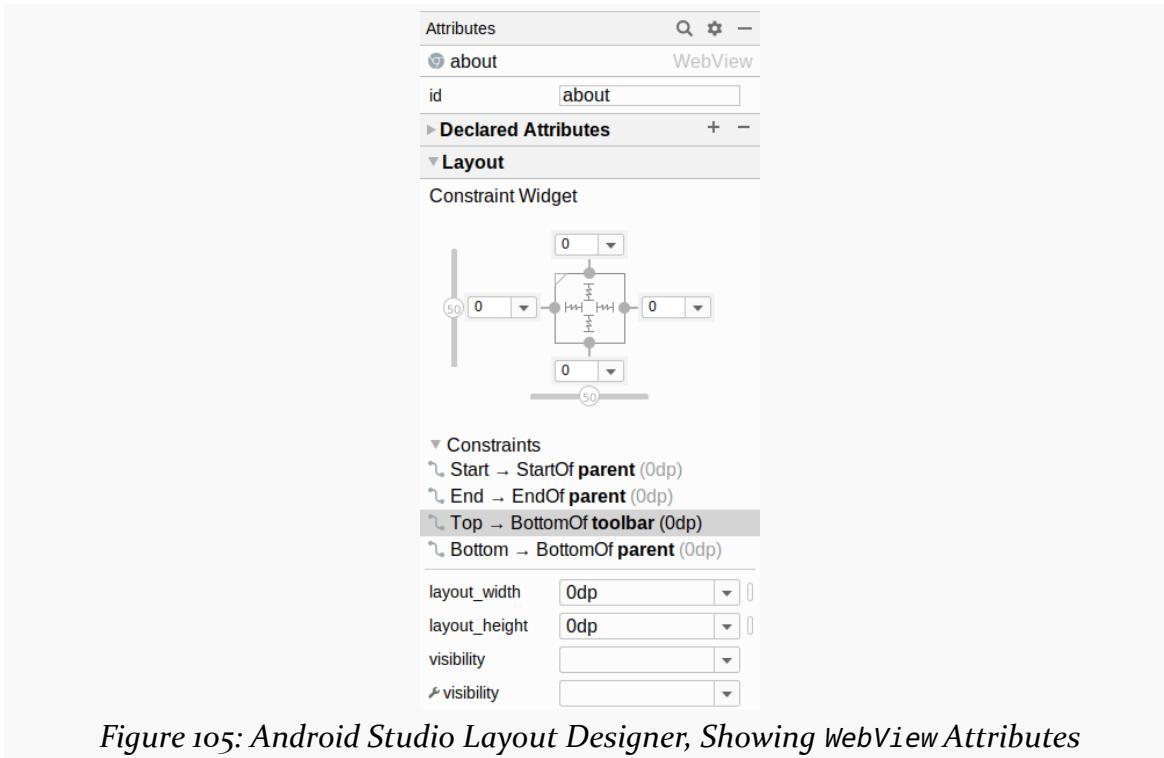


Figure 105: Android Studio Layout Designer, Showing WebView Attributes

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, by adding an `onOptionsItemSelected()` function:

```
override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
    R.id.about -> {
        startActivity(Intent(this, AboutActivity::class.java))
        true
    }
    else -> super.onOptionsItemSelected(item)
}
```

(from [Tio-Activities/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

SETTING UP AN ACTIVITY

`onOptionsItemSelected()` will be called when the user taps on one of the action bar items. We get passed a `MenuItem` identifying the item that the user tapped on, and we can examine its `itemId` value and compare it to the IDs of the items that we put into the menu resource that we used to populate the action bar.

For the `R.id.about` menu item, we create an Intent, pointing at our new `AboutActivity`. Then, we call `startActivity()` on that Intent.

`onOptionsItemSelected()` returns a Boolean: `true` if we handled the event, `false` otherwise. So, in the `R.id.about` branch we return `true`, otherwise we chain to the superclass implementation and return whatever it returns.

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.

Instead of using `startActivity()`, we could have added an `<activity>` element to our navigation graph and then used the Navigation component to start the activity. That has some advantages, but using the Navigation component to start an activity is very modern and not all that common. Using `startActivity()` is far more representative of how existing code with multiple activities starts another activity.

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`.

SETTING UP AN ACTIVITY

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:

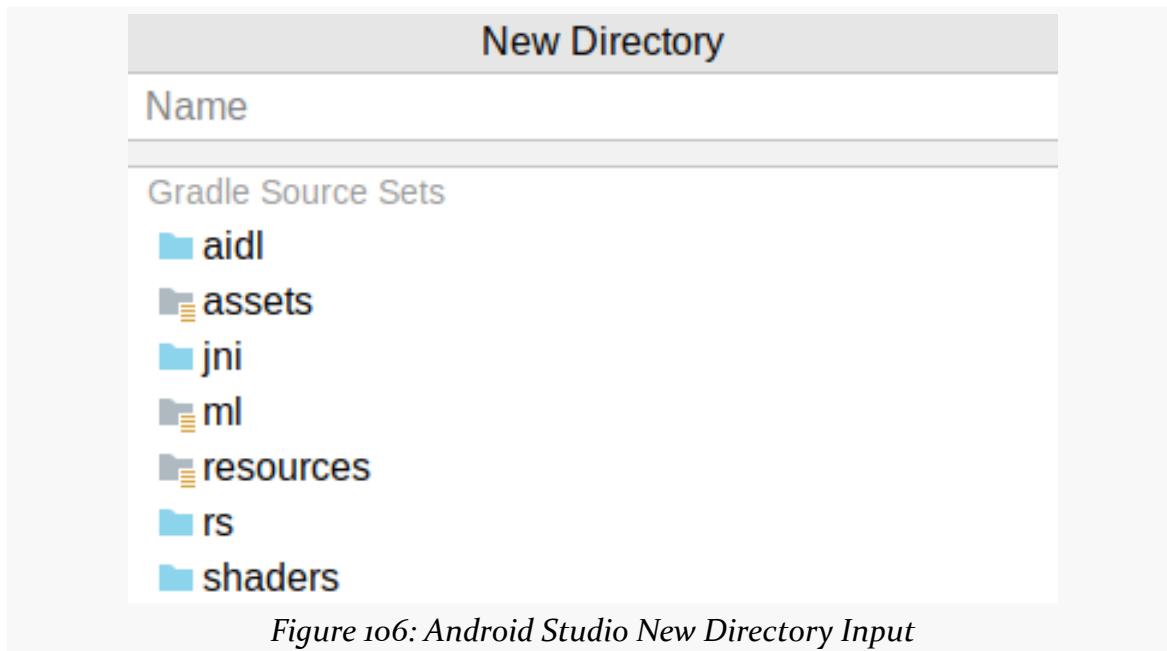


Figure 106: Android Studio New Directory Input

'assets' happens to be one of the pre-defined options. Double-click on that to create an 'assets' directory under 'main/'.

Then, right-click over your new 'assets' directory and choose “New” > “File” from the context menu. Once again, you will get an input area, this time to provide the filename. Fill in 'about.html' and click press **Enter** or **Return** 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 — this app is awesome!</p>
<div>
  .
  <br />
  .

```

SETTING UP AN ACTIVITY

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

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

(from [Tio-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 com.commonsware.todo.databinding.ActivityAboutBinding

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

        val binding = ActivityAboutBinding.inflate(layoutInflater)

        setContentView(binding.root)

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

(from [Tio-Activities/ToDo/app/src/main/java/com/commonsware/todo/AboutActivity.kt](#))

As with MainActivity, we are using view binding in AboutActivity. We create an instance of ActivityAboutBinding using inflate() and use the binding's root for our content view. Then, we configure one item on each widget in the layout:

- We set the title of our Toolbar to be our app's name, which we obtain by calling getString(R.string.app_name) to retrieve the value of the app_name string resource
- We tell the WebView to load our asset

loadUrl() normally takes an https URL, but in this case, we use the special

SETTING UP AN ACTIVITY

`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...)

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

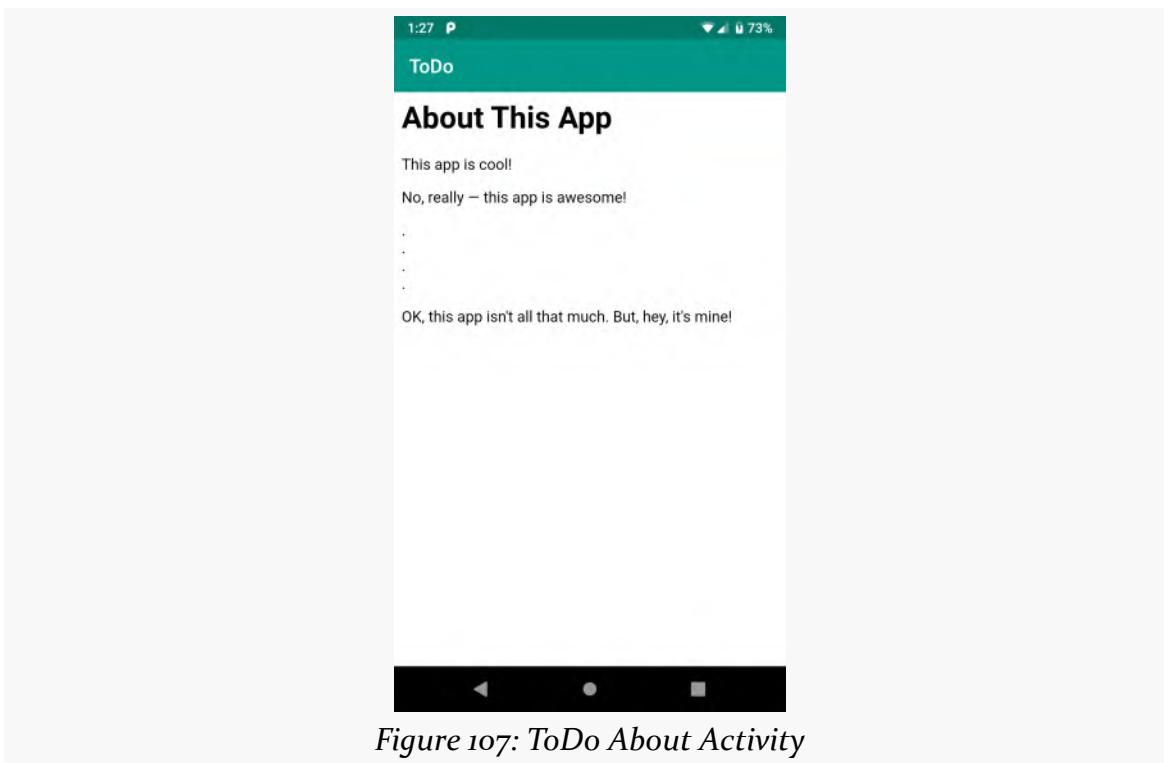


Figure 107: ToDo About Activity

Final Results

The new `res/layout/activity_about.xml` resource 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"
```

SETTING UP AN ACTIVITY

```
    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 [T1o-Activities/ToDo/app/src/main/res/layout/activity_about.xml](#))

MainActivity should now be something like:

```
package com.commonsware.todo

import android.content.Intent
import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import androidx.appcompat.app.AppCompatActivity
import com.commonsware.todo.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {

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

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)
        setSupportActionBar(binding.toolbar)
    }
}
```

SETTING UP AN ACTIVITY

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.actions, menu)

    return super.onCreateOptionsMenu(menu)
}

override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
    R.id.about -> {
        startActivity(Intent(this, AboutActivity::class.java))
        true
    }
    else -> super.onOptionsItemSelected(item)
}
```

(from [Tio-Activities/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

And, as shown above, `AboutActivity` should look like:

```
package com.commonsware.todo

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import com.commonsware.todo.databinding.ActivityAboutBinding

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

        val binding = ActivityAboutBinding.inflate(layoutInflater)

        setContentView(binding.root)

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

(from [Tio-Activities/ToDo/app/src/main/java/com/commonsware/todo/AboutActivity.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/AndroidManifest.xml](#)

SETTING UP AN ACTIVITY

- [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](#)

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 Kotlin 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 GitLab 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.commonware.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 list of available Kotlin structures. Then press `Enter` or `Return` to create this class. `ToDoModel` should show up in an editor, with an implementation like this:

```
package com.commonware.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. data classes with `val` properties are immutable: you do not change a model, but instead

DEFINING A MODEL

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:

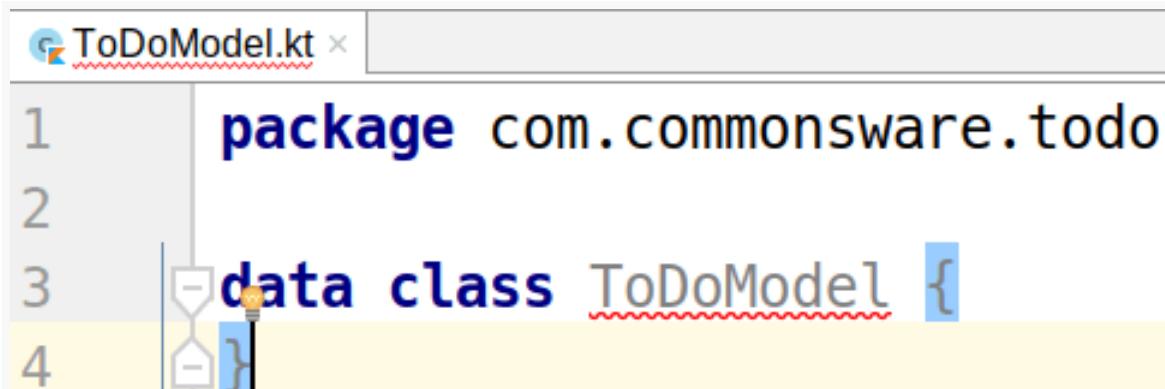


Figure 108: Android Studio, Yelling

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

DEFINING A MODEL

- 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.time.Instant
import java.util.*

data class ToDoModel(
    val description: String,
    val id: String = UUID.randomUUID().toString(),
    val isCompleted: Boolean = false,
    val notes: String = "",
    val createdOn: Instant = Instant.now()
)
```

(from [Tu-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 `Instant` for tracking the created-on time for this to-do item, held in the `createdOn` property

Step #4: Supporting Instant on Older Devices

However, you probably have a new red undersquiggle, this time for the `now()` call on `Instant`. If you hover your mouse over that error, you should see something to the effect of “Call requires API level 26 (current min is 21): `java.time.Instant#now`”.

The `java.time` classes were not added to Android until API Level 26 (Android 8.0). Our project’s `minSdkVersion` is 21 (Android 5.0). The error is pointing out that this code will crash if we try running it on an Android 5.0-7.1 device.

That does not sound good.

DEFINING A MODEL

Fortunately, Google has added a way for us to support Instant and other `java.time` classes on older versions of Android. To do that, we need to make a tweak to `app/build.gradle`.

If you open up that file, you should see lines like these:

```
compileOptions {  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}
```

(from [T10-Activities/ToDo/app/build.gradle](#))

These tell the Android build tools that we are using Java 8 syntax underneath the Kotlin that we are writing.

Add a `coreLibraryDesugaringEnabled true` line to that `compileOptions` closure, giving you:

```
compileOptions {  
    coreLibraryDesugaringEnabled true  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}
```

(from [T11-Model/ToDo/app/build.gradle](#))

Also, in our list of dependencies, add:

```
coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
```

(from [T11-Model/ToDo/app/build.gradle](#))

This is a corresponding library used by this “desugaring” mechanism that supplies implementations of the missing logic on those older devices.

You should have another “Sync Now” banner — go ahead and sync the project with the Gradle files. After that completes, the error for `now()` should be gone.

Final Results

Our new `ToDoModel` should look like:

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

DEFINING A MODEL

```
import java.time.Instant
import java.util.*

data class ToDoModel(
    val description: String,
    val id: String = UUID.randomUUID().toString(),
    val isCompleted: Boolean = false,
    val notes: String = "",
    val createdOn: Instant = Instant.now()
)
```

(from [ToDoModel/ToDo/app/src/main/java/com/commonsware/todo/ToDoModel.kt](#))

And our revised app/build.gradle should resemble:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'androidx.navigation.safeargs.kotlin'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

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

    buildFeatures {
        viewBinding true
    }

    compileOptions {
```

DEFINING A MODEL

```
coreLibraryDesugaringEnabled true
sourceCompatibility JavaVersion.VERSION_1_8
targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation 'com.google.android.material:material:1.4.0'
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

(from [Tu-Model/ToDo/app/build.gradle](#))

What We Changed

The book's GitLab 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](#)
- [app/build.gradle](#)

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.

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 GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding the Repository Class

Once again, we need another Kotlin class.

Right-click over the `com.commonware.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 “Class” from the list of Kotlin structures. Then press `Enter` or `Return` to create this file. `ToDoRepository` should show up in an editor, with an implementation like this:

```
package com.commonware.todo

class 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 we have not built any forms to allow the user to create new to-do items either. 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.commonware.todo

class ToDoRepository {
    var items = listOf(
        ToDoModel(
            description = "Buy a copy of _Exploring Android_",
            isCompleted = true,
            notes = "See https://wares.commonware.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 [T12-Repository/ToDo/app/src/main/java/com/commonware/todo/ToDoRepository.kt](#))

SETTING UP A REPOSITORY

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.

Final Results

`ToDoRepository` should look like:

```
package com.commonsware.todo

class ToDoRepository {
    var 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 [T12-Repository/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

SETTING UP A REPOSITORY

- [app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#)

Inverting Our Dependencies

In general, layers of an app should be loosely coupled.

For example, `ToDoRepository` will be hiding all of the details of exactly where our to-do items get stored. Right now, they are “stored” in memory. Later, they will be stored in a database. They could be stored on a server. And so on. This allows our UI layer to be independent of those storage details.

Our upcoming UI needs access to those to-do items. One approach for this would be to make `ToDoRepository` be a Kotlin object. That is a global singleton, and our activities and fragments could access it as needed.

On the surface, this is fine. This is a fairly simple app. We are not going to be adding smarts to allow users to “plug in” alternative places for storing the to-do items. One `ToDoRepository`, in theory, should be enough.

However, even for a small app like this, that argument starts to break down when it comes to testing. We may need to set up specific test implementations of `ToDoRepository` to test various scenarios, such as what happens when the repository throws an exception (e.g., could not connect to the server). And many apps are much more complicated than this one, where we might really need to have different repository implementations at runtime.

“Dependency inversion” is an approach for dealing with this. In a nutshell, it means that loosely-coupled layers should not be defining the implementations of those other layers. In our app, our activities and fragments should not be declaring that some particular `ToDoRepository` singleton is the one-and-only repository that those fragments should work from. Rather, our fragments should have their repository objects “injected” from outside, so that in the “real app” we can do one thing and in tests we can do something else.

INVERTING OUR DEPENDENCIES

Part of the problem with dependency inversion in Android is that the historically dominant solution — Dagger — is very complex and has difficult-to-understand documentation. While there have been recent moves to simplify it, such as the Jetpack's Hilt library, those are very new and are very much “up for debate” at this time.

Kotlin opened up new opportunities for simplifying dependency inversion. One of the more popular Kotlin dependency inversion libraries is [Koin](#). While it may lack some of the power of Dagger, it is good enough for many apps, including the one that we are building here.



You can learn more about dependency inversion with Koin in the “Inverting Your Dependencies” chapter of [*Elements of Android Jetpack!*](#)

So, in this chapter, we will integrate Koin and set it up that `ToDoRepository` is able to be injected into other objects.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding the Dependencies

There are a couple of new dependencies that we will need to be able to add Koin to the app. And, similar to the Navigation component, we will have dependencies that need to share a common version number. So, we should define that version number in one place, so when we want to upgrade Koin, we can change the version number in that one place and cover everything.

When we created the `nav_version` constant, we did so in the `buildscript` closure of the top-level `build.gradle` file. That is because the Navigation component includes plugins, so we needed the constant in the `buildscript` edition of the dependencies. However, Koin does not have a plugin that we will be using, so we should define this constant outside of `buildscript`, since we do not need it there.

With that in mind, add the following to the bottom of the top-level `build.gradle` file:

INVERTING OUR DEPENDENCIES

```
ext {  
    koin_version = "3.1.2"  
}
```

(from [T13-DI/ToDo/build.gradle](#))

This is equivalent to:

```
ext.koin_version = "3.1.2"
```

The `ext {}` syntax is to simplify matters when we need to define more such `ext` constants, and we will be adding a few more before the tutorials are over.

Then, in the `app/build.gradle` file, add this line to the dependencies closure:

```
implementation "io.insert-koin:koin-android:$koin_version"
```

(from [T13-DI/ToDo/app/build.gradle](#))

Android Studio should be asking you to “Sync Now” in a banner — go ahead and click that link.

Step #2: Creating a Custom Application

We need to configure Koin and teach it what objects we want it to make available to the rest of our app.

In Android, the typical place to configure something like Koin is in a custom Application subclass. The Android framework creates a singleton instance of Application — or of a custom subclass — when your process starts. That Application object will be around for the life of the process. And, it has an `onCreate()` method where we can initialize libraries like Koin.

So, we need another Kotlin class.

Right-click over the `com.commonsware.todo` class where (presently) all of our Kotlin classes reside, and choose “New” > “Kotlin File/Class” from the context menu. Fill in `ToDoApp` for the “Name” and choose “Class” as the kind. Press `Enter` or `Return`, and you will get an empty `ToDoApp` class.

Then, modify it to have it extend from `android.app.Application`:

INVERTING OUR DEPENDENCIES

```
package com.commonsware.todo

import android.app.Application

class ToDoApp : Application() {
```

Next, open up the `AndroidManifest.xml` file. On the `<application>` element, add in an `android:name` attribute:

```
<application
    android:name=".ToDoApp"
    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/Theme.ToDo">
    <activity
        android:name=".AboutActivity"
        android:exported="true" />
    <activity
        android:name=".MainActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

(from [T13-DI/ToDo/app/src/main/AndroidManifest.xml](#))

This tells the Android framework to use our subclass of `Application`, rather than `Application` itself, when it comes time to create this singleton.

Step #3: Defining Our Module

Now we need to teach Koin how to make our `ToDoRepository` available via dependency inversion.

Back in `ToDoApp`, add this property:

INVERTING OUR DEPENDENCIES

```
private val koinModule = module {
    single { ToDoRepository() }
}
```

(from [T13-DI/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Here, `module()` is an extension function supplied by Koin, and it will need to be imported:

```
import org.koin.dsl.module
```

(from [T13-DI/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

`module()` is part of a Koin domain-specific language (DSL) that describes the roster of objects to be available via dependency inversion. An app can have one or several Koin modules — for our purposes, one will be enough.

In that module, `single()` defines an object that will be available as a Koin-managed singleton. In our case, it is an instance of our `ToDoRepository`. The nice thing about Koin — and about dependency inversion frameworks in general — is that a singleton like this can be replaced where needed, such as for testing.

Simply having a Koin module is insufficient — we need to tell Koin about it. To that end, add this `onCreate()` function to `ToDoApp`:

```
override fun onCreate() {
    super.onCreate()

    startKoin {
        androidLogger()
        modules(koinModule)
    }
}
```

(from [T13-DI/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

`startKoin()` and `androidLogger()` are other extension functions that will need to be imported:

```
import org.koin.android.ext.koin.androidLogger
import org.koin.core.context.startKoin
```

(from [T13-DI/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

As the name suggests, `startKoin()` starts the Koin dependency inversion engine. Like `module()`, `startKoin()` has a DSL for configuring Koin. Here, we use two

INVERTING OUR DEPENDENCIES

configuration options, each handled via a function call:

- `androidLogger()`, telling Koin that if it has any messages to log, use Logcat
- `modules()`, where we can provide one or more modules that we want Koin to support (in our case, just the one we declared as `koinModule`)

When we start our app and Android forks a process for us, the framework will create a `ToDoApp` instance for our process and call `onCreate()`. That allows us to set up Koin before any of the rest of our code might need it.

Final Results

Your overall top-level `build.gradle` file should now resemble:

```
buildscript {  
    ext.nav_version = '2.3.5'  
  
    repositories {  
        google()  
        mavenCentral()  
    }  
  
    dependencies {  
        classpath 'com.android.tools.build:gradle:7.0.2'  
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"  
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"  
    }  
}  
  
task clean(type: Delete) {  
    delete rootProject.buildDir  
}  
  
ext {  
    koin_version = "3.1.2"  
}
```

(from [T3-DI/ToDo/build.gradle](#))

Your `app/build.gradle` file should look like:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
    id 'androidx.navigation.safeargs.kotlin'
```

INVERTING OUR DEPENDENCIES

```
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

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

    buildFeatures {
        viewBinding true
    }

    compileOptions {
        coreLibraryDesugaringEnabled true
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    kotlinOptions {
        jvmTarget = '1.8'
    }
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation 'com.google.android.material:material:1.4.0'
    implementation "io.insert-koin:koin-android:$koin_version"
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
```

INVERTING OUR DEPENDENCIES

```
testImplementation 'junit:junit:4.13.2'  
androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
}
```

(from [T13-DI/ToDo/app/build.gradle](#))

The manifest should look something like:

```
<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="com.commonsware.todo">  
  
    <supports-screens  
        android:largeScreens="true"  
        android:normalScreens="true"  
        android:smallScreens="true"  
        android:xlargeScreens="true" />  
  
    <application  
        android:name=".ToDoApp"  
        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/Theme.ToDo">  
        <activity  
            android:name=".AboutActivity"  
            android:exported="true" />  
        <activity  
            android:name=".MainActivity"  
            android:exported="true">  
            <intent-filter>  
                <action android:name="android.intent.action.MAIN" />  
  
                <category android:name="android.intent.category.LAUNCHER" />  
            </intent-filter>  
        </activity>  
    </application>  
  
</manifest>
```

(from [T13-DI/ToDo/app/src/main/AndroidManifest.xml](#))

And, our new ToDoApp should look like:

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

INVERTING OUR DEPENDENCIES

```
import android.app.Application
import org.koin.android.ext.koin.androidLogger
import org.koin.core.context.startKoin
import org.koin.dsl.module

class ToDoApp : Application() {
    private val koinModule = module {
        single { ToDoRepository() }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            modules(koinModule)
        }
    }
}
```

(from [T13-DI/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [build.gradle](#)
- [app/build.gradle](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/AndroidManifest.xml](#)

Incorporating a ViewModel

The Jetpack has a class named `ViewModel`. Its name evokes GUI architecture patterns like Model-View-ViewModel (MVVM). In reality, `ViewModel` and its supporting classes are there to help us with a key challenge in Android: configuration changes.

A configuration change is any change in the device condition where Google thinks that we might want different resources. The most common configuration change is a change in the screen orientation, such as moving from portrait to landscape. We may want different layouts in this case, as our portrait layouts might be too tall for a landscape device, or our landscape layouts might be too wide for a portrait device.

Android's default behavior when a configuration change occurs is to destroy all visible activities and recreate them from scratch, so you can load the desired resources. However, we need some means to hold onto information during this change, so our new activity has access to the same data that our old activity did. There are many solutions to this problem, but a `ViewModel` works fairly nicely, which is why we will use it here.

So, in this tutorial, we will set up a basic `ViewModel` for `RosterListFragment`.



You can learn more about `ViewModel` in the "Integrating `ViewModel`" chapter of [*Elements of Android Jetpack*](#)!

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Creating a Stub ViewModel

So, once again, we create a new Kotlin class. Right-click over the com.commonsware.todo package in the java/ directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in RosterMotor, then choose “Class” for the kind. Then press **Enter** or **Return** to create the empty RosterMotor class.

Then, modify RosterMotor to extend from ViewModel:

```
package com.commonsware.todo

import androidx.lifecycle.ViewModel

class RosterMotor : ViewModel() {
```

Step #2: Getting and Using Our Repository

Ideally, an activity or fragment does not work directly with a repository. Instead, the ViewModel works with the repository, and the activity or fragment work with the ViewModel. The big benefit that we get from a ViewModel is that it is stable across configuration changes, so data that we have retrieved from the repository is not lost when the user rotates the screen and our activity/fragments are destroyed and recreated. Right now, that is not a big benefit, since our model objects are just held in memory. If it took network I/O to get those model objects, though... now caching that data becomes a lot more important. So, we will be switching to having the repository be something the ViewModel talks to.

That implies that RosterMotor will need access to the repository, and that RosterMotor will need to expose an API that our RosterListFragment can use in lieu of the fragment working directly with the repository.

Revise RosterMotor to look like this:

```
package com.commonsware.todo

import androidx.lifecycle.ViewModel

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    val items = repo.items
}
```

INCORPORATING A VIEWMODEL

(from [T14-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/RosterMotor.kt](#))

Here, we get our `ToDoRepository` via the constructor. In our next step, Koin will be supplying our `RosterMotor`, and Koin will be able to give the motor its repository. We also expose the list of items, right now just by having a reference to the repository's list of items. That part will change a lot later on, as we start moving towards having the to-do items in a database, but this will do for now.

Step #3: Depositing a Koin

As was noted earlier, Koin can supply `ViewModel` objects via dependency injection to activities and fragments. However, we have to teach it what `ViewModel` classes are available for injection.

So, in `ToDoApp`, modify the `koinModule` property to add in a `viewModel` line:

```
private val koinModule = module {
    single { ToDoRepository() }
    viewModel { RosterMotor(get()) }
}
```

(from [T14-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

`single()` is a Koin DSL function that says “make a singleton instance of this object available to those needing it”. `viewModel()` is a Koin DSL function that says “use the AndroidX `ViewModel` system to make this `ViewModel` available to those activities and fragments that need it”. There are a few possible Koin `import` statements you could have for `viewModel()` — the one that you want is:

```
import org.koin.androidx.viewmodel.dsl.viewModel
```

(from [T14-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

In our case, we are saying that we are willing to supply instances of `RosterMotor` to interested activities and fragments. To satisfy the `RosterMotor` constructor, we use `get()` to retrieve a `ToDoRepository` from Koin itself. When it comes time to create an instance of `RosterMotor`, Koin will get the `ToDoRepository` singleton and supply it to the `RosterMotor` constructor.

Step #4: Injecting the Motor

Now, we can have `RosterListFragment` use the `RosterMotor`.

INCORPORATING A VIEWMODEL

Add a new motor property:

```
private val motor: RosterMotor by viewModel()
```

(from [T14-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

`viewModel()` is another Koin extension function, one specifically designed to get AndroidX `ViewModel` objects from Koin:

```
import org.koin.androidx.viewmodel.ext.android.viewModel
```

In particular, `viewModel()` will:

- Create a new instance of the `ViewModel` if needed, and
- Will reuse an existing instance of the `ViewModel` if an activity or fragment was destroyed and recreated as part of a configuration change and is now trying to get the `ViewModel` again

Our code does not care which of those scenarios occurs. We know that `motor` will give us our `RosterMotor`, and whether it is a brand-new `RosterMotor` or an existing one from a previous `RosterListFragment` does not matter.

We will start using `motor`, and the associated `ToDoRepository`, in [the next tutorial](#), when we show our list of to-do items on the screen.

Final Results

The modified `app/build.gradle` should resemble:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'androidx.navigation.safeargs.kotlin'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"
```

INCORPORATING A VIEWMODEL

```
    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

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

buildFeatures {
    viewBinding true
}

compileOptions {
    coreLibraryDesugaringEnabled true
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation 'com.google.android.material:material:1.4.0'
    implementation "io.insert-koin:koin-android:$koin_version"
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

(from [T14-ViewModel/ToDo/app/build.gradle](#))

Our new RosterMotor should look like:

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

INCORPORATING A VIEWMODEL

```
import androidx.lifecycle.ViewModel

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    val items = repo.items
}
```

(from [T14-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/RosterMotor.kt](#))

ToDoApp should resemble:

```
package com.commonsware.todo

import android.app.Application
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.dsl.module

class ToDoApp : Application() {
    private val koinModule = module {
        single { ToDoRepository() }
        viewModel { RosterMotor(get()) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            modules(koinModule)
        }
    }
}
```

(from [T14-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

And the altered RosterListFragment should 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
import org.koin.androidx.viewmodel.ext.android.viewModel

class RosterListFragment : Fragment() {
```

INCORPORATING A VIEWMODEL

```
private val motor: RosterMotor by viewModel()

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    return inflater.inflate(R.layout.todo_roster, container, false)
}
```

(from [T14-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/build.gradle](#)
- [app/src/main/java/com/commonsware/todo/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)

Populating Our RecyclerView

We now have a repository with some fake to-do items. It would be helpful if the user could see these items in our `MainActivity` and its `RosterListFragment`. We have a `RecyclerView` in that fragment, and now we need to tie the data from the repository into the `RecyclerView`.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab 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 `RecyclerView` in the "Employing `RecyclerView`" chapter of [*Elements of Android Jetpack!*](#)

Step #1: Defining a Row Layout

Next, we need to define a layout resource to use for the rows in our roster of to-do items.

Right-click over the `res/layout/` directory and choose “New” > “Layout resource file” from the context menu. In the dialog that appears, fill in `todo_row` as the “File name” and ensure that the “Root element” is set to `androidx.constraintlayout.widget.ConstraintLayout`. Then, click “OK” to close the dialog and create the mostly-empty resource file.

We want a `CheckBox` in the rows. We can then arrange to allow the users to mark to-do items as completed by checking the `CheckBox`.

POPULATING OUR RECYCLERVIEW

In Android, a CheckBox widget consists of the actual “box” plus an associated text caption. In principle, we could use that caption to show the description of the to-do item. The downside of this approach is that CheckBox does not distinguish between click events on the box itself and clicks on the caption. Both serve to check (or uncheck) the CheckBox. In many situations, that is fine. In this case, though, we *also* want the user to be able to click on a row in our RecyclerView and be able to navigate to a screen with the full details of this to-do item. Ideally, we would have the user click the caption to navigate to the detail screen, with clicks on the box to check and uncheck it. Unfortunately, CheckBox does not support that.

As a result, what we are going to do is use a CheckBox but leave its caption empty. Instead, we will place a TextView next to the CheckBox and use that for the description. Then, we can distinguish between clicks on the box and clicks on anything else in the row.

With all that in mind... let's start off by setting up the CheckBox.

So, drag a CheckBox from the “Buttons” category in the “Palette” into the preview area:



Figure 109: Android Studio Layout Designer, Showing CheckBox Widget

POPULATING OUR RECYCLERVIEW

Use the round grab handles to drag connections from the CheckBox to the top, bottom, and start sides of the ConstraintLayout:

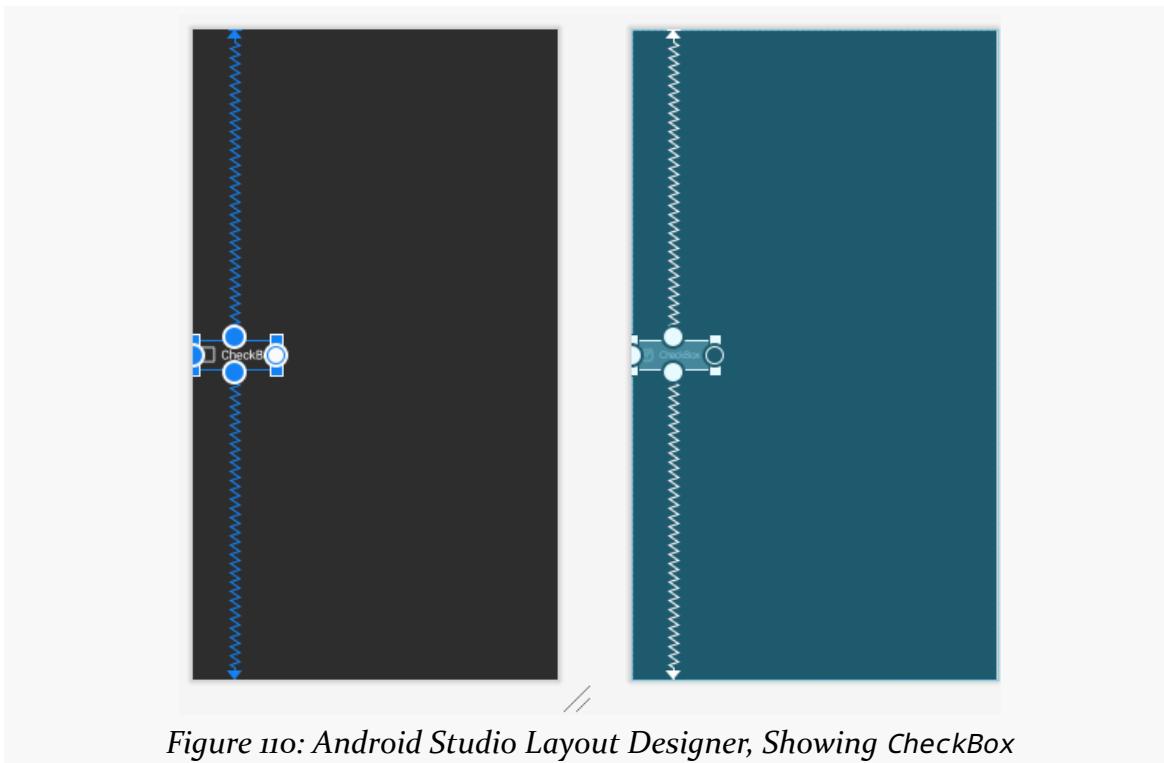
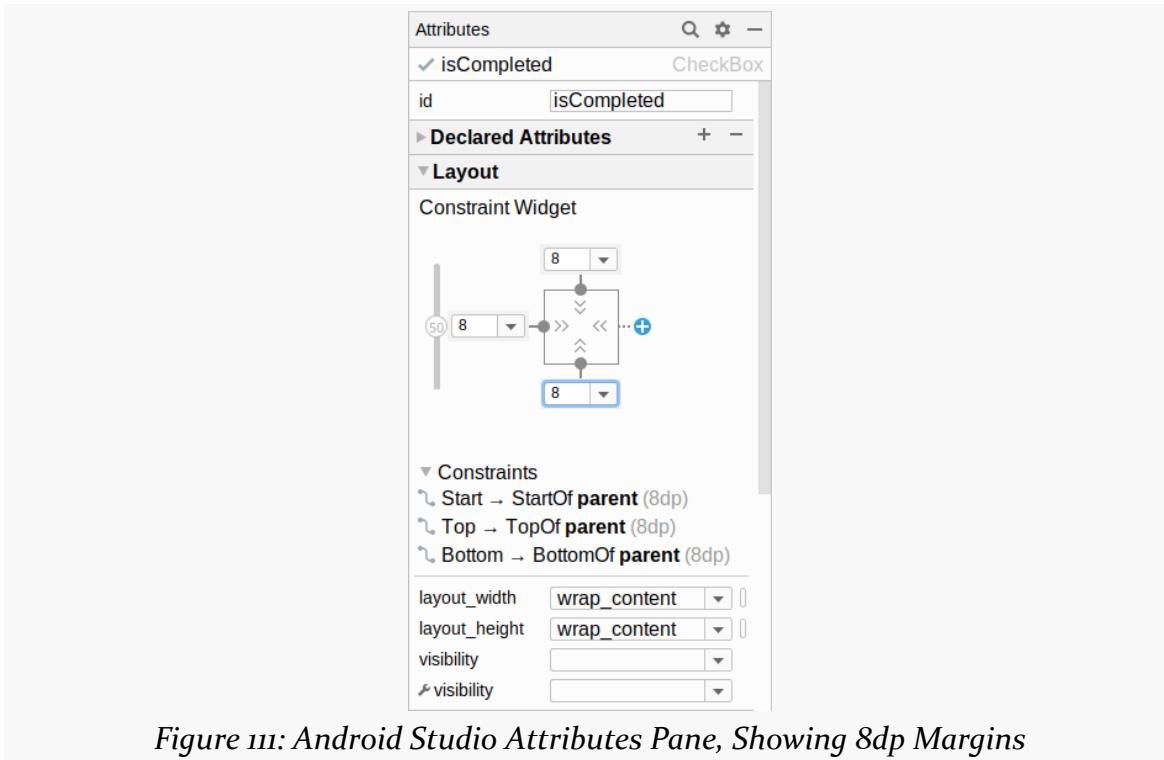


Figure 110: Android Studio Layout Designer, Showing CheckBox

POPULATING OUR RECYCLERVIEW

In the “Attributes” tool, change the “id” to `isCompleted`. Also, in the “Layout” section, change the three drop-downs surrounding the square to be `8dp`, setting margins on those sides:



Also, clear out the “text” attribute, leaving that blank.

Next, from the “Common” category in the “Palette”, drag a `TextView` into the layout. Using the round circles, add constraints from the `TextView` to:

POPULATING OUR RECYCLERVIEW

- the top, bottom, and end edges of the ConstraintLayout, and
- the end side of the CheckBox

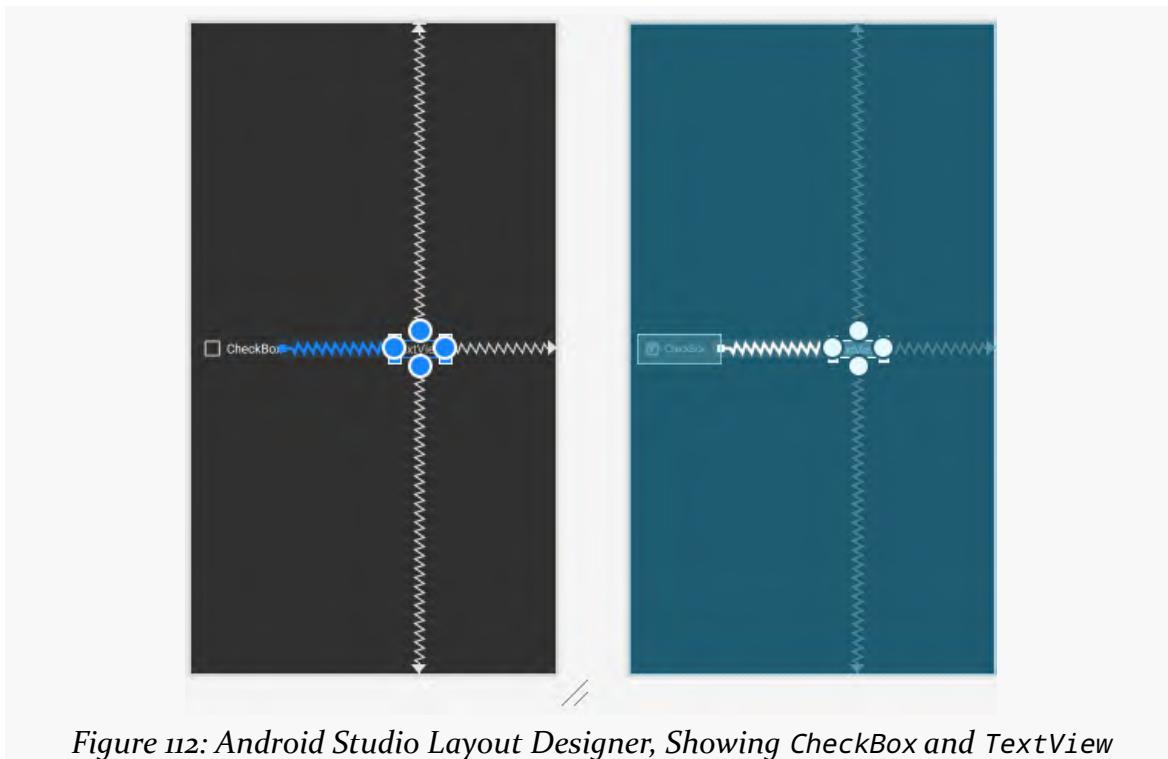


Figure 112: Android Studio Layout Designer, Showing CheckBox and TextView

POPULATING OUR RECYCLERVIEW

In the “Attributes” pane, set the “id” to desc and the “layout_width” to “match_constraint” (a.k.a., 0dp). Also, clear out the “text” attribute, leaving it blank. Then, in the “Layout” section, change the four drop-downs surrounding the square to be 8dp, setting margins on those sides:

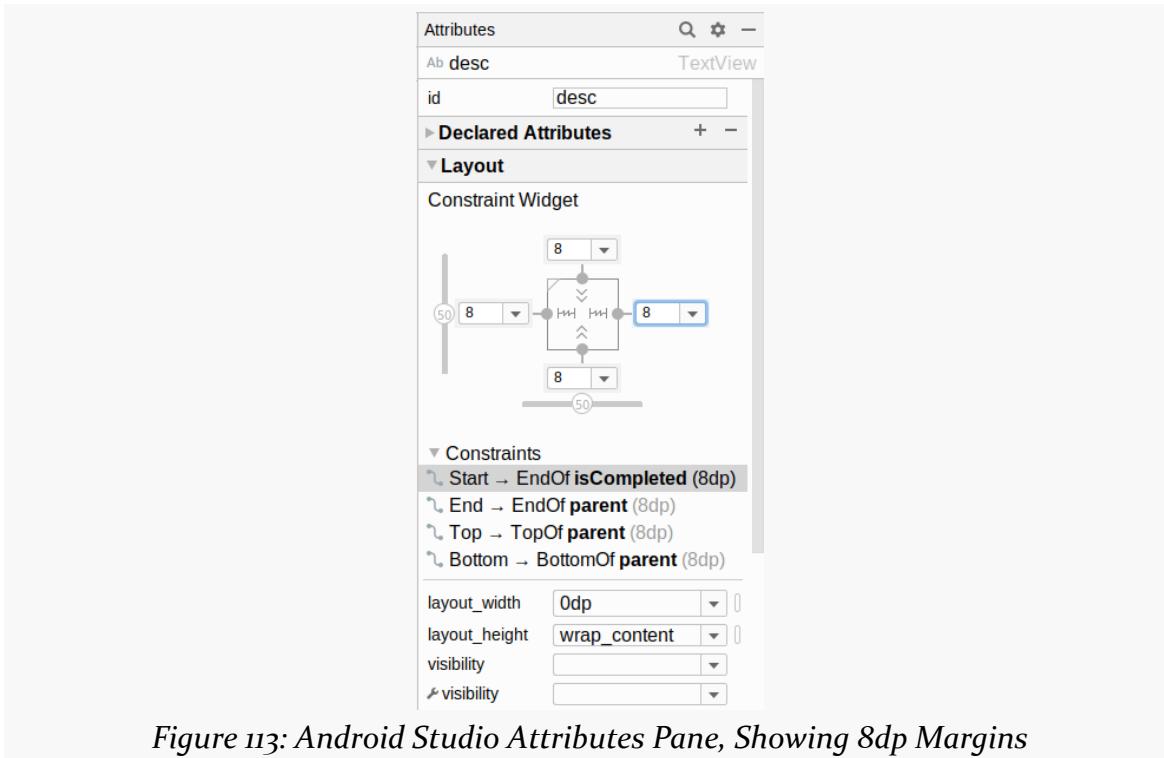


Figure 113: Android Studio Attributes Pane, Showing 8dp Margins

Then, in the “Attributes” pane, fold open the “All Attributes” section. This brings up a *long* list of possible attributes to change. In there, change “ellipsize” to end, by choosing end from the drop-down for that attribute. And, set “maxLines” to 3. This says “show at most 3 lines of text, and if our description is longer than that, truncate the end and show an ellipsis (...) instead”.

Next, select the ConstraintLayout itself in the “Component Tree”. Then, in the “Attributes” pane, set the “layout_height” to be wrap_content. This will keep our rows to be only as tall as is the content in the row.

Finally, switch to the “Code” view in the layout editor and add three attributes to the root <ConstraintLayout> element:

- android:clickable="true", to indicate that this widget represents something that can be clicked

POPULATING OUR RECYCLERVIEW

- `android:focusable="true"`, to indicate that this widget represents something that should be focusable if the user is using arrow keys, a D-pad, or other similar sort of non-touchscreen form of input
- `android:background="?attr/selectableItemBackground"`, to give this widget a background that is the stock background from our theme for things that can be clicked upon

The reason for these three attributes is that rows should be clickable elements, and we want to provide the proper visual response when the user clicks upon them. In Material Design, the standard visual response is a ripple effect in a contrasting color, and `android:background="?attr/selectableItemBackground"` will give that to us automatically for clickable-and-focusable widgets.

At this point, with those manual edits, the `todo_row` layout XML should 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"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:clickable="true"
    android:focusable="true"
    android:background="?attr/selectableItemBackground">

    <CheckBox
        android:id="@+id/isCompleted"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginBottom="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        android:ellipsize="end"
```

POPULATING OUR RECYCLERVIEW

```
    android:maxLines="3"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/isCompleted"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [T15-RecyclerView/ToDo/app/src/main/res/layout/todo_row.xml](#))

Step #2: Adding a Stub ViewHolder

RecyclerView relies upon custom subclasses of RecyclerView.Adapter and RecyclerView.ViewHolder to do “the heavy lifting” of populating its contents. The ViewHolder is responsible for a single item in the RecyclerView, such as a single row in a scrolling list. The Adapter is responsible for creating and populating the ViewHolder instances for each of our model objects, as needed.

So, let’s start by creating a stub subclass of RecyclerView.ViewHolder.

Right-click over the com.commonsware.todo Java package and choose “New” > “Kotlin File/Class” from the context menu. Fill in RosterRowHolder as the “Name” and choose “Class” from the list of Kotlin structures. Then, press **Enter** or **Return** to create a stub Kotlin class.

Then, replace the stub with:

```
package com.commonsware.todo

import androidx.recyclerview.widget.RecyclerView

class RosterRowHolder : RecyclerView.ViewHolder() {
```

This has RosterRowHolder inherit from RecyclerView.ViewHolder.

This will give you an error, complaining that you are not passing a required parameter to the RecyclerView.ViewHolder constructor. We will address that in a later step, so ignore that error for now.

Step #3: Creating a Stub Adapter

A RecyclerView.ViewHolder is managed by a RecyclerView.Adapter. The Adapter

POPULATING OUR RECYCLERVIEW

knows how to create instances of `ViewHolder` and how to populate them with data as the user views items in the list. So, we need a `RecyclerView.Adapter` implementation.

Right-click over the `com.commonware.todo` Java package and choose “New” > “Kotlin File/Class” from the context menu. Fill in `RosterAdapter` as the “Name” and choose “Class” from the list of Kotlin structures. Then, press `Enter` or `Return` to create a stub Kotlin class.

Then, replace the generated contents with:

```
package com.commonware.todo

import androidx.recyclerview.widget.ListAdapter

class RosterAdapter : ListAdapter<ToDoModel, RosterRowHolder>() { }
```

Here, we are using a subclass of `RecyclerView.Adapter` named `ListAdapter`. There are two classes named `ListAdapter` in the Android SDK — be sure that you are using `androidx.recyclerview.widget.ListAdapter`. `ListAdapter` knows how to manage a list of items. In particular, when we replace that list, it knows how to make incremental changes to the `RecyclerView` contents to update it to match the new list. `ListAdapter` takes two data types:

- The type of model data that will be in the list (`ToDoModel`)
- The `RecyclerView.ViewHolder` that will be used for the views (`RosterRowHolder`)

The stub `RosterAdapter` will show two errors. One is that we are not passing a required constructor parameter to `ListAdapter`— we will address that shortly. The other error is that we are missing some functions required by `ListAdapter`, as it is an abstract class.

POPULATING OUR RECYCLERVIEW

To address that bug, with the text cursor in the RosterAdapter name, press **Alt-Enter** (**Option-Return** on macOS) and choose “Implement members” from the quick-fix popup menu:

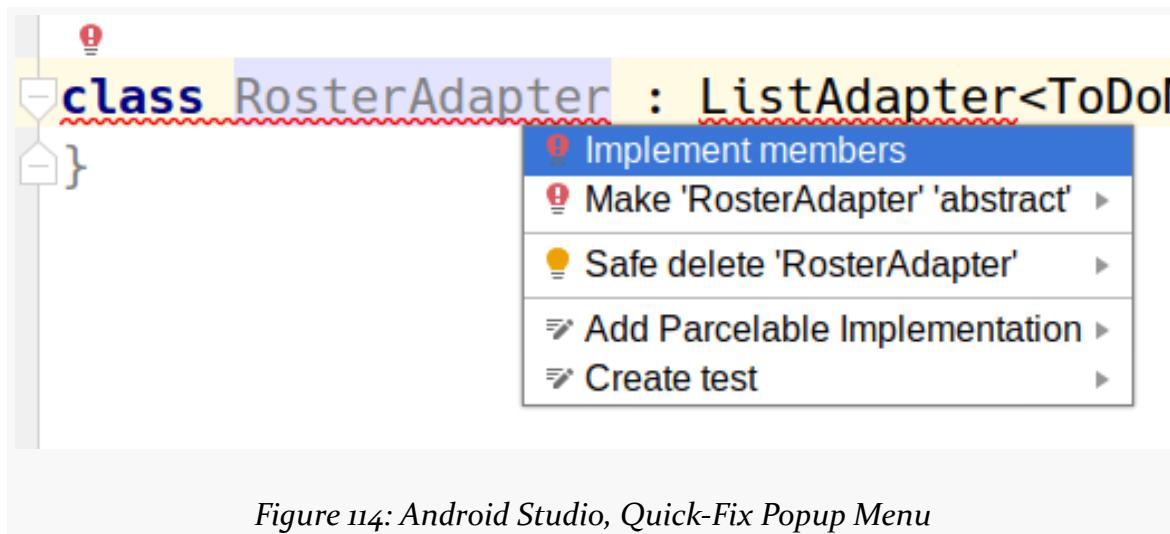


Figure 114: Android Studio, Quick-Fix Popup Menu

This will pop up a dialog box with functions that you can implement:

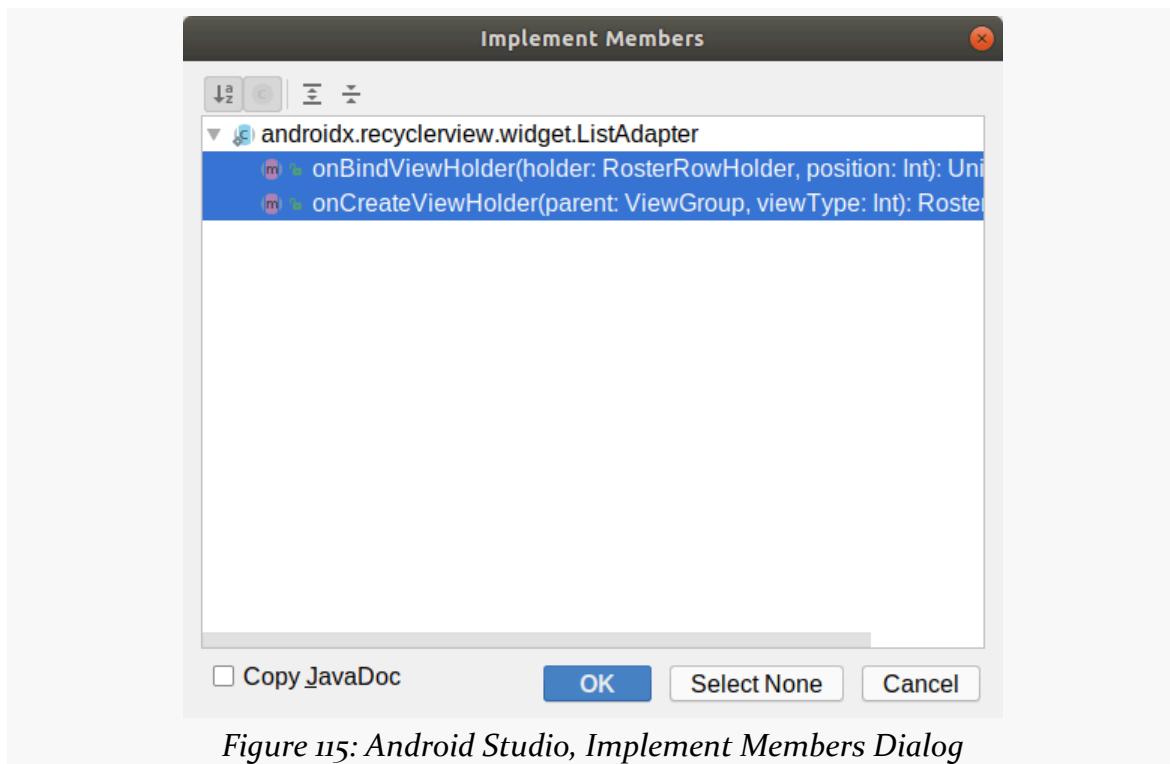


Figure 115: Android Studio, Implement Members Dialog

POPULATING OUR RECYCLERVIEW

Select both functions in the list, then click “OK”.

That will update the Kotlin code to look something like:

```
package com.commonsware.todo

import android.view.ViewGroup
import androidx.recyclerview.widget.ListAdapter

class RosterAdapter : ListAdapter<ToDoModel, RosterRowHolder>() {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): RosterRowHolder {
        TODO("Not yet implemented")
    }

    override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
        TODO("Not yet implemented")
    }
}
```

`onCreateViewHolder()` and `onBindViewHolder()` will need real implementations at some point — we will address that later in this tutorial.

Step #4: Comparing Our Models

The constructor parameter that we are missing to the `ListAdapter` constructor is an instance of the awkwardly-named `DiffUtil.ItemCallback` interface. This interface tells `ListAdapter` how to compare two model objects. In particular, it tells `ListAdapter` whether two model objects should be visually identical, so `RecyclerView` does not have to re-draw or move around views that have not changed their appearance. So, we need an object that can do this for us to provide to `ListAdapter`.

We can take advantage of a couple of Kotlin features as part of this work:

- A Kotlin source file is not limited to a single class, the way Java source files are
- Kotlin has an `object` keyword for creating single instances of objects, for places where we only need one

With that in mind, at the bottom of the `RosterAdapter` Kotlin file, add this:

POPULATING OUR RECYCLERVIEW

```
private object DiffCallback : DiffUtil.ItemCallback<ToDoModel>() {
    override fun areItemsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.id == newItem.id

    override fun areContentsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.isCompleted == newItem.isCompleted &&
            oldItem.description == newItem.description
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

This implements `DiffUtil.ItemCallback` for `ToDoModel`. `areItemsTheSame()` needs to return `true` if the two models are really the same thing — in our case, that would be determined using their unique IDs. `areContentsTheSame()` should return `true` if the two models' visual representations are the same. Our `CheckBox` will use the `description` property for the text and the `isCompleted` property for the checked state, so `areContentsTheSame()` compares those two values. In particular, the `notes` property is ignored for this comparison, since it will not appear in the list rows.

Then, add `DiffCallback` as the missing constructor parameter to `ListAdapter`. This means the entire Kotlin source file at this point should look like:

```
package com.commonsware.todo

import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter

class RosterAdapter : ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): RosterRowHolder {
        TODO("Not yet implemented")
    }

    override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
        TODO("Not yet implemented")
    }
}

private object DiffCallback : DiffUtil.ItemCallback<ToDoModel>() {
    override fun areItemsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.id == newItem.id

    override fun areContentsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
```

POPULATING OUR RECYCLERVIEW

```
    oldItem.isCompleted == newItem.isCompleted &&
        oldItem.description == newItem.description
}
```

Step #5: Completing the Adapter and ViewHolder

Now, we can start filling in the implementations of those stub methods in our `RosterAdapter`, plus get our `RosterRowHolder` working.

The job of `onCreateViewHolder()` is to create instances of a `ViewHolder`, including working with the `ViewHolder` to set up the widgets. Since our widgets are defined in a layout resource, we will need a `LayoutInflater` to accomplish this. The best way to get a `LayoutInflater` is to call `getLayoutInflater()` on an activity or fragment... but `RosterAdapter` has none of these.

So, add a constructor parameter to `RosterAdapter` to take in a `LayoutInflater`:

```
class RosterAdapter(private val inflater: LayoutInflater) :
    ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
```

Then, modify `onCreateViewHolder()` in `RosterAdapter` to be:

```
override fun onCreateViewHolder(
    parent: ViewGroup,
    viewType: Int
) = RosterRowHolder(TodoRowBinding.inflate(inflater, parent, false))
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

Here, we are using `inflate()` on the generated `TodoRowBinding` class to not only inflate the `todo_row` layout, but also to set up the binding object we can use to populate that row's widgets. The particular flavor of `inflate()` that we are calling says:

- Use this `LayoutInflater` to inflate the layout resource (`host.getLayoutInflater()`)
- The widgets in that layout resource eventually will be children of a certain parent (`parent`)...
- ...but do not add them as children right away (`false`)

(some container classes, like `RelativeLayout`, really need to know their parent in order to work properly, so we use this standard recipe for calling `inflate()`)

POPULATING OUR RECYCLERVIEW

However, `onCreateViewHolder()` will have a compile error, as we are passing a constructor parameter to `RosterRowHolder` that does not exist. So, modify `RosterRowHolder` to look like this:

```
package com.commonsware.todo

import androidx.recyclerview.widget.RecyclerView
import com.commonsware.todo.databinding.TodoRowBinding

class RosterRowHolder(private val binding: TodoRowBinding) :
    RecyclerView.ViewHolder(binding.root) {
}
```

`getRoot()` on a binding object returns the root widget of the inflated layout, which in our case is the `ConstraintLayout`. We need to pass that to the `ViewHolder` constructor, so this change fixes that compile error that we had from when we originally set up this class.

Now our `RosterAdapter` knows to create `RosterRowHolder` objects as needed. However, somewhere, we need to get a `ToDoModel` object and fill in the text and is-completed state for the `CheckBox`.

With that in mind, modify `onBindViewHolder()` on `RosterAdapter` to look like this:

```
override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
    holder.bind(getItem(position))
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

`onBindViewHolder()` is called when `RecyclerView` wants us to update a `ViewHolder` to reflect data from some item in the `RecyclerView`. We are given the `position` of that item, along with the `RosterRowHolder` that needs to be updated. Since `RosterAdapter` extends `ListAdapter`, we have a `getItem()` function that gives us our `ToDoModel` for a given position, and we pass that to a `bind()` function on `RosterRowHolder`.

This will have a compile error, as there is no `bind()` function on `RosterRowHolder`. The objective of `bind()` is to populate our widgets, and since we are using the data binding framework, that comes in the form of calling binding methods on the `TodoRowBinding` object.

So, add a `bind()` function to `RosterRowHolder`:

POPULATING OUR RECYCLERVIEW

```
fun bind(model: ToDoModel) {  
    binding.apply {  
        isCompleted.isChecked = model.isCompleted  
        desc.text = model.description  
    }  
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

Here, we use our binding property to access each of our widgets and update their states to match that of the associated ToDoModel.

Step #6: Wiring Up the RecyclerView

Now, we can teach RosterListFragment to use our RosterAdapter.

So far, we have not needed to reference any widgets from our todo_roster layout, so we have a simple use of LayoutInflator in onCreateView() of RosterListFragment. But, we are going to need to start configuring the RecyclerView, so we should switch to view binding for this layout.

First, add a binding property to RosterListFragment:

```
private var binding: TodoRosterBinding? = null
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

This property is nullable, and we initialize it to be null.

Then, change onCreateView() in RosterListFragment to be:

```
override fun onCreateView(  
    inflater: LayoutInflater,  
    container: ViewGroup?,  
    savedInstanceState: Bundle?  
) : View = TodoRosterBinding.inflate(inflater, container, false)  
    .also { binding = it }  
    .root
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Now, we return the results of calling inflate() on the TodoRosterBinding class created from our todo_roster layout resource. And, we use Kotlin's also() scope function to say "in addition, please set binding to be this value", so we can have a

POPULATING OUR RECYCLERVIEW

reference to it later on.

Then, add this onViewCreated() function to RosterListFragment:

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

    val adapter = RosterAdapter(layoutInflater)

    binding?.items?.apply {
        setAdapter(adapter)
        layoutManager = LinearLayoutManager(context)

        addItemDecoration(
            DividerItemDecoration(
                activity,
                DividerItemDecoration.VERTICAL
            )
        )
    }
}

adapter.submitList(motor.items)
binding?.empty?.visibility = View.GONE
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Here we:

- Create a RosterAdapter instance
- Attach that RosterAdapter to our RecyclerView via setAdapter()
- Tell the RecyclerView that it is to be in the form of a vertically-scrolling list, by supplying a LinearLayoutManager to the layoutManager property
- Add divider lines between the rows by creating a DividerItemDecoration and adding it as a decoration to the RecyclerView
- Populate the list by calling submitList() on the RosterAdapter, providing the list of ToDoModel objects that we get from asking our RosterMotor instance for the items
- Hide the empty widget, by setting its visibility to be GONE

Finally, add this onDestroyView() function:

POPULATING OUR RECYCLERVIEW

```
override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

This sets `binding` back to `null`. Ideally, anything that you create in `onCreateView()` and explicitly hold onto, you clean up in `onDestroyView()`, to prevent possible memory leaks. We populated `binding` in `onCreateView()`, so we should clean up `binding` in `onDestroyView()`, so we do not leak the `binding` object after our UI is destroyed.

You can now run the app, and it will show your hard-coded to-do items in the list:

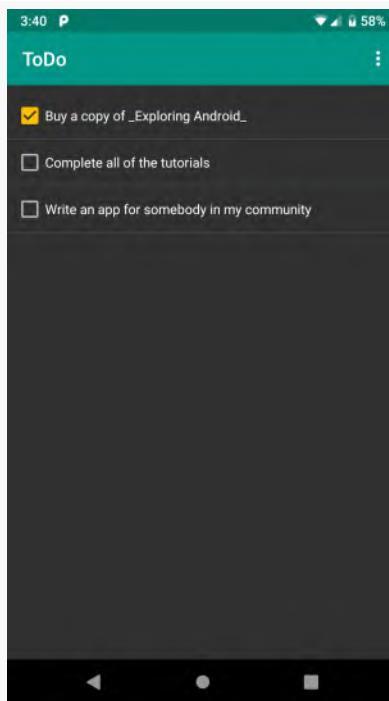


Figure 116: ToDo App, When Launched, Showing Hard-Coded Items

And, if you click on rows in the list (away from the checkboxes), you should see a ripple effect. We got that from those manually-added attributes on the `ConstraintLayout` in the row layout XML. Long-term, that ripple effect will have less of an impact, because we will be launching another screen when the user taps on a row, and that will happen quickly enough that users may not notice the ripple

POPULATING OUR RECYCLERVIEW

effect. But, having that sort of visual cue for clickable widgets is still considered to be a good idea.

Final Results

Our todo_row layout resource should 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"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:clickable="true"
    android:focusable="true"
    android:background="?attr/selectableItemBackground">

    <CheckBox
        android:id="@+id/isCompleted"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginBottom="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <TextView
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        android:ellipsize="end"
        android:maxLines="3"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/isCompleted"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [T15-RecyclerView/ToDo/app/src/main/res/layout/todo_row.xml](#))

POPULATING OUR RECYCLERVIEW

RosterAdapter should look like:

```
package com.commonsware.todo

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import com.commonsware.todo.databinding.TodoRowBinding

class RosterAdapter(private val inflator: LayoutInflater) :
    ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ) = RosterRowHolder(TodoRowBinding.inflate(inflator, parent, false))

    override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
        holder.bind(getItem(position))
    }
}

private object DiffCallback : DiffUtil.ItemCallback<ToDoModel>() {
    override fun areItemsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.id == newItem.id

    override fun areContentsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.isCompleted == newItem.isCompleted &&
        oldItem.description == newItem.description
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

...and RosterRowHolder should look like:

```
package com.commonsware.todo

import androidx.recyclerview.widget.RecyclerView
import com.commonsware.todo.databinding.TodoRowBinding

class RosterRowHolder(private val binding: TodoRowBinding) :
    RecyclerView.ViewHolder(binding.root) {

    fun bind(model: ToDoModel) {
        binding.apply {
            isChecked = model.isCompleted
            desc.text = model.description
        }
    }
}
```

POPULATING OUR RECYCLERVIEW

```
    }
}
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

Finally, RosterListFragment now should 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
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.databinding.TodoRosterBinding
import org.koin.androidx.viewmodel.ext.android.viewModel

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private var binding: TodoRosterBinding? = null

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root

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

        val adapter = RosterAdapter(layoutInflater)

        binding?.items?.apply {
            setAdapter(adapter)
            layoutManager = LinearLayoutManager(context)

            addItemDecoration(
                DividerItemDecoration(
                    activity,
                    DividerItemDecoration.VERTICAL
                )
            )
        }
    }
}
```

POPULATING OUR RECYCLERVIEW

```
        adapter.submitList(motor.items)
        binding?.empty?.visibility = View.GONE
    }

    override fun onDestroyView() {
        binding = null

        super.onDestroyView()
    }
}
```

(from [T15-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/res/layout/todo_row.xml](#)
- [app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)

Tracking the Completion Status

We have checkboxes in the list to show the completion status. However, the user can toggle these checkboxes. Right now, that is only affecting the UI — our models still have the old data. We should find out when the user toggles the checked state of a checkbox, then update the associated model to match. So, that is what we will work on in this tutorial.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Registering for Events

We need to register an `OnCheckedChangeListener` on the `CheckBox` in each row, so we find out when that checkbox is checked and unchecked. We also will need to know which `ToDoModel` is bound to this row, so we can update the correct model.

To that end, modify `bind()` in `RosterRowHolder` to look like:

```
fun bind(model: ToDoModel) {
    binding.apply {
        isCompleted.isChecked = model.isCompleted
        isCompleted.setOnCheckedChangeListener { _, _ -> TODO() }
        desc.text = model.description
    }
}
```

We now have attached a lambda expression to our `CheckBox` check events. Of note:

- The first parameter to the lambda expression is the `CheckBox` that was

TRACKING THE COMPLETION STATUS

checked. We do not need this, so we use `_` to indicate that this is an unused parameter. Similarly, the second parameter to the lambda expression is a Boolean indicating the current state of the CheckBox. We do not need that either (it is the opposite of the current state of our model), so we use `_` as the parameter name for it as well.

- `TODO()` is a Kotlin function that serves as a marker, indicating that we are not yet done with this logic.

If you were to run this and try checking or unchecking a CheckBox, your app will crash, as `TODO()` throws an exception as a way of getting the point across that you are not done yet.

(hence, do not run the app right now, unless you like crashing)

Step #2: Passing the Event Up the Chain

Eventually, we need to modify our repository to reflect the change in the model state.

We could have the `RosterRowHolder` do that. However, it is best to minimize the number of places that you are modifying your data. The more your model-manipulating code is scattered, the more difficult it will be to change that code, such as when we want to start storing this stuff in a database. Since we already are working with the repository in `RosterMotor`, we may as well have it handle the model modifications as well.

However, our `RosterRowHolder` does not have access to the `RosterMotor`. We could pass it down from the `RosterListFragment` if we wanted. Alternatively, we can pass the event up the Kotlin object hierarchy, from the `RosterRowHolder` through the `RosterAdapter` to the `RosterListFragment`, and from there affect our `RosterMotor`.

With that in mind, modify the constructor of `RosterRowHolder` to look like:

```
class RosterRowHolder(  
    private val binding: TodoRowBinding,  
    val onCheckboxToggle: (ToDoModel) -> Unit  
) :
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

Here, `onCheckboxToggle` is a function type. We are passing some sort of function or lambda expression into `RosterRowHolder` that takes a `ToDoModel` as input and

TRACKING THE COMPLETION STATUS

returns nothing (i.e., `Unit`, roughly analogous to `void` in Java).

Then, revise `bind()` on `RosterRowHolder` to look like:

```
fun bind(model: ToDoModel) {
    binding.apply {
        isCompleted.isChecked = model.isCompleted
        isCompleted.setOnCheckedChangeListener { _, _ -> onCheckboxToggle(model) }
        desc.text = model.description
    }
}
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

Now, our `setOnCheckedChangeListener()` calls the `onCheckboxToggle` function type, passing in the current model. This replaces the `TODO()` that we used in the previous step.

Now, though, `RosterAdapter` will have a compile error, as we are not passing in this value. So, add a similar constructor parameter to `RosterAdapter`:

```
class RosterAdapter(
    private val inflater: LayoutInflater,
    private val onCheckboxToggle: (ToDoModel) -> Unit
) :
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

Then, pass `onCheckboxToggle` to the `RosterRowHolder` constructor call in `onCreateViewHolder()`:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
    RosterRowHolder(
        TodoRowBinding.inflate(inflater, parent, false),
        onCheckboxToggle
    )
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

So now we are passing the `onCheckboxToggle` that the `RosterAdapter` receives to each of the `RosterRowHolder` instances. This, though, has broken `RosterListFragment`, as it is not passing a value for `onCheckboxToggle` in its `RosterAdapter` constructor call.

To fix this, modify the creation of the `RosterAdapter` instance in `RosterListFragment` to look like:

TRACKING THE COMPLETION STATUS

```
val adapter = RosterAdapter(layoutInflater) { model ->
    TODO()
}
```

When a function type is the last parameter for a function call, we can use a lambda expression outside of the function call parentheses. So, the lambda expression that we have here turns into onCheckboxToggle.

Right now, we have a TODO() function call in the lambda expression. So, if you run the app and you click on one of the CheckBox widgets... you crash with a error of:

```
kotlin.NotImplementedError: An operation is not implemented.
```

However, this is an expected crash. NotImplementedError is what TODO() throws. So, this confirms that we got control in RosterListFragment when the user clicked the CheckBox. Now, we need to replace the TODO() with something more useful and less crash-prone.

Step #3: Saving the Change

Now, we need to update our repository, given that the user toggled the completed state of the model.

To that end, add a save() function to ToDoRepository:

```
fun save(model: ToDoModel) {
    items = if (items.any { it.id == model.id }) {
        items.map { if (it.id == model.id) model else it }
    } else {
        items + model
    }
}
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

Here, we see if the items list already contains the supplied model, based on the id values. If it does, this means we are replacing an existing ToDoModel with an updated copy, so we generate a new list of models, replacing the old one with the new one via map(). If, however, the list does not contain a model with this id, then we must be adding some new model to the list, so we just create a list that adds the new model to the end.

Next, add a similar save() function to RosterMotor:

TRACKING THE COMPLETION STATUS

```
fun save(model: ToDoModel) {  
    repo.save(model)  
}
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterMotor.kt](#))

Right now, all this does is call through to `save()` on the repository. Later, when we have to start taking database I/O and threading into account, `save()` will do more work. But, for now, this is all that we need.

Then, we can call `save()` on `RosterMotor` from our `onCheckedChange` lambda expression, over in `RosterListFragment`:

```
val adapter = RosterAdapter(layoutInflater) {  
    motor.save(it.copy(isCompleted = !it.isCompleted))  
}
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Here, we create the updated model by using `copy()`, a function added to all Kotlin data classes. As the name suggests, `copy()` makes a copy of the immutable object, except it replaces whatever properties we include as parameters to the `copy()` call. In our case, we replace `isCompleted` with the opposite of its current value.

If you run this revised sample... nothing much seems to change, except that our `TODO()` crash is gone. You cannot readily see the objects in the repository, to see what the `ToDoModel` objects look like. Plus, we are only affecting memory, so these changes go away when the app's process does. All of those limitations will be addressed in upcoming tutorials.

Final Results

`RosterRowHolder` should resemble:

```
package com.commonsware.todo  
  
import androidx.recyclerview.widget.RecyclerView  
import com.commonsware.todo.databinding.TodoRowBinding  
  
class RosterRowHolder(  
    private val binding: TodoRowBinding,  
    val onCheckboxToggle: (ToDoModel) -> Unit  
) :  
    RecyclerView.ViewHolder(binding.root) {
```

TRACKING THE COMPLETION STATUS

```
fun bind(model: ToDoModel) {
    binding.apply {
        isCompleted.isChecked = model.isCompleted
        isCompleted.setOnCheckedChangeListener { _, _ -> onCheckboxToggle(model) }
        desc.text = model.description
    }
}
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

RosterAdapter should look like:

```
package com.commonsware.todo

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import com.commonsware.todo.databinding.TodoRowBinding

class RosterAdapter(
    private val inflater: LayoutInflater,
    private val onCheckboxToggle: (ToDoModel) -> Unit
) : ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        RosterRowHolder(
            TodoRowBinding.inflate(inflater, parent, false),
            onCheckboxToggle
        )

    override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
        holder.bind(getItem(position))
    }
}

private object DiffCallback : DiffUtil.ItemCallback<ToDoModel>() {
    override fun areItemsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.id == newItem.id

    override fun areContentsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.isCompleted == newItem.isCompleted &&
            oldItem.description == newItem.description
}
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

TRACKING THE COMPLETION STATUS

`ToDoRepository` should resemble:

```
package com.commonsware.todo

class ToDoRepository {
    var 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!"
        )
    )

    fun save(model: ToDoModel) {
        items = if (items.any { it.id == model.id }) {
            items.map { if (it.id == model.id) model else it }
        } else {
            items + model
        }
    }
}
```

(from [Ti6-Completion/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

`RosterMotor` should have:

```
package com.commonsware.todo

import androidx.lifecycle.ViewModel

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    val items = repo.items

    fun save(model: ToDoModel) {
        repo.save(model)
    }
}
```

(from [Ti6-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterMotor.kt](#))

TRACKING THE COMPLETION STATUS

And RosterListFragment should contain:

```
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
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.databinding.TodoRosterBinding
import org.koin.androidx.viewmodel.ext.android.viewModel

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private var binding: TodoRosterBinding? = null

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root

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

        val adapter = RosterAdapter(layoutInflater) {
            motor.save(it.copy(isCompleted = !it.isCompleted))
        }

        binding?.items?.apply {
            setAdapter(adapter)
            layoutManager = LinearLayoutManager(context)

            addItemDecoration(
                DividerItemDecoration(
                    activity,
                    DividerItemDecoration.VERTICAL
                )
            )
        }
    }

    adapter.submitList(motor.items)
    binding?.empty?.visibility = View.GONE
}
```

TRACKING THE COMPLETION STATUS

```
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

}
```

(from [T16-Completion/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)

Displaying an Item

We are storing things, like notes, in the `ToDoModel` that do not appear in the roster list. That sort of list usually shows limited information, with the rest of the details shown when you tap on an item in the list. That is the approach that we will use here, where we will show a separate fragment with the details of the to-do item when the user taps on the item.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Creating the Fragment

Once again, we need to set up a fragment.

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. For the name, fill in `DisplayFragment`. For the kind, choose “Class”. Press `Enter` or `Return` to create the class.

That will give you a `DisplayFragment` that looks like:

```
package com.commonsware.todo

class DisplayFragment {
```

Then, have it extend the `Fragment` class:

DISPLAYING AN ITEM

```
package com.commonsware.todo

import androidx.fragment.app.Fragment

class DisplayFragment : Fragment() {
```

Step #2: Updating the Navigation Graph

We need to add this new fragment to our navigation graph, so we can navigate to it.

Open up the `res/navigation/nav_graph.xml` resource. In the graphical design view, once again click the new-destination toolbar button (rectangle with green plus). This will drop down a list of candidate fragments, and `DisplayFragment` should be among them:

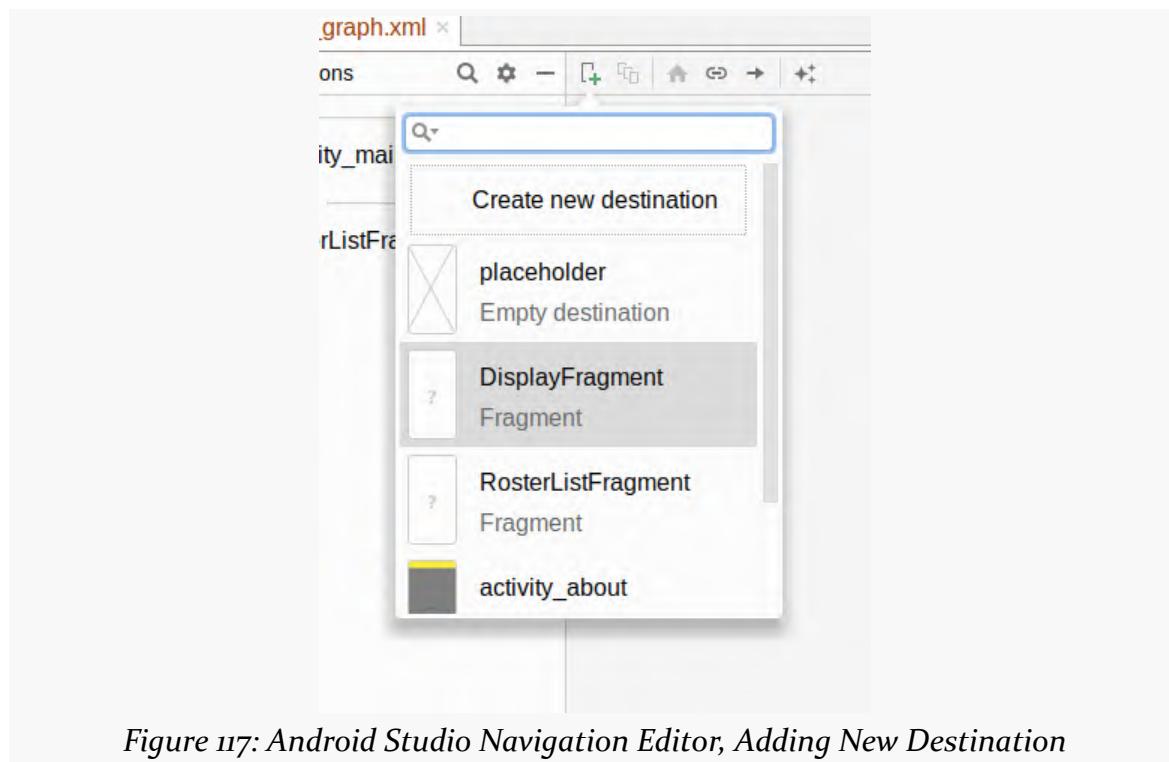


Figure 117: Android Studio Navigation Editor, Adding New Destination

DISPLAYING AN ITEM

Choose `DisplayFragment`, and you should see it appear in your editor:

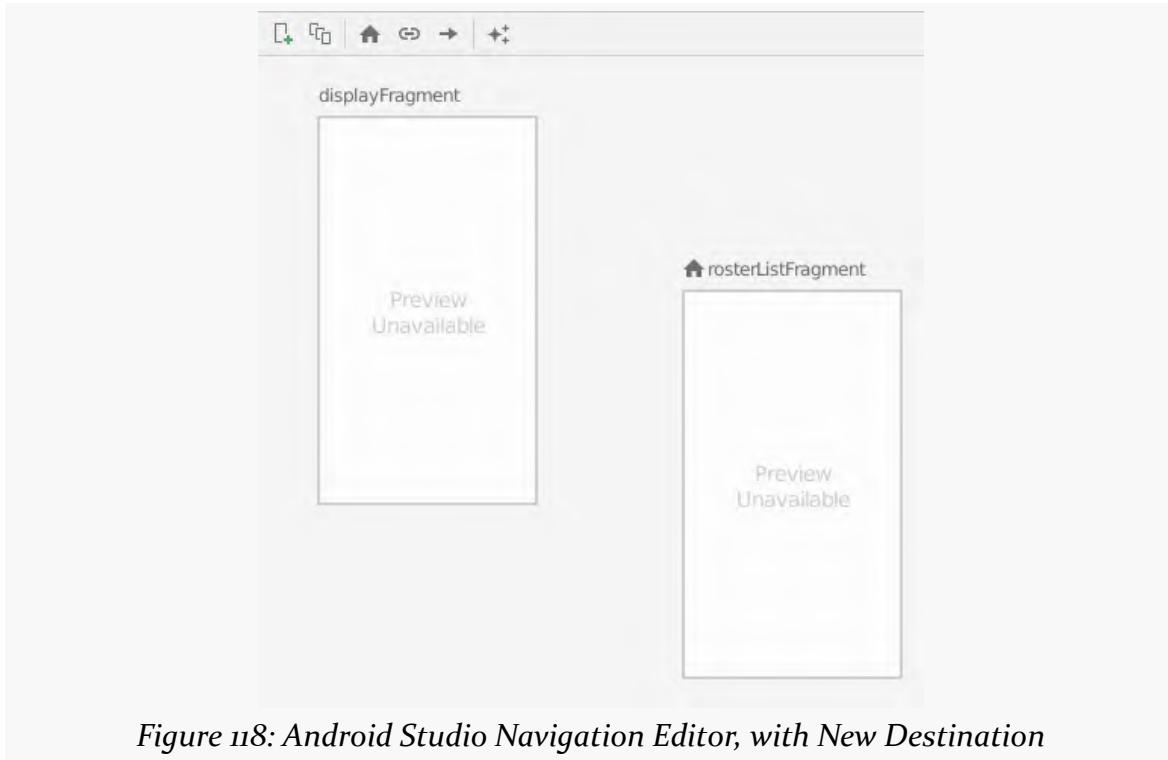


Figure 118: Android Studio Navigation Editor, with New Destination

DISPLAYING AN ITEM

If, as in the screenshot shown above, `displayFragment` shows up to the left of `rosterListFragment`, drag it more to the right:

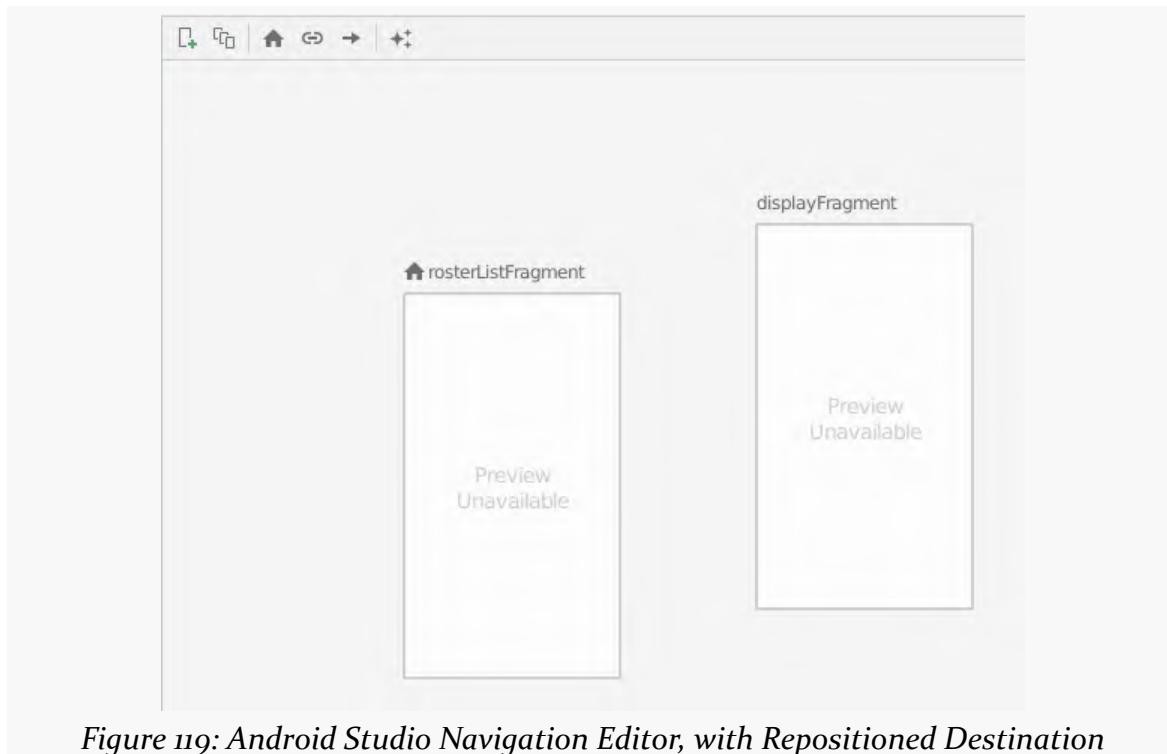


Figure 119: Android Studio Navigation Editor, with Repositioned Destination

DISPLAYING AN ITEM

Then, click on `rosterListFragment`, and drag the circle that appears on the right over to `displayFragment`, creating an arrow:

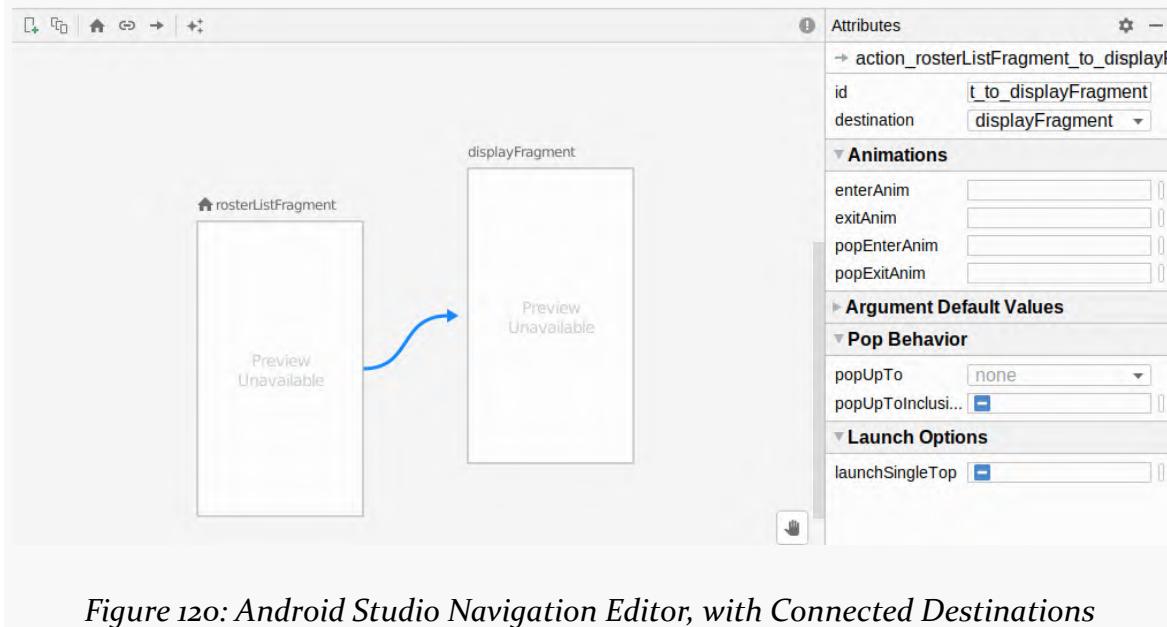


Figure 120: Android Studio Navigation Editor, with Connected Destinations

This sets up an “action”. It indicates that we want to be able to navigate from our `RosterListFragment` to our `DisplayFragment`.

Now, we need to adjust some attributes of our navigation graph.

Click on the `rosterListFragment`, so its attributes appear in the “Attributes” pane. Replace its current “Label” with `@string/app_name`, so we use our existing app name string resource.

Then, click on `displayFragment` and do the same thing: replace its current “Label” with `@string/app_name`.

Finally, click on the arrow connecting the two destinations. This allows us to modify attributes of the action itself. It has a generated ID of `@+id/action_rosterListFragment_to_displayFragment`, which is fine but rather long. Change it to `@+id/displayModel`.

At this point, the XML of the navigation resource should look like:

DISPLAYING AN ITEM

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph.xml"
    app:startDestination="@+id/rosterListFragment">

    <fragment
        android:id="@+id/rosterListFragment"
        android:name="com.commonware.todo.RosterListFragment"
        android:label="@string/app_name">
        <action
            android:id="@+id/displayModel"
            app:destination="@+id/displayFragment" />
    </fragment>
    <fragment
        android:id="@+id/displayFragment"
        android:name="com.commonware.todo.DisplayFragment"
        android:label="@string/app_name" />
</navigation>
```

Step #3: Responding to List Clicks

The typical way lists work in Android is that if you tap on a row in the list, the user is taken to some UI (activity or fragment) that pertains to the tapped-upon row. More generally, we have ways of finding out when the row gets clicked upon. Once we get control when the user taps on a row, we need to show that `DisplayFragment`... even if at the moment it will be empty.

At this point, we have two separate events to track: clicking on the row and toggling the checked state of the `CheckBox`. We can set up separate callback functions for those.

To that end, modify the constructor for `RosterRowHolder` to have both `onCheckboxToggle` and `onRowClick` parameters:

```
class RosterRowHolder(
    private val binding: TodoRowBinding,
    val onCheckboxToggle: (ToDoModel) -> Unit,
    val onRowClick: (ToDoModel) -> Unit
) :
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonware/todo/RosterRowHolder.kt](#))

Then, update `bind()` to add a new line to the `apply()` call:

DISPLAYING AN ITEM

```
fun bind(model: ToDoModel) {
    binding.apply {
        root.setOnClickListener { onRowClick(model) }
        isCompleted.isChecked = model.isCompleted
        isCompleted.setOnCheckedChangeListener { _, _ -> onCheckboxToggle(model) }
        desc.text = model.description
    }
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

Click events that are not handled by child widgets ripple up the view hierarchy. So, by putting a click listener on the ConstraintLayout — the root of our binding — we will find out if the user clicks on the TextView or any empty space in the row. And, here, we invoke our onRowClick function type. At this point, we are getting control for both UI events and routing them to the appropriate function types.

At this point, though, RosterAdapter will have a compile error, as we are not passing in both onRowClick and onCheckboxToggle. So, modify its constructor to accept both of those:

```
class RosterAdapter(
    private val inflater: LayoutInflater,
    private val onCheckboxToggle: (ToDoModel) -> Unit,
    private val onRowClick: (ToDoModel) -> Unit
) :
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

...and modify onCreateViewHolder() to pass both along to the RosterRowHolder constructor:

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
    RosterRowHolder(
        TodoRowBinding.inflate(inflater, parent, false),
        onCheckboxToggle,
        onRowClick
    )
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

Now, of course, RosterListFragment has a compile error, as we are not providing both onRowClick and onCheckboxToggle. We need to provide the onRowClick value and do something to show the DisplayFragment.

With that in mind, add the following function to RosterListFragment:

DISPLAYING AN ITEM

```
private fun display(model: ToDoModel) {
    findNavController().navigate(RosterListFragmentDirections.displayModel())
}
```

Our primary gateway to the Navigation component from our Kotlin code is by `findNavController()`. We can call this on any activity or fragment and get a `NavController` back that we can use to navigate through our navigation graph, among other things.

Here, we call `navigate()` on the `NavController`, to indicate that we want to transition from this destination to another one in our navigation graph. Because we added the Safe Args plugin back in [the preceding tutorial](#), our parameter to `navigate()` is a “directions” object. We get ours by calling `displayModel()` on `RosterListFragmentDirections`. The `...Directions` class will have a name based on the destination that we are coming from — we are in the `rosterListFragment` destination in our navigation graph, so our class is `RosterListFragmentDirections`. The function that we call is based on the ID of the action that we want to perform. We gave our action an ID of `displayModel` earlier in this tutorial, so our function is `displayModel()`. We pass whatever that function returns to `navigate()`, and the Navigation component will arrange to perform that action and send us to whatever destination it designates.

In the IDE, you will see that our `model` parameter to our `display()` function is unused. This hints at a flaw in our implementation. The idea is that we want `DisplayFragment` to display the details of some to-do item. That implies that `DisplayFragment` knows what that to-do item actually is. We have the `model` parameter, but we are not somehow getting that information over to the `DisplayFragment`. We will address this shortcoming in a later step of this tutorial.

So, modify the `RosterAdapter` constructor call in `onViewCreated()` of `RosterListFragment` to look like this:

```
container: ViewGroup?,
savedInstanceState: Bundle?
): View = TodoRosterBinding.inflate(inflater, container, false)
    .also { binding = it }
    .root

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

    val adapter = RosterAdapter(
        layoutInflater,
        onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
        onRowClick = ::display)
```

DISPLAYING AN ITEM

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Here, to help us keep track of our parameters, we are taking advantage of Kotlin's named arguments. We have our original lambda expression assigned to `onCheckboxToggle`, and we have a function reference to our `display()` function assigned to `onRowClick`.

If you run the sample app now, and you click on one of the to-do items, you will be taken to the `DisplayFragment`... which happens to not display anything. We will fix that in the upcoming steps of this tutorial.

If you press BACK when viewing the (empty) `DisplayFragment`, you will return to the list of to-do items.

Step #4: Teaching Navigation About the App Bar

Google, however, is not happy.

Google wants apps to have "up navigation". This involves having a back arrow visible in the Toolbar when the user is somewhere deeper in the navigation graph than the start destination, such as being on our `DisplayFragment`:

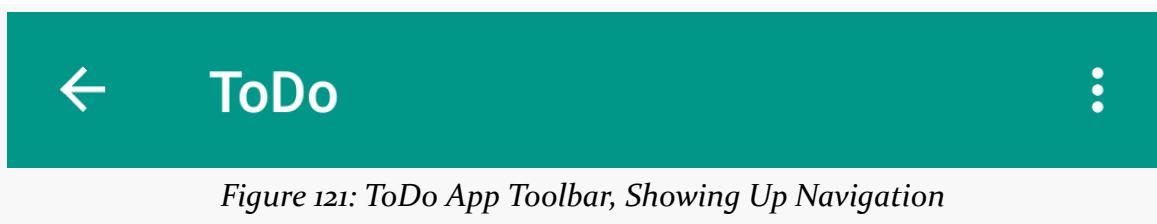


Figure 121: ToDo App Toolbar, Showing Up Navigation

(if you are wondering why this is called "up navigation" when the arrow points to the side... well, that's complicated...)

To make Google happy, we need to add a few things to our `MainActivity`, so that the Navigation component knows about the app bar and can add the up navigation icon when needed.

First, in `MainActivity`, add this property:

```
private lateinit var appBarConfiguration: AppBarConfiguration
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

DISPLAYING AN ITEM

An `AppBarConfiguration` is... a configuration for our app bar. In this app, our Toolbar will serve as our app bar.

(if you are wondering why we have action bars, toolbars, and app bars... well, that's complicated...)

Next, add this block of code to the end of the `onCreate()` function in `MainActivity`:

```
supportFragmentManager.findFragmentById(R.id.nav_host)?.findNavController()?.let { nav ->
    appBarConfiguration = AppBarConfiguration(nav.graph)
    setupActionBarWithNavController(nav, appBarConfiguration)
}
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

To get our `NavController`, we need to call `findNavController()` on something. Unfortunately, [Google made that relatively complex](#) when you are using `FragmentContainerView` for these fragments. We need to:

- Get a `FragmentManager` by referencing `supportFragmentManager`
- Find our `NavHostFragment` in there by calling `findFragmentById()` and passing in the ID of our `FragmentContainerView` (`R.id.nav_host`)
- Call `findNavController()` on that Fragment

We then:

- Create an `AppBarConfiguration` for the navigation graph managed by that `NavController`
- Call `setupActionBarWithNavController()`, to tell the Navigation component that we want it to automatically update the app bar based on our navigation through our navigation graph

Finally, add this function to `MainActivity`:

```
override fun onSupportNavigateUp() =
    navigateUp(findNavController(R.id.nav_host), appBarConfiguration)
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

When the user taps that arrow in the app bar, this function will be called. Here, we just pass that event along to the Navigation component, asking it to perform whatever would be appropriate for up navigation at this point. Here, it is safe for us to just use `findNavController()`. The issues requiring us to go through the extra complexity to call `findNavController()` up in `onCreate()` are tied to that code

DISPLAYING AN ITEM

being in `onCreate()`, before the Navigation system has gotten everything wired up. By the time `onSupportNavigateUp()` is called, Navigation is fully initialized and we can just use the simpler straightforward `findNavController()` call.

If you run the app, not only will that arrow appear in the app bar when we are viewing the `DisplayFragment`, but tapping it will return you to the `RosterListFragment`

Step #5: Creating an Empty Layout

To have `DisplayFragment` display the contents of a `ToDoModel`, it helps to have a layout resource.

Right-click over the `res/layout/` directory and choose “New” > “Layout resource file” from the context menu. In the dialog that appears, fill in `todo_display` as the “File name” and ensure that the “Root element” is set to `androidx.constraintlayout.widget.ConstraintLayout`. Then, click “OK” to close the dialog and create the mostly-empty resource file.

Then, we are going to need to describe the UI that we want, setting up our widgets to display the different pieces of data in our `ToDoModel`. We will do that over the next several tutorial steps.

Step #6: Adding the Completed Icon

Part of what we need to display is whether or not this to-do item is completed. In the roster rows, that was handled by the `CheckBox`. However, a `CheckBox` is a widget designed for input. We have two choices:

1. We could use a `CheckBox` and allow the user to change the completion status from within the `DisplayFragment`
2. We could use something else to represent the current completion status, restricting the user to changing the status somewhere else

Since we are also going to allow the user to change the completion status from the fragment that allows editing of the whole `ToDoModel`, it seems reasonable to make `DisplayFragment` be display-only. We could still use a `CheckBox` and simply ignore any changes that the user makes in it, but that's just rude.

Instead, let's use an `ImageView` to display an icon for completed items, hiding it for

DISPLAYING AN ITEM

items that are not completed.

To do this, first we should set up the icon artwork. Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard, that we used when creating the app bar item [in an earlier tutorial](#).

There, click the “Clip Art” button and search for check:

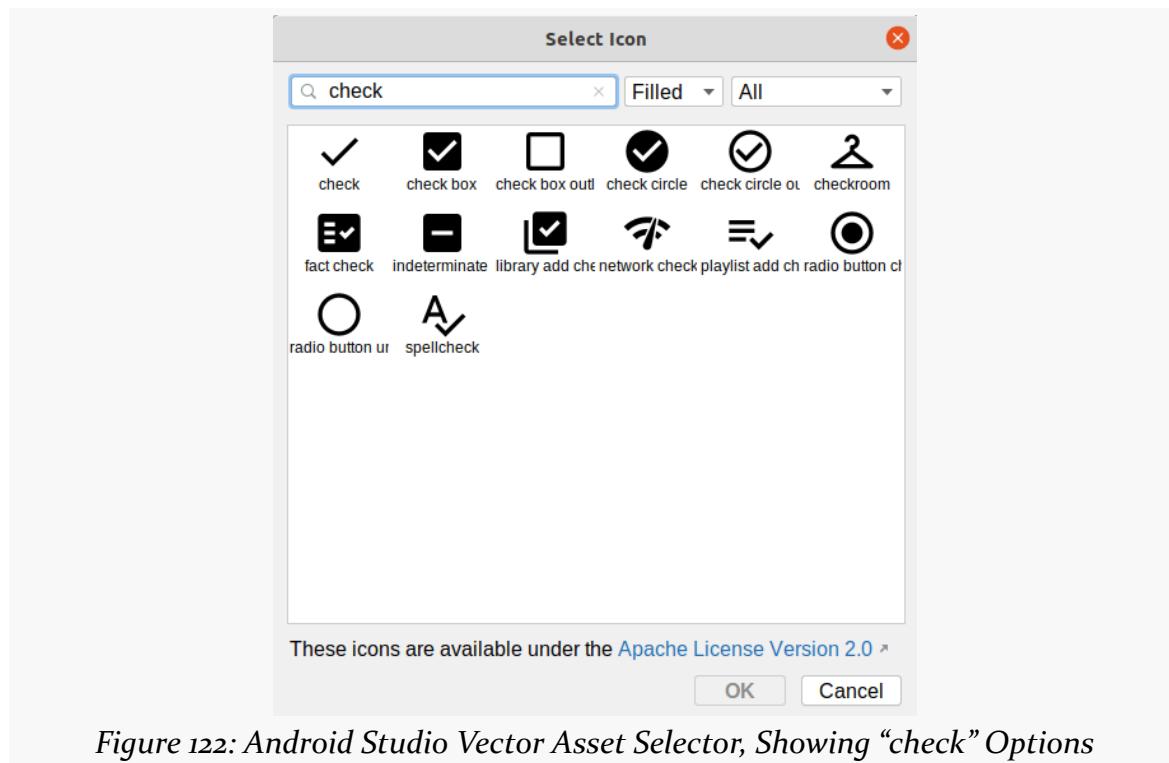


Figure 122: Android Studio Vector Asset Selector, Showing “check” Options

Choose the “check circle” icon and click “OK” to close up the icon selector. Then, change the name to `ic_check_circle`. Finally, click “Next” and “Finish” to close up the wizard and set up our icon.

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

DISPLAYING AN ITEM

Then, in the graphical designer for `todo_display.xml`, drag an `ImageView` from the “Common” category of the “Palette” into the layout. This immediately displays a dialog from which you can choose a resource to display:

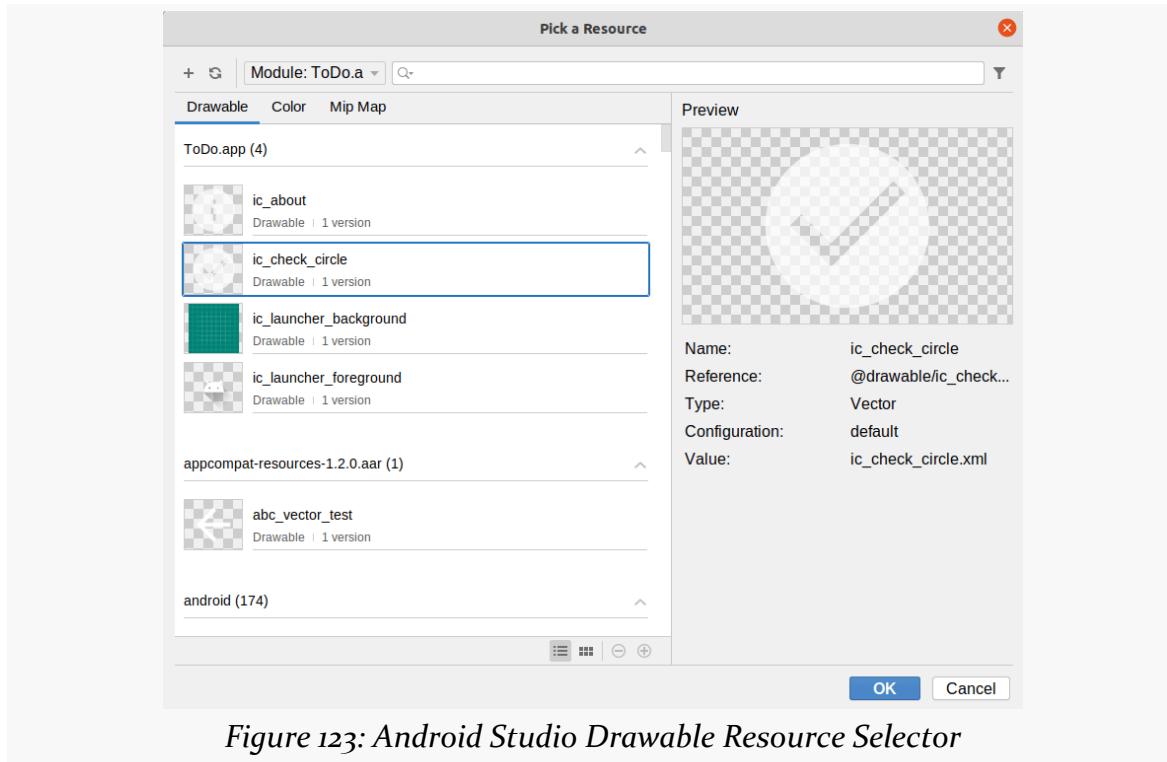


Figure 123: Android Studio Drawable Resource Selector

In the “app” section, choose the `ic_check_circle` resource that we just created, then click “OK” to close up the dialog.

Add constraints to tie the top and end side of the `ImageView` to the top and end side of the `ConstraintLayout`. Then, in the Attributes pane, give the `ImageView` an “ID” of `completed`. And, in the “Layout” section, give it 8dp of top and end margin.

The icon is a bit small by default, at only 24dp. We can make it bigger by changing its width and height. We want it to be square, and we might want the size to be different on larger-screen devices. So, we should set up a dimension resource for a larger size, then apply it to both the width and the height.

DISPLAYING AN ITEM

To do that, click the “O” button to the right of the “layout_width” drop-down in the Attributes pane:



Figure 124: Android Studio Layout Designer, Showing Tiny “O” Button

That brings up a dialog to choose a dimension resource:

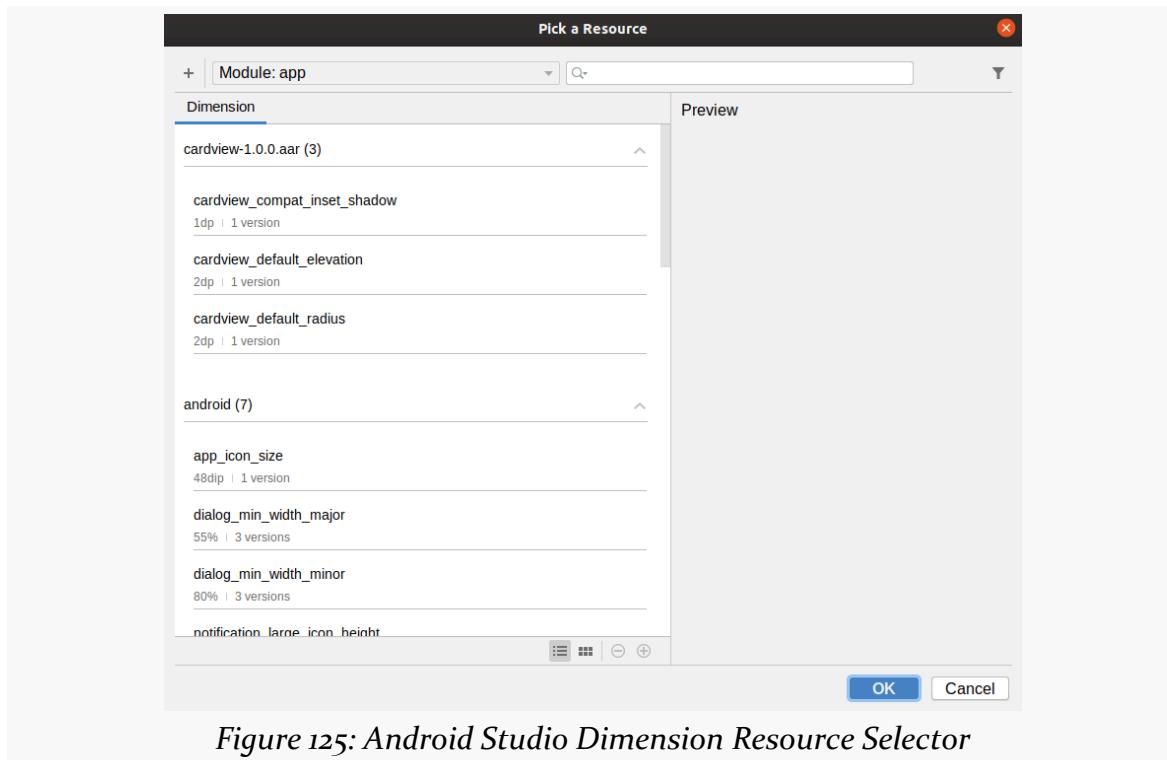


Figure 125: Android Studio Dimension Resource Selector

DISPLAYING AN ITEM

Click the + icon and choose “Dimension Value...” from the drop-down menu to bring up the new-resource dialog:

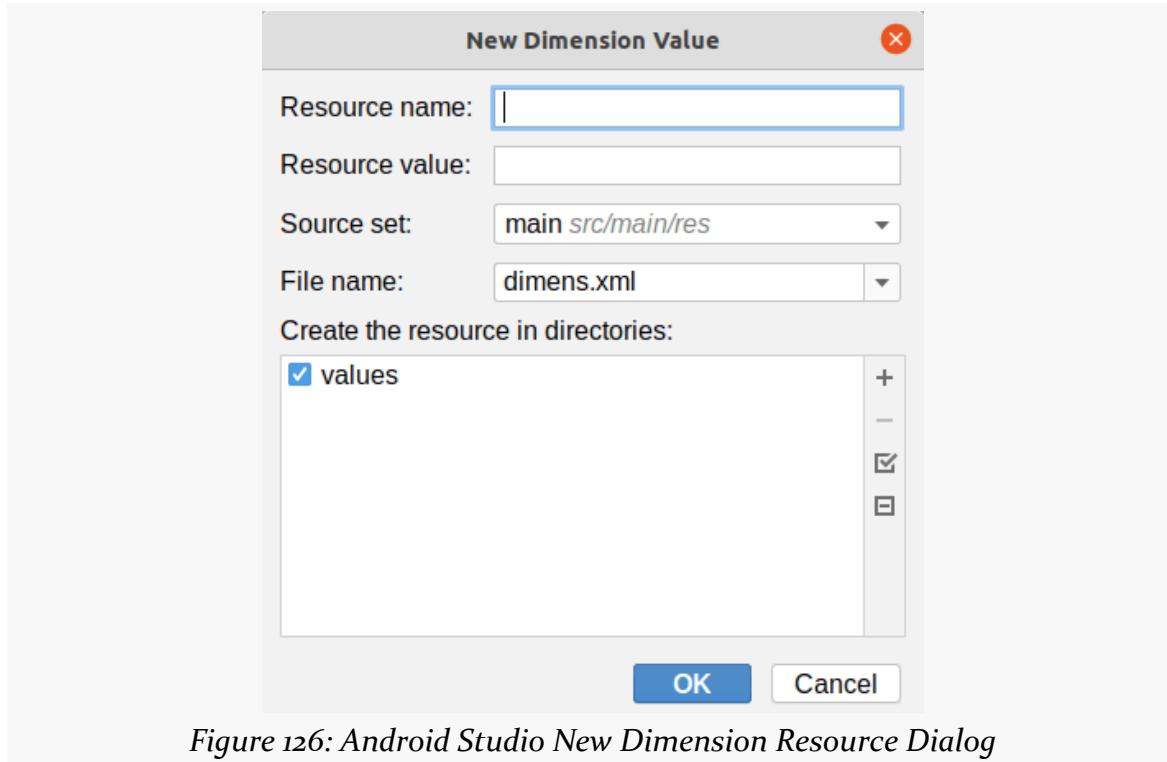


Figure 126: Android Studio New Dimension Resource Dialog

DISPLAYING AN ITEM

For the name, fill in `checked_icon_size`, and use `48dp` for the value. Click “OK” to close up both dialogs and fill in that dimension. Then, click the “O” next to the “layout_height” drop-down in the Attributes pane, and choose the `checked_icon_size` dimension from the list. This will give you a larger icon, with both height and width set to `48dp`:

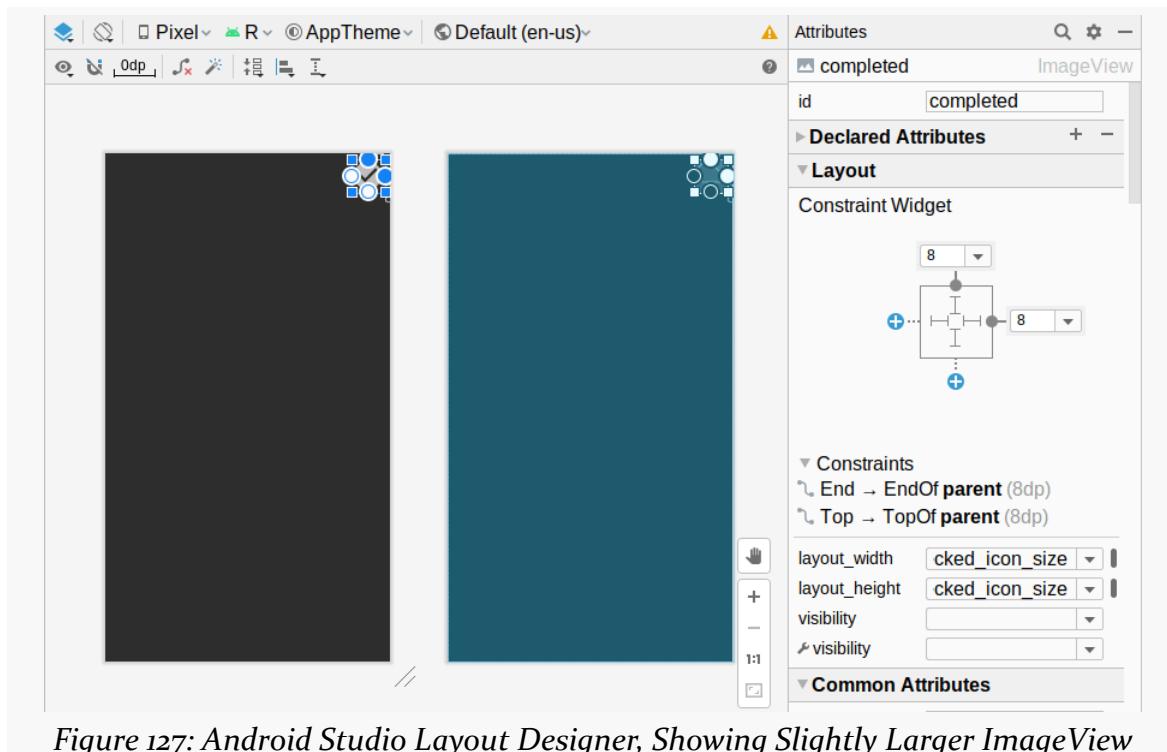


Figure 127: Android Studio Layout Designer, Showing Slightly Larger ImageView

For accessibility, it is good to supply a “content description” for an ImageView, which is some text to announce using a screen reader. To do that, in the “Common Attributes” section of the “Attributes” pane, click the “O” button next to the “contentDescription” field, to bring up a string resource chooser. Click the + icon and choose “String Value” from the drop-down menu to bring up the new string resource dialog. For the resource name, use `is_completed`, and for the resource value, use `Item is completed`. Click “OK” to close up both dialogs and apply this new string resource to the `android:contentDescription` attribute.

DISPLAYING AN ITEM

The icon appears gray. That works, but it is a bit boring, and it does not blend in with the rest of the colors used in the app. To change it to use our accent color, switch to the “Code” view and add `app:tint="@color/colorAccent"` as an attribute to the `<ImageView>` element. This will apply our amber accent color as a tint, which you will see if you switch back to the “Design” view:



Figure 128: Android Studio Layout Designer, Showing Tinted ImageView

Your layout XML should now 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">

    <ImageView
        android:id="@+id/completed"
        android:layout_width="@dimen/checked_icon_size"
        android:layout_height="@dimen/checked_icon_size"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:contentDescription="@string/is_completed"
        android:tint="@color/colorAccent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/ic_check_circle" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step #7: Displaying the Description

Next is the description of the to-do item.

In the graphical designer view of the layout resource editor, from the “Common” category of the “Palette” pane, drag a TextView into the preview area. Using the grab handles, set up three constraints:

- Tie the top and start edges to the corresponding edges of the ConstraintLayout
- Tie the end edge to the start edge of the ImageView



Figure 129: Android Studio Layout Designer, Showing Added TextView

In the “Attributes” pane, change the “layout_width” attribute to `match_constraint` (a.k.a., `0dp`). In the “Layout” section give it 8dp of top, start, and end margin. And, clear the contents of the “text” attribute, as we will fill that in at runtime.

To change the ID, switch to the “Code” view. There, in the `<TextView>` element, change the `android:id` value to be “`@+id/desc`”. Then, switch back to the “Design” view.

Then, in the “Common Attributes” section of the “Attributes” pane, find the `textAppearance` attribute and fill in `?attr/textAppearanceHeadline1`. This says “use a text appearance defined by `textAppearanceHeadline1` in our theme”. According to Material Design, this should format the text as a headline, and that seems like a reasonable choice for the description, since it is the most important element of our to-do item.

Unfortunately, Google’s AppCompat system does not adhere particularly well to Material Design. So, we need to make some adjustments, customizing this style a bit.

Open the `res/values/styles.xml` resource and add the following element to it:

```
<style name="HeadlineOneAppearance" parent="@style/TextAppearance.AppCompat.Large">
    <item name="android:textStyle">bold</item>
</style>
```

DISPLAYING AN ITEM

(from [T17-Display/ToDo/app/src/main/res/values/styles.xml](#))

This defines a custom style, `HeadlineOneAppearance`, that is based on the existing `TextAppearance.AppCompat.Large` style. It also overrides the `textStyle` attribute to be `bold`. Since `TextAppearance.AppCompat.Large` has a large font, this custom style defines a large bold font.

Then, add this element to the `Theme ToDo` resource definition in that file:

```
<item name="textAppearanceHeadline1">@style/HeadlineOneAppearance</item>
```

(from [T17-Display/ToDo/app/src/main/res/values/styles.xml](#))

This says “when a widget tries to use `textAppearanceHeadline1`, use this style resource for that”.

The layout should now 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">

    <ImageView
        android:id="@+id/completed"
        android:layout_width="@dimen/checked_icon_size"
        android:layout_height="@dimen/checked_icon_size"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:contentDescription="@string/is_completed"
        android:tint="@color/colorAccent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/ic_check_circle" />

    <TextView
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:textAppearance="?attr/textAppearanceHeadline1"
        app:layout_constraintEnd_toStartOf="@+id/completed"
```

DISPLAYING AN ITEM

```
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step #8: Showing the Created-On Date

The next bit of data to display is the date on which the to-do item was created. Also, we need to provide a label for the date, as otherwise the user may not realize what this date means.

First, let's set up the label. In the graphical designer, drag another `TextView` into the layout. Drag the grab handles to set up constraints:

- On the start end of the label, to the start side of the `ConstraintLayout`
- On the top of the label, to the bottom of the desc `TextView`

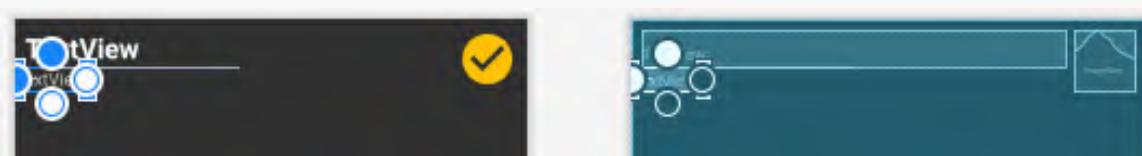


Figure 130: Android Studio Layout Designer, Showing Another Added TextView

Give the widget an ID of `labelCreated`. And, in the “Layout” section of the “Attributes” pane, give it 8dp of top and start margin, via the drop-downs. Set `layout_width` and `layout_height` each to `wrap_content`.

To set the text to a fixed value, we can set up another string resource. However, the Attributes pane has two attributes that look like they set the text of the `TextView`:

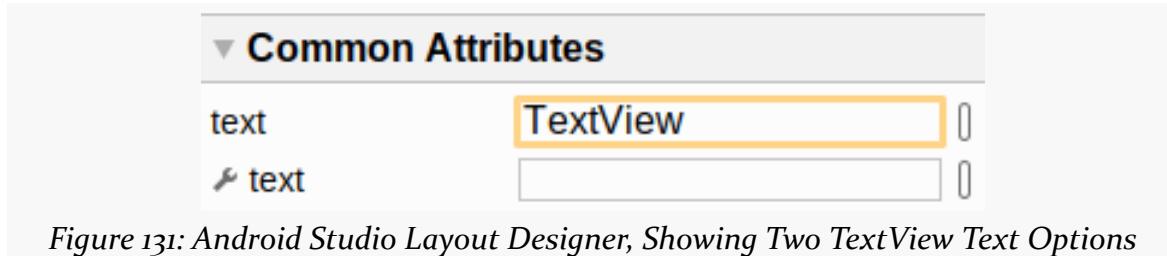


Figure 131: Android Studio Layout Designer, Showing Two TextView Text Options

The one with the wrench icon sets up separate text to show when working in the design view of the layout resource editor. We want the other one, that sets the text for actual app — it has `TextView` as the value right now. Click the “O” button next to

DISPLAYING AN ITEM

that field, and choose the drop-down option to create a new string resource. Give it a name of `created_on` and a value of “Created on:”. Clicking “OK” will close the dialog and assign that string resource to the `TextView` for the `android:text` attribute.

Then, find the `textAppearance` attribute in the “Attributes” pane and set its value to `?attr/textAppearanceHeadline2`. As before, this delegates the design of the text to whatever our theme has for `textAppearanceHeadline2`.

However, while this is supposed to be sized like a secondary headline, that is not what AppCompat has. So, back over in `res/values/styles.xml`, add a new style resource:

```
<style name="HeadlineTwoAppearance" parent="@style/TextAppearance.AppCompat.Medium">
</style>
```

(from [T17-Display/ToDo/app/src/main/res/values/styles.xml](#))

This `HeadlineTwoAppearance` simply inherits from `TextAppearance.AppCompat.Medium`. We could override other attributes here, but at the moment, we do not need any.

Then, add this attribute to `Theme ToDo`:

```
<item name="textAppearanceHeadline2">@style/HeadlineTwoAppearance</item>
```

(from [T17-Display/ToDo/app/src/main/res/values/styles.xml](#))

This indicates that the theme’s `textAppearanceHeadline2` maps to our new `HeadlineTwoAppearance` style. And this gives us a better text size for our creation date.

Now, we can show the created-on date, next to our newly-created label.

Drag yet another `TextView` into the layout. Drag the grab handles to set up constraints:

DISPLAYING AN ITEM

- On the start side of this TextView, to the end side of the labelCreated TextView
- On the top of this TextView, to the bottom of the desc TextView
- On the end side of this TextView, to the start side of the icon



Figure 132: Android Studio Layout Designer, Showing Yet Another TextView

Then, set the “layout_width” to `match_constraint` (a.k.a., `0dp`). Give the widget an ID of `createdOn`. In the “Layout” section of the “Attributes” pane, give it 8dp of top, start, and end margin, via the drop-downs. And, clear the contents of the “text” attribute, as we will fill that in at runtime.

Next, find the `textAppearance` attribute in the “Attributes” pane and set its value to `?attr/textAppearanceHeadline2`, the same as we used for its label.

At this point, the XML of the layout resource 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">

    <ImageView
        android:id="@+id/completed"
        android:layout_width="@dimen/checked_icon_size"
        android:layout_height="@dimen/checked_icon_size"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:contentDescription="@string/is_completed"
        android:tint="@color/colorAccent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/ic_check_circle" />

    <TextView
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
```

DISPLAYING AN ITEM

```
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:textAppearance="?attr/textAppearanceHeadline1"
        app:layout_constraintEnd_toStartOf="@+id/completed"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:id="@+id/labelCreated"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@string/created_on"
    android:textAppearance="?attr/textAppearanceHeadline2"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/desc" />

<TextView
    android:id="@+id/createdOn"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:textAppearance="?attr/textAppearanceHeadline2"
    app:layout_constraintEnd_toStartOf="@+id/completed"
    app:layout_constraintStart_toEndOf="@+id/labelCreated"
    app:layout_constraintTop_toBottomOf="@+id/desc" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step #9: Adding the Notes

There is only one more widget to add: another `TextView`, this time for the notes.

DISPLAYING AN ITEM

Over in the design tab, drag one more TextView out of the “Palette” pane into the preview area. Set the constraints to have the top of the TextView attach to the bottom of the createdOn TextView, and have the other three sides attach to the edges of the ConstraintLayout:

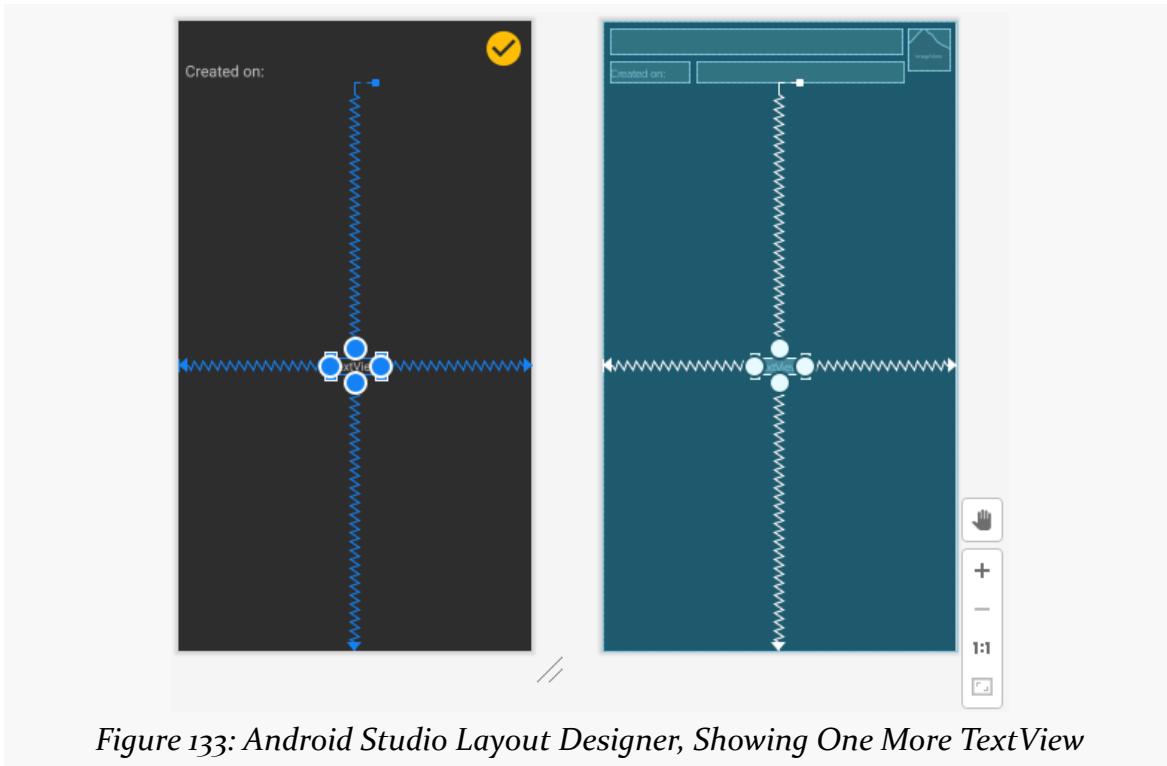


Figure 133: Android Studio Layout Designer, Showing One More TextView

DISPLAYING AN ITEM

Change both the “layout_width” and “layout_height” attributes to `match_constraint`. Give the widget an ID of `notes`. Clear the “text” attribute, as we will fill that value in at runtime. And, in the “Layout” section of the “Attributes” pane, give it 8dp of margin on all four sides, via the drop-downs:

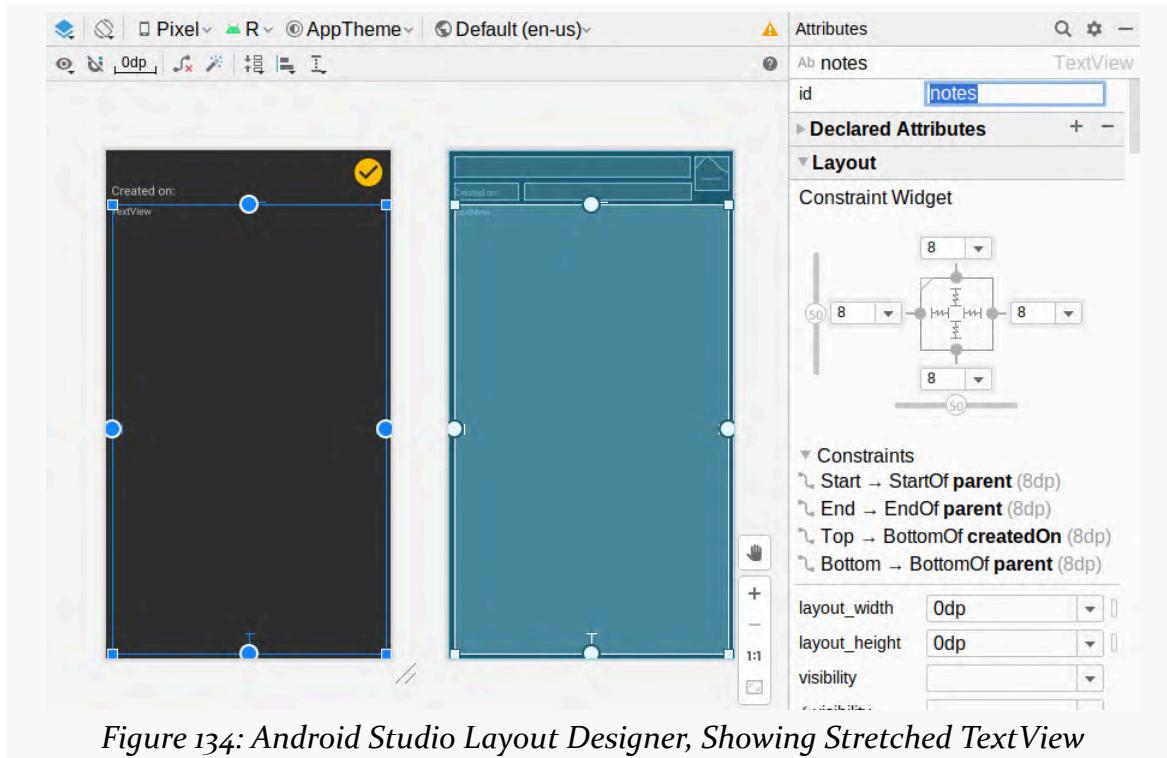


Figure 134: Android Studio Layout Designer, Showing Stretched TextView

For the `textAppearance`, fill in `?attr/textAppearanceBody1`, to use our theme’s `textAppearanceBody1` rules for formatting the text. Over in `res/values/styles.xml`, add this XML element:

```
<style name="BodyAppearance" parent="@style/TextAppearance.AppCompat.Medium">
</style>
```

(from [T17-Display/ToDo/app/src/main/res/values/styles.xml](#))

This is the same as `HeadlineTwoAppearance`, other than the name. Having two separate styles allows us to format the text differently in the future (e.g., have the body be monospace), should we choose to do so.

Then, add another `<item>` element to `Theme.ToDo` to tie in this new style:

```
<item name="textAppearanceBody1">@style/BodyAppearance</item>
```

DISPLAYING AN ITEM

(from [T17-Display/ToDo/app/src/main/res/values/styles.xml](#))

Step #10: Adding Navigation Arguments

Before we can display our ToDoModel, we need to get it in DisplayFragment. And before we can do *that*, we need DisplayFragment to know what model that is.

This gets back to the gap we saw [several steps ago](#), where our display() function in RosterListFragment was not doing anything with the ToDoModel that the user clicked on. We need to use that somehow.

And for that, we will add an argument to our navigation graph.

Open res/navigation/nav_graph.xml and, in the graphical editor, click on the displayFragment. In the “Attributes” pane, you will see an “Arguments” section with a + icon:

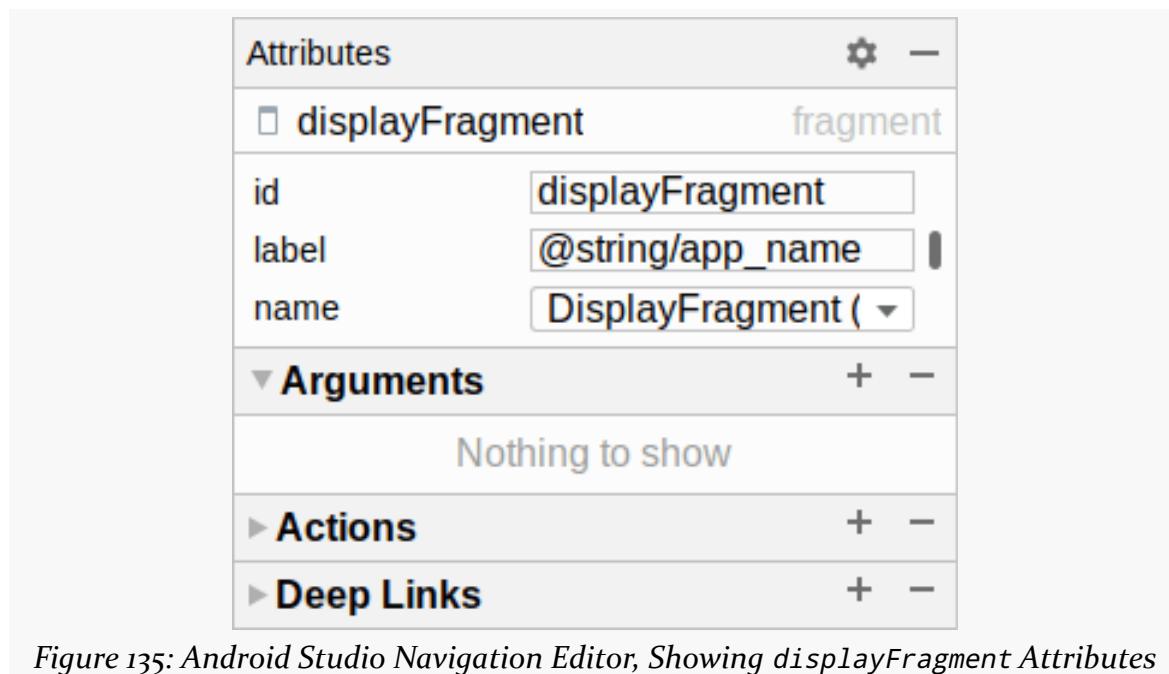


Figure 135: Android Studio Navigation Editor, Showing displayFragment Attributes

DISPLAYING AN ITEM

Click that + icon in the “Arguments” to open up a dialog to define an argument:

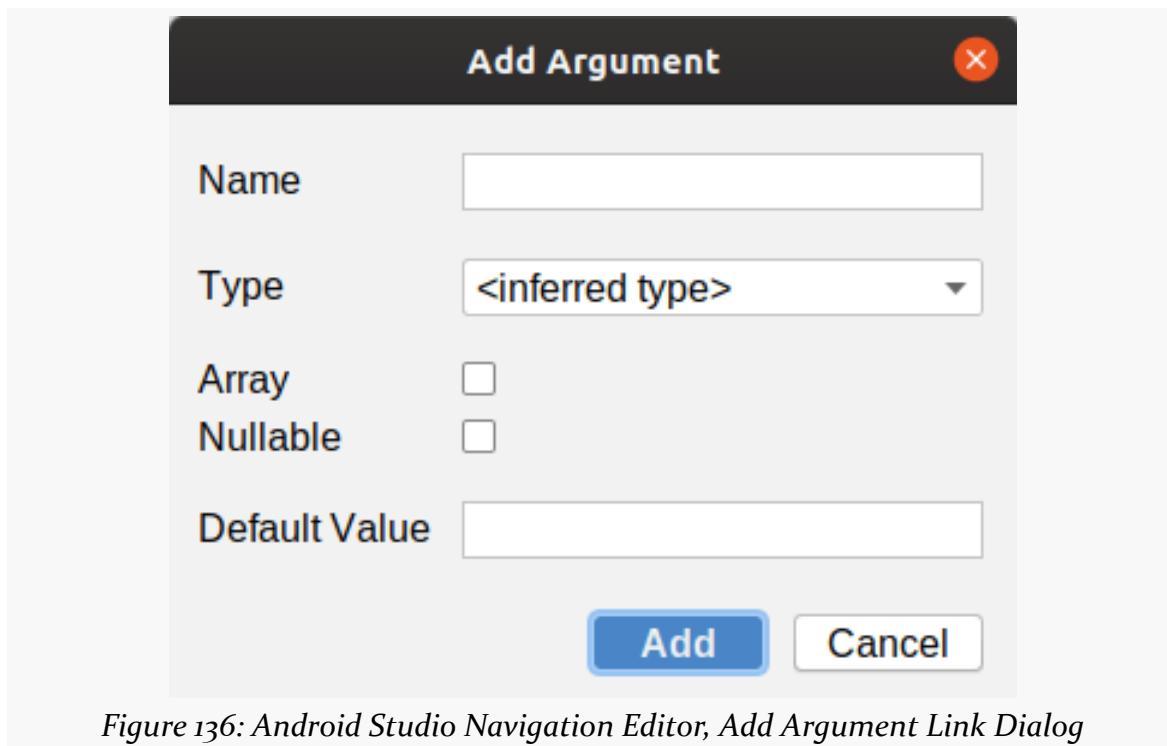


Figure 136: Android Studio Navigation Editor, Add Argument Link Dialog

Fill it in as follows:

DISPLAYING AN ITEM

Property	Value
Name	modelId
Type	String
Array	unchecked
Nullable	unchecked
Default Value	(leave empty)

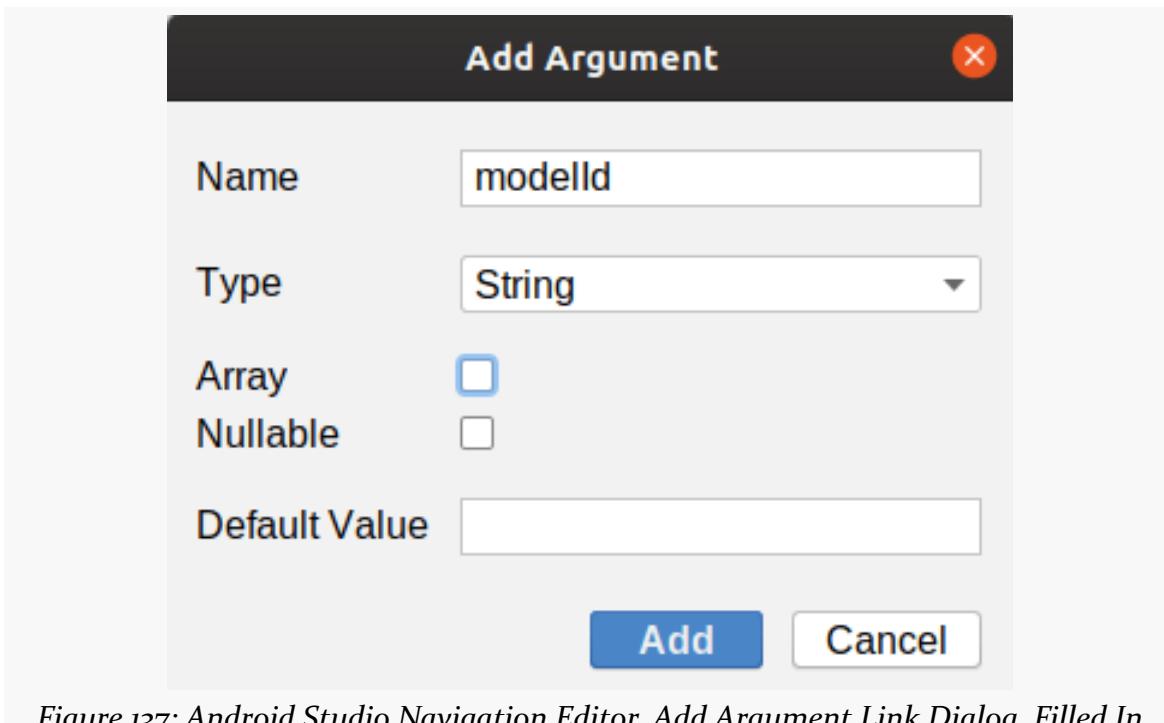


Figure 137: Android Studio Navigation Editor, Add Argument Link Dialog, Filled In

Click “Add” to add it to the navigation graph.

From the Android Studio main menu, choose “Build” > “Make Module ‘app’”. After a few moments, you should get a compile error, from our `display()` function in `RosterListFragment`:

```
private fun display(model: ToDoModel) {  
    findNavController().navigate(RosterListFragmentDirections.displayModel())  
}
```

DISPLAYING AN ITEM

We added an argument to `displayFragment`. Now our action that will navigate to `displayFragment` needs a value for that argument, so the `RosterListFragmentDirections.displayModel()` function needs our `modelId` value. So, modify the function to look like:

```
private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Then, over in `DisplayFragment`, add this property:

```
private val args: DisplayFragmentArgs by navArgs()
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

`DisplayFragmentArgs` is code-generated by the Safe Args plugin for the Navigation component. It looks at our declared arguments and creates a Kotlin class that represents them. Moreover, we get a `navArgs()` delegate that will build that `DisplayFragmentArgs` for us when we first access the `args` property. We will be able to use this to access our `modelId` value.

Step #11: Displaying the Layout

In `DisplayFragment`, add a binding field, pointing to our newly-generated `TodoDisplayBinding` class from our `todo_display` layout resource:

```
private var binding: TodoDisplayBinding? = null
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

Then, in `DisplayFragment`, add an `onCreateView()` function:

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
) = TodoDisplayBinding.inflate(inflater, container, false)
    .apply { binding = this }
    .root
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

DISPLAYING AN ITEM

This works akin to how `onCreateViewHolder()` does in `RosterAdapter`, inflating the binding from the resource, using the code-generated `TodoDisplayBinding` class. Here, we assign the binding itself to the `binding` property, while returning the root View of the inflated layout.

Also, add this `onDestroyView()` function to `DisplayFragment`:

```
override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

As with `RosterListFragment`, this sets `binding` back to null, so we do not leak the binding after our fragment's UI is destroyed.

Step #12: Making Another Motor

Now, let's create another `ViewModel` implementation that uses the same pattern as `RosterMotor`, but for a single model based on its ID. We can use this both for `DisplayFragment` and the `EditFragment` that we will create in [the next tutorial](#).

Right-click over the `com.commonsware.todo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `SingleModelMotor`, then choose “Class” for the kind. Then press or to create the empty `SingleModelMotor` class.

Then, replace its contents with:

```
package com.commonsware.todo

import androidx.lifecycle.ViewModel

class SingleModelMotor(
    private val repo: ToDoRepository,
    private val modelId: String
) : ViewModel()
```

Like `RosterMotor`, this takes a `ToDoRepository` as a constructor parameter. Unlike `RosterMotor`, this also takes the ID of the `ToDoModel` that we want to use, as another

DISPLAYING AN ITEM

constructor parameter.

To actually retrieve the `ToDoModel` for this ID, we could just rummage through `repo.items` here in `RosterMotor`. It will be cleaner to have `ToDoRepository` do this. So, add this `find()` function to `ToDoRepository`:

```
fun find(modelId: String) = items.find { it.id == modelId }
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

Right now, all this does is scan through the `items` list and find the matching `ToDoModel`. Later on, we will do a database query to find the to-do item in the database.

Then, add a corresponding `getModel()` function to `SingleModelMotor`:

```
fun getModel() = repo.find(modelId)
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#))

This just uses the `repo` and `modelId` to retrieve the `ToDoModel` and returns it.

We need to teach Koin about this viewmodel, the same way that we did with `RosterMotor`. So, in `ToDoApp`, add this line to the `koinModule` declaration:

```
viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This is a bit different than the `viewModel()` call to set up `RosterMotor`:

```
viewModel { RosterMotor(get()) }
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This time, we want to provide a parameter when we retrieve the `SingleModelMotor`: the model ID of the `ToDoModel` that we want to display. Koin has no way of getting this value on its own, the way it can for the `ToDoRepository` singleton. So, we set up the lambda expression to return a function type, one that takes our `modelId` as a parameter and uses it to construct the `SingleModelMotor` instance.

Then, in `DisplayFragment`, add this property:

```
private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }
```

DISPLAYING AN ITEM

(from [Ti7-Display/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

This time, instead of just using `by viewModel()`, we use a variant that employs `parametersOf()` to supply the parameters that get passed to our function type that we used in the Koin declaration. Here, we get the `modelId` out of our Navigation component arguments, wrap them using `parametersOf()`, and use that to set up the `SingleModelMotor`. The net result is that the model ID that we had in `RosterListFragment` — from when the user clicked the row — winds up in the hands of our viewmodel, and from there can be used to get the `ToDoModel`.

Step #13: Populating the Layout

Finally, we can use our `ToDoModel` to fill in the widgets of our layout.

Add this `onViewCreated()` function to `DisplayFragment`:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    motor.getModel()?.let {
        binding?.apply {
            completed.visibility = if (it.isCompleted) View.VISIBLE else View.GONE
            desc.text = it.description
            createdOn.text = DateUtils.getRelativeDateTimeString(
                requireContext(),
                it.createdOn.toEpochMilli(),
                DateUtils.MINUTE_IN_MILLIS,
                DateUtils.WEEK_IN_MILLIS,
                0
            )
            notes.text = it.notes
        }
    }
}
```

(from [Ti7-Display/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

This retrieves the model given its ID and updates the widgets in the `ToDoBinding` based on that model:

- For the checkmark icon, we control its visibility based on the `isCompleted` model value, to show the icon if the model is completed (`View.VISIBLE`) or hide it if the model is not (`View.GONE`).
- For the description and notes, we just populate the `TextView` widgets from the strings held in the model.
- For `createdOn`, we get the `Instant` from the model, convert it into a standard

DISPLAYING AN ITEM

“milliseconds since the Unix epoch” value, and pass that to `DateUtils.getRelativeDateTimeString()`.

`DateUtils.getRelativeDateTimeString()` will return a value formatted in accordance with the user’s locale and device configuration, plus use a relative time (e.g., “35 minutes ago”) for recent times.

At this point, if you run the app, and you click on one of the to-do items in the list, the full details should appear in the `DisplayFragment`:

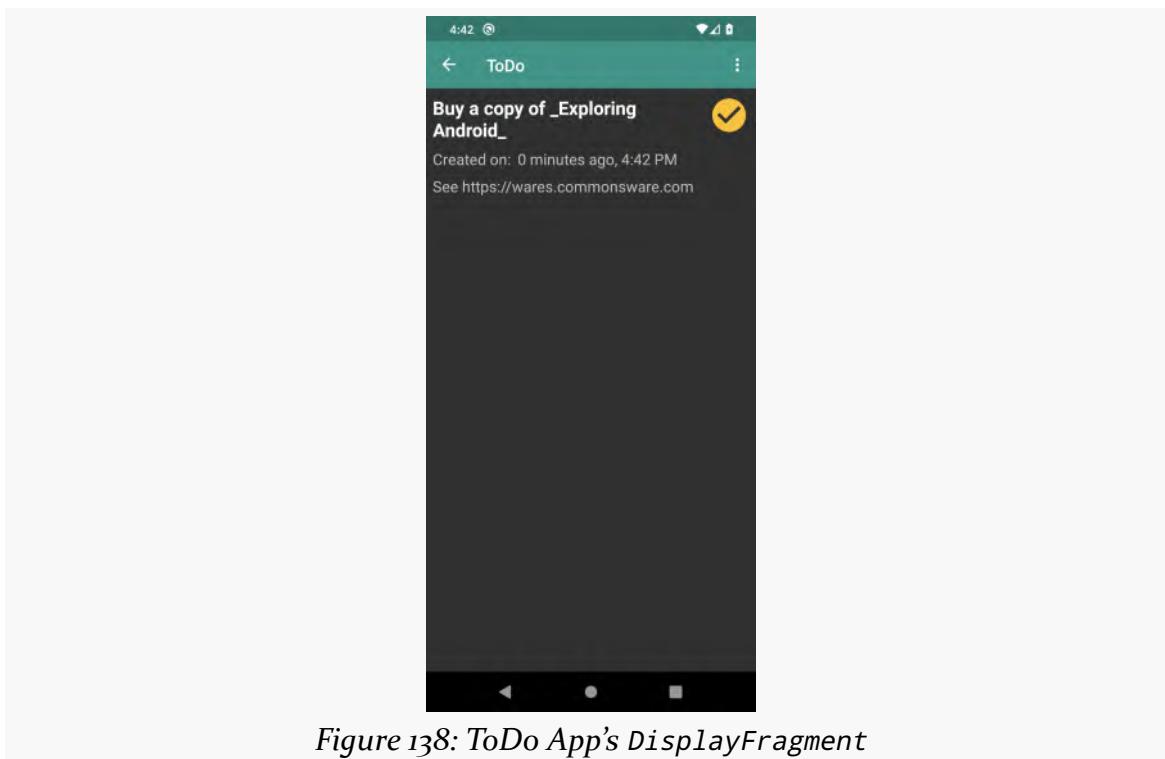


Figure 138: ToDo App’s `DisplayFragment`

Pressing BACK returns you to the list, as before.

Final Results

We changed a *lot* of stuff in this tutorial!

The `nav_graph` navigation resource should contain:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
```

DISPLAYING AN ITEM

```
    android:id="@+id/nav_graph.xml"
    app:startDestination="@+id/rosterListFragment">

    <fragment
        android:id="@+id/rosterListFragment"
        android:name="com.commonsware.todo.RosterListFragment"
        android:label="@string/app_name">
        <action
            android:id="@+id/displayModel"
            app:destination="@+id/displayFragment" />
    </fragment>
    <fragment
        android:id="@+id/displayFragment"
        android:name="com.commonsware.todo.DisplayFragment"
        android:label="@string/app_name" >
        <argument
            android:name="modelId"
            app:argType="string" />
    </fragment>
</navigation>
```

(from [Tr7-Display/ToDo/app/src/main/res/navigation/nav_graph.xml](#))

RosterRowHolder should look like:

```
package com.commonsware.todo

import androidx.recyclerview.widget.RecyclerView
import com.commonsware.todo.databinding.TodoRowBinding

class RosterRowHolder(
    private val binding: TodoRowBinding,
    val onCheckboxToggle: (ToDoModel) -> Unit,
    val onRowClick: (ToDoModel) -> Unit
) :
    RecyclerView.ViewHolder(binding.root) {

    fun bind(model: ToDoModel) {
        binding.apply {
            root.setOnClickListener { onRowClick(model) }
            isCompleted.isChecked = model.isCompleted
            isCompleted.setOnCheckedChangeListener { _, _ -> onCheckboxToggle(model) }
            desc.text = model.description
        }
    }
}
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

DISPLAYING AN ITEM

RosterAdapter should resemble:

```
package com.commonsware.todo

import android.view.LayoutInflater
import android.view.ViewGroup
import androidx.recyclerview.widget.DiffUtil
import androidx.recyclerview.widget.ListAdapter
import com.commonsware.todo.databinding.TodoRowBinding

class RosterAdapter(
    private val inflator: LayoutInflater,
    private val onCheckboxToggle: (ToDoModel) -> Unit,
    private val onRowClick: (ToDoModel) -> Unit
) :
    ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
        RosterRowHolder(
            TodoRowBinding.inflate(inflator, parent, false),
            onCheckboxToggle,
            onRowClick
        )

    override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
        holder.bind(getItem(position))
    }
}

private object DiffCallback : DiffUtil.ItemCallback<ToDoModel>() {
    override fun areItemsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.id == newItem.id

    override fun areContentsTheSame(oldItem: ToDoModel, newItem: ToDoModel) =
        oldItem.isCompleted == newItem.isCompleted &&
            oldItem.description == newItem.description
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#))

RosterListFragment should look like:

```
package com.commonsware.todo

import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
```

DISPLAYING AN ITEM

```
import androidx.fragment.app.Fragment
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.databinding.TodoRosterBinding
import org.koin.android.viewmodel.ext.android.viewModel
import androidx.navigation.fragment.findNavController

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private var binding: TodoRosterBinding? = null

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root

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

        val adapter = RosterAdapter(
            layoutInflater,
            onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
            onRowClick = ::display)

        binding?.items?.apply {
            setAdapter(adapter)
            layoutManager = LinearLayoutManager(context)

            addItemDecoration(
                DividerItemDecoration(
                    activity,
                    DividerItemDecoration.VERTICAL
                )
            )
        }
    }

    adapter.submitList(motor.items)
    binding?.empty?.visibility = View.GONE
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}
```

DISPLAYING AN ITEM

```
private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

MainActivity should contain:

```
package com.commonsware.todo

import android.content.Intent
import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import androidx.appcompat.app.AppCompatActivity
import androidx.navigation.findNavController
import androidx.navigation.fragment.findNavController
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.NavigationUI.navigateUp
import androidx.navigation.ui.setupActionBarWithNavController
import com.commonsware.todo.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var appBarConfiguration: AppBarConfiguration

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

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)
        setSupportActionBar(binding.toolbar)

        supportFragmentManager.findFragmentById(R.id.nav_host)?.findNavController()?.let
        { nav ->
            appBarConfiguration = AppBarConfiguration(nav.graph)
            setupActionBarWithNavController(nav, appBarConfiguration)
        }
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.actions, menu)

        return super.onCreateOptionsMenu(menu)
    }
```

DISPLAYING AN ITEM

```
override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
    R.id.about -> {
        startActivity(Intent(this, AboutActivity::class.java))
        true
    }
    else -> super.onOptionsItemSelected(item)
}

override fun onSupportNavigateUp() =
    navigateUp(findNavController(R.id.nav_host), appBarConfiguration)
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

The todo_display layout resource should contain:

```
<?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"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:id="@+id/completed"
        android:layout_width="@dimen/checked_icon_size"
        android:layout_height="@dimen/checked_icon_size"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:contentDescription="@string/is_completed"
        app:tint="@color/colorAccent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:srcCompat="@drawable/ic_check_circle" />

    <TextView
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:textAppearance="?attr/textAppearanceHeadline1"
        app:layout_constraintEnd_toStartOf="@+id/completed"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
```

DISPLAYING AN ITEM

```
<TextView
    android:id="@+id/labelCreated"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@string/created_on"
    android:textAppearance="?attr/textAppearanceHeadline2"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/desc" />

<TextView
    android:id="@+id/createdOn"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:textAppearance="?attr/textAppearanceHeadline2"
    app:layout_constraintEnd_toStartOf="@+id/completed"
    app:layout_constraintStart_toEndOf="@+id/labelCreated"
    app:layout_constraintTop_toBottomOf="@+id/desc" />

<TextView
    android:id="@+id/notes"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginBottom="8dp"
    android:textAppearance="?attr/textAppearanceBody1"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/createdOn" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [T17-Display/ToDo/app/src/main/res/layout/todo_display.xml](#))

The styles resource should resemble:

```
<resources>

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

DISPLAYING AN ITEM

```
<item name="colorPrimaryDark">@color/colorPrimaryDark</item>
<item name="colorAccent">@color/colorAccent</item>
<item name="textAppearanceHeadline1">@style/HeadlineOneAppearance</item>
<item name="textAppearanceHeadline2">@style/HeadlineTwoAppearance</item>
<item name="textAppearanceBody1">@style/BodyAppearance</item>
</style>

<style name="HeadlineOneAppearance" parent="@style/TextAppearance.AppCompat.Large">
    <item name="android:textStyle">bold</item>
</style>

<style name="HeadlineTwoAppearance" parent="@style/TextAppearance.AppCompat.Medium">
</style>

<style name="BodyAppearance" parent="@style/TextAppearance.AppCompat.Medium">
</style>

</resources>
```

(from [T17-Display/ToDo/app/src/main/res/values/styles.xml](#))

SingleModelMotor should contain:

```
package com.commonsware.todo

import androidx.lifecycle.ViewModel

class SingleModelMotor(
    private val repo: ToDoRepository,
    private val modelId: String
) : ViewModel() {
    fun getModel() = repo.find(modelId)
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#))

ToDoRepository should contain:

```
package com.commonsware.todo

class ToDoRepository {
    var items = listOf(
        ToDoModel(
            description = "Buy a copy of _Exploring Android_",
            isCompleted = true,
            notes = "See https://wares.commonsware.com"
        ),
        ToDoModel(
```

DISPLAYING AN ITEM

```
        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!"
)
)

fun save(model: ToDoModel) {
    items = if (items.any { it.id == model.id }) {
        items.map { if (it.id == model.id) model else it }
    } else {
        items + model
    }
}

fun find(modelId: String) = items.find { it.id == modelId }
}
```

(from [Tr7-Display/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

ToDoApp should contain:

```
package com.commonsware.todo

import android.app.Application
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.dsl.module

class ToDoApp : Application() {
    private val koinModule = module {
        single { ToDoRepository() }
        viewModel { RosterMotor(get()) }
        viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            modules(koinModule)
        }
    }
}
```

DISPLAYING AN ITEM

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Finally, `DisplayFragment` should now resemble:

```
package com.commonsware.todo

import android.os.Bundle
import android.text.format.DateUtils
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.databinding.TodoDisplayBinding
import org.koin.androidx.viewmodel.ext.android.viewModel
import org.koin.core.parameter.parametersOf

class DisplayFragment : Fragment() {
    private val args: DisplayFragmentArgs by navArgs()
    private var binding: TodoDisplayBinding? = null
    private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) = TodoDisplayBinding.inflate(inflater, container, false)
        .apply { binding = this }
        .root

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        motor.getModel()?.let {
            binding?.apply {
                completed.visibility = if (it.isCompleted) View.VISIBLE else View.GONE
                desc.text = it.description
                createdOn.text = DateUtils.getRelativeDateTimeString(
                    requireContext(),
                    it.createdOn.toEpochMilli(),
                    DateUtils.MINUTE_IN_MILLIS,
                    DateUtils.WEEK_IN_MILLIS,
                    0
                )
                notes.text = it.notes
            }
        }
    }

    override fun onDestroyView() {
```

DISPLAYING AN ITEM

```
binding = null  
  
super.onDestroy()  
}  
}
```

(from [T17-Display/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#)
- [app/src/main/res/navigation/nav_graph.xml](#)
- [app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterAdapter.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)
- [app/src/main/java/com/commonsware/todo/MainActivity.kt](#)
- [app/src/main/res/layout/todo_display.xml](#)
- [app/src/main/res/drawable/ic_check_circle.xml](#)
- [app/src/main/res/values/styles.xml](#)
- [app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#)

Editing an Item

Displaying to-do items is nice. However, right now, all of the to-do items are fake. We need to start allowing the user to fill in their own to-do items.

The first task is to set up an edit fragment. Just as we can click on a to-do item in the list to bring up the details, we need to be able to click on something in the details to be able to edit the description, notes, etc. So, just as we created a `DisplayFragment` in the preceding tutorial, here we will create an `EditFragment` and arrange to display it.

This tutorial has many similarities to the preceding one:

- We create a fragment
- We create a layout for that fragment
- We use data binding to populate the layout from the fragment

The differences come in the layout itself, as we have a different mix of widgets than before. Plus, we need to add toolbar button to the app bar, to allow the user to request to edit that to-do item.

You might wonder “hey, shouldn’t we use inheritance or something here?” In theory, we could. In practice, the `DisplayFragment` is going to change quite a bit in a later tutorial, and so we would have to undo the inheritance work at that point anyway.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Creating the Fragment

Yet again, we need to set up a fragment.

Right-click over the `com.commonware.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. For the name, fill in `EditFragment`. For the kind, choose “Class”. Press `Enter` or `Return` to create the class.

That will give you an `EditFragment` that looks like:

```
package com.commonware.todo

class EditFragment {  
}
```

Then, have it extend the `Fragment` class:

```
package com.commonware.todo

import androidx.fragment.app.Fragment

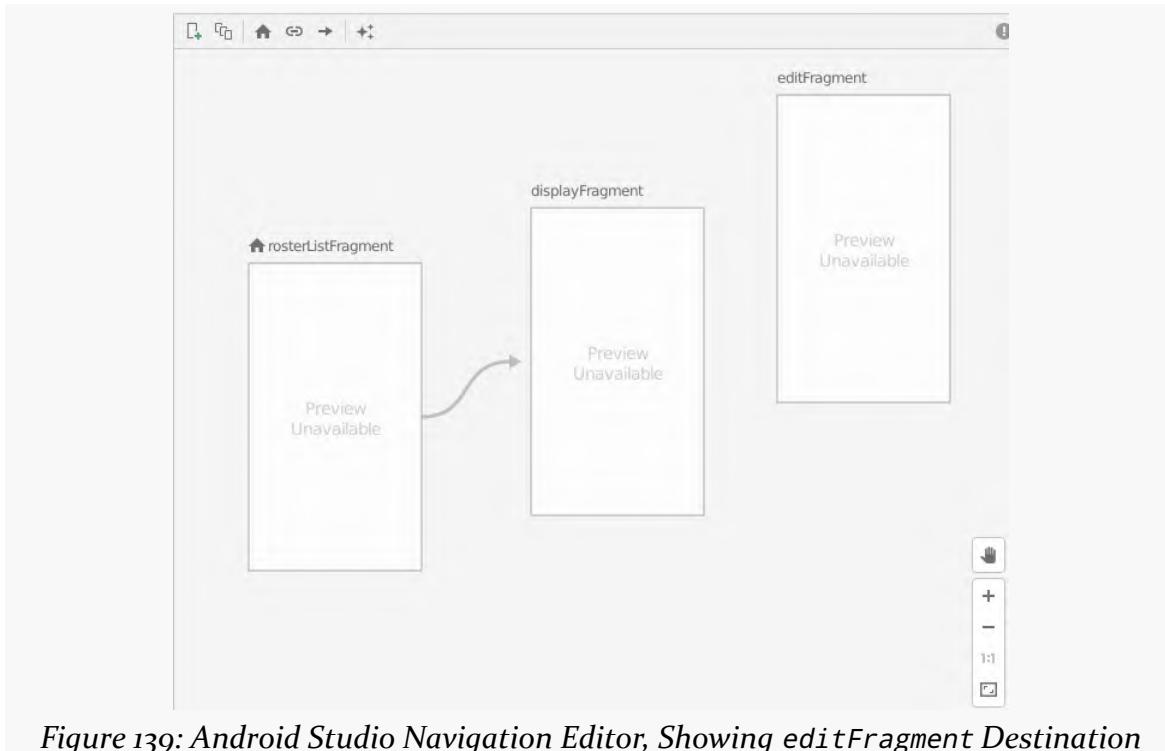
class EditFragment : Fragment() {  
}
```

Step #2: Setting Up the Navigation

This class needs the ID of the `ToDoModel` to edit, just as `DisplayFragment` needed the model to display. So, we are going to set up the same sort of navigation logic as we used to get from `RosterListFragment` to `DisplayFragment`, this time to get from `DisplayFragment` to `EditFragment`.

EDITING AN ITEM

Once again, open `res/navigation/nav_graph.xml`. This time, when you click the new-destination toolbar button, `EditFragment` should be among the options. Choose it, adding it to your graph. If needed, drag it over to the right side of the space, perhaps adjusting the zoom level using the `+/ -` toolbar buttons:



EDITING AN ITEM

Next, click on `displayFragment` and drag an arrow from it to `editFragment`:

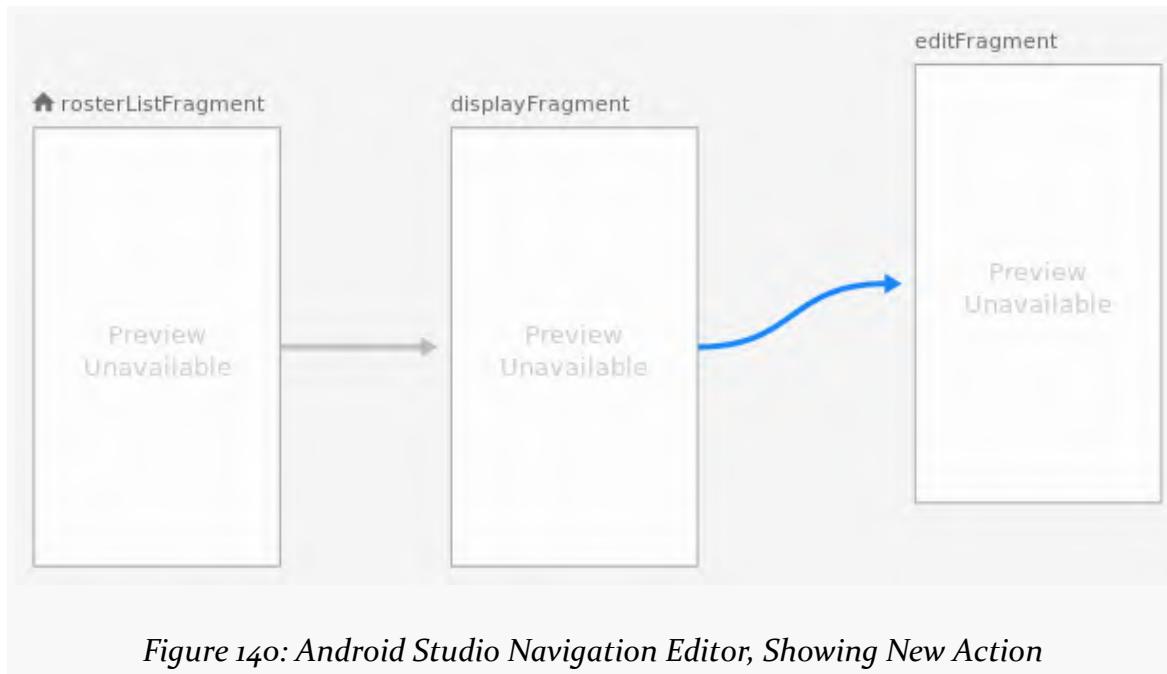


Figure 140: Android Studio Navigation Editor, Showing New Action

In the “Attributes” pane, with the new action selected, change the ID to `editModel`.

Then, click on `editFragment`. Change the “Label” to be `@string/app_name`.

Next, in the “Arguments” section of the “Attributes” pane, click the + icon to add a new argument. As before, give it a name of `modelId` and a type of `String`. Then click “Add” to add it to the navigation graph.

Step #3: Setting Up a Menu Resource

Somewhere, somehow, the user has to be able to get to this fragment. A typical pattern is for there to be an “edit” option somewhere where we are displaying the thing to be edited. In the case of this app, that implies having an “edit” option on the `DisplayFragment`, and we can do this by adding an app bar item.

EDITING AN ITEM

First, though, we need an icon for that button. Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Clip Art” button and search for `edit`:

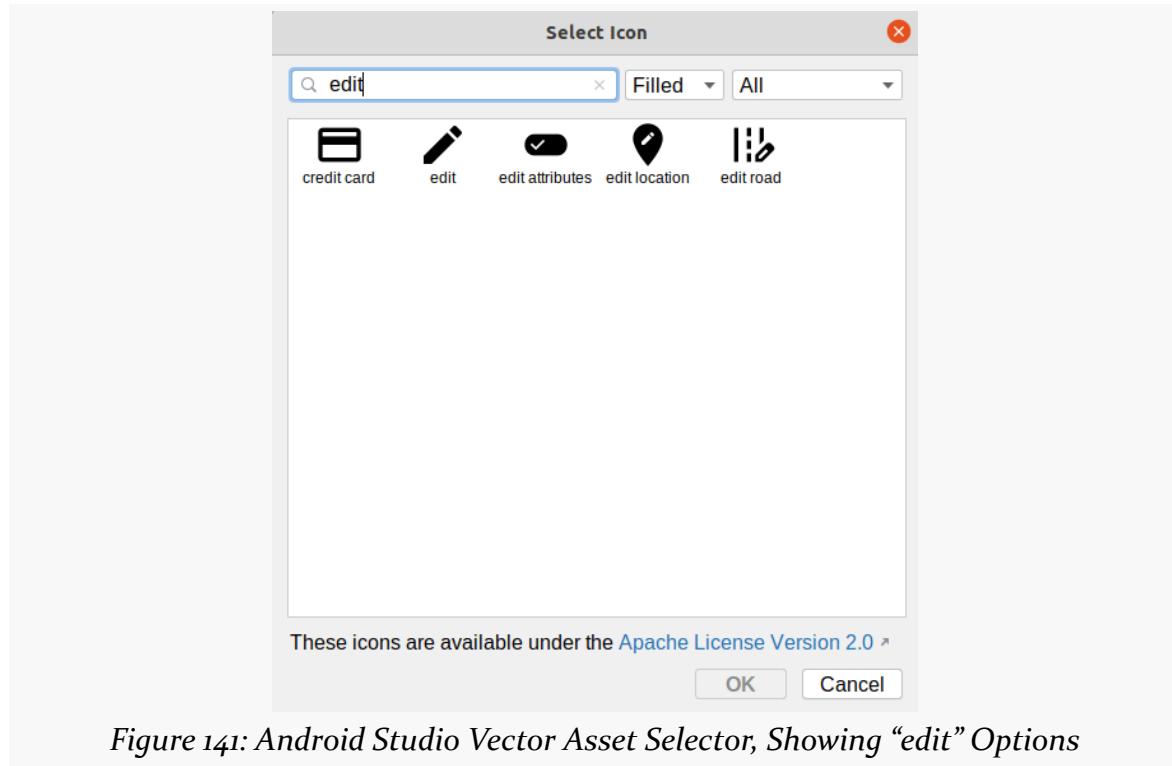


Figure 141: Android Studio Vector Asset Selector, Showing “edit” Options

Choose the “edit” icon and click “OK” to close up the icon selector. Change the icon’s name to `ic_edit`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

EDITING AN ITEM

Then, right-click over the `res/menu/` directory and choose `New > "Menu resource file"` from the context menu. Fill in `actions_display.xml` in the “New Menu Resource File” dialog, then click `OK` to create the file and open it up in the menu editor. In the Palette, drag a “Menu Item” into the preview area. This will appear as an item in an overflow area:

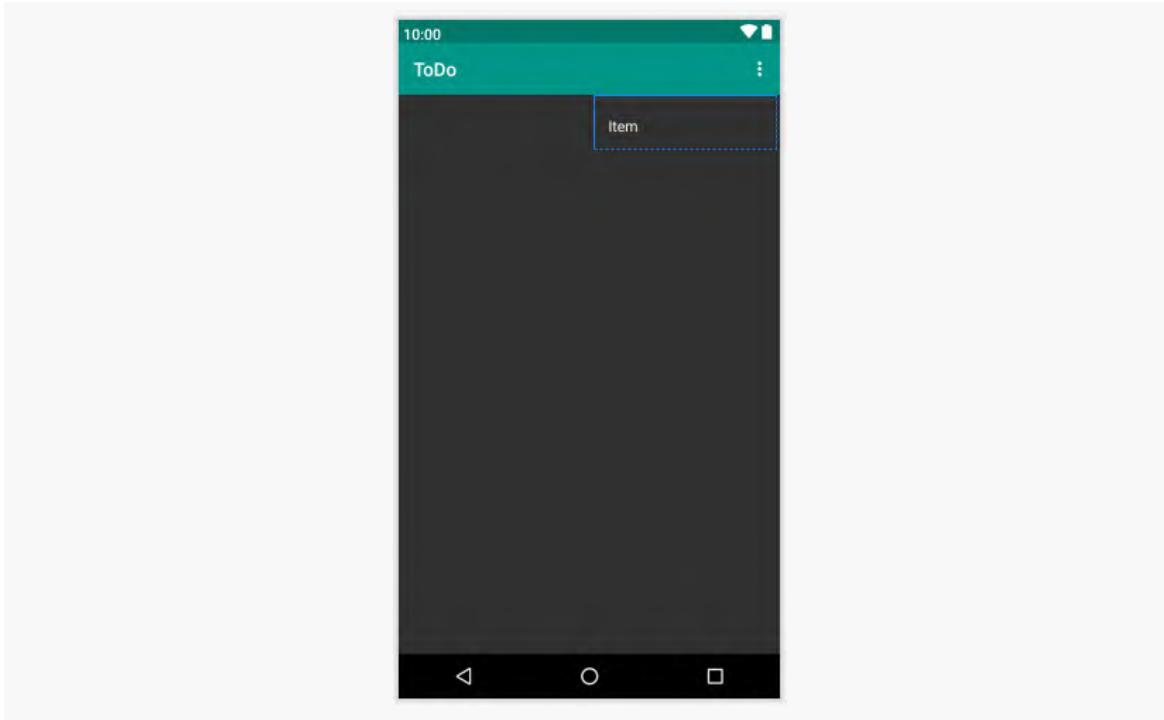


Figure 142: Android Studio Menu Editor, Showing Added MenuItem

EDITING AN ITEM

In the Attributes pane, fill in edit for the “id”. Then, choose both “ifRoom” and “withText” for the “showAsAction” option:

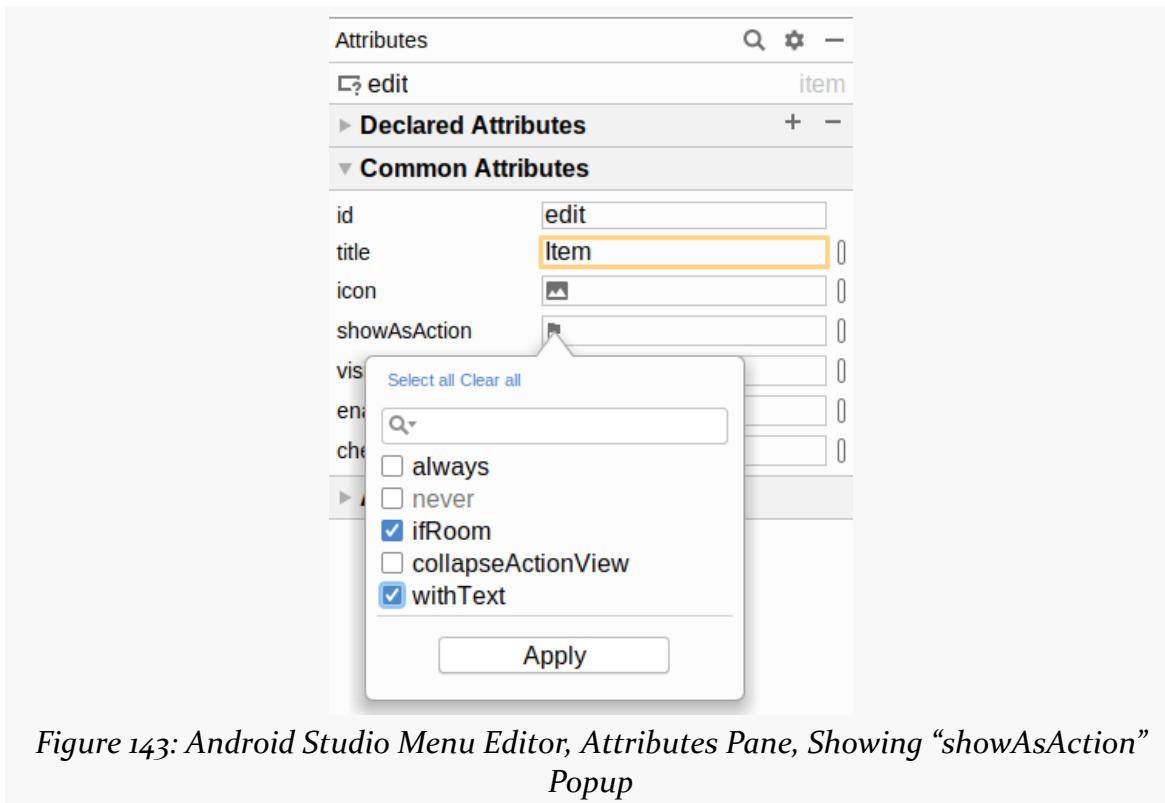


Figure 143: Android Studio Menu Editor, Attributes Pane, Showing “showAsAction” Popup

Click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Open the “Project” category, then click on `ic_edit` in the list of drawables, then click OK to accept that choice of icon.

Then, click the “O” button next to the “title” field. As before, this brings up a string resource selector. Click on “String Value” in the “+” drop-down towards the top. In the dialog, fill in `menu_edit` as the resource name and “Edit” as the resource value. Click OK to close both dialogs.

EDITING AN ITEM

At this point, your menu editor preview should resemble:

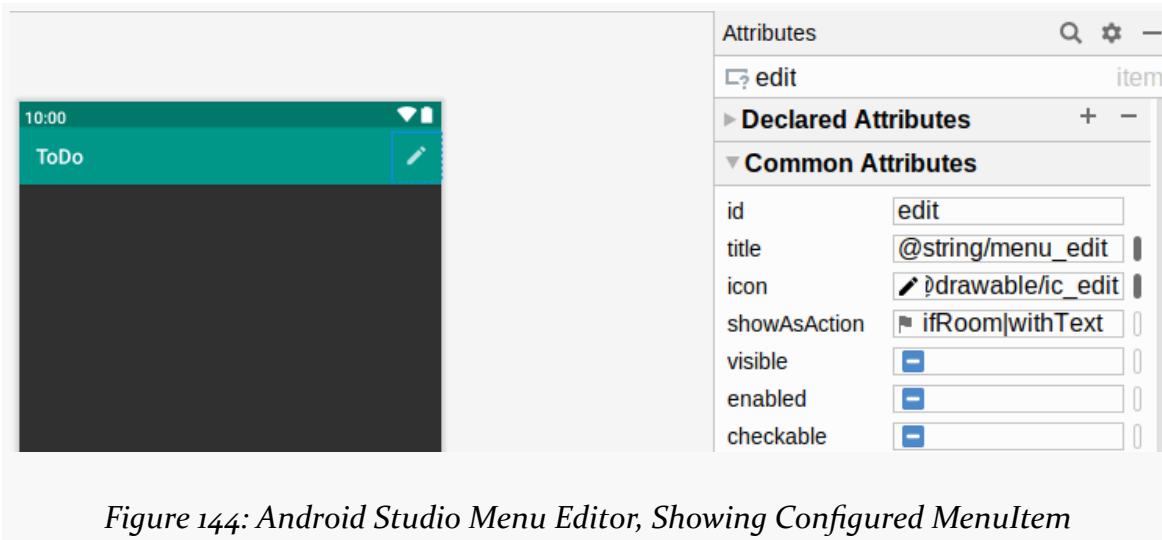


Figure 144: Android Studio Menu Editor, Showing Configured MenuItem

Step #4: Showing the App Bar Item

We also need to take steps to arrange to show this app bar item on `DisplayFragment`. [Previously](#), we defined an app bar item that would be available to the entire activity. Now we want one that will be for just this one fragment. The way to do that is to have the fragment itself add this item to the app bar — Android will only show this item when the fragment itself is visible.

Add this `onCreate()` method to `DisplayFragment`:

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

    setHasOptionsMenu(true)
}
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

`onCreate()` is called when the fragment is created, and here we indicate that we want to add items to the app bar, via `setHasOptionsMenu(true)`.

Next, add this `onCreateOptionsMenu()` method to `DisplayFragment`:

EDITING AN ITEM

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_display, menu)

    super.onCreateOptionsMenu(menu, inflater)
}
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

Here, we use a `MenuInflater` to “inflate” the menu resource and add its item to the app bar. Plus, we chain to the superclass, in case the superclass wants to add things to the app bar as well.

If you run the app and tap on a to-do item in the list, you should see the new app bar item on the `DisplayFragment`:

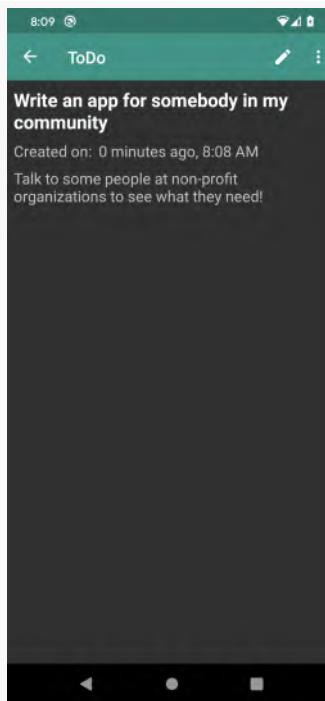


Figure 145: ToDo App, DisplayFragment, with Edit App Bar Item

This is in addition to the overflow menu, which still has our “About” item. By having our activity’s Toolbar serve as the app bar, the activity and the currently-visible fragment(s) can all contribute items.

Step #5: Displaying the (Empty) Fragment

Now that we are displaying the app bar item, we can get control and show the presently-empty `EditFragment`.

First, add this `edit()` function to `DisplayFragment`:

```
private fun edit() {
    findNavController().navigate(
        DisplayFragmentDirections.editModel(
            args.modelId
        )
    )
}
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

As we did in `RosterListFragment`, we use `findNavController()` to get the `NavController` for the navigation graph associated with the `DisplayFragment`. Then, we use `navigate()` to go somewhere. Specifically, we use `DisplayFragmentDirections.editModel()` to invoke the action that we added to `editFragment` in the navigation graph. And, since `editFragment` requires an argument, we supply that model ID to `editModel()`, getting the `modelId` from our own `args`.

Then, add this `onOptionsItemSelected()` function to `DisplayFragment`:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.edit -> {
            edit()
            return true
        }
    }

    return super.onOptionsItemSelected(item)
}
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

Here, if the `MenuItem` is our `edit` one, we call `edit()` and return `true` to indicate that we consumed the event. Otherwise, we chain to the superclass.

If you run the sample app now, and you click on one of the to-do items, and then

EDITING AN ITEM

click on the “edit” app bar item, you will be taken to the empty EditFragment.

If you press BACK when viewing the (empty) EditFragment, you will return to the DisplayFragment, and pressing BACK from there will return you to the list of to-do items.

Step #6: Creating an Empty Layout

As was the case with DisplayFragment, to have EditFragment show the contents of a ToDoModel and allow editing, it helps to have a layout resource.

Right-click over the res/layout/ directory and choose “New” > “Layout resource file” from the context menu. In the dialog that appears, fill in todo_edit as the “File name” and ensure that the “Root element” is set to androidx.constraintlayout.widget.ConstraintLayout. Then, click “OK” to close the dialog and create the mostly-empty resource file.

Step #7: Adding the CheckBox

As with the roster rows — but unlike the DisplayFragment layout — we should have a CheckBox to allow the user to toggle the completion status of the to-do item being edited.

In the graphical designer for todo_edit, drag a CheckBox from the “Buttons” group in the “Palette” pane into the preview area. Use the grab handles to add constraints tying the CheckBox to the top and start sides of the ConstraintLayout:



Figure 146: Android Studio Layout Designer, Showing Added CheckBox

In the Attributes pane, clear out the contents of the “text” attribute, as we just want a bare checkbox, without a caption. Also, in the “Layout” section of the “Attributes” pane, give it 8dp of margin on the top and start sides, via the drop-downs.

Then, switch to the “Code” view and modify the android:id attribute of the <CheckBox> element to have a value of @+id/isCompleted.

EDITING AN ITEM

At this point, the 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">

    <CheckBox
        android:id="@+id/isCompleted"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step #8: Creating the Description Field

The other two things that the user should be able to edit here are the description and the notes. They should not be able to edit the created-on date — that is the date on which the to-do item was created, and so it should not change after creation. For the description and the notes, we will use `EditText` widgets.

Switch back to graphical designer in the layout editor. In the “Palette” pane, in the “Text” category, drag a “Plain Text” widget into the design area. Using the grab handles, set up constraints to:

- Tie the top and end sides of the `EditText` to the top and end sides of the `ConstraintLayout`
- Connect the start side of the `EditText` to the end side of the `CheckBox`

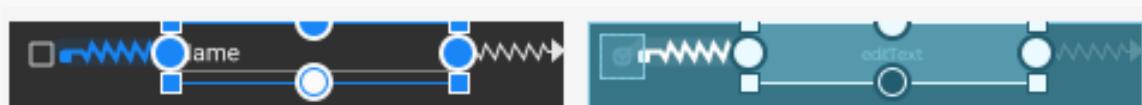


Figure 147: Android Studio Layout Designer, Showing Added `EditText`

EDITING AN ITEM

Next, change the “layout_width” attribute in the Attributes pane to match_constraint:



Figure 148: Android Studio Layout Designer, Showing Stretched EditText

Then, in the “Layout” section of the “Attributes” pane, give it 8dp of margin on the top, start, and end sides, via the drop-downs.

Next, switch to the “Code” view, and change the android:id value for this <EditText> to be @+id/desc. Afterwards, switch back to the “Design” view.

If you look closely, you will see that our CheckBox is not very well aligned vertically with respect to the EditText:

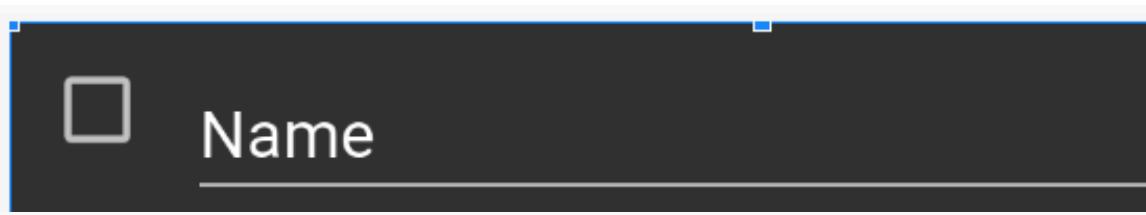


Figure 149: Android Studio Layout Designer, Showing Vertical Alignment Problem

Ideally, it would be vertically centered. To do that, re-drag a constraint from the top of the CheckBox to the top of the EditText. Then, create a similar constraint, from the bottom of the CheckBox to the bottom of the EditText.

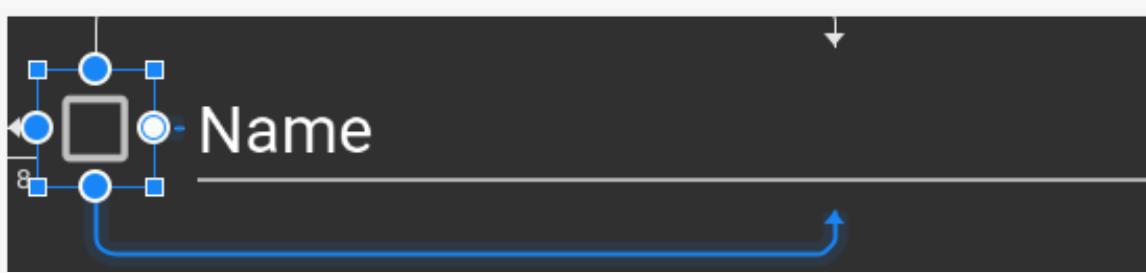


Figure 150: Android Studio Layout Designer, Showing Aligned Widgets

EDITING AN ITEM

In the Attributes pane, the “Plain Text” widget that we dragged into the preview gave us an EditText set up with an “inputType” of textPersonName:

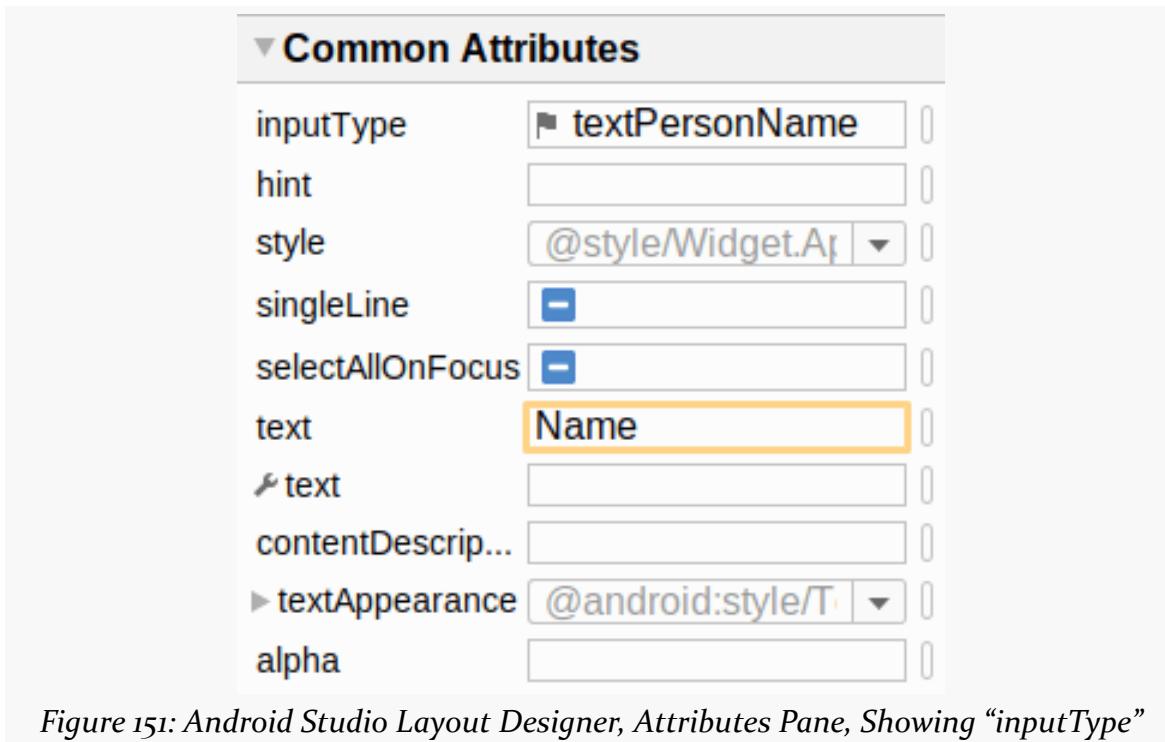


Figure 151: Android Studio Layout Designer, Attributes Pane, Showing “inputType”

EDITING AN ITEM

The `android:inputType` attribute provides hints to soft keyboards as to what we expect to use as input. For example, in languages where there is a distinction between uppercase and lowercase letters, `textPersonName` might trigger a switch to an uppercase keyboard for each portion of a name. In this case, we really want plain text, so click on the flag adjacent to `textPersonName`. Then, in the pop-up panel that appears, uncheck `textPersonName` and check `text`:

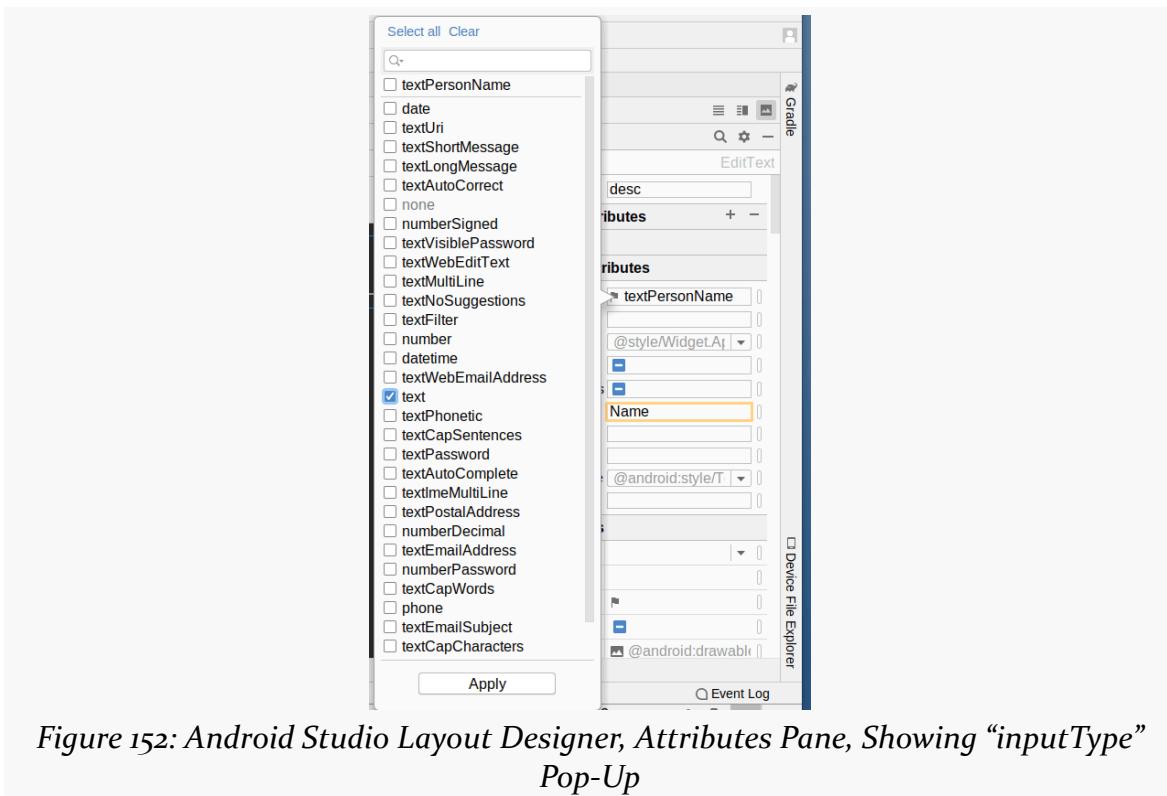


Figure 152: Android Studio Layout Designer, Attributes Pane, Showing “`inputType`” Pop-Up

Then, click the “Apply” button in the popup to close that popup.

An `EditText` has an `android:hint` attribute. This provides some text that will appear in the field in gray when there is no actual text entered by the user in the field. Once the user starts typing, the hint goes away. This is used to save space over having a separate label or caption for the field. With that in mind, click the “O” button for the “hint” attribute in the Attributes pane. Create a new string resource using the drop-down menu. Give the resource a name of `desc` and a value of `Description`. Then, click OK to define the string resource and apply it to the hint.

Finally, clear out the “text” attribute, as we will set that at runtime.

EDITING AN ITEM

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">

    <CheckBox
        android:id="@+id/isCompleted"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        app:layout_constraintBottom_toBottomOf="@+id/desc"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="@+id/desc" />

    <EditText
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:ems="10"
        android:hint="@string/desc"
        android:inputType="text"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/isCompleted"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Step #9: Adding the Notes Field

The other widget is another `EditText`, this time for the notes.

Switch back to the graphical designer in the layout editor. In the “Palette” pane, in the “Text” category, drag a “Multiline Text” widget into the design area. Using the grab handles, set up constraints to:

EDITING AN ITEM

- Tie the bottom, start, and end sides of the `EditText` to the bottom, start, and end sides of the `ConstraintLayout`
- Tie the top of the `EditText` to the bottom of the previous `EditText`

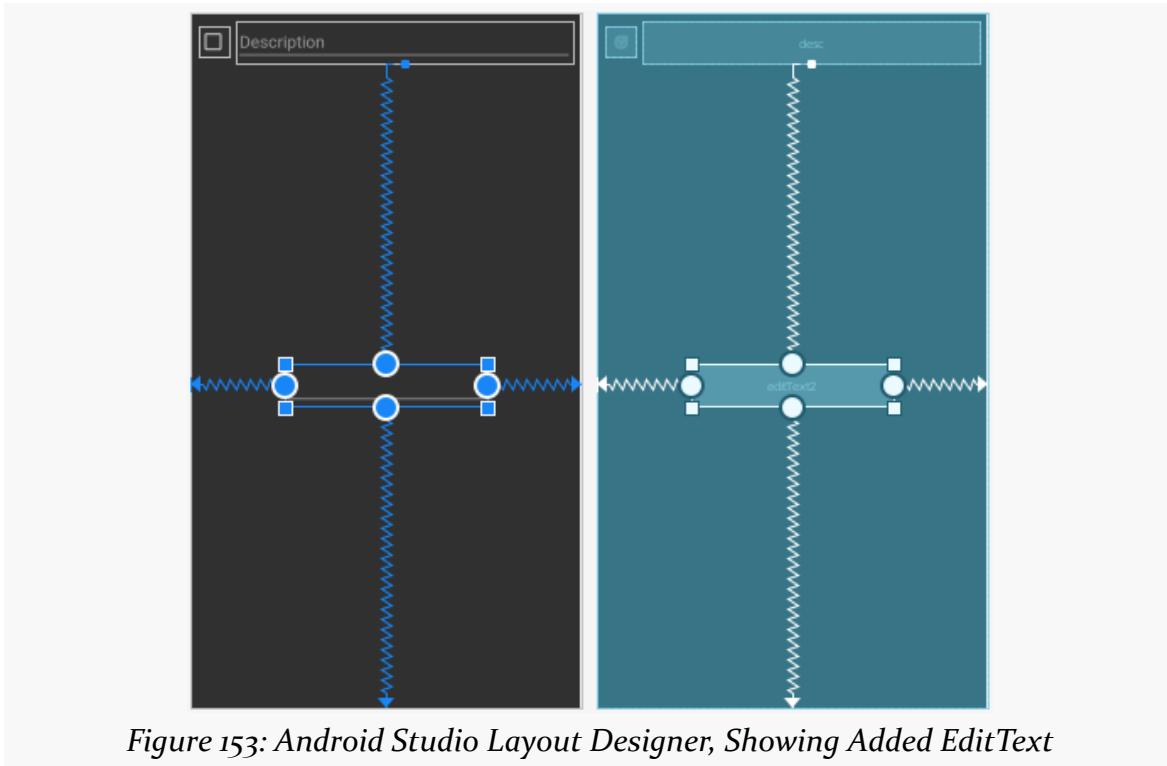


Figure 153: Android Studio Layout Designer, Showing Added `EditText`

EDITING AN ITEM

Then, set both the “layout_height” and “layout_width” attributes to `match_constraint`:

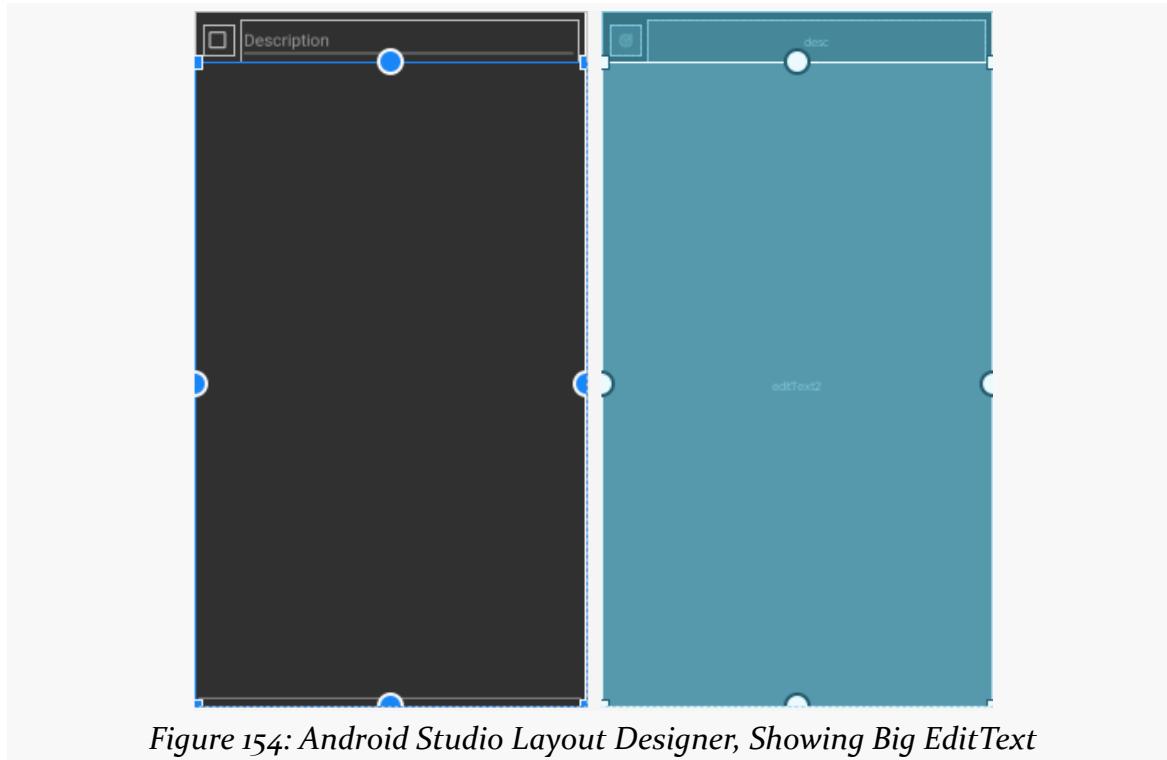


Figure 154: Android Studio Layout Designer, Showing Big EditText

In the “Layout” section of the “Attributes” pane, give it 8dp of margin on all four sides, via the drop-downs.

Next, switch to the “Code” view, and change the `android:id` value for this new `<EditText>` to be `@+id/notes`. Afterwards, switch back to the “Design” view.

Click the “O” button for the “hint” attribute in the Attributes pane. Create a new string resource using the drop-down menu. Give the resource a name of `notes` and a value of `Notes`. Then, click OK to define the string resource and apply it to the hint.

Step #10: Populating the Layout

Now, we can add the same sort of logic in `EditFragment` to bind the `ToDoModel` that we added to `DisplayFragment`.

In `EditFragment`, add properties for our binding, our navigation arguments, and our

EDITING AN ITEM

viewmodel:

```
private var binding: TodoEditBinding? = null
private val args: EditFragmentArgs by navArgs()
private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

(from T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt)
```

Then, add functions to inflate the binding, bind our model, and clear the binding:

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
) = TodoEditBinding.inflate(inflater, container, false)
    .apply { binding = this }
    .root

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    motor.getModel()?.let {
        binding?.apply {
            isCompleted.isChecked = it.isCompleted
            desc.setText(it.description)
            notes.setText(it.notes)
        }
    }
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

EDITING AN ITEM

If you run the app, click on a to-do item to display it, then click on the “edit” app bar item, you will get a form for modifying the to-do item:

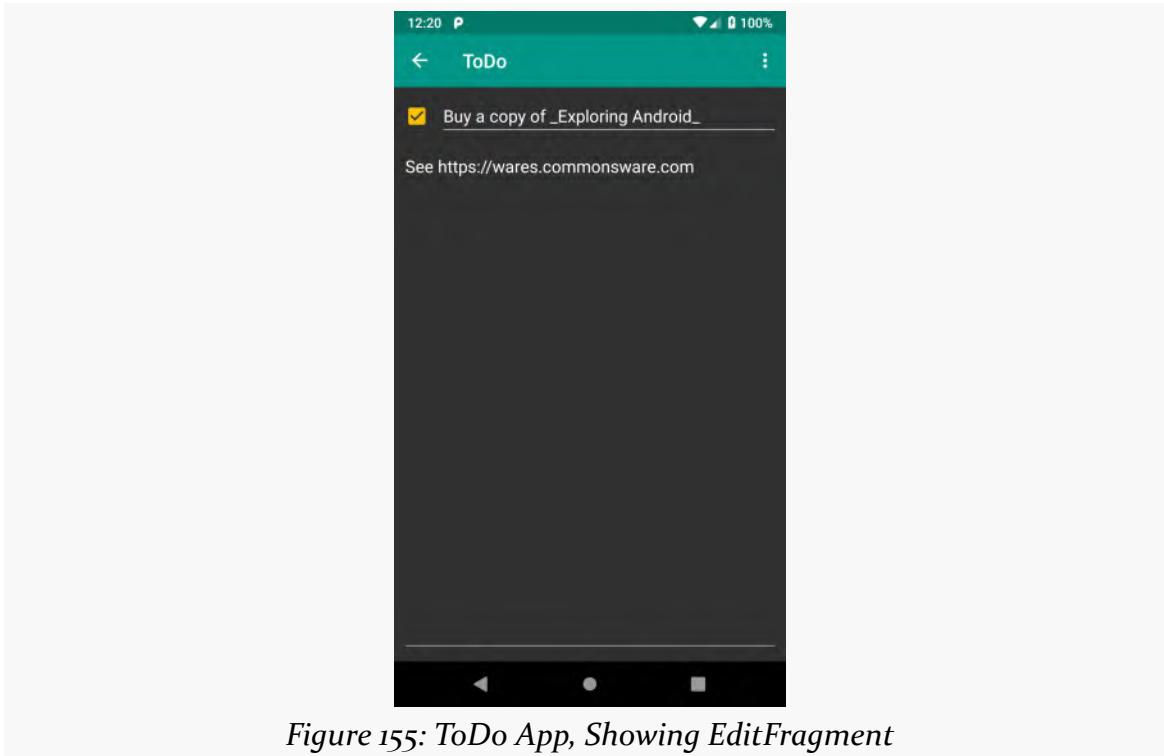


Figure 155: ToDo App, Showing EditFragment

Note that `EditText` only word-wraps when set up for multiline. Otherwise, long text just scrolls off the end. This is perfectly normal.

A bigger problem is that our changes are not being reflected anywhere. For that, we will need to update our models, and we will deal with that in the next tutorial.

Final Results

Your `nav_graph` navigation resource should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph.xml"
    app:startDestination="@+id/rosterListFragment">

    <fragment
        android:id="@+id/rosterListFragment"
```

EDITING AN ITEM

```
    android:name="com.commonsware.todo.RosterListFragment"
    android:label="@string/app_name">
    <action
        android:id="@+id/displayModel"
        app:destination="@+id/displayFragment" />
</fragment>
<fragment
    android:id="@+id/displayFragment"
    android:name="com.commonsware.todo.DisplayFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string" />
    <action
        android:id="@+id/editModel"
        app:destination="@+id/editFragment" />
</fragment>
<fragment
    android:id="@+id/editFragment"
    android:name="com.commonsware.todo.EditFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string" />
</fragment>
</navigation>
```

(from [T18-Edit/ToDo/app/src/main/res/navigation/nav_graph.xml](#))

The new actions_display resource should have this XML:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/edit"
        android:icon="@drawable/ic_edit"
        android:title="@string/menu_edit"
        app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T18-Edit/ToDo/app/src/main/res/menu/actions_display.xml](#))

The new todo_edit layout resource should resemble:

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

EDITING AN ITEM

```
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:app="http://schemas.android.com/apk/res-auto"
android:layout_width="match_parent"
android:layout_height="match_parent">

<CheckBox
    android:id="@+id/isCompleted"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    app:layout_constraintBottom_toBottomOf="@+id/desc"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="@+id/desc" />

<EditText
    android:id="@+id/desc"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:ems="10"
    android:hint="@string/desc"
    android:inputType="text"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/isCompleted"
    app:layout_constraintTop_toTopOf="parent" />

<EditText
    android:id="@+id/notes"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginBottom="8dp"
    android:ems="10"
    android:gravity="start|top"
    android:hint="@string/notes"
    android:inputType="textMultiLine"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/desc"
    app:layout_constraintVertical_bias="0.505" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [T18-Edit/ToDo/app/src/main/res/layout/todo_edit.xml](#))

EDITING AN ITEM

DisplayFragment should look like:

```
package com.commonsware.todo

import android.os.Bundle
import android.text.format.DateUtils
import android.view.*
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.databinding.TodoDisplayBinding
import org.koin.androidx.viewmodel.ext.android.viewModel
import org.koin.core.parameter.parametersOf

class DisplayFragment : Fragment() {
    private val args: DisplayFragmentArgs by navArgs()
    private var binding: TodoDisplayBinding? = null
    private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) = TodoDisplayBinding.inflate(inflater, container, false)
        .apply { binding = this }
        .root

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        motor.getModel()?.let {
            binding?.apply {
                completed.visibility = if (it.isCompleted) View.VISIBLE else View.GONE
                desc.text = it.description
                createdOn.text = DateUtils.getRelativeDateTimeString(
                    requireContext(),
                    it.createdOn.toEpochMilli(),
                    DateUtils.MINUTE_IN_MILLIS,
                    DateUtils.WEEK_IN_MILLIS,
                    0
                )
                notes.text = it.notes
            }
        }
    }
}
```

EDITING AN ITEM

```
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_display, menu)

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.edit -> {
            edit()
            return true
        }
    }

    return super.onOptionsItemSelected(item)
}

private fun edit() {
    findNavController().navigate(
        DisplayFragmentDirections.editModel(
            args.modelId
        )
    )
}
}
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

EditFragment at this point should resemble:

```
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
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.databinding.TodoEditBinding
import org.koin.androidx.viewmodel.ext.android.viewModel
```

EDITING AN ITEM

```
import org.koin.core.parameter.parametersOf

class EditFragment : Fragment() {
    private var binding: TodoEditBinding? = null
    private val args: EditFragmentArgs by navArgs()
    private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) = TodoEditBinding.inflate(inflater, container, false)
        .apply { binding = this }
        .root

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        motor.getModel()?.let {
            binding?.apply {
                isCompleted.isChecked = it.isCompleted
                desc.setText(it.description)
                notes.setText(it.notes)
            }
        }
    }

    override fun onDestroyView() {
        binding = null

        super.onDestroyView()
    }
}
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/java/com/commonsware/todo/EditFragment.kt](#)
- [app/src/main/res/navigation/nav_graph.xml](#)
- [app/src/main/res/drawable/ic_edit.xml](#)
- [app/src/main/res/menu/actions_display.xml](#)
- [app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#)
- [app/src/main/res/layout/todo_edit.xml](#)

Saving an Item

Having the `EditFragment` is nice, but we are not saving the changes anywhere. As soon as we leave the fragment, the “edits” vanish.

This is not ideal.

So, in this tutorial, we will allow the user to save their changes, by clicking a suitable app bar item.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding the App Bar Item

First, let’s set up the Save app bar item.

SAVING AN ITEM

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Clip Art” button and search for `save`:

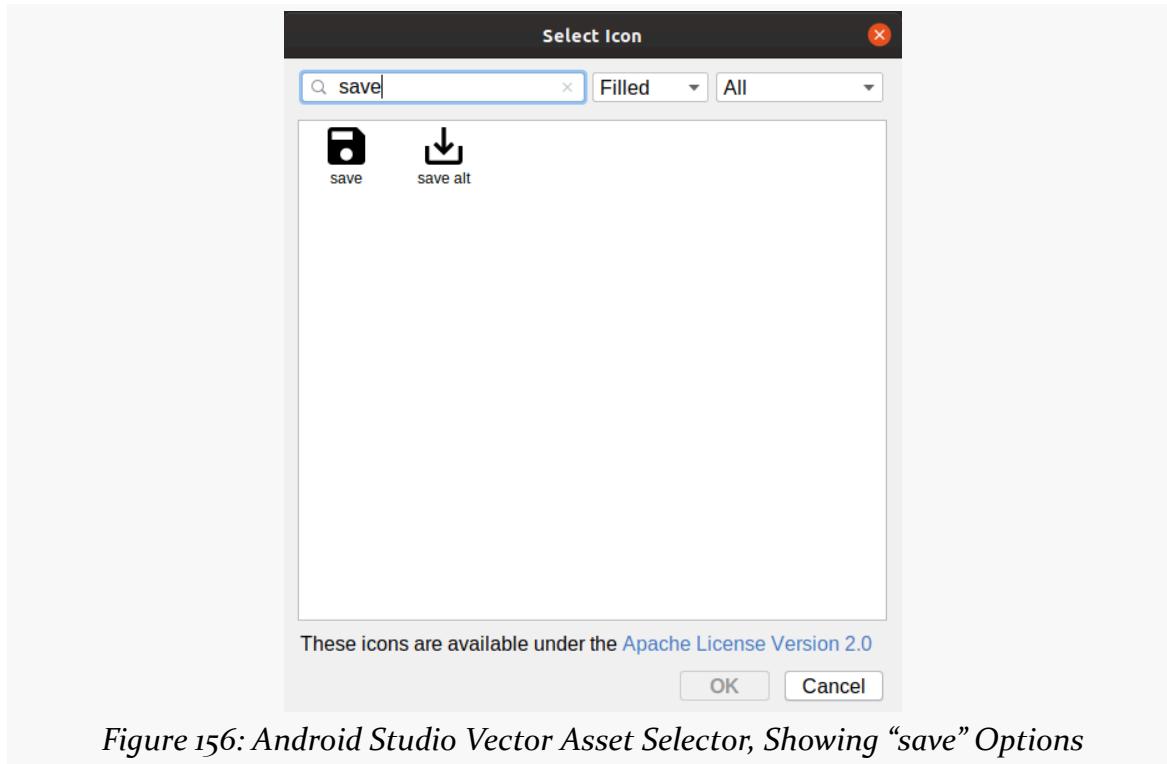


Figure 156: Android Studio Vector Asset Selector, Showing “save” Options

Choose the “`save`” icon and click “OK” to close up the icon selector. Change the name of the icon to `ic_save`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

Then, right-click over the `res/menu/` directory and choose New > “Menu resource file” from the context menu. Fill in `actions_edit.xml` in the “New Menu Resource File” dialog, then click OK to create the file and open it in the menu editor.

In the Palette, drag a “Menu Item” into the preview area. This will appear as an item in an overflow area as before.

In the Attributes pane, fill in `save` for the “`id`”. Then, choose both “`ifRoom`” and

SAVING AN ITEM

“withText” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector — click on `ic_save` in the list of drawables, then click OK to accept that choice of icon.

Then, click the “O” button next to the “title” field. As before, this brings up a string resource selector. Click on “+”, then click on “String Value” in the resulting dropdown. In the dialog, fill in `menu_save` as the resource name and “Save” as the resource value. Click OK to close the dialog, to complete our work on setting up the app bar item:

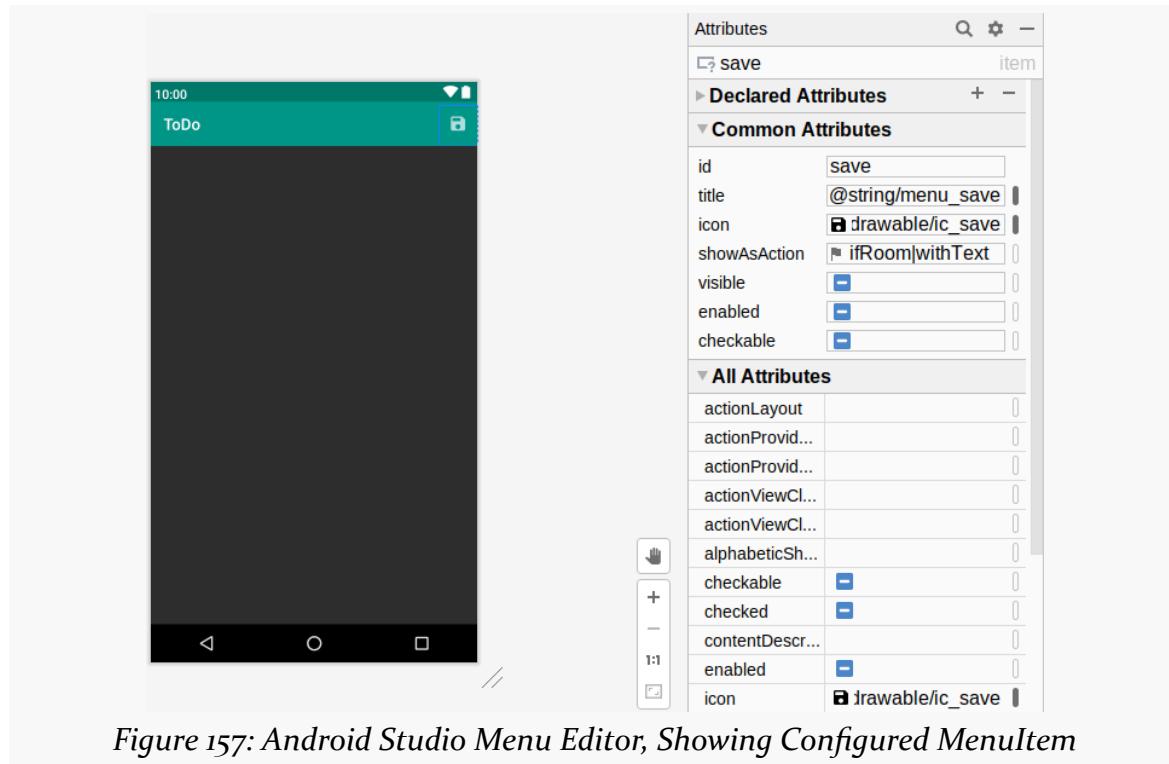


Figure 157: Android Studio Menu Editor, Showing Configured MenuItem

We also need to take steps to arrange to show this app bar item on `EditFragment`, as we did with `DisplayFragment` for the “edit” item.

Add this `onCreate()` function to `EditFragment`, to indicate that this fragment wishes to participate in the app bar:

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

    setHasOptionsMenu(true)
}
```

SAVING AN ITEM

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

Next, add this `onCreateOptionsMenu()` function to `EditFragment`, to inflate our newly-created menu resource:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {  
    inflater.inflate(R.menu.actions_edit, menu)  
  
    super.onCreateOptionsMenu(menu, inflater)  
}
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

If you run the app and edit a to-do item, you should see the new app bar item on the `EditFragment`:

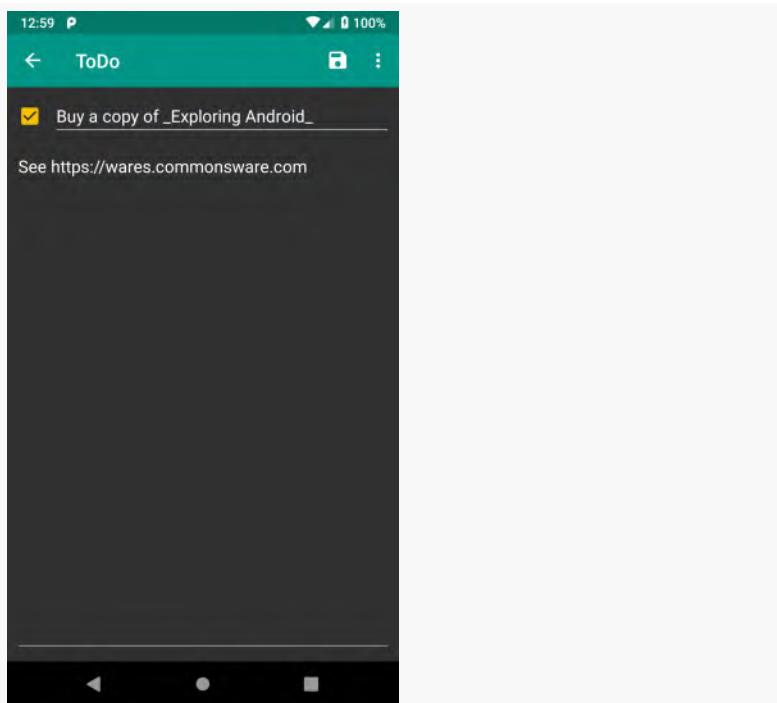


Figure 158: ToDo App, `EditFragment`, with Save App Bar Item

Step #2: Improving the Motor

We need to be able to save changes from `EditFragment`. We already have logic in `ToDoRepository` for this, in the form of a `save()` function that we used from `RosterMotor`. However, `EditFragment` is using `SingleModelMotor`, which right now

SAVING AN ITEM

lacks a `save()` function.

So... let's add it!

Add this `save()` function to `SingleModelMotor`, perhaps by copying it from `RosterMotor`:

```
fun save(model: ToDoModel) {  
    repo.save(model)  
}
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#))

Step #3: Replacing the Item

Now that we have the app bar item, we can get control when it is clicked and update our repository with a revised `ToDoModel`.

Create this `save()` function on `EditFragment`:

```
private fun save() {  
    binding?.apply {  
        val model = motor.getModel()  
        val edited = model?.copy(  
            description = desc.text.toString(),  
            isCompleted = isCompleted.isChecked,  
            notes = notes.text.toString()  
        ) ?: ToDoModel(  
            description = desc.text.toString(),  
            isCompleted = isCompleted.isChecked,  
            notes = notes.text.toString()  
        )  
  
        edited.let { motor.save(it) }  
    }  
}
```

Here we:

- Retrieve our current `ToDoModel` from the viewmodel
- Use the `copy()` function on our data class to create a revised instance of `ToDoModel` with the data from the form
- If, inexplicably, `getModel()` returned null, create a new `ToDoModel` with the data from the form

SAVING AN ITEM

- Tell the `SingleModelMotor` to replace the existing `ToDoModel` for this ID with this new or revised model

Then, arrange to call this `save()` method when the user clicks on the “save” app bar item, by adding this `onOptionsItemSelected()` function to `EditFragment`:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.save -> {
            save()
            return true
        }
    }

    return super.onOptionsItemSelected(item)
}
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

If you run the app, select some to-do item, make some change to that item, then click that app bar item... nothing seems to happen. But, if you then press BACK to return to the `DisplayFragment`, then BACK again to return to the list, you will see that... the list appears to be unchanged.

These are problems. And, we will fix them in the next couple of steps.

Step #4: Returning to the Display Fragment

The “nothing seems to happen” bit from the preceding step is a problem. Usually, when the user clicks a “save” option in an app, not only does the data get saved, but the user is taken to some other portion of the app. In the case of `EditFragment`, we could send the user back to the `DisplayFragment` that they came from.

With that in mind, add this `navToDisplay()` function to `EditFragment`:

```
private fun navToDisplay() {
    findNavController().popBackStack()
}
```

This simply “pops the back stack”, doing the same thing as if the user pressed the device BACK button or clicked the “up” arrow in the app bar.

Then, add a call to `navToDisplay()` from `save()`:

SAVING AN ITEM

```
private fun save() {
    binding?.apply {
        val model = motor.getModel()
        val edited = model?.copy(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        ) ?: ToDoModel(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        )
        edited.let { motor.save(it) }
    }

    navToDisplay()
}
```

(from [Tip-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

If you run the sample app now, when you click “save” in the `EditFragment`, you go back to the `DisplayFragment`.

However, if you are using a device with a soft keyboard, that soft keyboard may still be visible after clicking “save”. This is annoying. But, with some trickery, we can fix it.

Add this function to `EditFragment`:

```
private fun hideKeyboard() {
    view?.let {
        val imm = context?.getSystemService<InputMethodManager>()

        imm?.hideSoftInputFromWindow(
            it.windowToken,
            InputMethodManager.HIDE_NOT_ALWAYS
        )
    }
}
```

(from [Tip-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

This method, based on [this Stack Overflow answer](#), hides the soft keyboard (a.k.a., “input method editor”). This code is clunky but is unavoidable. Basically, we get the `InputMethodManager` system service and call `hideSoftInputFromWindow()` on it. Note that the `getSystemService()` function we are using is an extension function

SAVING AN ITEM

provided by Android KTX, the Kotlin extension functions supplied by the Jetpack:

```
import androidx.core.content.getSystemService
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

Then, modify navToDisplay() in EditFragment to call hideKeyboard():

```
private fun navToDisplay() {
    hideKeyboard()
    findNavController().popBackStack()
}
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

Now, if you run the sample app and edit a to-do item, saving your changes both returns you to the DisplayFragment *and* hides the soft keyboard if needed.

However, we still do not see the changes from our edits. The list appears as it did originally.

Step #5: Getting Updated Items

The problem lies in RosterMotor. There, we have an items property, populated from the repository's list of items:

```
val items = repo.items
```

(from [T18-Edit/ToDo/app/src/main/java/com/commonsware/todo/RosterMotor.kt](#))

When we return to our RosterListFragment, we go through onCreateView() and onViewCreated() again. We ask the motor for the items and display them in the list. The problem is that we still have the same RosterMotor instance as we did originally, and it has the same items collection as we did originally. However, we *replaced* the items collection in the repository... but RosterMotor does not know about this.

Ideally, we would be using some sort of “reactive” system that allows RosterMotor to respond as soon as ToDoRepository has updated contents. We will switch to that sort of solution later in the book. For now, though, we can settle for simply getting the list from the repository every time.

With that in mind, replace the items property in RosterMotor with a nearly-identical getItems() function:

SAVING AN ITEM

```
fun getItems() = repo.items
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/RosterMotor.kt](#))

Then, in RosterListFragment, change our submitList() call in onViewCreated() to use this new function, instead of the now-replaced items property:

```
adapter.submitList(motor.getItems())
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Now, when you save changes to an item, you return to the list, and the list will reflect the changes that you made.

Final Results

The new actions_edit menu resource should contain:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/save"
        android:icon="@drawable/ic_save"
        android:title="@string/menu_save"
        app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T19-Save/ToDo/app/src/main/res/menu/actions_edit.xml](#))

EditFragment should resemble:

```
package com.commonsware.todo

import android.os.Bundle
import android.view.*
import android.view.inputmethod.InputMethodManager
import androidx.core.content.getSystemService
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.databinding.TodoEditBinding
import org.koin.androidx.viewmodel.ext.android.viewModel
import org.koin.core.parameter.parametersOf
```

SAVING AN ITEM

```
class EditFragment : Fragment() {
    private var binding: TodoEditBinding? = null
    private val args: EditFragmentArgs by navArgs()
    private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) = TodoEditBinding.inflate(inflater, container, false)
        .apply { binding = this }
        .root

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        motor.getModel()?.let {
            binding?.apply {
                isCompleted.isChecked = it.isCompleted
                desc.setText(it.description)
                notes.setText(it.notes)
            }
        }
    }

    override fun onDestroyView() {
        binding = null

        super.onDestroyView()
    }

    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        inflater.inflate(R.menu.actions_edit, menu)

        super.onCreateOptionsMenu(menu, inflater)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        when (item.itemId) {
            R.id.save -> {
                save()
                return true
            }
        }
    }
}
```

SAVING AN ITEM

```
    return super.onOptionsItemSelected(item)
}

private fun save() {
    binding?.apply {
        val model = motor.getModel()
        val edited = model?.copy(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        ) ?: ToDoModel(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        )
        edited.let { motor.save(it) }
    }

    navToDisplay()
}

private fun navToDisplay() {
    hideKeyboard()
    findNavController().popBackStack()
}

private fun hideKeyboard() {
    view?.let {
        val imm = context?.getSystemService<InputMethodManager>()

        imm?.hideSoftInputFromWindow(
            it.windowToken,
            InputMethodManager.HIDE_NOT_ALWAYS
        )
    }
}
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

RosterMotor should look like:

```
package com.commonsware.todo

import androidx.lifecycle.ViewModel
```

SAVING AN ITEM

```
class RosterMotor(private val repo: ToDoRepository) : ViewModel() {  
    fun getItems() = repo.items  
  
    fun save(model: ToDoModel) {  
        repo.save(model)  
    }  
}
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/RosterMotor.kt](#))

And RosterListFragment should resemble:

```
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  
import androidx.recyclerview.widget.DividerItemDecoration  
import androidx.recyclerview.widget.LinearLayoutManager  
import com.commonsware.todo.databinding.TodoRosterBinding  
import org.koin.androidx.viewmodel.ext.android.viewModel  
import androidx.navigation.fragment.findNavController  
  
class RosterListFragment : Fragment() {  
    private val motor: RosterMotor by viewModel()  
    private var binding: TodoRosterBinding? = null  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View = TodoRosterBinding.inflate(inflater, container, false)  
        .also { binding = it }  
        .root  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
  
        val adapter = RosterAdapter(  
            layoutInflater,  
            onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },  
            onRowClick = ::display)  
  
        binding?.items?.apply {  
            setAdapter(adapter)  
            layoutManager = LinearLayoutManager(context)
```

SAVING AN ITEM

```
addItemDecoration(  
    DividerItemDecoration(  
        activity,  
        DividerItemDecoration.VERTICAL  
    )  
)  
  
}  
  
adapter.submitList(motor.getItems())  
binding?.empty?.visibility = View.GONE  
}  
  
override fun onDestroyView() {  
    binding = null  
  
    super.onDestroyView()  
}  
  
private fun display(model: ToDoModel) {  
    findNavController()  
    .navigate(RosterListFragmentDirections.displayModel(model.id))  
}  
}
```

(from [T19-Save/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/res/drawable/ic_save.xml](#)
- [app/src/main/res/menu/actions_edit.xml](#)
- [app/src/main/java/com/commonsware/todo/EditFragment.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)

Adding and Deleting Items

Now, we can edit our to-do items. However, the app is still pretty limited, in that we can only have *exactly* three to-do items. While we can now change what appears in those to-do items, we cannot add or remove any.

We really should fix that.

So, in this tutorial, we will wrap up the “glassware” portion of the app, by getting rid of the fake starter data and giving the user the ability to add new to-do items and delete existing ones.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Trimming Our Repository

First, let’s get rid of the sample data. That is merely a matter of changing the `items` declaration in `ToDoRepository` to be:

```
var items = emptyList<ToDoModel>()
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

We already have a `save()` function in the repository to add items, but we need a function to remove them as well. To that end, add this `delete()` function to `ToDoRepository`:

ADDING AND DELETING ITEMS

```
fun delete(model: ToDoModel) {  
    items = items.filter { it.id != model.id }  
}
```

(from [ToDo-Add/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

Here, we just replace `items` with a filtered edition of `items`, keeping any item that has an ID different than the one that we are trying to remove.

If you now run the sample app, it runs, but we have no to-do items:

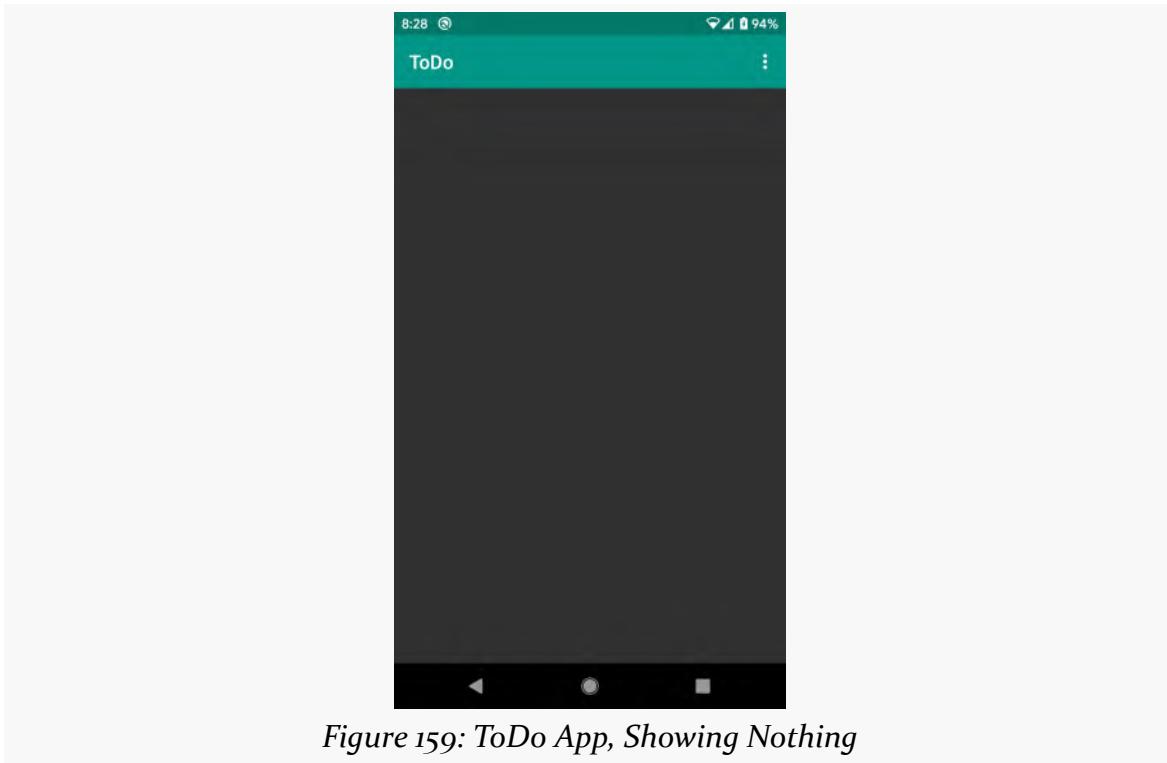


Figure 159: ToDo App, Showing Nothing

Step #2: Showing an Empty View

Dumping the user onto an empty screen at the outset is rather unfriendly. A typical solution is to have an “empty view” that is displayed when there is nothing else to show. That “empty view” usually has a message that tells the user what to do first.

We created the empty view back in [an earlier tutorial](#), but we set its visibility to GONE. Let’s revert that change, so the empty view appears to the user.

ADDING AND DELETING ITEMS

In onViewCreated() of RosterListFragment, remove the binding.empty.visibility = View.GONE line, leaving you with:

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

    val adapter = RosterAdapter(
        layoutInflater,
        onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
        onRowClick = ::display)

    binding.items.apply {
        setAdapter(adapter)
        layoutManager = LinearLayoutManager(context)

        addItemDecoration(
            DividerItemDecoration(
                activity,
                DividerItemDecoration.VERTICAL
            )
        )
    }

    adapter.submitList(motor.items)
}
```

ADDING AND DELETING ITEMS

Now when you run the app, you will see... some placeholder text:

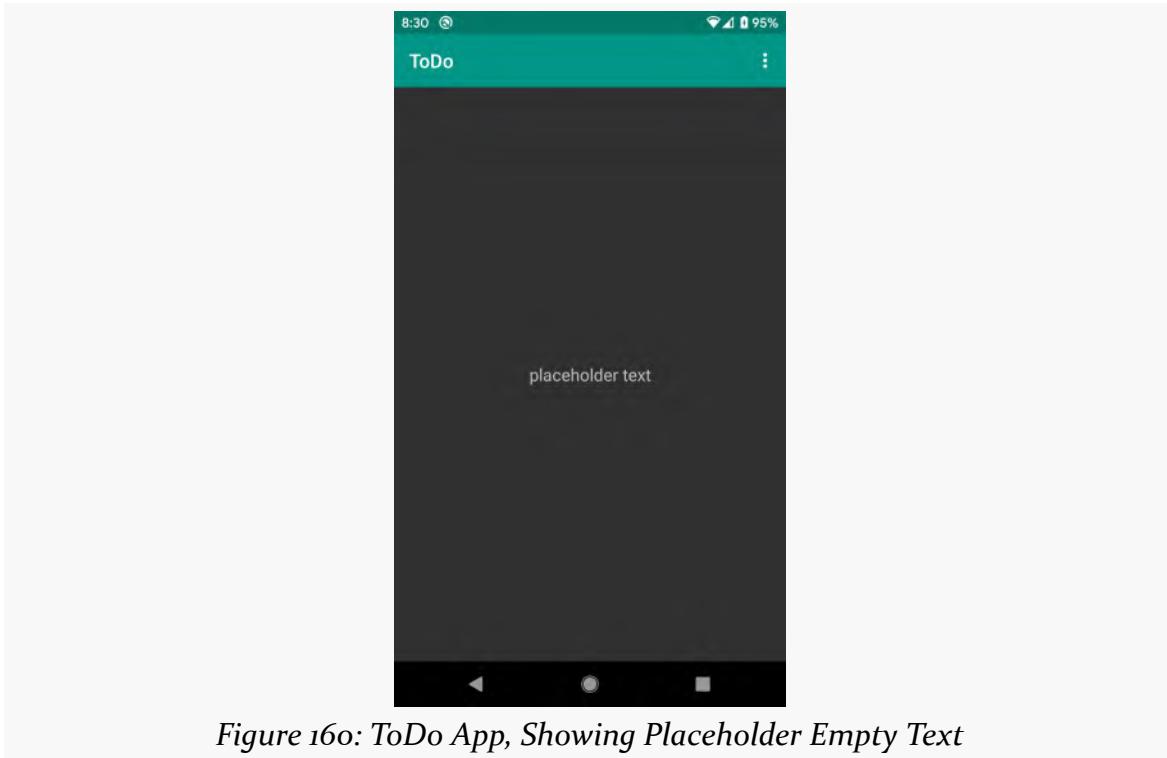


Figure 16o: ToDo App, Showing Placeholder Empty Text

We will replace that text with a better message shortly.

Step #3: Adding an Add App Bar Item

We need to add another app bar item, this one in the roster fragment, to allow the user to add a new to-do item.

ADDING AND DELETING ITEMS

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Clip Art” button and search for add:

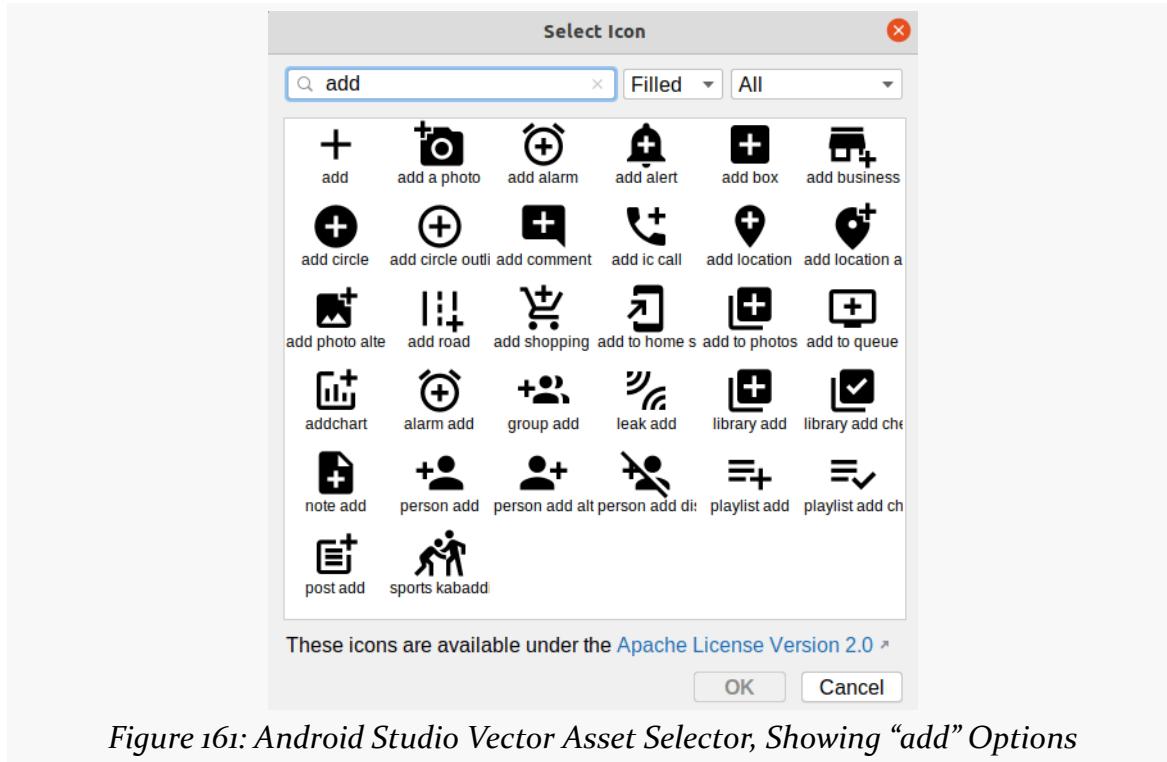


Figure 161: Android Studio Vector Asset Selector, Showing “add” Options

(we really like to add things in Android...)

Choose the “add” icon and click “OK” to close up the icon selector. Change the name to `ic_add`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

While it feels like we keep adding app bar items, we have never added one directly to the `RosterListFragment`. All previous app bar items were added to the other fragments or to `MainActivity`. So, we need to set up a new menu resource and the corresponding Kotlin code.

Right-click over the `res/menu/` directory and choose New > “Menu resource file” from the context menu. Fill in `actions_roster.xml` in the “New Menu Resource

ADDING AND DELETING ITEMS

File” dialog, then click OK to create the file to open it in the menu editor.

In the Palette, drag a “Menu Item” into the preview area. In the Attributes pane, fill in add for the “id”. Then, choose both “ifRoom” and “withText” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Click on ic_add in the list of drawables, then click OK to accept that choice of icon.

Then, click the “O” button next to the “title” field. As before, this brings up a string resource selector. Click on “+”, then click on “String Value” in the resulting drop-down. In the dialog, fill in menu_add as the resource name and “Add” as the resource value. Click OK to close the dialog and complete the configuration of this app bar item.

Add this `onCreate()` function to `RosterListFragment`, to indicate that this fragment wishes to participate in the app bar:

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

    setHasOptionsMenu(true)
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Next, add this `onCreateOptionsMenu()` function to `RosterListFragment`, to inflate our newly-created menu resource:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)

    super.onCreateOptionsMenu(menu, inflater)
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Finally, open up the `res/values/strings.xml` resource file. You should find a string resource named `msg_empty` in there, with a value of `placeholder_text`. Replace that value with `Click the + icon to add a todo item!`.

ADDING AND DELETING ITEMS

Now, when you run the app, not only do you get the “add” app bar item, but the empty view text is more useful:

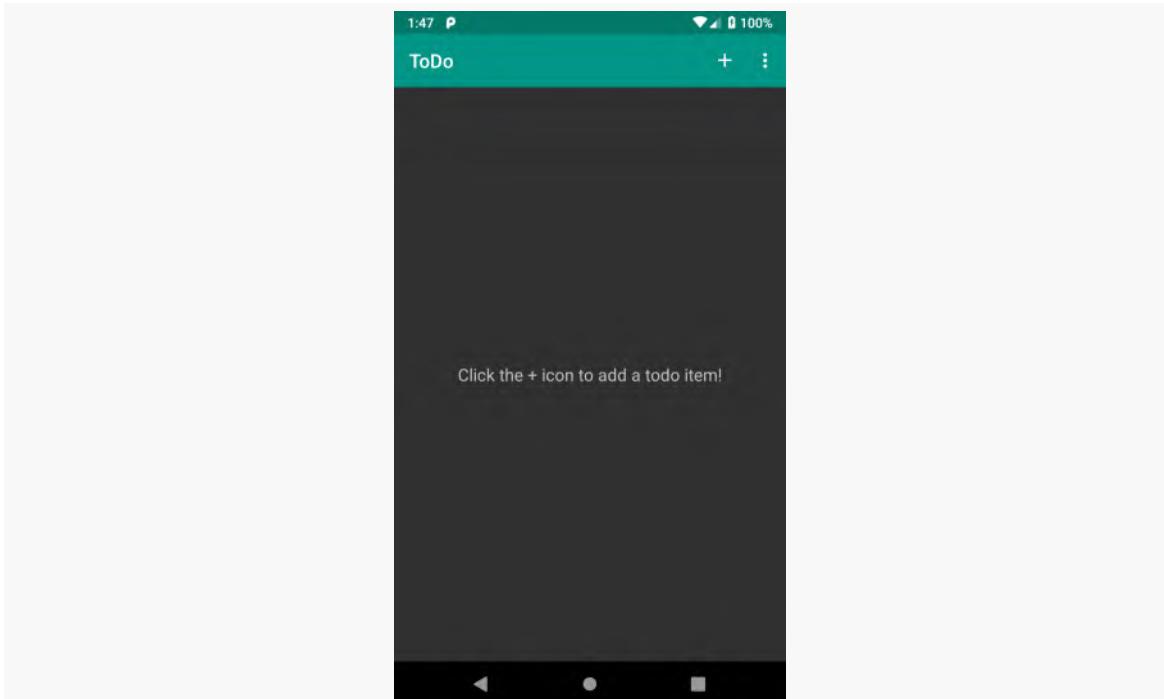


Figure 162: ToDo App, with Add App Bar Item and Better Empty Text

Step #4: Launching the EditFragment for Adds

Next, we need to add some logic to do some work when the user taps that “add” app bar item. Specifically, we want to navigate from the `RosterListFragment` to the `EditFragment`... but do so in a way that tells the `EditFragment` that we should be adding a new to-do item, not editing an existing one.

Right now, to navigate to the `EditFragment`, we need to provide a `modelId` value, identifying the existing to-do item to be edited. In this case, though, we do not have an existing to-do item — we want to create a new one. So, we can change the navigation graph to allow `modelId` to support `null` as a value. Then, we can have a `null` `modelId` indicate that we are creating a new to-do item, while a non-null `modelId` would indicate that we are editing an existing to-do item.

With all that in mind, open `res/navigation/nav_graph.xml`, and click on the `editFragment` destination. In the “Attributes” pane, we have our `modelId` argument.

ADDING AND DELETING ITEMS

Double-click on it to bring up a dialog to update its configuration:

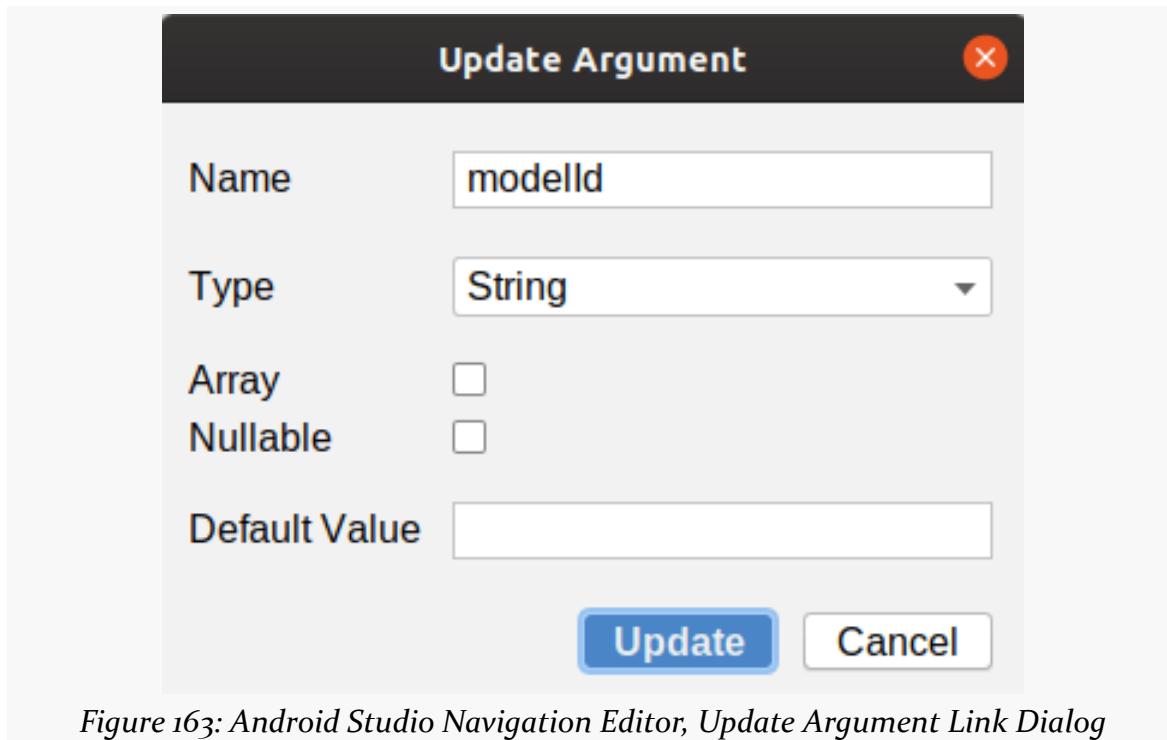


Figure 163: Android Studio Navigation Editor, Update Argument Link Dialog

Check the “Nullable” checkbox, then click Update to close the dialog.

ADDING AND DELETING ITEMS

Next, click on the rosterListFragment destination. Using the circle on the right edge, drag a new action, connecting it to editFragment. When you have done that, you may want to click the toolbar button that looks like... well... plusses or stars or something. It will “auto-arrange” the destinations to help make the actions more visible:



Figure 164: Android Studio Navigation Editor, With Auto-Arrange Toolbar Button Highlighted

In the “Attributes” pane for this new action, set the ID to `createModel`. Then, there should be an “Argument Default Values” section, showing `modelId`. Fill in `@null` in the “default value” field, where `@null` means “no, I really mean `null`, and not the string “`null`””.

From the Android Studio main menu, choose “Build” > “Make Module ‘app’” to get Android Studio to generate fresh Safe Args code for our navigation resource.

Next, add this `add()` function to `RosterListFragment`:

```
private fun add() {  
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

ADDING AND DELETING ITEMS

This does the same sort of thing as `display()`, except that it uses the `createModel()` action instead of the `displayModel()` action.

Then, add this `onOptionsItemSelected()` function to `RosterListFragment`:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
    }

    return super.onOptionsItemSelected(item)
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

If the user clicks the add app bar item, we call the `add()` function.

If you run the app and click the new “Add” action bar item... you should crash. Specifically, you will get an error from Kotlin, complaining that something was null but was declared to be not-null.

That error is coming from this line in `SingleModelMotor`:

```
private val modelId: String
```

We declared this constructor parameter as taking `String`, and that used to work. But now we allow our `modelId` to be `null` to represent the case where we are creating a new model. To fix that, change that parameter to be `String?` instead of `String`:

```
class SingleModelMotor(
    private val repo: ToDoRepository,
    private val modelId: String?
) : ViewModel()
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#))

That, in turn, gives you an error in the `getModel()` function, as `find()` on `ToDoRepository` is set to take a `String` parameter, not `String?`. So, adjust `find()` on `ToDoRepository` to take `String?` instead:

```
fun find(modelId: String?) = items.find { it.id == modelId }
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

ADDING AND DELETING ITEMS

The implementation of `find()` does not need to change; it will simply return `null` when none of the items matches the `null` ID.

Now, if you run the app and click the “add” app bar item, you should get an empty `EditFragment` form, showing our hints for the description and notes fields:

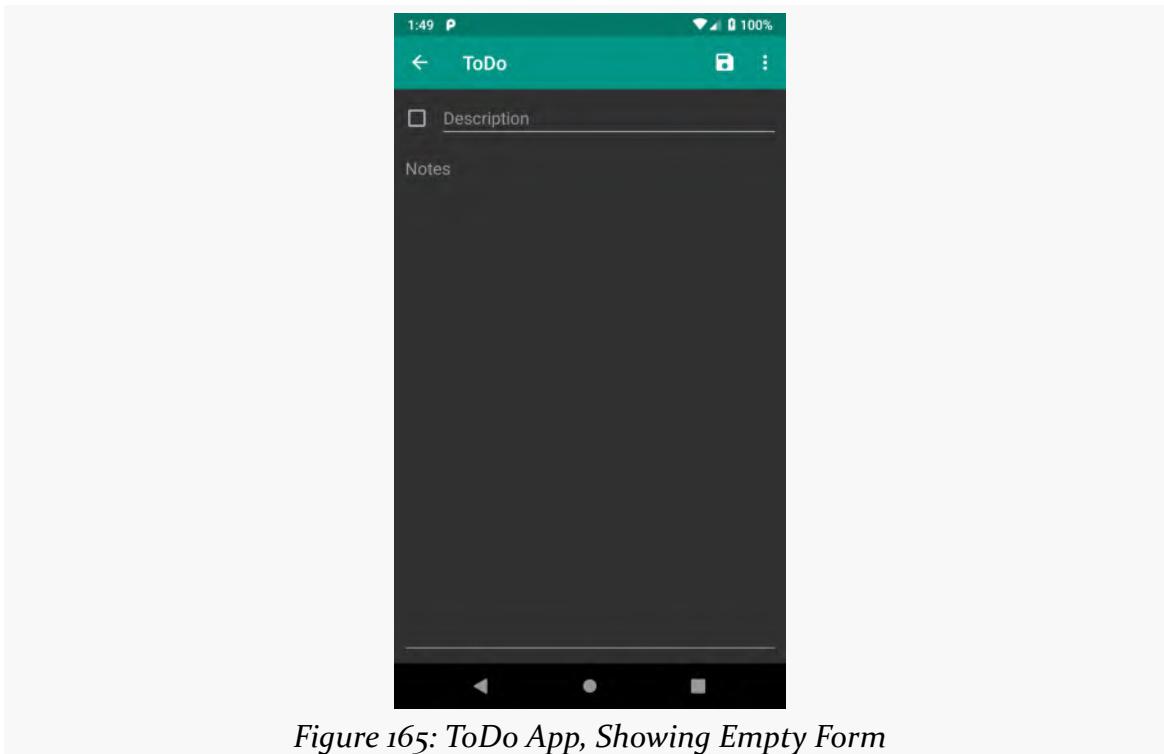


Figure 165: ToDo App, Showing Empty Form

ADDING AND DELETING ITEMS

If you run the sample app, click the “add” app bar item, fill in the form, and click the “save” app bar item, you wind up seeing the list of to-do items... with the empty text still visible:

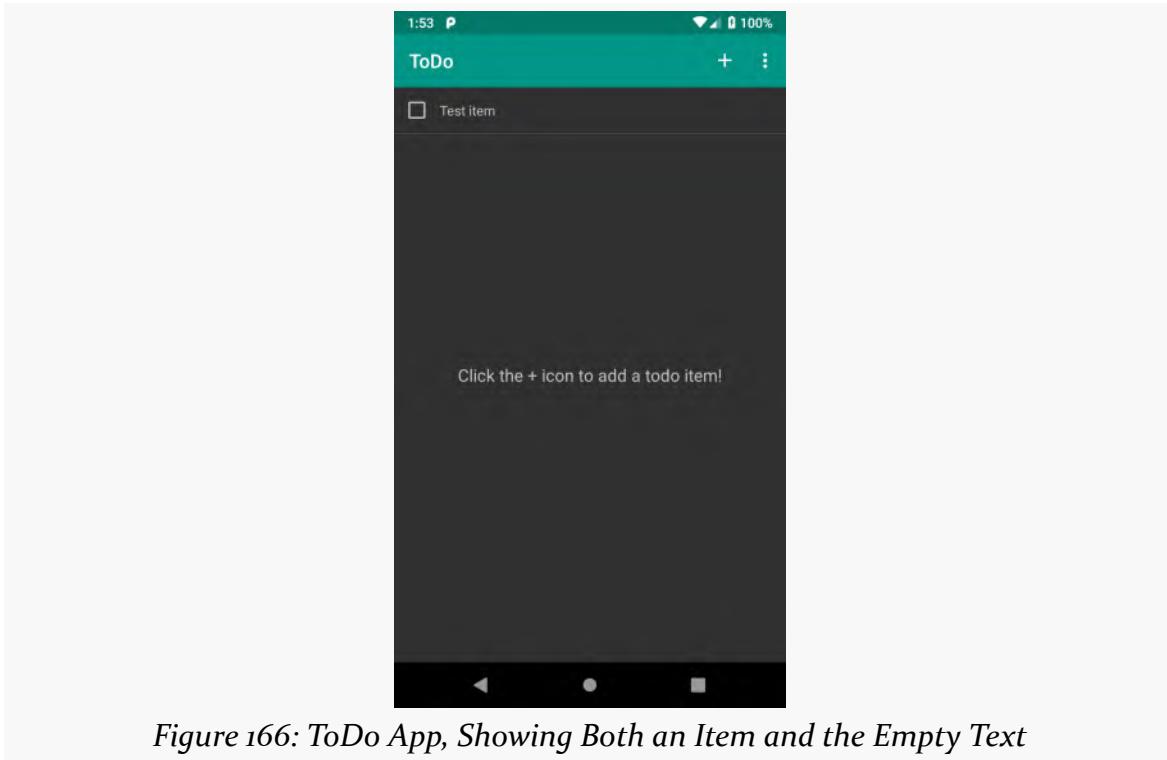


Figure 166: ToDo App, Showing Both an Item and the Empty Text

Step #5: Hiding the Empty View

Showing the empty view with just one to-do item is not so bad. The problem is that when we get enough to-do items, we wind up with overlapping text:

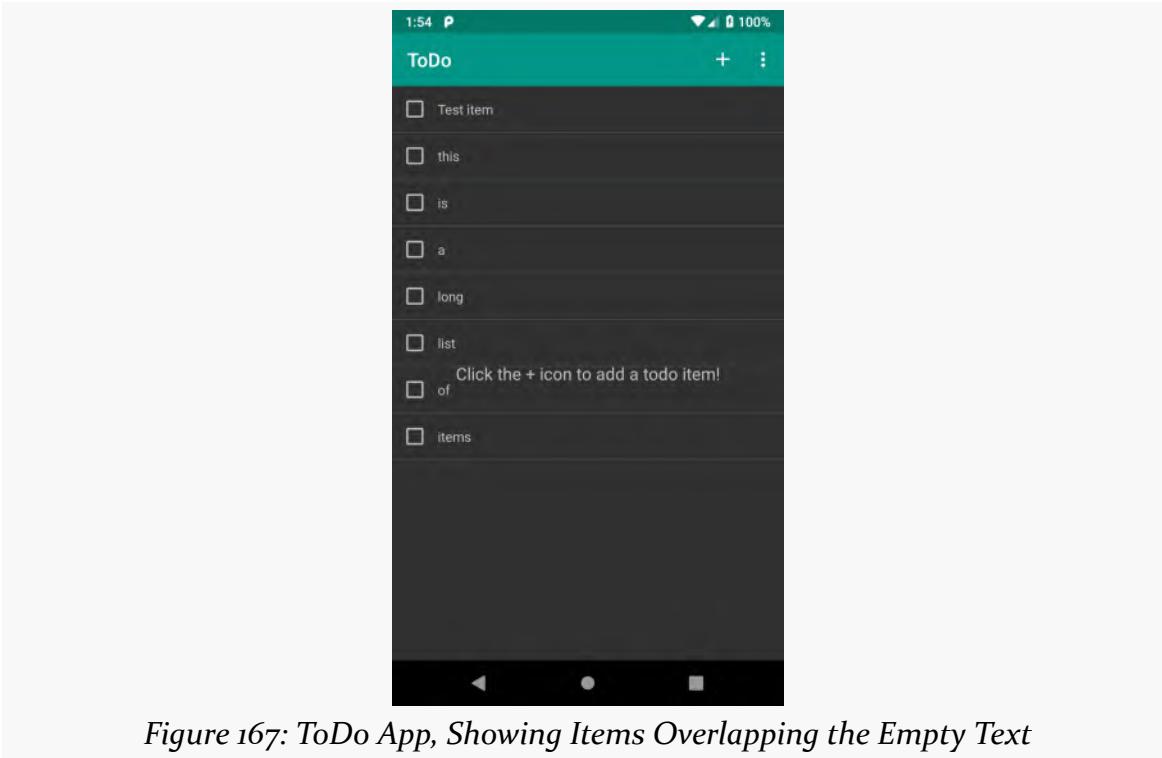


Figure 167: ToDo App, Showing Items Overlapping the Empty Text

Besides, the point of the empty view is to show it only when the list is empty.

To make that happen, add this line to the bottom of `onViewCreated()` on `RosterListFragment`:

```
binding?.empty?.visibility =  
    if (adapter.itemCount == 0) View.VISIBLE else View.GONE
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

So, we check our `RosterAdapter` to see if we have any items, and if we do, we set the empty view to be `GONE`.

Now, if you run the app, you will see the empty view at the outset — when we have no items — but the empty view will go away once you start adding items.

Step #6: Adding a Delete App Bar Item

We have one more app bar item to create, this one to allow the user to delete an item. We will add that to the app bar on `EditFragment`, so the user can delete the item from there.

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Clip Art” button and search for delete:

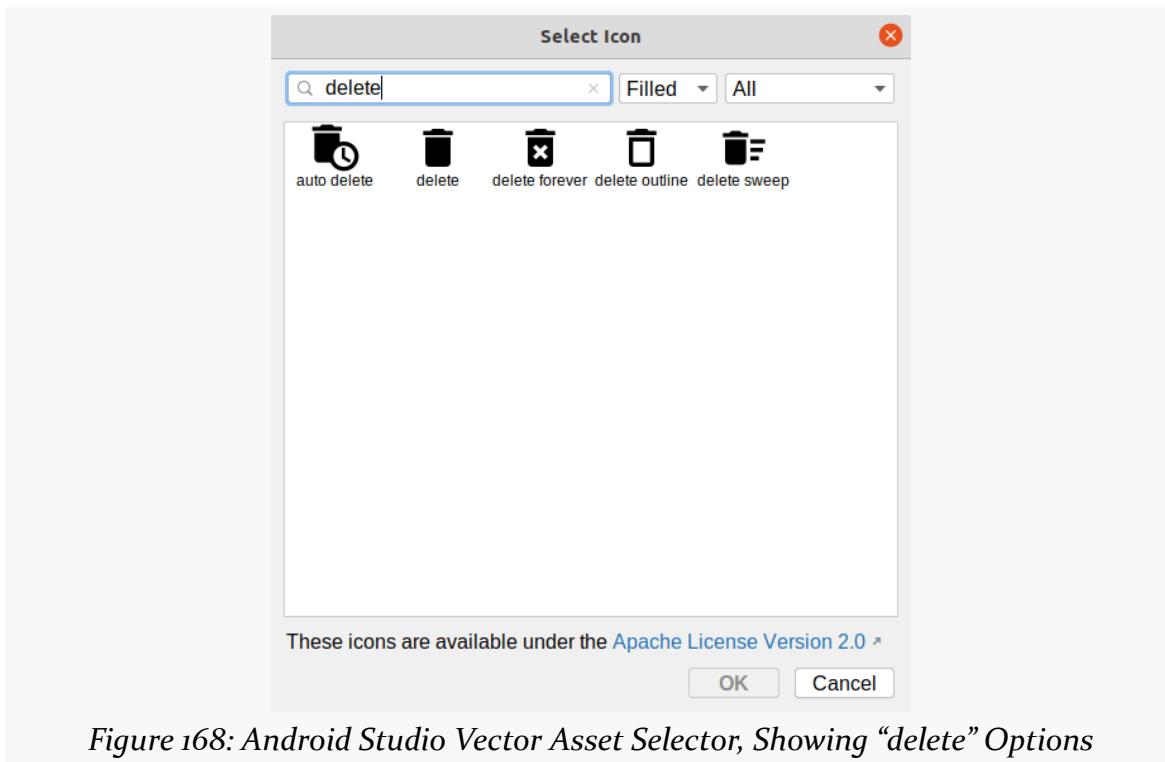


Figure 168: Android Studio Vector Asset Selector, Showing “delete” Options

Choose the “delete” icon and click “OK” to close up the icon selector. Change the name to `ic_delete`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

Once again, if the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead.

Open `res/menu/actions_edit.xml` in the IDE. In the graphical designer view, drag a

ADDING AND DELETING ITEMS

second “Menu Item” into the preview area.

In the Attributes pane, fill in `delete` for the “id”. Then, choose both “ifRoom” and “withText” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Click on `ic_delete` in the list of drawables, then click OK to accept that choice of icon. Then, click the “O” button next to the “title” field. As before, this brings up a string resource selector. Click on “+”, then click on “String Value” in the resulting drop-down. In the dialog, fill in `menu_delete` as the resource name and “Delete” as the resource value. Click OK to close the dialog, to complete the configuration of this app bar item.

Now, when you run the app and you go to add a new to-do item, or later you edit an existing to-do item, you will see the “delete” app bar item:

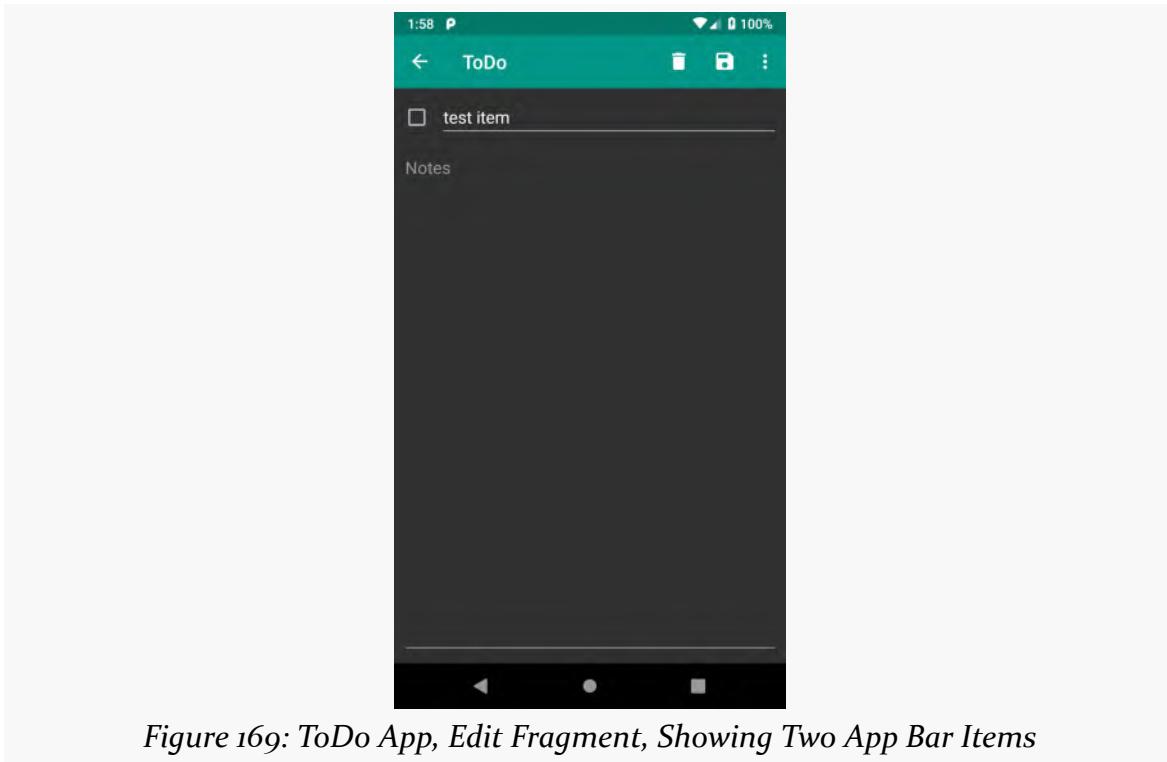


Figure 169: ToDo App, Edit Fragment, Showing Two App Bar Items

The fact that there is a delete icon for an add operation is... disturbing. We will address that later in this tutorial.

Step #7: Deleting the Item

Deleting the `ToDoModel` seems fairly straightforward: call `delete()` on the `ToDoRepository`, supplying the model to be deleted. To help, add this `delete()` function to `SingleModelMotor`, which forwards the call to the repository:

```
fun delete(model: ToDoModel) {  
    repo.delete(model)  
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#))

However, there is one wrinkle: we do not want to go back to the `DisplayFragment` after deleting the item, as there is nothing to display. Instead, we should head back to the `RosterListFragment`.

To that end, add this `navToList()` function to `EditFragment`:

```
private fun navToList() {  
    hideKeyboard()  
    findNavController().popBackStack(R.id.rosterListFragment, false)  
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

This hides the keyboard, then uses the `NavController` to pop the back stack. The default `popBackStack()` just pops one level off of the stack, akin to the user pressing BACK or the “up” arrow. In this case, we are telling the Navigation component:

- Pop all the way back to `rosterListFragment`...
- ...but do not remove `rosterListFragment` itself (the `false` value)

Then, add this `delete()` function to `EditFragment`:

```
private fun delete() {  
    val model = motor.getModel()  
  
    model?.let { motor.delete(it) }  
    navToList()  
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

This deletes the current model in the binding, plus calls the new `navToList()`

ADDING AND DELETING ITEMS

function.

Then, add another option to the when in `onOptionsItemSelected()` on `EditFragment` to call `delete()` if the user taps the “delete” app bar item:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.save -> {
            save()
            return true
        }
        R.id.delete -> {
            delete()
            return true
        }
    }

    return super.onOptionsItemSelected(item)
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

If you run the sample app, add a new item, go back in to edit it, and click the delete app bar item, that newly-added item is deleted, and you return to an empty list.

Step #8: Fixing the Delete-on-Add Problem

Right now, when you edit an existing to-do item, the “delete” app bar item appears. It *also* appears when you are adding a new to-do item. This is unnecessary and may confuse the user. Plus, it may not work all that well, since we cannot delete an item that has not been added.

Fortunately, fixing this requires just one line of code: updating `isVisible` on the `MenuItem` corresponding to “delete”.

Modify `onCreateOptionsMenu()` of `EditFragment` to look like:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_edit, menu)
    menu.findItem(R.id.delete).isVisible = args.modelId != null

    super.onCreateOptionsMenu(menu, inflater)
}
```

ADDING AND DELETING ITEMS

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

Here, we retrieve the delete MenuItem and call setVisibility() with true if we have a model, false otherwise. This has the desired effect: removing the “delete” item if we do not have anything to delete.

And, if you run the app and go to add a new item, the delete icon does not appear in the app bar, but it will appear if you try to edit an existing item.

Final Results

Our new actions_roster menu resource should contain:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/add"
        android:icon="@drawable/ic_add"
        android:title="@string/menu_add"
        app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T2o-Add/ToDo/app/src/main/res/menu/actions_roster.xml](#))

The nav_graph resource should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
             xmlns:app="http://schemas.android.com/apk/res-auto"
             android:id="@+id/nav_graph.xml"
             app:startDestination="@+id/rosterListFragment">

    <fragment
        android:id="@+id/rosterListFragment"
        android:name="com.commonsware.todo.RosterListFragment"
        android:label="@string/app_name">
        <action
            android:id="@+id/displayModel"
            app:destination="@+id/displayFragment" />
        <action
            android:id="@+id/createModel"
            app:destination="@+id/editFragment" >
            <argument
```

ADDING AND DELETING ITEMS

```
        android:name="modelId"
        android.defaultValue="@null" />
    </action>
</fragment>
<fragment
    android:id="@+id/displayFragment"
    android:name="com.commonware.todo.DisplayFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string" />
    <action
        android:id="@+id/editModel"
        app:destination="@+id/editFragment" />
</fragment>
<fragment
    android:id="@+id/editFragment"
    android:name="com.commonware.todo.EditFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string"
        app:nullable="true" />
</fragment>
</navigation>
```

(from [T2o-Add/ToDo/app/src/main/res/navigation/nav_graph.xml](#))

SingleModelMotor should look like:

```
package com.commonware.todo

import androidx.lifecycle.ViewModel

class SingleModelMotor(
    private val repo: ToDoRepository,
    private val modelId: String?
) : ViewModel() {
    fun getModel() = repo.find(modelId)

    fun save(model: ToDoModel) {
        repo.save(model)
    }

    fun delete(model: ToDoModel) {
        repo.delete(model)
    }
}
```

ADDING AND DELETING ITEMS

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#))

ToDoRepository should resemble:

```
package com.commonsware.todo

class ToDoRepository {
    var items = emptyList<ToDoModel>()

    fun save(model: ToDoModel) {
        items = if (items.any { it.id == model.id }) {
            items.map { if (it.id == model.id) model else it }
        } else {
            items + model
        }
    }

    fun find(modelId: String?) = items.find { it.id == modelId }

    fun delete(model: ToDoModel) {
        items = items.filter { it.id != model.id }
    }
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

RosterListFragment should look like:

```
package com.commonsware.todo

import android.os.Bundle
import android.view.*
import androidx.fragment.app.Fragment
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.databinding.TodoRosterBinding
import org.koin.androidx.viewmodel.ext.android.viewModel
import androidx.navigation.fragment.findNavController

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private var binding: TodoRosterBinding? = null

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

        setHasOptionsMenu(true)
    }
```

ADDING AND DELETING ITEMS

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View = TodoRosterBinding.inflate(inflater, container, false)
    .also { binding = it }
    .root

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

    val adapter = RosterAdapter(
        layoutInflater,
        onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
        onRowClick = ::display)

    binding?.items?.apply {
        setAdapter(adapter)
        layoutManager = LinearLayoutManager(context)

        addItemDecoration(
            DividerItemDecoration(
                activity,
                DividerItemDecoration.VERTICAL
            )
        )
    }
}

adapter.submitList(motor.getItems())
binding?.empty?.visibility =
    if (adapter.itemCount == 0) View.VISIBLE else View.GONE
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
```

ADDING AND DELETING ITEMS

```
R.id.add -> {
    add()
    return true
}

return super.onOptionsItemSelected(item)
}

private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

The actions_edit menu resource should now resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">

    <item
        android:id="@+id/save"
        android:icon="@drawable/ic_save"
        android:title="@string/menu_save"
        app:showAsAction="ifRoom|withText" />
    <item
        android:id="@+id/delete"
        android:icon="@drawable/ic_delete"
        android:title="@string/menu_delete"
        app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T2o-Add/ToDo/app/src/main/res/menu/actions_edit.xml](#))

SingleModelMotor should look like:

```
package com.commonsware.todo

import androidx.lifecycle.ViewModel

class SingleModelMotor(
```

ADDING AND DELETING ITEMS

```
private val repo: ToDoRepository,
private val modelId: String?
) : ViewModel() {
    fun getModel() = repo.find(modelId)

    fun save(model: ToDoModel) {
        repo.save(model)
    }

    fun delete(model: ToDoModel) {
        repo.delete(model)
    }
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#))

Finally, EditFragment overall should look like:

```
package com.commonsware.todo

import android.os.Bundle
import android.view.*
import android.view.inputmethod.InputMethodManager
import androidx.core.content.getSystemService
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.databinding.TodoEditBinding
import org.jetbrains.kotlinx.viewmodel.ext.android.viewModel
import org.koin.core.parameter.parametersOf

class EditFragment : Fragment() {
    private var binding: TodoEditBinding? = null
    private val args: EditFragmentArgs by navArgs()
    private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ) = TodoEditBinding.inflate(inflater, container, false)
        .apply { binding = this }
```

ADDING AND DELETING ITEMS

```
.root

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    motor.getModel()?.let {
        binding?.apply {
            isCompleted.isChecked = it.isCompleted
            desc.setText(it.description)
            notes.setText(it.notes)
        }
    }
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_edit, menu)
    menu.findItem(R.id.delete).isVisible = args.modelId != null

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.save -> {
            save()
            return true
        }
        R.id.delete -> {
            delete()
            return true
        }
    }
}

return super.onOptionsItemSelected(item)
}

private fun save() {
    binding?.apply {
        val model = motor.getModel()
        val edited = model?.copy(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        ) ?: ToDoModel(

```

ADDING AND DELETING ITEMS

```
        description = desc.text.toString(),
        isCompleted = isCompleted.isChecked,
        notes = notes.text.toString()
    )

    edited.let { motor.save(it) }
}

navToDisplay()
}

private fun delete() {
    val model = motor.getModel()

    model?.let { motor.delete(it) }
    navToList()
}

private fun navToDisplay() {
    hideKeyboard()
    findNavController().popBackStack()
}

private fun navToList() {
    hideKeyboard()
    findNavController().popBackStack(R.id.rosterListFragment, false)
}

private fun hideKeyboard() {
    view?.let {
        val imm = context?.getSystemService<InputMethodManager>()

        imm?.hideSoftInputFromWindow(
            it.windowToken,
            InputMethodManager.HIDE_NOT_ALWAYS
        )
    }
}
}
```

(from [T2o-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

ADDING AND DELETING ITEMS

- [app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)
- [app/src/main/res/drawable/ic_add.xml](#)
- [app/src/main/res/menu/actions_roster.xml](#)
- [app/src/main/res/drawable/ic_delete.xml](#)
- [app/src/main/res/menu/actions_edit.xml](#)
- [app/src/main/java/com/commonsware/todo/SingleModelMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/EditFragment.kt](#)

Interlude: So, What's Wrong?

We're done! Right?

Right?!?

.

.

.

Well, OK, we are not done.

There are some features that we could add, such as filtering the list of items based on whether they are completed or not, or being able to save the items out to a Web page.

However, beyond that, there are some fairly fundamental flaws in our app implementation, and some corresponding features of the Jetpack components that can help us address those.

Issues With What We Have

The app has few, if any, outright bugs. However, it does have some shortcomings that affect users, ourselves, and (theoretical) future maintainers of the code.

Lack of Persistence

The biggest gap is that our to-do items are not stored anywhere other than memory.

INTERLUDE: SO, WHAT'S WRONG?

As soon as our process is terminated, the to-do items will go away. And our app's process may not live that long in the background. So, after a period of time, it is all but guaranteed that the user will have no more to-do items.

Admittedly, some users will consider that to be a feature, not a bug.

However, in general, people put items in a to-do list in order to keep track of what needs to be done, and for that, we need to save the items somewhere.

Synchronous Work

Adding persistence will cause another problem: all of our interactions with the `ToDoRepository` are synchronous. That means that our I/O will tie up the main application thread, freezing our UI while that I/O is being done.

This is not good.

Instead, we need to switch to an asynchronous interaction with the repository. For example, when we `save()` a `ToDoModel`, it should not be a blocking call. Instead, the actual work for saving the data should happen on a background thread, with us finding out about the result asynchronously. That way, the UI remains responsive while we are doing the I/O.

We Can Do Better

The next several tutorials will be focused on addressing these concerns, using solutions from the Jetpack components and from popular approaches in the Android development ecosystem.

Persistence: Room

In theory, we could save our to-do items to “the cloud”, persisting them on a server somewhere. However, that is complicated, in ways that go far beyond complicated Android code. It would require you to set up a server, or sign up for some service, to have something in the cloud for the app to talk to.

Besides, this is just a list of to-do items. Not everything needs a server.

So, we will keep the to-do items locally on the device. Specifically, we will use Room, the Jetpack solution for local databases. Room is a wrapper around Android's built-

INTERLUDE: SO, WHAT'S WRONG?

in SQLite database. We can create classes that represent databases, tables, and operations (e.g., queries, inserts), and our `ToDoRepository` can use those to store the to-do items.

Asynchronous Work: Coroutines

There are a wide range of solutions for doing work asynchronously in Android. In these tutorials, we will use two, the first being Kotlin coroutines.

Coroutines are a recent addition to Kotlin that make it easy to designate some work to be done on background threads, while still making it easy to write code that consumes the results of that work on the main application thread.

Phase Two: Asynchronous Architecture

Refactoring Our Code

Right now, all of our code resides in a single package: `com.commonsware.todo`. For tiny apps, that is reasonable. The more complex your app gets, the more likely it is that you will want to organize the classes into sub-packages. We will be adding many more classes to the app, so now seems like a good time to refactor the code and set up some sub-packages.

Fortunately, Android Studio makes this very easy!

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Creating Some Packages

First, we should set up the packages into which we want to reorganize the classes.

Right-click over the `com.commonsware.todo` package and choose “New” > “Package” from the context menu. In the field, fill in `repo` and press `Enter` or `Return` to create the package.

Do that again, but this time fill in `ui.display`. This creates a package (`ui`) and a sub-package (`display`) in one shot.

Do that again, but this time fill in `ui.edit`.

Now, right-click over the `com.commonsware.todo.ui` package and choose “New” > “Package” from the context menu. Enter `roster` in the field and click OK. Since we started from the `com.commonsware.todo.ui` package, `roster` becomes a sub-package

REFACTORING OUR CODE

of that, a peer of the display and edit ones.

At this point, your project tree should resemble:

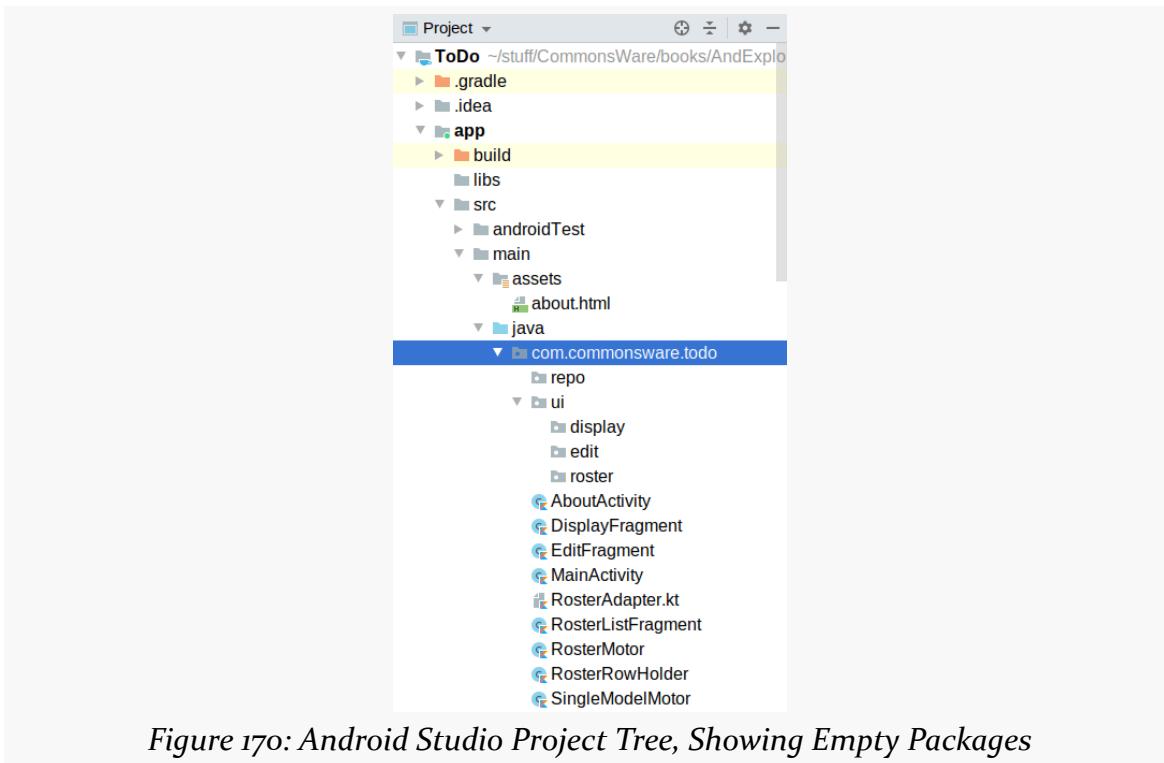


Figure 170: Android Studio Project Tree, Showing Empty Packages

Step #2: Moving Our Classes

From here, it is a matter of dragging our classes from where they are to the desired package. Android Studio will take care of fixing up any import statements, view binding references, the manifest, and related stuff.

REFACTORING OUR CODE

Start by dragging `ToDoRepository` to the `repo` package. When you drop the class in the `repo` package, a “Move” dialog will appear:

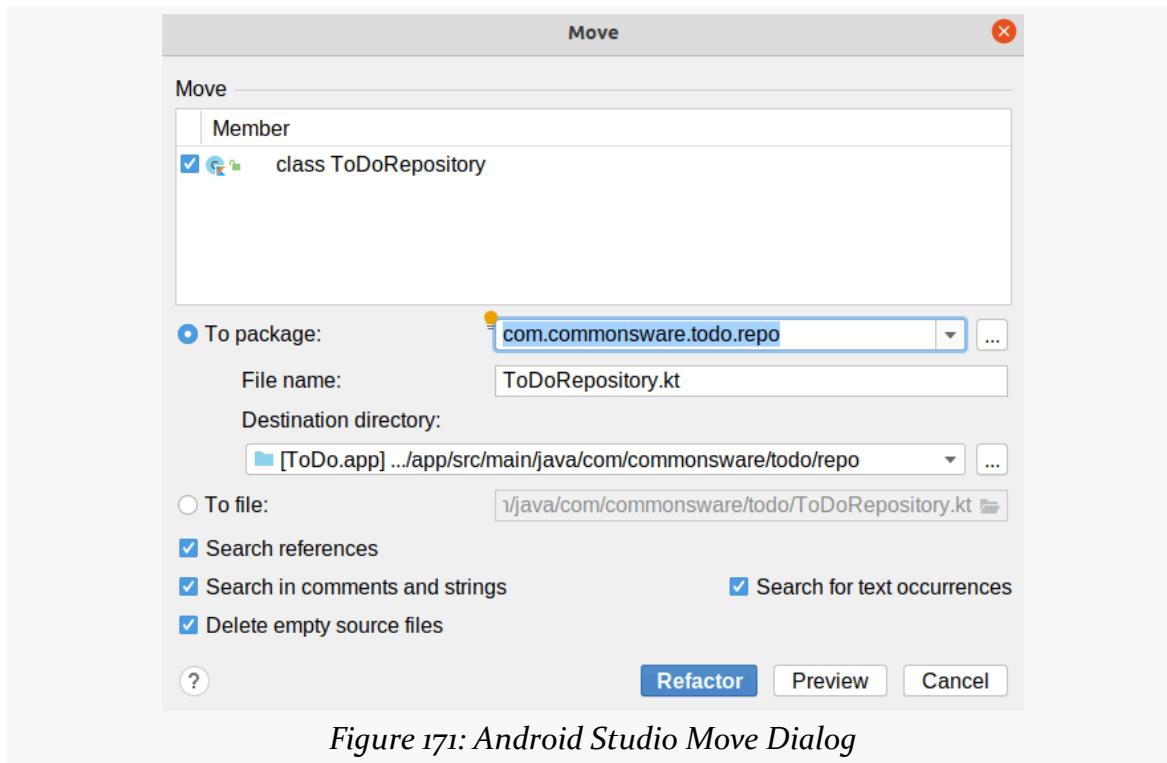


Figure 171: Android Studio Move Dialog

This confirms that we are going to move this class to the designated package. The checkboxes towards the bottom of the dialog indicate where Android Studio will look for this class, in order to change that code to point to the new package (if needed).

Click “Refactor”, and in a moment, your class will now be in the `repo` package.

Then, move most of the remaining classes to the new packages:

REFACTORING OUR CODE

Class(es)	Package
ToDoModel	repo
AboutActivity, MainActivity, SingleModelMotor	ui
RosterAdapter, RosterListFragment, RosterMotor, RosterRowHolder	roster
DisplayFragment	display
EditFragment	edit

In the end, only ToDoApp should be directly in `com.commonsware.todo`, with everything else in one of the sub-packages off of `com.commonsware.todo`.

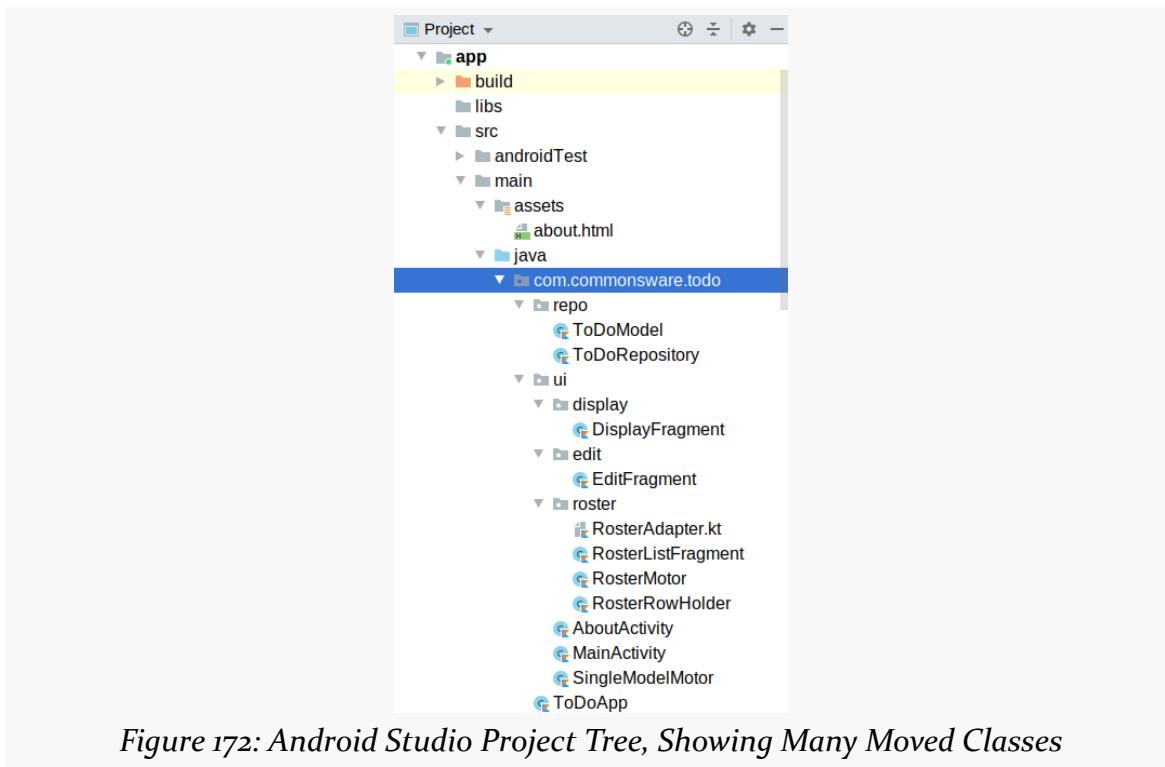


Figure 172: Android Studio Project Tree, Showing Many Moved Classes

You may find Android Studio showing a bunch of red errors in the editors for your fragments. If that happens, choose “File” > “Sync Project with Gradle Files” from the main menu, and that may clear up the problem.

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#).

Many files were changed, both directly and indirectly, as a result of the steps in this tutorial, including all but one of the Kotlin source files.

Getting a Room (And Some Coroutines)

So far, we have been content to have our to-do items vanish when we re-run our app. This was simple and easy to write. However, it is not realistic. Users will expect their to-do items to remain until deleted. To do that, we need our items to survive process termination, and that requires that we save those items somewhere, such as on disk.

In this tutorial, we will start in on that work, setting up database support using Room, a Google-supplied framework that layers atop Android's native SQLite support. SQLite is a relational database. Through Room, we will create a database containing a table for our to-do items.



You can learn more about Room in the "Storing Data in a Room" chapter of [*Elements of Android Jetpack!*](#)

In truth, this app is trivial enough that you could use something simpler for storage, such as storing the items in a JSON file. The bigger the app, the more likely it is that SQLite and Room will be better options for you.

However, even trivial database I/O takes some time, so we want to move that work to background threads. To do that, we will use Kotlin coroutines. Coroutines are a relatively new addition to Kotlin. They try to make it easy for you to write code that looks like it is happening all on one thread, statement after statement, when in reality multiple threads are involved.

GETTING A ROOM (AND SOME COROUTINES)



You can learn more about Kotlin coroutines in the "Introducing Coroutines" chapter of [Elements of Kotlin Coroutines](#)!

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Requesting More Dependencies

Room has its own set of dependencies that we need to add to the dependencies closure in `app/build.gradle`.

Room has its own series of versions, independent of anything else that we have used. So, let's define another version constant in our top-level `build.gradle` file. Add this line to the ext closure:

```
room_version = "2.3.0"
```

(from [T22-Room/ToDo/build.gradle](#))

Then, in `app/build.gradle`, add three new dependencies that reference that version constant... and one other dependency:

```
implementation "androidx.room:room-runtime:$room_version"
implementation "androidx.room:room-ktx:$room_version"
kapt "androidx.room:room-compiler:$room_version"
```

(from [T22-Room/ToDo/app/build.gradle](#))

Room is based heavily on the use of Java annotations, and the `androidx.room:room-compiler` dependency will handle those annotations for us at compile time. The `androidx.room:room-runtime` dependency is for core Room functionality, while the `androidx.room:room-ktx` dependency adds support for Room doing database I/O using coroutines (along with a few other Kotlin extension functions).

The `kapt` directive that we are using for `room-compiler` says that this dependency contains an annotation processor. That, in turn, requires a new plugin. So, add this line to the other `apply plugin` statements towards the top of `app/build.gradle`:

```
id 'kotlin-kapt'
```

GETTING A ROOM (AND SOME COROUTINES)

(from [T22-Room/ToDo/app/build.gradle](#))

After adding these lines, go ahead and allow Android Studio to sync the project with the Gradle build files.

Step #2: Defining an Entity

In Room, an entity is a class that is our in-memory representation of a SQLite table. Instances of the entity class represent rows in that table.

So, we need an entity to create a SQLite table for our to-do items.

Which means... we need another Kotlin class!

Right-click over the `com.commonware.todo.repo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `ToDoEntity` and choose “Class” as the kind. Press `Enter` or `Return` to create the class, giving you:

```
package com.commonware.todo.repo

class ToDoEntity { }
```

Then, replace that stub implementation with this:

```
package com.commonware.todo.repo

import androidx.room.Entity
import androidx.room.Index
import androidx.room.PrimaryKey
import java.time.Instant
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Instant = Instant.now(),
    val isCompleted: Boolean = false
)
```

GETTING A ROOM (AND SOME COROUTINES)

This class has the same properties as `ToDoModel`. You might wonder why we did not just use `ToDoModel`. Mostly, that is for realism: there is no guarantee that your entities will have a 1:1 relationship with models. Room puts restrictions on how entities can be constructed, particularly when it comes to relationships with other entities. Things that you might do in model objects (e.g., a category object holding a collection of item objects) wind up having to be implemented significantly differently using Room entities. Those details will get hidden by your repositories. A repository exists in part to convert specialized forms of your data (Room entities, Web service responses, etc.) into the model objects that your UI is set up to use.

What makes `ToDoEntity` an entity is the `@Entity` annotation at the top. There, we can provide metadata about the table that we want to have created. Here, we specify that we want the underlying table name to be `todos`, as opposed to the default, which is the same as the class name (`ToDoEntity`).

Room knows that the `id` property is our primary key because we gave it the `@PrimaryKey` annotation. Room wants us to declare some primary key, typically via that `@PrimaryKey` annotation. We also have an index on our `id` column, courtesy of the `@Index` nested annotation inside of the `@Entity` annotation.

Step #3: Crafting a DAO

The `@Entity` class says “this is what my table should look like”. A `@Dao` class says “this is how I want to read and write from that table”. With Room, we define an interface or abstract class to describe the API that we want to have for working with the database. Room then code-generates an implementation for us, dealing with all of the SQLite code for getting our entities to and from our table.

Inside the `ToDoEntity` class (i.e., inside a {} that you add after the constructor), add this nested interface:

```
@Dao
interface Store {
    @Query("SELECT * FROM todos ORDER BY description")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String?): Flow<ToDoEntity?>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(vararg entities: ToDoEntity)
```

GETTING A ROOM (AND SOME COROUTINES)

```
@Delete
suspend fun delete(vararg entities: ToDoEntity)
}
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

The `@Dao` annotation tells Room that this interface serves as a DAO and defines an API that we want to use. On it, we have four functions. Each has an annotation indicating what is the database operation that this method should apply:

- `@Insert` for inserts
- `@Update` for updates
- `@Delete` for deletions
- `@Query` for anything, but mostly used for data retrieval

The `@Query` annotations always take a SQL statement as an annotation property, to indicate what SQL should be executed when this function is called. That SQL statement sometimes stands alone, as does `SELECT * FROM todos` for the `all()` function. However, the SQL can reference function parameters, such as in the case of the `find()` function. It has a `modelId` parameter, and our SQL statement refers to that, using a `:` prefix to identify that it is a reference to a parameter name (`SELECT * FROM todos WHERE id = :modelId`).

Query functions based on `SELECT` statements return whatever it is that the query is supposed to return. In our case, we are querying all columns from the `todos` table, and we are asking Room to map those rows to instances of our `ToDoEntity` class. For the `all()` function, we are expecting that there may be more than one, so the return type is based on a `List` of entities. By contrast, `find()` expects at most one result, so the return type is based on a single `ToDoEntity`.

We could have written `all()` and `find()` like this:

```
@Query("SELECT * FROM todos")
fun all(): List<ToDoEntity>

@Query("SELECT * FROM todos WHERE id = :modelId")
fun find(modelId: String): ToDoEntity
```

In that case, those functions would be synchronous, blocking until the query is complete.

Instead, our functions wrap our desired return values in `Flow`, from Kotlin's coroutines system. This has two key effects:

GETTING A ROOM (AND SOME COROUTINES)

1. Room will perform the queries on a background thread and post the results to the Flow when the results are ready. Hence, our functions are asynchronous, returning immediately, rather than blocking waiting for the database I/O to complete.
2. So long as we have 1+ observers of the Flow, if we do other database operations that affect the todos table, Room will automatically deliver a new result to those observers via the Flow. So, if we insert or delete a row from our table, observers will get updated data, which (if appropriate) will reflect those data changes.

Our other two functions — `save()` and `delete()` — use other Room annotations. `save()` uses `@Insert`, while `delete()` uses `@Delete`.

We are using `save()` for both inserts and updates. The `onConflict = OnConflictStrategy.REPLACE` property in our `@Insert` annotation says “if there already is a row with this primary key in the database, replace it with new contents”. So, if we pass in a brand-new `ToDoEntity`, it will be inserted, but if we pass in a `ToDoEntity` that reflects a change to an existing row, that row will be updated.

Note that both `save()` and `delete()` use `vararg`. This allows us to pass as many entities as we want, with all of them being saved or deleted. This is not required — you can have `@Insert` or `@Delete` functions that accept a single entity, a `List` of entities, etc.

And, note that both `save()` and `delete()` are suspend functions. As with Flow, `suspend` comes from Kotlin coroutines. Room will have `save()` and `delete()` perform their I/O on background threads, but from a programming standpoint, it will feel like we are making synchronous calls on the current thread.

Step #4: Adding a Database

The third major piece of any Room usage is a `@Database`. Here, we not only need to add the annotation to a class, but we need to have that class inherit from Room’s own `RoomDatabase` base class.

Which means... we need another Kotlin class! Again!

Right-click over the `com.commonsware.todo.repo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `ToDoDatabase`, and choose “Class” as the kind. Then, Press `Enter` or `Return` to

GETTING A ROOM (AND SOME COROUTINES)

create the class, giving you:

```
package com.commonware.todo.repo

class ToDoDatabase {
```

Then, replace that implementation with:

```
package com.commonware.todo.repo

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

private const val DB_NAME = "stuff.db"

@Database(entities = [ToDoEntity::class], version = 1)
abstract class ToDoDatabase : RoomDatabase() {
    abstract fun todoStore(): ToDoEntity.Store

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME).build()
    }
}
```

The `@Database` annotation is where we provide metadata about the database that we want Room to manage for us. Specifically:

- We tell it which classes have `@Entity` annotations and should have their tables in this database
- What is the version code of this database schema — usually, we start at 1, and we increment from there, any time that we add tables, columns, indexes, and so on

The `todoStore()` method returns an instance of our `@Dao`-annotated interface. This, coupled with the `@Database` annotation, tells Room's annotation processor to code-generate an implementation of our abstract `ToDoDatabase` class that has an implementation of `todoStore()` that returns a code-generated implementation of `ToDoEntity.Store`.

To create the `ToDoDatabase` instance, in our `newInstance()` factory function, we use

GETTING A ROOM (AND SOME COROUTINES)

`Room.databaseBuilder()`, passing it three values:

- a Context to use — and since this is a singleton, we need to use the Application to avoid any memory leaks
- the class representing the RoomDatabase to create
- a String with the filename to use for the database

The resulting `RoomDatabase.Builder` could be further configured, but we do not need that here, so we just have it `build()` the database and return it.

`ToDoDatabase` is marked as abstract — the actual class that is used by `RoomDatabase.Builder` will be a subclass created by Room's annotation processor.

Step #5: Creating a Transmogrifier

If you try building the project — for example, Build > “Make module ‘app’” from the Android Studio main menu — you will get build errors, such as:

```
error: Cannot figure out how to save this field into database. You can  
consider adding a type converter for it.  
    private final java.time.Instant createdOn = null;
```

The problem is that Room does not know what to do with an `Instant` object. SQLite does not have a native date/time column type, and Room cannot convert arbitrary objects into arbitrary SQLite column types. Instead, Room's annotation processor detects the issue and fails the build.

To fix this, we need to teach Room how to convert `Instant` objects to and from some standard SQLite column type. And for that... we could really use another Kotlin class. Fortunately, you can never have too many Kotlin classes!

(NARRATOR: you definitely can have too many Kotlin classes, but one more will not hurt)

Right-click over the `com.commonware.todo.repo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `TypeTransmogrifier`. But, this time, choose “Object” for the kind. Press `Enter` or `Return` to create the class, giving you:

GETTING A ROOM (AND SOME COROUTINES)

```
package com.commonsware.todo.repo

object TypeTransmogrifier {
```

A [transmogrifier](#) is a ~30-year-old piece of advanced technology that can convert one thing into another. Here, we are creating a type transmogrifier: a set of functions that turn one type into another.

To that end, replace the stub generated class with this:

```
package com.commonsware.todo.repo

import androidx.room.TypeConverter
import java.time.Instant

object TypeTransmogrifier {
    @TypeConverter
    fun fromInstant(date: Instant?): Long? = date?.toEpochMilli()

    @TypeConverter
    fun toInstant(millisSinceEpoch: Long?): Instant? = millisSinceEpoch?.let {
        Instant.ofEpochMilli(it)
    }
}
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/TypeTransmogrifier.kt](#))

The `@TypeConverter` annotations tell Room that this is a function that can convert one type into another. Here, we convert `Instant` objects into `Long` objects, using the time-since-the-Unix-epoch methods on `Instant`.

Then, add this annotation to the `ToDoDatabase` class declaration, under the existing `@Database` annotation:

```
@TypeConverters(TypeTransmogrifier::class)
```

This tells Room that for any entities used by this `ToDoDatabase`, if you need to convert a type, try looking for `@TypeConverter` methods on `TypeTransmogrifier`.

Now, if you choose `Build > “Make module ‘app’”` from the Android Studio main menu, the app should build successfully.

Step #6: Add Our Database to Koin

Usually, a Room database is a singleton. And, since we are using Koin, we can have Koin supply our database to other classes via dependency injection.

In ToDoApp, add this line to the koinModule declaration:

```
single { ToDoDatabase.newInstance(androidContext()) }
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This simply invokes our newInstance() factory function and exposes that instance as a single object. It uses an androidContext() function, supplied by Koin, to get the Application singleton and supply that as a Context to our newInstance() factory function.

However, to enable androidContext() to work in our single() call, we need to teach Koin about our ToDoApp object. To that end, modify onCreate() in ToDoApp to look like this:

```
override fun onCreate() {
    super.onCreate()

    startKoin {
        androidLogger()
        androidContext(this@ToDoApp)
        modules(koinModule)
    }
}
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Now, our startKoin() call contains an androidContext() setter call, where we provide the Context to use for our androidContext() call up in koinModule.

Technically, we could bypass all of this and have our single() in koinModule use this instead of androidContext(). The downside of that approach is that if we wanted a *different* Context in testing, we would be unable to provide it. Basically, Koin allows us to inject the top-level Context in addition to injecting our own classes.

Step #7: Adding a Store to the Repository

Next, we need to have our `ToDoRepository` get access to a `ToDoEntity.Store`, so it can manipulate the database instead of an in-memory transient copy of data.

Update the `ToDoRepository` to add a pair of constructor parameters:

```
class ToDoRepository(  
    private val store: ToDoEntity.Store,  
    private val appScope: CoroutineScope  
) {
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

The first parameter is our DAO, `ToDoEntity.Store`. We will use that to work with the database. The second parameter is a `CoroutineScope`. Take it on faith for the moment that we need that parameter — we will apply it in the next section and see more about why we need it in the next tutorial.

Next, in `ToDoApp`, add this `single` to our `koinModule`:

```
single(named("appScope")) { CoroutineScope(SupervisorJob()) }
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This sets up a singleton instance of a `CoroutineScope`, wrapped around a `SupervisorJob`. That will be important when we tie in our viewmodels to our updated repository, as we will see in the next tutorial.

The `named("appScope")` parameter to the `single()` call tells Koin that there might be more than one `CoroutineScope` in our module, and we only want to use *this* `CoroutineScope` if somebody asks for it by name. In reality, we will only have this one `CoroutineScope`, but using named components like this is good practice for a general-purpose object like `CoroutineScope`.

Then, in `ToDoApp`, change the `ToDoRepository` line in `koinModule` to be:

```
single {  
    ToDoRepository(  
        get<ToDoDatabase>().todoStore(),  
        get(named("appScope"))  
    )  
}
```

GETTING A ROOM (AND SOME COROUTINES)

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

We use `get()` twice to find our dependencies and inject them. However, the `get()` calls are a bit different than the ones we have used previously:

- The first one uses generics to indicate that we want to fetch a `ToDoDatabase`. Normally, `get()` works on the type of the parameter, but that is a `ToDoEntity.Store`. Koin does not know how to get such a thing, so we tell it to `get()` the `ToDoDatabase`, and from there we call `todoStore()` ourselves to get the `ToDoEntity.Store` that `ToDoRepository` needs.
- The second one uses a similar named(`"appScope"`) parameter to the one we used in the `CoroutineScope` single declaration. So, we are asking to `get()` the object of the desired type (`CoroutineScope`, based on the parameter type) that is named `appScope`.

Step #8: Fixing the Repository

Now, we need to have the `ToDoRepository` really use the `ToDoEntity.Store`, rather than just get it in a constructor parameter.

However, we have problems. `ToDoRepository` works with models. `ToDoEntity.Store` works with entities. We are going to need to be able to convert between these two types.

To that end, add this constructor and function to `ToDoEntity`:

```
constructor(model: ToDoModel) : this(  
    id = model.id,  
    description = model.description,  
    isCompleted = model.isCompleted,  
    notes = model.notes,  
    createdOn = model.createdOn  
)  
  
fun toModel(): ToDoModel {  
    return ToDoModel(  
        id = id,  
        description = description,  
        isCompleted = isCompleted,  
        notes = notes,  
        createdOn = createdOn  
    )  
}
```

GETTING A ROOM (AND SOME COROUTINES)

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

These offer bi-directional conversion between a `ToDoModel` and a `ToDoEntity`. If we needed more data conversion between things that Room knows how to store and how we wanted to represent them in the models, we could have that logic here as well.

Then, replace the contents of `ToDoRepository` with the following:

```
package com.commonsware.todo.repo

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.withContext

class ToDoRepository(
    private val store: ToDoEntity.Store,
    private val appScope: CoroutineScope
) {
    fun items(): Flow<List<ToDoModel>> =
        store.all().map { all -> all.map { it.toModel() } }

    fun find(id: String?): Flow<ToDoModel?> = store.find(id).map { it?.toModel() }

    suspend fun save(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.save(ToDoEntity(model))
        }
    }

    suspend fun delete(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.delete(ToDoEntity(model))
        }
    }
}
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

`items()` now calls `all()` on our `ToDoEntity.Store`, to retrieve all of the entities. We use the `map()` operator on `Flow` to convert the `List` of `ToDoEntity` into a corresponding `List` of `ToDoModel`. So, `items()` now returns a `Flow` for that list of models. Every time `ToDoEntity.Store` emits a new list of entities, our repository emits the corresponding list of models.

Similarly, `find()` now calls `find()` on the `ToDoEntity.Store` and uses `map()` to convert the entity into a model.

We also delegate our `save()` and `delete()` calls to their corresponding ones on `ToDoEntity.Store`. We use the constructor that we added to `ToDoEntity` to map

GETTING A ROOM (AND SOME COROUTINES)

from our models to our entities. And, we wrap the actual DAO calls in `withContext()`, using a `CoroutineContext` obtained from our `CoroutineScope`. This says “use this context (and job) to manage the work in this coroutine”. Since the scope and context are tied to that `SupervisorJob`, that job manages the work, rather than any job that was set up by callers of these suspend functions. We will see how this comes into play when we update our viewmodels, in the next tutorial.

So, right now, our app is broken. `SingleModelMotor` and `RosterMotor` are expecting the old, synchronous repository API, instead of this new coroutines-based one. We will fix that, and get our app working once again, in the next tutorial.

Final Results

Your top-level `build.gradle` file should look like:

```
buildscript {
    ext.nav_version = '2.3.5'

    repositories {
        google()
        mavenCentral()
    }

    dependencies {
        classpath 'com.android.tools.build:gradle:7.0.2'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}

ext {
    koin_version = "3.1.2"
    room_version = "2.3.0"
}
```

(from [T22-Room/ToDo/build.gradle](#))

And your app module's `build.gradle` file should resemble:

```
plugins {
    id 'com.android.application'
```

GETTING A ROOM (AND SOME COROUTINES)

```
id 'kotlin-android'
id 'androidx.navigation.safeargs.kotlin'
id 'kotlin-kapt'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

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

    buildFeatures {
        viewBinding true
    }

    compileOptions {
        coreLibraryDesugaringEnabled true
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }

    kotlinOptions {
        jvmTarget = '1.8'
    }
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
```

GETTING A ROOM (AND SOME COROUTINES)

```
implementation 'com.google.android.material:material:1.4.0'
implementation "io.insert-koin:koin-android:$koin_version"
implementation "androidx.room:room-runtime:$room_version"
implementation "androidx.room:room-ktx:$room_version"
kapt "androidx.room:room-compiler:$room_version"
coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
testImplementation 'junit:junit:4.13.2'
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

(from [T22-Room/ToDo/app/build.gradle](#))

ToDoEntity should now look like:

```
package com.commonsware.todo.repo

import androidx.room.*
import kotlinx.coroutines.flow.Flow
import java.time.Instant
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Instant = Instant.now(),
    val isCompleted: Boolean = false
) {
    constructor(model: ToDoModel) : this(
        id = model.id,
        description = model.description,
        isCompleted = model.isCompleted,
        notes = model.notes,
        createdOn = model.createdOn
    )

    fun toModel(): ToDoModel {
        return ToDoModel(
            id = id,
            description = description,
            isCompleted = isCompleted,
            notes = notes,
            createdOn = createdOn
        )
    }
}
```

GETTING A ROOM (AND SOME COROUTINES)

```
@Dao
interface Store {
    @Query("SELECT * FROM todos ORDER BY description")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String?): Flow<ToDoEntity?>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(vararg entities: ToDoEntity)

    @Delete
    suspend fun delete(vararg entities: ToDoEntity)
}
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

ToDoDatabase should look like:

```
package com.commonsware.todo.repo

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters

private const val DB_NAME = "stuff.db"

@Database(entities = [ToDoEntity::class], version = 1)
@TypeConverters(TypeTransmogrifier::class)
abstract class ToDoDatabase : RoomDatabase() {
    abstract fun todoStore(): ToDoEntity.Store

    companion object {
        fun newInstance(context: Context) =
            Room.databaseBuilder(context, ToDoDatabase::class.java, DB_NAME).build()
    }
}
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#))

And ToDoApp now should resemble:

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

GETTING A ROOM (AND SOME COROUTINES)

```
import android.app.Application
import com.commonsware.todo.repo.ToDoDatabase
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.ui.SingleModelMotor
import com.commonsware.todo.ui.roster.RosterMotor
import kotlinx.coroutinesCoroutineScope
import kotlinx.coroutinesSupervisorJob
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.core.qualifier.named
import org.koin.dsl.module

class ToDoApp : Application() {
    private val koinModule = module {
        single(named("appScope")) { CoroutineScope(SupervisorJob()) }
        single { ToDoDatabase.newInstance(androidContext()) }
        single {
            ToDoRepository(
                get<ToDoDatabase>().todoStore(),
                get(named("appScope"))
            )
        }
        viewModel { RosterMotor(get()) }
        viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@ToDoApp)
            modules(koinModule)
        }
    }
}
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

GETTING A ROOM (AND SOME COROUTINES)

- [build.gradle](#)
- [app/build.gradle](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/TypeTransmogrifier.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#)

Completing the Reactive Architecture

We have our database, and our repository now offers a reactive and asynchronous API for working with the data. However:

- Our viewmodels do not know how to work with this new repository API, and
- Our fragments will need to be adjusted to adapt to whatever changes we make to the viewmodels to address the preceding bullet
- Our data model is getting more complex, as we no longer can just worry about a handful of to-do items

For example, right now, `RosterListFragment` only needs the list of to-do items. However, that is very simplistic. Most UIs are more complex than that. Even in the scope of this book, this is too simple. Later on, we will add filtering into the app, so the user can restrict the output to only show a subset of the items. Similarly, we could add searches, so the user could find items that match some search expression. Now, we need to keep track of filter modes, search expressions, and so on, in addition to the items to be displayed. And, as we move into asynchronous operations, we will want to track whether or not we are working on loading the data, so we can show some sort of progress indicator while that is going on. And so forth.

To help deal with that complexity, rather than having our motors keep track just of items, we will have them emit “view-state” objects. The view-state represents the data needed to render the UI. The fragments will observe those view-states and update their UIs to match, based on what is in those states.

COMPLETING THE REACTIVE ARCHITECTURE

This idea of a “view-state” is part of implementing a unidirectional data flow architecture. In this style of UI development, UI actions trigger updates to repositories, where those updates in turn trigger new view-states to be emitted, which trigger changes to the UI itself:

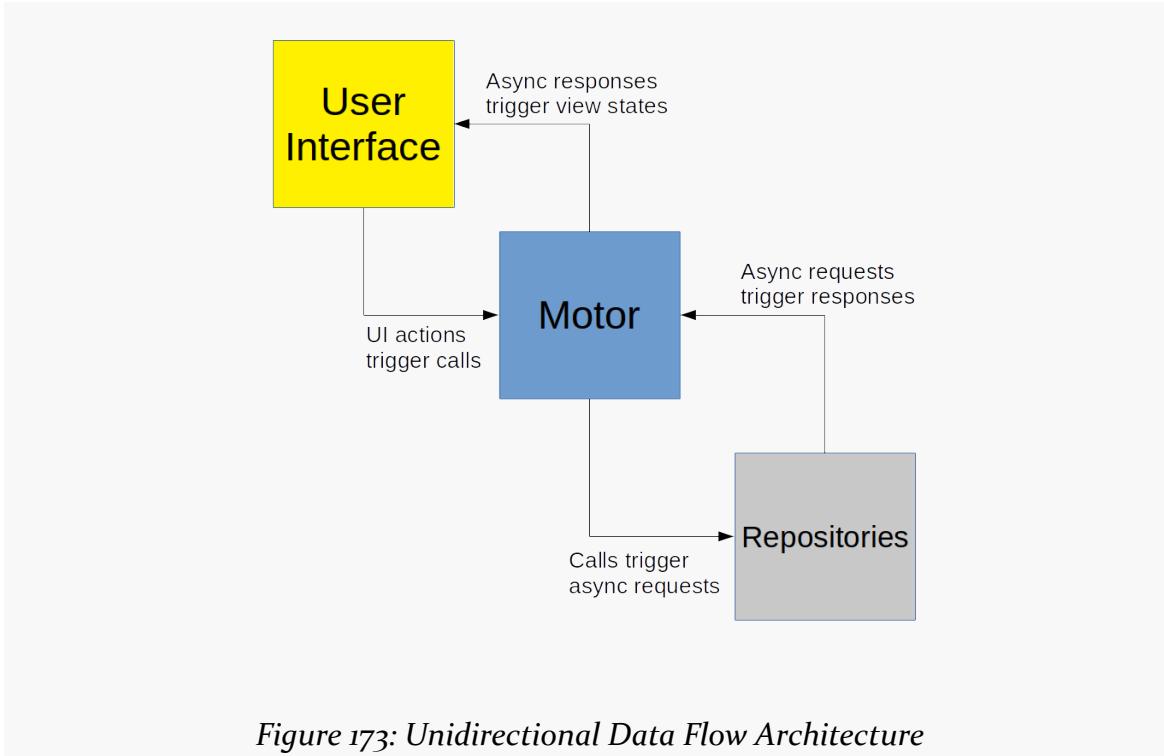


Figure 173: Unidirectional Data Flow Architecture

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Defining a Roster View State

Next, let’s create a `RosterViewState` class to represent our view-state for our `RosterListFragment`. Since this will be a small class that is tightly tied to `RosterMotor`, we can take advantage of Kotlin’s support for multiple classes in a single source file, to reduce clutter in our project tree a bit.

So, in `RosterMotor`, above the `RosterMotor` class itself, add this class:

COMPLETING THE REACTIVE ARCHITECTURE

```
data class RosterViewState(  
    val items: List<ToDoModel> = listOf()  
)
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This just holds onto our list of to-do items, though over time we will add other properties to this class.

Step #2: Emitting View States

Then, replace `getItems()` in `RosterMotor` with:

```
val states = repo.items()  
.map { RosterViewState(it) }  
.stateIn(viewModelScope, SharingStarted.Eagerly, RosterViewState())
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Flow has a `map()` operator for converting data between types. Here, we `map()` the list of `ToDoModel` objects into a `RosterViewState`.

We then use an `stateIn()` extension function. This converts a Flow into a `StateFlow`, ready to be consumed by our `RosterListFragment`.



You can learn more about `StateFlow` in the "Opting Into SharedFlow and StateFlow" chapter of [*Elements of Kotlin Coroutines!*](#)

A `StateFlow` is a flow of states. It holds onto a current state and gives that to any new observer once it starts observing. And, it emits new states to current observers if it is handed a new state.

`stateIn()` takes three parameters:

- A `CoroutineScope` (more on this below)
- A value indicating when states should start flowing — in this case, we start immediately
- The initial state for the flow — in this case, one with an empty list

For the `CoroutineScope`, we use `viewModelScope`. `viewModelScope` is an extension

COMPLETING THE REACTIVE ARCHITECTURE

function supplied by lifecycle-viewmodel-ktx, to give us a CoroutineScope associated with our ViewModel. The major feature of viewModelScope is that it is aware of the viewmodel's lifecycle. When the viewmodel is cleared (after the user exits the fragment), any outstanding coroutines being run in the context of the viewModelScope get canceled.

We use a property (states), rather than a function. For the view-state pattern, it works best if you have a stable stream of states. That will make our viewmodels a bit more complicated in the future, but it means that our fragments are simpler: just subscribe to the one source of view-states and use them. In fact, we will do just that in the next step.

Step #3: Consuming Roster View States

Our RosterListFragment now will complain that it no longer has a `getItems()` function on our RosterMotor. Specifically, we have an error on this line in `onViewCreated()`:

```
adapter.submitList(motor.getItems())
```

(from [T22-Room/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Replace that line with:

```
viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            when {
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

We start off by referencing `viewLifecycleOwner.lifecycleScope`. This gives us a CoroutineScope tied to the lifecycle of this fragment's views. Whereas a

COMPLETING THE REACTIVE ARCHITECTURE

`viewModelScope` of a `ViewModel` will cancel its outstanding coroutines once the `ViewModel` is cleared, the `CoroutineScope` that we get with `viewLifecycleOwner.lifecycleScope` is one that will cancel its outstanding coroutines once the fragment's views are destroyed (i.e., when the fragment is called with `onDestroyView()`).

The specific subtype of `CoroutineScope` that we get from `viewLifecycleOwner.lifecycleScope` is a `LifecycleCoroutineScope`, and it has some functions that give us sub-scopes based on specific lifecycle events. In this case, we use `launchWhenStarted()`. This returns a `CoroutineScope` that will:

- Start running the supplied lambda expression as a coroutine once the fragment's views are visible
- Suspend running that coroutine once the fragment's views are no longer visible
- Resumes running that coroutine if the fragment is restarted and its views become visible again
- Cancels the coroutine if the fragment's views are destroyed

The net of all of that is we have a `CoroutineScope` that does work during useful times (when the views are visible) and cleans up when our views are destroyed.

Inside that scope, we call `motor.states.collect()`. `states` is our `StateFlow` from our `RosterMotor` (`motor`). `collect()` observes the states emitted by the `StateFlow` for as long as the coroutine runs. Our supplied lambda expression gets called for each such state, including the initial empty state. So, the `state` parameter in the lambda expression is our `RosterViewState`. However, that view-state not only represents the *initial* list of to-do items, but any *changed* editions of the list that are published by the `ToDoRepository` — we keep getting view-states pushed to us as the data changes, as Room gives updates to the repository, which gives them to the `RosterMotor`, which in turn streams them to the fragment.

So, our lambda expression can update the `RosterAdapter` and the `empty` visibility, both for our initial state and after we make any changes to that state. That is why if you run the app after making these changes, you will see that the items that you add, edit, and delete have those actions reflected in the list.

Note that the `collect()` function used in `motor.states.collect()` is an extension function that needs to be imported:

```
import kotlinx.coroutines.flow.collect
```

COMPLETING THE REACTIVE ARCHITECTURE

Also, in onViewCreated() of RosterListFragment, replace:

```
binding?.empty?.visibility =  
    if (adapter.itemCount == 0) View.VISIBLE else View.GONE
```

with:

```
binding?.empty?.visibility = View.GONE
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

We are now making the empty state visible inside our when() for the view-states, so we do not need to be manipulating it here anymore.

Step #4: Wrapping the suspend Functions

RosterMotor still is showing an error message, where we cannot call save() on ToDoRepository because:

Suspend function 'save' should be called only from a coroutine or another suspend function

That is because you cannot call a suspend function from a normal function as we are doing here. Either:

- The caller needs to be a suspend function itself, or
- We need to do something that accepts suspend functions safely

To fix that, revise save() in RosterMotor to be:

```
fun save(model: ToDoModel) {  
    viewModelScope.launch {  
        repo.save(model)  
    }  
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

To consume a suspend function from a normal function, you can use launch on a CoroutineScope. In effect, launch says “I am willing to deal with suspend functions, allowing my work to be suspended as needed to wait for the suspend work to complete”.

COMPLETING THE REACTIVE ARCHITECTURE

As noted earlier, `viewModelScope` is aware of the viewmodel's lifecycle. When the viewmodel is cleared (after the user exits the fragment), any outstanding coroutines being run in the context of the `viewModelScope` get canceled. For a read operation, that is fine. However, usually, we want a write operation to proceed even if the user moved along in the UI. That is why, in `ToDoRepository`, we are using the `appScope` `COROUTINE_SCOPE` as a wrapper around the Room coroutines. `appScope` is set up to live for as long as our process does, so any coroutines executed from within it will get to run to completion, even if `viewModelScope` gets canceled.

Step #5: Updating SingleModelMotor

We need to make similar adjustments to `SingleModelMotor` that we made to `RosterMotor`.

With that in mind, add this view-state class above the declaration of `SingleModelMotor`:

```
data class SingleModelState(  
    val item: ToDoModel? = null  
)
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

And, replace the current `SingleModelMotor` implementation with:

```
class SingleModelMotor(  
    private val repo: ToDoRepository,  
    modelId: String?  
) : ViewModel() {  
    val states = repo.find(modelId)  
        .map { SingleModelState(it) }  
        .stateIn(viewModelScope, SharingStarted.Eagerly, SingleModelState())  
  
    fun save(model: ToDoModel) {  
        viewModelScope.launch {  
            repo.save(model)  
        }  
    }  
  
    fun delete(model: ToDoModel) {  
        viewModelScope.launch {  
            repo.delete(model)  
        }  
    }  
}
```

COMPLETING THE REACTIVE ARCHITECTURE

```
    }
}
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

SingleModelState is akin to RosterViewState, wrapping a single model object... or null, since we may not have a model (e.g., for a new to-do item).

states works like the RosterMotor edition, except that it calls `find()` on the `ToDoRepository` rather than `all()`. But, like RosterMotor, it maps the result to a view-state and it converts the Flow into a StateFlow.

`save()` and `delete()` both wrap their corresponding `ToDoRepository` calls in `viewModelScope.launch()`, so that those coroutines get run in our desired `CouroutineScope`.

Step #6: Adapting DisplayFragment

Similarly, we need to update `DisplayFragment` and `EditFragment` to handle the changes that we made to `SingleModelMotor`. First, let's fix `DisplayFragment`, as it is simpler: change `onViewCreated()` to be:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    viewLifecycleOwner.lifecycleScope.launchWhenStarted {
        motor.states.collect { state ->
            state.item?.let {
                binding?.apply {
                    completed.visibility =
                        if (it.isCompleted) View.VISIBLE else View.GONE
                    desc.text = it.description
                    createdOn.text = DateUtils.getRelativeDateTimeString(
                        requireContext(),
                        it.createdOn.toEpochMilli(),
                        DateUtils.MINUTE_IN_MILLIS,
                        DateUtils.WEEK_IN_MILLIS,
                        0
                    )
                    notes.text = it.notes
                }
            }
        }
    }
}
```

COMPLETING THE REACTIVE ARCHITECTURE

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/display/DisplayFragment.kt](#))

Here, we observe the view-states from our `SingleModelMotor`. When we get one, we do the same work as before: if the model is not null, we populate the widgets based on that model.

Step #7: Adapting EditFragment

Fixing `EditFragment` is more involved.

Partly, that is because we use more functions from `SingleModelMotor`, such as in the fragment's `save()` and `delete()` functions. Change those to get the `ToDoModel` by getting the value from the `StateFlow`:

```
private fun save() {
    binding?.apply {
        val model = motor.states.value.item
        val edited = model?.copy(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        ) ?: ToDoModel(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        )
        edited.let { motor.save(it) }
    }

    navToDisplay()
}

private fun delete() {
    val model = motor.states.value.item

    model?.let { motor.delete(it) }
    navToList()
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt](#))

value on `StateFlow` is whatever the last-emitted object was. In our case, it is the last-emitted view-state. So, here, we get the last-emitted view-state, get the model from it (if there was a model), and proceed as we did before.

COMPLETING THE REACTIVE ARCHITECTURE

In terms of populating the widgets, you might think that we would use the same approach that we did in `DisplayFragment`: `collect()` the `StateFlow` and use the model (if we have one) for the widget contents. Indeed, we do that... but there is a problem.

We are only updating our model when the user clicks the save app bar item. In particular, the user can edit the description or notes, or check the is-completed `CheckBox`, and *then* undergo a configuration change. Those edits are not reflected in our model, because we have not yet updated it — the user did not click the save app bar item. However, we really should try to hold onto the user's edits, as the user may get irritated if we lose them just because they rotated the screen.

The good news is that Android automatically knows how to handle those edits. That `savedInstanceState` `Bundle` that we see in functions like `onViewCreated()` contains the edits, put there by Android as part of processing the configuration change. Even better is that Android automatically updates the widgets with those edits in the new fragment after the configuration change.

We just need to not screw it up.

Specifically, we need to make sure that if we have a saved instance state, it gets used to populate our widgets. Getting the data from the model is to be used if we do not have the state.

So, with all that in mind, replace `onViewCreated()` in `EditFragment` with this implementation:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    viewLifecycleOwner.lifecycleScope.launchWhenStarted {
        motor.states.collect { state ->
            if (savedInstanceState == null) {
                state.item?.let {
                    binding?.apply {
                        isChecked = it.isCompleted
                        desc.setText(it.description)
                        notes.setText(it.notes)
                    }
                }
            }
        }
    }
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt](#))

COMPLETING THE REACTIVE ARCHITECTURE

We once again collect() our StateFlow, and we once again use the view-state to update the widgets... but only if our savedInstanceState is null. Otherwise, we assume that our widgets already have what we want from a pre-configuration change instance of our fragment.

At this point, the app should compile and run. More importantly, courtesy of the changes that we made in the past few tutorials, any to-do items that you enter will be saved and will be available in future runs of the app.

Final Results

Our module's build.gradle file should resemble:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'androidx.navigation.safeargs.kotlin'
    id 'kotlin-kapt'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

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

    buildFeatures {
        viewBinding true
    }
}
```

COMPLETING THE REACTIVE ARCHITECTURE

```
compileOptions {  
    coreLibraryDesugaringEnabled true  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}  
  
kotlinOptions {  
    jvmTarget = '1.8'  
}  
}  
  
dependencies {  
    implementation 'androidx.core:core-ktx:1.6.0'  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
    implementation "androidx.recyclerview:recyclerview:1.2.1"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation 'com.google.android.material:material:1.4.0'  
    implementation "io.insert-koin:koin-android:$koin_version"  
    implementation "androidx.room:room-runtime:$room_version"  
    implementation "androidx.room:room-ktx:$room_version"  
    kapt "androidx.room:room-compiler:$room_version"  
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'  
    testImplementation 'junit:junit:4.13.2'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
}
```

(from [T23-Arch/ToDo/app/build.gradle](#))

In the end, RosterMotor should contain:

```
package com.commonsware.todo.ui.roster  
  
import androidx.lifecycle.ViewModel  
import androidx.lifecycle.viewModelScope  
import com.commonsware.todo.repo.ToDoModel  
import com.commonsware.todo.repo.ToDoRepository  
import kotlinx.coroutines.flow.SharingStarted  
import kotlinx.coroutines.flow.map  
import kotlinx.coroutines.flow.stateIn  
import kotlinx.coroutines.launch  
  
data class RosterViewState(  
    val items: List<ToDoModel> = listOf()  
)
```

COMPLETING THE REACTIVE ARCHITECTURE

```
class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    val states = repo.items()
        .map { RosterViewState(it) }
        .stateIn(viewModelScope, SharingStarted.Eagerly, RosterViewState())

    fun save(model: ToDoModel) {
        viewModelScope.launch {
            repo.save(model)
        }
    }
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

The updated RosterListFragment should look like:

```
package com.commonsware.todo.ui.roster

import android.os.Bundle
import android.view.*
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoRosterBinding
import com.commonsware.todo.repo.ToDoModel
import kotlinx.coroutines.flow.collect
import org.koin.android.viewmodel.ext.android.viewModel

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private var binding: TodoRosterBinding? = null

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root
```

COMPLETING THE REACTIVE ARCHITECTURE

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

    val adapter = RosterAdapter(
        layoutInflater,
        onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
        onRowClick = ::display
    )

    binding?.items?.apply {
        setAdapter(adapter)
        layoutManager = LinearLayoutManager(context)

        addItemDecoration(
            DividerItemDecoration(
                activity,
                DividerItemDecoration.VERTICAL
            )
        )
    }
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            when {
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }
}

binding?.empty?.visibility = View.GONE
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
```

COMPLETING THE REACTIVE ARCHITECTURE

```
inflater.inflate(R.menu.actions_roster, menu)

super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
    }
}

return super.onOptionsItemSelected(item)
}

private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Our revised SingleModelMotor should contain:

```
package com.commonsware.todo.ui

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch

data class SingleModelState(
    val item: ToDoModel? = null
)

class SingleModelMotor(
    private val repo: ToDoRepository,
    modelId: String?
```

COMPLETING THE REACTIVE ARCHITECTURE

```
) : ViewModel() {
    val states = repo.find(modelId)
        .map { SingleModelViewState(it) }
        .stateIn(viewModelScope, SharingStarted.Eagerly, SingleModelViewState())

    fun save(model: ToDoModel) {
        viewModelScope.launch {
            repo.save(model)
        }
    }

    fun delete(model: ToDoModel) {
        viewModelScope.launch {
            repo.delete(model)
        }
    }
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

The tweaked `DisplayFragment` should resemble:

```
package com.commonsware.todo.ui.display

import android.os.Bundle
import android.text.format.DateUtils
import android.view.*
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoDisplayBinding
import com.commonsware.todo.ui.SingleModelMotor
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel
import org.koin.core.parameter.parametersOf

class DisplayFragment : Fragment() {
    private val args: DisplayFragmentArgs by navArgs()
    private var binding: TodoDisplayBinding? = null
    private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

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

        setHasOptionsMenu(true)
    }
```

COMPLETING THE REACTIVE ARCHITECTURE

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
) = TodoDisplayBinding.inflate(inflater, container, false)
    .apply { binding = this }
    .root

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    viewLifecycleOwner.lifecycleScope.launchWhenStarted {
        motor.states.collect { state ->
            state.item?.let {
                binding?.apply {
                    completed.visibility =
                        if (it.isCompleted) View.VISIBLE else View.GONE
                    desc.text = it.description
                    createdOn.text = DateUtils.getRelativeDateTimeString(
                        requireContext(),
                        it.createdOn.toEpochMilli(),
                        DateUtils.MINUTE_IN_MILLIS,
                        DateUtils.WEEK_IN_MILLIS,
                        0
                    )
                    notes.text = it.notes
                }
            }
        }
    }
}

override fun onDestoryView() {
    binding = null

    super.onDestoryView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_display, menu)

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.edit -> {
            edit()
            return true
        }
    }
}
```

COMPLETING THE REACTIVE ARCHITECTURE

```
        }
    }

    return super.onOptionsItemSelected(item)
}

private fun edit() {
    findNavController().navigate(
        DisplayFragmentDirections.editModel(
            args.modelId
        )
    )
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/display/DisplayFragment.kt](#))

And the current EditFragment should look like:

```
package com.commonsware.todo.ui.edit

import android.os.Bundle
import android.view.*
import android.view.inputmethod.InputMethodManager
import androidx.core.content.getSystemService
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoEditBinding
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.ui.SingleModelMotor
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel
import org.koin.core.parameter.parametersOf

class EditFragment : Fragment() {
    private var binding: TodoEditBinding? = null
    private val args: EditFragmentArgs by navArgs()
    private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }

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

        setHasOptionsMenu(true)
    }
```

COMPLETING THE REACTIVE ARCHITECTURE

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
) = TodoEditBinding.inflate(inflater, container, false)
    .apply { binding = this }
    .root

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    viewLifecycleOwner.lifecycleScope.launchWhenStarted {
        motor.states.collect { state ->
            if (savedInstanceState == null) {
                state.item?.let {
                    binding?.apply {
                        isChecked = it.isCompleted
                        desc.setText(it.description)
                        notes.setText(it.notes)
                    }
                }
            }
        }
    }
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_edit, menu)
    menu.findItem(R.id.delete).isVisible = args.modelId != null

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.save -> {
            save()
            return true
        }
        R.id.delete -> {
            delete()
            return true
        }
    }
}
```

COMPLETING THE REACTIVE ARCHITECTURE

```
    return super.onOptionsItemSelected(item)
}

private fun save() {
    binding?.apply {
        val model = motor.states.value.item
        val edited = model?.copy(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        ) ?: ToDoModel(
            description = desc.text.toString(),
            isCompleted = isCompleted.isChecked,
            notes = notes.text.toString()
        )
        edited.let { motor.save(it) }
    }
}

navToDisplay()
}

private fun delete() {
    val model = motor.states.value.item

    model?.let { motor.delete(it) }
    navToList()
}

private fun navToDisplay() {
    hideKeyboard()
    findNavController().popBackStack()
}

private fun navToList() {
    hideKeyboard()
    findNavController().popBackStack(R.id.rosterListFragment, false)
}

private fun hideKeyboard() {
    view?.let {
        val imm = context?.getSystemService<InputMethodManager>()

        imm?.hideSoftInputFromWindow(
            it.windowToken,
            InputMethodManager.HIDE_NOT_ALWAYS
        )
    }
}
```

COMPLETING THE REACTIVE ARCHITECTURE

```
    }  
}  
}
```

(from [T23-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/build.gradle](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/
RosterListFragment.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/display/
DisplayFragment.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt](#)

Phase Three: Testing This Thing

Testing a Motor

We think that our app works, in that we can see it working when we use the app's UI. Besides, this is a book, and books *never* have mistakes, right?

(right?!?)

In the real world, though, you do not have a set of tutorials for every bit of code that you want to write. Along the way, writing tests will help you confirm that the code that you wrote actually works, including for scenarios that are supported by the API that you created but might not be used yet by the UI. So, in this tutorial, we will start adding some tests to our project.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab 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 basics of testing in the "Touring the Tests" chapter of [Elements of Android Jetpack](#)!

Step #1: Examine Our Existing Tests

The good news is that the project you imported to start these tutorials already has some tests written for you.

(no, this does not mean that you are done with testing)

Tests in Android modules go into “source sets” that are peers of `main/`. If you

TESTING A MOTOR

examine your project in Android Studio, you will see that there are three directories in app/src/: androidTest/, main/, and test/:

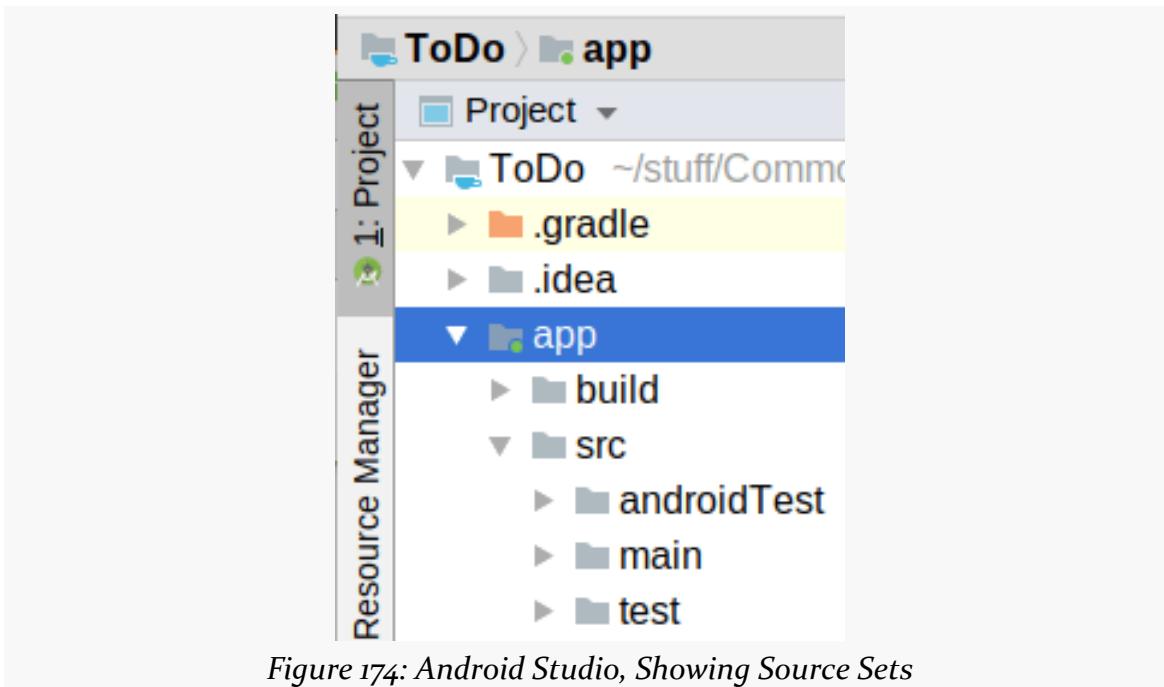


Figure 174: Android Studio, Showing Source Sets

androidTest/ holds “instrumented tests”. Simply put, these are tests of our code that run on an Android device or emulator, just as our app does. If you go into that directory, you will see that it has its own java/ tree, with an ExampleInstrumentedTest defined there:

```
package com.commonsware.todo

import androidx.test.platform.app.InstrumentationRegistry
import androidx.test.ext.junit.runners.AndroidJUnit4

import org.junit.Test
import org.junit.runner.RunWith

import org.junit.Assert.*

/**
 * Instrumented test, which will execute on an Android device.
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
@RunWith(AndroidJUnit4::class)
```

TESTING A MOTOR

```
class ExampleInstrumentedTest {
    @Test
    fun useApplicationContext() {
        // Context of the app under test.
        val appContext = InstrumentationRegistry.getInstrumentation().targetContext
        assertEquals("com.commonsware.todo", appContext.packageName)
    }
}
```

(from [T23-Arch/ToDo/app/src/androidTest/java/com/commonsware/todo/ExampleInstrumentedTest.kt](#))

test/ holds “unit tests”. These are tests of our code that run directly on our development machine. On the plus side, they run much faster, as we do not have to copy the test code over to a device or emulator, and a device or emulator is going to be slower than our development machine (usually). On the other hand, our development machine is not running Android, so we cannot easily test code that touches Android-specific classes and methods. Like androidTest/, test/ has its own java/ tree, with an ExampleUnitTest defined there:

```
package com.commonsware.todo

import org.junit.Test

import org.junit.Assert.*

/**
 * Example local unit test, which will execute on the development machine (host).
 *
 * See [testing documentation](http://d.android.com/tools/testing).
 */
class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        assertEquals(4, 2 + 2)
    }
}
```

(from [T23-Arch/ToDo/app/src/test/java/com/commonsware/todo/ExampleUnitTest.kt](#))

Neither of these test very much, let alone anything related to our own code.

Step #2: Decide on Instrumented Tests vs. Unit Tests

So, which should we use? Instrumented tests? Unit tests? Both?

If you only wanted to worry about one, choose instrumented tests. Everything can be tested using instrumented tests, while unit tests cannot readily test everything.

TESTING A MOTOR

And, for a small project like this one, going with instrumented tests for everything would be perfectly reasonable. However, most projects are not this small.

For larger projects — particularly those where tests will be run frequently — the speed gain from unit tests can be significant. So, a typical philosophy is:

- Test what you can with unit tests
- Test the other stuff, such as the UI, with instrumented tests

That is the approach that we will take over the next few tutorials, starting with some unit tests.

Step #3: Adding Some Unit Test Dependencies

So far, all of the dependencies that we have been adding to our app have used the `implementation` keyword. Those dependencies become part of the main app.

However, our dependencies closure in `app/build.gradle` also has `androidTestImplementation` and `testImplementation` statements. These are for instrumented tests and unit tests, respectively:

Test Type	Where the Source Goes	How You Add Dependencies
instrumented test	<code>androidTest</code>	<code>androidTestImplementation</code>
unit test	<code>test</code>	<code>testImplementation</code>

Right now, we have just one `testImplementation` dependency, for JUnit. JUnit is the foundation of all Android unit tests and instrumented tests, so we will be writing JUnit-based tests for both types.

Technically, we do not need anything more than that for our unit tests. In practice, though, usually we add some more dependencies, ones that will help us test more effectively.

With that in mind, add these lines to the dependencies closure in `app/build.gradle`:

TESTING A MOTOR

```
testImplementation "org.mockito:mockito-inline:3.12.1"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"
testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
```

(from [T24-Tests/ToDo/app/build.gradle](#))

The `org.mockito:mockito-inline` and `com.nhaarman.mockitokotlin2:mockito-kotlin` bring in [Mockito](#) and [a Kotlin wrapper for Mockito](#). Mockito is a popular “mocking” library, allowing us to define *ad hoc* implementations of classes. Sometimes, we use these to test scenarios that would be difficult to test otherwise. Sometimes, we use these as “call recorders”, to see whether certain functions were called as part of our tests.

The `org.jetbrains.kotlinx:kotlinx-coroutines-test` library helps you test coroutines. The author hopes that this was not a surprise.

(if it was... surprise!)

Step #4: Renaming Our Unit Test

Our unit test class is named `ExampleUnitTest`. That is not a particularly useful name for us. Since we will be using this class to test `SingleModelMotor`, we should rename it to something like `SingleModelMotorTest`. Typically, a unit test class focuses on testing one main project class, so `SingleModelMotorTest` would focus on testing `SingleModelMotor`.

Also, typically, the test class resides in the same package as is the class that it is testing. So, just as `SingleModelMotor` is in `com.commonsware.todo.ui`, so should `SingleModelMotorTest`. However, we do not have a `ui` sub-package in the test source set — the one that we added earlier is in the main source set.

In the test source set, right-click over the `com.commonsware.todo` package and choose “New” > “Package” from the context menu. Fill in `com.commonsware.todo.ui` for the new package name, then click “OK” to create the sub-package.

TESTING A MOTOR

Next, drag and drop the ExampleUnitTest class from its current location (inside com.commonware.todo) into this new ui sub-package. As before when we moved around classes, this will bring up a “Move” dialog:

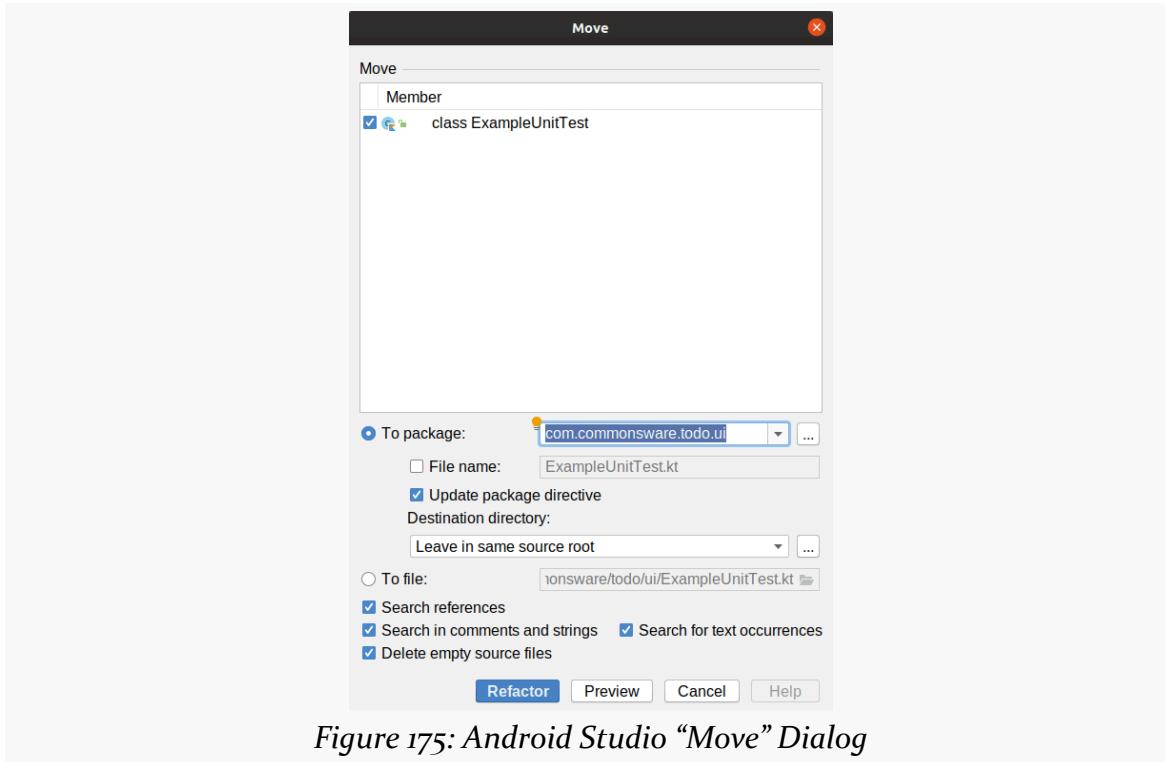


Figure 175: Android Studio “Move” Dialog

Just click the “Refactor” button to complete the move.

TESTING A MOTOR

Then, right-click over the newly-moved ExampleUnitTest and choose “Refactor” > “Rename” from the context menu. In the “Rename” dialog, fill in SingleModelMotorTest as the new name:

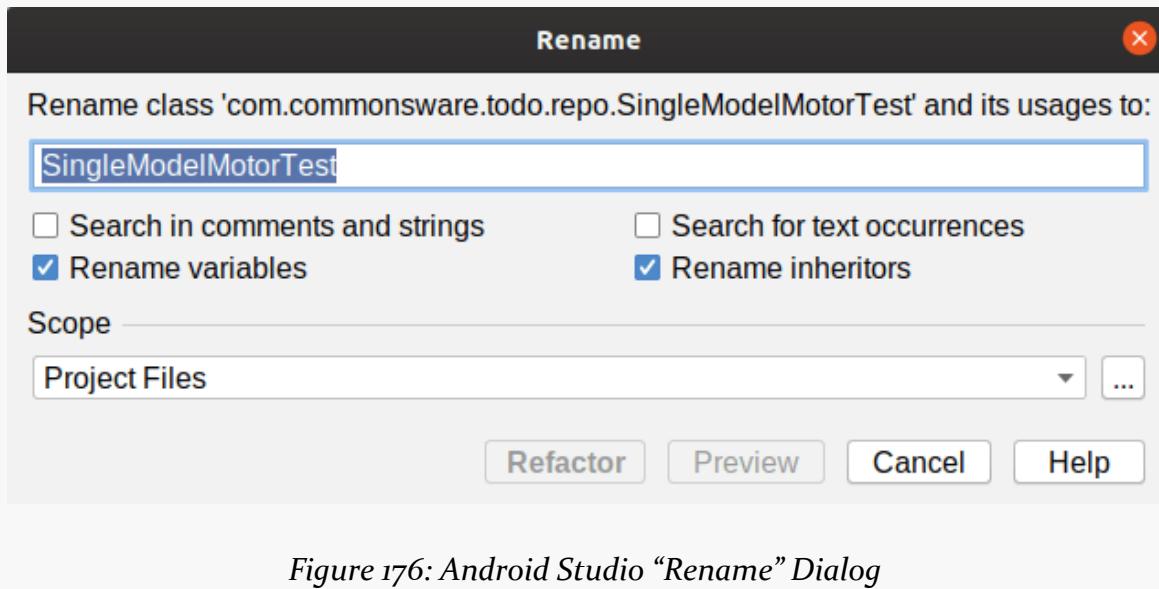


Figure 176: Android Studio “Rename” Dialog

Then click “Refactor”. This will rename both the file and the Kotlin class, so we now have a SingleModelMotorTest.

Step #5: Running the Stub Unit Test

You have a variety of ways to run the unit test. The simplest ones come from “run” icons in the gutter of the editor:

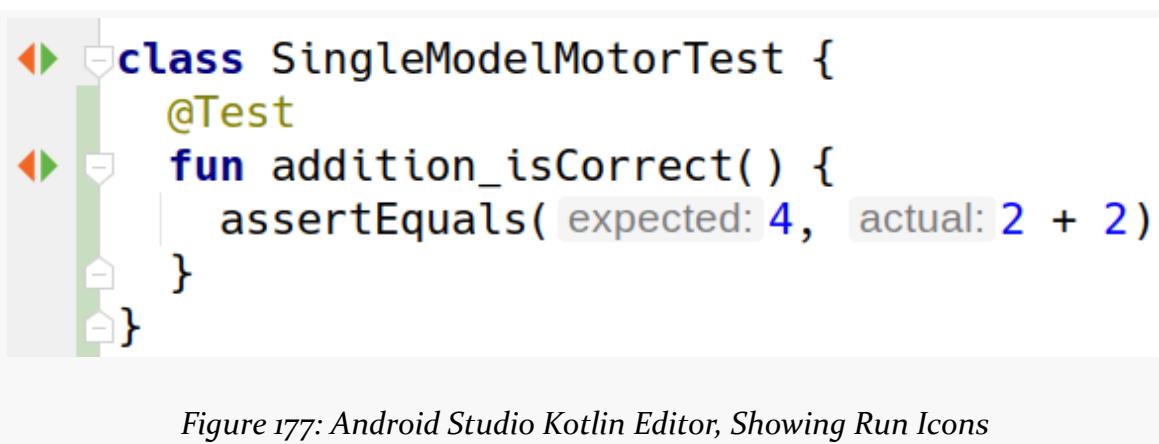


Figure 177: Android Studio Kotlin Editor, Showing Run Icons

TESTING A MOTOR

Functions that implement tests will have the @Test annotation. Clicking the run icon next to a test function will run just that test function. Clicking the run icon next to a class will run all of the test functions in that class.

If you click the run icon next to SingleModelMotorTest, that class' test functions will be run, and the “Run” tool window will open in Android Studio to show you the results:

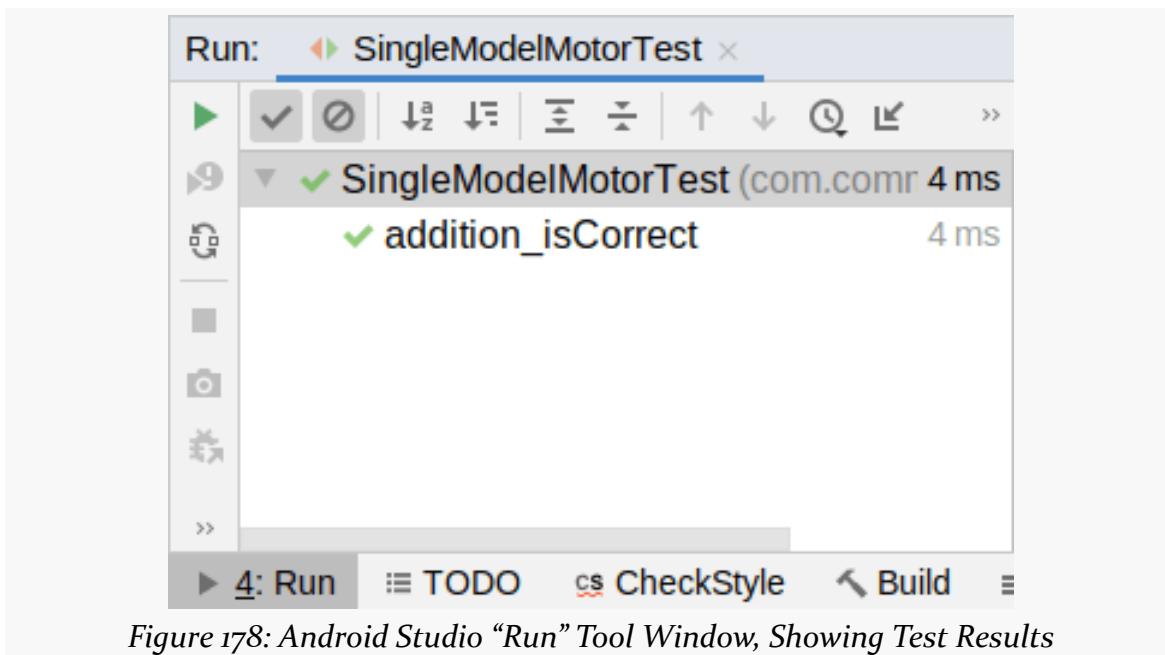


Figure 178: Android Studio “Run” Tool Window, Showing Test Results

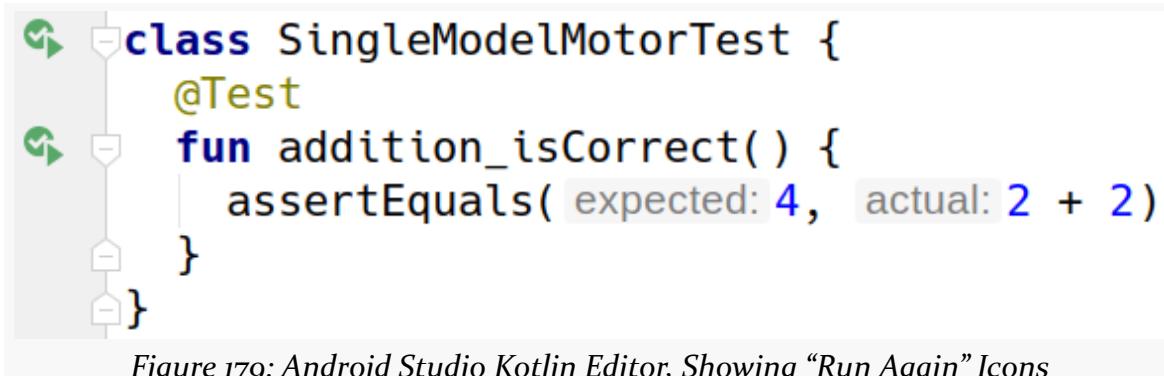
Our test function uses an `assertEquals()` method supplied by JUnit. `assertEquals()` compares two values and fails the test if they are not equal.

Not surprisingly, $2 + 2$ does indeed equal 4.

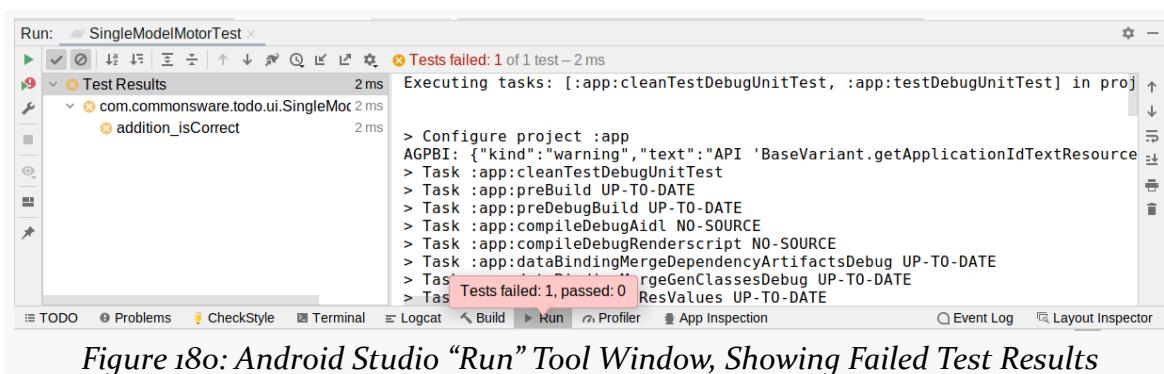
(if you were surprised by this, once again... surprise!)

TESTING A MOTOR

The test output shows passing tests with a green checkmark, so we can see that our test passed. Also, at this point, the run icons in the editor become “run again” icons, with the green check-circle indicating that the previous test passed:



If you change the `assertEquals()` call to be `assertEquals(5, 2 + 2)` and run the test again, you will see that it fails:



The yellow icon indicates that the test failed due to a failed assertion.

TESTING A MOTOR

If you change the `assertEquals()` call to be `assertEquals(4, 2 / 0)` and run the test again, you will see that the test fails again. This time, though, the test output uses a red icon, to indicate that our test crashed:



Figure 181: Android Studio “Run” Tool Window, Showing Crashed Test Results

Usually, after a test failure, our editor icons turn red, indicating that the previous run of the test failed:

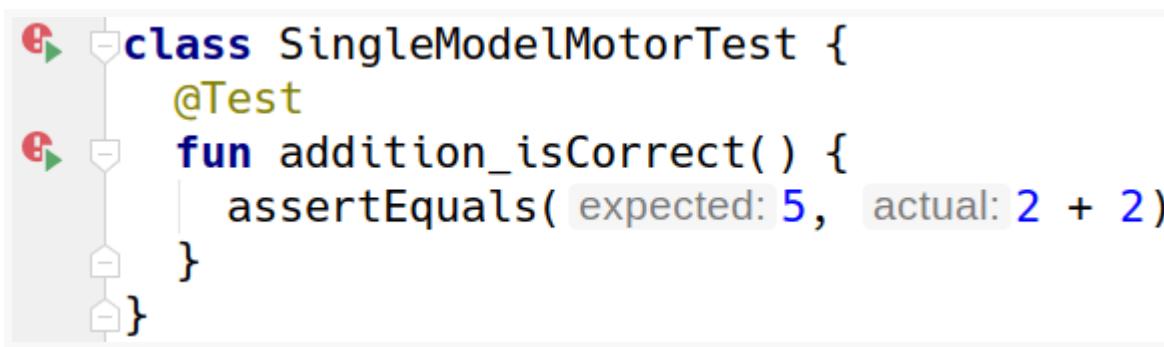


Figure 182: Android Studio Kotlin Editor, Showing “Run Failed Test Again” Icons

As we add more test functions, you may get a mix of results, with some tests succeeding and some tests failing. The test class is only considered to have succeeded if all of its test functions succeed.

Step #6: Adding a MainDispatcherRule

Implicitly, `SingleModelMotor` uses `Dispatchers.Main` — that is the default coroutine dispatcher for `viewModelScope.launch()`. Dispatchers in coroutines control the threads that coroutines run on. In an Android app, `Dispatchers.Main` says “run this code on the main application thread”.

TESTING A MOTOR

In JUnit, all tests run on a test thread — and in a unit test, such as this, since we are not running on Android, there is no “Android’s main application thread”. As a side effect, there is no definition of `Dispatchers.Main`.

We need to do something in our app to teach the coroutines system what to use when we reference `Dispatchers.Main` in our code. The coroutines testing library that we just added contains a `TestCoroutineDispatcher` that we can use, but we need to tell the coroutines system to use a `TestCoroutineDispatcher` for `Dispatchers.Main`.

Right-click over the `com.commonsware.todo.ui` package in the test/ source set and choose “New” > “Kotlin File/Class” from the context menu. Fill in `MainDispatcherRule` as the name, and choose “Class” for the kind. Click “OK” to create the empty class.

This puts `MainDispatcherRule` in `com.commonsware.todo.ui`. We did not have much of a choice but to put it there, as that is our one-and-only package, and the new-class “dialog” does not let us choose a different package. However, this class is not strictly tied to the UI classes. So, let’s move it into `com.commonsware.todo` instead.

To do that, right-click over `MainDispatcherRule` and choose “Refactor” > “Move” from the context menu. In the “To package:” field, change the package to be `com.commonsware.todo`, then click “Refactor”. This will move `MainDispatcherRule` to that package.

Then, replace the contents of `MainDispatcherRule` with this Kotlin code:

```
package com.commonsware.todo

import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.test.TestCoroutineDispatcher
import kotlinx.coroutines.test.resetMain
import kotlinx.coroutines.test.setMain
import org.junit.rules.TestWatcher
import org.junit.runner.Description

// inspired by https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471

class MainDispatcherRule(paused: Boolean) : TestWatcher() {
    val dispatcher =
        TestCoroutineDispatcher().apply { if (paused) pauseDispatcher() }

    override fun starting(description: Description?) {
        super.starting(description)

        Dispatchers.setMain(dispatcher)
    }
}
```

TESTING A MOTOR

```
override fun finished(description: Description?) {
    super.finished(description)

    Dispatchers.resetMain()
    dispatcher.cleanupTestCoroutines()
}
}
```

(from [T24-Tests/ToDo/app/src/test/java/com/commonsware/todo/MainDispatcherRule.kt](#))

In JUnit, a Rule is a standard way to package reusable bits of test logic, particularly related to common test configuration. Here, we define our own custom rule, by extending TestWatcher. MainDispatcherRule says “do threading differently for coroutines in this test”.

Specifically, we create a TestCoroutineDispatcher and use that for Dispatchers.Main.starting() is called on our TestWatcher when a test is starting, and there we call Dispatches.setMain() to provide a dispatcher to use for Dispatchers.Main.finished() is called on our TestWatcher when a test is ending, and there we call Dispatchers.resetMain() to reset the Dispatchers.Main definition to its default. We also call cleanupTestCoroutines() on our TestCoroutineDispatcher, to indicate that we are done with this dispatcher and anything still outstanding should be canceled.

If you add this code to your project, Android Studio will have some complaints:

```
class MainDispatcherRule(paused: Boolean) : TestWatcher() {
    val dispatcher: TestCoroutineDispatcher =
        TestCoroutineDispatcher().apply { if (paused) pauseDispatcher() }

    override fun starting(description: Description?) {
        super.starting(description)

        Dispatchers.setMain(dispatcher)
    }

    override fun finished(description: Description?) {
        super.finished(description)

        Dispatchers.resetMain()
        dispatcher.cleanupTestCoroutines()
    }
}
```

Figure 183: Android Studio Warnings

If you hover your mouse over those yellow warnings, you will find that the problem is that all of those things are considered “experimental” by JetBrains (the creators of Kotlin). It is possible that these classes and functions will be renamed or even removed in some future version of coroutines. That is a problem for the future — for now, this code will work fine.

Step #7: Setting Up a Mock Repository

Now, we can start setting up some tests. As part of this, you will start to discover that testing code frequently has strange restrictions and requirements, above and beyond the strange restrictions and requirements that you see in standard Android app development.

Next, change `SingleModelMotorTest` to be:

```
package com.commonsware.todo.ui

import com.commonsware.todo.MainDispatcherRule
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.nhaarman.mockitokotlin2.doReturn
import com.nhaarman.mockitokotlin2.mock
import kotlinx.coroutines.flow.flowOf
import org.junit.Test

import org.junit.Assert.*
import org.junit.Before
import org.junit.Rule

class SingleModelMotorTest {
    @get:Rule
    val mainDispatcherRule = MainDispatcherRule(paused = true)

    private val testModel = ToDoModel("this is a test")

    private val repo: ToDoRepository = mock {
        on { find(testModel.id) } doReturn flowOf(testModel)
    }

    private lateinit var underTest: SingleModelMotor

    @Before
    fun setUp() {
```

TESTING A MOTOR

```
    underTest = SingleModelMotor(repo, testModel.id)
}
}
```

There is quite a bit to explain in these few lines of code.

```
@get:Rule
val mainDispatcherRule = MainDispatcherRule(paused = true)
```

This applies our `MainDispatcherRule` as a JUnit rule to our JUnit test.

The `@get:Rule` syntax is a side-effect of the way Kotlin integrates with Java. If this were a Java class, we would annotate our rule field with `@Rule`. `@get:Rule` says “add the `@Rule` annotation to the getter function associated with this property”. JUnit’s annotation processor supports the `@Rule` annotation being on a field or on a getter method, so `@get:Rule` allows that annotation processor to work with a Kotlin property.

```
private lateinit var underTest: SingleModelMotor

@Before
fun setUp() {
    underTest = SingleModelMotor(repo, testModel.id)
}
```

We then have an `underTest` property for our `SingleModelMotor`. `underTest` is a common name in unit tests for “the instance of the class that we are testing”.

`@Before` is a JUnit annotation that says “run this function before each of the test functions”. Here, we create our `SingleModelMotor` instance. Ideally, we would just use a `val` and initialize our `SingleModelMotor` that way, skipping this `setUp()` function. Unfortunately, our `MainDispatcherRule` will not have had a chance to do its work yet. So, we are forced to use this approach, so the `MainDispatcherRule` can fix up the threading before we try creating a `SingleModelMotor` instance.

```
private val testModel = ToDoModel("this is a test")

private val repo: ToDoRepository = mock {
    on { find(testModel.id) } doReturn flowOf(testModel)
}
```

In Android testing, we use mocks for two main things.

One is for creating a fake instance of some object, one that we teach how to respond

TESTING A MOTOR

to various function calls. This is not the object that we are trying to test, but it is some object that is needed by what we are trying to test... such as a motor needing a repository. We use the mock instead of a real instance of the object for a variety of reasons:

- To have faster tests (e.g., to avoid database I/O)
- To provide specific responses to calls (particularly for server calls that we cannot control in the tests)
- To test scenarios that are difficult to recreate using the real object (e.g., server failures)

Another is to track which calls are made on the object. That way not only can our tests supply input to the object being tested, but we can examine the output, in the form of calls to the mock, and confirm that those calls did what we want.

Here, we use a `mock()` function from Mockito to set up a mock implementation of `ToDoRepository`. It generates an instance of a generated subclass of `ToDoRepository`, one where we can dictate how it behaves in our test code, rather than relying on the real `ToDoRepository` implementation. Specifically, we have it return a manually-created `ToDoModel` instance (`testModel`) when something tries finding a model with that model's id.

Step #8: Adding a Test Function

Now, we can start testing `SingleModelMotor`.

Add this test function to `SingleModelMotorTest`:

```
@Test
fun `initial state`() {
    mainDispatcherRule.dispatcher.runCurrent()

    runBlocking {
        val item = underTest.states.first().item

        assertEquals(testModel, item)
    }
}
```

(from [T24-Tests/ToDo/app/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

Here, we start by calling `mainDispatcherRule.dispatcher.runCurrent()` to get our

TESTING A MOTOR

TestCoroutineDispatcher from the MainDispatcherRule and tell that dispatcher to run any coroutines that are set up for that dispatcher. Our viewmodel functions are setting up coroutines to run on Dispatchers.Main, and Dispatchers.Main is tied to our TestCoroutineDispatcher through our MainDispatcherRule. The effect is that when we call runCurrent(), we cause those coroutines to be executed, making their requests of our (mock) repository. runCurrent() shows up with warning highlights, as it too is experimental, as with much of the test coroutines API that we are calling in MainDispatcherRule.

After that, we:

- Use runBlocking() to say that we want to execute a block of code synchronously even though it uses a suspend function (first())
- In that block, call first() on our StateFlow to get the view-state
- Use JUnit's assertEquals() function to confirm that the item from the view-state is our test ToDoModel instance

If you run this test function, it should succeed.

Step #9: Adding Another Test Function

That tests our motor's states — we should also test the actions.

Add this test function to SingleModelMotorTest:

```
@Test
fun `actions pass through to repo`() {
    val replacement = testModel.copy("whatevs")

    underTest.save(replacement)
    mainDispatcherRule.dispatcher.runCurrent()

    runBlocking { verify(repo).save(replacement) }

    underTest.delete(replacement)
    mainDispatcherRule.dispatcher.runCurrent()

    runBlocking { verify(repo).delete(replacement) }
}
```

(from [T24-Tests/ToDo/app/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

Here, we are confirming that we can save and delete properly. To do this, we need to

TESTING A MOTOR

determine if `save()` and `delete()` on the motor are calling `save()` and `delete()` on the repository. Since our repository is a mock, it tracks all calls made to it during a test run. We can then have test code check to see if the calls were made as expected.

So, we call `save()` and `delete()` on our `SingleModelMotor` that we are testing. After each of those calls, we call `mainDispatcherRule.dispatcher.runCurrent()`, to ensure that the actual work executes, now that we are ready for it to do so.

We then want to verify that our repository was called to save and delete those models. `verify()`, from Mockito, lets us see if a particular call was made on our mock. To do this, we pass the `mock()` to `verify`, then make the call that we want verified on the `ToDoRepository` object returned by `verify()`. This will either work or fail with an assertion error.

However, since `save()` and `delete()` in our repository are suspend functions, though, we need to run them inside of some `CoroutineScope`, even though those are really mock functions. `runBlocking()` will suffice for our mock call verification.

If you run all the tests on `SingleModelMotorTest`, they should all succeed.

Final Results

Our `app/build.gradle` file with the updated dependencies list should resemble:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'androidx.navigation.safeargs.kotlin'
    id 'kotlin-kapt'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
}
```

TESTING A MOTOR

```
buildTypes {  
    release {  
        minifyEnabled false  
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),  
        'proguard-rules.pro'  
    }  
}  
  
buildFeatures {  
    viewBinding true  
}  
  
compileOptions {  
    coreLibraryDesugaringEnabled true  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}  
  
kotlinOptions {  
    jvmTarget = '1.8'  
}  
}  
  
dependencies {  
    implementation 'androidx.core:core-ktx:1.6.0'  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
    implementation "androidx.recyclerview:recyclerview:1.2.1"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation 'com.google.android.material:material:1.4.0'  
    implementation "io.insert-koin:koin-android:$koin_version"  
    implementation "androidx.room:room-runtime:$room_version"  
    implementation "androidx.room:room-ktx:$room_version"  
    kapt "androidx.room:room-compiler:$room_version"  
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'  
    testImplementation 'junit:junit:4.13.2'  
    testImplementation "org.mockito:mockito-inline:3.12.1"  
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"  
    testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
}
```

(from [T24-Tests/ToDo/app/build.gradle](#))

And our SingleModelMotorTest should look like:

TESTING A MOTOR

```
package com.commonsware.todo.ui

import com.commonsware.todo.MainDispatcherRule
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.nhaarman.mockitokotlin2.doReturn
import com.nhaarman.mockitokotlin2.mock
import com.nhaarman.mockitokotlin2.verify
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.flow.flowOf
import kotlinx.coroutines.runBlocking
import org.junit.Assert.assertEquals
import org.junit.Before
import org.junit.Rule
import org.junit.Test

class SingleModelMotorTest {
    @get:Rule
    val mainDispatcherRule = MainDispatcherRule(paused = true)

    private val testModel = ToDoModel("this is a test")

    private val repo: ToDoRepository = mock {
        on { find(testModel.id) } doReturn flowOf(testModel)
    }

    private lateinit var underTest: SingleModelMotor

    @Before
    fun setUp() {
        underTest = SingleModelMotor(repo, testModel.id)
    }

    @Test
    fun `initial state`() {
        mainDispatcherRule.dispatcher.runCurrent()

        runBlocking {
            val item = underTest.states.first().item

            assertEquals(testModel, item)
        }
    }

    @Test
    fun `actions pass through to repo`() {
        val replacement = testModel.copy("whatevs")
    }
}
```

TESTING A MOTOR

```
underTest.save(replacement)
mainDispatcherRule.dispatcher.runCurrent()

runBlocking { verify(repo).save(replacement) }

underTest.delete(replacement)
mainDispatcherRule.dispatcher.runCurrent()

runBlocking { verify(repo).delete(replacement) }
}

}
```

(from [T24-Tests/ToDo/app/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/build.gradle](#)
- [app/src/test/java/com/commonsware/todo/MainDispatcherTest.kt](#)
- [app/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#)

Testing the Repository

The objective of a test suite is to completely test the functionality of the main code, including all code paths. Often, this gets measured in the form of “test coverage”, where we confirm:

- Whether all lines of code were executed
- Whether both the `true` and `false` branches of an `if` condition were taken
- Whether a loop was executed 0, 1, and N times
- And so on

This project does not have 100% test coverage. Few projects presented in books have 100% test coverage.

This tutorial extends our test coverage a bit, by testing `ToDoRepository` and our Room code. To do that, though, we will switch to writing instrumented tests. Room is designed to use Android’s SQLite by default, and so it is much easier to test this stuff when we run our tests on Android, rather than on our development machine.

Testing in instrumented tests is a lot like unit testing:

- We write JUnit tests with `@Test` functions and so on
- We use assertions to determine whether our tests succeed or fail
- We can run the tests from Android Studio to see whether they work

On the other hand, there are substantial differences as well:

- The tests will run in an Android environment, on our chosen device or emulator
- The tests will be subject to some Android limitations, just as the regular Kotlin code that we might use on a server will not necessarily work in

Android

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Renaming Our Instrumented Test

The existing instrumented test class, inside the androidTest source set, is ExampleInstrumentedTest. This is not a very useful name. Since we will be testing some of the functionality from ToDoRepository, we should rename it to ToDoRepositoryTest. And, since ToDoRepository is in the repo sub-package, we should have the test class mimic that.

In the androidTest source set, right click over the com.commonsware.todo.repo package and choose “New” > “Package” from the context menu. Fill in com.commonsware.todo.repo for the name, then click “OK” to make this sub-package.

Then, drag-and-drop the ExampleInstrumentedTest into this new repo sub-package. The default values in the “Move” dialog should be fine, so just click “Refactor” to make the move.

Finally, right-click over the ExampleInstrumentedTest class and choose “Refactor” > “Rename” from the context menu. Fill in ToDoRepositoryTest as the replacement name, and click “Refactor” to make the change.

Step #2: Adding Some Instrumented Test Dependencies

Right now, our dependencies closure in app/build.gradle has two androidTestImplementation statements:

```
androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
```

(from [T24-Tests/ToDo/app/build.gradle](#))

Add these lines to those:

TESTING THE REPOSITORY

```
androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
```

(from [T25-RepoTests/ToDo/app/build.gradle](#))

The `androidx.arch.core:core-testing` library contains some JUnit rules and related classes that are commonly needed in Android app testing.

Similarly, the `org.jetbrains.kotlinx:kotlinx-coroutines-test` library is one that we used in unit testing, but we now also want to use it for instrumented testing.

Also, add these lines inside the `android` closure:

```
packagingOptions {
    exclude 'META-INF/AL2.0'
    exclude 'META-INF/LGPL2.1'
}
```

(from [T25-RepoTests/ToDo/app/build.gradle](#))

Sometimes, libraries package open source license files along with their compiled code. And, sometimes, that results in collisions, where two libraries put the same license text in the same files. This snippet of Gradle code says to exclude all of those from the app that we are building.

Step #3: Supporting a Test Database

Right now, `ToDoDatabase` is set up to have a database file named `stuff.db`. For our “production” code, that is what we want. For tests, though, it is very convenient to have an in-memory database:

- Tests run faster
- All of our test stuff gets cleared between tests, as a side-effect of how Room and in-memory databases work

With that in mind, add this function to the `companion object` in `ToDoDatabase`:

```
fun newTestInstance(context: Context) =
    Room.inMemoryDatabaseBuilder(context, ToDoDatabase::class.java).build()
```

(from [T26-Espresso/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#))

This is almost identical to the `newInstance()` function that already exists. However, `newTestInstance()` uses `inMemoryDatabaseBuilder()` instead of

TESTING THE REPOSITORY

databaseBuilder(), to create an in-memory SQLite database.

Step #4: Testing Adds

Now, we can start putting in test logic for testing ToDoRepository itself.

Replace the ToDoRepositoryTest implementation with:

```
package com.commonsware.todo.repo

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import kotlinx.coroutines.collect
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import kotlinx.coroutines.test.runBlockingTest
import org.hamcrest.Matchers.empty
import org.hamcrest.Matchers.equalTo
import org.hamcrest.collection.IsIterableContainingInOrder.contains
import org.junit.Assert.assertThat
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class ToDoRepositoryTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    private val context = InstrumentationRegistry.getInstrumentation().targetContext
    private val db = ToDoDatabase.newTestInstance(context)

    @Test
    fun canAddItems() = runBlockingTest {
        val underTest = ToDoRepository(db.todoStore(), this)
        val results = mutableListOf<List<ToDoModel>>()

        val itemsJob = launch {
            underTest.items().collect { results.add(it) }
        }

        assertThat(results.size, equalTo(1))
        assertThat(results[0], empty())

        val testModel = ToDoModel("test model")
```

TESTING THE REPOSITORY

```
underTest.save(testModel)

assertThat(results.size, equalTo(2))
assertThat(results[1], contains(testModel))
assertThat(underTest.find(testModel.id).first(), equalTo(testModel))

    itemsJob.cancel()
}
}
```

Once again, we have a lot to explain.

```
@RunWith(AndroidJUnit4::class)
```

The `@RunWith` annotation gives JUnit a specific class to use to orchestrate running the test functions in this test class. For unit tests, by default we do not need to use this annotation, though certain libraries that you might use could require one. For instrumented tests, though, we need to point to a class that knows how to run the test and get the results off the device or emulator and over to the IDE. That is what `AndroidJUnit4` helps with, in part. Unless you are using some other library that requires a different `@RunWith` annotation, all of your instrumented tests will start with this line.

```
@get:Rule
val instantTaskExecutorRule = InstantTaskExecutorRule()
```

This is another JUnit rule, one provided by the Jetpack testing library that we added to our dependencies. Like our `MainDispatcherRule`, `InstantTaskExecutorRule` ensures that our Room and other Jetpack asynchronous work really happens synchronously, to simplify our tests.

```
private val context = InstrumentationRegistry.getInstrumentation().targetContext
```

We are going to need a Context to be able to set up our Room database. Specifically, we want a Context in the “context” of the code being tested (our app code). To get such a Context, we can ask an `InstrumentationRegistry` to give us an `Instrumentation` object representing our instrumented tests, and on there retrieve `targetContext`.

```
private val db = ToDoDatabase.newTestInstance(context)
```

From there, we can set up a `ToDoDatabase`, using our newly-added

TESTING THE REPOSITORY

`newTestInstance()` function and the context that we just obtained.

```
@Test  
fun canAddItems() = runBlockingTest {
```

As with `SingleModelMotorTest`, we are going to be working with Kotlin coroutines. This time, though, we are running on Android, so we do not need to fuss with trying to change the nature of `Dispatchers.Main`. However, we do need to worry about ensuring that we have a `CoroutineScope` to use for our tests. In `SingleModelMotorTest`, we used `runBlocking()` where needed, and we used a `TestCoroutineDispatcher` inside of `MainDispatcherRule`. This time, we are using `runBlockingTest()`, which sets up a `TestCoroutineDispatcher` and uses that to have all of our coroutines run synchronously. Using `runBlocking()` has more flexibility; using `runBlockingTest()` frequently is simpler.

```
    val underTest = ToDoRepository(db.todoStore(), this)
```

We then can set up our `ToDoRepository` that we want to test. We use the DAO from our test `ToDoDatabase` for the first parameter. The second parameter — `this` — in the scope of `runBlockingTest()` is a `TestCoroutineScope` that we can use to manage the work done by our repository.

```
    val results = mutableListOf<List<ToDoModel>>()  
  
    val itemsJob = launch {  
        underTest.items().collect { results.add(it) }  
    }
```

We then need to be able to see what gets put into the repository, to confirm that our changes to that database work. We already have an `items()` function to retrieve the items from the database. That is a Flow, emitting a new database result when we make changes to the database, in addition to emitting an initial result when we make the `items()` call. So, here, we manually `collect()` that flow, piping its results into a `results` object. `results`, therefore, is a List of our query results, with one element in the list per emission from the Flow. We hold onto the Job object created by `launch()`, because we need to `cancel()` that Job before the test completes — otherwise, `runBlockingTest()` will complain.

```
    assertThat(results.size, equalTo(1))  
    assertThat(results[0], empty())
```

We then test to confirm that we got an initial result from our repository, and it shows that we have no entries in the database. `assertThat()`, `equalTo()`, and

TESTING THE REPOSITORY

`empty()` are functions from Hamcrest, a testing library that we have access to via transitive dependencies from our other `androidTestImplementation` dependencies. [Hamcrest](#), apparently named for a wave of cured pork products.

(the author of this book would like to point out that he is not responsible for naming these libraries)

Hamcrest is a large function library of “matchers” that, in the end, can perform some inspections of objects and return a boolean indicating whether the objects matched expectations or not. `assertThat()` uses those matchers, such as `equalTo()` and `empty()`, to examine an object and see if it meets expectations. Here, we are confirming that the `results` list has one element and that that element itself is an empty list.

```
val testModel = ToDoModel("test model")

underTest.save(testModel)

assertThat(results.size, equalTo(2))
assertThat(results[1], contains(testModel))
assertThat(underTest.find(testModel.id).first(), equalTo(testModel))
```

We then:

- Create a test model object
- `save()` that to the repository
- Validate that we got another emission from the Flow, and that it contains our test model object
- Validate that if we use `find()` to retrieve that model object based on its `id` value, that we get the model object back

Remember: `ToDoModel` is a data class. As a result, equality is based on the properties. We are not literally getting `testModel` back from Room — we are getting an equivalent model object, containing the same data.

```
itemsJob.cancel()
```

Finally, we `cancel()` that Job that we set up, to make `runBlockingTest()` happy.

If you have a device or emulator set up, and you run `ToDoRepositoryTest`, you will see that `canAddItems()` succeeds.

Step #5: Writing and Running More Tests

That test function tests our ability to `save()` an item to an empty repository. Now, let's test some more scenarios.

Add these two test functions to `ToDoRepositoryTest`:

```
@Test
fun canModifyItems() = runBlockingTest {
    val underTest = ToDoRepository(db.todoStore(), this)
    val testModel = ToDoModel("test model")
    val replacement = testModel.copy(notes = "This is the replacement")
    val results = mutableListOf<List<ToDoModel>>()

    val itemsJob = launch {
        underTest.items().collect { results.add(it) }
    }

    assertThat(results[0], empty())

    underTest.save(testModel)

    assertThat(results[1], contains(testModel))

    underTest.save(replacement)

    assertThat(results[2], contains(replacement))

    itemsJob.cancel()
}

@Test
fun canRemoveItems() = runBlockingTest {
    val underTest = ToDoRepository(db.todoStore(), this)
    val testModel = ToDoModel("test model")
    val results = mutableListOf<List<ToDoModel>>()

    val itemsJob = launch {
        underTest.items().collect { results.add(it) }
    }

    assertThat(results[0], empty())

    underTest.save(testModel)

    assertThat(results[1], contains(testModel))
```

TESTING THE REPOSITORY

```
underTest.delete(testModel)

assertThat(results[2], empty())

itemsJob.cancel()
}
```

These use all the same techniques that the first test function did. The `can modify items()` test function confirms that if we `save()` a modified version of our model, that the repository is updated with that modification. `can remove items()` confirms that if we `delete()` a model that was saved earlier, that the model is removed from the repository.

If you run all the test functions for `ToDoRepositoryTest`, they should all succeed.

There are lots of other tests that we could write:

- What happens if you try removing a model that is not in the repository?
- What happens if you try saving a second model?
- What happens if you change other properties of the model, besides notes?

However, for the purposes of showing how to test our repository, these three test functions will be enough.

Final Results

Our updated `app/build.gradle` should resemble:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'androidx.navigation.safeargs.kotlin'
    id 'kotlin-kapt'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
    }
}
```

TESTING THE REPOSITORY

```
versionName "1.0"

    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

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

buildFeatures {
    viewBinding true
}

compileOptions {
    coreLibraryDesugaringEnabled true
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}

packagingOptions {
    exclude 'META-INF/AL2.0'
    exclude 'META-INF/LGPL2.1'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation 'com.google.android.material:material:1.4.0'
    implementation "io.insert-koin:koin-android:$koin_version"
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    kapt "androidx.room:room-compiler:$room_version"
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
    testImplementation 'junit:junit:4.13.2'
    testImplementation "org.mockito:mockito-inline:3.12.1"
```

TESTING THE REPOSITORY

```
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"
testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
}
```

(from [T25-RepoTests/ToDo/app/build.gradle](#))

And our new `ToDoRepositoryTest` should contain:

```
package com.commonsware.todo.repo

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import kotlinx.coroutines.test.runBlockingTest
import org.hamcrest.Matchers.empty
import org.hamcrest.Matchers.equalTo
import org.hamcrest.collection.IsIterableContainingInOrder.contains
import org.hamcrest.MatcherAssert.assertThat
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class ToDoRepositoryTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    private val context = InstrumentationRegistry.getInstrumentation().targetContext
    private val db = ToDoDatabase.newTestInstance(context)

    @Test
    fun canAddItems() = runBlockingTest {
        val underTest = ToDoRepository(db.todoStore(), this)
        val results = mutableListOf<List<ToDoModel>>()

        val itemsJob = launch {
            underTest.items().collect { results.add(it) }
        }

        assertThat(results.size, equalTo(1))
        assertThat(results[0], empty())
    }
}
```

TESTING THE REPOSITORY

```
val testModel = ToDoModel("test model")

underTest.save(testModel)

assertThat(results.size, equalTo(2))
assertThat(results[1], contains(testModel))
assertThat(underTest.find(testModel.id).first(), equalTo(testModel))

    itemsJob.cancel()
}

@Test
fun canModifyItems() = runBlockingTest {
    val underTest = ToDoRepository(db.todoStore(), this)
    val testModel = ToDoModel("test model")
    val replacement = testModel.copy(notes = "This is the replacement")
    val results = mutableListOf<List<ToDoModel>>()

    val itemsJob = launch {
        underTest.items().collect { results.add(it) }
    }

    assertThat(results[0], empty())

    underTest.save(testModel)

    assertThat(results[1], contains(testModel))

    underTest.save(replacement)

    assertThat(results[2], contains(replacement))

    itemsJob.cancel()
}

@Test
fun canRemoveItems() = runBlockingTest {
    val underTest = ToDoRepository(db.todoStore(), this)
    val testModel = ToDoModel("test model")
    val results = mutableListOf<List<ToDoModel>>()

    val itemsJob = launch {
        underTest.items().collect { results.add(it) }
    }

    assertThat(results[0], empty())
}
```

TESTING THE REPOSITORY

```
underTest.save(testModel)

assertThat(results[1], contains(testModel))

underTest.delete(testModel)

assertThat(results[2], empty())

itemsJob.cancel()
}
```

(from [T25-RepoTests/ToDo/app/src/androidTest/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/build.gradle](#)
- [app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#)

Testing a UI

To test a UI, we need to be able to set up that UI, perform some actions, and see what the results are. That will involve messing with our widgets.

For that widget manipulation, the Jetpack solution is Espresso. This provides a succinct (albeit strange) API for accessing widgets, checking their states, and performing actions on them (like clicks).

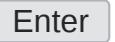
In this tutorial, we will write an Espresso test to test the RecyclerView created by RosterListFragment.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding a New Test Class

If we are going to create tests for RosterListFragment, we should create a RosterListFragmentTest. And, since we do not have a ui.roster sub-package in androidTest, we will need to add that.

In the androidTest source set, right click over the com.commonsware.todo.repo package and choose “New” > “Package” from the context menu. Fill in com.commonsware.todo.ui.roster for the name, then click “OK” to make this sub-package.

Then, right-click over the new com.commonsware.todo.ui.roster package and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in RosterListFragmentTest and choose “Class” as the kind. Press  or

[Return](#) to create the class, giving you:

```
package com.commonsware.todo.repo

class RosterListFragmentTest {
```

Step #2: Initializing Our Repository

Next, replace the current implementation of `RosterListFragmentTest` with this:

```
package com.commonsware.todo.ui.roster

import androidx.test.platform.app.InstrumentationRegistry
import com.commonsware.todo.repo.ToDoDatabase
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutinesCoroutineScope
import kotlinx.coroutinesSupervisorJob
import kotlinx.coroutines.runBlocking
import org.junit.Before
import org.koin.core.context.loadKoinModules
import org.koin.dsl.module

class RosterListFragmentTest {
    private lateinit var repo: ToDoRepository
    private val items = listOf(
        ToDoModel("this is a test"),
        ToDoModel("this is another test"),
        ToDoModel("this is... wait for it... yet another test")
    )

    @Before
    fun setUp() {
        val context = InstrumentationRegistry.getInstrumentation().targetContext
        val db = ToDoDatabase.newTestInstance(context)
        val appScope = CoroutineScope(SupervisorJob())

        repo = ToDoRepository(db.todoStore(), appScope)

        loadKoinModules(module {
            single { repo }
        })
    }
}
```

TESTING A UI

```
    runBlocking { items.forEach { repo.save(it) } }
}
```

As with `ToDoRepositoryTest`, we are creating our own `ToDoRepository` instance. That way, we can have a fresh one for each test run, and we do not need to worry about the results of a previous test affecting the next test. However, there is one small problem: the activity and fragments know nothing about this test repository. They will want to use the one supplied by Koin.

So, via `loadKoinModules()`, we *replace* the repository that Koin normally would return with a fresh instance. `loadKoinModules()` works in conjunction with the `single()` to replace the true singleton `ToDoRepository` with this replacement instance.

We then populate our test repository with three model objects, using `save()` on the repository, wrapped in `runBlocking()` to have that work happen on the current thread.

Step #3: Testing Our List

Now, add this test function to `RosterListFragmentTest`:

```
@Test
fun testListContents() {
    ActivityScenario.launch(MainActivity::class.java)

    onView(withId(R.id.items)).check(matches(hasChildCount(3)))
}
```

While containing only two lines of code (not counting several `import` statements), quite a bit is done here.

`ActivityScenario.launch()` will start up our `MainActivity`, which in turn will display our `RosterListFragment`. `launch()` will not return until our UI is up and ready for testing. `ActivityScenario` comes from the `androidx.test.ext:junit` library that we added.

The other line is a fairly typical Espresso statement. Espresso uses a lot of imported functions to try to keep the code terse.

An Espresso statement usually takes one of two forms:

TESTING A UI

- `onView().check()`, to see if a widget is in a particular state
- `onView().perform()`, to perform some action on a widget, such as clicking it

Here, we have a statement that is of the first form, where we want to `check()` the state of a widget and confirm that it meets our expectations.

`onView()` is the Espresso way of looking up widgets in the current activity's view hierarchy. It takes a `ViewMatcher` as a parameter, where that `ViewMatcher` encodes some rule(s) for what widget we want to access. The `withId()` function creates a `ViewMatcher` that finds a view by its ID, in this case `R.id.items`. So, `onView(withId(R.id.items))` looks up our `RecyclerView` and returns... a `ViewInteraction`.

One thing that you can do with a `ViewInteraction` is to call `check()` on it. `check()` takes a `ViewAssertion` as a parameter. A `ViewAssertion` works a bit like the Kluent assertions that we used in the unit tests, in that it checks our view and will fail the test if the view does not match expectations.

The most common way of getting a `ViewAssertion` is to call the `matches()` function. This returns a `ViewAssertion` wrapped around a Hamcrest Matcher.

`hasChildCount()` is a `ViewMatcher`, which is a `Matcher` that knows how to “match” some view property against some expected value. `hasChildCount()` looks at the number of child widgets of a `ViewGroup` and compares it against the expected value. In this case, we are expecting that our `RecyclerView` has three rows, because we put three model objects into our test repository. `hasChildCount(3)` will return `true` if the `RecyclerView` has three rows, `false` otherwise. So, overall, `check(matches(hasChildCount(3)))` will fail the test if the `RecyclerView` has anything other than three rows.

If you run the test, the test succeeds. Moreover, if you run the test, you will actually see the activity flash onto the screen for a brief moment, as `ActivityScenario.launch()` displays our `MainActivity`. This is one of the reasons why instrumented tests are slow: we often are doing a lot of setup work, such as launching an activity.

This is obviously a very limited test of the UI. Unfortunately, Espresso gets very complex very quickly. Trying to do more — such as clicking on a `CheckBox` to confirm the repository is updated — will get to be more complex than is suitable for a tutorial.

TESTING A UI

Note: you might wonder why the function name is `testListContents()` and not something like `test list contents()`. The backticks style of writing function names works in unit tests but not in instrumented tests, due to some Android limitations.

Final Results

`RosterListFragmentTest` should look like:

```
package com.commonsware.todo.ui.roster

import androidx.test.core.app.ActivityScenario
import androidx.test.espresso.onView
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.hasChildCount
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.platform.app.InstrumentationRegistry
import com.commonsware.todo.R
import com.commonsware.todo.repo.ToDoDatabase
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.ui.MainActivity
import kotlinx.coroutinesCoroutineScope
import kotlinx.coroutinesSupervisorJob
import kotlinx.coroutines.runBlocking
import org.junit.Before
import org.junit.Test
import org.koin.core.context.loadKoinModules
import org.koin.dsl.module

class RosterListFragmentTest {
    private lateinit var repo: ToDoRepository
    private val items = listOf(
        ToDoModel("this is a test"),
        ToDoModel("this is another test"),
        ToDoModel("this is... wait for it... yet another test")
    )

    @Before
    fun setUp() {
        val context = InstrumentationRegistry.getInstrumentation().targetContext
        val db = ToDoDatabase.newTestInstance(context)
        val appScope = CoroutineScope(SupervisorJob())

        repo = ToDoRepository(db.todoStore(), appScope)
    }
}
```

TESTING A UI

```
loadKoinModules(module {
    single { repo }
})

runBlocking { items.forEach { repo.save(it) } }

@Test
fun testListContents() {
    ActivityScenario.launch(MainActivity::class.java)

    onView(withId(R.id.items)).check(matches(hasChildCount(3)))
}
}
```

(from [T26-Espresso/ToDo/app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#)

Part Four: Adding Additional Features

Tracking Our Load Status

There are three logical states that our `RosterListFragment` and its `RecyclerView` can be in:

- We have to-do items, and we are displaying them
- We do not have to-do items, because the user has not entered any, and so we should show the “empty” view to help guide the user
- We do not know whether we have to-do items or not, because we have not yet loaded them from the database

That third state is not being handled by the app. Instead, we treat “do not know” as being the same as “we do not have to-do items” — we show the “empty” view if our `RosterListAdapter` is empty, no matter *why* it is empty. Plus, it would be nice to show some sort of “loading” indicator while the data load is in progress... such as a `ProgressBar`.

So, in this tutorial, we will fix this. For most Android devices, and for shorter to-do lists, the difference will not be visible, as the data will load very rapidly. However, on slower devices, or with large to-do lists, the difference may be noticeable.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adjusting Our Layout

We need to make a couple of changes to the layout used by `RosterListFragment`.

Open `res/layout/todo_roster.xml` in the IDE. In the design view, click on the

TRACKING OUR LOAD STATUS

empty TextView in the “Component Tree”. In the list of all attributes, find the visibility attribute, and set it to gone:

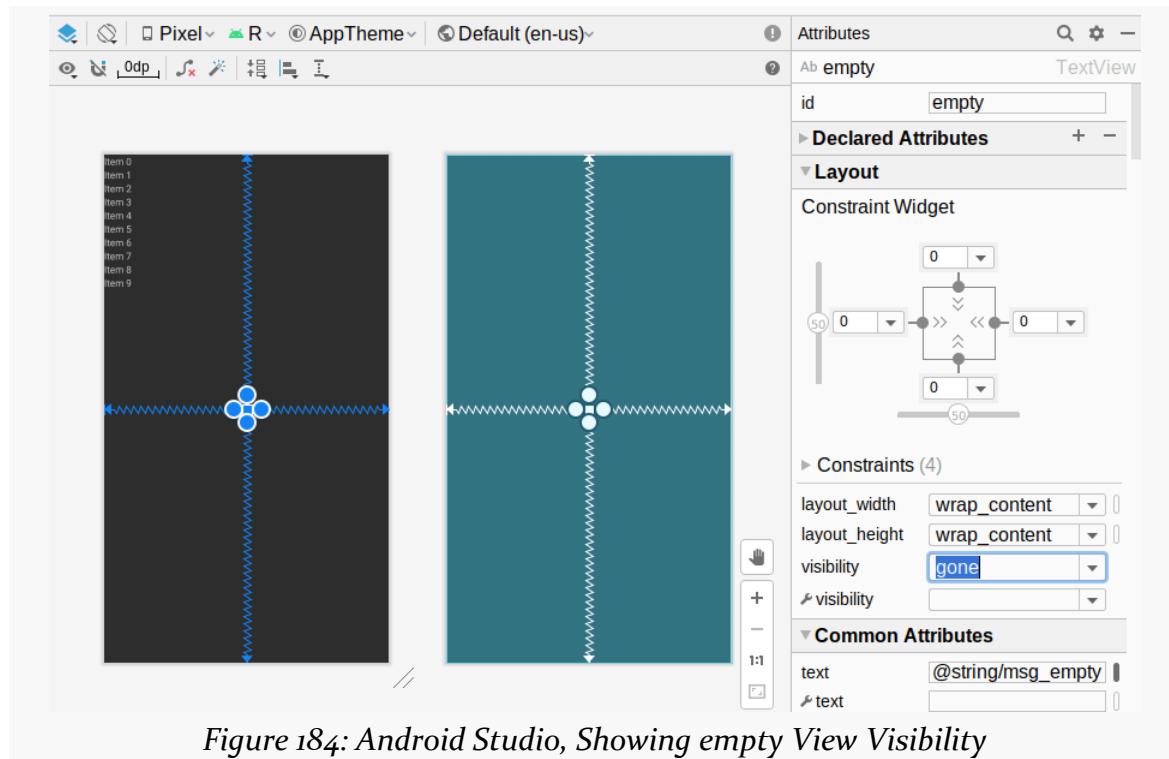


Figure 184: Android Studio, Showing empty View Visibility

TRACKING OUR LOAD STATUS

Next, choose the “Widgets” category in the “Palette” view. You will see two labeled “ProgressBar”, one with a circle and one that is an actual bar:

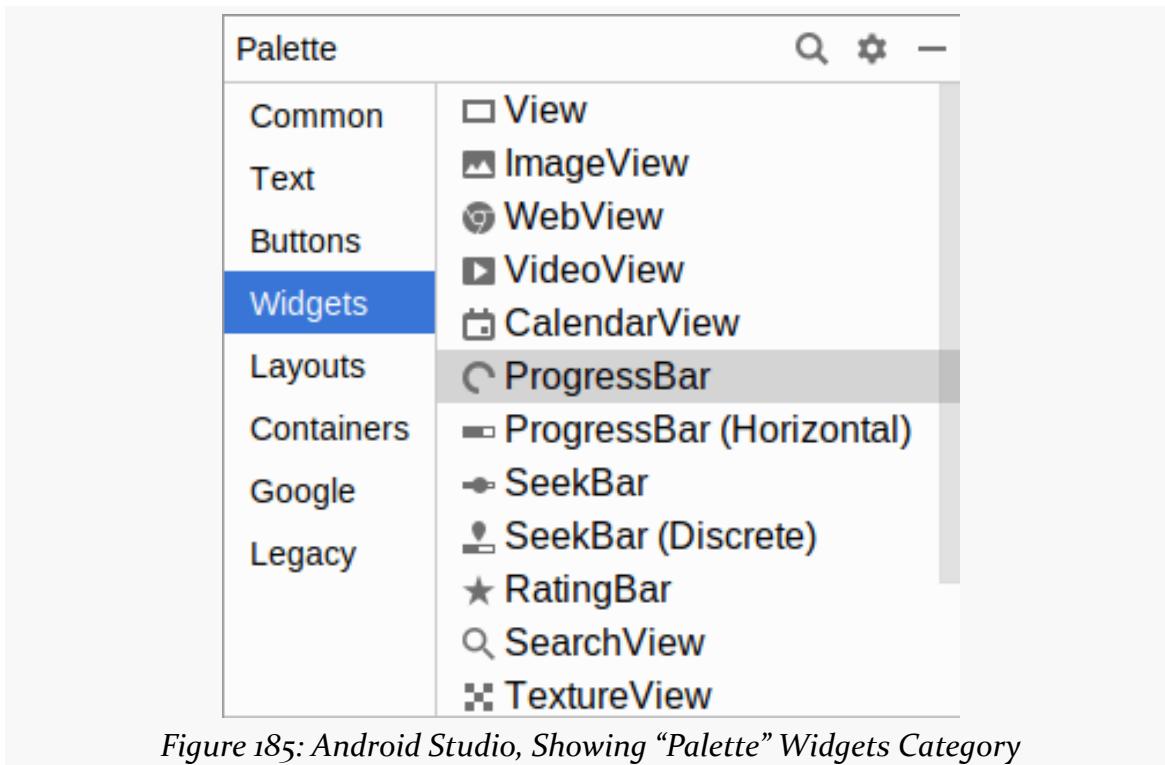


Figure 185: Android Studio, Showing “Palette” Widgets Category

Typically, the circular ProgressBar is used for indefinite progress, where we do not know how long the work will take. The horizontal ProgressBar is more often used for cases where we can let the user know how far we have progressed.

In this case, the work is fairly atomic: either our data is loaded or it is not. We have no intermediate steps with which to provide progress updates, so we should use the circular indefinite ProgressBar.

However, we cannot drag and drop a widget into the preview area, since the preview is mostly our RecyclerView. The IDE will attempt to make our widget be a child of the RecyclerView, and that does not work very well. Instead, drag the circular ProgressBar from the “Palette” and drop it on the ConstraintLayout entry in the “Component Tree” view. This will add it as a child to the ConstraintLayout, which is what we want.

Then, use the grab handles on the ProgressBar to set up constraints to all four edges of the ConstraintLayout. However, there is a decent chance that you will

TRACKING OUR LOAD STATUS

sometimes get a constraint that ties the ProgressBar to the items RecyclerView, instead of to the parent ConstraintLayout:

```
<ProgressBar  
    android:id="@+id/progressBar"  
    style="?android:attr/progressBarStyle"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:layout_constraintBottom_toBottomOf="@+id/items"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />
```

Here, app:layout_constraintBottom_toBottomOf wound up being set to @+id/items instead of parent. The simplest thing to do is to change the XML manually, so that all four constraints are set to parent.

Also, change the widget's ID to loading.

Step #2: Reporting our Loaded Status

Right now, RosterListFragment cannot detect when we are loading data. Because of the rules of StateFlow, we have to provide an initial value, which has an empty list of to-do items. RosterListFragment has no good way to distinguish that from the case where we have loaded the data and the database is empty. So, we need to do something to clarify “empty but not yet loaded” from “empty after loading”.

To that end, add an `isLoading` property to `RosterViewState`:

```
data class RosterViewState(  
    val items: List<ToDoModel> = listOf(),  
    val isLoading: Boolean = false  
)
```

(from [T27-Load/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

We will use `false` to indicate that we are loading the data and `true` to indicate that the data is loaded. Since we default it to `false`, we need to update our `map()` call in the `states` definition of `RosterMotor` to pass in `true`:

```
val states = repo.items()  
.map { RosterViewState(it, true) }  
.stateIn(viewModelScope, SharingStarted.Eagerly, RosterViewState())
```

TRACKING OUR LOAD STATUS

(from [T27-Load/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Now, `isLoading` will be `false` for the initial state and `true` once our data is loaded.

Step #3: Reacting to the Loaded Status

Right now, we have the loading widget set as `VISIBLE` and the empty widget set as `GONE`. We already have code to display the empty widget when that is appropriate. We need to add in some smarts to hide the loading widget at the same time.

So, change the observer in the `onViewCreated()` function of `RosterListFragment` to be:

```
viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = if (state.isLoading) View.GONE else View.VISIBLE

            when {
                state.items.isEmpty() && state.isLoading -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }
}
```

(from [T27-Load/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Then, remove this line from `onViewCreated()`:

```
binding?.empty?.visibility = View.GONE
```

If you run the app, you will not see any differences, most likely. Loading a few to-do items — if any — from the database will be fairly quick. And, we have no good way to tell Room to pretend to be slow.

If you would like to see the `ProgressBar`, you could delay the response from `ToDoRepository` to the `items()` call, such as:

TRACKING OUR LOAD STATUS

```
fun items(): Flow<List<ToDoModel>> =  
    store.all().map { all -> all.map { it.toModel() } }  
    .onStart { delay(5000) }
```

`onStart()` tells the Flow to do some work when we start observing the flow. Here, we introduce a five-second delay, using `delay(5000)`.

If you make that change, then run the app, you will see the `ProgressBar` for five seconds, after which it vanishes and is replaced by the items list or the empty state. After seeing the effect, remove the `.onStart { delay(5000) }`, so you do not have to wait the extra time in the remaining tutorials.

Final Results

Your `todo_roster` layout resource 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=".ui.MainActivity">  
  
<ProgressBar  
    android:id="@+id/loading"  
    style="?android:attr/progressBarStyle"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    app:layout_constraintBottom_toBottomOf="parent"  
    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:text="@string/msg_empty"  
    android:textAppearance="?android:attr/textAppearanceMedium"  
    android:visibility="gone"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintLeft_toLeftOf="parent"  
    app:layout_constraintRight_toRightOf="parent"
```

TRACKING OUR LOAD STATUS

```
    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 [T27-Load/ToDo/app/src/main/res/layout/todo_roster.xml](#))

The modified RosterMotor should look like:

```
package com.commonsware.todo.ui.roster

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn
import kotlinx.coroutines.launch

data class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val isLoaded: Boolean = false
)

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    val states = repo.items()
        .map { RosterViewState(it, true) }
        .stateIn(viewModelScope, SharingStarted.Eagerly, RosterViewState())

    fun save(model: ToDoModel) {
        viewModelScope.launch {
            repo.save(model)
        }
    }
}
```

(from [T27-Load/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

And the tweaked RosterListFragment should look like:

TRACKING OUR LOAD STATUS

```
package com.commonsware.todo.ui.roster

import android.os.Bundle
import android.view.*
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoRosterBinding
import com.commonsware.todo.repo.ToDoModel
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private var binding: TodoRosterBinding? = null

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root

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

        val adapter = RosterAdapter(
            layoutInflater,
            onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
            onRowClick = ::display
        )

        binding?.items?.apply {
            setAdapter(adapter)
            layoutManager = LinearLayoutManager(context)

            addItemDecoration(
                DividerItemDecoration(

```

TRACKING OUR LOAD STATUS

```
        activity,
        DividerItemDecoration.VERTICAL
    )
)
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = if (state.isLoaded) View.GONE else View.VISIBLE

            when {
                state.items.isEmpty() && state.isLoaded -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
    }

    return super.onOptionsItemSelected(item)
}
```

TRACKING OUR LOAD STATUS

```
private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}
}
```

(from [T27-Load/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/res/layout/todo_roster.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)

Filtering Our Items

It is entirely possible that a user of this app will have a lot of to-do items. Rather than force the user to have to scroll through all of them in the list, we could offer some options for working with a subset of those items. In this tutorial, we will add a “filter” feature, to allow the user to work with either the outstanding to-do items, the completed items, or all of the items.

In reality, given the scope of this app, we could do all of our filtering in the `RosterListFragment`, or perhaps in the `RosterMotor`. This is a book sample, and you are not likely to create lots and lots of to-do items.

In theory, though, there *could* be lots and lots of to-do items. Or, we could have a more complex data model. Or, we could have to call out to a server to do some sort of search, rather than just filtering some subset of model objects already in memory.

So, in this tutorial, we will pretend that we really do need to request a new roster of items from our repository when the user elects to filter (or stop filtering) the list of items. That makes things a bit more complex but a bit more realistic.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding a Query

Let’s work “back to front”, updating our `ToDoEntity.Store` first, before we start changing `ToDoRepository`, `RosterMotor`, etc.

In terms of the filtering, we need to have a way of asking the `ToDoEntity.Store` to

FILTERING OUR ITEMS

give us the items that match our filter criterion: is the to-do item completed or not.

To that end, add this `filtered()` function to `ToDoEntity.Store`:

```
@Query("SELECT * FROM todos WHERE isCompleted = :isCompleted ORDER BY description")
fun filtered(isCompleted: Boolean): Flow<List<ToDoEntity>>
    (from T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt)
```

`filtered()` takes a Boolean parameter, indicating if we want the completed or outstanding to-do items. We use that in the `WHERE` clause by putting `:isCompleted` where we want the value to show up. `filtered()` otherwise works like `all()`, returning our items via a `Flow`.

Step #2: Defining a FilterMode

From the standpoint of the UI, we have three possible filter conditions:

- We want to show the completed to-do items
- We want to show the outstanding to-do items (i.e., the ones not yet completed)
- We want to show all items, regardless of completion status

That is beyond a simple Boolean value, but we can model that via an `enum class`.

In the `ToDoRepository.kt` source file, add this `enum class` before the `ToDoRepository` definition:

```
enum class FilterMode { ALL, OUTSTANDING, COMPLETED }
    (from T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt)
```

Step #3: Consuming a FilterMode

Now, we can update `ToDoRepository` to help us get at a filtered edition of the to-do items.

First, we need to map from the `FilterMode` enum to the functions and parameters that we need for `ToDoEntity.Store`. To handle that, add this function to `ToDoRepository`:

FILTERING OUR ITEMS

```
private fun filteredEntities(filterMode: FilterMode) = when (filterMode) {
    FilterMode.ALL -> store.all()
    FilterMode.OUTSTANDING -> store.filtered(isCompleted = false)
    FilterMode.COMPLETED -> store.filtered(isCompleted = true)
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

Here, we just use Kotlin’s “exhaustive when” to handle the three `FilterMode` cases, calling the appropriate function on `ToDoEntity.Store` for each.

Then, replace the `items` function in `ToDoRepository` with this implementation:

```
fun items(filterMode: FilterMode = FilterMode.ALL): Flow<List<ToDoModel>> =
    filteredEntities(filterMode).map { all -> all.map { it.toModel() } }
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

Here, we use the new `filteredEntities()` function to get the `Flow` of entities. We have `items()` use a default value for its `filterMode` parameter, so a call to `items()` with no parameters will retrieve the unfiltered list (`FilterMode.ALL`).

Step #4: Augmenting Our Motor

`RosterMotor` now needs to offer a way for the `RosterListFragment` to request a particular `FilterMode`.

First, add a `filterMode` property to `RosterViewState`:

```
data class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val isLoading: Boolean = false,
    val filterMode: FilterMode = FilterMode.ALL
)
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This will allow us to keep track of the currently-active filter mode, with an initial state of `ALL`.

Then, replace the current `RosterMotor` implementation with:

```
class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    private val _states = MutableStateFlow(RosterViewState())
    val states = _states.asStateFlow()
```

FILTERING OUR ITEMS

```
private var job: Job? = null

init {
    load(FilterMode.ALL)
}

fun load(filterMode: FilterMode) {
    job?.cancel()

    job = viewModelScope.launch {
        repo.items(filterMode).collect {
            _states.emit(RosterViewState(it, true, filterMode))
        }
    }
}

fun save(model: ToDoModel) {
    viewModelScope.launch {
        repo.save(model)
    }
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

The `save()` function towards the bottom is unchanged from what we had before. The rest is quite different.

Before, `states` was very simple:

```
val states = repo.items()
    .map { RosterViewState(it) }
    .stateIn(viewModelScope, SharingStarted.Eagerly, RosterViewState())
```

That is because we always loaded all of the to-do items. We still could have kept this code, but it would not give our UI the ability to change the filter mode, which is what we are trying to achieve.

However, if we later call `repo.items(FilterMode.COMPLETED)` or `repo.items(FilterMode.OUTSTANDING)`, we get a *different* Flow than the one we had originally. That highlights a limitation of `stateIn()`: it can only give us *one* StateFlow. In our case, we may have several, as the user toggles between various filter options.

Moreover, it will simplify our `RosterListFragment` if there always is one StateFlow

FILTERING OUR ITEMS

supplying the RosterViewState objects, rather than having to know to subscribe to different StateFlow objects at different times for different reasons. So, we have one or more Flow objects with our items, and we want to funnel them all into a single StateFlow of RosterViewState objects.

One solution for that is MutableStateFlow.

MutableStateFlow is a StateFlow that we manage ourselves. We supply an initial state in the constructor, then call `emit()` whenever our state changes.

So, what RosterMotor is doing is using a MutableStateFlow as the stable StateFlow that RosterListFragment observes.

With all that in mind...

- `_states` is our MutableStateFlow, and it is private
- Since we do not want RosterListFragment to know the implementation details, `states` is a plain StateFlow property that happens to point to the MutableStateFlow in `_states`
- `load()` will observe a call to `items()` on the repo, supplying whatever the requested FilterMode is, and `emit()` any changes as new RosterViewState objects through `_states`
- `init {}` triggers our initial `load()` call, to retrieve ALL items

However, we also track a Job object, representing our current Flow collection. On each `load()` call, we `cancel()` the preceding Job (if there was one), then save the `launch()` result as the next Job.

What happens if we fail to do this? Each `items()` call keeps getting collected, piling up if we call `load()` multiple times:

- We call `load()` first when the RosterMotor is created, via the `init {}` block, and we start observing a Flow from `items()`.
- Later, when we add logic to RosterListFragment to switch filters, we will call `load()` again for a new filter. So, we start observing a Flow from `items()`. However, if we did not cancel the Job from the first `load()` call, *that Flow will continue to be collected*.
- The user goes and adds another to-do item, and both Flow objects report the revised database contents, because Room does not know that these Flow objects are, in effect, duplicates. So, we `emit()` two RosterViewState objects, one triggered by each Flow.

FILTERING OUR ITEMS

- And, if the user toggles the filter mode several times, we might pile up several outstanding Flow objects.

So, we track the Job from the last Flow collection and `cancel()` it before observing the next Flow.

And, if you run the app, it should work as it did before, showing you all of the to-do items in the list.

Step #5: Adding a Checkable Submenu

We have added quite a few app bar items in these tutorials. This time, we need to add one to allow the user to filter the list of items. To do that, we will use an app bar item that has a submenu of radio buttons, so the user can toggle between the different filter modes.

But, first, we need another icon.

FILTERING OUR ITEMS

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Icon” button and search for `filter`:

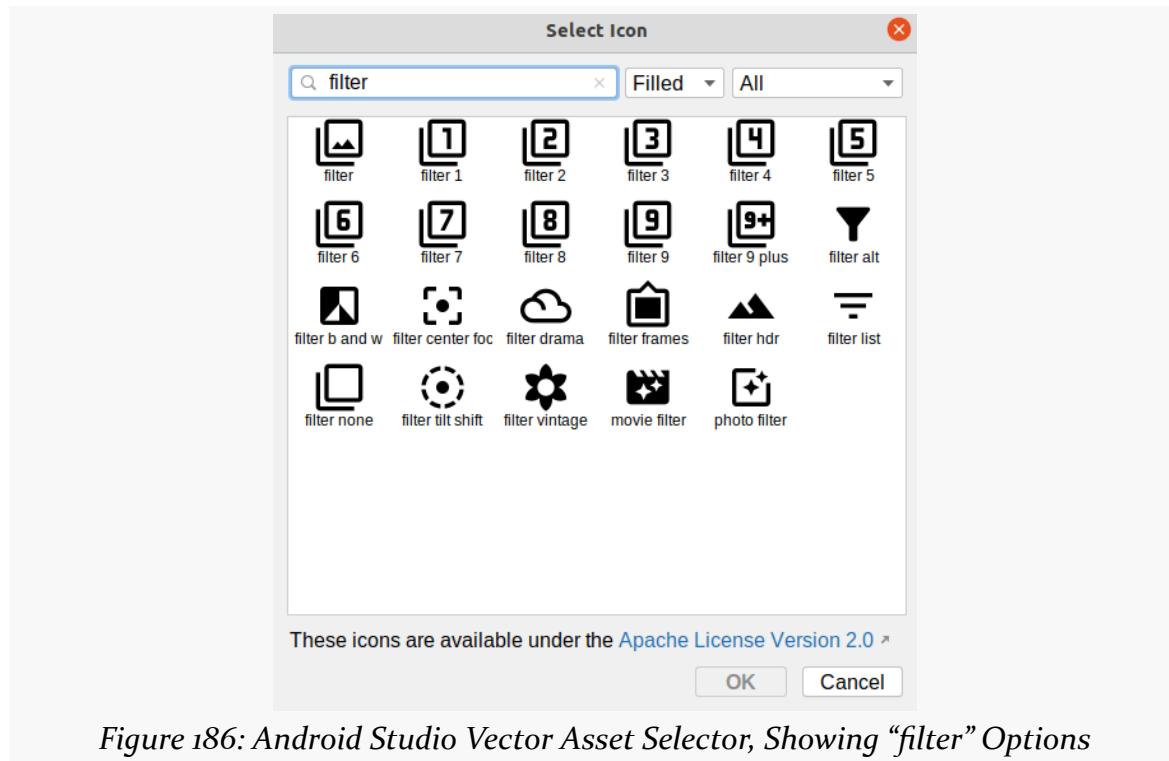


Figure 186: Android Studio Vector Asset Selector, Showing “filter” Options

Choose the “filter list” icon and click “OK” to close up the icon selector. Change the icon’s name to `ic_filter`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

FILTERING OUR ITEMS

Next, open up the `res/menu/actions_roster.xml` resource file, and switch to the graphical designer. Drag a “Menu Item” from the “Palette” view into the Component Tree, slotting it before the existing “add” item:

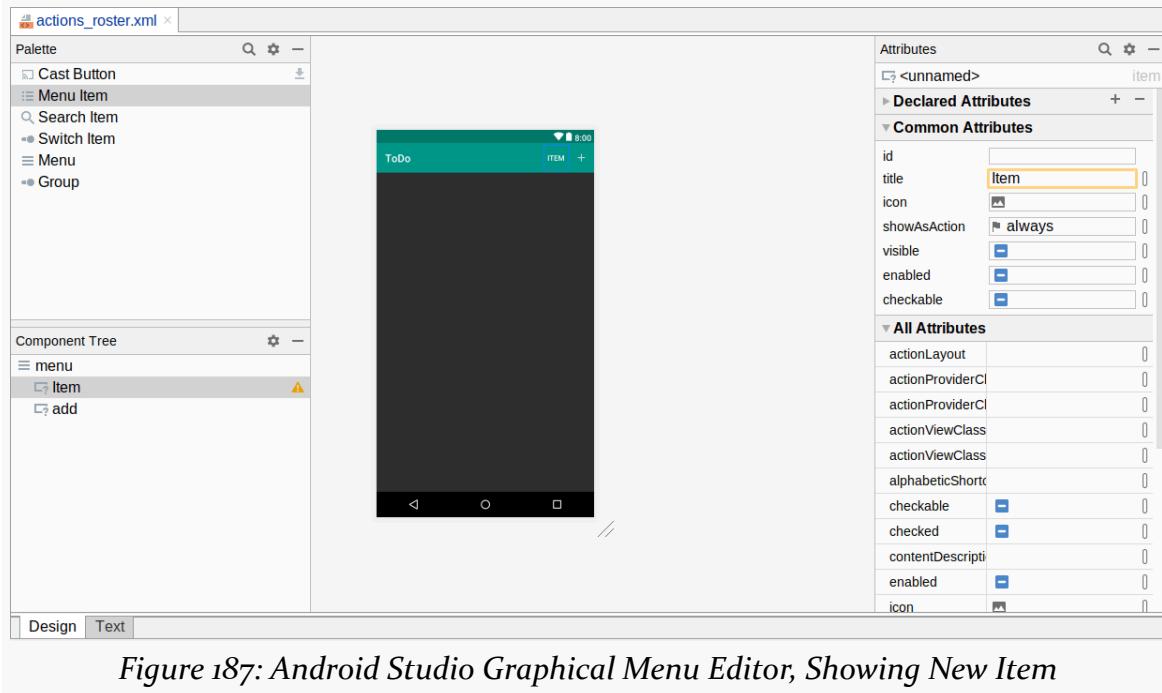


Figure 187: Android Studio Graphical Menu Editor, Showing New Item

In the Attributes view for this new item, assign it an ID of `filter`. Then, choose both “ifRoom” and “withText” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Click on `ic_filter` in the list of drawables, then click OK to accept that choice of icon.

FILTERING OUR ITEMS

Then, click the “O” 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_filter` as the resource name and “Filter” as the resource value. Click OK to close the dialog and complete the configuration of this app bar item:

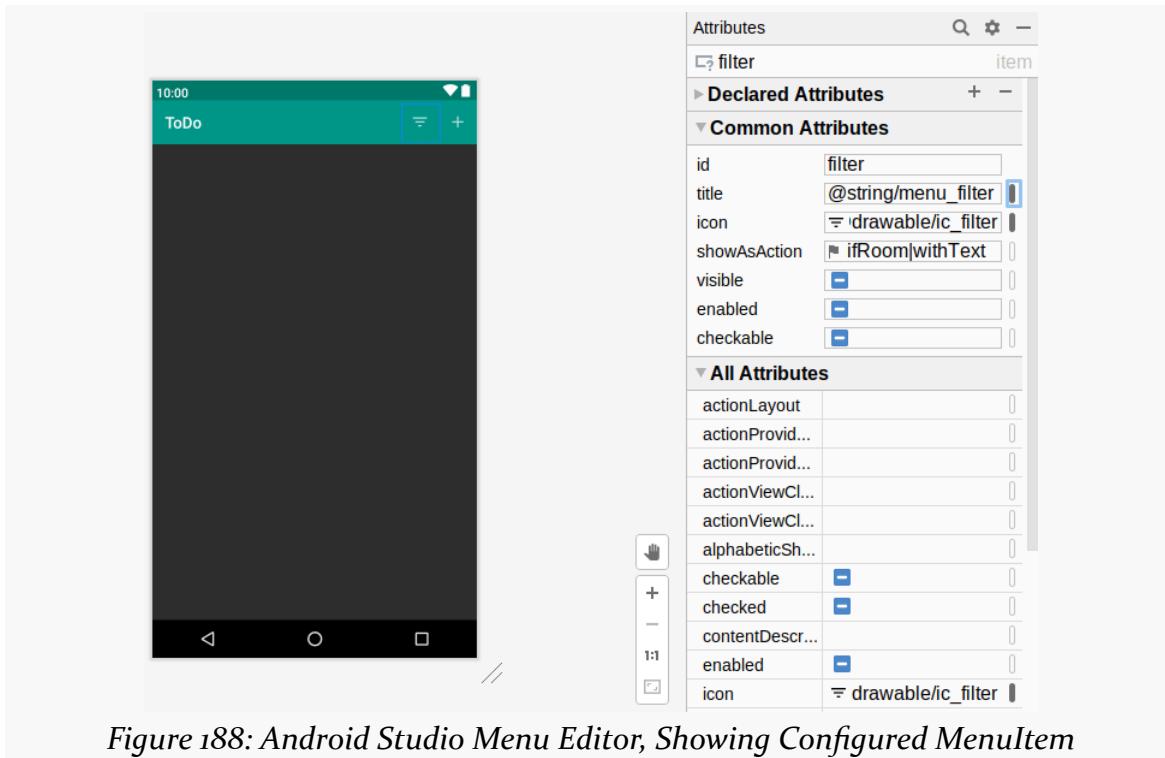


Figure 188: Android Studio Menu Editor, Showing Configured MenuItem

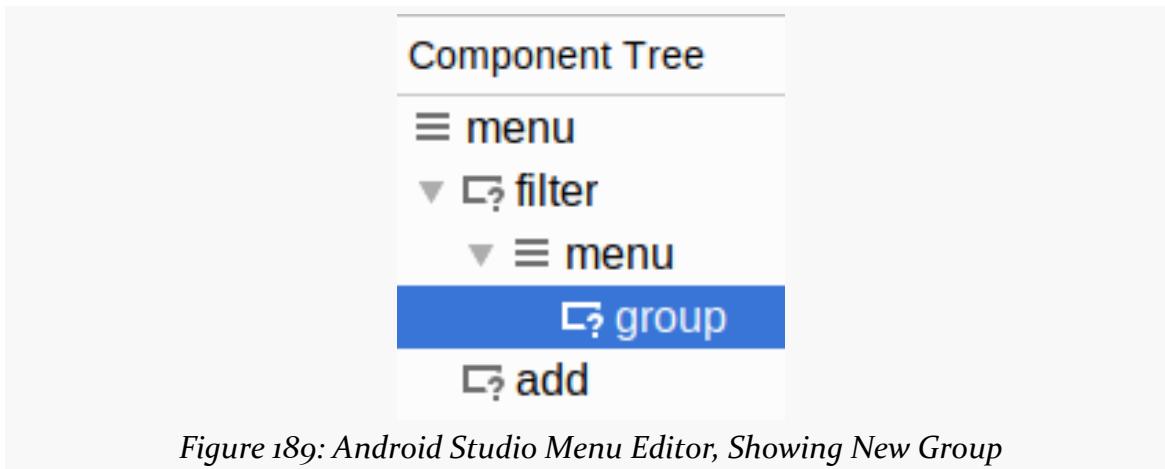
Unfortunately, at this point, Android Studio bugs crop up yet again, and we cannot readily add a checkable submenu to this item via drag-and-drop. So, switch to the “Code” view and add an empty `<menu>` element as a child of the `filter` `<item>` element:

```
<item
    android:id="@+id/filter"
    android:icon="@drawable/ic_filter"
    android:title="@string/menu_filter"
    app:showAsAction="ifRoom|withText">
    <menu />
</item>
```

Then, switch back to the “Design” view. From the Palette, drag a “Group” into the

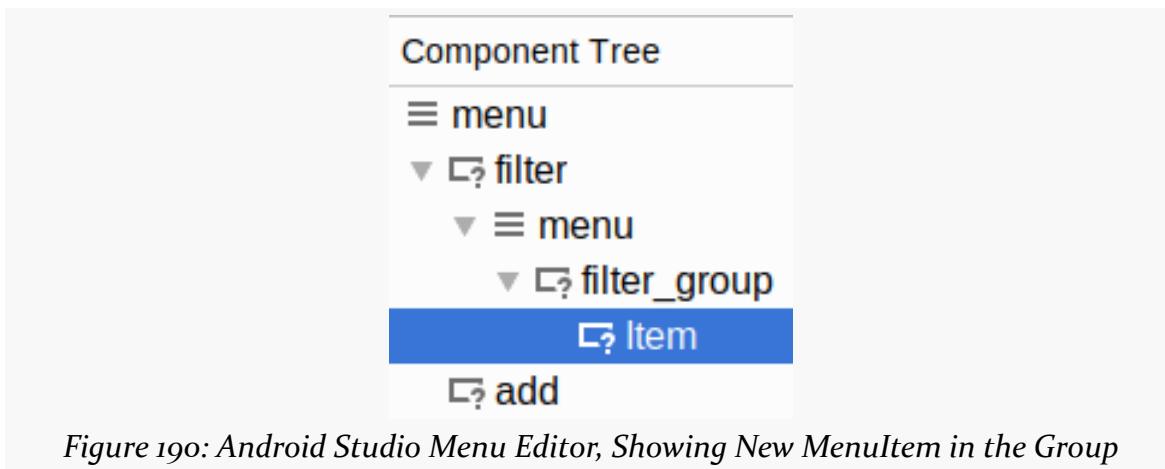
FILTERING OUR ITEMS

new “menu” in the Component Tree:



In the Attributes pane, give the group an ID of `filter_group` and set the “checkableBehavior” to “single”.

Then, from the Palette, drag a “MenuItem” into the new group in the Component Tree:



Drag two more “MenuItem” entries from the “Palette” and drop them in the group in the Component Tree, to give you a total of three items in the group.

Select the first of the three submenu items in the Component Tree. In the Attributes pane, give it an ID of `a11`. In the “All Attributes” section, check the “checked” checkbox, so that it contains a checkmark. Then, click the “O” button next to the

FILTERING OUR ITEMS

“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_filter_all` as the resource name and “All” as the resource value. Click OK to close the dialog and complete the configuration of this submenu item.

Select the second submenu item in the Component Tree. In the Attributes pane, give it an ID of `completed`. Then, for the “title”, use the “O” button to assign it a new string resource, named `menu_filter_completed`, with a value of “Completed”.

Select the third submenu item in the Component Tree. In the Attributes pane, give it an ID of `outstanding`. Then, for the “title”, use the “O” button to assign it a new string resource, named `menu_filter_outstanding`, with a value of “Outstanding”.

If you run your app, you should see the new filter app bar item. Clicking it will expose the submenu, although clicking on the submenu items will have no effect.

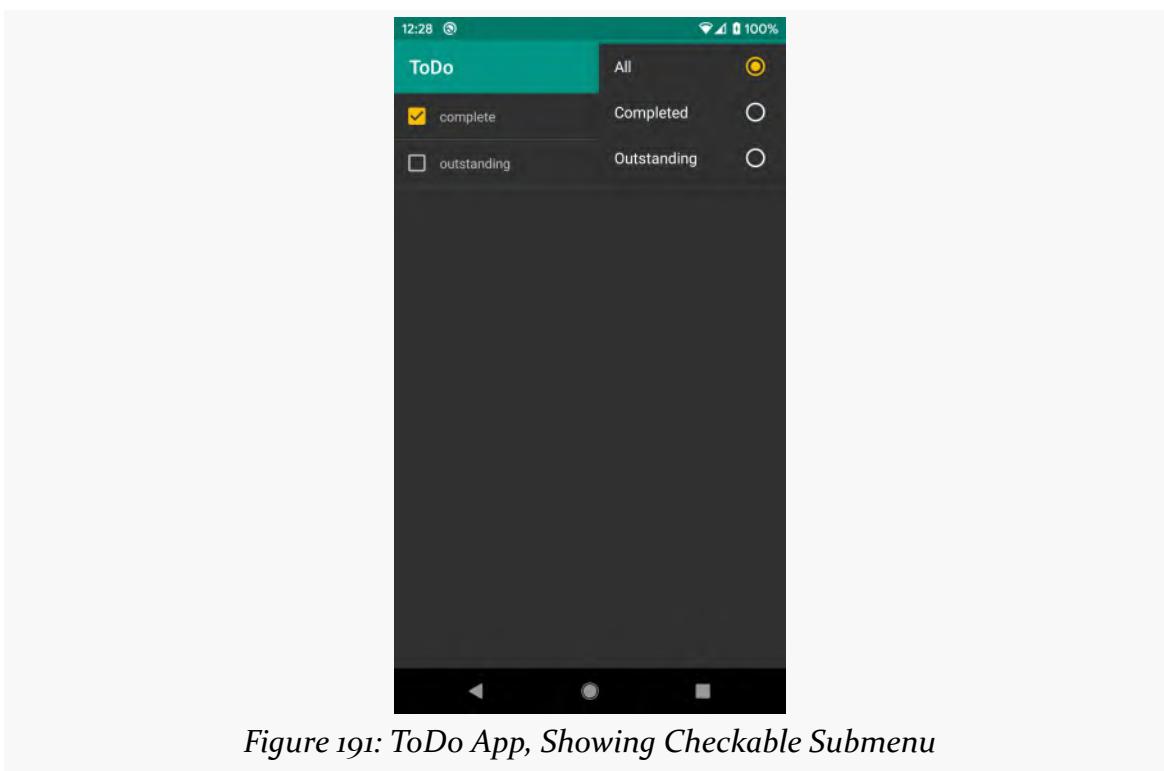


Figure 191: ToDo App, Showing Checkable Submenu

Step #6: Getting Control on Filter Choices

In particular, clicking on the submenu items does not even change their checked

FILTERING OUR ITEMS

state. Even though our submenu looks like a group of radio buttons, it does not behave like one automatically. Instead, we need to add some code for that. Plus, we really ought to consider actually doing the filtering.

In `RosterListFragment`, replace the current `onOptionsItemSelected()` function with:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
        R.id.all -> {
            item.isChecked = true
            motor.load(FilterMode.ALL)
            return true
        }
        R.id.completed -> {
            item.isChecked = true
            motor.load(FilterMode.COMPLETED)
            return true
        }
        R.id.outstanding -> {
            item.isChecked = true
            motor.load(FilterMode.OUTSTANDING)
            return true
        }
    }

    return super.onOptionsItemSelected(item)
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

`onOptionsItemSelected()` will get called when the user clicks on the checkable submenu items, so we add cases to our `when` for those three menu items. For each, we mark it as checked, so the radio button associated with that “checkable” menu item becomes checked (and others as unchecked). Plus, we call `load()` on our `RosterMotor` with the appropriate `FilterMode` for that menu item.

At this point, if you run the app, and you have some to-do items, the filtering should work:

- “All” will show all of the items

FILTERING OUR ITEMS

- “Completed” will show those where the `isCompleted` checkbox is checked
- “Outstanding” will show those where the `isCompleted` checkbox is unchecked

However, there are a couple of minor UI glitches that we still need to fix, which we will handle in the remaining steps of this tutorial.

Step #7: Fixing the Empty Text

At this point, there are two situations when we have an empty list:

1. If there are no to-do items at all
2. If there are no to-do items in the current filter mode (e.g., all of the items are outstanding, and the filter mode is set to COMPLETED)

This is going to be confusing to the user — the user might not realize that the reason their list is empty is that all of the relevant to-do items have been removed from the list via the filter.

We should improve this.

First, go into `res/values/strings.xml` and add a new string resource:

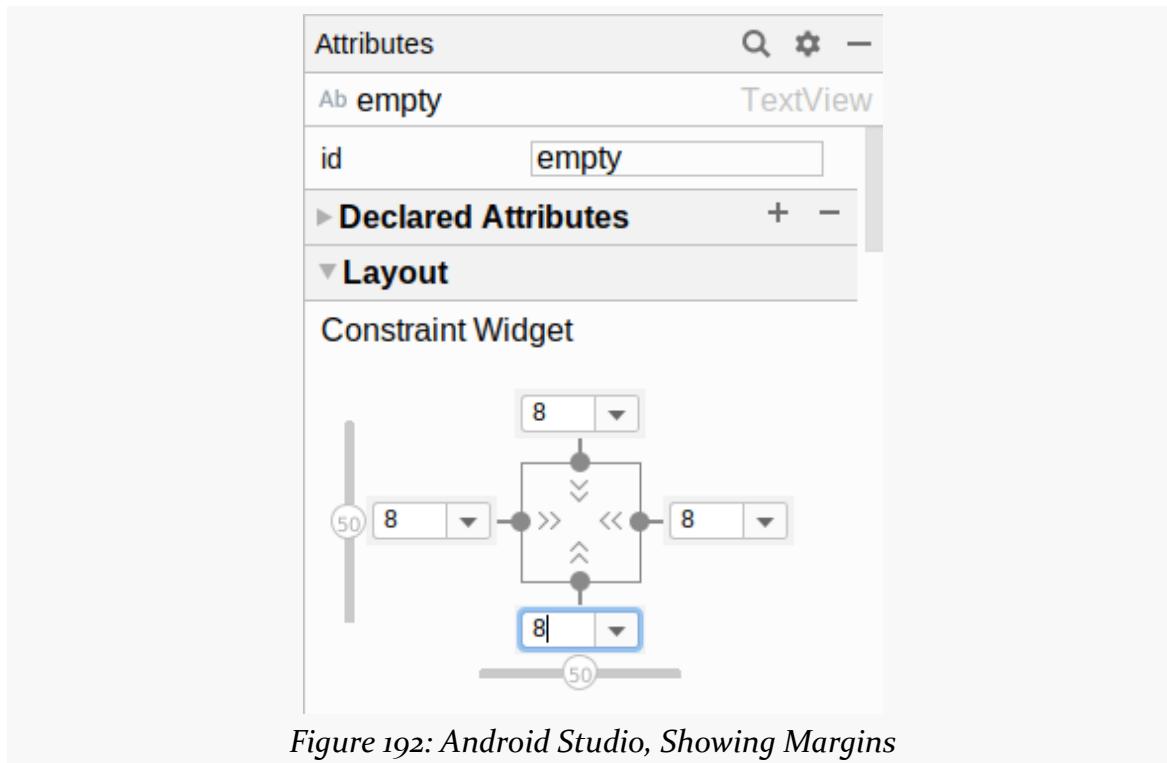
```
<string name="msg_empty_filtered">Click the + icon to add a todo item, or change your filter to show other items</string>
```

(from [T28-Filter/ToDo/app/src/main/res/values/strings.xml](#))

(note: this will be shown in the book as split across multiple lines, but you are welcome to have it be all on one line in your project, if you wish)

FILTERING OUR ITEMS

Next, open `res/layout/todo_roster.xml` in the IDE. Click on our empty `TextView`. In the “Layout” section of the “Attributes” pane, give the widget 8dp of margin on all four sides, so it does not run all the way to the edges of the screen:



FILTERING OUR ITEMS

Then, open the “gravity” options:

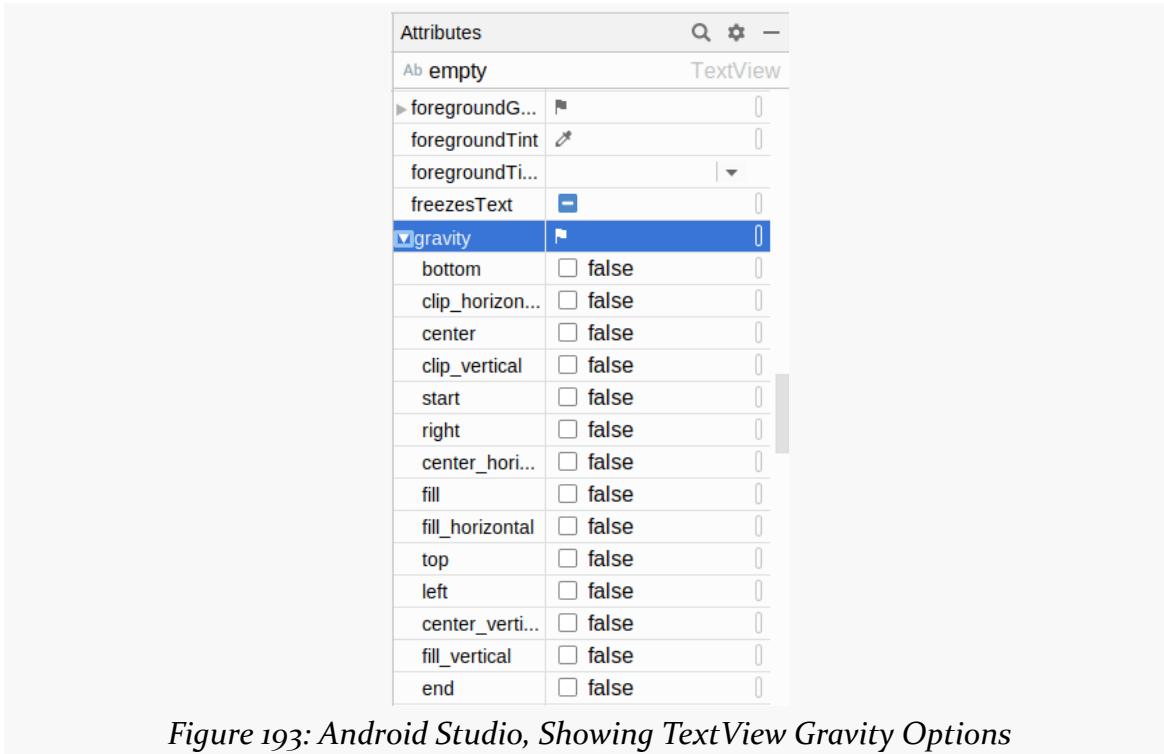


Figure 193: Android Studio, Showing TextView Gravity Options

Check the “center” option, which will cause our text to be centered within the space being occupied by the TextView.

Then, in RosterListFragment, update the motor.states observer in onViewCreated() to be:

```
viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = View.GONE

            when {
                state.items.isEmpty() && state.filterMode == FileMode.ALL -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                }
            }
        }
    }
}
```

FILTERING OUR ITEMS

```
        empty.setText(R.string.msg_empty_filtered)
    }
    else -> empty.visibility = View.GONE
}
}
}
}
```

Here, we use a when block to handle the three cases:

- Showing the original empty message if we have no items and have a `FilterMode` of ALL
- Showing the new empty message if we have no items and have some other `FilterMode`
- Removing the empty message if we have items to show

Now, if you run the app, you will see the empty message centered, and you will see the new empty message if you have items but they are all hidden by the filter:



Figure 194: ToDo App, Showing Revised Empty Message

Step #8: Addressing the Menu Problem

We have one more glitch to fix.

If you filter the list, then click on a to-do item to view its details, then click BACK, you will find that the list is filtered, but the menu checked state is back to having the “All” option checked. That is because the UI of the RosterListFragment was rebuilt, and our menu reverted to its default state.

What we need to do is to have the menu reflect the current RosterViewState filterMode value. This is a bit annoying to implement:

- We cannot easily access a menu item at an arbitrary point in time, so we need to hold onto the menu items when we set up the menu
- We need to be able to get the right menu item for the current FilterMode
- We need to handle this work both when the menu is created *and* when the state gets updated, as there is no guaranteed order of when those two things happen

To handle all of this, first add a menuMap property to RosterListFragment:

```
private val menuMap = mutableMapOf<FilterMode, MenuItem>()  
(from T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt)
```

Then, modify the onCreateOptionsMenu() function in RosterListFragment to be:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {  
    inflater.inflate(R.menu.actions_roster, menu)  
  
    menuMap.apply {  
        put(FilterMode.ALL, menu.findItem(R.id.all))  
        put(FilterMode.COMPLETED, menu.findItem(R.id.completed))  
        put(FilterMode.OUTSTANDING, menu.findItem(R.id.outstanding))  
    }  
  
    menuMap[motor.states.value.filterMode]?.isChecked = true  
  
    super.onCreateOptionsMenu(menu, inflater)  
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Here, we:

FILTERING OUR ITEMS

- Populate the menuMap to map each FilterMode value to its corresponding MenuItem
- See if we have a RosterViewState, and if we do, mark the MenuItem for the current FilterMode as checked

To find out the current value of the StateFlow, we can just reference value. That will either be the initial value or whatever the last emitted RosterViewState was.

Then, add this line to the bottom of the RosterViewState observer that we set up in onViewCreated():

```
menuMap[state.filterMode]?.isChecked = true
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This ensures that when we get a new RosterViewState that the appropriate MenuItem is checked.

Now, if you run the app, you should see that the filtering applied to the list matches the checked MenuItem, even after some navigation.

Final Results

At this point, ToDoEntity should look like:

```
package com.commonsware.todo.repo

import androidx.room.*
import kotlinx.coroutines.flow.Flow
import java.time.Instant
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Instant = Instant.now(),
    val isCompleted: Boolean = false
) {
    constructor(model: ToDoModel) : this(
        id = model.id,
        description = model.description,
```

FILTERING OUR ITEMS

```
    isCompleted = model.isCompleted,
    notes = model.notes,
    createdOn = model.createdOn
)

fun toModel(): ToDoModel {
    return ToDoModel(
        id = id,
        description = description,
        isCompleted = isCompleted,
        notes = notes,
        createdOn = createdOn
    )
}

@Dao
interface Store {
    @Query("SELECT * FROM todos ORDER BY description")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE isCompleted = :isCompleted ORDER BY
description")
    fun filtered(isCompleted: Boolean): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String?): Flow<ToDoEntity?>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(vararg entities: ToDoEntity)

    @Delete
    suspend fun delete(vararg entities: ToDoEntity)
}
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

ToDoRepository.kt should resemble:

```
package com.commonsware.todo.repo

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Flow
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.withContext

enum class FilterMode { ALL, OUTSTANDING, COMPLETED }
```

FILTERING OUR ITEMS

```
class ToDoRepository(
    private val store: ToDoEntity.Store,
    private val appScope: CoroutineScope
) {
    fun items(filterMode: FilterMode = FilterMode.ALL): Flow<List<ToDoModel>> =
        filteredEntities(filterMode).map { all -> all.map { it.toModel() } }

    private fun filteredEntities(filterMode: FilterMode) = when (filterMode) {
        FilterMode.ALL -> store.all()
        FilterMode.OUTSTANDING -> store.filtered(isCompleted = false)
        FilterMode.COMPLETED -> store.filtered(isCompleted = true)
    }

    fun find(id: String?): Flow<ToDoModel?> = store.find(id).map { it?.toModel() }

    suspend fun save(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.save(ToDoEntity(model))
        }
    }

    suspend fun delete(model: ToDoModel) {
        withContext(appScope.coroutineContext) {
            store.delete(ToDoEntity(model))
        }
    }
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

RosterMotor.kt should now be:

```
package com.commonsware.todo.ui.roster

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.Job
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.launch

data class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val isLoading: Boolean = false,
```

FILTERING OUR ITEMS

```
val filterMode: FilterMode = FilterMode.ALL
)

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    private val _states = MutableStateFlow(RosterViewState())
    val states = _states.asStateFlow()
    private var job: Job? = null

    init {
        load(FilterMode.ALL)
    }

    fun load(filterMode: FilterMode) {
        job?.cancel()

        job = viewModelScope.launch {
            repo.items(filterMode).collect {
                _states.emit(RosterViewState(it, true, filterMode))
            }
        }
    }

    fun save(model: ToDoModel) {
        viewModelScope.launch {
            repo.save(model)
        }
    }
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

The `actions_roster` menu resource XML should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/filter"
        android:icon="@drawable/ic_filter"
        android:title="@string/menu_filter"
        app:showAsAction="ifRoom|withText">
        <menu>

            <group
                android:id="@+id/filter_group"
                android:checkableBehavior="single" >
                <item
```

FILTERING OUR ITEMS

```
    android:id="@+id/all"
    android:checked="true"
    android:title="@string/menu_filter_all" />
<item
    android:id="@+id/completed"
    android:title="@string/menu_filter_completed" />
<item
    android:id="@+id/outstanding"
    android:title="@string/menu_filter_outstanding" />
</group>
</menu>
</item>
<item
    android:id="@+id/add"
    android:icon="@drawable/ic_add"
    android:title="@string/menu_add"
    app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T28-Filter/ToDo/app/src/main/res/menu/actions_roster.xml](#))

The strings resource XML should resemble:

```
<resources>
    <string name="app_name">ToDo</string>
    <string name="msg_empty">Click the + icon to add a todo item!</string>
    <string name="msg_empty_filtered">Click the + icon to add a todo item, or change
your filter to show other items</string>
    <string name="menu_about">About</string>
    <string name="is_completed">Item is completed</string>
    <string name="created_on">Created on:</string>
    <string name="menu_edit">Edit</string>
    <string name="desc">Description</string>
    <string name="notes">Notes</string>
    <string name="menu_save">Save</string>
    <string name="menu_add">Add</string>
    <string name="menu_delete">Delete</string>
    <string name="menu_filter">Filter</string>
    <string name="menu_filter_all">All</string>
    <string name="menu_filter_completed">Completed</string>
    <string name="menu_filter_outstanding">Outstanding</string>
</resources>
```

(from [T28-Filter/ToDo/app/src/main/res/values/strings.xml](#))

RosterListFragment now should look like:

```
package com.commonsware.todo.ui.roster
```

FILTERING OUR ITEMS

```
import android.os.Bundle
import android.view.*
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoRosterBinding
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private val menuMap = mutableMapOf<FilterMode, MenuItem>()
    private var binding: TodoRosterBinding? = null

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root

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

        val adapter = RosterAdapter(
            layoutInflater,
            onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
            onRowClick = ::display
        )

        binding?.items?.apply {
            setAdapter(adapter)
            layoutManager = LinearLayoutManager(context)

            addItemDecoration(

```

FILTERING OUR ITEMS

```
        DividerItemDecoration(
            activity,
            DividerItemDecoration.VERTICAL
        )
    )
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = if (state.isLoaded) View.GONE else View.VISIBLE

            when {
                state.items.isEmpty() && state.filterMode == FilterMode.ALL -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty_filtered)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }

    menuMap[state.filterMode]?.isChecked = true
}
}
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)

    menuMap.apply {
        put(FilterMode.ALL, menu.findItem(R.id.all))
        put(FilterMode.COMPLETED, menu.findItem(R.id.completed))
        put(FilterMode.OUTSTANDING, menu.findItem(R.id.outstanding))
    }

    menuMap[motor.states.value.filterMode]?.isChecked = true
}
```

FILTERING OUR ITEMS

```
super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
        R.id.all -> {
            item.isChecked = true
            motor.load(FilterMode.ALL)
            return true
        }
        R.id.completed -> {
            item.isChecked = true
            motor.load(FilterMode.COMPLETED)
            return true
        }
        R.id.outstanding -> {
            item.isChecked = true
            motor.load(FilterMode.OUTSTANDING)
            return true
        }
    }
}

return super.onOptionsItemSelected(item)
}

private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}
}
```

(from [T28-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

And the todo_roster menu resource 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"
```

FILTERING OUR ITEMS

```
xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".ui.MainActivity">

    <ProgressBar
        android:id="@+id/loading"
        style="?android:attr/progressBarStyle"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        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_marginLeft="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginBottom="8dp"
        android:gravity="center"
        android:text="@string/msg_empty"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:visibility="gone"
        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 [T28-Filter/ToDo/app/src/main/res/layout/todo_roster.xml](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)
- [app/src/main/res/drawable/ic_filter_list_black_24dp.xml](#)
- [app/src/main/res/menu/actions_roster.xml](#)
- [app/src/main/res/values/strings.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)
- [app/src/main/res/layout/todo_roster.xml](#)

Generating a Report

Right now, our to-do information is held in a SQLite database, whose contents are viewable via the app. This is fine, as far as it goes... but it does not go very far. We have no good means of getting this information to any other device or any other person.

In the next two tutorials, we will work on some options for doing just that. In this tutorial, we will generate a simple Web page containing our to-do list, filtered by whatever filter mode we have applied. That Web page will be saved in a location specified by the user, and we will view the Web page in a Web browser when we are done.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding a Save App Bar Item

It's time for another app bar item! This time, though, it uses an existing icon and string resource, as we already have a "save" app bar item in the `EditFragment`. We will reuse that for a "save" app bar item in the `RosterListFragment`.

GENERATING A REPORT

Open up the `res/menu/actions_roster.xml` resource file, and switch to the graphical designer. Drag a “Menu Item” from the “Palette” view into the Component Tree, slotting it after the existing “add” item:

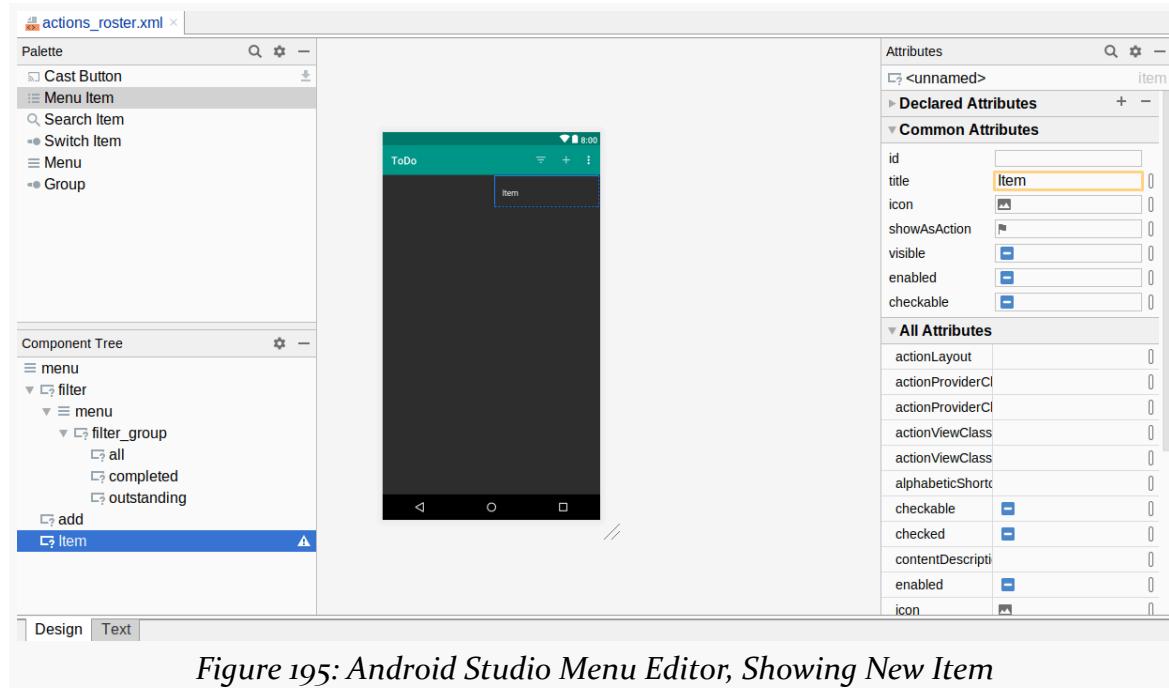


Figure 195: Android Studio Menu Editor, Showing New Item

In the Attributes view for this new item, assign it an ID of `save`. Then, choose both “ifRoom” and “withText” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Click on `ic_save` in the list of drawables, then click OK to accept that choice of icon.

GENERATING A REPORT

Then, click the “O” button next to the “title” field. As before, this brings up a string resource selector. This time, we actually have a `menu_save` string resource in the selector:

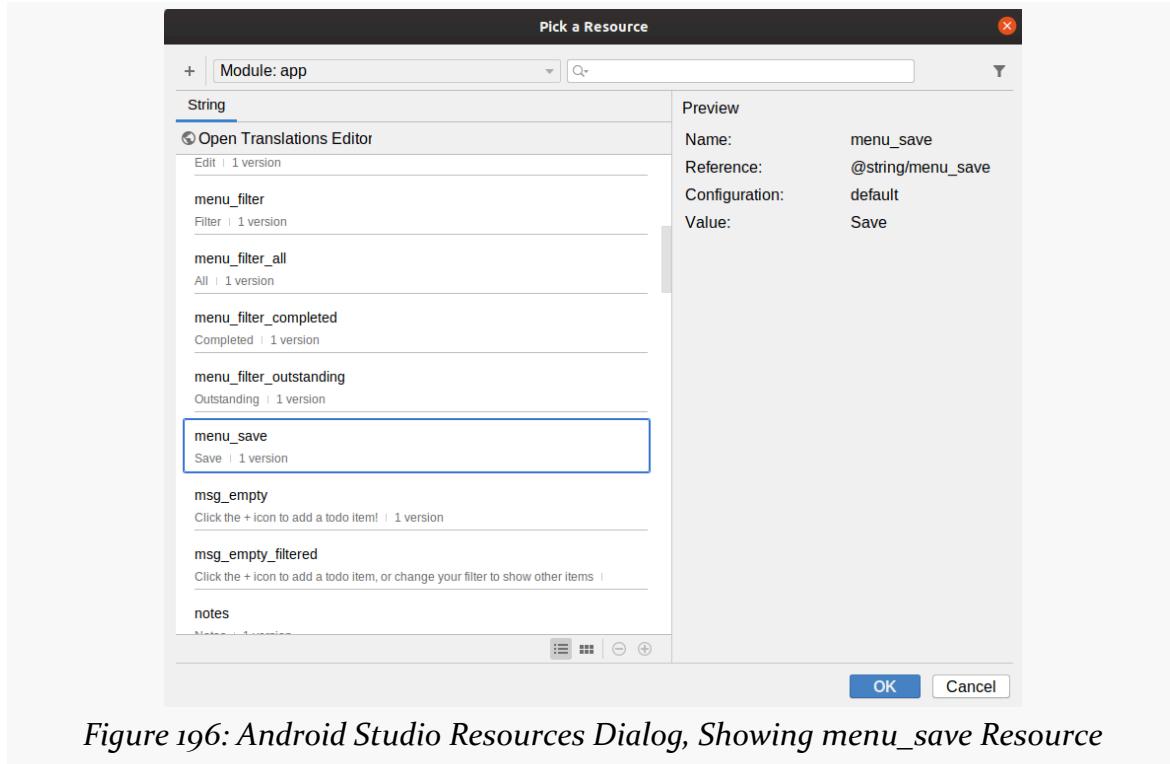


Figure 196: Android Studio Resources Dialog, Showing `menu_save` Resource

Double-click on it to choose it and complete configuration of our app bar item.

Step #2: Making a Save

Now, we need to respond to that action item click by asking the user where we should save the Web page generated from the filtered to-do items. To accomplish that, we are going to use the Storage Access Framework, which provides us with UI akin to the “file open” and “file ‘save as’” dialogs that you see in desktop operating systems.

`ACTION_CREATE_DOCUMENT` is an Intent action that, as the name suggests, guides the user to create a new document for your use. In modern editions of the Jetpack, that Intent is wrapped up in an “activity result” request that will give us a `Uri` pointing to where that document resides.

GENERATING A REPORT

Up in your list of properties for RosterListFragment (before the `onCreate()` function), add this new property:

```
private val createDoc =  
    registerForActivityResult(ActivityResultContracts.CreateDocument()) {  
    }
```

`registerForActivityResult()` tells the Jetpack “hey, we want to register a way to make a request and get a response”. The request is in the form of an `ActivityResultContract` instance, in this case `ActivityResultContracts.CreateDocument`. `CreateDocument` maps to `ACTION_CREATE_DOCUMENT`, and it will request that the user choose a place on their device (or in their cloud storage) to create a new document. We get a `Uri` pointing to that new document in the lambda expression that we provide to `registerForActivityResult()`.

Next, add this `saveReport()` function to RosterListFragment:

```
private fun saveReport() {  
    createDoc.launch("report.html")  
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Here, we call `launch()` on the `createDoc` that we just created. `launch()` says “OK, let’s make an actual request”, and for our case, that will ask the user to pick the place to create the document. We supply `report.html` as the default name to use for this new document, though the user might change that name.

Then, add another branch to the `when` in `onOptionsItemSelected()` in RosterListFragment, to call `saveReport()` if the user clicks the “Save” action bar item:

```
R.id.save -> {  
    saveReport()  
    return true  
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

GENERATING A REPORT

If you run this on your device or emulator and click that “Save” item, you should be presented with a screen where you can choose where to save the content:

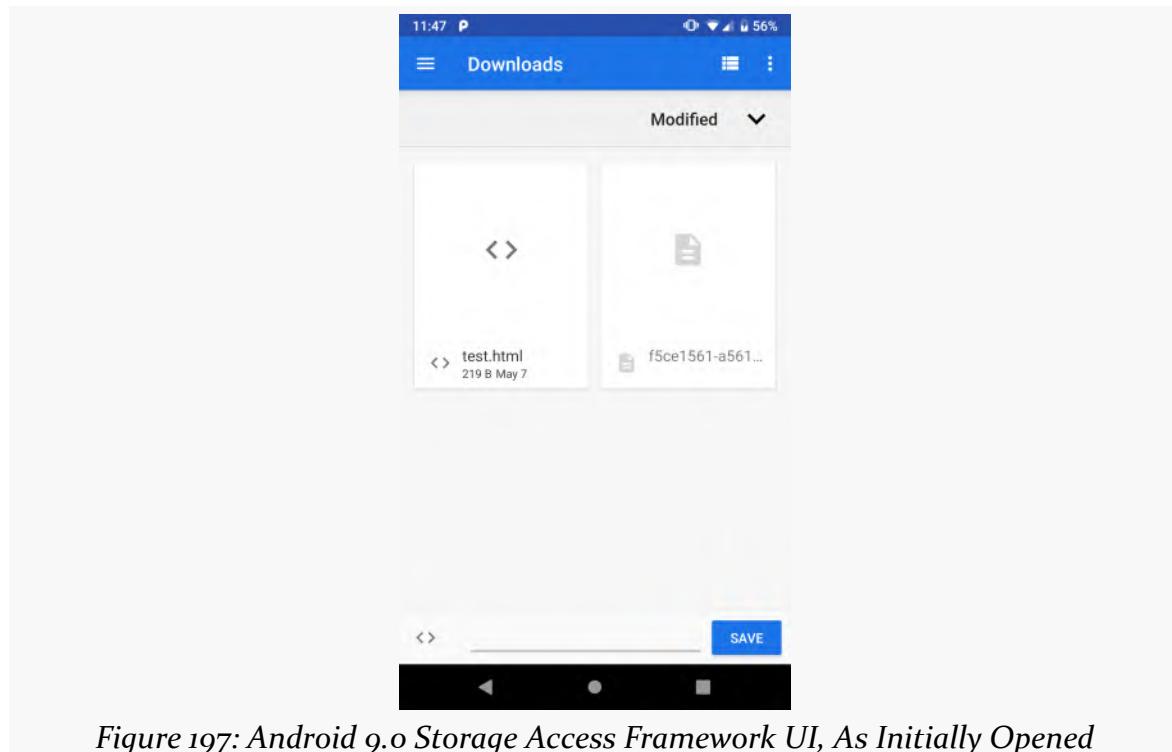


Figure 197: Android 9.0 Storage Access Framework UI, As Initially Opened

GENERATING A REPORT

The user can choose “Show internal storage” from the overflow menu to add more options of where to save the content:

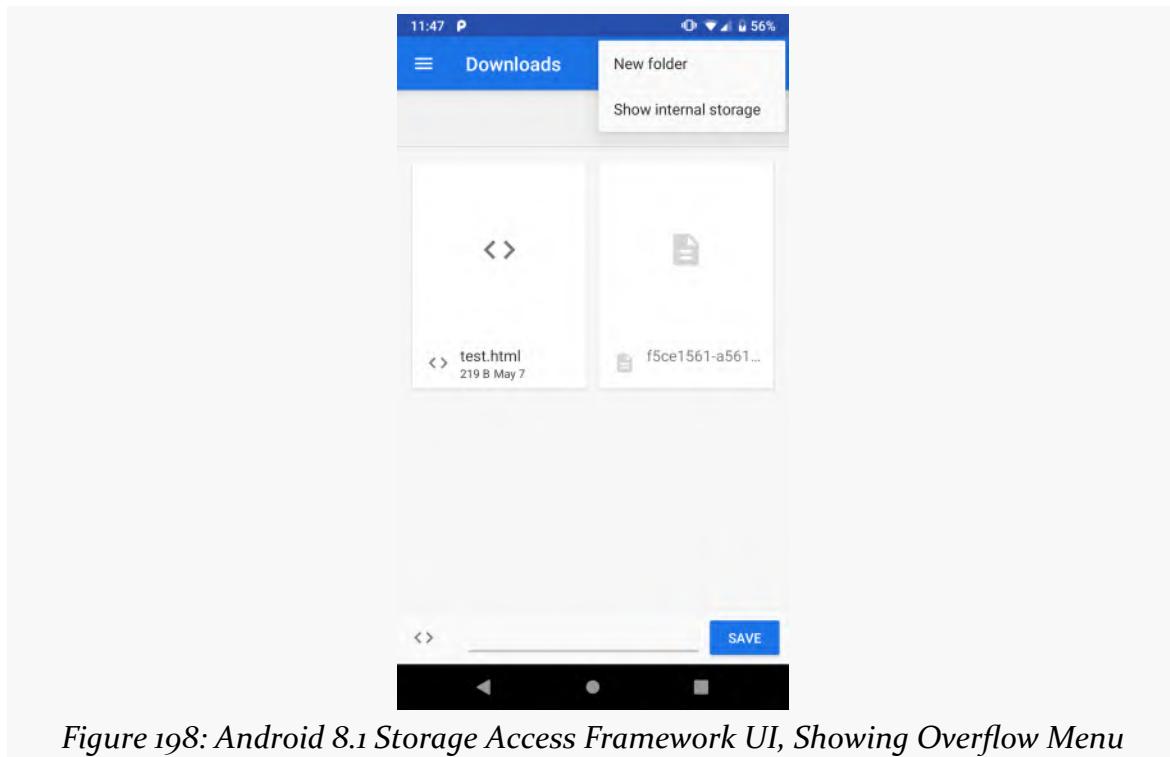


Figure 198: Android 8.1 Storage Access Framework UI, Showing Overflow Menu

GENERATING A REPORT

If your device happened to show the “Save” item in the overflow, you might have noticed that “About” appeared before “Save”:

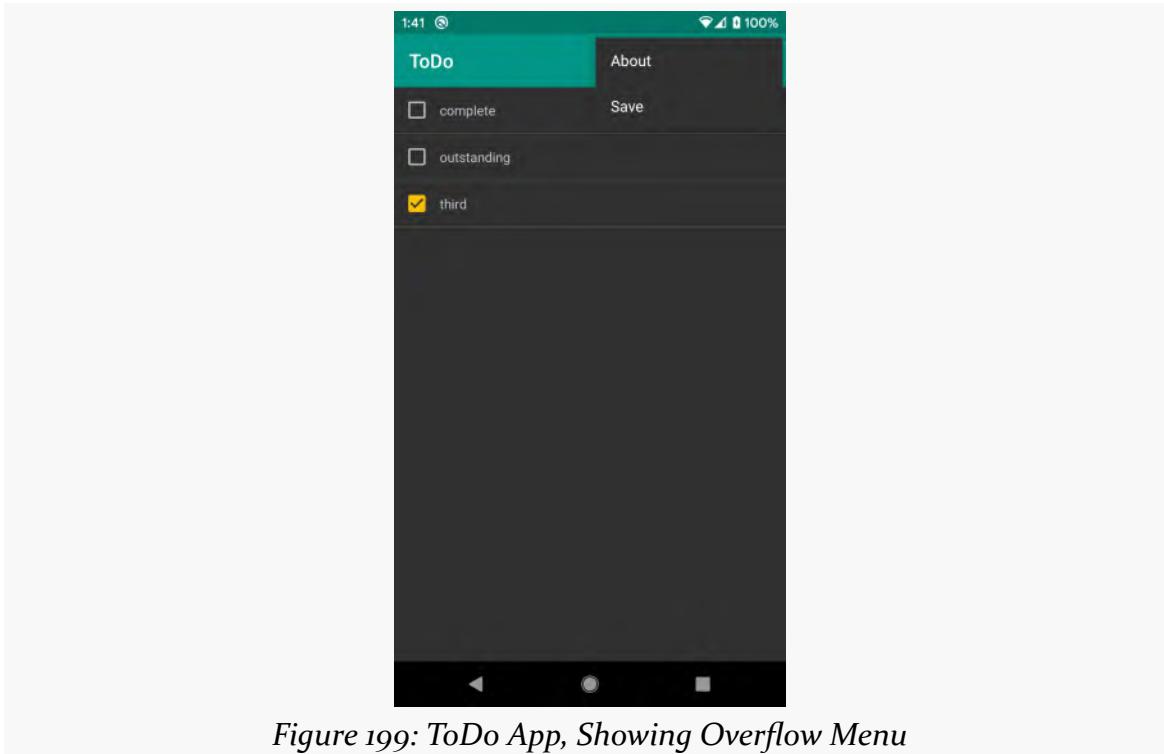


Figure 199: ToDo App, Showing Overflow Menu

Ideally, “About” would be last, as it is the least important of our overflow items. To fix this, open `res/menu/actions.xml` — the resource file containing the “About” item. Then, in the full list of the Attributes pane for the “About” item, set “`orderInCategory`” to 100. Each item is placed into a category; we are just using the default category for everything, as menu categories are rarely used. Higher numbers for “`orderInCategory`” appear later in the overflow, and so we are pushing the “About” item down by setting its “`orderInCategory`” value to 100.

GENERATING A REPORT

Now, “Save” appears before “About”:

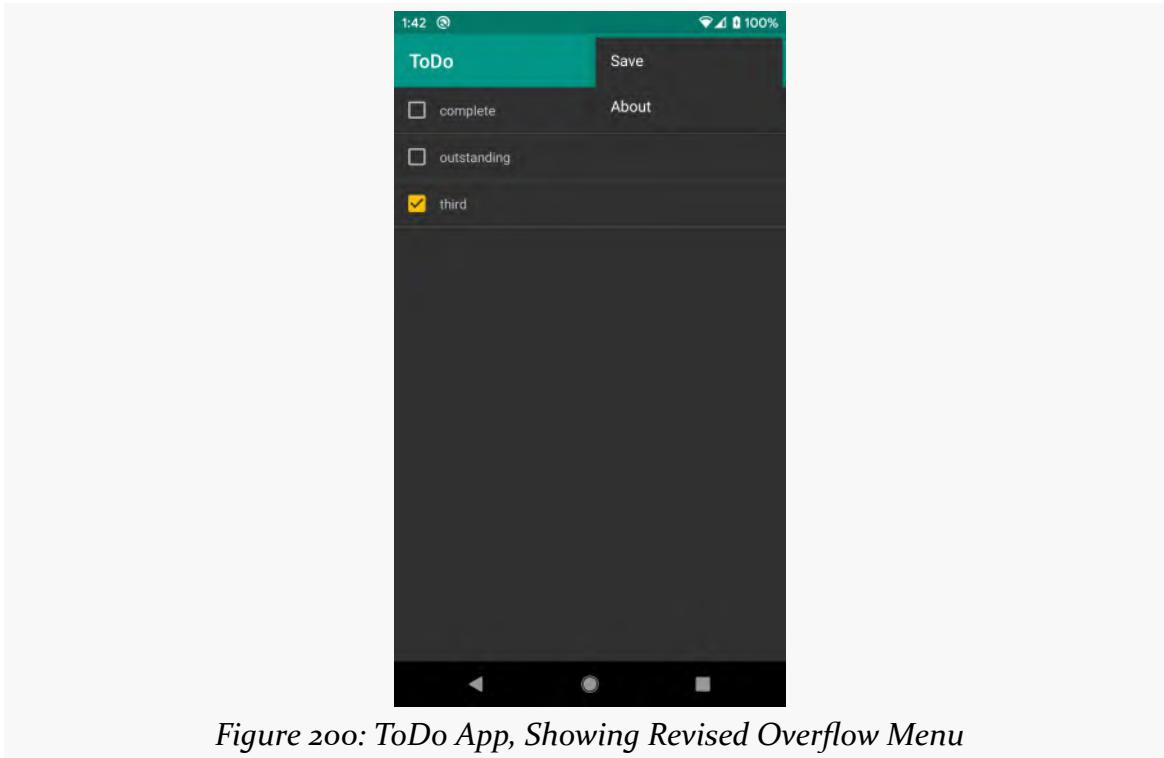


Figure 200: ToDo App, Showing Revised Overflow Menu

Step #3: Adding Some Handlebars

To generate HTML, it is often convenient to use a template language. There are lots of those, with a popular one being [Handlebars](#). While the original Handlebars is in JavaScript, there is [a port to Java](#), which we can use in our app. So, we will use it to pour our to-do item data into an HTML template to generate a Web page.

To that end, add this line to the dependencies closure in the `app/build.gradle` file:

```
implementation "com.github.jknack:handlebars:4.1.2"
```

(from [T29-Report/ToDo/app/build.gradle](#))

Then, in `ToDoApp`, add this `single` to our `koinModule`:

```
single {
    Handlebars().apply {
        registerHelper("dateFormat", Helper<Instant> { value, _ ->
            DateUtils.getRelativeDateTimeString(
                Instant.ofEpochMilli(value.toInstant())
            )
        })
    }
}
```

GENERATING A REPORT

```
        androidContext(),
        value.toEpochMilli(),
        DateUtils.MINUTE_IN_MILLIS,
        DateUtils.WEEK_IN_MILLIS, 0
    )
}
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This creates a Handlebars singleton that Koin can inject into various places in our app. We configure the Handlebars object as part of setting it up, calling a `registerHelper()` function. This registers a “helper”, which we can refer to from templates to perform a bit of formatting work for us. Specifically, we are registering a `dateFormat` helper that takes an `Instant` object and formats it using `DateUtils`, as we are doing in our `DisplayFragment`.

Step #4: Creating the Report

Now, we can work on some code to use Handlebars for converting our to-do items into a simple Web page.

First, let’s create a new package to hold our report code, since it is neither part of the UI nor part of the repository. Right-click over the `com.commonsware.todo` package in the `java/` directory, choose “New” > “Package” from the context menu, fill in `com.commonsware.todo.report` for the package name, and press or .

Then, right-click over the new `com.commonsware.todo.report` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `RosterReport`, and choose “Class” for the kind. Press or to create the class, giving you:

```
package com.commonsware.todo.report

class RosterReport {
```

Next, open up `res/values/strings.xml` again and add this odd-looking string resource:

GENERATING A REPORT

```
<string name="report_template"><![CDATA[<h1>To-Do Items</h1>
{{#this}}
<h2>{{description}}</h2>
<p>{{#completed}}<b>COMPLETED</b> &mdash; {{/completed}}Created on: {{dateFormat createdOn}}</p>
<p>{{notes}}</p>
{{/this}}
]]></string>
```

(from [T29-Report/ToDo/app/src/main/res/values/strings.xml](#))

Handlebars uses `{{ }}` syntax to indicate parts of a template that should be replaced at runtime with dynamic data. The dynamic data is represented by what Handlebars calls the “context” (which has nothing to do with Android’s `Context` class). In our case, the “context” will be a `List` of `ToDoModel` objects, representing the filtered items. Given that context:

- `{{#this}}` and `{{/this}}` represent the beginning and ending markers of a loop over that list
- `{{description}}` and `{{notes}}` pull values out of our models
- `{{#completed}}` and `{{/completed}}` represent the beginning and ending markers of a conditional section, which will only be included if `isCompleted` is true (Handlebars uses “Java beans” notation, which is why we drop off the “is” portion of the name)
- `{{dateFormat createdOn}}` will apply a `dateFormat` “helper” to format our `createdOn` value into something human-readable

Therefore, this template will create a chunk of HTML for each `ToDoModel`, with the `<h1>` heading at the top. We need the `CDATA` stuff so that Android does not try interpreting the HTML tags inside of this string resource.

Next, add a constructor to `RosterReport` that gives us a `Context`, our Handlebars object, and the `appScope` that we used in `ToDoRepository`:

```
class RosterReport(
    private val context: Context,
    engine: Handlebars,
    private val appScope: CoroutineScope
) {
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/report/RosterReport.kt](#))

Handlebars compiles a template like the one from our string resource into a `Template` object. So, add this property to `RosterReport`:

```
private val template =
    engine.compileInline(context.getString(R.string.report_template))
```

GENERATING A REPORT

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/report/RosterReport.kt](#))

This retrieves the string resource and has the Handlebars object compile it for us. We still need some code to actually use this template, which we will work on shortly.

Finally, add another line to koinModule in ToDoApp to allow us to inject our RosterReport where needed:

```
single { RosterReport(androidContext(), get(), get(named("appScope")))} 
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Step #5: Writing Where the User Asked

We now need to start connecting the createDoc request to our RosterReport.

What we get from createDoc is a Uri pointing to... something. It is up to the user where to save the Web page, and that could be anything from a local file to an entry in Google Drive. A ContentResolver has an openOutputStream() method that will work with any Uri returned by createDoc, so we will not need to worry about the location details.

This means that RosterReport needs a function where we can hand it the user-supplied Uri and have it write the report to that location.

To that end, add this function to RosterReport:

```
suspend fun generate(content: List<ToDoModel>, doc: Uri) {
    withContext(Dispatchers.IO + appScope.coroutineContext) {
        context.contentResolver.openOutputStream(doc, "rwt")?.writer()?.use { osw ->
            osw.write(template.apply(content))
            osw.flush()
        }
    }
} 
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/report/RosterReport.kt](#))

generate() takes a List of models, along with that Uri. In a coroutine set to run on a background thread and managed by our appScope (withContext(Dispatchers.IO + appScope.coroutineContext)), we:

- Open an OutputStream on the location specified by the Uri
- Wrap that in an OutputStreamWriter
- Call use() on the writer to automatically close it when we are done

GENERATING A REPORT

- Call `apply()` on our template to have it generate the HTML for our models
- Write that to the `OutputStreamWriter`

You may get a warning on the `openOutputStream()` call, complaining about an “inappropriate blocking method call”. That is [an IDE bug](#).

The “rwt” parameter to `openOutputStream()` represents the “mode”, indicating what we want to do with the stream. The default mode, if you leave off this parameter, is “w”, indicating that you want to overwrite parts of the content. The “t” in “rwt” indicates that we want to truncate the output — whatever we write becomes the complete content. The “r” means we want to be able to read the content. We are not using that capability, but “wt” is not a documented option for the mode, so we settle for “rwt” to get write and truncate capability.

Step #6: Saving the Report

Now, we can wrap all this up, saving the report to the desired location.

First, add a new constructor parameter to `RosterMotor`, so we can get access to a `RosterReport` instance:

```
class RosterMotor(  
    private val repo: ToDoRepository,  
    private val report: RosterReport  
) : ViewModel() {
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This will require a change to its corresponding line in `ToDoApp` to get() the second parameter:

```
viewModel { RosterMotor(get(), get()) }
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Then, add this function to `RosterMotor`:

```
fun saveReport(doc: Uri) {  
    viewModelScope.launch {  
        report.generate(_states.value.items, doc)  
    }  
}
```

GENERATING A REPORT

Here, we launch() our generate() coroutine, supplying that list of items from the current RosterViewState, along with the Uri identifying where we want the report to be written.

Next, in RosterListFragment, modify the createDoc property to call saveReport() on our RosterMotor:

```
private val createDoc =  
    registerForActivityResult(ActivityResultContracts.CreateDocument()) {  
        motor.saveReport(it)  
    }
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Step #7: Viewing the Report

One limitation of what we have now is that we do not do anything once the report is saved. We should have some sort of acknowledgment, so the user knows the report is ready for use.

One possibility is to simply show the user the report. We can use an ACTION_VIEW Intent to display the report, using the Uri pointing to where we saved it.

First, though, we need our fragment to find out when the report is ready to be viewed and that we should navigate to the Web browser app to display it.

However, we need to be careful about how we do that. We could just tuck the Uri into a new RosterViewState and have our fragment see the Uri and launch the browser. However, we only want to launch the browser *once*, not on every future updated viewstate.

For Kotlin, the current recommended pattern for handling this is to use a SharedFlow. Whereas StateFlow is for states, a SharedFlow is better for events.

Right now, we have a single thing that we want to treat as an event: the report is ready to be viewed. However, in [the next tutorial](#), we will add another. A typical way of representing this in Kotlin is to use a sealed class, which is basically “an enum with superpowers”. So, add this Nav sealed class to RosterMotor.kt for representing all of our navigation requests:

GENERATING A REPORT

```
sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This has a `ViewReport` subclass for representing the “hey! let’s view the report!” navigation request. `ViewReport` wraps the `Uri` that identifies where the report is stored.

Then, add these properties to `RosterMotor`:

```
private val _navEvents = MutableSharedFlow<Nav>()
val navEvents = _navEvents.asSharedFlow()
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This sets up a `MutableSharedFlow`, this time for a `Nav` object. Like `MutableStateFlow`, `MutableSharedFlow` is a `SharedFlow` that we manage ourselves, calling `emit()` when we want to publish an event. This is `private`; we use `asSharedFlow()` to make a `SharedFlow` available for the fragment to use to consume the events off of the channel (`navEvents`).

Next, modify `saveReport()` in `RosterMotor` to be:

```
fun saveReport(doc: Uri) {
    viewModelScope.launch {
        report.generate(_states.value.items, doc)
        _navEvents.emit(Nav.ViewReport(doc))
    }
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Here, once the report has been saved, we `emit()` a `ViewReport` request to our `MutableSharedFlow`.

In order to view the report, we are going to want to use an `ACTION_VIEW` Intent and `startActivity()`. It is very likely that the user will have an app that supports `ACTION_VIEW` for HTML, such as a Web browser. But, it is not guaranteed. The problem is that `startActivity()` will throw an `ActivityNotFoundException` if the user does not have anything that supports `ACTION_VIEW` for HTML, which will lead to a crash if we do not take some steps.

To that end, add this `safeStartActivity()` function to `RosterListFragment`:

GENERATING A REPORT

```
private fun safeStartActivity(intent: Intent) {
    try {
        startActivity(intent)
    } catch (t: Throwable) {
        Log.e(TAG, "Exception starting $intent", t)
        Toast.makeText(requireActivity(), R.string.oops, Toast.LENGTH_LONG).show()
    }
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

The big thing is that we use a try/catch block to handle any exception that might get raised by trying to start an activity. The most likely exception would be `ActivityNotFoundException`, meaning that no activity was found that matched the Intent that we used to try to start the activity. `safeStartActivity()` has a few other “bells and whistles”:

- If we do catch an exception, we log a message to Logcat with the exception itself (so the stack trace shows up)
- And, if we catch an exception, we show a `Toast` to the user, which presents a message in little temporary popup window

This code will have a couple of errors due to some missing symbols. The first missing symbol is `TAG`. This is a label that is included in our Logcat output. Since this string is not visible to users, we do not need to worry about translating it, so a plain string is fine. So, add this `TAG` constant towards the top of the `RosterListFragment.kt` source file:

```
private const val TAG = "ToDo"
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Also, the `Toast.makeText()` call references an `oops` string resource that we have not defined. So, in `res/values/strings.xml`, add:

```
<string name="oops">Sorry! Something went wrong!</string>
```

(from [T29-Report/ToDo/app/src/main/res/values/strings.xml](#))

Then, in `RosterListFragment`, add this `viewReport()` function:

```
private fun viewReport(uri: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_VIEW, uri)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    )
}
```

GENERATING A REPORT

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This sets up an ACTION_VIEW Intent, where ACTION_VIEW is the standard Intent action for “I want to view... something...”. Here, the “something...” is the report, identified by the supplied Uri. We need to add FLAG_GRANT_READ_URI_PERMISSION to the Intent to ensure that the Web browser (or other app responding to our Intent) is given read access to our content. Then, we call safeStartActivity() to bring up the Web browser (or whatever).

Finally, in RosterListFragment, towards the bottom of onViewCreated(), add the following:

```
viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.navEvents.collect { nav ->
        when (nav) {
            is Nav.ViewReport -> viewReport(nav.doc)
        }
    }
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This is the same basic structure that we use for the states StateFlow, this time for the navEvents SharedFlow

Now, if you choose “Save” from the toolbar and pick a spot to write the report, you will either be taken to the saved report or, possibly, see the Toast popup indicating that the report was saved. You may or may not have a Web browser that supports the particular sort of Uri that we get back from the Storage Access Framework.

Final Results

The actions_roster menu resource should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/filter"
        android:icon="@drawable/ic_filter"
        android:title="@string/menu_filter"
        app:showAsAction="ifRoom|withText">
        <menu>
```

GENERATING A REPORT

```
<group
    android:id="@+id/filter_group"
    android:checkableBehavior="single" >
    <item
        android:id="@+id/all"
        android:checked="true"
        android:title="@string/menu_filter_all" />
    <item
        android:id="@+id/completed"
        android:title="@string/menu_filter_completed" />
    <item
        android:id="@+id/outstanding"
        android:title="@string/menu_filter_outstanding" />
</group>
</menu>
</item>
<item
    android:id="@+id/add"
    android:icon="@drawable/ic_add"
    android:title="@string/menu_add"
    app:showAsAction="ifRoom|withText" />
<item
    android:id="@+id/save"
    android:icon="@drawable/ic_save"
    android:title="@string/menu_save"
    app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T29-Report/ToDo/app/src/main/res/menu/actions_roster.xml](#))

RosterListFragment, after all of our changes, should resemble:

```
package com.commonsware.todo.ui.roster

import android.content.Intent
import android.net.Uri
import android.os.Bundle
import android.util.Log
import android.view.*
import android.widget.Toast
import androidx.activity.result.contract.ActivityResultContracts
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
```

GENERATING A REPORT

```
import com.commonsware.todo.databinding.TodoRosterBinding
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel

private const val TAG = "ToDo"

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private val menuMap = mutableMapOf<FilterMode, MenuItem>()
    private var binding: TodoRosterBinding? = null

    private val createDoc =
        registerForActivityResult(ActivityResultContracts.CreateDocument()) {
            motor.saveReport(it)
        }

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root

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

        val adapter = RosterAdapter(
            layoutInflater,
            onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
            onRowClick = ::display
        )

        binding?.items?.apply {
            setAdapter(adapter)
            layoutManager = LinearLayoutManager(context)

            addItemDecoration(
                DividerItemDecoration(
                    activity,

```

GENERATING A REPORT

```
        DividerItemDecoration.VERTICAL
    )
)
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = if (state.isLoaded) View.GONE else View.VISIBLE

            when {
                state.items.isEmpty() && state.filterMode == FilterMode.ALL -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty_filtered)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }

    menuMap[state.filterMode]?.isChecked = true
}
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.navEvents.collect { nav ->
        when (nav) {
            is Nav.ViewReport -> viewReport(nav.doc)
        }
    }
}
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)

    menuMap.apply {
```

GENERATING A REPORT

```
        put(FilterMode.ALL, menu.findItem(R.id.all))
        put(FilterMode.COMPLETED, menu.findItem(R.id.completed))
        put(FilterMode.OUTSTANDING, menu.findItem(R.id.outstanding))
    }

    menuMap[motor.states.value.filterMode]?.isChecked = true

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
        R.id.all -> {
            item.isChecked = true
            motor.load(FilterMode.ALL)
            return true
        }
        R.id.completed -> {
            item.isChecked = true
            motor.load(FilterMode.COMPLETED)
            return true
        }
        R.id.outstanding -> {
            item.isChecked = true
            motor.load(FilterMode.OUTSTANDING)
            return true
        }
        R.id.save -> {
            saveReport()
            return true
        }
    }
}

return super.onOptionsItemSelected(item)
}

private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}
```

GENERATING A REPORT

```
private fun saveReport() {
    createDoc.launch("report.html")
}

private fun viewReport(uri: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_VIEW, uri)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    )
}

private fun safeStartActivity(intent: Intent) {
    try {
        startActivity(intent)
    } catch (t: Throwable) {
        Log.e(TAG, "Exception starting $intent", t)
        Toast.makeText(requireActivity(), R.string.oops, Toast.LENGTH_LONG).show()
    }
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

The strings value resource should contain:

```
<resources>
    <string name="app_name">ToDo</string>
    <string name="msg_empty">Click the + icon to add a todo item!</string>
    <string name="msg_empty_filtered">Click the + icon to add a todo item, or change
your filter to show other items</string>
    <string name="menu_about">About</string>
    <string name="is_completed">Item is completed</string>
    <string name="created_on">Created on:</string>
    <string name="menu_edit">Edit</string>
    <string name="desc">Description</string>
    <string name="notes">Notes</string>
    <string name="menu_save">Save</string>
    <string name="menu_add">Add</string>
    <string name="menu_delete">Delete</string>
    <string name="menu_filter">Filter</string>
    <string name="menu_filter_all">All</string>
    <string name="menu_filter_completed">Completed</string>
    <string name="menu_filter_outstanding">Outstanding</string>
    <string name="oops">Sorry! Something went wrong!</string>
    <string name="report_template"><![CDATA[<h1>To-Do Items</h1>
{{#this}}
<h2>{description}</h2>
```

GENERATING A REPORT

```
<p>{{#completed}}<b>COMPLETED</b> &mdash; {{/completed}}Created on: {{dateFormat  
createdOn}}</p>  
<p>{{notes}}</p>  
<{/this}>  
]]></string>  
</resources>
```

(from [T29-Report/ToDo/app/src/main/res/values/strings.xml](#))

Our module's build.gradle file should resemble:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
    id 'androidx.navigation.safeargs.kotlin'  
    id 'kotlin-kapt'  
}  
  
android {  
    compileSdk 31  
  
    defaultConfig {  
        applicationId "com.commonsware.todo"  
        minSdk 21  
        targetSdk 31  
        versionCode 1  
        versionName "1.0"  
  
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    }  
  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'),  
            'proguard-rules.pro'  
        }  
    }  
  
    buildFeatures {  
        viewBinding true  
    }  
  
    compileOptions {  
        coreLibraryDesugaringEnabled true  
        sourceCompatibility JavaVersion.VERSION_1_8  
        targetCompatibility JavaVersion.VERSION_1_8  
    }  
}
```

GENERATING A REPORT

```
kotlinOptions {  
    jvmTarget = '1.8'  
}  
  
packagingOptions {  
    exclude 'META-INF/AL2.0'  
    exclude 'META-INF/LGPL2.1'  
}  
}  
  
dependencies {  
    implementation 'androidx.core:core-ktx:1.6.0'  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
    implementation "androidx.recyclerview:recyclerview:1.2.1"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation 'com.google.android.material:material:1.4.0'  
    implementation "io.insert-koin:koin-android:$koin_version"  
    implementation "com.github.jknack:handlebars:4.1.2"  
    implementation "androidx.room:room-runtime:$room_version"  
    implementation "androidx.room:room-ktx:$room_version"  
    kapt "androidx.room:room-compiler:$room_version"  
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'  
    testImplementation 'junit:junit:4.13.2'  
    testImplementation "org.mockito:mockito-inline:3.12.1"  
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"  
    testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
    androidTestImplementation "androidx.arch.core:core-testing:2.1.0"  
    androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'  
}
```

(from [T29-Report/ToDo/app/build.gradle](#))

ToDoApp, after a few revisions, should resemble:

```
package com.commonsware.todo  
  
import android.app.Application  
import android.text.format.DateUtils  
import com.commonsware.todo.repo.ToDoDatabase  
import com.commonsware.todo.repo.ToDoRepository  
import com.commonsware.todo.report.RosterReport  
import com.commonsware.todo.ui.SingleModelMotor  
import com.commonsware.todo.ui.roster.RosterMotor
```

GENERATING A REPORT

```
import com.github.jknack.handlebars.Handlebars
import com.github.jknack.handlebars.Helper
import kotlinx.coroutinesCoroutineScope
import kotlinx.coroutinesSupervisorJob
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.core.qualifier.named
import org.koin.dsl.module
import java.time.Instant

class ToDoApp : Application() {
    private val koinModule = module {
        single(named("appScope")) { CoroutineScope(SupervisorJob()) }
        single { ToDoDatabase.newInstance(androidContext()) }
        single {
            ToDoRepository(
                get<ToDoDatabase>().todoStore(),
                get(named("appScope"))
            )
        }
        single {
            Handlebars().apply {
                registerHelper("dateFormat", Helper<Instant> { value, _ ->
                    DateUtils.getRelativeDateTimeString(
                        androidContext(),
                        value.toEpochMilli(),
                        DateUtils.MINUTE_IN_MILLIS,
                        DateUtils.WEEK_IN_MILLIS, 0
                    )
                })
            }
        }
        single { RosterReport(androidContext(), get(), get(named("appScope"))) }
        viewModel { RosterMotor(get(), get()) }
        viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@ToDoApp)
            modules(koinModule)
        }
    }
}
```

GENERATING A REPORT

```
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Our new RosterReport class should look like:

```
package com.commonsware.todo.report

import android.content.Context
import android.net.Uri
import com.commonsware.todo.R
import com.commonsware.todo.repo.ToDoModel
import com.github.jknack.handlebars.Handlebars
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext

class RosterReport(
    private val context: Context,
    engine: Handlebars,
    private val appScope: CoroutineScope
) {
    private val template =
        engine.compileInline(context.getString(R.string.report_template))

    suspend fun generate(content: List<ToDoModel>, doc: Uri) {
        withContext(Dispatchers.IO + appScope.coroutineContext) {
            context.contentResolver.openOutputStream(doc, "rwt")?.writer()?.use { osw ->
                osw.write(template.apply(content))
                osw.flush()
            }
        }
    }
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/report/RosterReport.kt](#))

And, our updated RosterMotor should contain:

```
package com.commonsware.todo.ui.roster

import android.net.Uri
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.report.RosterReport
```

GENERATING A REPORT

```
import kotlinx.coroutines.Job
import kotlinx.coroutines.flow.*
import kotlinx.coroutines.launch

data class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val isLoading: Boolean = false,
    val filterMode: FilterMode = FilterMode.ALL
)

sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
}

class RosterMotor(
    private val repo: ToDoRepository,
    private val report: RosterReport
) : ViewModel() {
    private val _states = MutableStateFlow(RosterViewState())
    val states = _states.asStateFlow()
    private val _navEvents = MutableSharedFlow<Nav>()
    val navEvents = _navEvents.asSharedFlow()
    private var job: Job? = null

    init {
        load(FilterMode.ALL)
    }

    fun load(filterMode: FilterMode) {
        job?.cancel()

        job = viewModelScope.launch {
            repo.items(filterMode).collect {
                _states.emit(RosterViewState(it, true, filterMode))
            }
        }
    }

    fun save(model: ToDoModel) {
        viewModelScope.launch {
            repo.save(model)
        }
    }

    fun saveReport(doc: Uri) {
        viewModelScope.launch {
            report.generate(_states.value.items, doc)
            _navEvents.emit(Nav.ViewReport(doc))
        }
    }
}
```

GENERATING A REPORT

```
    }  
}  
}
```

(from [T29-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/res/menu/actions_roster.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)
- [app/build.gradle](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/report/RosterReport.kt](#)
- [app/src/main/res/values/strings.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)

Sharing the Report

We have the HTML form of our to-do list, and, on some devices, we can view it automatically in a Web browser.

However, for getting the report off of the device, the user has only clunky options:

- The user could copy the report from wherever they stored it to wherever they want, but that requires launching other apps or using desktop software in many cases
- If a Web browser appeared to view the report, it might have a “share” option that the user could use, but not all devices will have a compatible browser

Ideally, *our app* would have its own “share” option, so the report can be handed to any app that can share HTML. In this tutorial, we will add such an option to our app bar.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding a Share App Bar Item

Right now, it may seem like Android app development is just a series of app bar items, with little bits of code between them. This is a gross exaggeration, as there is quite a bit of Android development that does not involve creating app bar items.

That being said... we need to create another app bar item.

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector

SHARING THE REPORT

Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Icon” button and search for share:

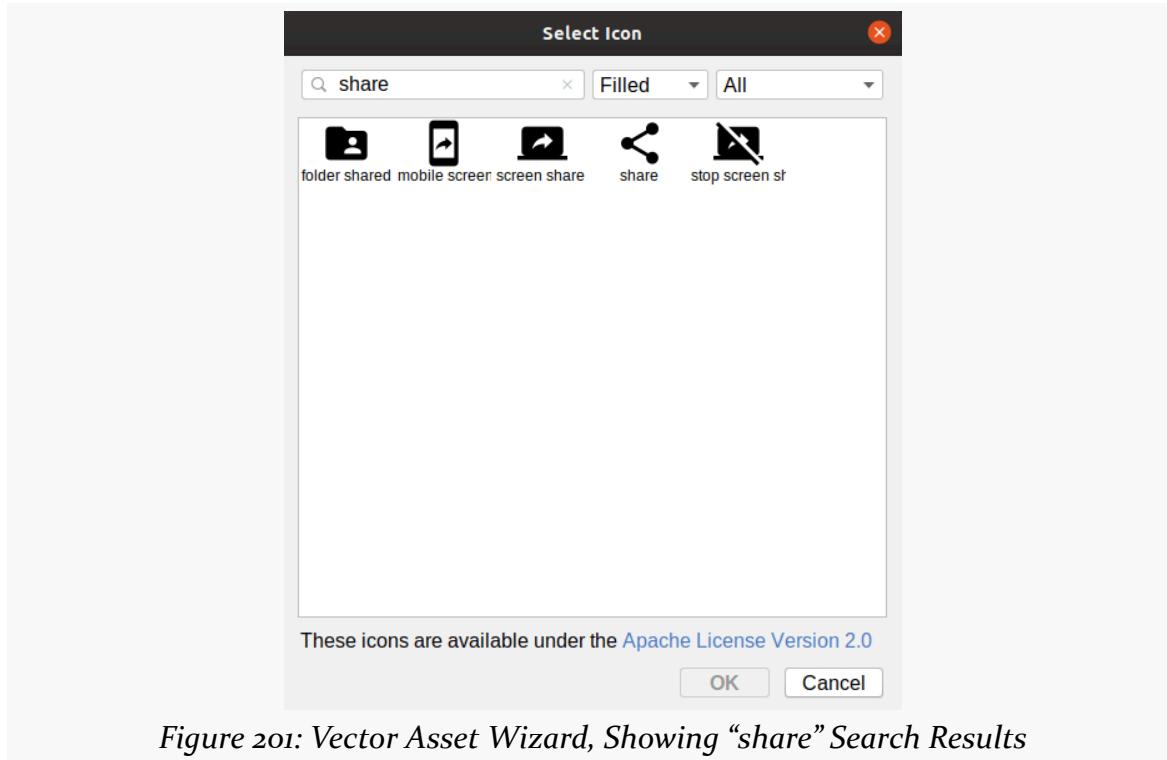


Figure 201: Vector Asset Wizard, Showing “share” Search Results

Choose the “share” icon and click “OK” to close up the icon selector. Change the icon’s name to `ic_share`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

SHARING THE REPORT

Open up the `res/menu/actions_roster.xml` resource file, and switch to the graphical designer. Drag an “Item” from the “Palette” view into the Component Tree, slotting it after the existing “save” item:

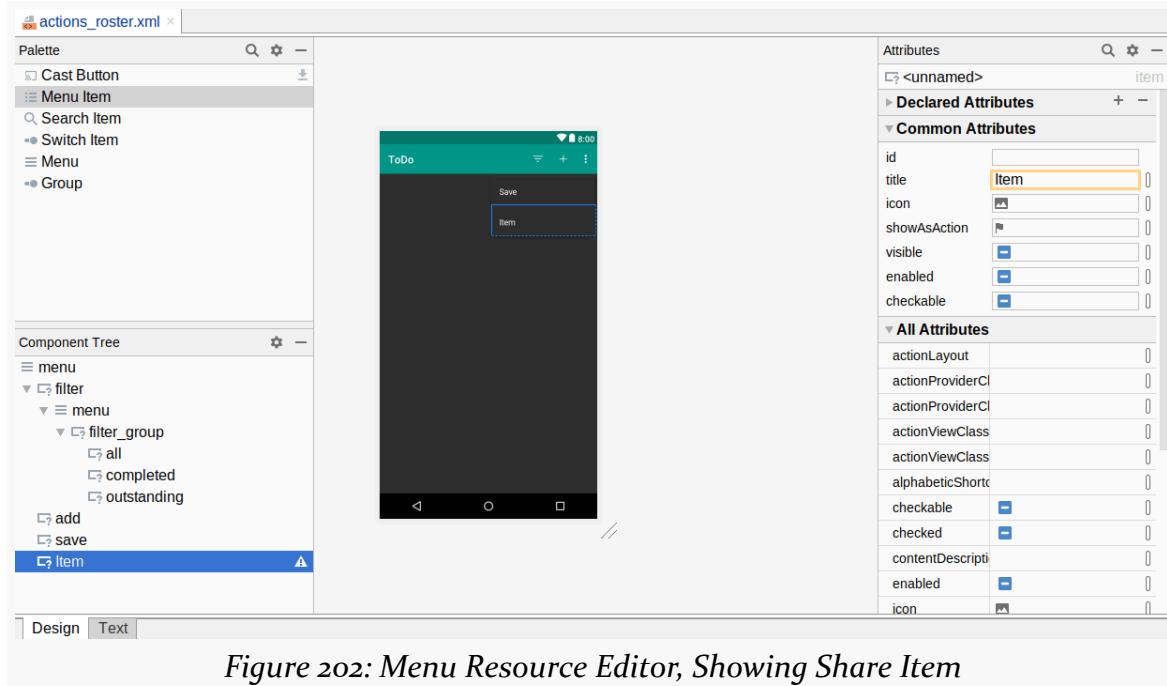


Figure 202: Menu Resource Editor, Showing Share Item

In the Attributes view for this new item, assign it an ID of “share”. Then, choose both “ifRoom” and “withText” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Click on `ic_share` in the list of drawables, then click OK to accept that choice of icon.

Then, click the “O” 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_share` as the resource name and “Share” as the resource value. Click OK to close the dialog and complete the configuration of this app bar item.

Step #2: Adding FileProvider

For the purposes of a “share” app bar item, we want the experience to be fairly seamless: the user clicks the item, then is prompted with options for sharing it. The flow should not be: the user clicks the item, goes through some UI to choose where

SHARING THE REPORT

to save it, then is prompted with options for sharing it. In other words, we should not be using ACTION_CREATE_DOCUMENT as we are in the “save” scenario.

We can save the report to a file fairly easily. However, on modern versions of Android, we cannot share a file with other apps. However, Jetpack offers FileProvider, which is way for us to serve files to other apps. All we need to do is add it to our manifest and configure it.

Open the app/src/main/AndroidManifest.xml file and add this XML element to the manifest, below the two existing <activity> elements:

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="${applicationId}.provider"
    android:exported="false"
    android:grantUriPermissions="true">
</provider>
```

FileProvider is a ContentProvider, which is an Android component that... provides content. Just as an <activity> element identifies an Activity in our app, a <provider> element identifies a ContentProvider in our app. In this case, instead of it being one that we wrote, we are going to use FileProvider. As a result, our android:name attribute has to be the fully-qualified class name to FileProvider (androidx.core.content.FileProvider).

SHARING THE REPORT

The android:authorities attribute indicates what name we wish to use for our provider. This name needs to be unique, and it fills a similar role as does a domain name in Web development. Here, we use \${applicationId}.provider. The \${applicationId} part is a “manifest placeholder” — a macro that will be expanded when our app is compiled and turned into our app’s application ID. If you click on the “Merged Manifest” sub-tab, you will see the results of this expansion:



The screenshot shows the AndroidManifest.xml file in the Android Studio editor. The 'Merged Manifest' tab is selected. The code is as follows:

```
AndroidManifest.xml
<manifest>
    <activity
        android:name="com.commonsware.todo.ui.AboutActivity" />
    <activity
        android:name="com.commonsware.todo.ui.MainActivity" >
        <intent-filter>
            <action
                android:name="android.intent.action.MAIN" />
            <category
                android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <provider
        android:authorities="com.commonsware.todo.provider"
        android:exported="false"
        android:grantUriPermissions="true"
        android:name="androidx.core.content.FileProvider" />
    <service
        android:exported="false"
        android:name="androidx.room.MultiInstanceInvalidation" />
</manifest>
```

The line `android:authorities="com.commonsware.todo.provider"` is highlighted in blue, indicating it is a manifest placeholder.

Figure 203: Merged Manifest, Showing Expanded applicationId Placeholder

By using \${applicationId}, we are helping to ensure that our authority value is unique, as the applicationId value itself is guaranteed to be unique on the device.

The android:exported="false" value indicates that our provider is not to be exported, meaning that by default, other apps have no ability to access our provider’s content. This may seem silly, as the point of having this provider is to get our report to other apps. However, FileProvider does not support android:exported="true". Instead, we use android:grantUriPermissions="true" to indicate that we will grant rights to other apps on a case-by-case basis at runtime.

We need to tell our FileProvider what files it should serve. To do this, we need to create an XML file with instructions, a bit reminiscent of how you configure a Web server to say what directories it should serve.

SHARING THE REPORT

To that end, right-click over `res/` and choose “New” > “Android Resource Directory” from the context menu. This brings up the “New Resource Directory” dialog:

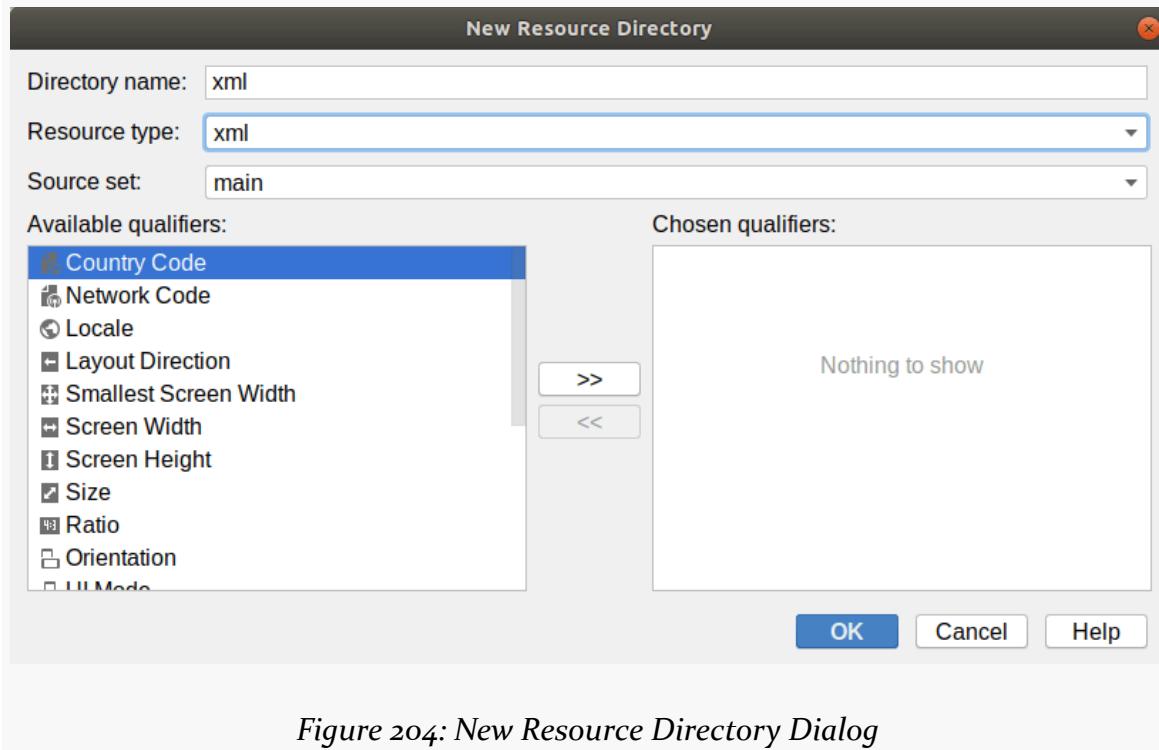


Figure 204: New Resource Directory Dialog

In the “Resource type” drop-down, choose “xml”. Leave everything else alone, and click “OK” to create a `res/xml/` directory in our project. This directory is good for holding arbitrary XML files — so long as they are well-formed XML, the build tools do not care about the exact contents of those files.

Then, right-click over the new `res/xml/` directory and choose “New” > “XML resource file” from the context menu. Fill in `provider_paths.xml` as the name, fill in `paths` for the “Root element”, then click “OK” to create this file.

This should give you a file like this:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">

</paths>
```

Replace that with:

SHARING THE REPORT

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <cache-path name="shared" path="shared" />
</paths>
```

(from [T3o-Share/ToDo/app/src/main/res/xml/provider_paths.xml](#))

The `<paths>` list all of the locations that we want `FileProvider` to serve. In our case, there is only one child element, so we will only serve from this one place.

The child element name — `cache-path` — says that we start with the filesystem location that represents the “cache” portion of our app’s internal storage. This is the location identified by the `getCacheDir()` method on `Context`, and we will use that method to save our report to a file. The `path` attribute further constrains `FileProvider` to only serve files from the `shared/` directory inside of `getCacheDir()`. The `name` attribute indicates that the `Uri` values that `FileProvider` uses to identify its content should have a shared path segment that maps to this filesystem location.

Then, to teach `FileProvider` about this XML, modify the manifest entry to have a child `<meta-data>` element:

```
<provider
    android:name=" androidx.core.content.FileProvider"
    android:authorities="${applicationId}.provider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name=" android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/provider_paths" />
</provider>
```

(from [T3o-Share/ToDo/app/src/main/AndroidManifest.xml](#))

In a manifest, `<meta-data>` refers to additional information that the component can use to configure its operation, or that users of that component can use to know what that component is supposed to do. In this case, we are using it to configure the `FileProvider`. The `FileProvider` class knows to look up its `android.support.FILE_PROVIDER_PATHS` metadata entry and read the `<paths>` out of the associated XML resource. This way, we do not need to subclass `FileProvider` and override methods to teach it what files to serve — it can handle that on its own via this metadata.

Step #3: Caching the Report

To share the report with other apps, we need to write it to a file that can be served by the `FileProvider`. Based on the `FileProvider` metadata, that would be in a shared/ subdirectory off of the location supplied by the `getCacheDir()` method on a Context. The work to save the report to this file should be done on a background thread. When that work is done, then we can actually share the report with other apps.

Add these functions to `RosterMotor`:

```
fun shareReport() {
    viewModelScope.launch {
        saveForSharing()
    }
}

private suspend fun saveForSharing() {
    withContext(Dispatchers.IO + appScope.coroutineContext) {
        val shared = File(context.cacheDir, "shared").also { it.mkdirs() }
        val reportFile = File(shared, "report.html")
        val doc = FileProvider.getUriForFile(context, AUTHORITY, reportFile)

        _states.value.let { report.generate(it.items, doc) }
        _navEvents.emit(Nav.ShareReport(doc))
    }
}
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

`saveReport()` is simply a wrapper around `saveForSharing()`, launching another coroutine. `saveForSharing()` implements that coroutine, where we:

- Create the shared directory under `getCacheDir()`
- Create a `File` object pointing to a `report.html` file in that shared directory
- Use `FileProvider.getUriForFile()` to get a `Uri` from `FileProvider` that maps to our `File`
- Ask our `RosterReport` to write the report to that `Uri`
- Post another navigation request, this time indicating that our report is ready for sharing

You will have a few compile errors. One is that `AUTHORITY` is undefined. This needs to match the value that we have in the `android:authorities` attribute in the

SHARING THE REPORT

<provider> element in the manifest. That, in turn, is being created based on our application ID. So, add this constant to RosterMotor.kt:

```
private const val AUTHORITY = BuildConfig.APPLICATION_ID + ".provider"
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

BuildConfig is a code-generated class that contains constants associated with our build, and BuildConfig.APPLICATION_ID is our application ID. As a result, AUTHORITY is being assembled the same way that the android:authorities value is being assembled.

Another compile error is that there is no Nav.ShareReport class. Fix that by changing Nav to look like:

```
sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
    data class ShareReport(val doc: Uri) : Nav()
}
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This adds another subclass to Nav called ShareReport, which also wraps a Uri.

saveForSharing() refers to context in a couple of places. That needs to be a Context, so we can use it for getCacheDir(), and for FileProvider.getUriForFile(). It also refers to appScope, which is the custom CoroutineScope that we are using for write operations. So, add two more constructor parameters to RosterMotor for context and appScope:

```
class RosterMotor(
    private val repo: ToDoRepository,
    private val report: RosterReport,
    private val context: Application,
    private val appScope: CoroutineScope
) : ViewModel() {
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

We use Application for the context parameter to avoid an invalid Lint check complaint. Google is worried that a ViewModel with a Context parameter might result in a memory leak, but the Lint check cannot distinguish between valid and invalid uses of Context. To make the Lint error go away, we use Application, which means that our ToDoApp will need to be the Context.

SHARING THE REPORT

This requires a corresponding change to ToDoApp, adding androidApplication() and get(named("appScope")) to our RosterMotor constructor call:

```
viewModel { RosterMotor(get(), get(), androidApplication(), get(named("appScope"))) }
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

androidApplication() is functionally equivalent to androidContext(), but it returns an Application to satisfy compiler type checking.

Step #4: Sharing the Report

To actually share the report, we need to start an ACTION_SEND activity. ACTION_SEND is the implicit Intent action used for most of the “share” options that you see in Android apps. We can provide it with the Uri to the report, via an EXTRA_STREAM extra. Usually, a device will have 2+ apps that support ACTION_SEND for a text/html MIME type, so frequently the user will get a chooser, asking which of those apps to use. If the user has only one compatible app, or if the user chose a default share target on some past ACTION_SEND Intent, then there will be no chooser, and the user will be taken straight to some ACTION_SEND-supporting activity.

But, there is also the chance that there are zero apps that support ACTION_SEND for text/html. You might encounter this on an emulator, for example, which usually has few apps installed. So we need to handle this scenario as well, just as we did in the preceding tutorial, where we needed to handle the case where there was no ACTION_VIEW Intent for our content.

In RosterListFragment, add this shareReport() function:

```
private fun shareReport(doc: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_SEND)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
            .setType("text/html")
            .putExtra(Intent.EXTRA_STREAM, doc)
    )
}
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This is nearly identical to viewReport(). We create an ACTION_SEND Intent, tucking our Uri into the EXTRA_STREAM extra. We use setType() to indicate that this is HTML — this is needed due to the way that ACTION_SEND needs the Uri to be in an

SHARING THE REPORT

extra rather than in the main portion of the Intent the way our ACTION_VIEW Intent was set up.

Then, in `onOptionsItemSelected()`, add another branch to handle the share app bar item, routing it to `shareReport()` on our `RosterMotor`:

```
R.id.share -> {
    motor.shareReport()
    return true
}
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Finally, in `onViewCreated()` of `RosterListFragment`, modify the `navEvents` collector configuration to look like:

```
viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.navEvents.collect { nav ->
        when (nav) {
            is Nav.ViewReport -> viewReport(nav.doc)
            is Nav.ShareReport -> shareReport(nav.doc)
        }
    }
}
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Now, if you run the app and click the “share” action item, you should get some options for sharing the generated report.

Final Results

The `actions_roster` menu resource should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/filter"
        android:icon="@drawable/ic_filter"
        android:title="@string/menu_filter"
        app:showAsAction="ifRoom|withText">
        <menu>
```

SHARING THE REPORT

```
<group
    android:id="@+id/filter_group"
    android:checkableBehavior="single" >
    <item
        android:id="@+id/all"
        android:checked="true"
        android:title="@string/menu_filter_all" />
    <item
        android:id="@+id/completed"
        android:title="@string/menu_filter_completed" />
    <item
        android:id="@+id/outstanding"
        android:title="@string/menu_filter_outstanding" />
</group>
</menu>
</item>
<item
    android:id="@+id/add"
    android:icon="@drawable/ic_add"
    android:title="@string/menu_add"
    app:showAsAction="ifRoom|withText" />
<item
    android:id="@+id/save"
    android:icon="@drawable/ic_save"
    android:title="@string/menu_save"
    app:showAsAction="ifRoom|withText" />
<item
    android:id="@+id/share"
    android:icon="@drawable/ic_share"
    android:title="@string/menu_share"
    app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T3o-Share/ToDo/app/src/main/res/menu/actions_roster.xml](#))

The overall `AndroidManifest.xml` file should now look like:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.todo">

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

    <application>
```

SHARING THE REPORT

```
    android:name=".ToDoApp"
    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/Theme.ToDo">
    <activity
        android:name=".ui.AboutActivity"
        android:exported="true" />
    <activity
        android:name=".ui.MainActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    <provider
        android:name="androidx.core.content.FileProvider"
        android:authorities="${applicationId}.provider"
        android:exported="false"
        android:grantUriPermissions="true">
        <meta-data
            android:name="android.support.FILE_PROVIDER_PATHS"
            android:resource="@xml/provider_paths" />
    </provider>
</application>

</manifest>
```

(from [T3o-Share/ToDo/app/src/main/AndroidManifest.xml](#))

Our new provider_paths XML resource should contain:

```
<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <cache-path name="shared" path="shared" />
</paths>
```

(from [T3o-Share/ToDo/app/src/main/res/xml/provider_paths.xml](#))

At this point, RosterMotor should resemble:

```
package com.commonsware.todo.ui.roster

import android.app.Application
```

SHARING THE REPORT

```
import android.content.Context
import android.net.Uri
import androidx.core.content.FileProvider
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.BuildConfig
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.report.RosterReport
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import java.io.File

private const val AUTHORITY = BuildConfig.APPLICATION_ID + ".provider"

data class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val isLoaded: Boolean = false,
    val filterMode: FilterMode = FilterMode.ALL
)

sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
    data class ShareReport(val doc: Uri) : Nav()
}

class RosterMotor(
    private val repo: ToDoRepository,
    private val report: RosterReport,
    private val context: Application,
    private val appScope: CoroutineScope
) : ViewModel() {
    private val _states = MutableStateFlow(RosterViewState())
    val states = _states.asStateFlow()
    private val _navEvents = MutableSharedFlow<Nav>()
    val navEvents = _navEvents.asSharedFlow()
    private var job: Job? = null

    init {
        load(FilterMode.ALL)
    }

    fun load(filterMode: FilterMode) {
        job?.cancel()

        job = viewModelScope.launch {
            repo.items(filterMode).collect {
```

SHARING THE REPORT

```
        _states.emit(RosterViewState(it, true, filterMode))
    }
}

fun save(model: ToDoModel) {
    viewModelScope.launch {
        repo.save(model)
    }
}

fun saveReport(doc: Uri) {
    viewModelScope.launch {
        report.generate(_states.value.items, doc)
        _navEvents.emit(Nav.ViewReport(doc))
    }
}

fun shareReport() {
    viewModelScope.launch {
        saveForSharing()
    }
}

private suspend fun saveForSharing() {
    withContext(Dispatchers.IO + appScope.coroutineContext) {
        val shared = File(context.cacheDir, "shared").also { it.mkdirs() }
        val reportFile = File(shared, "report.html")
        val doc = FileProvider.getUriForFile(context, AUTHORITY, reportFile)

        _states.value.let { report.generate(it.items, doc) }
        _navEvents.emit(Nav.ShareReport(doc))
    }
}
}
```

(from [T3o-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Also, ToDoApp should look like:

```
package com.commonsware.todo

import android.app.Application
import android.text.format.DateUtils
import com.commonsware.todo.repo.ToDoDatabase
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.report.RosterReport
import com.commonsware.todo.ui.SingleModelMotor
```

SHARING THE REPORT

```
import com.commonsware.todo.ui.roster.RosterMotor
import com.github.jknack.handlebars.Handlebars
import com.github.jknack.handlebars.Helper
import kotlinx.coroutinesCoroutineScope
import kotlinx.coroutinesSupervisorJob
import org.koin.android.ext.koin.androidApplication
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.core.qualifier.named
import org.koin.dsl.module
import java.time.Instant

class ToDoApp : Application() {
    private val koinModule = module {
        single(named("appScope")) { CoroutineScope(SupervisorJob()) }
        single { ToDoDatabase.newInstance(androidContext()) }
        single {
            ToDoRepository(
                get<ToDoDatabase>().todoStore(),
                get(named("appScope"))
            )
        }
        single {
            Handlebars().apply {
                registerHelper("dateFormat", Helper<Instant> { value, _ ->
                    DateUtils.getRelativeDateTimeString(
                        androidContext(),
                        value.toEpochMilli(),
                        DateUtils.MINUTE_IN_MILLIS,
                        DateUtils.WEEK_IN_MILLIS, 0
                    )
                })
            }
        }
        single { RosterReport(androidContext(), get(), get(named("appScope"))) }
        viewModel { RosterMotor(get(), get(), androidApplication(),
            get(named("appScope")) ) }
        viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin {
            androidLogger()
            androidContext(this@ToDoApp)
```

SHARING THE REPORT

```
    modules(koinModule)
}
}
```

(from [T30-Share/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Finally, our updated RosterListFragment should look like:

```
package com.commonsware.todo.ui.roster

import android.content.Intent
import android.net.Uri
import android.os.Bundle
import android.util.Log
import android.view.*
import android.widget.Toast
import androidx.activity.result.contract.ActivityResultContracts
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoRosterBinding
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel

private const val TAG = "ToDo"

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private val menuMap = mutableMapOf<FilterMode, MenuItem>()
    private var binding: TodoRosterBinding? = null

    private val createDoc =
        registerForActivityResult(ActivityResultContracts.CreateDocument()) {
            motor.saveReport(it)
        }

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

        setHasOptionsMenu(true)
    }
}
```

SHARING THE REPORT

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View = TodoRosterBinding.inflate(inflater, container, false)
    .also { binding = it }
    .root

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

    val adapter = RosterAdapter(
        layoutInflater,
        onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
        onRowClick = ::display
    )

    binding?.items?.apply {
        setAdapter(adapter)
        layoutManager = LinearLayoutManager(context)

        addItemDecoration(
            DividerItemDecoration(
                activity,
                DividerItemDecoration.VERTICAL
            )
        )
    }
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = if (state.isLoading) View.GONE else View.VISIBLE

            when {
                state.items.isEmpty() && state.filterMode == FileMode.ALL -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty_filtered)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }
}
```

SHARING THE REPORT

```
        menuMap[state.filterMode]?.isChecked = true
    }
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.navEvents.collect { nav ->
        when (nav) {
            is Nav.ViewReport -> viewReport(nav.doc)
            is Nav.ShareReport -> shareReport(nav.doc)
        }
    }
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)

    menuMap.apply {
        put(FilterMode.ALL, menu.findItem(R.id.all))
        put(FilterMode.COMPLETED, menu.findItem(R.id.completed))
        put(FilterMode.OUTSTANDING, menu.findItem(R.id.outstanding))
    }
}

menuMap[motor.states.value.filterMode]?.isChecked = true

super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
        R.id.all -> {
            item.isChecked = true
            motor.load(FilterMode.ALL)
            return true
        }
        R.id.completed -> {
            item.isChecked = true
        }
    }
}
```

SHARING THE REPORT

```
        motor.load(FilterMode.COMPLETED)
        return true
    }
    R.id.outstanding -> {
        item.isChecked = true
        motor.load(FilterMode.OUTSTANDING)
        return true
    }
    R.id.save -> {
        saveReport()
        return true
    }
    R.id.share -> {
        motor.shareReport()
        return true
    }
}

return super.onOptionsItemSelected(item)
}

private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}

private fun saveReport() {
    createDoc.launch("report.html")
}

private fun viewReport(uri: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_VIEW, uri)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    )
}

private fun shareReport(doc: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_SEND)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
            .setType("text/html")
            .putExtra(Intent.EXTRA_STREAM, doc)
    )
}
```

SHARING THE REPORT

```
}

private fun safeStartActivity(intent: Intent) {
    try {
        startActivity(intent)
    } catch (t: Throwable) {
        Log.e(TAG, "Exception starting $intent", t)
        Toast.makeText(requireActivity(), R.string.oops, Toast.LENGTH_LONG).show()
    }
}
```

(from [T3o-Share/Todo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/res/drawable/ic_share.xml](#)
- [app/src/main/res/menu/actions_roster.xml](#)
- [app/src/main/AndroidManifest.xml](#)
- [app/src/main/res/xml/provider_paths.xml](#)
- [app/src/main/res/values/strings.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)

Collecting a Preference

If there are aspects of your app that are user-configurable, you have two main options for allowing the user to configure them:

1. Integrate that configuration into your own UI
2. Set up a `PreferenceScreen`

A `PreferenceScreen` is a way to declare what sorts of configuration options your app has. The `PreferenceScreen` can then be used to generate a UI that resembles the Settings app and lets the user configure the items. That generated UI also handles storing the values in a `SharedPreferences` object, so you can use the values from within your app code.

In this tutorial, we will set up a `PreferenceScreen`, right now to collect just a single preference.



You can learn more about `SharedPreferences` and the preference UI system in the "Using Preferences" chapter of [*Elements of Android Jetpack!*](#)

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding a Dependency

There are framework classes related to preferences. However, most of them are deprecated, with replacements in the Jetpack components. To pull in those

COLLECTING A PREFERENCE

replacements, we need to add another dependency.

Add this line to the dependencies closure of your app/build.gradle file:

```
implementation "androidx.preference:preference-ktx:1.1.1"
```

(from [T31-Prefs/ToDo/app/build.gradle](#))

This pulls in the preference-ktx library, which will give us the Jetpack preference classes, along with some Kotlin extension functions related to preferences.

At this point, the dependencies closure should resemble:

```
dependencies {  
    implementation 'androidx.core:core-ktx:1.6.0'  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
    implementation "androidx.recyclerview:recyclerview:1.2.1"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "androidx.preference:preference-ktx:1.1.1"  
    implementation 'com.google.android.material:material:1.4.0'  
    implementation "io.insert-koin:koin-android:$koin_version"  
    implementation "com.github.jknack:handlebars:4.1.2"  
    implementation "androidx.room:room-runtime:$room_version"  
    implementation "androidx.room:room-ktx:$room_version"  
    kapt "androidx.room:room-compiler:$room_version"  
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'  
    testImplementation 'junit:junit:4.13.2'  
    testImplementation "org.mockito:mockito-inline:3.12.1"  
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"  
    testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'  
    androidTestImplementation "androidx.arch.core:core-testing:2.1.0"  
    androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'  
}
```

(from [T31-Prefs/ToDo/app/build.gradle](#))

This may look like a lot of libraries for a fairly trivial app. However, on the whole, Android app development has become very focused on libraries, and a production-grade app may have a lot more libraries than does this app.

Step #2: Defining a Preference Screen

Like our FileProvider configuration from [the preceding tutorial](#), a PreferenceScreen is defined as an XML resource in res/xml/. So, right-click over res/xml/ and choose “New” > “XML resource file” from the context menu. Fill in

COLLECTING A PREFERENCE

prefs for the name and leave the “Root element” alone. Then, click “OK” to create the resource.

Android Studio will bring up another graphical resource editor, this time set up to define preferences:

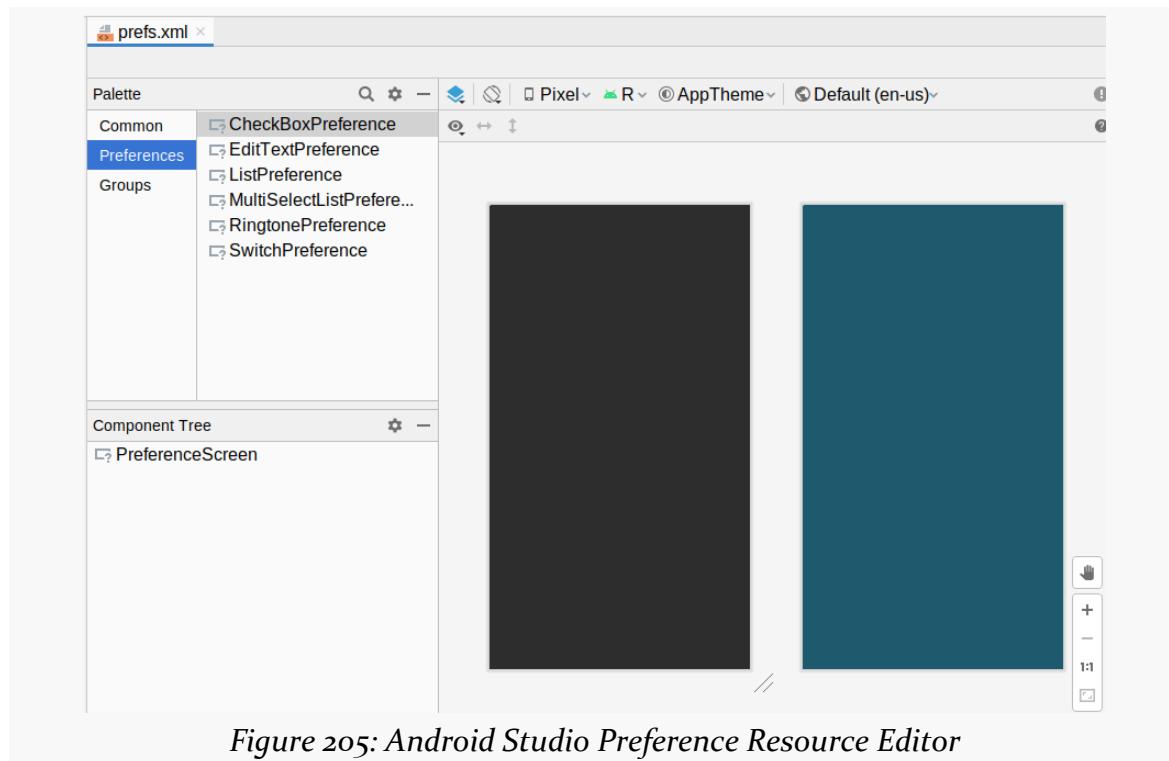


Figure 205: Android Studio Preference Resource Editor

Unfortunately, the Android Studio preference resource editor does not work all that well. So, switch over to the XML editor and replace your current XML with:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <EditTextPreference
        android:key="@string/web_service_url_key"
        android:selectAllOnFocus="true"
        android:title="@string/pref_url_title"
        app:defaultValue="@string/web_service_url_default" />

</PreferenceScreen>
```

(from [T31-Prefs/ToDo/app/src/main/res/xml/prefs.xml](#))

COLLECTING A PREFERENCE

The root element is a `PreferenceScreen`, which mostly contains different types of preferences.

Our one-and-only preference is an `EditTextPreference`. This will provide a `PreferenceScreen` entry that pops up a dialog with a field when the user taps on it. The user can adjust the preference value by typing in that field.

There are four attributes on our `EditTextPreference`. One of these — `android:selectAllOnFocus` — is actually an attribute available for `EditText`, indicating that the contents of the field should be selected automatically once the field gets the focus. This is a useful feature of an `EditText` for fields where you expect the user to replace the entire value much more often than they edit the existing value. `EditTextPreference` itself has a feature where it will accept many `EditText` attributes on the `EditTextPreference` element.

The other three attributes are ones that you will find on most preferences:

- `android:key` is the identifier of the preference. Unlike `android:id`, though, it can be whatever string you want — it is not limited to `@+id:/...` syntax.
- `android:title` is what the user sees on the screen when this preference is shown in the `PreferenceScreen`
- `app:defaultValue` is the default value to show in the field, if the user has not filled in their own value yet

To make this work, though, we need to add three more string resource. Edit your `res/values/strings.xml` file and add:

```
<string name="pref_url_title">Web service URL</string>
<string name="web_service_url_key">webServiceUrl</string>
<string name="web_service_url_default">https://commonsware.com/AndExplore/2.0/items.json</string>
```

(from [T31-Prefs/ToDo/app/src/main/res/values/strings.xml](#))

Step #3: Displaying Our Preference Screen

Now, we need some Kotlin code to arrange for that preference XML to get used. The typical approach is to use `PreferenceFragmentCompat`, which is a fragment class that knows how to work with the rest of the preference system to render the `PreferenceScreen`, collect preferences from the user, and save the changes.

However, this fragment does not match any of our existing `com.commonsware.todo.ui` sub-packages. So, right-click over the

COLLECTING A PREFERENCE

com.commonware.todo.ui package in the java/ directory, choose “New” > “Package” from the context menu, fill in com.commonware.todo.ui.prefs for the package name, and press **Enter** or **Return**.

Then, right-click over the new com.commonware.todo.ui.prefs package in the java/ directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in PrefsFragment, and choose “Class” for the kind. Press **Enter** or **Return** to create the class, giving you:

```
package com.commonware.todo.ui.prefs

class PrefsFragment {
```

Finally, replace that stub class with:

```
package com.commonware.todo.ui.prefs

import android.os.Bundle
import androidx.preference.PreferenceFragmentCompat
import com.commonware.todo.R

class PrefsFragment : PreferenceFragmentCompat() {
    override fun onCreatePreferences(state: Bundle?, rootKey: String?) {
        setPreferencesFromResource(R.xml.prefs, rootKey)
    }
}
```

(from [T31-Prefs/ToDo/app/src/main/java/com/commonware/todo/ui/prefs/PrefsFragment.kt](#))

Here, we are creating a subclass of PreferenceFragmentCompat. The only required function is `onCreatePreferences()`, where our job is to provide the details of the preferences that we wish to collect. For that, we can call `setPreferencesFromResource()`, indicating that we want to display the PreferenceScreen from `res/xml/prefs.xml`.

Step #4: Adding PrefsFragment to Our Navigation Graph

Just like our other fragments, we should add PrefsFragment to our navigation graph. The biggest difference is that it is not part of the existing fragment-to-fragment navigation flow, so we will need to set up a “global action” to allow MainActivity to

COLLECTING A PREFERENCE

display PrefsFragment when requested.

Open up res/navigation/nav_graph.xml and click the add-destination toolbar button (rectangle with green plus sign in the corner). You should see PrefsFragment as an option in the destination drop-down:

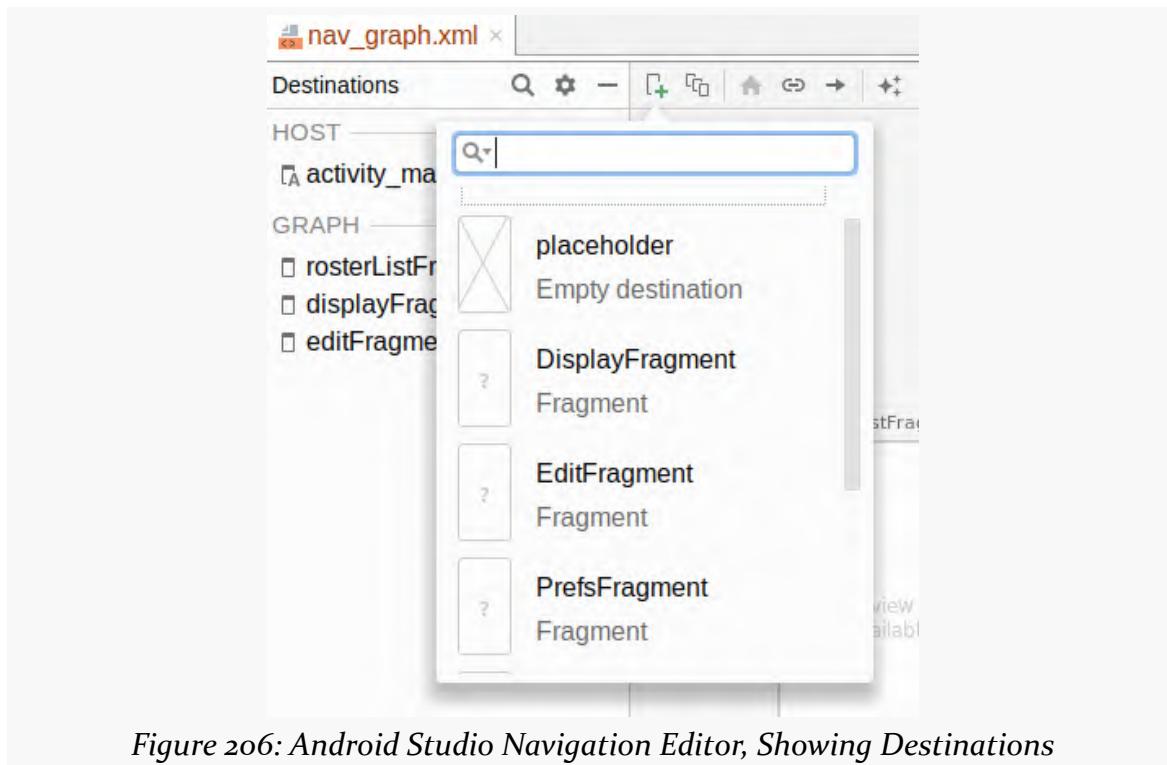


Figure 206: Android Studio Navigation Editor, Showing Destinations

Click on PrefsFragment, then drag its tile to some clean spot in the diagram.

COLLECTING A PREFERENCE

Then, right-click over the prefsFragment tile and choose “Add Action” > “Global” from the context menu. This global action gets represented by an arrow pointing from nowhere into the tile:



Figure 207: Android Studio Navigation Editor, Showing Global Action

COLLECTING A PREFERENCE

In the “Attributes” pane, with that arrow selected, you should see attributes for this global action. Set the “ID” to be editPrefs and leave the rest alone:

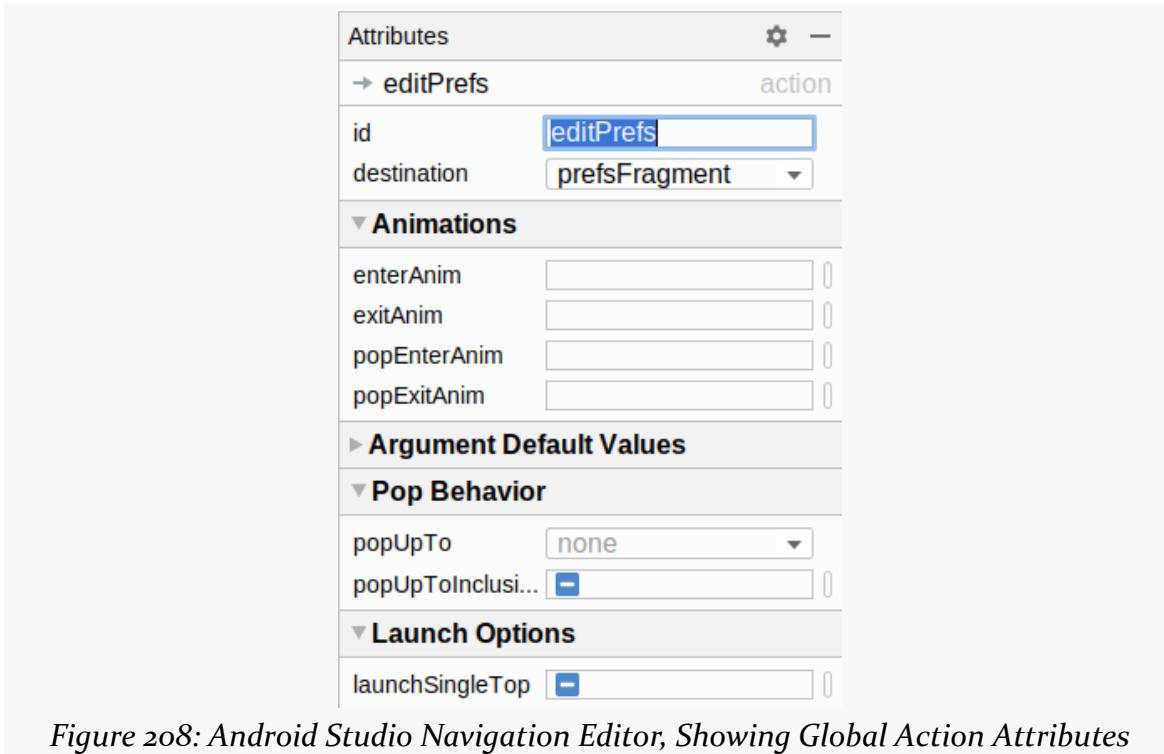


Figure 208: Android Studio Navigation Editor, Showing Global Action Attributes

Step #5: Navigating to Our Preference Screen

Now, we need to arrange to display that PrefsFragment. The first step is yet another app bar item, because we *love* app bar items!

COLLECTING A PREFERENCE

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Icon” button and search for `settings`:

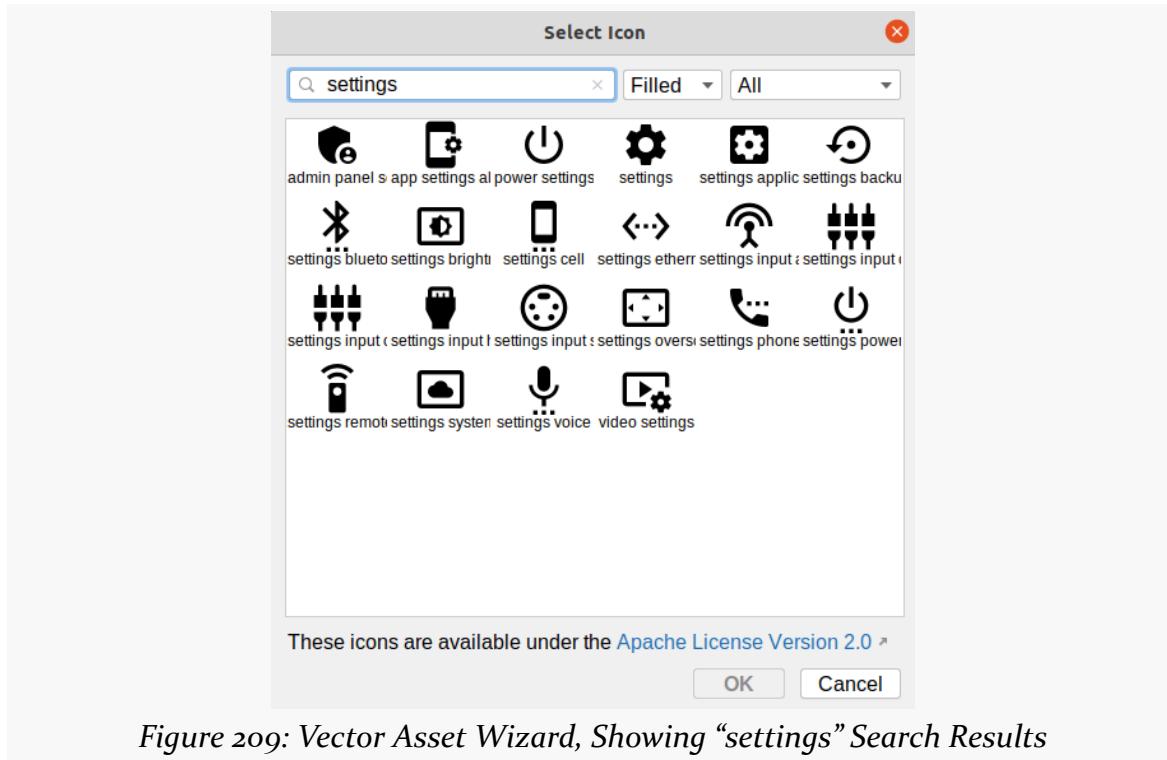


Figure 209: Vector Asset Wizard, Showing “settings” Search Results

Choose the “settings” icon and click “OK” to close up the icon selector. Change the icon’s name to `ic_settings`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into `res/drawable` instead. That is the desired icon, already set up for you.

Open up the `res/menu/actions.xml` resource file, and switch to the “Design” sub-tab. Drag a “Menu Item” from the “Palette” view into the Component Tree, slotting it before the existing “about” item.

In the Attributes view for this new item, assign it an ID of “settings”. Then, choose “never” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Click on `ic_settings` in the list of drawables, then click OK to accept that choice of icon.

COLLECTING A PREFERENCE

Then, click the “O” 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 **settings** as the resource name and “Settings” as the resource value. This time, we are not using the `menu_` prefix, as we are going to use this string somewhere else. Click OK to close the dialog.

Next, in the “All Attributes” section of the “Attributes” pane, find the `orderInCategory` attribute and set it to `90`. This will place it ahead of the “About” item (whose `orderInCategory` is set to `100`). And, both will appear after the items added by the fragments.

Then, switch back to the `res/navigation/nav_graph.xml` resource. Click on the `prefsFragment` item and in the “Label” field fill in `@string/settings`. This will have the title of our screen (as shown in our toolbar) match the menu item that we just added.

Finally, in `MainActivity`, replace the current `onOptionsItemSelected()` function with this:

```
override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
    R.id.about -> {
        startActivity(Intent(this, AboutActivity::class.java))
        true
    }
    R.id.settings -> {
        findNavController(R.id.nav_host).navigate(R.id.editPrefs)
        true
    }
    else -> super.onOptionsItemSelected(item)
}
```

(from [T31-Prefs/ToDo/app/src/main/java/com/commonsware/todo/ui/MainActivity.kt](#))

This adds a new branch for the `R.id.settings` case. There, we retrieve our NavController via `findNavController()` and ask to navigate using our new `editPrefs` action.

COLLECTING A PREFERENCE

At this point, if you run the project, you should see the new Settings app bar item:

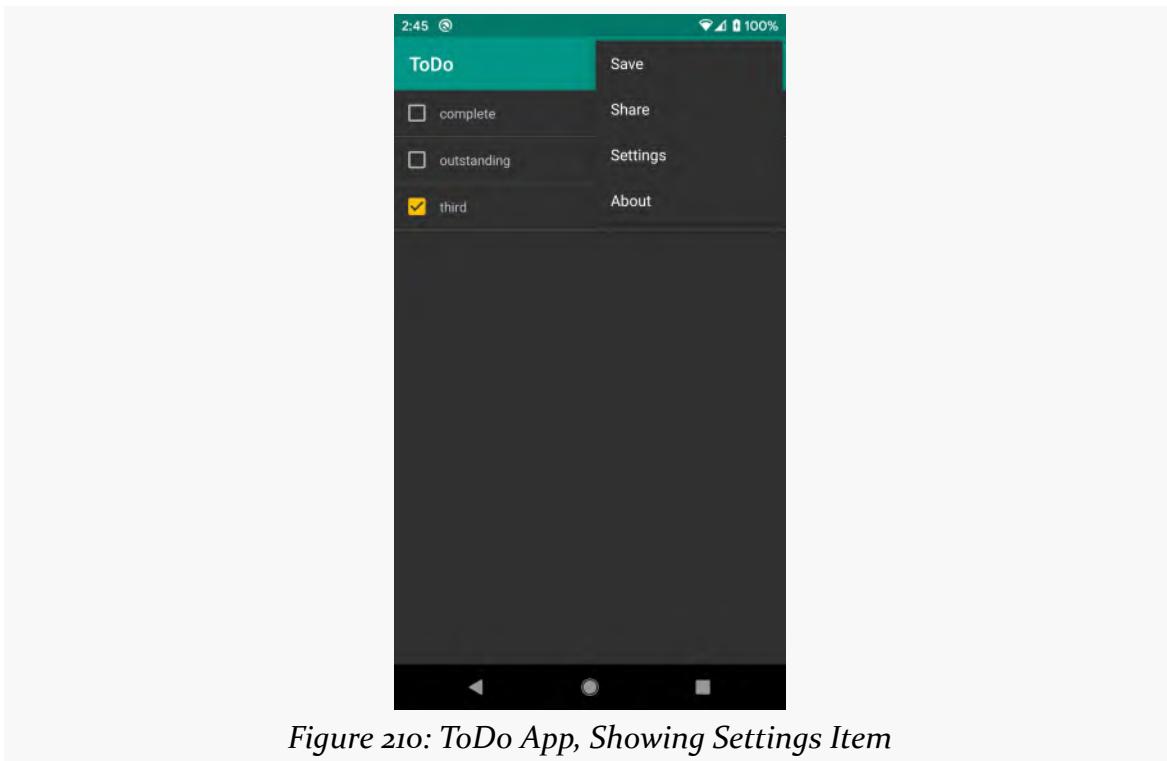


Figure 210: ToDo App, Showing Settings Item

COLLECTING A PREFERENCE

Clicking that will bring up the fairly boring `PrefsFragment`:

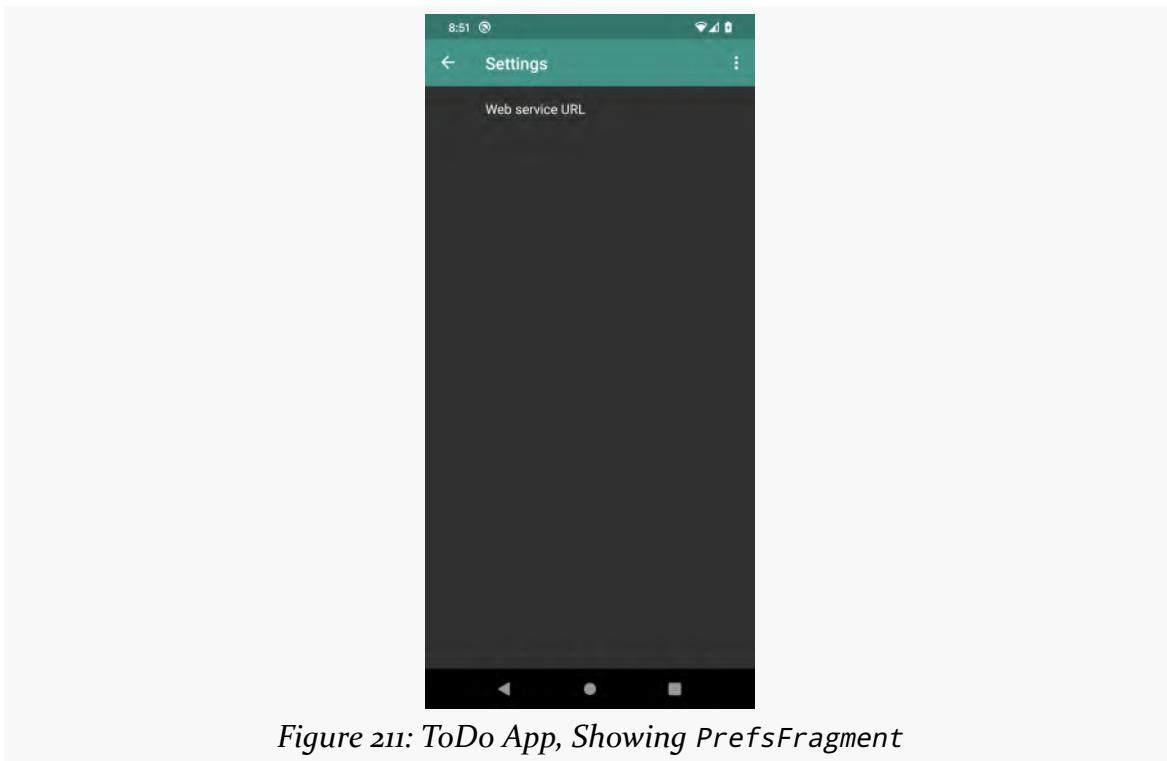


Figure 211: ToDo App, Showing PrefsFragment

COLLECTING A PREFERENCE

Tapping on the “Web service URL” row will bring up a dialog with a field containing our default value:

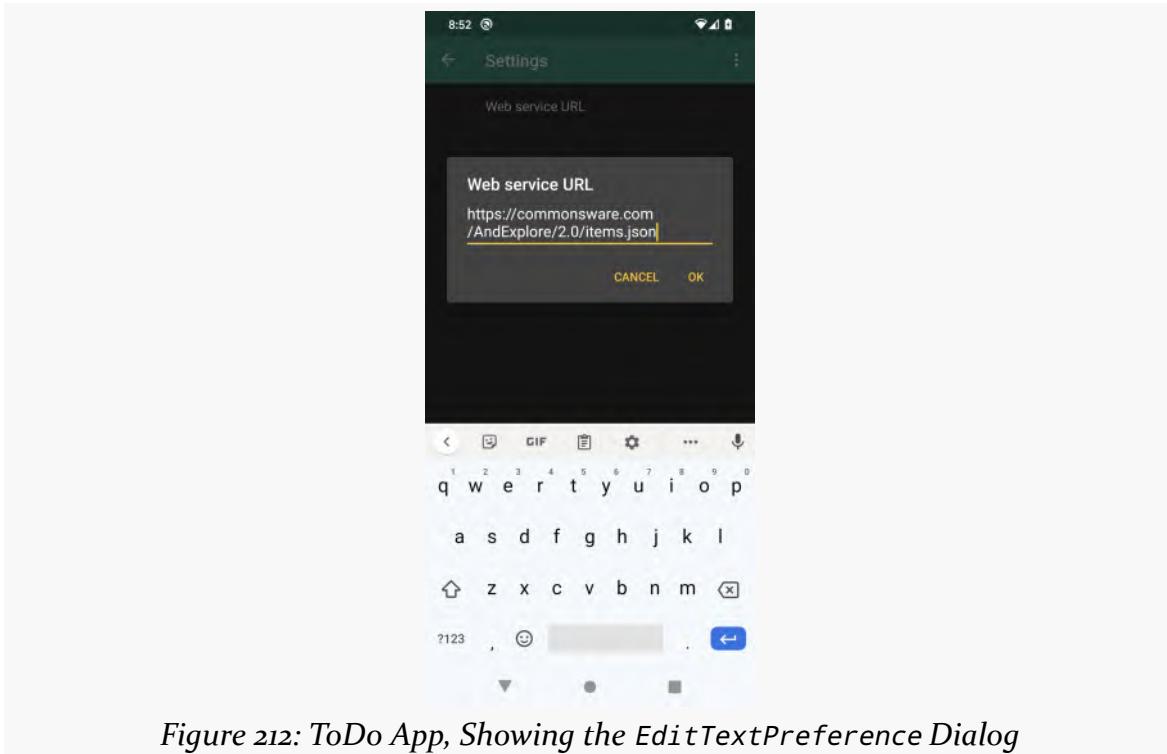


Figure 212: ToDo App, Showing the `EditTextPreference` Dialog

Right now, leave the value alone — just click BACK a few times to exit back to the main screen.

Final Results

Our revised app/build.gradle should resemble:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
    id 'androidx.navigation.safeargs.kotlin'  
    id 'kotlin-kapt'  
}  
  
android {  
    compileSdk 31  
  
    defaultConfig {
```

COLLECTING A PREFERENCE

```
applicationId "com.commonsware.todo"
minSdk 21
targetSdk 31
versionCode 1
versionName "1.0"

    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}

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

buildFeatures {
    viewBinding true
}

compileOptions {
    coreLibraryDesugaringEnabled true
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}

packagingOptions {
    exclude 'META-INF/AL2.0'
    exclude 'META-INF/LGPL2.1'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation "androidx.preference:preference-ktx:1.1.1"
    implementation 'com.google.android.material:material:1.4.0'
    implementation "io.insert-koin:koin-android:$koin_version"
    implementation "com.github.jknack:handlebars:4.1.2"
```

COLLECTING A PREFERENCE

```
implementation "androidx.room:room-runtime:$room_version"
implementation "androidx.room:room-ktx:$room_version"
kapt "androidx.room:room-compiler:$room_version"
coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
testImplementation 'junit:junit:4.13.2'
testImplementation "org.mockito:mockito-inline:3.12.1"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"
testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
}
```

(from [T31-Prefs/ToDo/app/build.gradle](#))

Our new prefs XML resource should contain:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <EditTextPreference
        android:key="@string/web_service_url_key"
        android:selectAllOnFocus="true"
        android:title="@string/pref_url_title"
        app:defaultValue="@string/web_service_url_default" />
</PreferenceScreen>
```

(from [T31-Prefs/ToDo/app/src/main/res/xml/prefs.xml](#))

Our updated strings resource should look like:

```
<resources>
    <string name="app_name">ToDo</string>
    <string name="msg_empty">Click the + icon to add a todo item!</string>
    <string name="msg_empty_filtered">Click the + icon to add a todo item, or change
    your filter to show other items</string>
    <string name="menu_about">About</string>
    <string name="is_completed">Item is completed</string>
    <string name="created_on">Created on:</string>
    <string name="menu_edit">Edit</string>
    <string name="desc">Description</string>
    <string name="notes">Notes</string>
    <string name="menu_save">Save</string>
    <string name="menu_add">Add</string>
    <string name="menu_delete">Delete</string>
    <string name="menu_filter">Filter</string>
```

COLLECTING A PREFERENCE

```
<string name="menu_filter_all">All</string>
<string name="menu_filter_completed">Completed</string>
<string name="menu_filter_outstanding">Outstanding</string>
<string name="oops">Sorry! Something went wrong!</string>
<string name="report_template"><![CDATA[<h1>To-Do Items</h1>
{{#this}}
<h2>{{description}}</h2>
<p>{{#completed}}<b>COMPLETED</b> &mdash; {{/completed}}Created on: {{dateFormat
createdOn}}</p>
<p>{{notes}}</p>
{{/this}}
]]></string>
<string name="menu_share">Share</string>
<string name="pref_url_title">Web service URL</string>
<string name="web_service_url_key">webServiceUrl</string>
<string name="web_service_url_default">https://commonsware.com/AndExplore/2.0/
items.json</string>
<string name="settings">Settings</string>
</resources>
```

(from [T31-Prefs/ToDo/app/src/main/res/values/strings.xml](#))

The new PrefsFragment should contain:

```
package com.commonsware.todo.ui.prefs

import android.os.Bundle
import androidx.preference.PreferenceFragmentCompat
import com.commonsware.todo.R

class PrefsFragment : PreferenceFragmentCompat() {
    override fun onCreatePreferences(state: Bundle?, rootKey: String?) {
        setPreferencesFromResource(R.xml.prefs, rootKey)
    }
}
```

(from [T31-Prefs/ToDo/app/src/main/java/com/commonsware/todo/ui/prefs/PrefsFragment.kt](#))

The revised nav_graph navigation resource should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/nav_graph.xml"
    app:startDestination="@+id/rosterListFragment">

    <fragment
        android:id="@+id/rosterListFragment"
```

COLLECTING A PREFERENCE

```
    android:name="com.commonsware.todo.ui.roster.RosterListFragment"
    android:label="@string/app_name">
    <action
        android:id="@+id/displayModel"
        app:destination="@+id/displayFragment" />
    <action
        android:id="@+id/createModel"
        app:destination="@+id/editFragment" >
        <argument
            android:name="modelId"
            android:defaultValue="@null" />
    </action>
</fragment>
<fragment
    android:id="@+id/displayFragment"
    android:name="com.commonsware.todo.ui.display.DisplayFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string" />
    <action
        android:id="@+id/editModel"
        app:destination="@+id/editFragment" />
</fragment>
<fragment
    android:id="@+id/editFragment"
    android:name="com.commonsware.todo.ui.edit.EditFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string"
        app:nullable="true" />
</fragment>
<fragment
    android:id="@+id/prefsFragment"
    android:name="com.commonsware.todo.ui.prefs.PrefsFragment"
    android:label="@string/settings" />
    <action android:id="@+id/editPrefs" app:destination="@+id/prefsFragment" />
</navigation>
```

(from [T31-Prefs/ToDo/app/src/main/res/navigation/nav_graph.xml](#))

The updated actions menu resource should look like:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
      xmlns:android="http://schemas.android.com/apk/res/android">
```

COLLECTING A PREFERENCE

```
<item
    android:id="@+id/settings"
    android:icon="@drawable/ic_settings"
    android:orderInCategory="90"
    android:title="@string/settings"
    app:showAsAction="never" />
<item
    android:id="@+id/about"
    android:icon="@drawable/ic_about"
    android:orderInCategory="100"
    android:title="@string/menu_about"
    app:showAsAction="never" />
</menu>
```

(from [T31-Prefs/ToDo/app/src/main/res/menu/actions.xml](#))

And the updated MainActivity should contain:

```
package com.commonsware.todo.ui

import android.content.Intent
import android.os.Bundle
import android.view.Menu
import android.view.MenuItem
import androidx.appcompat.app.AppCompatActivity
import androidx.navigation.findNavController
import androidx.navigation.fragment.findNavController
import androidx.navigation.ui.AppBarConfiguration
import androidx.navigation.ui.NavigationUI.navigateUp
import androidx.navigation.ui.setupActionBarWithNavController
import com.commonsware.todo.R
import com.commonsware.todo.databinding.ActivityMainBinding

class MainActivity : AppCompatActivity() {
    private lateinit var appBarConfiguration: AppBarConfiguration

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

        val binding = ActivityMainBinding.inflate(layoutInflater)

        setContentView(binding.root)
        setSupportActionBar(binding.toolbar)

        supportFragmentManager.findFragmentById(R.id.nav_host)?.findNavController()?.let
{ nav ->
    appBarConfiguration = AppBarConfiguration(nav.graph)
    setupActionBarWithNavController(nav, appBarConfiguration)
```

COLLECTING A PREFERENCE

```
    }

    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        menuInflater.inflate(R.menu.actions, menu)

        return super.onCreateOptionsMenu(menu)
    }

    override fun onOptionsItemSelected(item: MenuItem) = when (item.itemId) {
        R.id.about -> {
            startActivity(Intent(this, AboutActivity::class.java))
            true
        }
        R.id.settings -> {
            findNavController(R.id.nav_host).navigate(R.id.editPrefs)
            true
        }
        else -> super.onOptionsItemSelected(item)
    }

    override fun onSupportNavigateUp() =
        navigateUp(findNavController(R.id.nav_host), appBarConfiguration)
}
```

(from [T31-Prefs/ToDo/app/src/main/java/com/commonsware/todo/ui/MainActivity.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/build.gradle](#)
- [app/src/main/res/xml/prefs.xml](#)
- [app/src/main/res/values/strings.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/prefs/PrefsFragment.kt](#)
- [app/src/main/res/navigation/nav_graph.xml](#)
- [app/src/main/res/drawable/ic_settings.xml](#)
- [app/src/main/res/menu/actions.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/MainActivity.kt](#)

Contacting a Web Service

The URL that we collected in [the previous tutorial](#) is a Web service URL from which we can get to-do items... at least, in theory. In reality, it is a static JSON file pretending to be a Web service. And, since it is a static JSON file, we cannot implement a full synchronization routine, where we blend what is on the server and on the client into a unified depiction of what the state is of all of the to-do items.

However, we can implement a basic import operation. We can let the user request to import items from the server, and those that do not already exist can be added to our database and UI. So, in this tutorial, we will work on adding that capability to the app. Along the way, we will look at libraries for making HTTPS requests and for parsing JSON.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Adding Some Dependencies

Android has an HTTPS client API built in, but it is the ancient `HttpsURLConnection`, and its API leaves a lot to be desired. Similarly, Android has a couple of JSON parsers built in, but neither map JSON directly to your own objects — instead, they are classic manual parsers. None of these are especially popular.

Instead, we will use two more popular options:

- [OkHttp](#) is the most popular HTTPS client library, by far
- [Moshi](#) is a moderately popular JSON parser, from the same firm that created OkHttp

CONTACTING A WEB SERVICE

So, we need to add those to our list of dependencies. Add these three lines to the dependencies closure in `app/build.gradle`:

```
implementation "com.squareup.okhttp3:okhttp:4.9.1"
implementation "com.squareup.moshi:moshi:$moshi_version"
kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"
```

(from [T32-Internet/ToDo/app/build.gradle](#))

Moshi requires some special stuff to work with Kotlin. Specifically, we need a Kotlin annotation processor, one that will be able to code-generate some Moshi support classes for us. That is why the third line has `kapt` instead of `implementation` — we are pulling in a compile-time annotation processor, not adding a runtime dependency directly.

This will give you an error, as `moshi_version` is not yet defined. As before, we are using a constant to define the version, so we can have multiple dependencies with synchronized versions. Add a definition of `moshi_version` to the `ext` closure in the top-level `build.gradle` file:

```
moshi_version = "1.12.0"
```

(from [T32-Internet/ToDo/build.gradle](#))

Step #2: Requesting a Permission

Our app will be working directly with the Internet. For that, we need permission from the user.

Requesting permissions from the user starts with a `<uses-permission>` element in the manifest, identifying what it is that we want. For some permissions — those deemed to be “dangerous” — we also need to prompt the user at runtime to confirm whether they do indeed want to grant us this permission.

The permission that we need for Internet access — `android.permission.INTERNET` — is not a dangerous permission. So, all we need is the `<uses-permission>` element.

So, add this element as a child of the root `<manifest>` element in `AndroidManifest.xml`:

```
<uses-permission android:name="android.permission.INTERNET" />
```

(from [T32-Internet/ToDo/app/src/main/AndroidManifest.xml](#))

Step #3: Defining Our Response

Our “Web service” is going to send us JSON that looks like:

```
[  
  {  
    "id": "bce0dde0-5eee-0137-c042-38ca3ad2633d",  
    "description": "Write a JSON file containing to-do items",  
    "completed": true,  
    "notes": "Technically, this work was not completed when I wrote this, though it is completed now",  
    "created_on": "2019-05-22"  
  },  
  {  
    "id": "f42d74e8-6fd8-4eb1-a4fe-af1c1314573b",  
    "description": "Add a third object to this JSON file",  
    "completed": false,  
    "notes": "",  
    "created_on": "2019-05-22"  
  }  
]
```

(from [items.json](#))

This resembles our model objects, but it is not quite identical. Moreover, many times the maintainers of the Web service are not the same developers as those who maintain the Android app (let alone the iOS app, the Web app, etc.). The Web service API might change from time to time.

The recommended way of handling this is to treat the Web service data model as being distinct from the app’s data model, with conversions between them as needed. This is similar to how we have our Room entities defined separately from our models, so any changes in Room do not affect our core app logic. As it turns out, we are going to funnel our server responses into the database, so we will be focusing more on converting Web service responses into entities that we can attempt to insert into the database.

With that in mind, right-click over the `com.commonsware.todo.repo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `ToDoServerItem`, and choose “Class” for the kind. Press `Enter` or `Return` to create the class. Then, replace the class declaration with:

```
@JsonClass(generateAdapter = true)  
data class ToDoServerItem(  
  val description: String,  
  val id: String,  
  val completed: Boolean,
```

CONTACTING A WEB SERVICE

```
    val notes: String,  
    @Json(name = "created_on") val createdOn: Instant  
)
```

The properties of `ToDoServerItem` match that of the JSON that we will receive from the Web service, with one exception: the JSON has our creation date in a `created_on` property, and we would like to use lowerCamelCase formatting for our Kotlin property. The `@Json` annotation applied to the `createdOn` property tells Moshi that the `created_on` value in the JSON goes into this `createdOn` property on our class.

The `@JsonClass` annotation applied to `ToDoServerItem` overall indicates that we want Moshi to code-generate the code that can fill in a `ToDoServerItem` from a matching JSON object.

This will work, with one exception: Moshi knows only about standard Java/Kotlin primitive types and strings. In particular, Moshi knows nothing about `Instant` and knows nothing about how to take a value like "2019-05-22" and convert it into a `Instant`. For that, we need to create an adapter class.

So, below `ToDoServerItem` in the same file, add this code:

```
private val FORMATTER = DateTimeFormatter.ISO_INSTANT  
  
class MoshiInstantAdapter {  
    @ToJson  
    fun toJson(date: Instant) = FORMATTER.format(date)  
  
    @FromJson  
    fun fromJson(dateString: String): Instant =  
        FORMATTER.parse(dateString, Instant::from)  
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoServerItem.kt](#))

A Moshi type adapter is simply a class with two functions:

- One with the `@ToJson` annotation that takes a data type and returns a string representation suitable for use in a JSON property
- One with the `@FromJson` annotation that takes the string representation and returns the corresponding object in that data type

In this case, we are using `DateTimeFormatter` to convert a `Instant` to and from a string representation, specifically using the representation found in the Web

service's JSON file.

Step #4: Retrieving the Items

Now, we can add some code that will download the JSON and convert it into a list of `ToDoServerItem` objects.

Right-click over the `com.commonware.todo.repo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `ToDoRemoteDataSource`, and choose “Class” for the kind. Press `Enter` or `Return` to create the class. Then, replace the class contents with:

```
package com.commonware.todo.repo

import com.squareup.moshi.JsonAdapter
import com.squareup.moshi.Moshi
import com.squareup.moshi.Types
import java.io.IOException
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import okhttp3.OkHttpClient
import okhttp3.Request

class ToDoRemoteDataSource(private val ok: OkHttpClient) {
    private val moshi = Moshi.Builder().add(MoshiInstantAdapter()).build()
    private val adapter: JsonAdapter<List<ToDoServerItem>> = moshi.adapter(
        Types.newParameterizedType(
            List::class.java,
            ToDoServerItem::class.java
        )
    )

    suspend fun load(url: String) = withContext(Dispatchers.IO) {
        val response = ok.newCall(Request.Builder().url(url).build()).execute()

        if (response.isSuccessful) {
            response.body?.let { adapter.fromJson(it.source()) }
                ?: throw IOException("No response body: $response")
        } else {
            throw IOException("Unexpected HTTP response code: ${response.code}")
        }
    }
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonware/todo/repo/ToDoRemoteDataSource.kt](#))

CONTACTING A WEB SERVICE

This class has a `load()` function that orchestrates our work for downloading and parsing the JSON. This will involve network I/O, so `load()` is defined as a suspend function and it uses `withContext(Dispatchers.IO)` to use the coroutine system to run this code on a background thread.

First, we need to download the JSON, and for that, we use OkHttp. Our constructor gets an `OkHttpClient`, which is our entry point for using OkHttp. `load()` gets the URL of the JSON as a parameter. We then:

- Wrap that URL in an OkHttp Request object (`Request.Builder().url(url).build()`)
- Tell OkHttp to create a `Call` object representing our request (`newCall()`)
- Execute the HTTP request on the current thread (`execute()`)

This will make the connection to the server and try to download the JSON. We get a `Response` object back which (hopefully) contains our JSON along with other bits of information from the Web service, such as an HTTP response code (e.g., `200` for an “OK” response). We check to see if the `Response` is successful by checking the `isSuccessful` property. If it was not successful, we throw an exception indicating the nature of the problem (e.g., our URL is wrong and we got a `404` response from the server).

We then check to see if the response has a body (`body()`) — this represents the JSON itself. If, for some reason, we do not have a body, we throw an exception to indicate that fact.

Finally, if we have a successful response and it has a body, we need to try to parse the JSON. For that, we use Moshi. Our `ToDoRemoteDataSource` has a `moshi` property which is a `Moshi` object, created using `Moshi.Builder`. In our case, we teach Moshi how to handle `Calendar` properties by adding our `MoshiInstantAdapter()` to the `Moshi` instance.

By and large, Moshi is a series of adapter classes. Some we write ourselves, such as `MoshiInstantAdapter()`. Some are code-generated for us at compile time, such as the adapter for `ToDoServerItem` that we requested via the

`@JsonClass(generateAdapter = true)` annotation that we placed on the `ToDoServerItem` class. And some are code-generated for us at runtime, such as the `adapter` property that we have in `ToDoRemoteDataSource`. That builds a `JsonAdapter` that knows how to parse JSON into a `List` of `ToDoServerItem` objects. In `load()`, we pass our JSON (`source()` called on the `response.body()` object) to this `JsonAdapter`, and it will return our `List` of `ToDoRemoteDataSource` objects... or

CONTACTING A WEB SERVICE

will throw an exception if there is some parsing problem.

Kotlin does not use Java-style checked exceptions, but it is obvious that we have multiple possible exceptions coming from `load()`. Given that we are attempting to download data from the Internet, there are *lots* of ways that this can go wrong, all of which will lead to exceptions.

Step #5: Updating the Local Items

Now we need to integrate `ToDoRemoteDataSource` into the rest of the app. `ToDoRepository` should be the one to do that, as a repository is supposed to insulate the GUI code from this sort of external interaction.

So, `ToDoRepository` needs an instance of `ToDoRemoteDataSource`. We could have `ToDoRepository` create its own instance, but it is better to use Koin — that way, we can use a mock `ToDoRemoteDataSource` in testing.

So, in `ToDoApp`, add these two lines to the existing `koinModule` declaration:

```
single { OkHttpClient.Builder().build() }
single { ToDoRemoteDataSource(get()) }
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

The first line creates a singleton instance of `OkHttpClient` (using an `OkHttpClient.Builder`), while the second line creates a singleton instance of `ToDoRemoteDataSource`.

Next, add a `ToDoRemoteDataSource` to the `ToDoRepository` constructor:

```
class ToDoRepository(
    private val store: ToDoEntity.Store,
    private val appScope: CoroutineScope,
    private val remoteDataSource: ToDoRemoteDataSource
) {
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

That then requires us to update its corresponding code in the Koin configuration in `ToDoApp`, adding a second `get()` call to pull in the `ToDoRemoteDataSource`:

CONTACTING A WEB SERVICE

```
single {
    ToDoRepository(
        get<ToDoDatabase>().todoStore(),
        get(named("appScope")),
        get()
    )
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

The end goal is that we want to get these new to-do items into our database, if those items are not there already. More importantly, if they *are* already in our database, we want to leave the database alone, as we may have local changes to the items that we do not want to overwrite. However, our `save()` function in `ToDoEntity.Store` is set up to replace existing items:

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun save(vararg entities: ToDoEntity)
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

That is great for user edits, but it is not what we want for our server-defined items. So, add this `importItems()` function to `ToDoEntity.Store`:

```
@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun importItems(entities: List<ToDoEntity>)
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

This has two differences when compared with `save()`:

- It uses `OnConflictStrategy.IGNORE` to say “if this item already exists based on the primary key, skip the insert operation for that item”
- It uses a `List` of entities rather than `varargs`

The function is named `importItems()`, rather than just `import()`, because `import` is a keyword in Java/Kotlin. While we can still have an `import()` function, we cannot have fields or properties named `import`. To avoid this sort of collision, we are using `importItems` as the name of this function instead of `import`.

To use `importItems()`, we need a way to map `ToDoServerItem` objects to `ToDoEntity` objects. So, add this `toEntity()` function to `ToDoServerItem`:

CONTACTING A WEB SERVICE

```
fun toEntity(): ToDoEntity {
    return ToDoEntity(
        id = id,
        description = description,
        isCompleted = completed,
        notes = notes,
        createdOn = createdOn
    )
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoServerItem.kt](#))

This just does a property-level mapping of the `ToDoServerItem` to the corresponding `ToDoEntity` definition.

Now, we can glue all of this together. Add this `importItems()` function to `ToDoRepository`:

```
suspend fun importItems(url: String) {
    withContext(appScope.coroutineContext) {
        store.importItems(remoteDataSource.load(url).map { it.toEntity() })
    }
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

Here, we:

- Take a URL to our JSON as a parameter
- Use the `ToDoRemoteDataSource` to get the list of `ToDoServerItem` objects
- Use `map()` and `toEntity()` to convert that list into a list of `ToDoEntity` objects
- Use `importItems()` on `ToDoEntity.Store` to insert any new items into our database, skipping existing ones

Since both `load()` and `importItems()` (on `ToDoEntity.Store`) are `suspend` functions, we use `suspend` on `importItems()` in `ToDoRepository`. And, we use our `appScope` so the import will proceed even if the user exits our UI while that import is going on.

So, now our repository knows how to get items and import them into our database.

Step #6: Fixing the Existing Tests

At this point, some of our tests are broken, due to changes to the `ToDoRepository` constructor.

In `RosterListFragmentTest` (in the `androidTest/ source set`), in the `setUp()` function, replace our existing `ToDoRepository` setup with:

```
repo = ToDoRepository(  
    db.todoStore(),  
    appScope,  
    ToDoRemoteDataSource(OkHttpClient())  
)
```

(from [T32-Internet/ToDo/app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

This creates a valid `ToDoRemoteDataSource`, using a one-off copy of an `OkHttpClient`. Since we are not testing the import process here, these objects will not be used — they are just here to satisfy the compiler.

Then, in `ToDoRepositoryTest` (in the `androidTest/ source set`), add a new property for an implementation of `ToDoRemoteDataSource`:

```
private val remoteDataSource = ToDoRemoteDataSource(OkHttpClient())
```

(from [T32-Internet/ToDo/app/src/androidTest/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

We can use a real implementation of `ToDoRemoteDataSource` and `OkHttpClient` here, as neither of those depend on Android — they work fine in a regular Kotlin/JVM environment.

Then, modify each `underTest` in `ToDoRepositoryTest` to pass that `ToDoRemoteDataSource` to our `ToDoRepository` constructor:

```
val underTest = ToDoRepository(db.todoStore(), this, remoteDataSource)
```

(from [T32-Internet/ToDo/app/src/androidTest/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

There are three of these, one for each test function.

And, if you run `RosterListFragmentTest` and `ToDoRepositoryTest` now, they should pass.

Step #7: Retrieving Our Preference

All through this work, we have been passing around a URL as a parameter. We are getting the URL from the user in our `PrefsFragment`, but we need a way to get that value (or a default value) into our main code. And, since this involves disk I/O, we should set up another repository with a suspend function that can handle loading that data for us.

Right-click over the `com.commonware.todo.repo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `PrefsRepository`, and choose “Class” for the kind. Press `Enter` or `Return` to create the class. Then, replace the class contents with:

```
package com.commonware.todo.repo

import android.content.Context
import androidx.preference.PreferenceManager
import com.commonware.todo.R
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext

class PrefsRepository(context: Context) {
    private val prefs = PreferenceManager.getDefaultSharedPreferences(context)
    private val webServiceUrlKey = context.getString(R.string.web_service_url_key)
    private val defaultWebServiceUrl =
        context.getString(R.string.web_service_url_default)

    suspend fun loadWebServiceUrl(): String = withContext(Dispatchers.IO) {
        prefs.getString(webServiceUrlKey, defaultWebServiceUrl) ?: defaultWebServiceUrl
    }
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonware/todo/repo/PrefsRepository.kt](#))

Given a `Context` constructor parameter, we set up three properties:

- `prefs`, which is the `SharedPreferences` object that is used by our `PreferenceScreen`
- `webServiceUrlKey`, which is the name of the preference that we want
- `defaultWebServiceUrl`, which is the default URL to use, if the user did not override it in the `PrefsFragment`

`SharedPreferences` gives us read/write access to the preferences. Those preferences are stored on disk in an XML file. The first time we try reading (or writing) a preference, the `SharedPreferences` will load that XML file into memory. Therefore, the `loadWebServiceUrl()` function is a suspend function, so we ensure that loading and parsing that XML happens on a background thread.

CONTACTING A WEB SERVICE

To read a preference, you call a typed getter method, such as `getString()`, on the `SharedPreferences` object. This takes two parameters:

- The key under which the preference is stored, which should match the key that you specified in your `PreferenceScreen`; and
- The default value to return if the user has not supplied a preference value yet via `PrefsFragment`

`getString()` is marked as potentially returning `null`. That is because you could pass `null` as the default value, in which case `getString()` will return `null` if there is no value for the preference defined yet. `getString()` should not return `null` if you provide a non-`null` default value... but the Kotlin compiler has no way of knowing this. Since we need *some* URL to try, `loadWebServiceUrl()` is set up to return `String`, not `String?`. So we cannot just return the `String?` that we get back from `getString()`. We could use the Kotlin `!!` operator to force the type to be non-nullable. Here, we use the Elvis operator to say “OK, if `getString()` returns `null` unexpectedly, use our default value”.

Then, go into `ToDoApp` and add another line to our module closure:

```
single { PrefsRepository(androidContext()) }
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This will make a `PrefsRepository` available to other components via Koin.

Step #8: Offering the Download Option

At this point, `ToDoRepository` knows how to import JSON-encoded to-do items retrieved from a URL, and `PrefsRepository` knows how to give us that URL. Now, we need to add a way for the user to trigger this work — in our case, we will put an option in `RosterListFragment` for that.

So, let’s update `RosterMotor` to be able to import our to-do items. First, update the `RosterMotor` constructor to take a `PrefsRepository` along with all of its other parameters:

```
class RosterMotor(  
    private val repo: ToDoRepository,  
    private val report: RosterReport,  
    private val context: Application,
```

CONTACTING A WEB SERVICE

```
private val appScope: CoroutineScope,  
private val prefs: PrefsRepository  
) : ViewModel() {
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

That will also require us to add another `get()` to the line in `ToDoApp` where we are creating our `RosterMotor` instances:

```
viewModel {  
    RosterMotor(  
        get(),  
        get(),  
        androidApplication(),  
        get(named("appScope")),  
        get()  
    )  
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Then, add this function to `RosterMotor`:

```
fun importItems() {  
    viewModelScope.launch {  
        repo.importItems(prefs.loadWebServiceUrl())  
    }  
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This just calls `loadWebServiceUrl()` on our `PrefsRepository` and passes the result to `importItems()` on the `ToDoRepository`. Since both of those are suspend functions, that code is wrapped in a `viewModelScope` coroutine. We do not need to do anything here to update our viewstate, though — Room will deliver fresh results to us after the import, the same way it does after `EditFragment` modifies our data.

Now, we need to arrange to call that `importItems()` function... triggered by yet another app bar item.

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Icon” button and search for download. Choose the “cloud download” icon and click “OK” to close up the icon selector. Change the icon’s name to `ic_download`. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

CONTACTING A WEB SERVICE

If the icon selector did not open, that may be due to [this Arctic Fox bug](#). Instead, just close up the Vector Asset wizard, and download [this file](#) into res/drawable instead. That is the desired icon, already set up for you.

Open up the res/menu/actions_roster.xml resource file, and switch to the graphical designer. Drag an “Item” from the “Palette” view into the Component Tree, slotting it after the other items.

In the Attributes view for this new item, assign it an ID of “importItems”. Then, choose “never” for the “showAsAction” option. Next, click on the “O” button next to the “icon” field. This will bring up an drawable resource selector. Click on ic_download in the list of drawables, then click OK to accept that choice of icon.

Then, click the “O” 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_import as the resource name and “Import” as the resource value.

Finally, in RosterListFragment, add another branch to the when in onOptionsItemSelected() to handle our importItems case:

```
R.id.importItems -> {
    motor.importItems()
    return true
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Now, if you run the app, you should see the new “Import” app bar item in the overflow. If you click it, you will get a couple of new entries in your list of to-do items, reflecting the JSON shown earlier in this chapter. And, if you choose “Import” again... nothing will happen, as those items already exist in the local database, so they are ignored on a subsequent import. This assumes that your device has Internet access and the author of the book has not accidentally deleted the “Web service” that we are trying to access.

Final Results

The top-level build.gradle file should look a bit like:

```
buildscript {
    ext.nav_version = '2.3.5'
```

CONTACTING A WEB SERVICE

```
repositories {  
    google()  
    mavenCentral()  
}  
  
dependencies {  
    classpath 'com.android.tools.build:gradle:7.0.2'  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"  
    classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"  
}  
}  
  
task clean(type: Delete) {  
    delete rootProject.buildDir  
}  
  
ext {  
    koin_version = "3.1.2"  
    moshi_version = "1.12.0"  
    room_version = "2.3.0"  
}
```

(from [T32-Internet/ToDo/build.gradle](#))

app/build.gradle should contain:

```
plugins {  
    id 'com.android.application'  
    id 'kotlin-android'  
    id 'androidx.navigation.safeargs.kotlin'  
    id 'kotlin-kapt'  
}  
  
android {  
    compileSdk 31  
  
    defaultConfig {  
        applicationId "com.commonsware.todo"  
        minSdk 21  
        targetSdk 31  
        versionCode 1  
        versionName "1.0"  
  
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    }  
  
    buildTypes {
```

CONTACTING A WEB SERVICE

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

buildFeatures {
    viewBinding true
}

compileOptions {
    coreLibraryDesugaringEnabled true
    sourceCompatibility JavaVersion.VERSION_1_8
    targetCompatibility JavaVersion.VERSION_1_8
}

kotlinOptions {
    jvmTarget = '1.8'
}

packagingOptions {
    exclude 'META-INF/AL2.0'
    exclude 'META-INF/LGPL2.1'
}
}

dependencies {
    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'
    implementation "androidx.recyclerview:recyclerview:1.2.1"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation "androidx.preference:preference-ktx:1.1.1"
    implementation 'com.google.android.material:material:1.4.0'
    implementation "io.insert-koin:koin-android:$koin_version"
    implementation "com.github.jknack:handlebars:4.1.2"
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-ktx:$room_version"
    implementation "com.squareup.okhttp3:okhttp:4.9.1"
    implementation "com.squareup.moshi:moshi:$moshi_version"
    kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"
    kapt "androidx.room:room-compiler:$room_version"
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'
    testImplementation 'junit:junit:4.13.2'
    testImplementation "org.mockito:mockito-inline:3.12.1"
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"
```

CONTACTING A WEB SERVICE

```
testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
androidTestImplementation 'androidx.test.ext:junit:1.1.3'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
}
```

(from [T32-Internet/ToDo/app/build.gradle](#))

The manifest overall should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.commonsware.todo">

    <uses-permission android:name="android.permission.INTERNET" />

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

    <application
        android:name=".ToDoApp"
        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/Theme.ToDo">
        <activity
            android:name=".ui.AboutActivity"
            android:exported="true" />
        <activity
            android:name=".ui.MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <provider
            android:name="androidx.core.content.FileProvider"
            android:authorities="${applicationId}.provider"
            android:exported="false"
            android:grantUriPermissions="true">
```

CONTACTING A WEB SERVICE

```
<meta-data
    android:name="android.support.FILE_PROVIDER_PATHS"
    android:resource="@xml/provider_paths" />
</provider>
</application>

</manifest>
```

(from [T32-Internet/ToDo/app/src/main/AndroidManifest.xml](#))

Our new `ToDoServerItem.kt` should contain:

```
package com.commonsware.todo.repo

import android.annotation.SuppressLint
import com.squareup.moshi.FromJson
import com.squareup.moshi.Json
import com.squareup.moshi.JsonClass
import com.squareup.moshi.ToJson
import java.time.Instant
import java.time.format.DateTimeFormatter

@JsonClass(generateAdapter = true)
data class ToDoServerItem(
    val description: String,
    val id: String,
    val completed: Boolean,
    val notes: String,
    @Json(name = "created_on") val createdOn: Instant
) {
    fun toEntity(): ToDoEntity {
        return ToDoEntity(
            id = id,
            description = description,
            isCompleted = completed,
            notes = notes,
            createdOn = createdOn
        )
    }
}

private val FORMATTER = DateTimeFormatter.ISO_INSTANT

class MoshiInstantAdapter {
    @ToJson
    fun toJson(date: Instant) = FORMATTER.format(date)

    @FromJson
```

CONTACTING A WEB SERVICE

```
fun fromJson(dateString: String): Instant =  
    FORMATTER.parse(dateString, Instant::from)  
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoServerItem.kt](#))

ToDoRepository should resemble:

```
package com.commonsware.todo.repo  
  
import kotlinx.coroutines.CoroutineScope  
import kotlinx.coroutines.flow.Flow  
import kotlinx.coroutines.flow.map  
import kotlinx.coroutines.withContext  
  
enum class FilterMode { ALL, OUTSTANDING, COMPLETED }  
  
class ToDoRepository(  
    private val store: ToDoEntity.Store,  
    private val appScope: CoroutineScope,  
    private val remoteDataSource: ToDoRemoteDataSource  
) {  
    fun items(filterMode: FilterMode = FilterMode.ALL): Flow<List<ToDoModel>> =  
        filteredEntities(filterMode).map { all -> all.map { it.toModel() } }  
  
    private fun filteredEntities(filterMode: FilterMode) = when (filterMode) {  
        FilterMode.ALL -> store.all()  
        FilterMode.OUTSTANDING -> store.filtered(isCompleted = false)  
        FilterMode.COMPLETED -> store.filtered(isCompleted = true)  
    }  
  
    fun find(id: String?): Flow<ToDoModel?> = store.find(id).map { it?.toModel() }  
  
    suspend fun save(model: ToDoModel) {  
        withContext(appScope.coroutineContext) {  
            store.save(ToDoEntity(model))  
        }  
    }  
  
    suspend fun delete(model: ToDoModel) {  
        withContext(appScope.coroutineContext) {  
            store.delete(ToDoEntity(model))  
        }  
    }  
  
    suspend fun importItems(url: String) {  
        withContext(appScope.coroutineContext) {  
            store.importItems(remoteDataSource.load(url).map { it.toEntity() })  
        }  
    }  
}
```

CONTACTING A WEB SERVICE

```
    }
}
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

ToDoEntity should contain:

```
package com.commonsware.todo.repo

import androidx.room.*
import kotlinx.coroutines.flow.Flow
import java.time.Instant
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Instant = Instant.now(),
    val isCompleted: Boolean = false
) {
    constructor(model: ToDoModel) : this(
        id = model.id,
        description = model.description,
        isCompleted = model.isCompleted,
        notes = model.notes,
        createdOn = model.createdOn
    )

    fun toModel(): ToDoModel {
        return ToDoModel(
            id = id,
            description = description,
            isCompleted = isCompleted,
            notes = notes,
            createdOn = createdOn
        )
    }
}

@Dao
interface Store {
    @Query("SELECT * FROM todos ORDER BY description")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE isCompleted = :isCompleted ORDER BY
```

CONTACTING A WEB SERVICE

```
    description")
    fun filtered(isCompleted: Boolean): Flow<List<ToDoEntity>>
}

@Query("SELECT * FROM todos WHERE id = :modelId")
fun find(modelId: String?): Flow<ToDoEntity?>

@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun save(vararg entities: ToDoEntity)

@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun importItems(entities: List<ToDoEntity>)

@Delete
suspend fun delete(vararg entities: ToDoEntity)
}

}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

The repaired RosterListFragmentTest should look like:

```
package com.commonsware.todo.ui.roster

import androidx.test.core.app.ActivityScenario
import androidx.test.espresso.Espresso.onView
import androidx.test.espresso.assertion.ViewAssertions.matches
import androidx.test.espresso.matcher.ViewMatchers.hasChildCount
import androidx.test.espresso.matcher.ViewMatchers.withId
import androidx.test.platform.app.InstrumentationRegistry
import com.commonsware.todo.R
import com.commonsware.todo.repo.ToDoDatabase
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRemoteDataSource
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.ui.MainActivity
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.runBlocking
import okhttp3.OkHttpClient
import org.junit.Before
import org.junit.Test
import org.koin.core.context.loadKoinModules
import org.koin.dsl.module

class RosterListFragmentTest {
    private lateinit var repo: ToDoRepository
    private val items = listOf(
        ToDoModel("this is a test"),
        ToDoModel("another test")
    )
    private val fragment by fragmentFactory()
}
```

CONTACTING A WEB SERVICE

```
ToDoModel("this is another test"),
ToDoModel("this is... wait for it... yet another test")
)

@Before
fun setUp() {
    val context = InstrumentationRegistry.getInstrumentation().targetContext
    val db = ToDoDatabase.newTestInstance(context)
    val appScope = CoroutineScope( SupervisorJob() )

    repo = ToDoRepository(
        db.todoStore(),
        appScope,
        ToDoRemoteDataSource(OkHttpClient())
    )

    loadKoinModules(module {
        single { repo }
    })

    runBlocking { items.forEach { repo.save(it) } }
}

@Test
fun testListContents() {
    ActivityScenario.launch(MainActivity::class.java)

    onView(withId(R.id.items)).check(matches(hasChildCount(3)))
}
}
```

(from [T32-Internet/ToDo/app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

And the fixed ToDoRepositoryTest should contain:

```
package com.commonsware.todo.repo

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import androidx.test.platform.app.InstrumentationRegistry
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.launch
import kotlinx.coroutines.test.runBlockingTest
import okhttp3.OkHttpClient
import org.hamcrest.Matchers.empty
import org.hamcrest.Matchers.equalTo
import org.hamcrest.collection.IsIterableContainingInOrder.contains
```

CONTACTING A WEB SERVICE

```
import org.hamcrest.MatcherAssert.assertThat
import org.junit.Rule
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class ToDoRepositoryTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    private val context = InstrumentationRegistry.getInstrumentation().targetContext
    private val db = ToDoDatabase.newTestInstance(context)
    private val remoteDataSource = ToDoRemoteDataSource(OkHttpClient())

    @Test
    fun canAddItems() = runBlockingTest {
        val underTest = ToDoRepository(db.todoStore(), this, remoteDataSource)
        val results = mutableListOf<List<ToDoModel>>()

        val itemsJob = launch {
            underTest.items().collect { results.add(it) }
        }

        assertThat(results.size, equalTo(1))
        assertThat(results[0], empty())

        val testModel = ToDoModel("test model")

        underTest.save(testModel)

        assertThat(results.size, equalTo(2))
        assertThat(results[1], contains(testModel))
        assertThat(underTest.find(testModel.id).first(), equalTo(testModel))

        itemsJob.cancel()
    }

    @Test
    fun canModifyItems() = runBlockingTest {
        val underTest = ToDoRepository(db.todoStore(), this, remoteDataSource)
        val testModel = ToDoModel("test model")
        val replacement = testModel.copy(notes = "This is the replacement")
        val results = mutableListOf<List<ToDoModel>>()

        val itemsJob = launch {
            underTest.items().collect { results.add(it) }
        }
```

CONTACTING A WEB SERVICE

```
assertThat(results[0], empty())

underTest.save(testModel)

assertThat(results[1], contains(testModel))

underTest.save(replacement)

assertThat(results[2], contains(replacement))

    itemsJob.cancel()
}

@Test
fun canRemoveItems() = runBlockingTest {
    val underTest = ToDoRepository(db.todoStore(), this, remoteDataSource)
    val testModel = ToDoModel("test model")
    val results = mutableListOf<List<ToDoModel>>()

    val itemsJob = launch {
        underTest.items().collect { results.add(it) }
    }

    assertThat(results[0], empty())

    underTest.save(testModel)

    assertThat(results[1], contains(testModel))

    underTest.delete(testModel)

    assertThat(results[2], empty())

    itemsJob.cancel()
}
}
```

(from [T32-Internet/ToDo/app/src/androidTest/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

The new PrefsRepository should be:

```
package com.commonsware.todo.repo

import android.content.Context
import androidx.preference.PreferenceManager
import com.commonsware.todo.R
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
```

CONTACTING A WEB SERVICE

```
class PrefsRepository(context: Context) {  
    private val prefs = PreferenceManager.getDefaultSharedPreferences(context)  
    private val webServiceUrlKey = context.getString(R.string.web_service_url_key)  
    private val defaultWebServiceUrl =  
        context.getString(R.string.web_service_url_default)  
  
    suspend fun loadWebServiceUrl(): String = withContext(Dispatchers.IO) {  
        prefs.getString(webServiceUrlKey, defaultWebServiceUrl) ?: defaultWebServiceUrl  
    }  
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/PrefsRepository.kt](#))

ToDoApp should look like:

```
package com.commonsware.todo  
  
import android.app.Application  
import android.text.format.DateUtils  
import com.commonsware.todo.repo.PrefsRepository  
import com.commonsware.todo.repo.ToDoDatabase  
import com.commonsware.todo.repo.ToDoRemoteDataSource  
import com.commonsware.todo.repo.ToDoRepository  
import com.commonsware.todo.report.RosterReport  
import com.commonsware.todo.ui.SingleModelMotor  
import com.commonware.todo.ui.roster.RosterMotor  
import com.github.jknack.handlebars.Handlebars  
import com.github.jknack.handlebars.Helper  
import kotlinx.coroutines.CoroutineScope  
import kotlinx.coroutines.SupervisorJob  
import okhttp3.OkHttpClient  
import org.koin.android.ext.koin.androidApplication  
import org.koin.android.ext.koin.androidContext  
import org.koin.android.ext.koin.androidLogger  
import org.koin.androidx.viewmodel.dsl.viewModel  
import org.koin.core.context.startKoin  
import org.koin.core.qualifier.named  
import org.koin.dsl.module  
import java.time.Instant  
  
class ToDoApp : Application() {  
    private val koinModule = module {  
        single(named("appScope")) { CoroutineScope(SupervisorJob()) }  
        single { ToDoDatabase.newInstance(androidContext()) }  
        single {  
            ToDoRepository(  
                get<ToDoDatabase>().todoStore(),  
            )  
        }  
    }  
}
```

CONTACTING A WEB SERVICE

```
        get(named("appScope")),
        get()
    )
}
single {
    Handlebars().apply {
        registerHelper("dateFormat", Helper<Instant> { value, _ ->
            DateUtils.getRelativeDateTimeString(
                androidContext(),
                value.toEpochMilli(),
                DateUtils.MINUTE_IN_MILLIS,
                DateUtils.WEEK_IN_MILLIS, 0
            )
        })
    }
}
single { RosterReport(androidContext(), get(), get(named("appScope"))) }
single { OkHttpClient.Builder().build() }
single { ToDoRemoteDataSource(get()) }
single { PrefsRepository(androidContext()) }
viewModel {
    RosterMotor(
        get(),
        get(),
        androidApplication(),
        get(named("appScope")),
        get()
    )
}
viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
}

override fun onCreate() {
    super.onCreate()

    startKoin {
        androidLogger()
        androidContext(this@ToDoApp)
        modules(koinModule)
    }
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

RosterMotor, should contain:

```
package com.commonsware.todo.ui.roster
```

CONTACTING A WEB SERVICE

```
import android.app.Application
import android.content.Context
import android.net.Uri
import androidx.core.content.FileProvider
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.BuildConfig
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.PrefsRepository
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.report.RosterReport
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import java.io.File

private const val AUTHORITY = BuildConfig.APPLICATION_ID + ".provider"

data class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val isLoaded: Boolean = false,
    val filterMode: FilterMode = FilterMode.ALL
)

sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
    data class ShareReport(val doc: Uri) : Nav()
}

class RosterMotor(
    private val repo: ToDoRepository,
    private val report: RosterReport,
    private val context: Application,
    private val appScope: CoroutineScope,
    private val prefs: PrefsRepository
) : ViewModel() {
    private val _states = MutableStateFlow(RosterViewState())
    val states = _states.asStateFlow()
    private val _navEvents = MutableSharedFlow<Nav>()
    val navEvents = _navEvents.asSharedFlow()
    private var job: Job? = null

    init {
        load(FilterMode.ALL)
    }

    fun load(filterMode: FilterMode) {
```

CONTACTING A WEB SERVICE

```
job?.cancel()

job = viewModelScope.launch {
    repo.items(filterMode).collect {
        _states.emit(RosterViewState(it, true, filterMode))
    }
}

fun save(model: ToDoModel) {
    viewModelScope.launch {
        repo.save(model)
    }
}

fun saveReport(doc: Uri) {
    viewModelScope.launch {
        report.generate(_states.value.items, doc)
        _navEvents.emit(Nav.ViewReport(doc))
    }
}

fun shareReport() {
    viewModelScope.launch {
        saveForSharing()
    }
}

fun importItems() {
    viewModelScope.launch {
        repo.importItems(prefs.loadWebServiceUrl())
    }
}

private suspend fun saveForSharing() {
    withContext(Dispatchers.IO + appScope.coroutineContext) {
        val shared = File(context.cacheDir, "shared").also { it.mkdirs() }
        val reportFile = File(shared, "report.html")
        val doc = FileProvider.getUriForFile(context, AUTHORITY, reportFile)

        _states.value.let { report.generate(it.items, doc) }
        _navEvents.emit(Nav.ShareReport(doc))
    }
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

CONTACTING A WEB SERVICE

The actions_roster menu resource should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">

    <item
        android:id="@+id/filter"
        android:icon="@drawable/ic_filter"
        android:title="@string/menu_filter"
        app:showAsAction="ifRoom|withText">
        <menu>

            <group
                android:id="@+id/filter_group"
                android:checkableBehavior="single" >
                <item
                    android:id="@+id/all"
                    android:checked="true"
                    android:title="@string/menu_filter_all" />
                <item
                    android:id="@+id/completed"
                    android:title="@string/menu_filter_completed" />
                <item
                    android:id="@+id/outstanding"
                    android:title="@string/menu_filter_outstanding" />
            </group>
        </menu>
    </item>
    <item
        android:id="@+id/add"
        android:icon="@drawable/ic_add"
        android:title="@string/menu_add"
        app:showAsAction="ifRoom|withText" />
    <item
        android:id="@+id/save"
        android:icon="@drawable/ic_save"
        android:title="@string/menu_save"
        app:showAsAction="ifRoom|withText" />
    <item
        android:id="@+id/share"
        android:icon="@drawable/ic_share"
        android:title="@string/menu_share"
        app:showAsAction="ifRoom|withText" />
    <item
        android:id="@+id/importItems"
        android:icon="@drawable/ic_download"
```

CONTACTING A WEB SERVICE

```
    android:title="@string/menu_import"
    app:showAsAction="never" />
</menu>
```

(from [T32-Internet/ToDo/app/src/main/res/menu/actions_roster.xml](#))

The strings resource should look like:

```
<resources>
    <string name="app_name">ToDo</string>
    <string name="msg_empty">Click the + icon to add a todo item!</string>
    <string name="msg_empty_filtered">Click the + icon to add a todo item, or change
your filter to show other items</string>
    <string name="menu_about">About</string>
    <string name="is_completed">Item is completed</string>
    <string name="created_on">Created on:</string>
    <string name="menu_edit">Edit</string>
    <string name="desc">Description</string>
    <string name="notes">Notes</string>
    <string name="menu_save">Save</string>
    <string name="menu_add">Add</string>
    <string name="menu_delete">Delete</string>
    <string name="menu_filter">Filter</string>
    <string name="menu_filter_all">All</string>
    <string name="menu_filter_completed">Completed</string>
    <string name="menu_filter_outstanding">Outstanding</string>
    <string name="oops">Sorry! Something went wrong!</string>
    <string name="report_template"><![CDATA[<h1>To-Do Items</h1>
{{#this}}
<h2>{{description}}</h2>
<p>{{#completed}}<b>COMPLETED</b> &mdash; {{/completed}}Created on: {{dateFormat
createdOn}}</p>
<p>{{notes}}</p>
{{/this}}
]]></string>
    <string name="menu_share">Share</string>
    <string name="pref_url_title">Web service URL</string>
    <string name="web_service_url_key">webServiceUrl</string>
    <string name="web_service_url_default">https://commonsware.com/AndExplore/2.0/items.json</string>
    <string name="settings">Settings</string>
    <string name="menu_import">Import</string>
</resources>
```

(from [T32-Internet/ToDo/app/src/main/res/values/strings.xml](#))

And RosterListFragment should resemble:

CONTACTING A WEB SERVICE

```
package com.commonsware.todo.ui.roster

import android.content.Intent
import android.net.Uri
import android.os.Bundle
import android.util.Log
import android.view.*
import android.widget.Toast
import androidx.activity.result.contract.ActivityResultContracts
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoRosterBinding
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel

private const val TAG = "ToDo"

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private val menuMap = mutableMapOf<FilterMode, MenuItem>()
    private var binding: TodoRosterBinding? = null

    private val createDoc =
        registerForActivityResult(ActivityResultContracts.CreateDocument()) {
            motor.saveReport(it)
        }

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

        setHasOptionsMenu(true)
    }

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View = TodoRosterBinding.inflate(inflater, container, false)
        .also { binding = it }
        .root

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

CONTACTING A WEB SERVICE

```
super.onViewCreated(view, savedInstanceState)

    val adapter = RosterAdapter(
        layoutInflater,
        onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
        onRowClick = ::display
    )

    binding?.items?.apply {
        setAdapter(adapter)
        layoutManager = LinearLayoutManager(context)

        addItemDecoration(
            DividerItemDecoration(
                activity,
                DividerItemDecoration.VERTICAL
            )
        )
    }
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = if (state.isLoading) View.GONE else View.VISIBLE

            when {
                state.items.isEmpty() && state.filterMode == FilterMode.ALL -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty_filtered)
                }
                else -> empty.visibility = View.GONE
            }
        }
    }

    menuMap[state.filterMode]?.isChecked = true
}
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.navEvents.collect { nav ->
        when (nav) {
            is Nav.ViewReport -> viewReport(nav.doc)
```

CONTACTING A WEB SERVICE

```
        is Nav.ShareReport -> shareReport(nav.doc)
    }
}
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)

    menuMap.apply {
        put(FilterMode.ALL, menu.findItem(R.id.all))
        put(FilterMode.COMPLETED, menu.findItem(R.id.completed))
        put(FilterMode.OUTSTANDING, menu.findItem(R.id.outstanding))
    }

    menuMap[motor.states.value.filterMode]?.isChecked = true

    super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
        R.id.all -> {
            item.isChecked = true
            motor.load(FilterMode.ALL)
            return true
        }
        R.id.completed -> {
            item.isChecked = true
            motor.load(FilterMode.COMPLETED)
            return true
        }
        R.id.outstanding -> {
            item.isChecked = true
            motor.load(FilterMode.OUTSTANDING)
            return true
        }
        R.id.save -> {
    }
}
```

CONTACTING A WEB SERVICE

```
    saveReport()
    return true
}
R.id.share -> {
    motor.shareReport()
    return true
}
R.id.importItems -> {
    motor.importItems()
    return true
}
}

return super.onOptionsItemSelected(item)
}

private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}

private fun saveReport() {
    createDoc.launch("report.html")
}

private fun viewReport(uri: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_VIEW, uri)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    )
}

private fun shareReport(doc: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_SEND)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
            .setType("text/html")
            .putExtra(Intent.EXTRA_STREAM, doc)
    )
}

private fun safeStartActivity(intent: Intent) {
    try {
        startActivity(intent)
    }
}
```

CONTACTING A WEB SERVICE

```
    } catch (t: Throwable) {
        Log.e(TAG, "Exception starting $intent", t)
        Toast.makeText(requireActivity(), R.string.oops, Toast.LENGTH_LONG).show()
    }
}
```

(from [T32-Internet/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/build.gradle](#)
- [build.gradle](#)
- [app/src/main/AndroidManifest.xml](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoServerItem.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoRemoteDataSource.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#)
- [app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#)
- [app/src/androidTest/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/PrefsRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)
- [app/src/main/res/drawable/ic_download.xml](#)
- [app/src/main/res/menu/actions_roster.xml](#)
- [app/src/main/res/values/strings.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)

Showing a Dialog

When stuff goes wrong, the user might like to know about it.

Roughly speaking, you have three major ways of informing the user of a problem from your UI... besides simply crashing:

- You can show the error message in the current activity/fragment. This might be directly in an existing layout, for example. This is good for advisory messages, but it may be difficult to squeeze in a critical error message along with the rest of your content.
- You can navigate to a completely different activity/fragment. This can be a bit jarring for the user.
- You can display a dialog, having it float over top of your current activity and any fragments.

In this chapter, we will explore that third option. If there is a problem when importing the notes, we will display an error dialog, with options to cancel or retry the import.

In Android, there are several ways to get the visual effect of a floating dialog. We will take the most modern option, which is to use a `DialogFragment`. As the name suggests, this is a fragment that knows how to render itself as a dialog.

The Navigation component has support for `DialogFragment`. There is even an option to get a result from that dialog, such as whether the user opted to retry or cancel the import. We will leverage that as part of our work here.

This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab 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 Fragment

We need another fragment!

Rather than being tied to a specific piece of business logic, like displaying a to-do item, this error fragment can be more “general purpose”. Plus, for our limited needs, we can skip giving it a viewmodel. So, we can just place it in the ui sub-package rather than create a brand-new package for it.

Right-click over the `com.commonware.todo.ui` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `ErrorDialogFragment`, and choose “Class” for the kind. Press `Enter` or `Return` to create the class. Then, replace the class declaration with:

```
package com.commonware.todo.ui

import androidx.fragment.app.DialogFragment

class ErrorDialogFragment : DialogFragment() {

    enum class ErrorScenario { Import, None }
```

Unlike our other fragments, we inherit from `DialogFragment` this time. Otherwise, this stub is pretty much like the stubs of the other fragments that we created several chapters ago.

We also define an `ErrorScenario`. This is another enum, akin to the `FilterMode` that we created earlier. We will use this to allow users of `ErrorDialogFragment` to indicate what “scenario” the error is about. This will be important for fragments that might show multiple error dialogs and need to distinguish one from another for handling “retry” requests. `Import` represents an error in importing data, while `None` means that there is no current error.

Step #2: Updating the Navigation Graph

Just like our other fragments, we need to add `ErrorDialogFragment` to our navigation graph. For now, we will only use `ErrorDialogFragment` from `RosterListFragment`, so we can set up a more conventional action, rather than the global action that we used for `PrefsFragment`.

SHOWING A DIALOG

Open up `res/navigation/nav_graph.xml` and click the add-destination toolbar button (rectangle with green plus sign in the corner). You should see `ErrorDialogFragment` as an option in the destination drop-down. Click on it, then drag its tile to some clean spot in the diagram.

Next, click on the `errorDialogFragment` tile, if it is not already selected. In the “Attributes” pane, add three arguments, using the “+” button in the “Arguments” section:

- title as a string
- message as a string
- scenario as an ErrorScenario

For the `scenario` argument, you will need to choose “Custom Enum...” from the “Type” drop-down list in the “Add Argument” dialog:

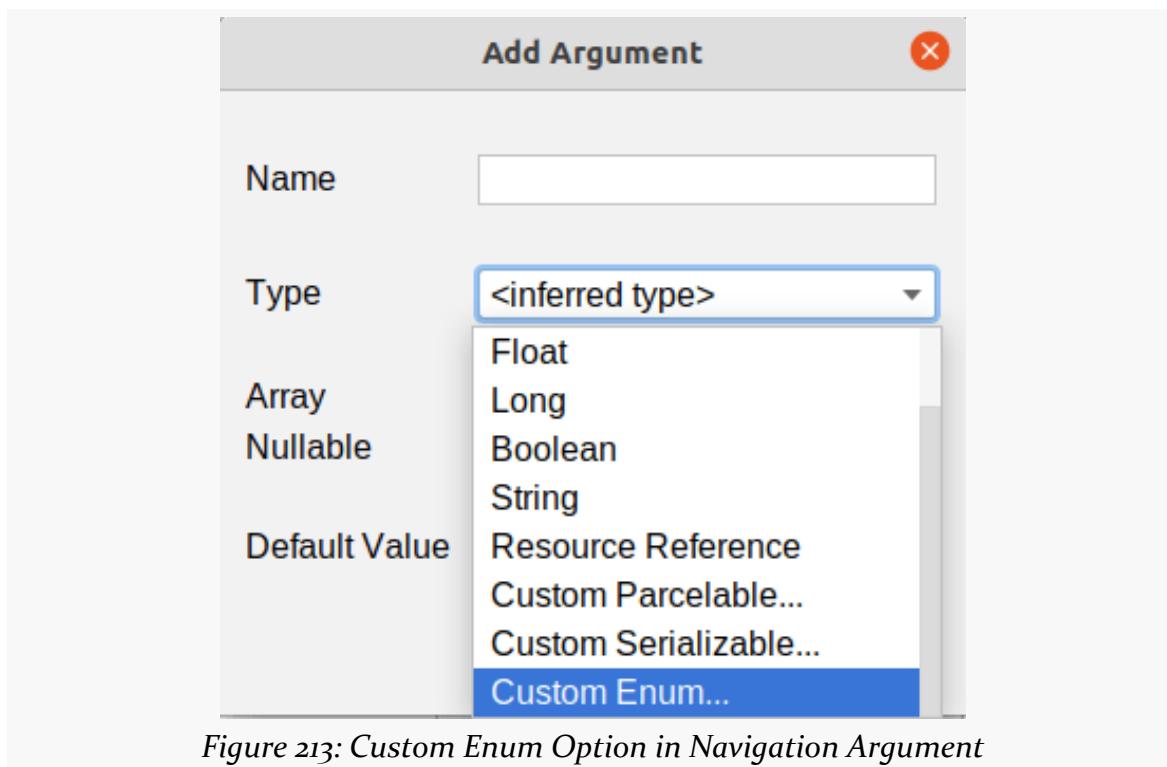


Figure 213: Custom Enum Option in Navigation Argument

SHOWING A DIALOG

That, in turn, will pop up a “Select Class” dialog for you to choose your desired enum from:

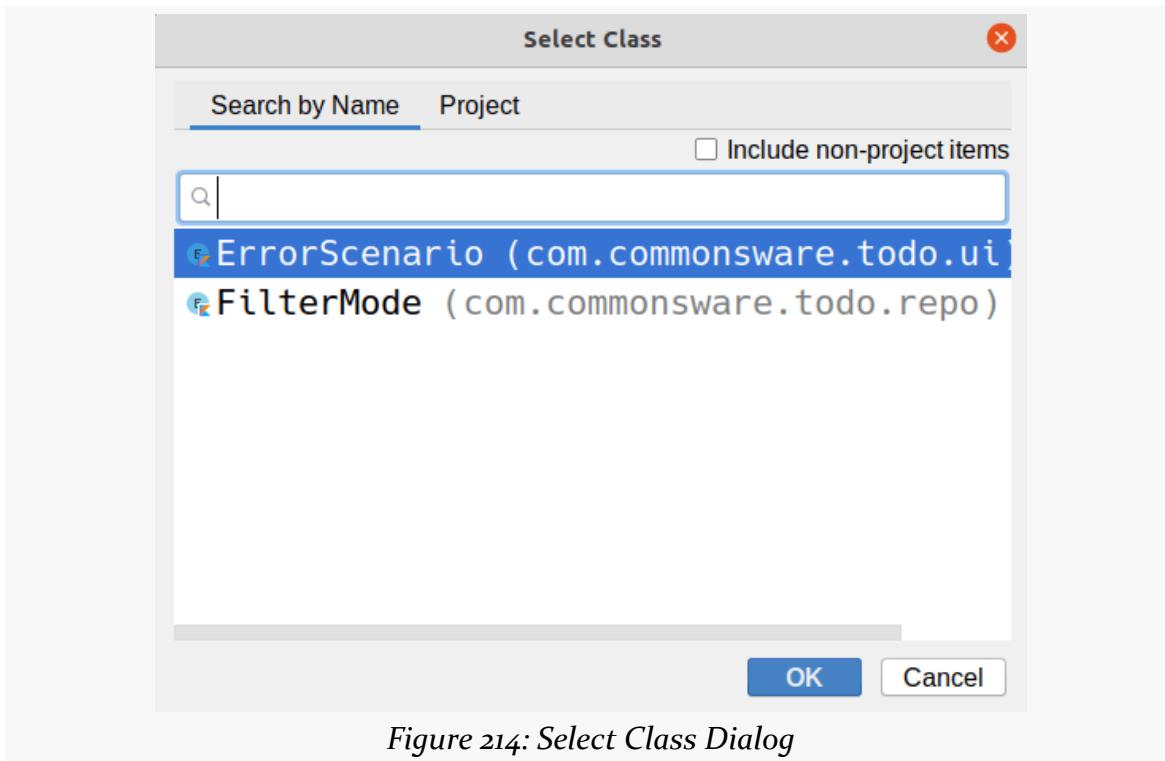


Figure 214: Select Class Dialog

Choose `ErrorScenario` from the list and click “OK” to use that for the “Type”.

Then, drag an arrow from the `rosterListFragment` tile to the `errorDialogFragment` tile, to set up an action between them. With that arrow selected, in the “Attributes” pane, change the “id” to `showError`.

Finally, build the app, so the Navigation code-generated classes get created.

Step #3: Defining the Dialog Content

We can now start to use those arguments to populate the dialog. There are a couple of approaches to defining what the dialog contains. We will take the simplest one: use `AlertDialog.Builder` to create a standard Android dialog.

With that in mind, add `args` and `onCreateDialog()` to `ErrorDialogFragment`:

SHOWING A DIALOG

```
class ErrorDialogFragment : DialogFragment() {
    private val args: ErrorDialogFragmentArgs by navArgs()

    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {
        return AlertDialog.Builder(requireActivity())
            .setTitle(args.title)
            .setMessage(args.message)
            .setPositiveButton(R.string.retry) { _, _ -> onRetryRequest() }
            .setNegativeButton(R.string.cancel) { _, _ ->  }
            .create()
    }

    private fun onRetryRequest() {
        // TODO
    }
}
```

The `args` property is like those we have in `DisplayFragment` and `EditFragment`, so we can get the values sent to us by whatever fragment wants to display the dialog.

`onCreateDialog()` needs to return a `Dialog` object; `AlertDialog` is a subclass of `Dialog`. `Dialog` knows how to create floating windows, while `AlertDialog` styles one of those in a typical Android fashion. So, we create an `AlertDialog.Builder`, configure it, and use `build()` to create the `AlertDialog` to return.

The configuration includes:

- Setting the title and message from our arguments
- Setting the captions of the positive and negative buttons to a pair of string resources, and tying the positive button click to an `onRetryRequest()` stub function

Those string resources do not exist yet, so you will need to add them:

```
<string name="cancel">Cancel</string>
<string name="retry">Retry</string>
```

(from [T33-Dialog/ToDo/app/src/main/res/values/strings.xml](#))

We will fill in that `onRetryRequest()` function a bit later in this chapter.

Step #4: Emitting Errors From the Motor

The point behind the dialog is to tell the user when there is a problem in importing

SHOWING A DIALOG

to-do items. That implies that something knows to show the dialog when that occurs. One way to do that is to have the viewmodel tell its activity or fragment about errors. In this case, that would be RosterMotor telling RosterListFragment about errors.

So, add these properties to RosterMotor:

```
private val _errorEvents = MutableSharedFlow<ErrorScenario>()
val errorEvents = _errorEvents.asSharedFlow()
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

These are similar to the _navEvents/navEvents pair that we set up earlier. However, they emit an ErrorScenario instead.

Then, revise importItems() on RosterMotor to be:

```
fun importItems() {
    viewModelScope.launch {
        try {
            repo.importItems(prefs.loadWebServiceUrl())
        } catch (ex: Exception) {
            Log.e("ToDo", "Exception importing items", ex)
            _errorEvents.emit(ErrorScenario.Import)
        }
    }
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Now we wrap our importItems() call on ToDoRepository() in a try/catch block. If we get an error from the repository, we log it to Logcat using Log.e() and emit an Import scenario on our new channel.

Step #5: Reacting to Errors

Now we can have RosterListFragment react to those errors and show the dialog.

First, though, we need a title and message to display in the dialog. Add these two string resources to res/values/strings.xml:

```
<string name="import_error_title">Import Failure</string>
<string name="import_error_message">Something went wrong with the import!</string>
```

(from [T33-Dialog/ToDo/app/src/main/res/values/strings.xml](#))

SHOWING A DIALOG

Next, in RosterListFragment, add this handleImportError() function:

```
private fun handleImportError() {
    findNavController().navigate(
        RosterListFragmentDirections.showError(
            getString(R.string.import_error_title),
            getString(R.string.import_error_message),
            ErrorScenario.Import
        )
    )
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This uses the Navigation component to navigate to the AlertDialogFragment, using RosterListFragmentDirections and its code-generated showError() function. We pass showError() our two new strings plus ErrorScenario.Import.

Then, add this block of code to the bottom of onViewCreated() in RosterListFragment:

```
viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.errorEvents.collect { error ->
        when (error) {
            ErrorScenario.Import -> handleImportError()
        }
    }
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This observes the new errorEvents SharedFlow from our motor and, when we receive an Import error, calls handleImportError(). This structure sets us up to be able to handle other types of ErrorScenario, if and when we add any.

Step #6: Responding to Input

So now we are able to display the dialog. However, we still need to find out if the user clicks the “Retry” button, so we can retry the import operation.

For that, first, add this companion object to AlertDialogFragment:

SHOWING A DIALOG

```
companion object {
    const val KEY_RETRY = "retryRequested"
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/ErrorDialogFragment.kt](#))

This just sets up a constant. We use a `companion object` here so other classes — such as `RosterListFragment` — have access to the constant while also keeping it tied to `ErrorDialogFragment`. If we were only using this constant inside of `ErrorDialogFragment`, we could have just used a `private const val` file-level property, as we have before.

Then, modify `onRetryRequest()` to be:

```
private fun onRetryRequest() {
    findNavController()
        .previousBackStackEntry?.savedStateHandle?.set(KEY_RETRY, args.scenario)
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/ErrorDialogFragment.kt](#))

This strange construction is how we can get a result out of this dialog and over to `RosterListFragment` as the one that displayed the dialog. Here, we:

- Get our `NavController`, for access to the Navigation component APIs
- Find the `BackStackEntry` corresponding to whatever it was that displayed this dialog
- Get a `SavedStateHandle` for that `BackStackEntry`, where `SavedStateHandle` is a bit like a `HashMap` (key-value store of data)
- Set the `KEY_RETRY` value in that state to be the `ErrorScenario` that was passed into us via the navigation arguments

With this in place, when the user clicks the “Retry” button, we update this `SavedStateHandle` with the `ErrorScenario` that triggered the dialog.

Over on `RosterListFragment`, we can now find out about changes in the `KEY_RETRY` value and use that to retry the import.

First, though, add this `clearImportError()` function to `RosterListFragment`:

```
private fun clearImportError() {
    findNavController()
        .getBackStackEntry(R.id.rosterListFragment)
```

SHOWING A DIALOG

```
.savedStateHandle  
.set(ErrorDialogFragment.KEY_RETRY, ErrorScenario.None)  
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Then, add this block of code to the bottom of onViewCreated():

```
findNavController()  
.getBackStackEntry(R.id.rosterListFragment)  
.savedStateHandle  
.getLiveData<ErrorScenario>(ErrorDialogFragment.KEY_RETRY)  
.observe(viewLifecycleOwner) { retryScenario ->  
    when (retryScenario) {  
        ErrorScenario.Import -> {  
            clearImportError()  
            motor.importItems()  
        }  
    }  
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Here, we:

- Get our NavController, for access to the Navigation component APIs
- Find our own BackStackEntry, based upon our destination ID (R.id.rosterListFragment)
- Get the SavedStateHandle for that BackStackEntry
- Observe a LiveData of objects associated with KEY_RETRY, so we find out when that value changes
- If we get an Import error, clear it, then retry the import using motor.importItems()

LiveData is a bit like StateFlow. It represents a source of states, in this case the states of our KEY_RETRY value. Our observe() lambda expression will be invoked when that state changes. LiveData is part of the Jetpack and works with both Java and Kotlin, whereas StateFlow is a Kotlin-specific construct. Jetpack Navigation works with both Java and Kotlin, so it uses LiveData for consistency between the two languages.

SHOWING A DIALOG



You can learn more about LiveData in the "Thinking About Threads and LiveData" chapter of [Elements of Android Jetpack!](#)

So, when the user clicks the “Retry” button, the `ErrorDialogFragment` emits an `ErrorScenario`, which `RosterListFragment` picks up and uses to handle whatever needs to be retried.

`clearImportError()` addresses the fact that these results are delivered by `LiveData`. `LiveData` intrinsically is a cache. If we do not remove this value from our `SavedStateHandle`, we can wind up with the error being re-delivered to us... erroneously:

- Get the error and see the dialog
- Click “Retry”, and wind up getting the error again
- Click “Cancel” to dismiss the dialog
- Navigate to another screen (e.g., choose Settings from the overflow menu)
- Navigate back to `RosterListFragment`
- When we start observing the `LiveData` again, we get the error re-delivered to us, so we retry once more, unexpectedly

`remove()` on the `SavedStateHandle` sets our value to `None`, so we skip over the Import error logic and avoid this infinite loop.

Step #7: Trying It Out

To see this dialog in action:

- Run the revised app
- Choose Settings from the overflow menu and edit the Web service URL to something that is invalid (e.g., add an x on the end)
- Choose Import from the overflow menu

SHOWING A DIALOG

At this point, you should get our AlertDialog:

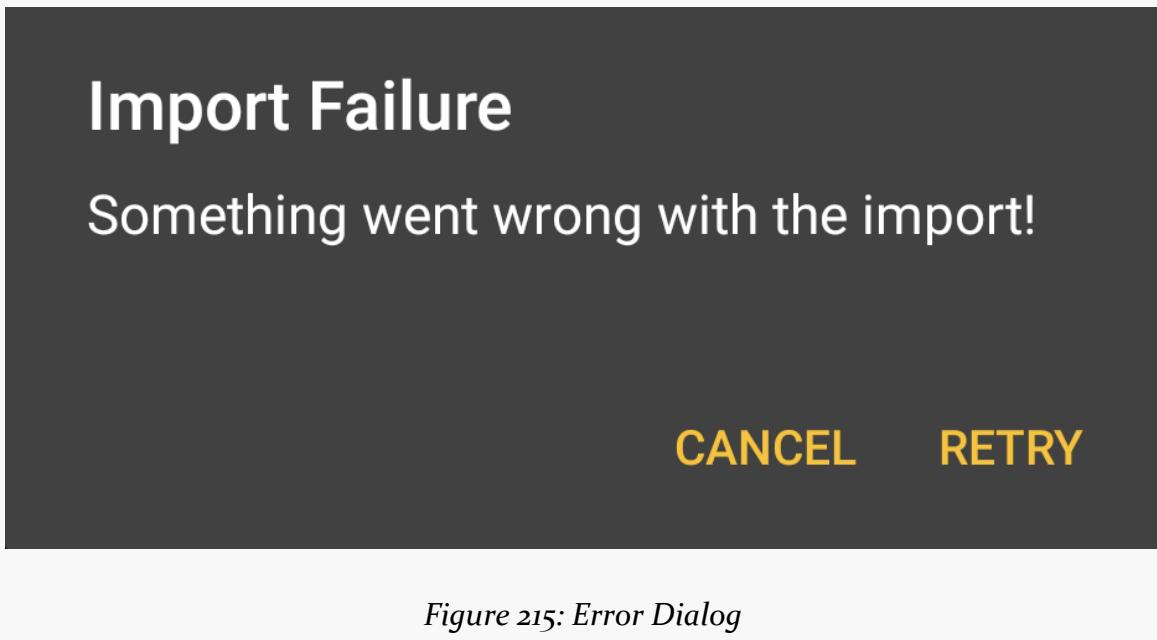


Figure 215: Error Dialog

If you click “Retry”, you will get the dialog again right away, as nothing has changed about our error conditions — we still have the incorrect URL.

If you click “Cancel” and revert the change to the URL that you introduced, then import again, it should succeed and you should not see the dialog.

We are going to be importing to-do items some more in the next tutorial, so if you did modify the URL, be sure to change it back to a valid value before continuing.

Final Results

Our completed `ErrorDialogFragment` should look like:

```
package com.commonsware.todo.ui

import android.app.Dialog
import android.os.Bundle
import androidx.appcompat.app.AlertDialog
import androidx.fragment.app.DialogFragment
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.R
```

SHOWING A DIALOG

```
class ErrorDialogFragment : DialogFragment() {  
    companion object {  
        const val KEY_RETRY = "retryRequested"  
    }  
  
    private val args: ErrorDialogFragmentArgs by navArgs()  
  
    override fun onCreateDialog(savedInstanceState: Bundle?): Dialog {  
        return AlertDialog.Builder(requireActivity())  
            .setTitle(args.title)  
            .setMessage(args.message)  
            .setPositiveButton(R.string.retry) { _, _ -> onRetryRequest() }  
            .setNegativeButton(R.string.cancel) { _, _ -> }  
            .create()  
    }  
  
    private fun onRetryRequest() {  
        findNavController()  
            .previousBackStackEntry?.savedStateHandle?.set(KEY_RETRY, args.scenario)  
    }  
}  
  
enum class ErrorScenario { Import, None }
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/ErrorDialogFragment.kt](#))

The updated navigation graph (`nav_graph.xml`) should resemble:

```
<?xml version="1.0" encoding="utf-8"?>  
<navigation xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:app="http://schemas.android.com/apk/res-auto"  
    android:id="@+id/nav_graph.xml"  
    app:startDestination="@+id/rosterListFragment">  
  
    <fragment  
        android:id="@+id/rosterListFragment"  
        android:name="com.commonsware.todo.ui.roster.RosterListFragment"  
        android:label="@string/app_name">  
        <action  
            android:id="@+id/displayModel"  
            app:destination="@+id/displayFragment" />  
        <action  
            android:id="@+id/createModel"  
            app:destination="@+id/editFragment" >  
            <argument  
                android:name="modelId"  
                android:defaultValue="@null" />
```

SHOWING A DIALOG

```
</action>
<action
    android:id="@+id/showError"
    app:destination="@+id/errorDialogFragment" />
</fragment>
<fragment
    android:id="@+id/displayFragment"
    android:name="com.commonsware.todo.ui.display.DisplayFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string" />
    <action
        android:id="@+id/editModel"
        app:destination="@+id/editFragment" />
</fragment>
<fragment
    android:id="@+id/editFragment"
    android:name="com.commonsware.todo.ui.edit.EditFragment"
    android:label="@string/app_name" >
    <argument
        android:name="modelId"
        app:argType="string"
        app:nullable="true" />
</fragment>
<fragment
    android:id="@+id/prefsFragment"
    android:name="com.commonsware.todo.ui.prefs.PrefsFragment"
    android:label="@string/settings" />
<action android:id="@+id/editPrefs" app:destination="@+id/prefsFragment" />
<dialog
    android:id="@+id/errorDialogFragment"
    android:name="com.commonsware.todo.ui.ErrorDialogFragment"
    android:label="ErrorDialogFragment" >
    <argument
        android:name="title"
        app:argType="string" />
    <argument
        android:name="message"
        app:argType="string" />
    <argument
        android:name="scenario"
        app:argType="com.commonsware.todo.ui.ErrorScenario" />
</dialog>
</navigation>
```

(from [T33-Dialog/ToDo/app/src/main/res/navigation/nav_graph.xml](#))

SHOWING A DIALOG

The strings.xml resource should now contain:

```
<resources>
    <string name="app_name">ToDo</string>
    <string name="msg_empty">Click the + icon to add a todo item!</string>
    <string name="msg_empty_filtered">Click the + icon to add a todo item, or change
your filter to show other items</string>
    <string name="menu_about">About</string>
    <string name="is_completed">Item is completed</string>
    <string name="created_on">Created on:</string>
    <string name="menu_edit">Edit</string>
    <string name="desc">Description</string>
    <string name="notes">Notes</string>
    <string name="menu_save">Save</string>
    <string name="menu_add">Add</string>
    <string name="menu_delete">Delete</string>
    <string name="menu_filter">Filter</string>
    <string name="menu_filter_all">All</string>
    <string name="menu_filter_completed">Completed</string>
    <string name="menu_filter_outstanding">Outstanding</string>
    <string name="oops">Sorry! Something went wrong!</string>
    <string name="report_template"><![CDATA[<h1>To-Do Items</h1>
{{#this}}
<h2>{{description}}</h2>
<p>{{#completed}}<b>COMPLETED</b> &mdash; {{/completed}}Created on: {{dateFormat
createdOn}}</p>
<p>{{notes}}</p>
{{/this}}
]]></string>
    <string name="menu_share">Share</string>
    <string name="pref_url_title">Web service URL</string>
    <string name="web_service_url_key">webServiceUrl</string>
    <string name="web_service_url_default">https://commonsware.com/AndExplore/2.0/
items.json</string>
    <string name="settings">Settings</string>
    <string name="menu_import">Import</string>
    <string name="cancel">Cancel</string>
    <string name="retry">Retry</string>
    <string name="import_error_title">Import Failure</string>
    <string name="import_error_message">Something went wrong with the import!</string>
</resources>
```

(from [T33-Dialog/ToDo/app/src/main/res/values/strings.xml](#))

Our RosterMotor should now resemble:

```
package com.commonsware.todo.ui.roster
```

SHOWING A DIALOG

```
import android.app.Application
import android.content.Context
import android.net.Uri
import android.util.Log
import androidx.core.content.FileProvider
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.BuildConfig
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.PrefsRepository
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.report.RosterReport
import com.commonsware.todo.ui.ErrorScenario
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*
import java.io.File

private const val AUTHORITY = BuildConfig.APPLICATION_ID + ".provider"

data class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val isLoading: Boolean = false,
    val filterMode: FilterMode = FilterMode.ALL
)

sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
    data class ShareReport(val doc: Uri) : Nav()
}

class RosterMotor(
    private val repo: ToDoRepository,
    private val report: RosterReport,
    private val context: Application,
    private val appScope: CoroutineScope,
    private val prefs: PrefsRepository
) : ViewModel() {
    private val _states = MutableStateFlow(RosterViewState())
    val states = _states.asStateFlow()
    private val _navEvents = MutableSharedFlow<Nav>()
    val navEvents = _navEvents.asSharedFlow()
    private val _errorEvents = MutableSharedFlow<ErrorScenario>()
    val errorEvents = _errorEvents.asSharedFlow()
    private var job: Job? = null

    init {
```

SHOWING A DIALOG

```
load(FilterMode.ALL)
}

fun load(filterMode: FilterMode) {
    job?.cancel()

    job = viewModelScope.launch {
        repo.items(filterMode).collect {
            _states.emit(RosterViewState(it, true, filterMode))
        }
    }
}

fun save(model: ToDoModel) {
    viewModelScope.launch {
        repo.save(model)
    }
}

fun saveReport(doc: Uri) {
    viewModelScope.launch {
        report.generate(_states.value.items, doc)
        _navEvents.emit(Nav.ViewReport(doc))
    }
}

fun shareReport() {
    viewModelScope.launch {
        saveForSharing()
    }
}

fun importItems() {
    viewModelScope.launch {
        try {
            repo.importItems(prefs.loadWebServiceUrl())
        } catch (ex: Exception) {
            Log.e("ToDo", "Exception importing items", ex)
            _errorEvents.emit(ErrorScenario.Import)
        }
    }
}

private suspend fun saveForSharing() {
    withContext(Dispatchers.IO + appScope.coroutineContext) {
        val shared = File(context.cacheDir, "shared").also { it.mkdirs() }
        val reportFile = File(shared, "report.html")
        val doc = FileProvider.getUriFromFile(context, AUTHORITY, reportFile)
```

SHOWING A DIALOG

```
        _states.value.let { report.generate(it.items, doc) }
        _navEvents.emit(Nav.ShareReport(doc))
    }
}
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Finally, our revised RosterListFragment should include:

```
package com.commonsware.todo.ui.roster

import android.content.Intent
import android.net.Uri
import android.os.Bundle
import android.util.Log
import android.view.*
import android.widget.Toast
import androidx.activity.result.contract.ActivityResultContracts
import androidx.fragment.app.Fragment
import androidx.lifecycle.lifecycleScope
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
import com.commonsware.todo.databinding.TodoRosterBinding
import com.commonsware.todo.repo.FilterMode
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.ui.ErrorDialogFragment
import com.commonsware.todo.ui.ErrorScenario
import kotlinx.coroutines.flow.collect
import org.koin.androidx.viewmodel.ext.android.viewModel

private const val TAG = "ToDo"

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()
    private val menuMap = mutableMapOf<FilterMode, MenuItem>()
    private var binding: TodoRosterBinding? = null

    private val createDoc =
        registerForActivityResult(ActivityResultContracts.CreateDocument()) {
            motor.saveReport(it)
        }

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

SHOWING A DIALOG

```
    setHasOptionsMenu(true)
}

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
): View = TodoRosterBinding.inflate(inflater, container, false)
    .also { binding = it }
    .root

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

    val adapter = RosterAdapter(
        layoutInflater,
        onCheckboxToggle = { motor.save(it.copy(isCompleted = !it.isCompleted)) },
        onRowClick = ::display
    )

    binding?.items?.apply {
        setAdapter(adapter)
        layoutManager = LinearLayoutManager(context)

        addItemDecoration(
            DividerItemDecoration(
                activity,
                DividerItemDecoration.VERTICAL
            )
        )
    }
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.states.collect { state ->
        adapter.submitList(state.items)

        binding?.apply {
            loading.visibility = if (state.isLoading) View.GONE else View.VISIBLE

            when {
                state.items.isEmpty() && state.filterMode == FilterMode.ALL -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty)
                }
                state.items.isEmpty() -> {
                    empty.visibility = View.VISIBLE
                    empty.setText(R.string.msg_empty_filtered)
                }
            }
        }
    }
}
```

SHOWING A DIALOG

```
        }
        else -> empty.visibility = View.GONE
    }
}

menuMap[state.filterMode]?.isChecked = true
}
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.navEvents.collect { nav ->
        when (nav) {
            is Nav.ViewReport -> viewReport(nav.doc)
            is Nav.ShareReport -> shareReport(nav.doc)
        }
    }
}

viewLifecycleOwner.lifecycleScope.launchWhenStarted {
    motor.errorEvents.collect { error ->
        when (error) {
            ErrorScenario.Import -> handleImportError()
        }
    }
}

findNavController()
    .getBackStackEntry(R.id.rosterListFragment)
    .savedStateHandle
    .getLiveData<ErrorScenario>(AlertDialogFragment.KEY_RETRY)
    .observe(viewLifecycleOwner) { retryScenario ->
        when (retryScenario) {
            ErrorScenario.Import -> {
                clearImportError()
                motor.importItems()
            }
        }
    }
}

override fun onDestroyView() {
    binding = null

    super.onDestroyView()
}

override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_roster, menu)
```

SHOWING A DIALOG

```
menuMap.apply {
    put(FilterMode.ALL, menu.findItem(R.id.all))
    put(FilterMode.COMPLETED, menu.findItem(R.id.completed))
    put(FilterMode.OUTSTANDING, menu.findItem(R.id.outstanding))
}

menuMap[motor.states.value.filterMode]?.isChecked = true

super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.add -> {
            add()
            return true
        }
        R.id.all -> {
            item.isChecked = true
            motor.load(FilterMode.ALL)
            return true
        }
        R.id.completed -> {
            item.isChecked = true
            motor.load(FilterMode.COMPLETED)
            return true
        }
        R.id.outstanding -> {
            item.isChecked = true
            motor.load(FilterMode.OUTSTANDING)
            return true
        }
        R.id.save -> {
            saveReport()
            return true
        }
        R.id.share -> {
            motor.shareReport()
            return true
        }
        R.id.importItems -> {
            motor.importItems()
            return true
        }
    }

    return super.onOptionsItemSelected(item)
}
```

SHOWING A DIALOG

```
}

private fun display(model: ToDoModel) {
    findNavController()
        .navigate(RosterListFragmentDirections.displayModel(model.id))
}

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel(null))
}

private fun saveReport() {
    createDoc.launch("report.html")
}

private fun viewReport(uri: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_VIEW, uri)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
    )
}

private fun shareReport(doc: Uri) {
    safeStartActivity(
        Intent(Intent.ACTION_SEND)
            .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
            .setType("text/html")
            .putExtra(Intent.EXTRA_STREAM, doc)
    )
}

private fun safeStartActivity(intent: Intent) {
    try {
        startActivity(intent)
    } catch (t: Throwable) {
        Log.e(TAG, "Exception starting $intent", t)
        Toast.makeText(requireActivity(), R.string.oops, Toast.LENGTH_LONG).show()
    }
}

private fun handleImportError() {
    findNavController().navigate(
        RosterListFragmentDirections.showError(
            getString(R.string.import_error_title),
            getString(R.string.import_error_message),
            ErrorScenario.Import
        )
    )
}
```

SHOWING A DIALOG

```
}

private fun clearImportError() {
    findNavController()
        .getBackStackEntry(R.id.rosterListFragment)
        .savedStateHandle
        .set(ErrorDialogFragment.KEY_RETRY, ErrorScenario.None)
}
}
```

(from [T33-Dialog/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

What We Changed

The book's GitLab repository contains [the entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/java/com/commonsware/todo/ui/ErrorDialogFragment.kt](#)
- [app/src/main/res/navigation/nav_graph.xml](#)
- [app/src/main/res/values/strings.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)

Scheduling Work

Sometimes, we need to do work periodically, even at times when the user is not using the app. Android has many ways to try to accomplish this, and most have problems. That is because developers have a tendency to abuse this sort of capability (e.g., trying to do something every 5 seconds) in ways that drain the battery. Users get very irritated when their battery level keeps plummeting when they are not using their device. So, Google has made it increasingly more difficult to do this sort of background work.

If your needs are fairly casual — do a small bit of work every so often, with no particular concerns over the precise timing — `WorkManager` can handle the job. With `WorkManager`, we describe the work that we want to do and under what conditions that it should be done, including timing (e.g., every 15 minutes). `WorkManager` takes care of the rest. `WorkManager` is not guaranteeing anything about the timing, though, as Android tends to block background work after a while of the device being idle, to conserve battery. But, for periodic work, `WorkManager` is about as good as we can get nowadays, particularly for a purely device-side solution.

In this tutorial, we will integrate `WorkManager`, to periodically import the to-do items from our “Web service”.

This is a continuation of the work we did in [the previous tutorial](#). The book’s GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

Step #1: Defining a `SwitchPreference`

We need some way for the user to control whether this sort of periodic import should be happening. Some users might like it, and some users might not.

SCHEDULING WORK

(for this sample app, *no* users should like it, since it will keep importing the same items over and over... but this is just a book sample)

One way to do that is to have another preference in our settings screen. In particular, Android offers CheckBoxPreference and SwitchPreference for this sort of on/off toggle.

We are going to need some string resources, as we did with the EditTextPreference. So, switch over to res/values/strings.xml and add these two strings:

```
<string name="pref_import_title">Import periodically</string>
<string name="import_key">doPeriodicImport</string>
```

(from [T34-Work/ToDo/app/src/main/res/values/strings.xml](#))

Then, open res/xml/prefs.xml in Android Studio, switch to the “Code” view, and replace the XML with:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <EditTextPreference
        android:key="@string/web_service_url_key"
        android:selectAllOnFocus="true"
        android:title="@string/pref_url_title"
        app:defaultValue="@string/web_service_url_default" />
    <SwitchPreference
        android:defaultValue="false"
        android:key="@string/import_key"
        android:title="@string/pref_import_title" />
</PreferenceScreen>
```

(from [T34-Work/ToDo/app/src/main/res/xml/prefs.xml](#))

This adds a SwitchPreference after our existing EditTextPreference.

As with the EditTextPreference, we have android:key, android:title, and android:defaultValue attributes. The first two point to our new string resources. And, we set the default value to false, so the user will need to opt into having this periodic work occur.

Step #2: Observing Preference Changes

With our preference for the server URL, we do not need to do anything special right at the moment the user makes a change. We just get the now-current value of that URL when the user requests an import. With this new preference, though, we need to find out in real time when the user flips that Switch, so we can either schedule or cancel our periodic background import work.

To help with that, add this property to PrefsRepository:

```
private val importKey = context.getString(R.string.import_key)
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/repo/PrefsRepository.kt](#))

This pulls in the value of the `doPeriodicImport` string resource that we added in the previous step.

Then, add this `observeImportChanges()` function:

```
fun observeImportChanges() = channelFlow {
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->
        if (importKey == key) {
            offer(prefs.getBoolean(importKey, false))
        }
    }

    prefs.registerOnSharedPreferenceChangeListener(listener)
    awaitClose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }
}
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/repo/PrefsRepository.kt](#))

`SharedPreferences` has a listener mechanism to find out when preference values get changed, such as via our preference screen. You can call `registerOnSharedPreferenceChangeListener()`, and your listener will be called with the keys that change as they change. You can use the key to pull in the latest value for the preference and do something with it. `unregisterOnSharedPreferenceChangeListener()` removes the listener.

`observeImportChanges()` wraps all of those up in a Flow, using the `channelFlow()` factory function. Our listener will watch for changes to the preference identified by our `importKey` resource. When we get a change, it will `offer()` that change, causing it to be emitted by the Flow that we are creating. When the Flow is no longer being collected, `awaitClose()` is called, and we `unregister` our listener.



You can learn more about channelFlow() in the "Bridging to Callback APIs" chapter of [Elements of Kotlin Coroutines!](#)

Step #3: Adding the Dependency

As with the other Jetpack libraries that we have used, we need to add another dependency to our ever-growing list of dependencies. So, add this line to the dependencies closure of the app/build.gradle file:

```
implementation "androidx.work:work-runtime-ktx:2.6.0"
```

(from [T34-Work/ToDo/app/build.gradle](#))

This specifically pulls in a KTX version of the WorkManager library, so we can better integrate WorkManager with Kotlin, specifically with coroutines.

Step #4: Creating a Stub Worker

Right-click over the com.commonsware.todo.repo package in the java/ directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in ImportWorker, and choose “Class” for the kind. Press **Enter** or **Return** to create the class. Then, replace the class declaration with:

```
package com.commonsware.todo.repo

import android.content.Context
import androidx.work CoroutineWorker
import androidx.work.WorkerParameters

class ImportWorker(context: Context, params: WorkerParameters) :
    CoroutineWorker(context, params) {

    override suspend fun doWork() = TODO()
}
```

A “worker” wraps up our code that will do the work as requested by WorkManager. So, ImportWorker will arrange to import our to-do items from our server.

Specifically, ImportWorker extends CoroutineWorker. A CoroutineWorker is a worker

SCHEDULING WORK

that knows how to integrate with coroutines. We override a `dowork()` function that does the work that we want. `dowork()` is a suspend function, and `WorkManager` can arrange to do that work on a background thread using ordinary coroutines. The `CoroutineWorker` constructor needs a `Context` and a `WorkerParameters` — the latter contains information about the work that we are to perform and is used in advanced `WorkManager` scenarios.

Right now, our `ImportWorker` would crash if we ran it, courtesy of the `TODO()` function. We will fix that shortly.

Step #5: Injecting Into the Worker

To do the import, we need:

- The `PrefsRepository`, to get the URL to use; and
- The `ToDoRepository`, to perform the actual import

Elsewhere in the app, we get those from Koin. However, elsewhere in the app, we tend to be working with either Koin-defined objects (e.g., repositories) or common Android/Jetpack classes (`Fragment`, `ViewModel`, etc.). A `CoroutineWorker` is none of those.

However, Koin still supports classes like `CoroutineWorker` (and our `ImportWorker` subclass). To do this, just add `KoinComponent` as an interface to `ImportWorker`:

```
class ImportWorker(context: Context, params: WorkerParameters) :  
    CoroutineWorker(context, params), KoinComponent {
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/repo/ImportWorker.kt](#))

This lets us use by `inject()` property delegates to retrieve objects from Koin. So, add these two properties to `ImportWorker` to pull in our repositories:

```
private val repo: ToDoRepository by inject()  
private val prefs: PrefsRepository by inject()
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/repo/ImportWorker.kt](#))

At runtime, by `inject()` will reach into the Koin object manager and retrieve our desired objects.

Step #6: Doing the Work

To wrap up our `ImportWorker` implementation, replace the stub `doWork()` function with this one:

```
override suspend fun doWork() = try {
    repo.importItems(prefs.loadWebServiceUrl())

    Result.success()
} catch (ex: Exception) {
    Log.e("ToDo", "Exception importing items in doWork()", ex)
    Result.failure()
}
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/repo/ImportWorker.kt](#))

`doWork()` now does the same basic thing as our manual import logic in `RosterMotor`, except that `WorkManager` provides the `CoroutineScope`.

`doWork()` needs to return a `Result` object. `Result.success()` and `Result.failure()` give us `Result` objects representing those statuses, and so our `doWork()` returns one depending upon whether or not we had an exception.

So, if `WorkManager` arranges to call `doWork()` on our `ImportWorker`, we will import the to-do items from our server.

Step #7: Scheduling the Work

Of course, it would be nice if we actually *taught* `WorkManager` to call `doWork()` on our `ImportWorker`. Otherwise, `ImportWorker` will be unused.

We want to teach `WorkManager` to check for new to-do items every so often. And that check should be automatic — other than opting into the checks via our new `SwitchPreference`, the use should not need to do anything else. So, one place we could add in the `WorkManager` configuration would be `ToDoApp`, as it could watch for changes in our `SwitchPreference` and update `WorkManager` to match.

This means that `ToDoApp` will not only need to *configure* Koin but to *get objects from Koin*. To do that, we use the same `KoinComponent` approach that we did with `ImportWorker`. So, add the `KoinComponent` interface to the `ToDoApp` declaration:

```
class ToDoApp : Application(), KoinComponent {
```

SCHEDULING WORK

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Now, `ToDoApp()` can use `by inject()`, the way `ImportWorker` did.

Then, add this line above the `ToDoApp` declaration:

```
private const val TAG_IMPORT_WORK = "doPeriodicImport"
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This sets up a constant `String` that we will reference shortly.

Next, add this function to `ToDoApp`:

```
private fun scheduleWork() {
    val prefs: PrefsRepository by inject()
    val appScope: CoroutineScope by inject(named("appScope"))
    val workManager = WorkManager.getInstance(this)

    appScope.launch {
        prefs.observeImportChanges().collect {
            if (it) {
                val constraints = Constraints.Builder()
                    .setRequiredNetworkType(NetworkType.CONNECTED)
                    .build()
                val request =
                    PeriodicWorkRequestBuilder<ImportWorker>(15, TimeUnit.MINUTES)
                        .setConstraints(constraints)
                        .addTag(TAG_IMPORT_WORK)
                        .build()

                workManager.enqueueUniquePeriodicWork(
                    TAG_IMPORT_WORK,
                    ExistingPeriodicWorkPolicy.REPLACE,
                    request
                )
            } else {
                workManager.cancelAllWorkByTag(TAG_IMPORT_WORK)
            }
        }
    }
}
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This will require an extension function for `collect()`:

SCHEDULING WORK

```
import kotlinx.coroutines.flow.collect
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

We use by `inject()` to obtain our `PrefsRepository` and our `appScope`-named `CoroutineScope`. We then get a `WorkManager` instance via the `getInstance()` factory method.

Then, we use `appScope` to launch() a coroutine that calls `collect()` on the `Flow` returned by `observeImportChanges()` on `PrefsRepository`. Each time the `SwitchPreference` changes, our `collect()` lambda expression will be executed.

In there, we branch based on the Boolean value that we get, scheduling the work if it is true (i.e., the switch was checked) and canceling the work if it is false (i.e., the switch was unchecked).

To schedule the work, we need to establish some constraints, telling `WorkManager` any requirements of our environment that should be met before bothering to have us do the work. For that, we use `Constraints.Builder`. Our one constraint is that we need an Internet connection, so we can reach our server. To specify that, we use `setRequiredNetworkType(NetworkType.CONNECTED)` on the `Constraints.Builder` to say that we need an active network connection. We then `build()` the resulting `Constraints`. There are other possible constraints that we could set (e.g., the device must be idle), but this all that we need.

We then create a `PeriodicWorkRequest` using a `PeriodicWorkRequestBuilder`. A `PeriodicWorkRequest` represents what should be done and when it should be done. In our case, we are saying that:

- Every 15 minutes ((15, `TimeUnit.MINUTES`))
- ...and if the constraints are met (`setConstraints(constraints)`)
- ...invoke `doWork()` on our `ImportWorker`
- ...and tag this work using that `TAG_IMPORT_WORK` value that we defined a moment ago

Then, to actually schedule the work, we call `enqueueUniquePeriodicWork()` on the `WorkManager` instance. Here, “enqueue” means that we want to add this `PeriodicWorkRequest` to the roster of work to be performed, and “unique” means “if there is already some work with our unique name, resolve it using the supplied policy”. In our case, the unique name is that same `TAG_IMPORT_WORK` value (though it could be something else if we wanted), and the policy is `REPLACE` (so if we try scheduling a duplicate piece of work, cancel the existing one).

SCHEDULING WORK

Cancelling the work when the switch is toggled off is much simpler: just call `cancelAllWorkByTag()` supplying the tag used in the `PeriodicWorkRequestBuilder`.

Finally, modify `onCreate()` of `ToDoApp` to call `scheduleWork()` after configuring Koin:

```
override fun onCreate() {
    super.onCreate()

    startKoin {
        androidLogger()
        androidContext(this@ToDoApp)
        modules(koinModule)
    }

    scheduleWork()
}
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

The net effect is that when the app starts up, we start watching for changes in the `SwitchPreference`, and we schedule or cancel work based on those changes. Note that `observeImportChanges()` does *not* emit the *existing* value of the `SwitchPreference`, only changes while the app is running. That's fine, because `WorkManager` periodic requests are durable and live after our process is terminated, so we only need to teach `WorkManager` about changes in what we want the work to be.

Step #8: Trying It Out

Start by clearing out all to-do items, deleting them one at a time or uninstalling and reinstalling the app. This will help make it easier for you to see the results.

If you run the app and tap on the “Settings” overflow menu item, you will see our two preferences. In the previous tutorial, we had you temporarily put a bad URL in the “Web service URL” preference — if you did not fix that, do so now.

Then, toggle the “Import periodically” switch to the on position, and use back navigation to return to the to-do roster. You should see imported to-do items, as `WorkManager` will do our work immediately, then again every 15 minutes (or thereabouts).

If you delete the imported items, exit out of the app, and return to the app later, if

SCHEDULING WORK

enough time elapsed and you had an Internet connection, you should see the imported items again, as our ImportWorker was asked to do its work again.

At this point, go back into “Settings” and uncheck the SwitchPreference, as you really do not need to be making lots of requests for the same set of to-do items. In a real app and a real Web service, the mix of to-do items might be changing (e.g., the user used a desktop Web browser to add or remove items), and so importing every so often might make sense.

Final Results

Our final res/values/strings.xml file should look like:

```
<resources>
    <string name="app_name">ToDo</string>
    <string name="msg_empty">Click the + icon to add a todo item!</string>
    <string name="msg_empty_filtered">Click the + icon to add a todo item, or change
your filter to show other items</string>
    <string name="menu_about">About</string>
    <string name="is_completed">Item is completed</string>
    <string name="created_on">Created on:</string>
    <string name="menu_edit">Edit</string>
    <string name="desc">Description</string>
    <string name="notes">Notes</string>
    <string name="menu_save">Save</string>
    <string name="menu_add">Add</string>
    <string name="menu_delete">Delete</string>
    <string name="menu_filter">Filter</string>
    <string name="menu_filter_all">All</string>
    <string name="menu_filter_completed">Completed</string>
    <string name="menu_filter_outstanding">Outstanding</string>
    <string name="oops">Sorry! Something went wrong!</string>
    <string name="report_template"><![CDATA[<h1>To-Do Items</h1>
{{#this}}
<h2>{{description}}</h2>
<p>{{#completed}}<b>COMPLETED</b> &mdash; {{/completed}}Created on: {{dateFormat
createdOn}}</p>
<p>{{notes}}</p>
{{/this}}
]]></string>
    <string name="menu_share">Share</string>
    <string name="pref_url_title">Web service URL</string>
    <string name="web_service_url_key">webServiceUrl</string>
    <string name="web_service_url_default">https://commonsware.com/AndExplore/2.0/
items.json</string>
```

SCHEDULING WORK

```
<string name="settings">Settings</string>
<string name="menu_import">Import</string>
<string name="cancel">Cancel</string>
<string name="retry">Retry</string>
<string name="import_error_title">Import Failure</string>
<string name="import_error_message">Something went wrong with the import!</string>
<string name="pref_import_title">Import periodically</string>
<string name="import_key">doPeriodicImport</string>
</resources>
```

(from [T34-Work/ToDo/app/src/main/res/values/strings.xml](#))

And our updated res/xml/prefs.xml should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <EditTextPreference
        android:key="@string/web_service_url_key"
        android:selectAllOnFocus="true"
        android:title="@string/pref_url_title"
        app:defaultValue="@string/web_service_url_default" />
    <SwitchPreference
        android:defaultValue="false"
        android:key="@string/import_key"
        android:title="@string/pref_import_title" />
</PreferenceScreen>
```

(from [T34-Work/ToDo/app/src/main/res/xml/prefs.xml](#))

The revised PrefsRepository should contain:

```
package com.commonsware.todo.repo

import android.content.Context
import android.content.SharedPreferences
import androidx.preference.PreferenceManager
import com.commonsware.todo.R
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.channels.awaitClose
import kotlinx.coroutines.flow.channelFlow
import kotlinx.coroutines.withContext

class PrefsRepository(context: Context) {
    private val prefs = PreferenceManager.getDefaultSharedPreferences(context)
    private val webServiceUrlKey = context.getString(R.string.web_service_url_key)
    private val defaultWebServiceUrl =
```

SCHEDULING WORK

```
    context.getString(R.string.web_service_url_default)
private val importKey = context.getString(R.string.import_key)

suspend fun loadWebServiceUrl(): String = withContext(Dispatchers.IO) {
    prefs.getString(webServiceUrlKey, defaultWebServiceUrl)
    ?: defaultWebServiceUrl
}

fun observeImportChanges() = channelFlow {
    val listener = SharedPreferences.OnSharedPreferenceChangeListener { _, key ->
        if (importKey == key) {
            offer(prefs.getBoolean(importKey, false))
        }
    }
    prefs.registerOnSharedPreferenceChangeListener(listener)
    awaitClose { prefs.unregisterOnSharedPreferenceChangeListener(listener) }
}
}
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/repo/PrefsRepository.kt](#))

The latest app/build.gradle should resemble:

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'androidx.navigation.safeargs.kotlin'
    id 'kotlin-kapt'
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.commonsware.todo"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }

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

SCHEDULING WORK

```
'proguard-rules.pro'  
    }  
}  
  
buildFeatures {  
    viewBinding true  
}  
  
compileOptions {  
    coreLibraryDesugaringEnabled true  
    sourceCompatibility JavaVersion.VERSION_1_8  
    targetCompatibility JavaVersion.VERSION_1_8  
}  
  
kotlinOptions {  
    jvmTarget = '1.8'  
}  
  
packagingOptions {  
    exclude 'META-INF/AL2.0'  
    exclude 'META-INF/LGPL2.1'  
}  
}  
  
dependencies {  
    implementation 'androidx.core:core-ktx:1.6.0'  
    implementation 'androidx.appcompat:appcompat:1.3.1'  
    implementation 'androidx.constraintlayout:constraintlayout:2.1.0'  
    implementation "androidx.recyclerview:recyclerview:1.2.1"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "androidx.preference:preference-ktx:1.1.1"  
    implementation "androidx.work:work-runtime-ktx:2.6.0"  
    implementation 'com.google.android.material:material:1.4.0'  
    implementation "io.insert-koin:koin-android:$koin_version"  
    implementation "com.github.jknack:handlebars:4.1.2"  
    implementation "androidx.room:room-runtime:$room_version"  
    implementation "androidx.room:room-ktx:$room_version"  
    implementation "com.squareup.okhttp3:okhttp:4.9.1"  
    implementation "com.squareup.moshi:moshi:$moshi_version"  
    kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"  
    kapt "androidx.room:room-compiler:$room_version"  
    coreLibraryDesugaring 'com.android.tools:desugar_jdk_libs:1.1.5'  
    testImplementation 'junit:junit:4.13.2'  
    testImplementation "org.mockito:mockito-inline:3.12.1"  
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.2.0"  
    testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
```

SCHEDULING WORK

```
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    androidTestImplementation "androidx.arch.core:core-testing:2.1.0"
    androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.5.1'
}
```

(from [T34-Work/ToDo/app/build.gradle](#))

Our new ImportWorker should look like:

```
package com.commonsware.todo.repo

import android.content.Context
import android.util.Log
import androidx.work CoroutineWorker
import androidx.work.WorkerParameters
import org.koin.core.component.KoinComponent
import org.koin.core.component.inject

class ImportWorker(context: Context, params: WorkerParameters) :
    CoroutineWorker(context, params), KoinComponent {

    private val repo: ToDoRepository by inject()
    private val prefs: PrefsRepository by inject()

    override suspend fun doWork() = try {
        repo.importItems(prefs.loadWebServiceUrl())

        Result.success()
    } catch (ex: Exception) {
        Log.e("ToDo", "Exception importing items in doWork()", ex)
        Result.failure()
    }
}
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/repo/ImportWorker.kt](#))

And our altered ToDoApp should contain:

```
package com.commonsware.todo

import android.app.Application
import android.text.format.DateUtils
import androidx.work.*
import com.commonsware.todo.repo.*
import com.commonsware.todo.report.RosterReport
import com.commonsware.todo.ui.SingleModelMotor
import com.commonsware.todo.ui.roster.RosterMotor
import com.github.jknack.handlebars.Handlebars
```

SCHEDULING WORK

```
import com.github.jknack.handlebars.Helper
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.SupervisorJob
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.launch
import okhttp3.OkHttpClient
import org.koin.android.ext.koin.androidApplication
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.component.KoinComponent
import org.koin.core.component.inject
import org.koin.core.context.startKoin
import org.koin.core.qualifier.named
import org.koin.dsl.module
import java.time.Instant
import java.util.concurrent.TimeUnit

private const val TAG_IMPORT_WORK = "doPeriodicImport"

class ToDoApp : Application(), KoinComponent {
    private val koinModule = module {
        single(named("appScope")) { CoroutineScope(SupervisorJob()) }
        single { ToDoDatabase.newInstance(androidContext()) }
        single {
            ToDoRepository(
                get<ToDoDatabase>().todoStore(),
                get(named("appScope")),
                get()
            )
        }
        single {
            Handlebars().apply {
                registerHelper("dateFormat", Helper<Instant> { value, _ ->
                    DateUtils.getRelativeDateTimeString(
                        androidContext(),
                        value.toEpochMilli(),
                        DateUtils.MINUTE_IN_MILLIS,
                        DateUtils.WEEK_IN_MILLIS, 0
                    )
                })
            }
        }
        single { RosterReport(androidContext(), get(), get(named("appScope"))) }
        single { OkHttpClient.Builder().build() }
        single { ToDoRemoteDataSource(get()) }
        single { PrefsRepository(androidContext()) }
        viewModel {
```

SCHEDULING WORK

```
RosterMotor(
    get(),
    get(),
    androidApplication(),
    get(named("appScope")),
    get()
)
}

viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }

}

override fun onCreate() {
    super.onCreate()

    startKoin {
        androidLogger()
        androidContext(this@ToDoApp)
        modules(koinModule)
    }

    scheduleWork()
}

private fun scheduleWork() {
    val prefs: PrefsRepository by inject()
    val appScope: CoroutineScope by inject(named("appScope"))
    val workManager = WorkManager.getInstance(this)

    appScope.launch {
        prefs.observeImportChanges().collect {
            if (it) {
                val constraints = Constraints.Builder()
                    .setRequiredNetworkType(NetworkType.CONNECTED)
                    .build()
                val request =
                    PeriodicWorkRequestBuilder<ImportWorker>(15, TimeUnit.MINUTES)
                        .setConstraints(constraints)
                        .addTag(TAG_IMPORT_WORK)
                        .build()

                workManager.enqueueUniquePeriodicWork(
                    TAG_IMPORT_WORK,
                    ExistingPeriodicWorkPolicy.REPLACE,
                    request
                )
            } else {
                workManager.cancelAllWorkByTag(TAG_IMPORT_WORK)
            }
        }
    }
}
```

```
    }  
}  
}  
}
```

(from [T34-Work/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

What We Changed

[The entire result of having completed this tutorial](#). In particular, it contains the changed files:

- [app/src/main/res/values/strings.xml](#)
- [app/src/main/res/xml/prefs.xml](#)
- [app/src/main/java/com/commonsware/todo/repo/PrefsRepository.kt](#)
- [app/build.gradle](#)
- [app/src/main/java/com/commonsware/todo/repo/ImportWorker.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)

