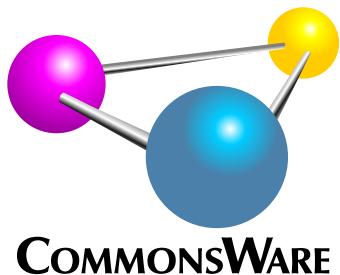


**Version 0.7**

# Exploring Android



**Mark L. Murphy**



**COMMONSWARE**

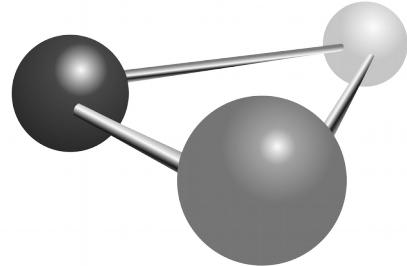
---

---

# Exploring Android

---

*by Mark L. Murphy*



**COMMONSWARE**

Licensed solely for use by Patrocinio Rodriguez

**Exploring Android**  
by Mark L. Murphy

Copyright © 2017-2019 CommonsWare, LLC. All Rights Reserved.  
Printed in the United States of America.

Printing History:  
May 2019: Version 0.6

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.

• <a href="#"><u>Preface</u></a>	◦ How the Book Is Structured .....	ix
	◦ Second-Generation Book .....	x
	◦ Prerequisites .....	x
	◦ About the Updates .....	x
	◦ <b><i>What's New in Version 0.7?</i></b> .....	<b>xi</b>
	◦ Warescription .....	xi
	◦ Book Bug Bounty .....	xii
	◦ Source Code and Its License .....	xiii
	◦ <b><i>Creative Commons and the Four-to-Free (42F) Guarantee</i></b> ....	<b>xiii</b>
• <a href="#"><u>What We Are Building</u></a>	◦ The Purpose .....	1
	◦ The Core UI .....	1
	◦ What We Are Missing .....	5
• <a href="#"><u>Installing the Tools</u></a>	◦ Step #1: Checking Your Hardware .....	9
	◦ <b><i>Step #2: Install Android Studio</i></b> .....	<b>10</b>
	◦ Step #3: Run Android Studio .....	12
• <a href="#"><u>Creating a Starter Project</u></a>	◦ Step #1: Importing the Project .....	19
	◦ Step #2: Setting Up the Emulator AVD .....	23
	◦ <b><i>Step #3: Setting Up the Device</i></b> .....	<b>29</b>
	◦ Step #4: Running the Project .....	33
• <a href="#"><u>Modifying the Manifest</u></a>	◦ Some Notes About Relative Paths .....	37
	◦ Step #1: Supporting Screens .....	38
	◦ Step #2: Blocking Backups .....	39
	◦ <b><i>What We Changed</i></b> .....	<b>40</b>
• <a href="#"><u>Changing Our Icon</u></a>	◦ Step #1: Getting the Replacement Artwork .....	41
	◦ <b><i>Step #2: Changing the Icon</i></b> .....	<b>42</b>
	◦ Step #3: Running the Result .....	52
	◦ What We Changed .....	52
• <a href="#"><u>Adding a Library</u></a>	◦ Step #1: Removing Unnecessary Cruff .....	53

◦ Step #2: Adding Support for RecyclerView .....	54
◦ <b>What We Changed</b> .....	<b>55</b>
• <a href="#"><u>Constructing a Layout</u></a>	
◦ Step #1: Examining What We Have And What We Want .....	57
◦ Step #2: Adding a RecyclerView .....	59
◦ Step #3: Adjusting the TextView .....	65
◦ <b>What We Changed</b> .....	<b>72</b>
• <a href="#"><u>Setting Up the App Bar</u></a>	
◦ Step #1: Defining Some Colors .....	74
◦ Step #2: Adjusting Our Theme .....	78
◦ Step #3: Adding a Toolbar .....	80
◦ Step #4: Adding an Icon .....	86
◦ Step #5: Defining an Item .....	87
◦ Step #6: Enabling Kotlin Synthetic Attributes .....	95
◦ Step #7: Loading Our Options .....	96
◦ Step 8: Trying It Out .....	99
◦ Step #9: Dealing with Crashes .....	100
◦ <b>What We Changed</b> .....	<b>103</b>
• <a href="#"><u>Setting Up an Activity</u></a>	
◦ Step #1: Creating the Stub Activity Class and Manifest Entry .....	105
◦ Step #2: Adding a Toolbar and a WebView .....	108
◦ Step #3: Launching Our Activity .....	113
◦ Step #4: Defining Some About Text .....	114
◦ Step #5: Populating the Toolbar and WebView .....	115
◦ <b>What We Changed</b> .....	<b>117</b>
• <a href="#"><u>Integrating Fragments</u></a>	
◦ <i>Step #1: Creating a Fragment</i> .....	<b>120</b>
◦ <i>Step #2: Updating the Toolbar</i> .....	<b>123</b>
◦ <i>Step #3: Add the KTX Dependency</i> .....	<b>125</b>
◦ Step #4: Displaying the Fragment .....	126
◦ Step #5: Renaming Our Layout Resource .....	128
◦ <b>What We Changed</b> .....	<b>129</b>
• <a href="#"><u>Defining a Model</u></a>	
◦ Step #1: Adding a Stub POJO .....	131
◦ Step #2: Switching to a data Class .....	131
◦ Step #3: Adding the Constructor .....	132
◦ <b>What We Changed</b> .....	<b>133</b>
• <a href="#"><u>Setting Up a Repository</u></a>	
◦ Step #1: Adding the Object .....	136
◦ Step #2: Creating Some Fake Data .....	136
◦ <b>What We Changed</b> .....	<b>137</b>

• <a href="#"><u>Populating Our RecyclerView</u></a>	
◦ Step #1: Adding Data Binding Support .....	140
◦ <b><i>Step #2: Defining a Row Layout</i></b> .....	<b>140</b>
◦ Step #3: Adding a Stub ViewHolder .....	145
◦ Step #4: Creating a Stub Adapter .....	146
◦ Step #5: Comparing Our Models .....	149
◦ Step #6: Adding the Data Binding .....	150
◦ Step #7: Completing the Adapter .....	152
◦ Step #8: Wiring Up the RecyclerView .....	156
◦ <b><i>What We Changed</i></b> .....	<b>159</b>
• <a href="#"><u>Tracking the Completion Status</u></a>	
◦ Step #1: Binding Our RosterRowHolder .....	161
◦ Step #2: Passing the Event Up the Chain .....	162
◦ Step #3: Binding to the Checked Event .....	164
◦ Step #4: Saving the Change .....	165
◦ <b><i>What We Changed</i></b> .....	<b>168</b>
• <a href="#"><u>Preparing for Navigation</u></a>	
◦ <b><i>Step #1: Defining the Version</i></b> .....	<b>170</b>
◦ <b><i>Step #2: Adding the Plugin Dependency</i></b> .....	<b>171</b>
◦ Step #3: Requesting the Plugins .....	172
◦ <b><i>Step #4: Augmenting Our Dependencies</i></b> .....	<b>173</b>
◦ Step #5: Defining Our Navigation Graph .....	174
◦ Step #6: Setting Up a New Activity Layout Resource .....	178
◦ Step #7: Wiring in the Navigation .....	181
◦ <b><i>Step #8: Addressing the Toolbar</i></b> .....	<b>183</b>
◦ <b><i>What We Changed</i></b> .....	<b>185</b>
• <a href="#"><u>Displaying an Item</u></a>	
◦ Step #1: Creating the Fragment .....	187
◦ Step #2: Updating the Navigation Graph .....	188
◦ Step #3: Responding to List Clicks .....	192
◦ <b><i>Step #4: Displaying the (Empty) Fragment</i></b> .....	<b>196</b>
◦ <b><i>Step #5: Teaching Navigation About the Action Bar</i></b> .....	<b>201</b>
◦ Step #6: Creating an Empty Layout .....	204
◦ Step #7: Setting Up Data Binding .....	204
◦ <b><i>Step #8: Adding the Completed Icon</i></b> .....	<b>205</b>
◦ Step #9: Displaying the Description .....	213
◦ Step #10: Showing the Created-On Date .....	216
◦ Step #11: Adding the Notes .....	221
◦ Step #12: Adding Navigation Arguments .....	226
◦ Step #13: Populating the Layout .....	229
◦ <b><i>What We Changed</i></b> .....	<b>231</b>

• <a href="#"><u>Editing an Item</u></a>	
◦ Step #1: Creating the Fragment .....	234
◦ Step #2: Setting Up the Navigation .....	234
◦ Step #3: Setting Up a Menu Resource .....	236
◦ Step #4: Showing the Action Item .....	240
◦ Step #5: Displaying the (Empty) Fragment .....	242
◦ Step #6: Creating an Empty Layout .....	243
◦ Step #7: Setting Up Data Binding .....	244
◦ Step #8: Adding the CheckBox .....	244
◦ <b>Step #9: Creating the Description Field</b> .....	<b>245</b>
◦ Step #10: Adding the Notes Field .....	250
◦ Step #11: Populating the Layout .....	253
◦ <b>What We Changed</b> .....	<b>255</b>
• <a href="#"><u>Saving an Item</u></a>	
◦ Step #1: Adding the Action Bar Item .....	257
◦ Step #2: Replacing the Item .....	261
◦ Step #3: Returning to the Display Fragment .....	262
◦ <b>What We Changed</b> .....	<b>266</b>
• <a href="#"><u>Adding and Deleting Items</u></a>	
◦ Step #1: Trimming Our Repository .....	267
◦ Step #2: Showing an Empty View .....	268
◦ Step #3: Adding an Add Action Bar Item .....	270
◦ Step #4: Launching the EditFragment for Adds .....	273
◦ Step #5: Adjusting Our Save Logic .....	278
◦ Step #6: Hiding the Empty View .....	281
◦ <b>Step #7: Adding a Delete Action Bar Item</b> .....	<b>283</b>
◦ Step #8: Deleting the Item .....	286
◦ Step #9: Fixing the Delete-on-Add Problem .....	287
◦ <b>What We Changed</b> .....	<b>290</b>
• <a href="#"><u>Interlude: So, What's Wrong?</u></a>	
◦ Issues With What We Have .....	291
◦ <b>We Can Do Better</b> .....	<b>294</b>
• <a href="#"><u>Injecting Our Dependencies</u></a>	
◦ Step #1: Adding the Dependencies .....	300
◦ Step #2: Creating a Custom Application .....	303
◦ <b>Step #3: Converting Our Repository</b> .....	<b>304</b>
◦ Step #4: Defining Our Module .....	305
◦ Step #5: Injecting Our Repository .....	307
◦ <b>What We Changed</b> .....	<b>315</b>
• <a href="#"><u>Refactoring Our Code</u></a>	
◦ Step #1: Creating Some Packages .....	317

◦ <i>Step #2: Moving Our Classes</i> .....	<b>319</b>
◦ What We Changed .....	<b>323</b>
• <a href="#"><u>Incorporating a ViewModel</u></a>	
◦ <i>Step #1: Adding the Dependencies</i> .....	<b>326</b>
◦ Step #2: Creating a Stub ViewModel .....	<b>327</b>
◦ Step #3: Getting and Using Our Repository .....	<b>327</b>
◦ Step #4: Depositing a Koin .....	<b>328</b>
◦ <i>Step #5: Injecting the Motor</i> .....	<b>329</b>
◦ What We Changed .....	<b>333</b>
• <a href="#"><u>Making Our Repository Live</u></a>	
◦ <i>Step #1: Adding the Dependency</i> .....	<b>336</b>
◦ Step #2: Making Our Items Live .....	<b>336</b>
◦ Step #3: Observing Our Items .....	<b>338</b>
◦ What We Changed .....	<b>339</b>
• <a href="#"><u>Using a Unidirectional Data Flow</u></a>	
◦ Step #1: Defining a View State .....	<b>342</b>
◦ <i>Step #2: Mapping View States</i> .....	<b>343</b>
◦ Step #3: Consuming View States .....	<b>344</b>
◦ What We Changed .....	<b>345</b>
• <a href="#"><u>Extending the Architecture</u></a>	
◦ Step #1: Making Single Items Live .....	<b>347</b>
◦ Step #2: Making Another Motor .....	<b>349</b>
◦ Step #3: Displaying with a Motor .....	<b>351</b>
◦ Step #4: Making Yet Another Motor .....	<b>352</b>
◦ Step #5: Editing with a Motor .....	<b>353</b>
◦ What We Changed .....	<b>354</b>
• <a href="#"><u>Switching to Coroutines</u></a>	
◦ <i>Step #1: Adding the Dependency</i> .....	<b>357</b>
◦ Step #2: Suspending Our Functions .....	<b>358</b>
◦ <i>Step #3: Launching Those Functions</i> .....	<b>360</b>
◦ What We Changed .....	<b>361</b>
• <a href="#"><u>Testing Our Repository</u></a>	
◦ Step #1: Examine Our Existing Tests .....	<b>365</b>
◦ Step #2: Decide on Instrumented Tests vs. Unit Tests .....	<b>367</b>
◦ <i>Step #3: Adding Some Unit Test Dependencies</i> .....	<b>368</b>
◦ Step #4: Renaming Our Unit Test .....	<b>370</b>
◦ Step #5: Running the Stub Unit Test .....	<b>372</b>
◦ Step #6: Writing and Running Our First Test .....	<b>374</b>
◦ Step #7: Writing and Running More Tests .....	<b>378</b>
◦ What We Changed .....	<b>381</b>
• <a href="#"><u>Testing a Motor</u></a>	

◦ Step #1: Adding Another Unit Test Class .....	384
◦ Step #2: Tweaking Our Motor .....	384
◦ Step #3: Setting Up a Mock Repository .....	387
◦ Step #4: Adding a Test Function .....	390
◦ Step #5: Adding Another Test Function .....	390
◦ <b><i>What We Changed</i></b> .....	<b>391</b>
• <a href="#"><u>Testing a UI</u></a>	
◦ Step #1: Renaming Our Instrumented Test .....	394
◦ Step #2: Running the Stub Instrumented Test .....	394
◦ <b><i>Step #3: Adding Some Instrumented Test Dependencies</i></b> .....	<b>394</b>
◦ Step #4: Initializing Our Repository .....	396
◦ Step #5: Testing Our List .....	397
◦ <b><i>What We Changed</i></b> .....	<b>399</b>
• <a href="#"><u>Getting a Room</u></a>	
◦ <b><i>Step #1: Requesting More Dependencies</i></b> .....	<b>403</b>
◦ Step #2: Defining an Entity .....	405
◦ Step #3: Crafting a DAO .....	407
◦ Step #4: Adding a Database (And Some Type Converters) .....	409
◦ <b><i>Step #5: Creating a Transmogrifier</i></b> .....	<b>411</b>
◦ Step #6: Add Our Database to Koin .....	413
◦ <b><i>What We Changed</i></b> .....	<b>414</b>
• <a href="#"><u>Integrating Room Into the Repository</u></a>	
◦ Step #1: Getting a Database .....	415
◦ Step #2: Fixing the CRUD .....	416
◦ Step #3: Fixing the Instrumented Test .....	418
◦ Step #4: Fixing the Unit Test .....	420
◦ <b><i>What We Changed</i></b> .....	<b>421</b>
• <a href="#"><u>Tracking Our Load Status</u></a>	
◦ Step #1: Adjusting Our Layout .....	423
◦ Step #2: Reacting to the Loaded Status .....	426
◦ <b><i>What We Changed</i></b> .....	<b>427</b>
• <a href="#"><u>Filtering Our Items</u></a>	
◦ Step #1: Updating Our Queries .....	429
◦ Step #2: Defining a FileMode .....	431
◦ Step #3: Consuming a FileMode .....	431
◦ Step #4: Augmenting Our Motor .....	432
◦ Step #5: Repairing Our Test .....	436
◦ <b><i>Step #6: Adding a Checkable Submenu</i></b> .....	<b>439</b>
◦ <b><i>Step #7: Getting Control on Filter Choices</i></b> .....	<b>446</b>
◦ Step #8: Fixing the Empty Text .....	447
◦ <b><i>Step #9: Addressing the Menu Problem</i></b> .....	<b>451</b>

◦ <i>What We Changed</i>	453
• <u>Generating a Report</u>	
◦ <i>Step #1: Adding a Save Action Bar Item</i>	457
◦ <i>Step #2: Making a Save</i>	459
◦ <i>Step #3: Adding Some Handlebars</i>	464
◦ <i>Step #4: Creating the Report</i>	466
◦ <i>Step #5: Writing Where the User Asked</i>	467
◦ <i>Step #6: Saving the Report</i>	469
◦ <i>Step #7: Viewing the Report</i>	471
◦ <i>What We Changed</i>	476
• <u>Sharing the Report</u>	
◦ <i>Step #1: Adding a Share Action Bar Item</i>	477
◦ <i>Step #2: Adding FileProvider</i>	479
◦ <i>Step #3: Caching the Report</i>	484
◦ <i>Step #4: Sharing the Report</i>	486
◦ <i>What We Changed</i>	487
• <u>Collecting a Preference</u>	
◦ <i>Step #1: Adding a Dependency</i>	489
◦ <i>Step #2: Defining a Preference Screen</i>	490
◦ <i>Step #3: Displaying Our Preference Screen</i>	492
◦ <i>Step #4: Adding PrefsFragment to Our Navigation Graph</i>	493
◦ <i>Step #5: Navigating to Our Preference Screen</i>	496
◦ <i>What We Changed</i>	501
• <u>Contacting a Web Service</u>	
◦ <i>Step #1: Adding Some Dependencies</i>	503
◦ <i>Step #2: Requesting a Permission</i>	506
◦ <i>Step #3: Defining Our Response</i>	507
◦ <i>Step #4: Retrieving the Items</i>	509
◦ <i>Step #5: Updating the Local Items</i>	511
◦ <i>Step #6: Fixing the Existing Tests</i>	515
◦ <i>Step #7: Retrieving Our Preference</i>	517
◦ <i>Step #8: Offering the Download Option</i>	518
◦ <i>What We Changed</i>	525

Licensed solely for use by Patrocinio Rodriguez

# Preface

---

Thanks!

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

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

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

## How the Book Is Structured

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

This book is different.

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

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

## Second-Generation Book

Android app development can be divided into two generations:

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

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

## Prerequisites

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

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

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

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

## About the Updates

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

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

## PREFACE

---

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

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

- The Table of Contents in the ebook formats (PDF, EPUB, MOBI/Kindle) shows sections with changes in ***bold-italic*** font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

And, there is the “What’s New” section, just below this paragraph.

## What’s New in Version 0.7?

This update adds four more tutorials:

- Generating a “report” and saving it to a location specified by the user using the Storage Access Framework
- Sharing that “report” using ACTION\_SEND and FileProvider
- Collecting a preference from the user
- Downloading to-do items from a (fake) Web service using OkHttp and Moshi

Also, some dependencies were updated to newer versions.

In addition, the usual round of bugs were fixed.

## Warescription

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

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

## PREFACE

---

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

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

1. The [CommonsBlog](#)
2. The [CommonsWare](#) Twitter feed
3. Opting into emails announcing each book release — log into the [Warescription](#) site and choose Configure from the nav bar
4. Just check back on the [Warescription](#) site every month or two

Subscribers also have access to other benefits, including:

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

## Book Bug Bounty

Find a problem in the book? Let CommonsWare know!

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

By “concrete” problem, we mean things like:

1. Typographical errors
2. Sample applications that do not work as advertised, in the environment described in the book

## PREFACE

---

3. Factual errors that cannot be open to interpretation

By “unique”, we mean ones not yet reported. Be sure to check [the book’s errata page](#), though, to see if your issue has already been reported. One coupon is given per email containing valid bug reports.

We appreciate hearing about “softer” issues as well, such as:

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

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

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to [bounty@commonsware.com](mailto:bounty@commonsware.com).

## 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.

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

Each CommonsWare book edition will be available for use under the [Creative Commons Attribution-Noncommercial-ShareAlike 4.0](#) license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-

## PREFACE

---

Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

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

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit [the Creative Commons Web site](#)

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

# **What We Are Building**

---

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

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

## **The Purpose**

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

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

So, let's build another one!

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

## **The Core UI**

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

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

## 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” action bar item to add a new item:

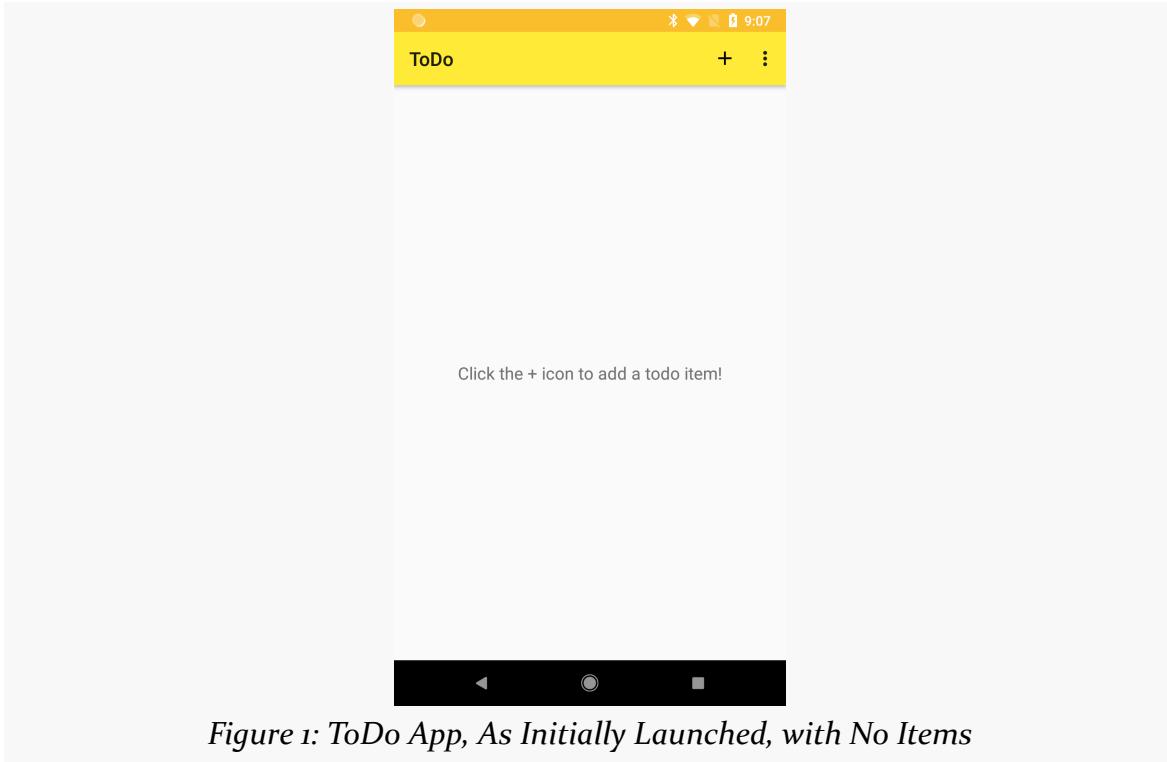


Figure 1: ToDo App, As Initially Launched, with No Items

## 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:

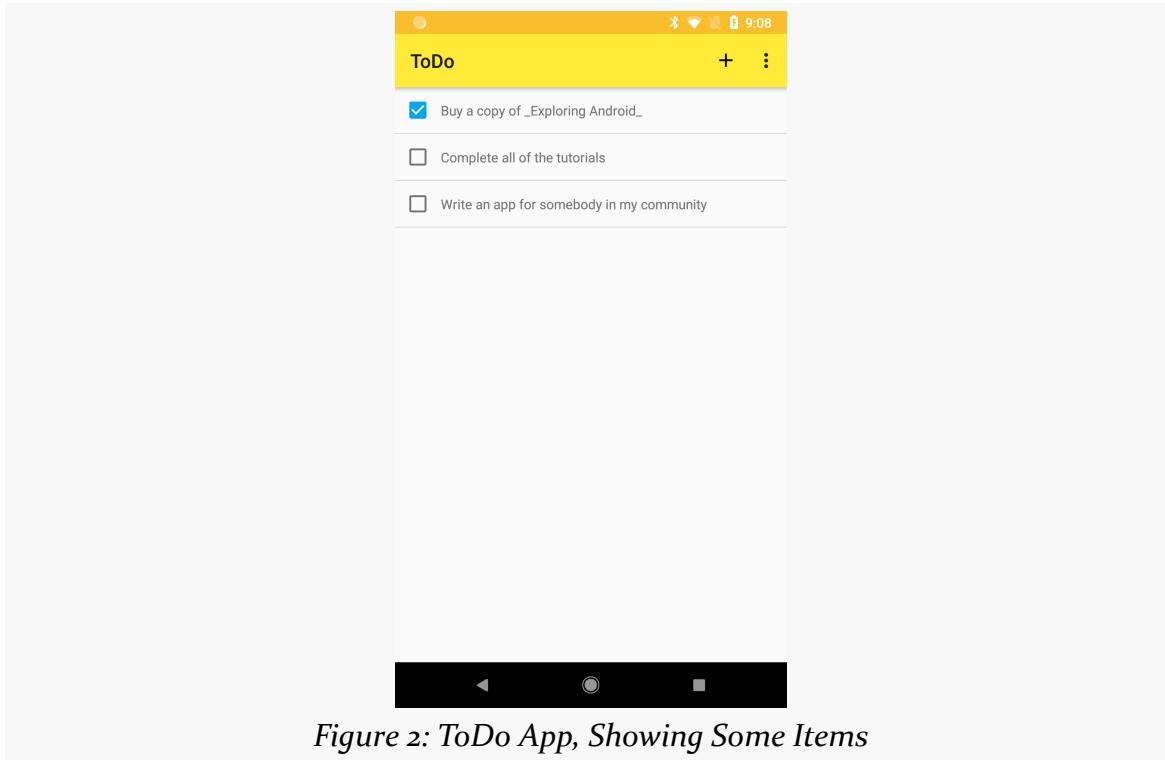


Figure 2: ToDo App, Showing Some Items

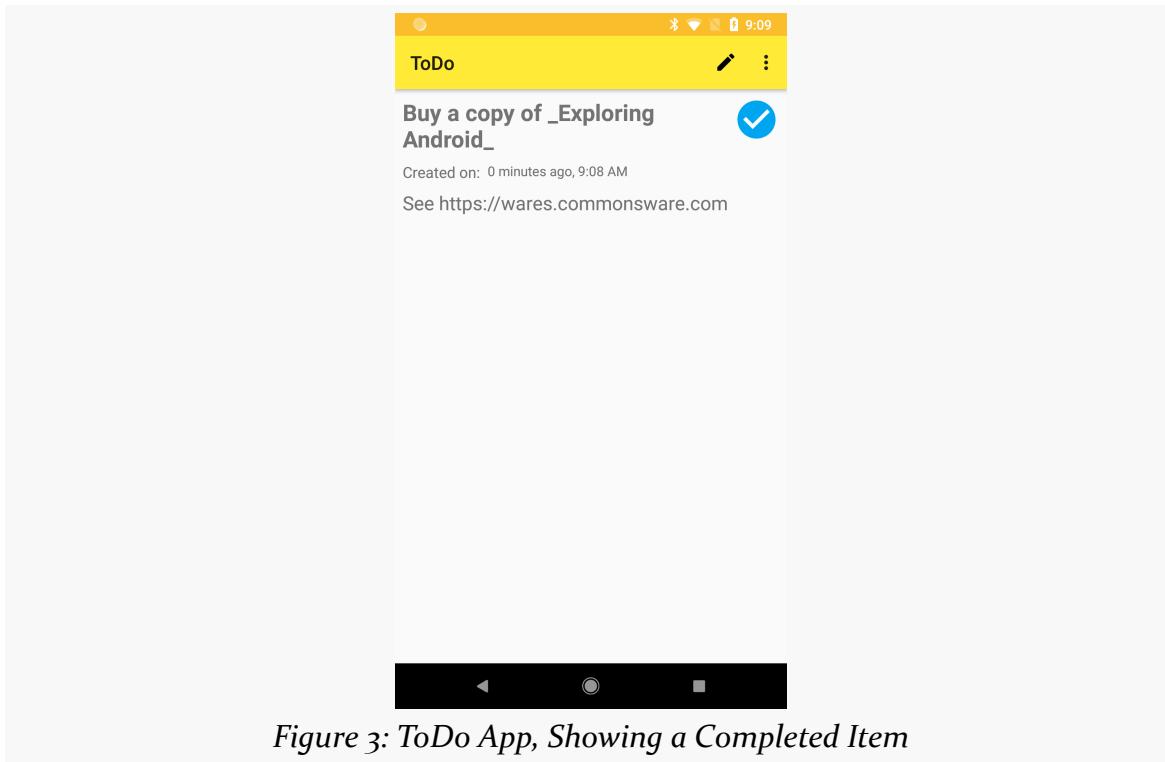
From here, the user can tap the checkbox to quickly mark an item as completed (or un-mark it if needed).

## WHAT WE ARE BUILDING

---

### 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).

## WHAT WE ARE BUILDING

---

### The Editor

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

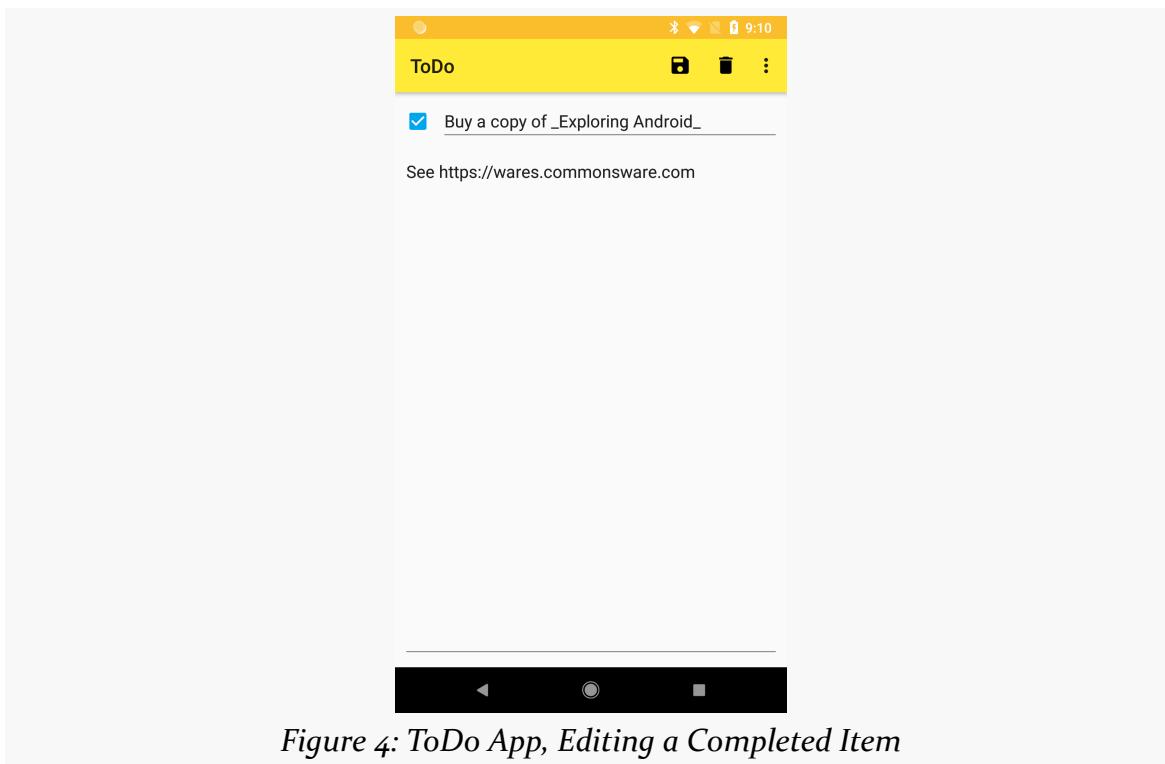


Figure 4: ToDo App, Editing a Completed Item

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

### What We Are Missing

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

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

## **WHAT WE ARE BUILDING**

---

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

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

---

---

## **Phase One: Getting a GUI**

---

---

Licensed solely for use by Patrocinio Rodriguez

# **Installing the Tools**

---

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

## **Step #1: Checking Your Hardware**

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

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

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

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

## INSTALLING THE TOOLS

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

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

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

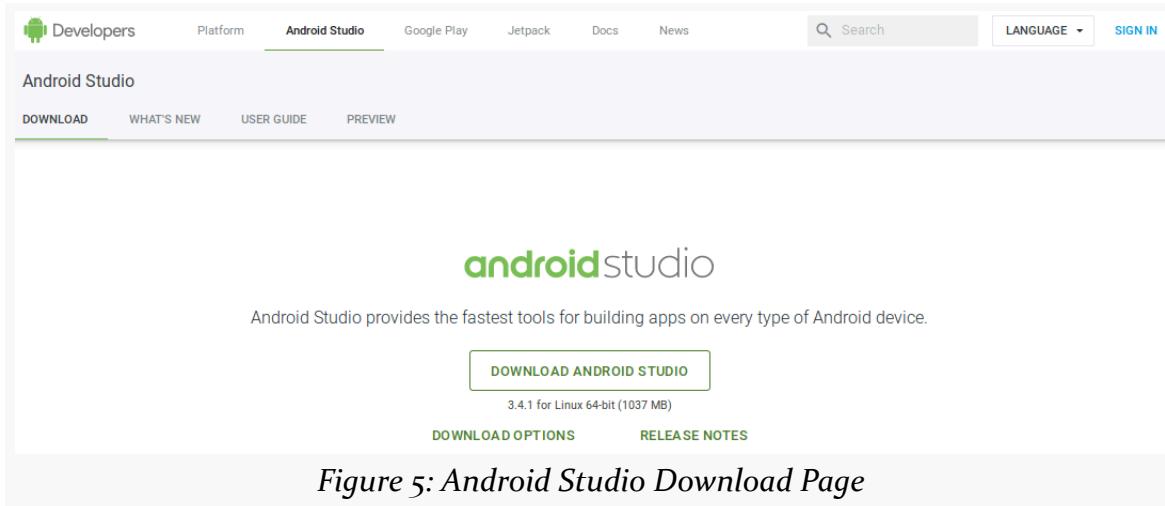
## Step #2: Install Android Studio

At the time of this writing, the current production version of Android Studio is 3.4.1, 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](#).



*Figure 5: Android Studio Download Page*

Or, you can download Android Studio 3.4.1 directly, for:

- [Windows](#)
- [macOS](#)
- [Linux](#)

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

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

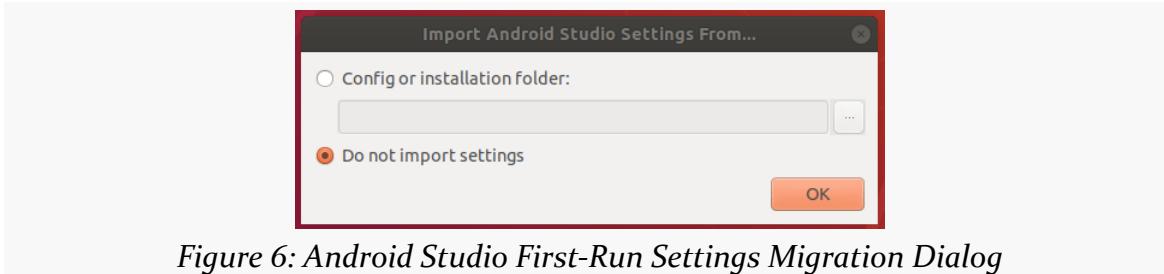
Linux users (and power Windows users) can download a ZIP file, then unZIP it to some likely spot on your hard drive. Android Studio can then be run from the `studio` batch file or shell script in your Android Studio installation's `bin/` directory.

## INSTALLING THE TOOLS

---

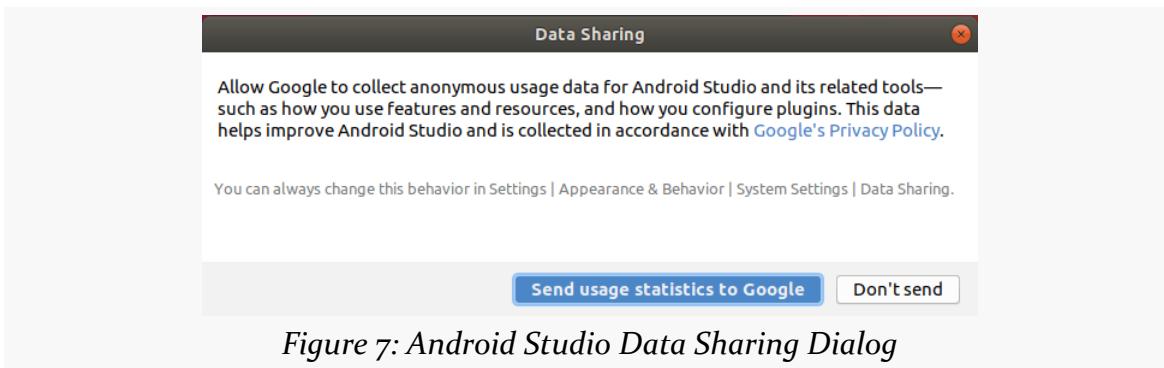
### Step #3: Run Android Studio

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



If you are using Android Studio for the first time, the “Do not import settings” option is the correct choice to make.

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



Click whichever button you wish.

## INSTALLING THE TOOLS

---

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

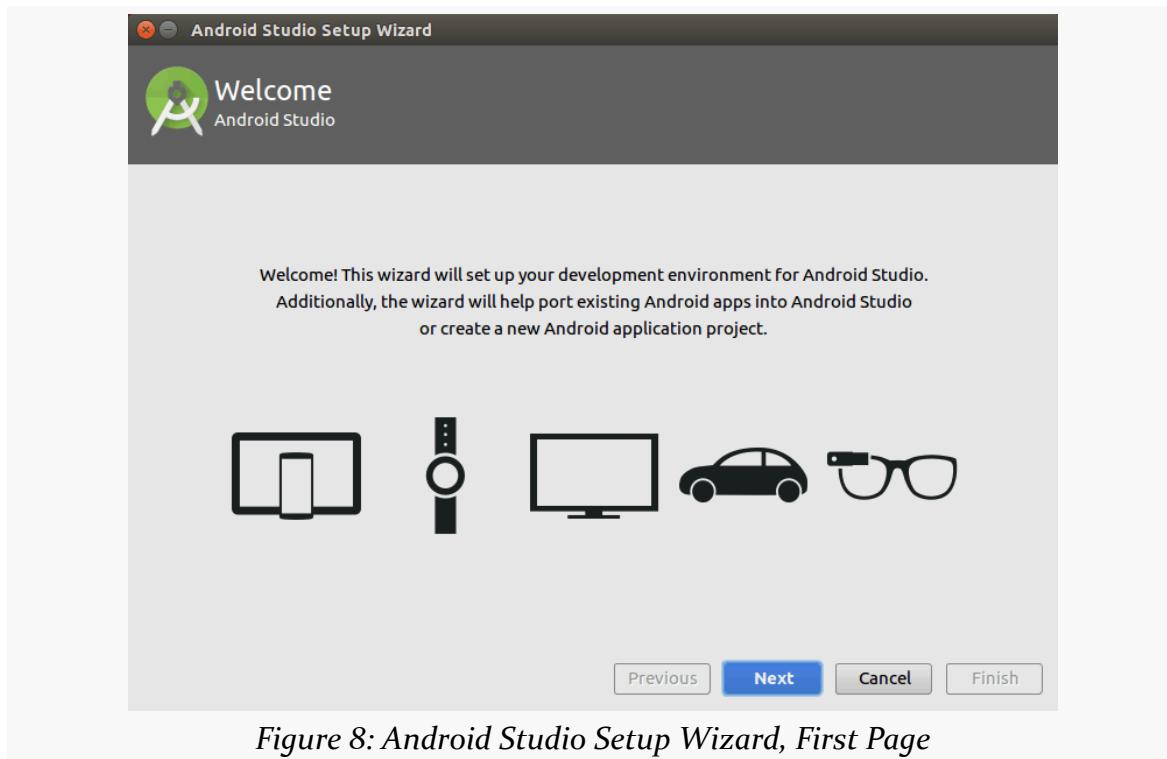
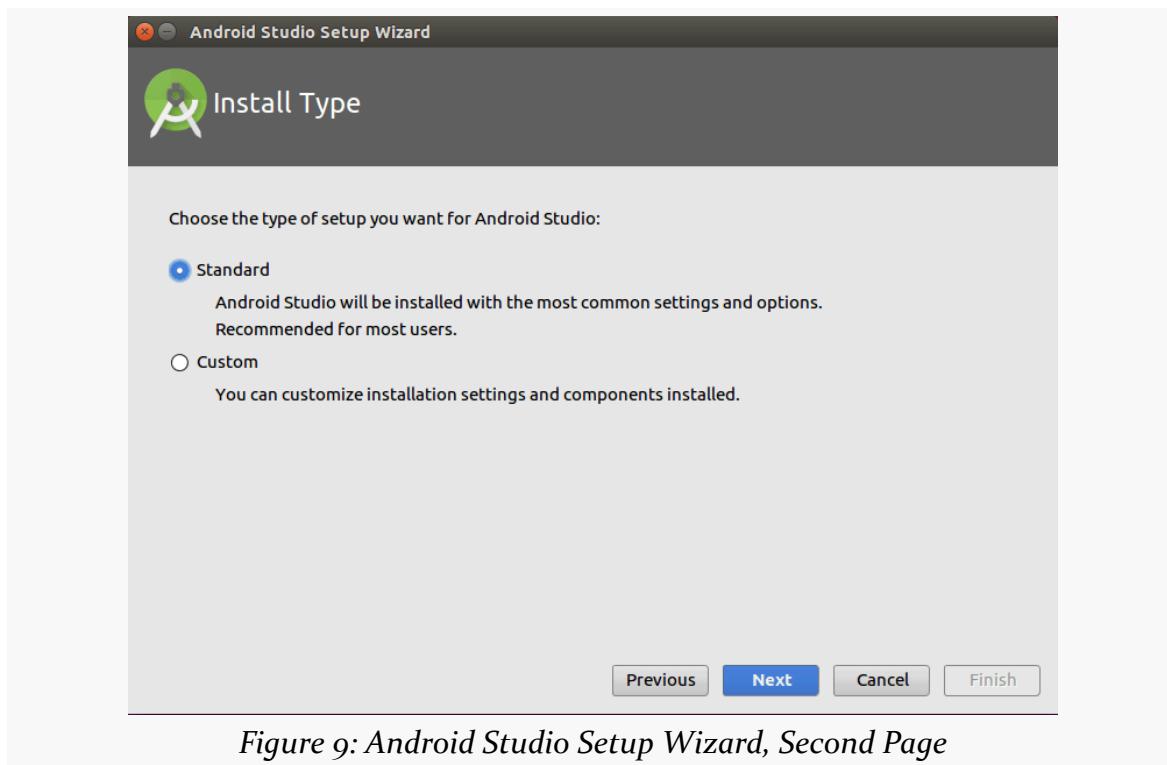


Figure 8: Android Studio Setup Wizard, First Page

## INSTALLING THE TOOLS

---

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



*Figure 9: Android Studio Setup Wizard, Second Page*

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

## 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:

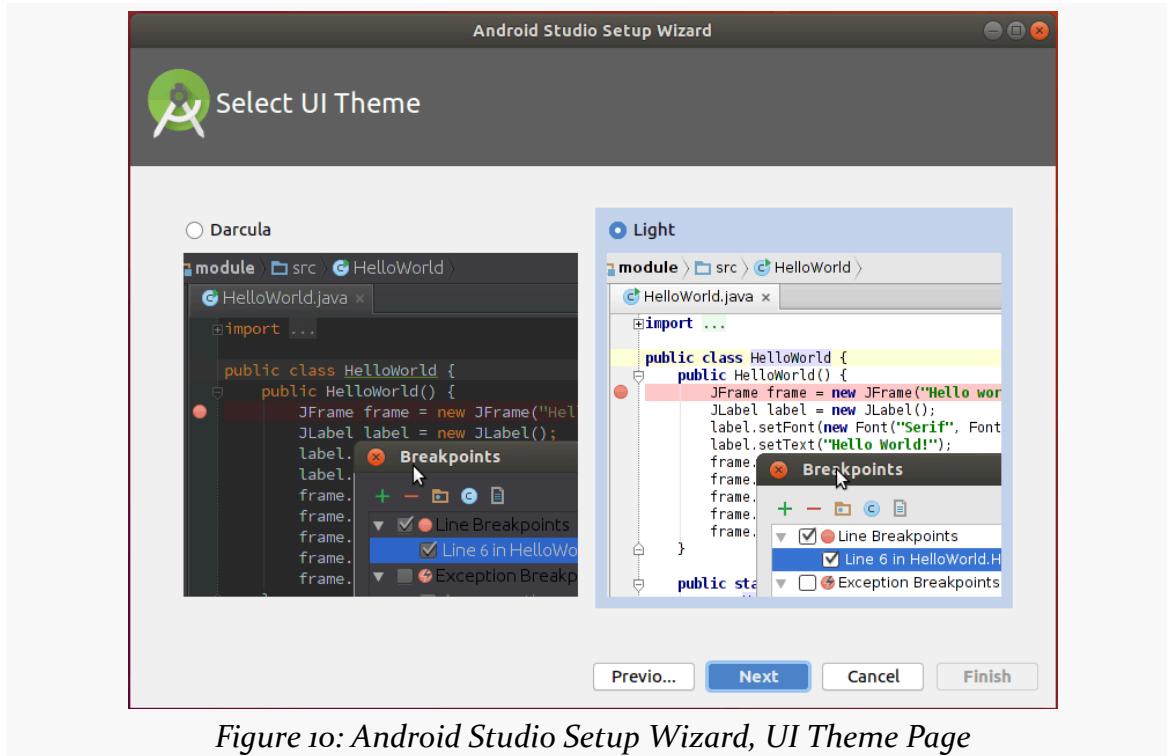
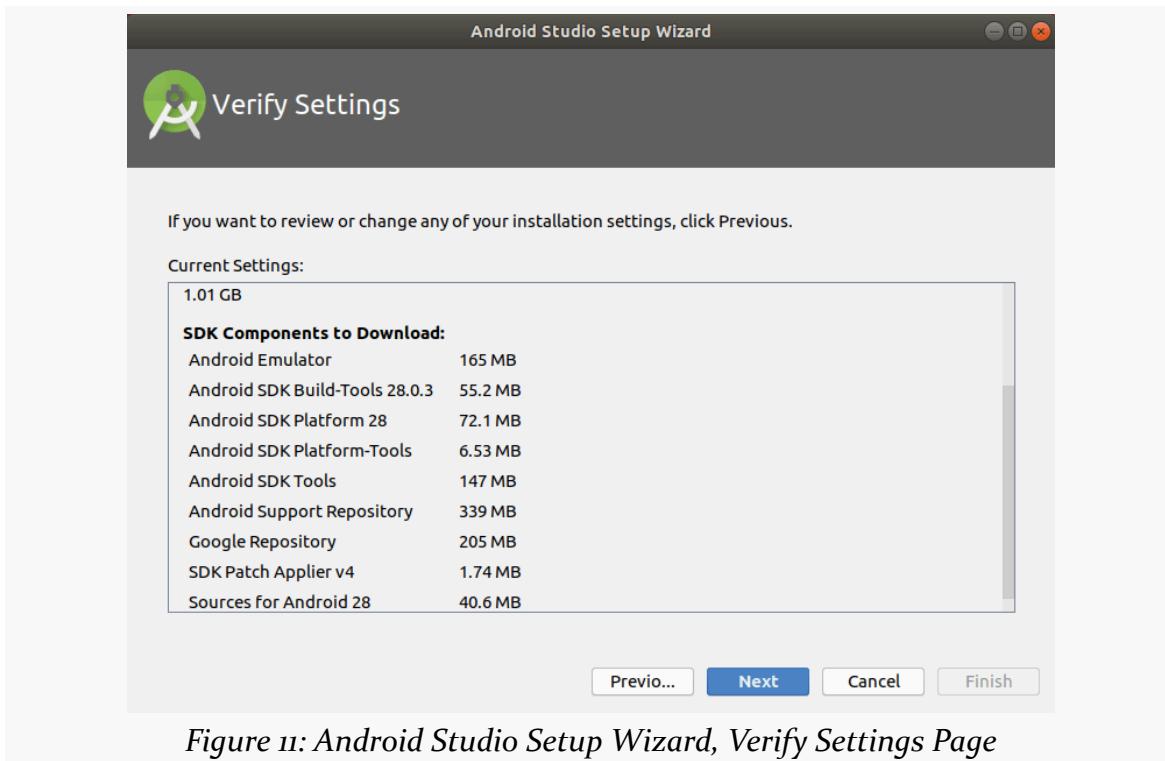


Figure 10: Android Studio Setup Wizard, UI Theme Page

## INSTALLING THE TOOLS

---

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

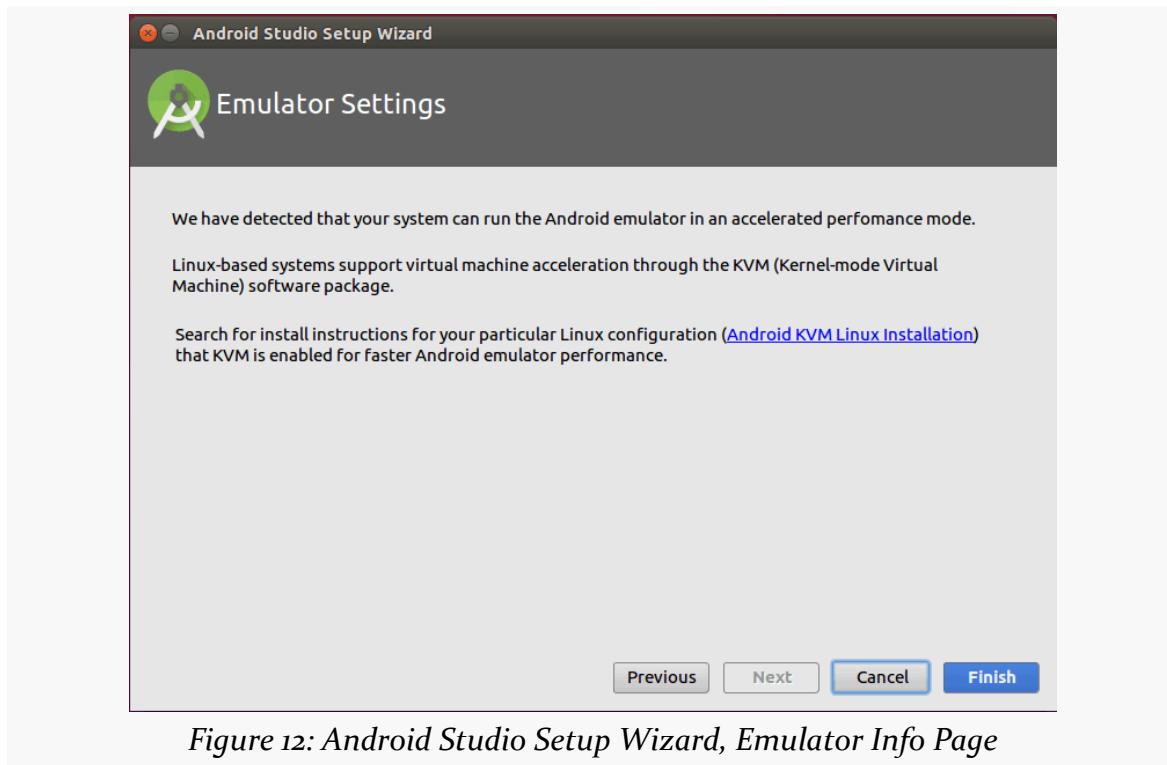


*Figure 11: Android Studio Setup Wizard, Verify Settings Page*

## INSTALLING THE TOOLS

---

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



*Figure 12: Android Studio Setup Wizard, Emulator Info Page*

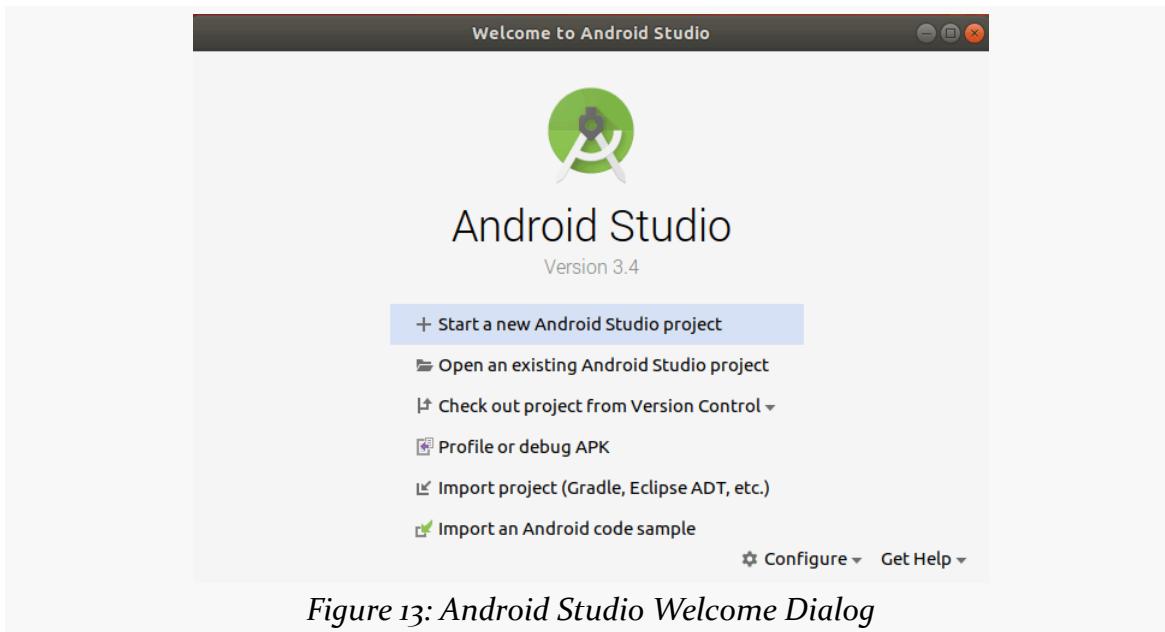
What is explained on this page may not make much sense to you. That is perfectly normal, and we will get into what this page is trying to say later in the book. Just click “Finish” to begin the setup process. This will include downloading a copy of the Android SDK and installing it into a directory adjacent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

When that is done, 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 13: Android Studio Welcome Dialog*

# **Creating a Starter Project**

---

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

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

## **Step #1: Importing the Project**

First, we need an Android project to work in.

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

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

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

## CREATING A STARTER PROJECT

---

```
#Mon Jan 28 17:30:12 EST 2019
distributionBase=GRADLE_USER_HOME
distributionPath=wrapper/dists
zipStoreBase=GRADLE_USER_HOME
zipStorePath=wrapper/dists
distributionUrl=https\://services.gradle.org/distributions/gradle-5.1.1-all.zip
```

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

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

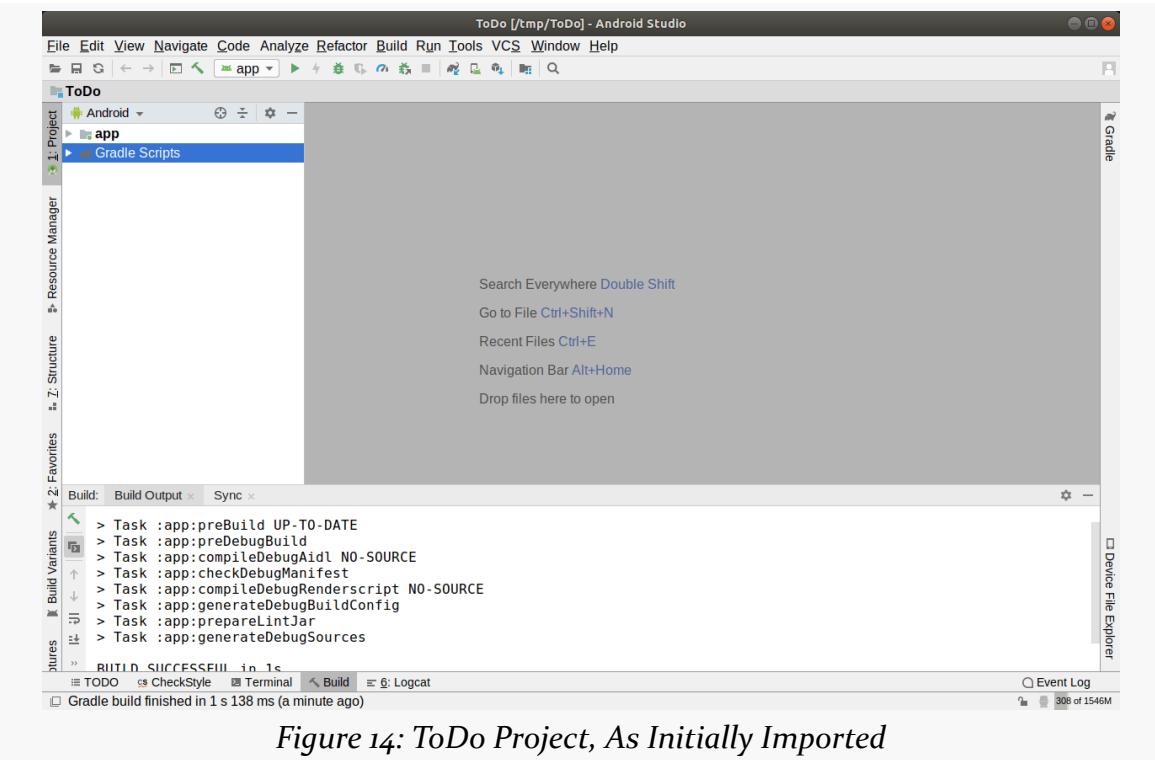
Then, import the project. From the Android Studio welcome dialog — where we left off in the previous tutorial — that is handled by the “Import project (Gradle, Eclipse ADT, etc.)” option. From an existing open Android Studio IDE window, you would use File > New > Import Project... from the main menu.

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

## CREATING A STARTER PROJECT

---

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



*Figure 14: 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”:

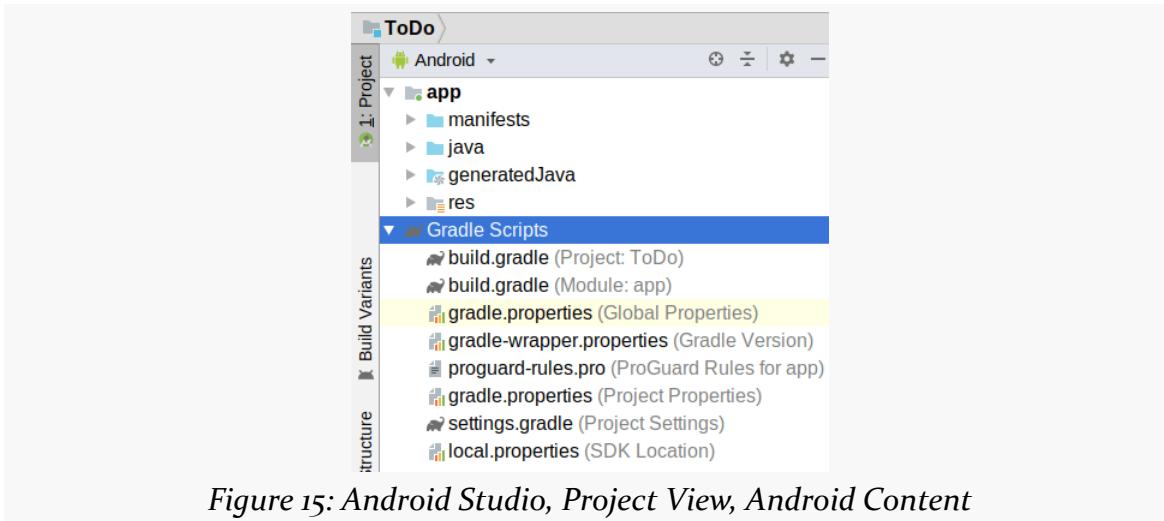


Figure 15: Android Studio, Project View, Android Content

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



Figure 16: Android Studio, Project View, Project Content

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

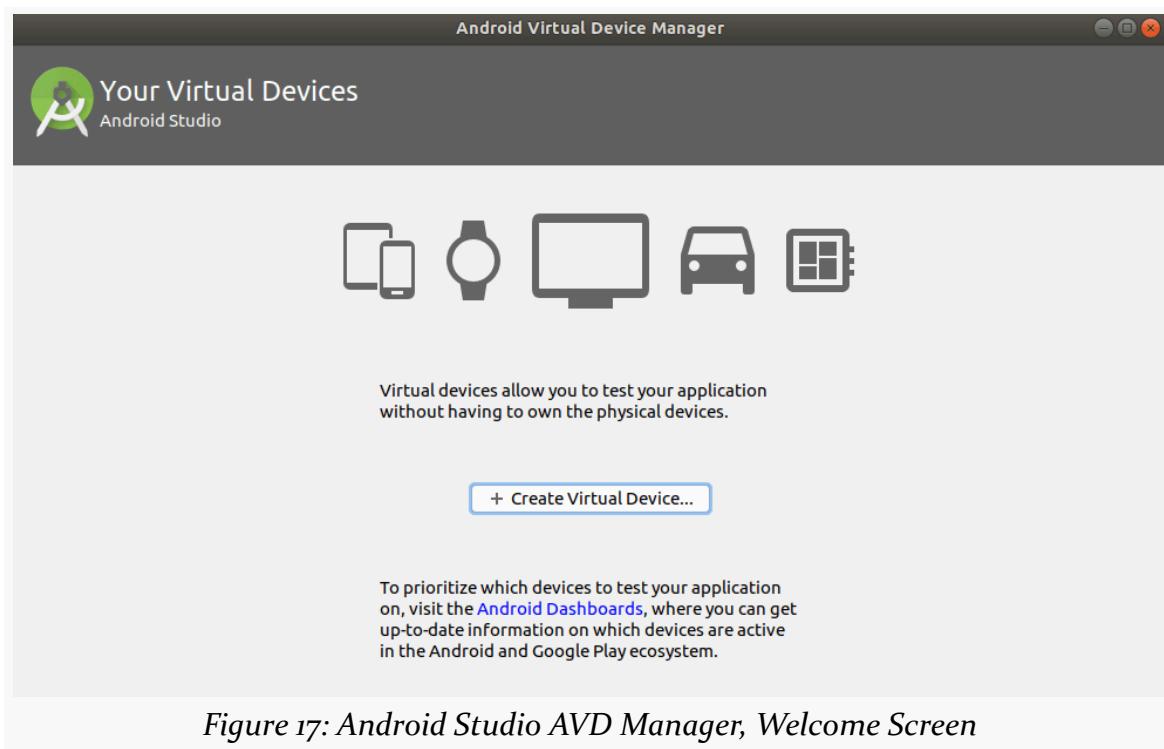
## Step #2: Setting Up the Emulator AVD

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

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

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

You should be taken to a “welcome”-type screen:



## CREATING A STARTER PROJECT

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

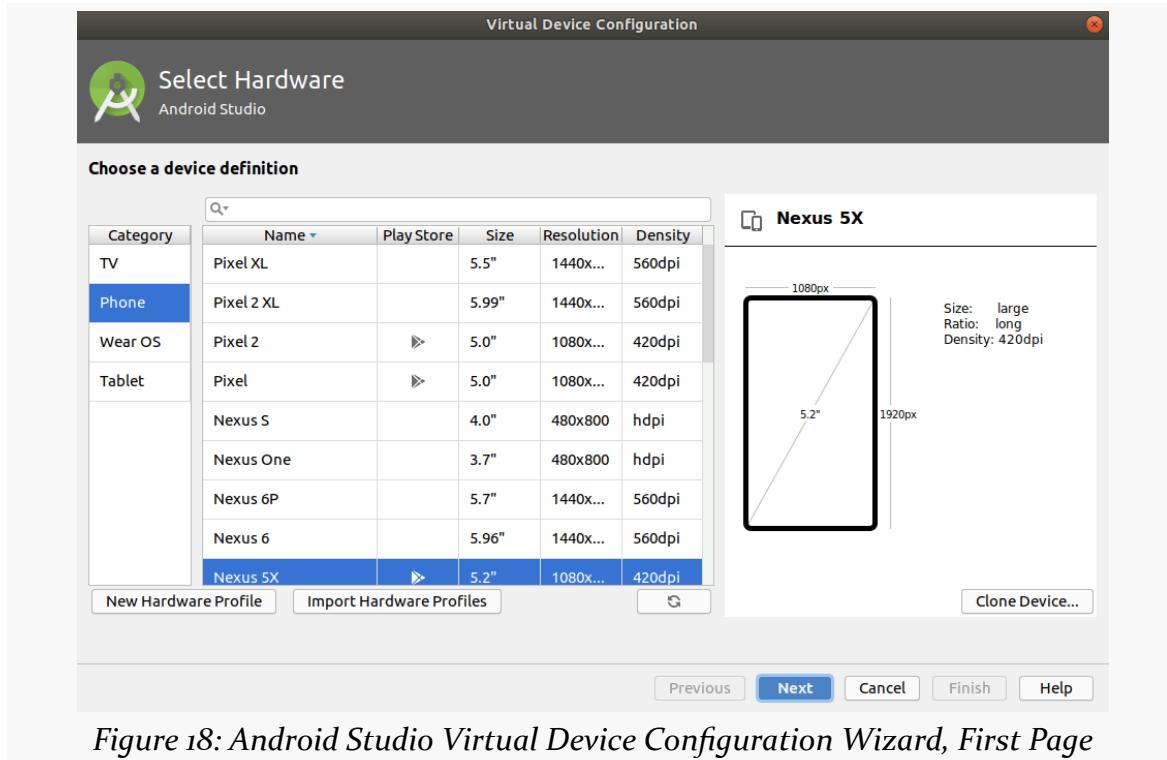


Figure 18: Android Studio Virtual Device Configuration Wizard, First Page

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

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

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

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

## CREATING A STARTER PROJECT

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

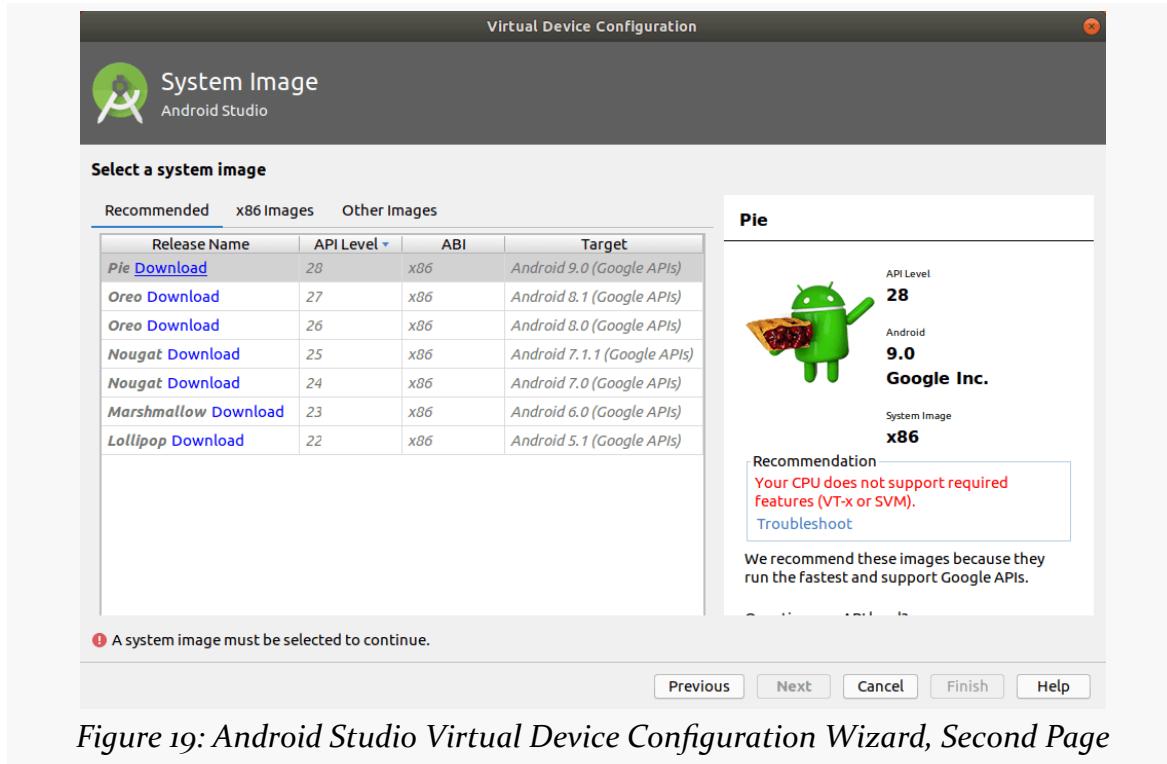


Figure 19: Android Studio Virtual Device Configuration Wizard, Second Page

The emulator images are spread across three tabs:

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

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

## CREATING A STARTER PROJECT

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

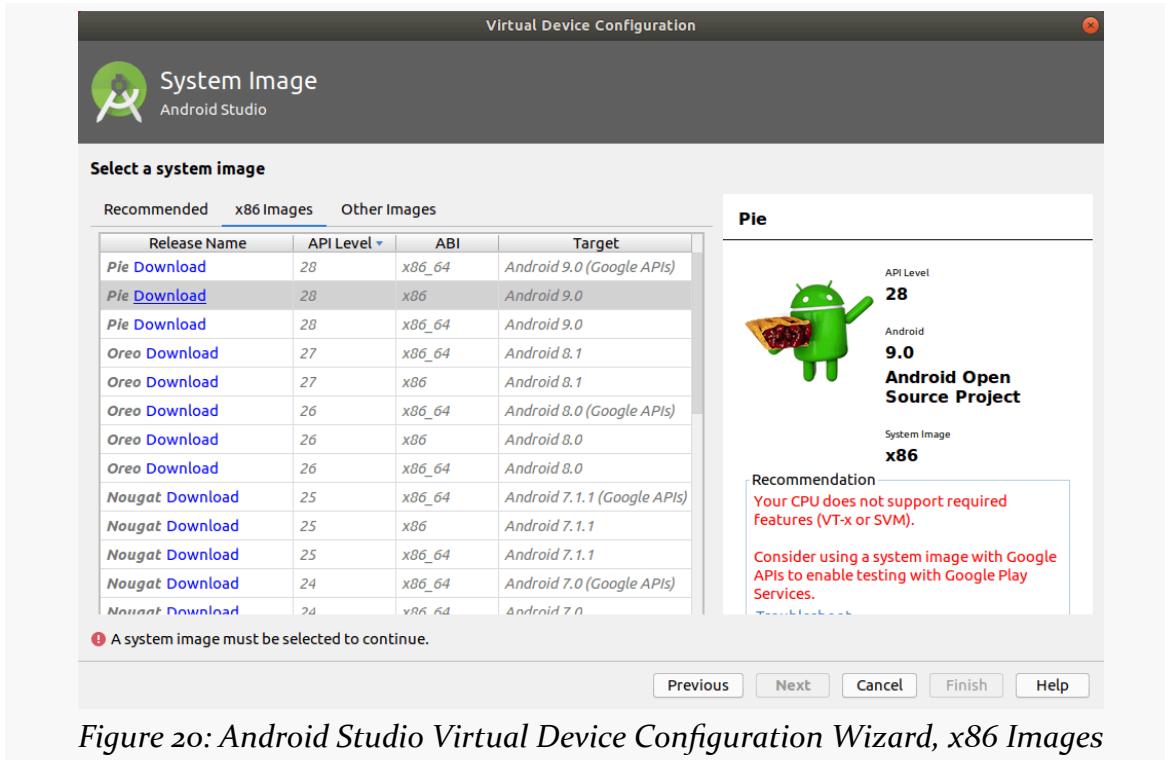


Figure 20: Android Studio Virtual Device Configuration Wizard, x86 Images

## CREATING A STARTER PROJECT

---

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

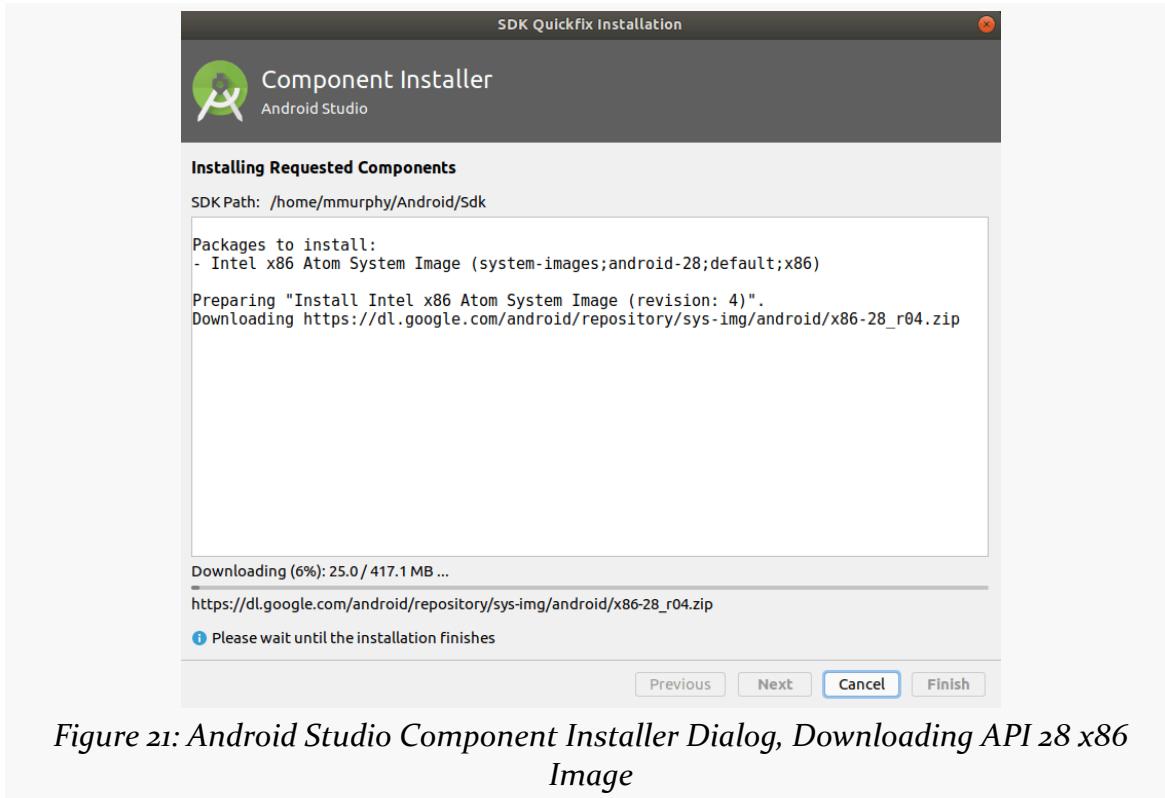


Figure 21: Android Studio Component Installer Dialog, Downloading API 28 x86 Image

## CREATING A STARTER PROJECT

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

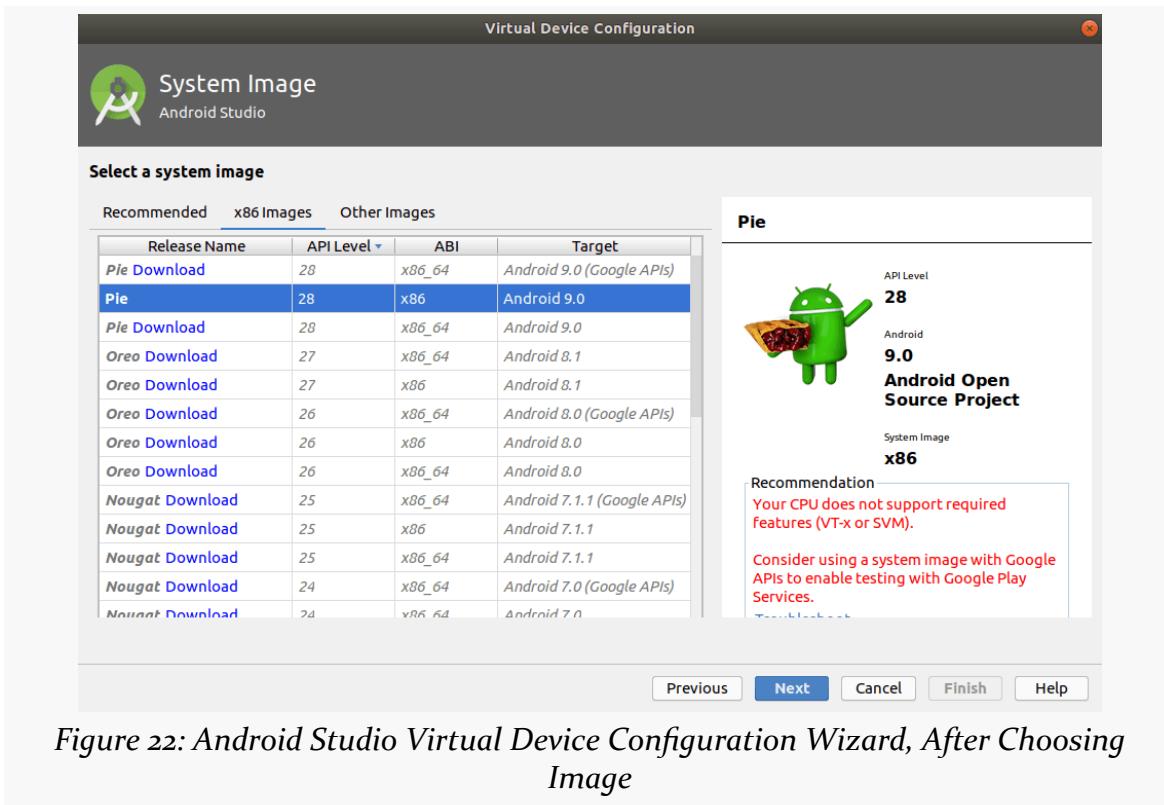


Figure 22: Android Studio Virtual Device Configuration Wizard, After Choosing Image

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

## CREATING A STARTER PROJECT

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

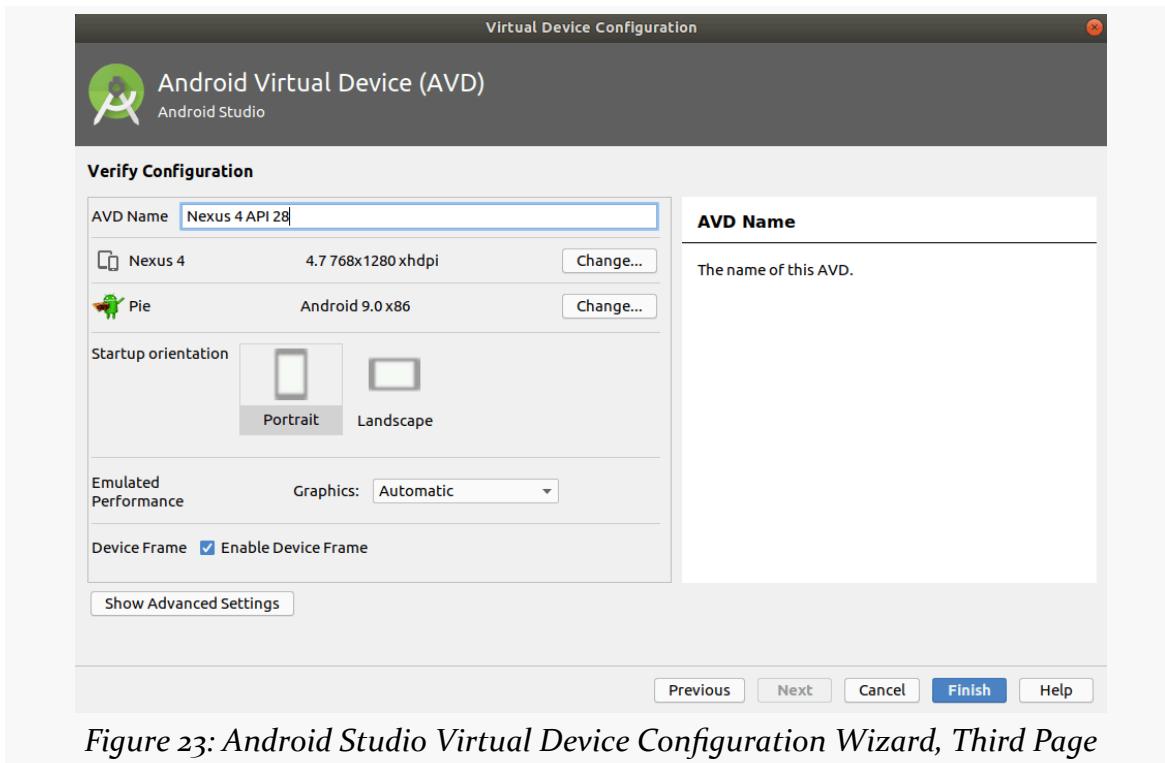


Figure 23: Android Studio Virtual Device Configuration Wizard, Third Page

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

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

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

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

## Step #3: Setting Up the Device

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

## CREATING A STARTER PROJECT

---

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

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

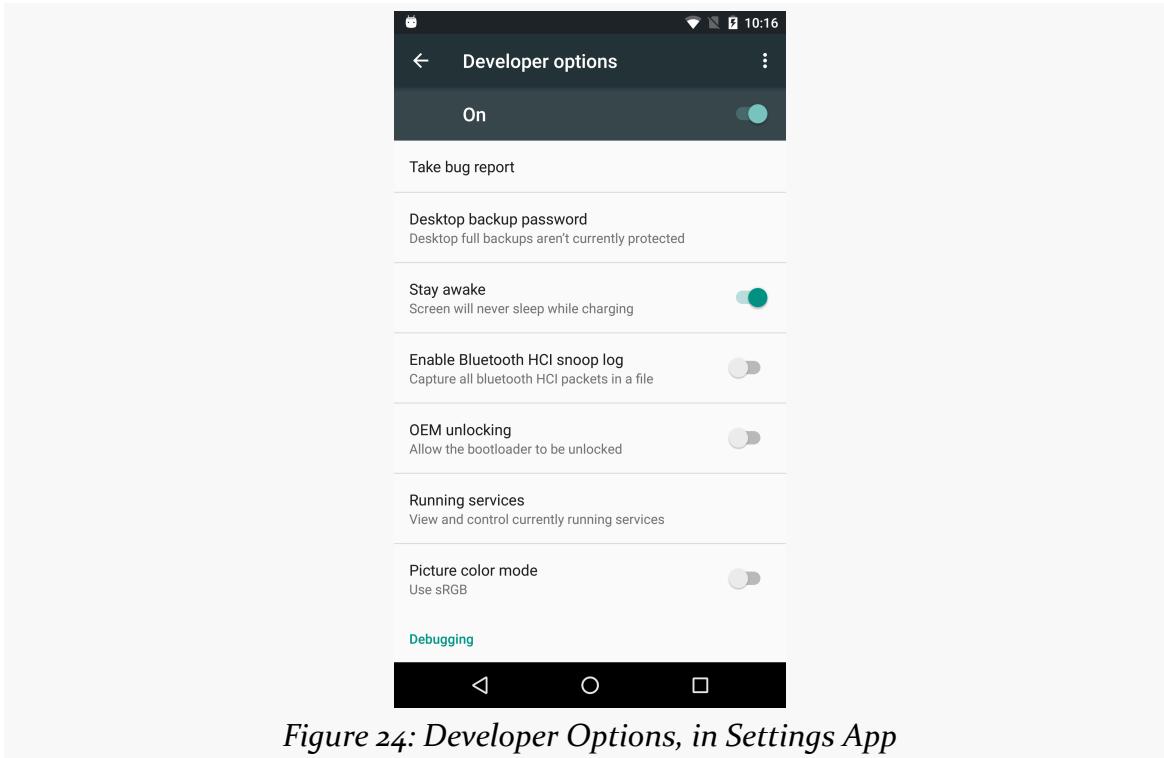


Figure 24: Developer Options, in Settings App

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

## 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:

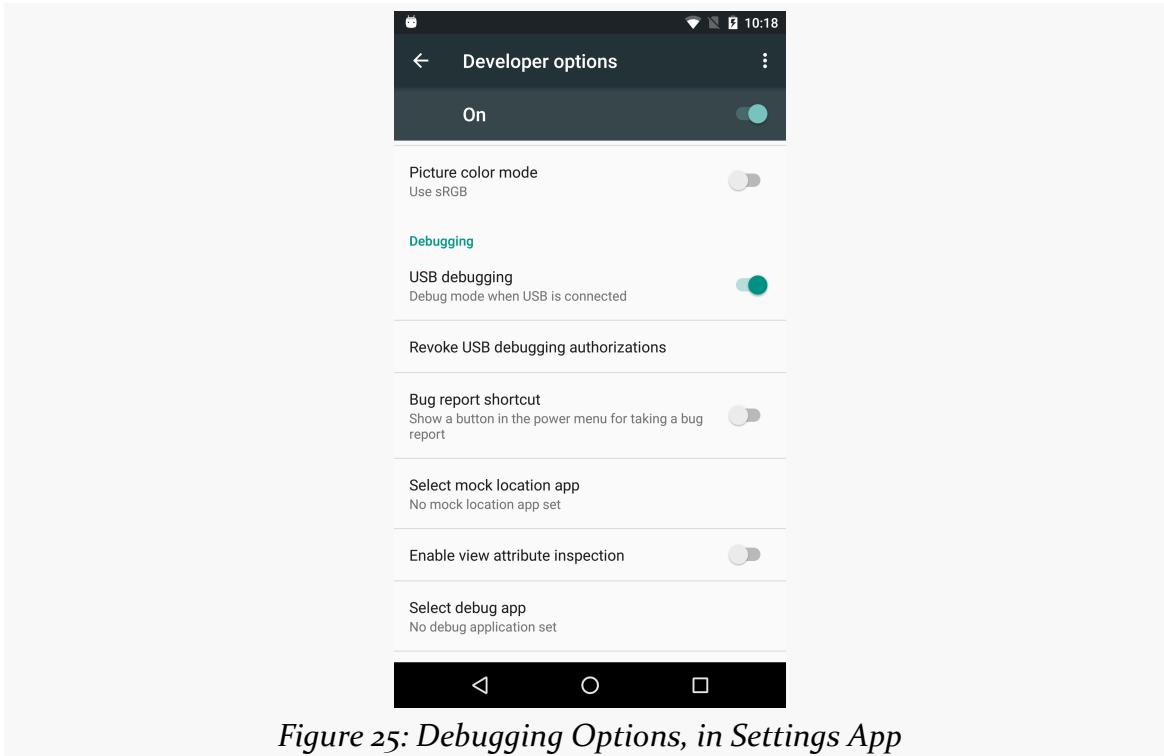


Figure 25: Debugging Options, in Settings App

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

## 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:

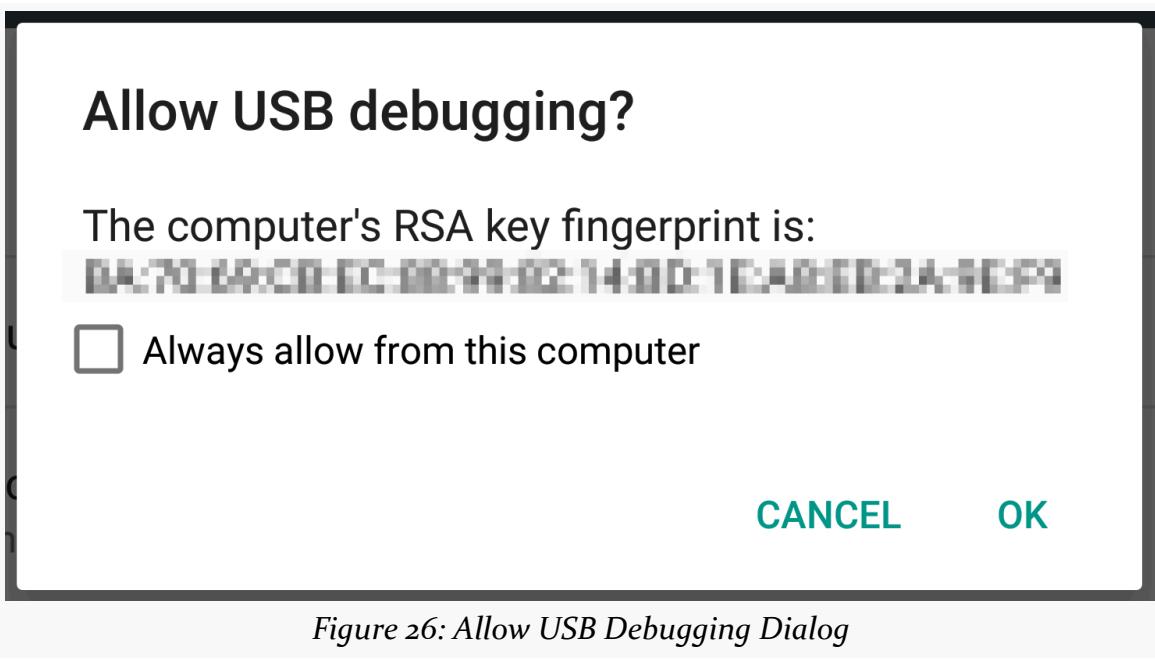


Figure 26: Allow USB Debugging Dialog

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

### Windows

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

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

#### Windows Update

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

### Standard Android Driver

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

### Manufacturer-Supplied Driver

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

### macOS and Linux

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.

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



Figure 27: Android Studio Toolbar, Showing Run Button

You will then be presented with a dialog indicating where you want the app to run:

## CREATING A STARTER PROJECT

---

on some existing device or emulator, or on some newly-launched emulator:

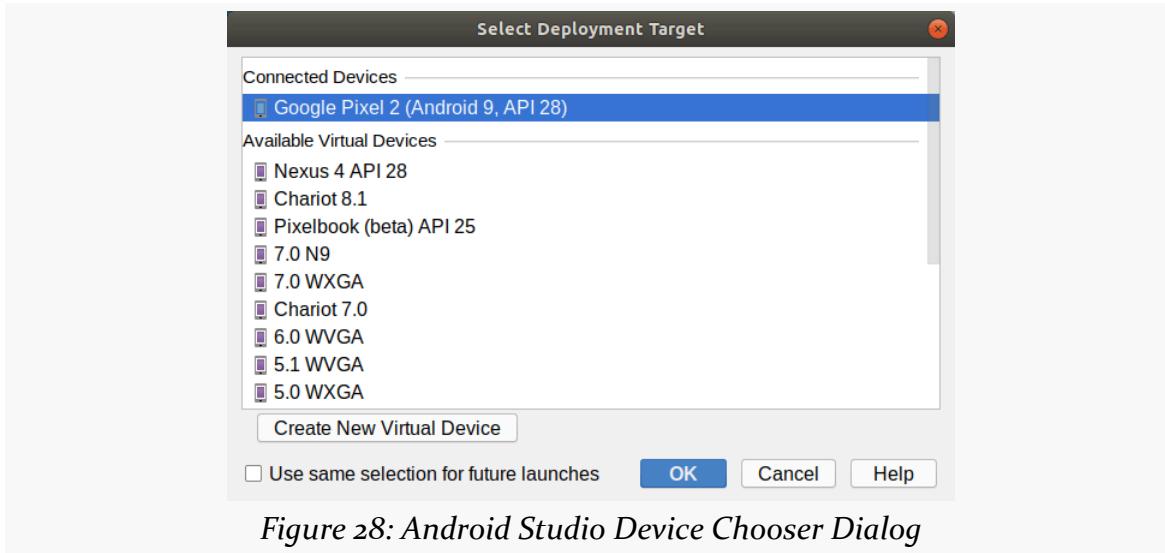


Figure 28: Android Studio Device Chooser Dialog

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

## CREATING A STARTER PROJECT

---

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

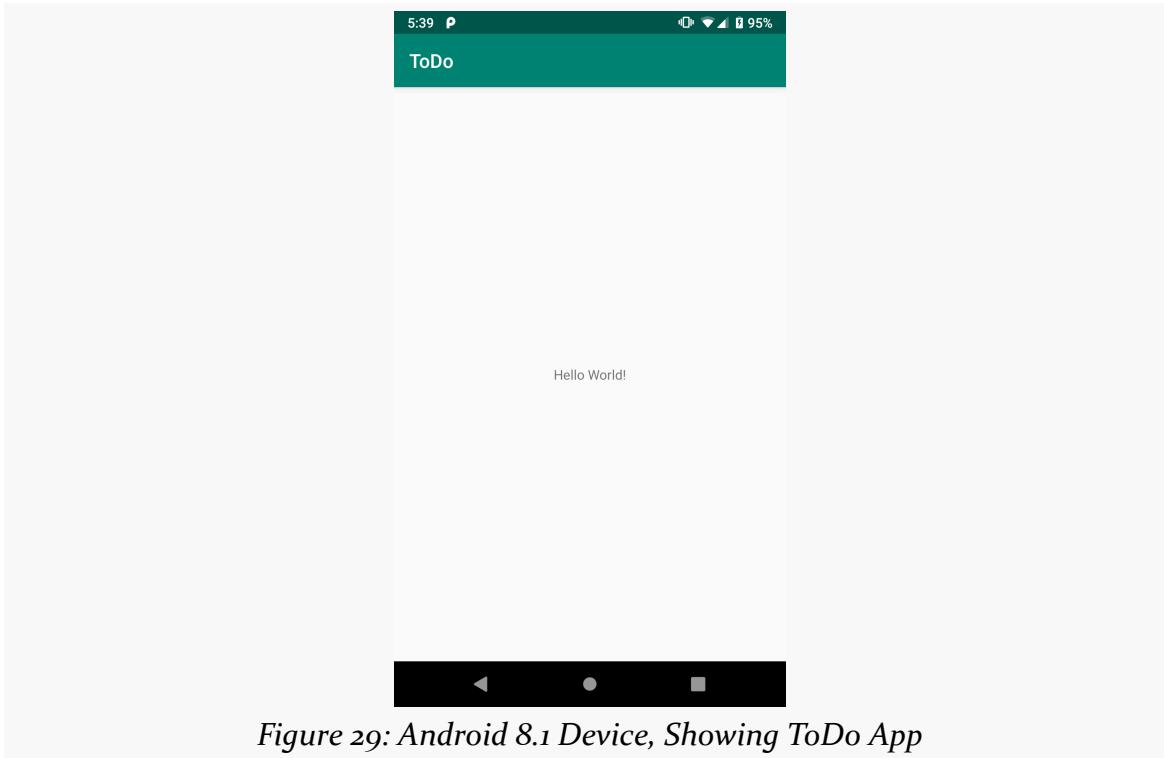


Figure 29: Android 8.1 Device, Showing ToDo App

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

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

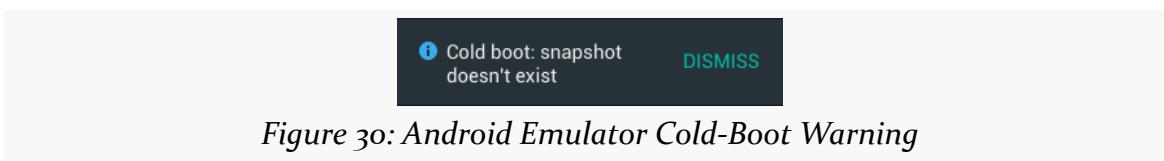


Figure 30: Android Emulator Cold-Boot Warning

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

## **CREATING A STARTER PROJECT**

---

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

# Modifying the Manifest

---

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

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

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

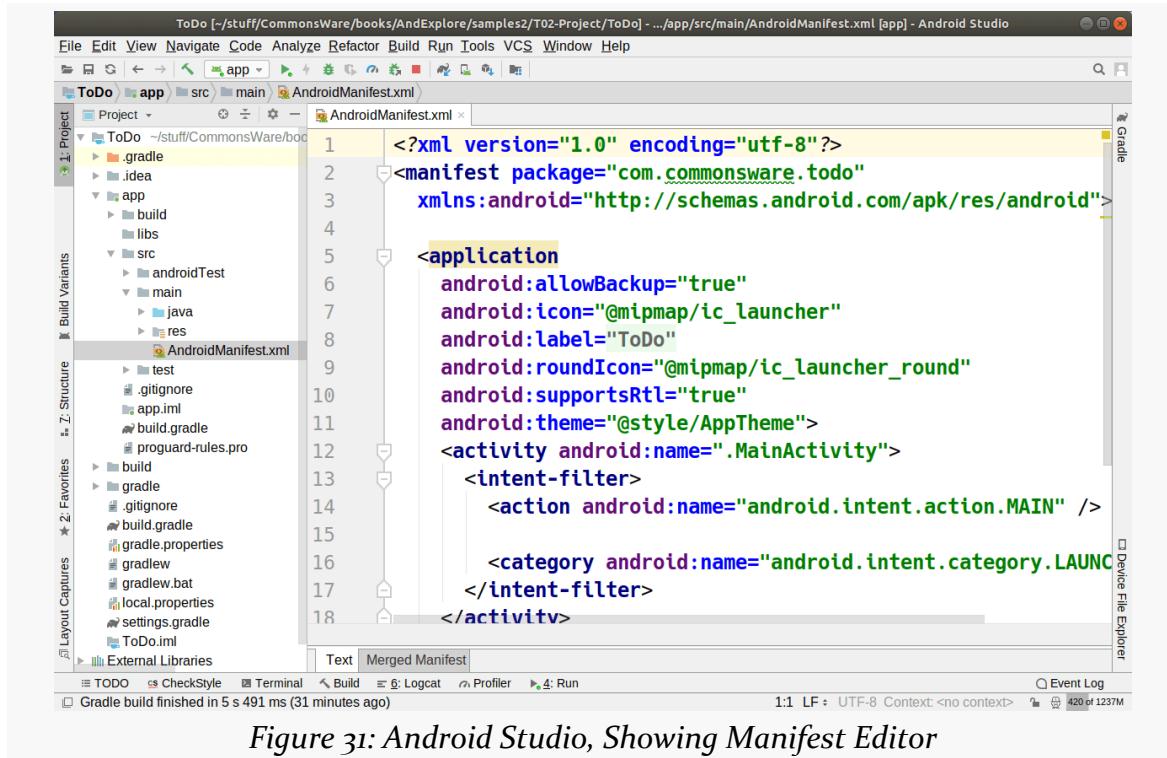


Figure 31: Android Studio, Showing Manifest Editor

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

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

At this point, the manifest should resemble:

## 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/AppTheme">
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

## Step #2: Blocking Backups

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

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

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

At this point, your manifest should look like:

```
<?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"
```

## MODIFYING THE MANIFEST

---

```
        android:normalScreens="true"
        android:smallScreens="true"
        android:xlargeScreens="true"/>

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

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

</manifest>
```

(from [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 home screen launcher.

## CHANGING OUR ICON

---

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



Figure 32: 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 adept at creating launcher icons. This is important, as while creating launcher icons used to be fairly simple, Android 8.0 made launcher icons a *lot* more complicated... but the Image Asset Wizard hides most of that complexity.

## CHANGING OUR ICON

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

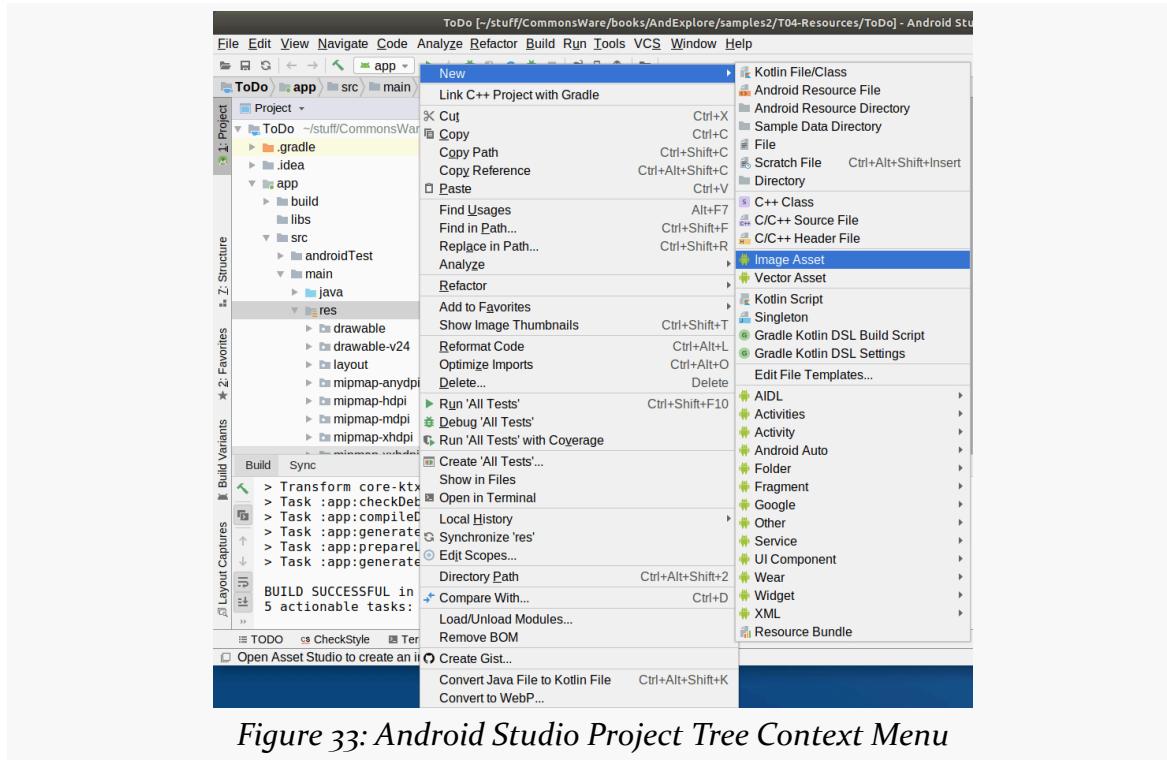


Figure 33: Android Studio Project Tree Context Menu

## CHANGING OUR ICON

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

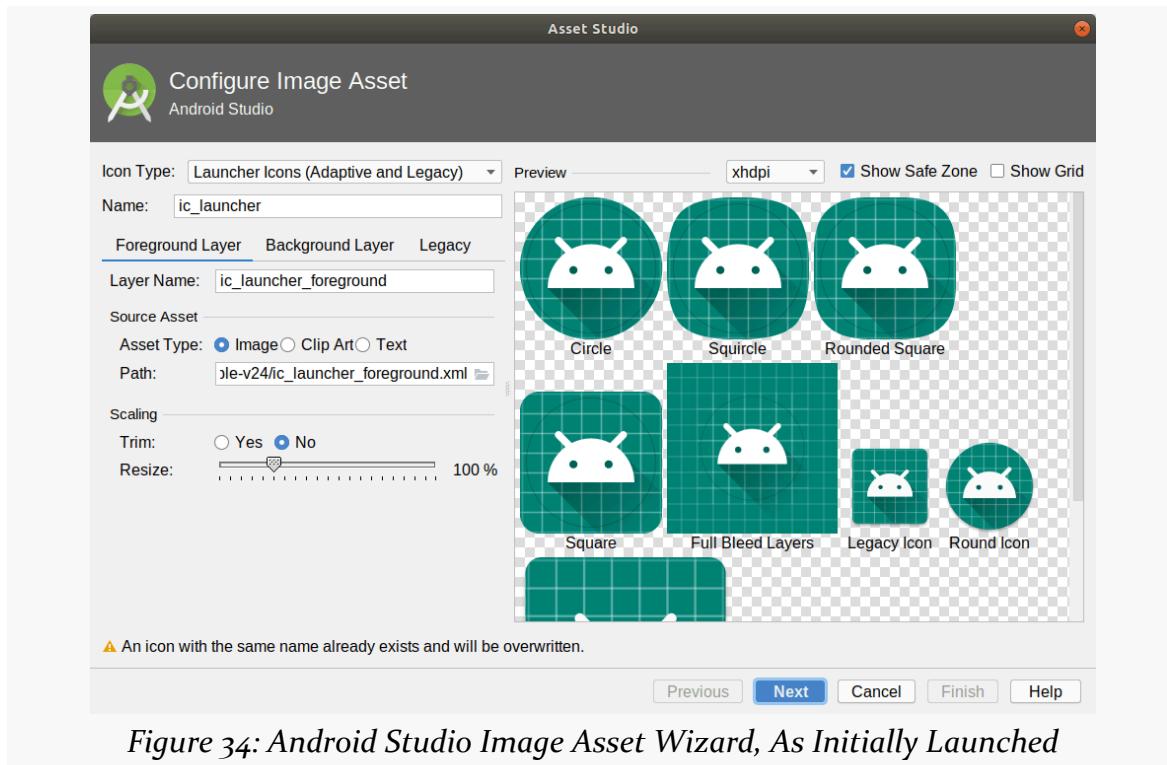


Figure 34: Android Studio Image Asset Wizard, As Initially Launched

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

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

## CHANGING OUR ICON

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

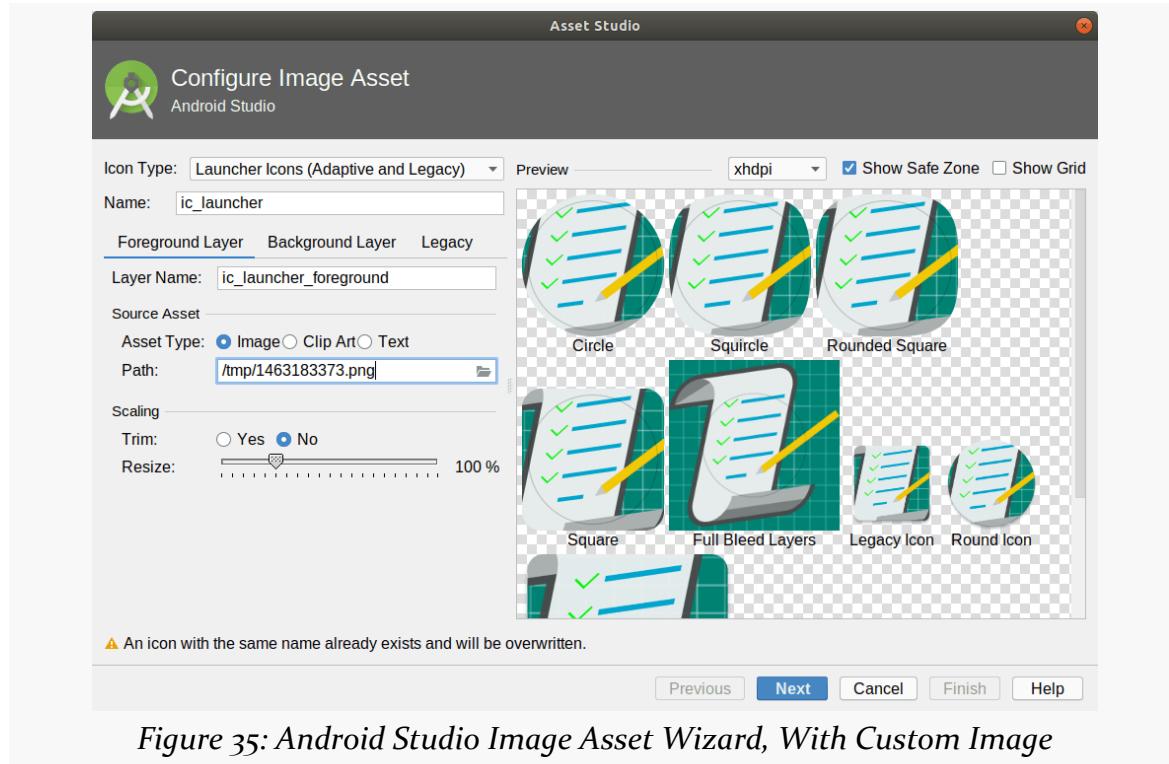


Figure 35: Android Studio Image Asset Wizard, With Custom Image

## 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:

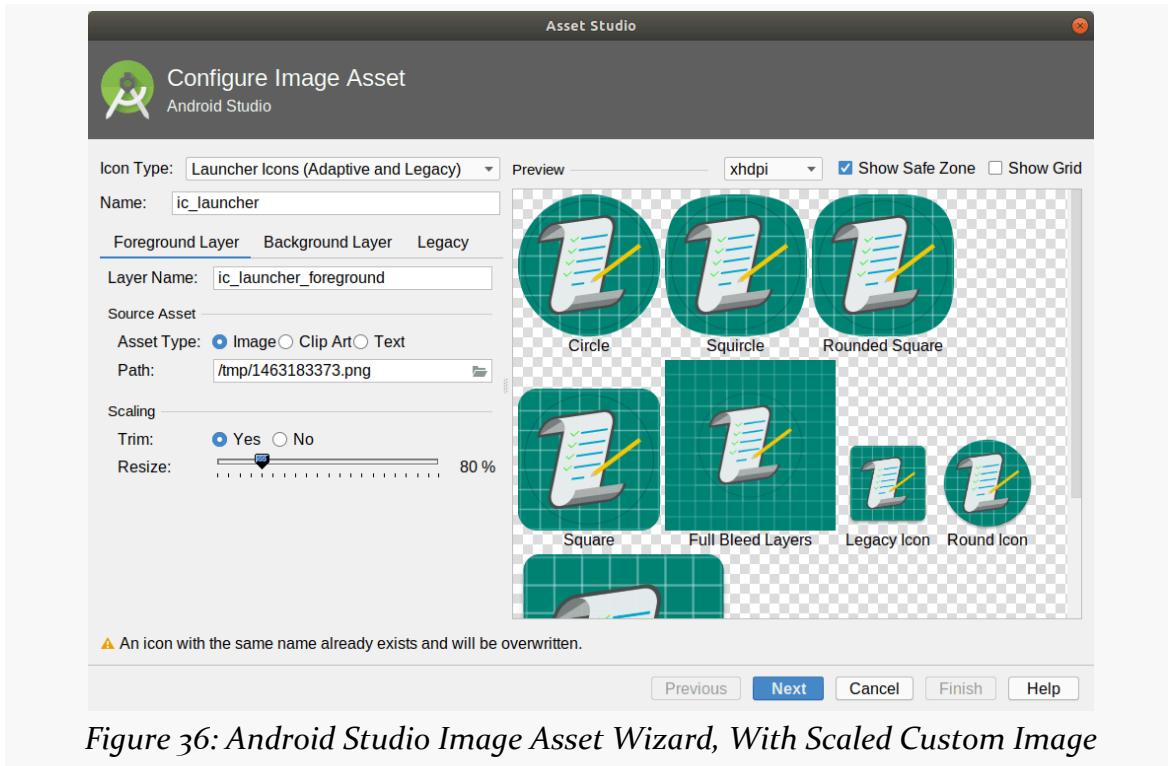


Figure 36: Android Studio Image Asset Wizard, With Scaled Custom Image

## 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”:

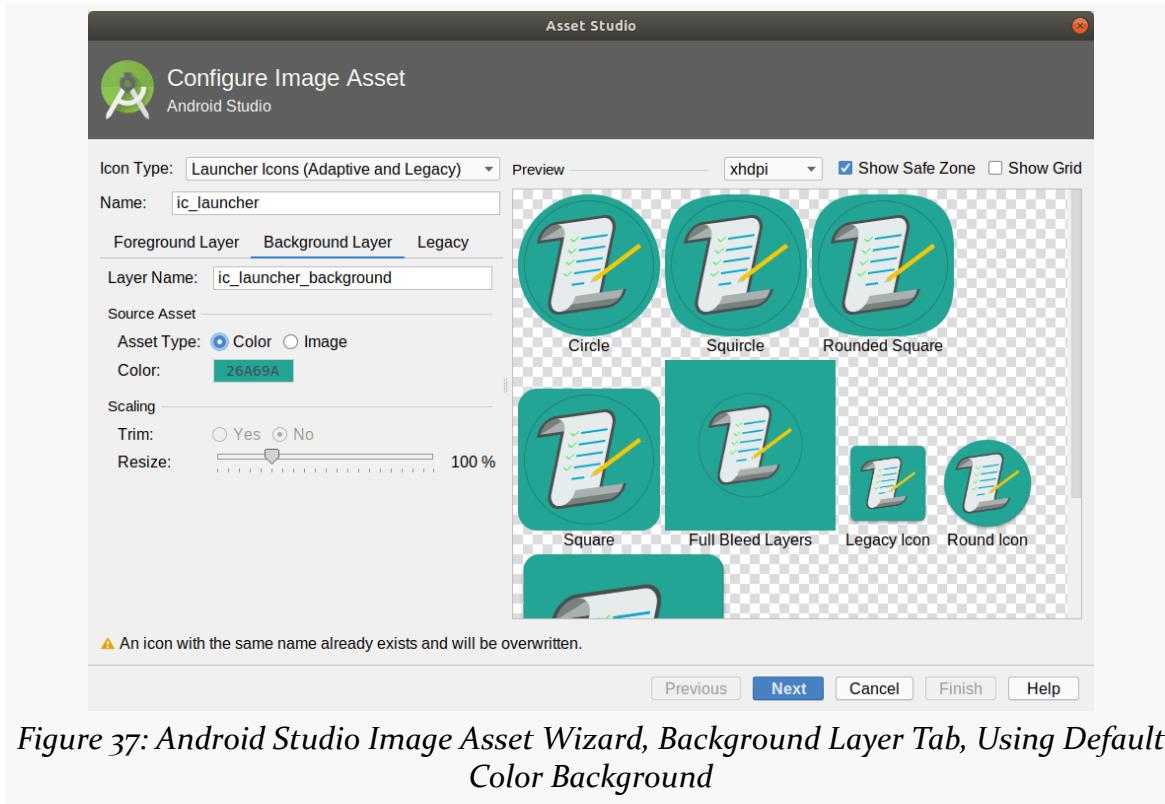


Figure 37: Android Studio Image Asset Wizard, Background Layer Tab, Using Default Color Background

## CHANGING OUR ICON

---

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

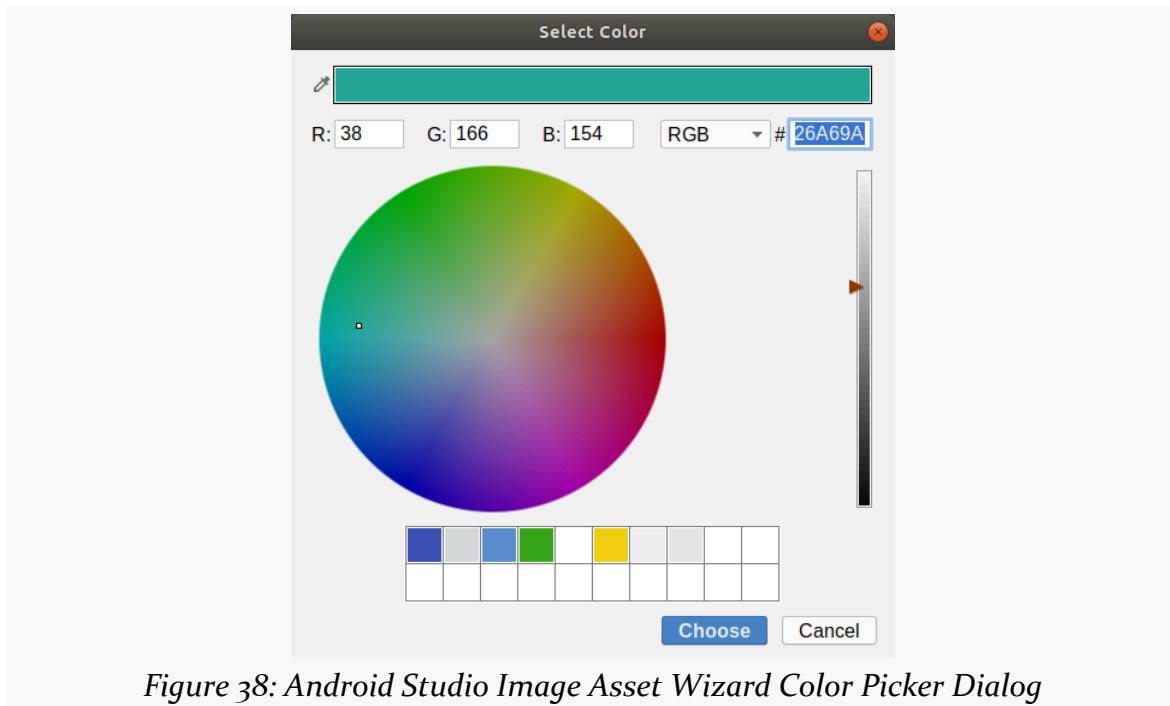


Figure 38: Android Studio Image Asset Wizard Color Picker Dialog

## CHANGING OUR ICON

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

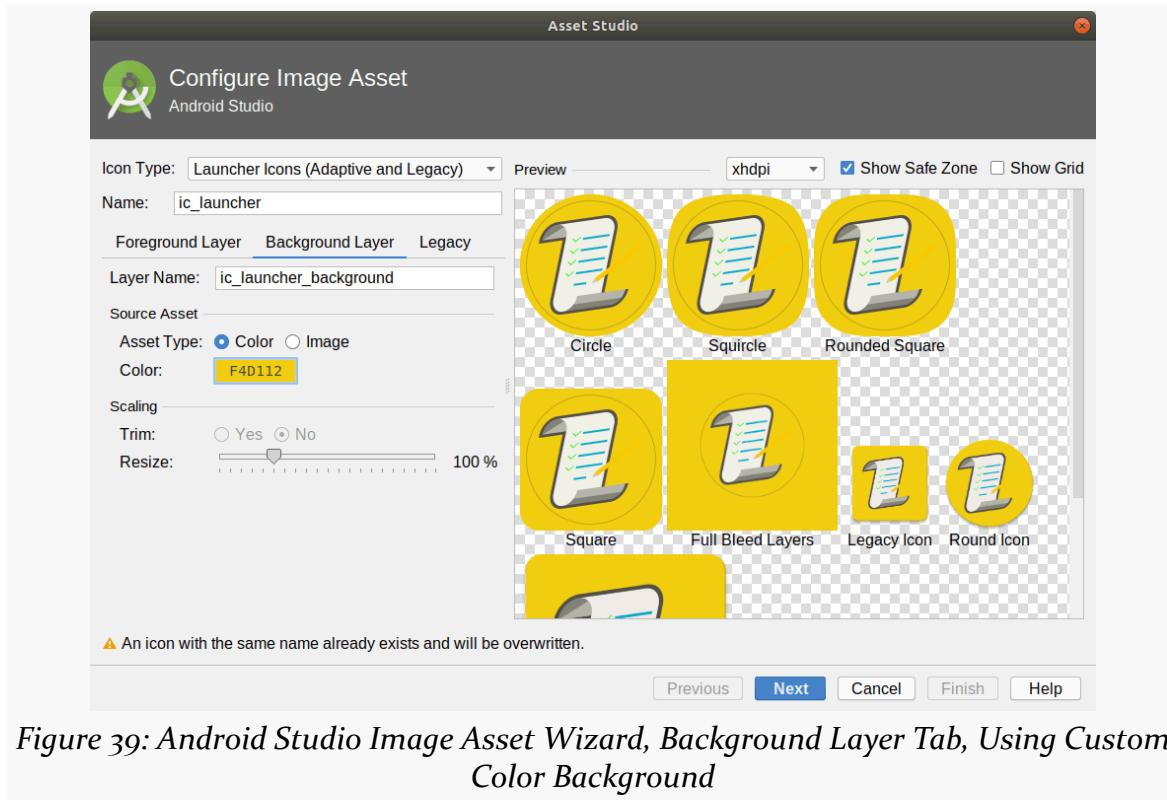


Figure 39: Android Studio Image Asset Wizard, Background Layer Tab, Using Custom Color Background

## CHANGING OUR ICON

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

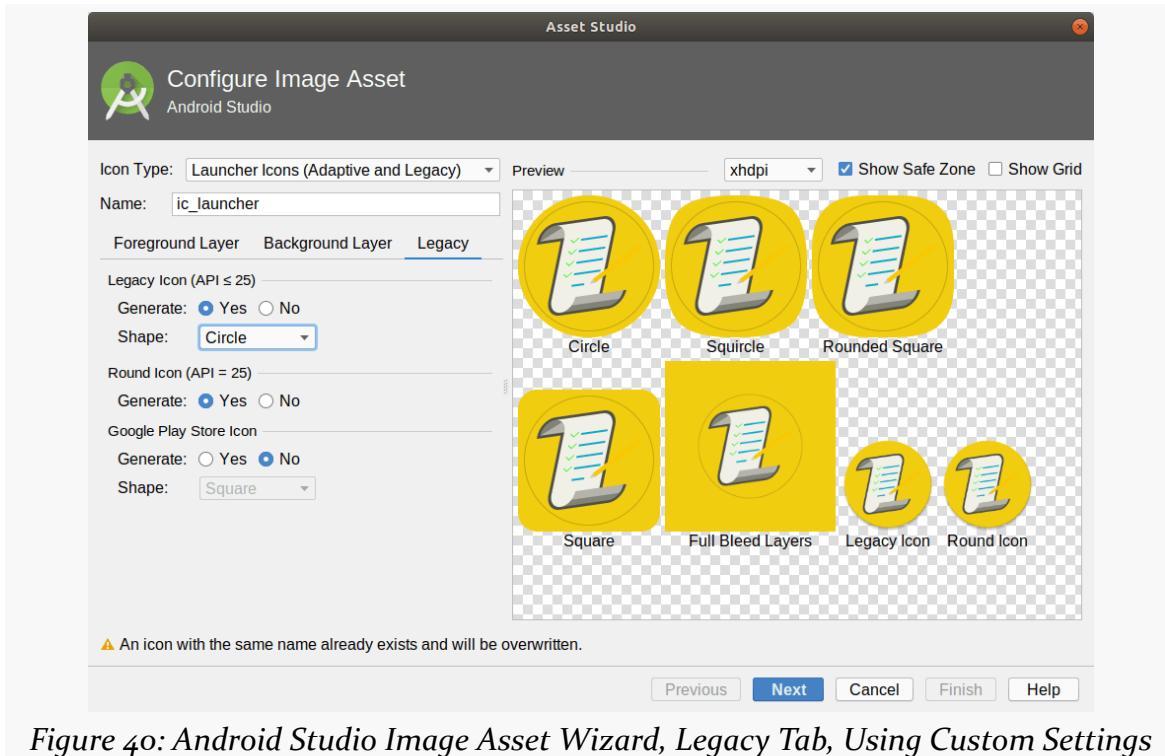


Figure 40: Android Studio Image Asset Wizard, Legacy Tab, Using Custom Settings

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

## CHANGING OUR ICON

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

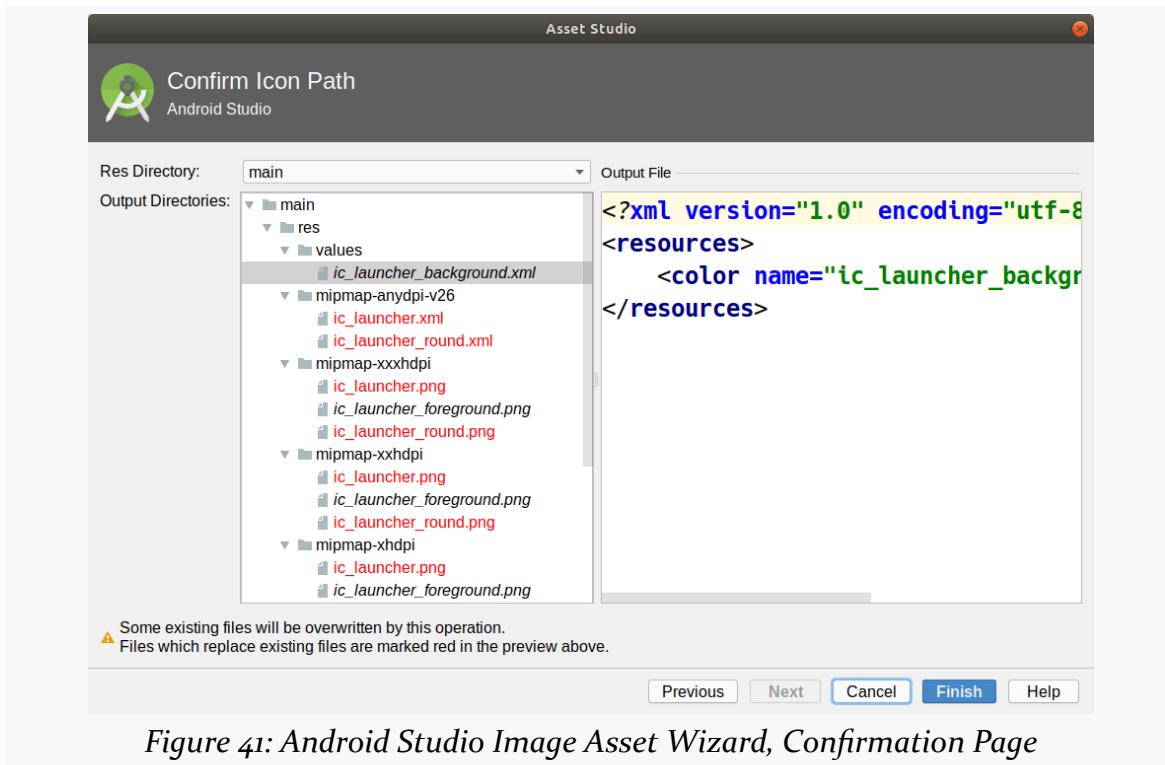


Figure 41: Android Studio Image Asset Wizard, Confirmation Page

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

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

### Step #3: Running the Result

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

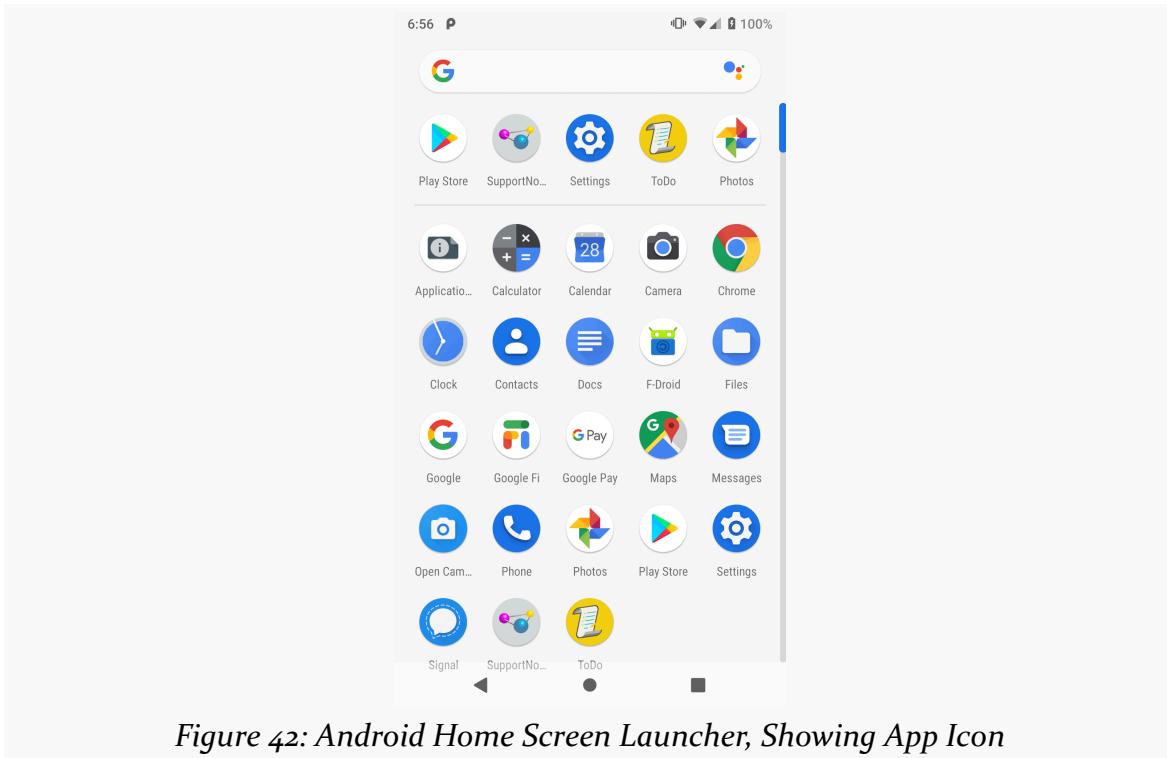


Figure 42: Android Home Screen Launcher, Showing App Icon

### 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: Removing Unnecessary Cruft

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

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

## ADDING A LIBRARY

---

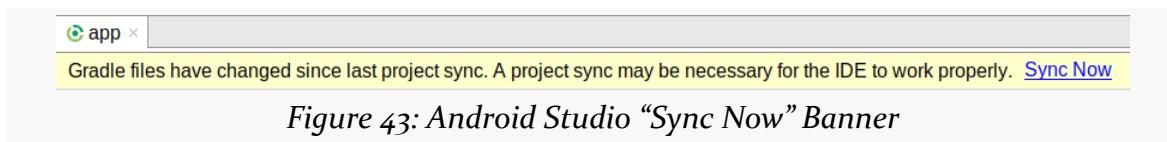
A new Android Studio project will contain this sort of initial set of dependencies, though the details will vary a bit depending on Android Studio version and the particular choices you make when creating the project. The `implementation`, `testImplementation`, and `androidTestImplementation` lines indicate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

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

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

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

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



*Figure 43: Android Studio “Sync Now” Banner*

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

## Step #2: Adding Support for RecyclerView

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

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

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

## ADDING A LIBRARY

---

dependencies in the app.

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

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

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

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

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

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

android {
    compileSdkVersion 28

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

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

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

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

## What We Changed

The book’s GitLab repository contains [the entire result of having completed this](#)

## ADDING A LIBRARY

---

[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`. If you open that up in the IDE and switch to the “Text” sub-tab, you will see XML like this:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
```

## CONSTRUCTING A LAYOUT

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Hello World!"  
    app:layout_constraintBottom_toBottomOf="parent"  
    app:layout_constraintLeft_toLeftOf="parent"  
    app:layout_constraintRight_toRightOf="parent"  
    app:layout_constraintTop_toTopOf="parent" />  
  
</androidx.constraintlayout.widget.ConstraintLayout>
```

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

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

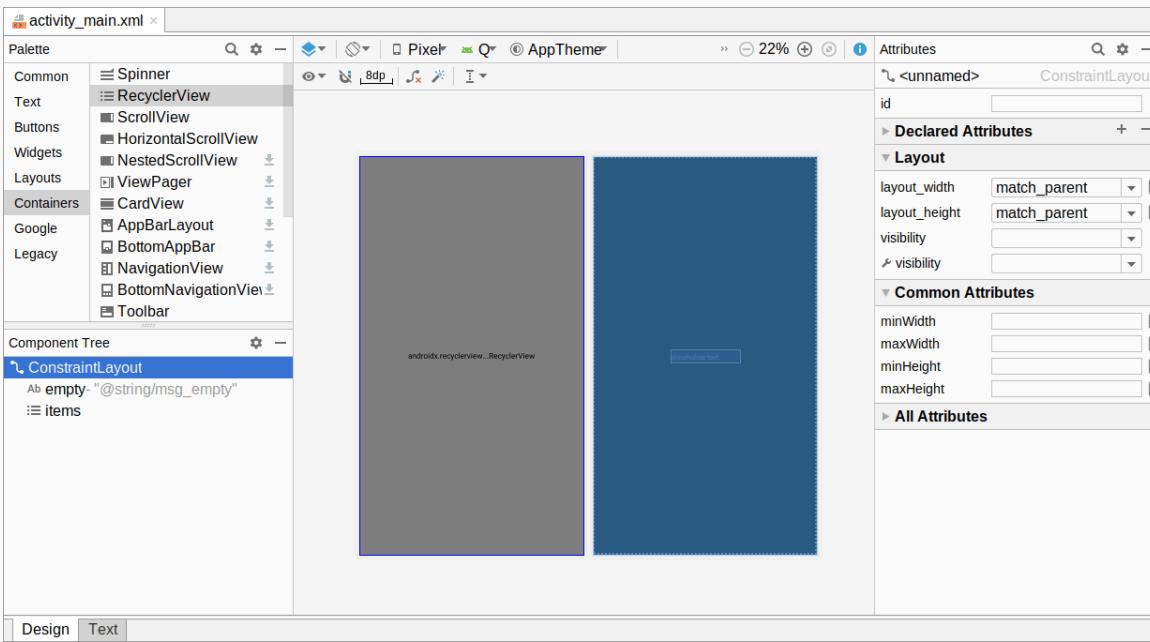


Figure 44: Android Studio Layout Designer, Showing End Result of This Tutorial

## CONSTRUCTING A LAYOUT

---

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

Switch to the “Design” sub-tab in the layout resource editor to get back to the GUI builder. Then, in the “Palette” area, switch to “Containers” category:

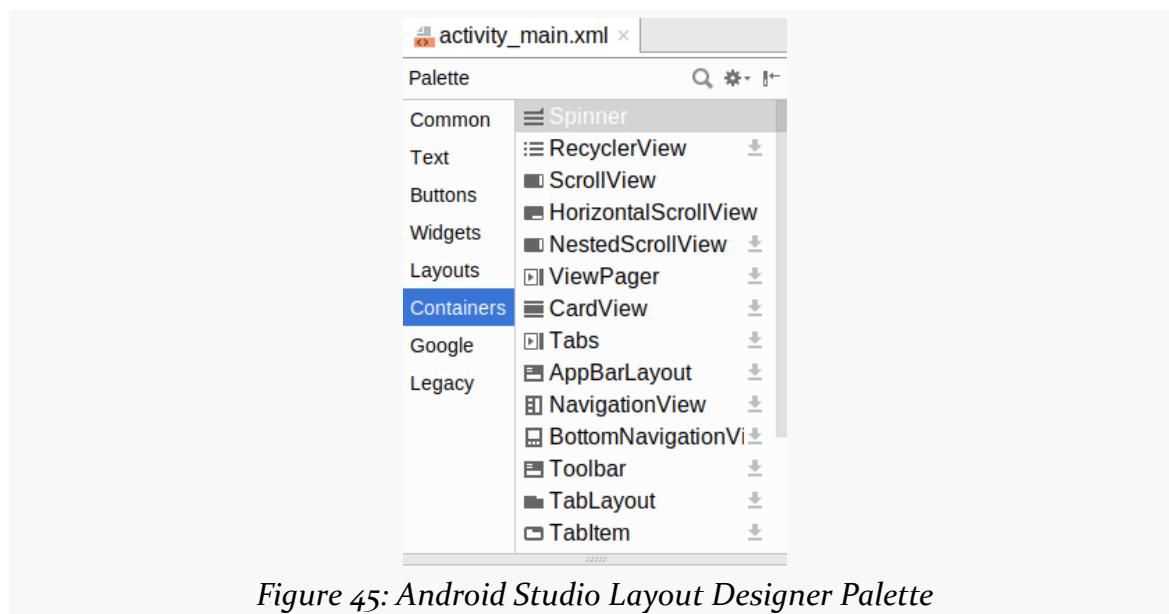
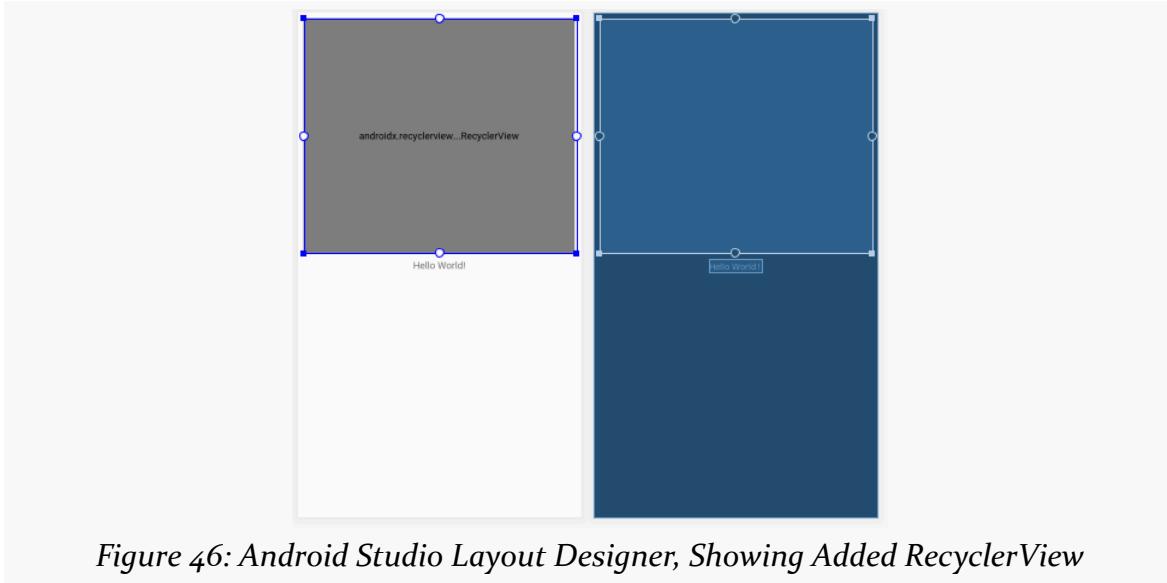


Figure 45: Android Studio Layout Designer Palette

## CONSTRUCTING A LAYOUT

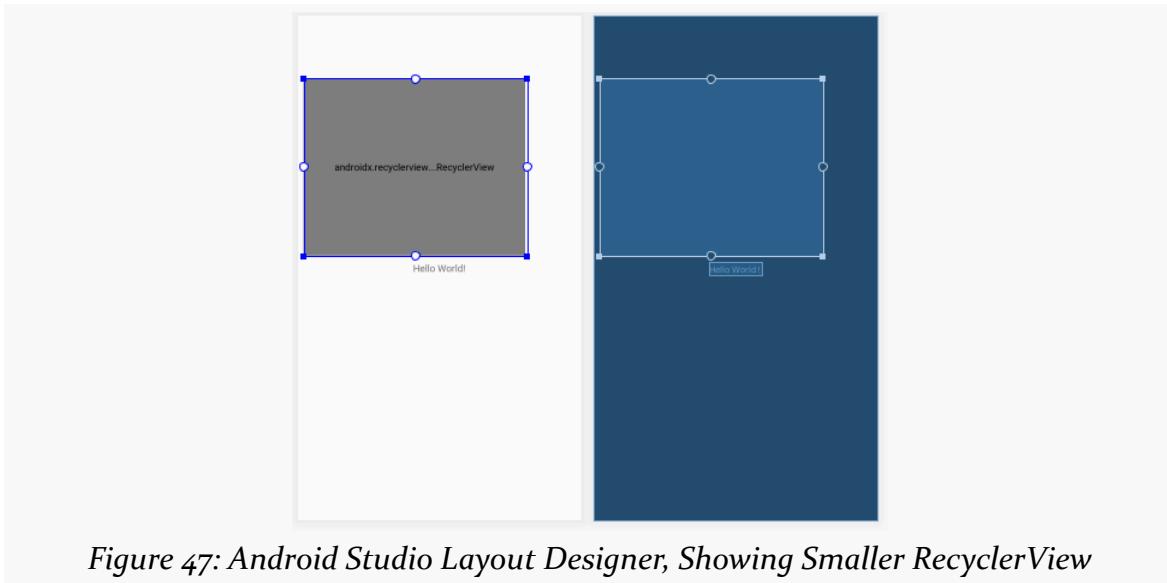
---

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



*Figure 46: Android Studio Layout Designer, Showing Added RecyclerView*

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



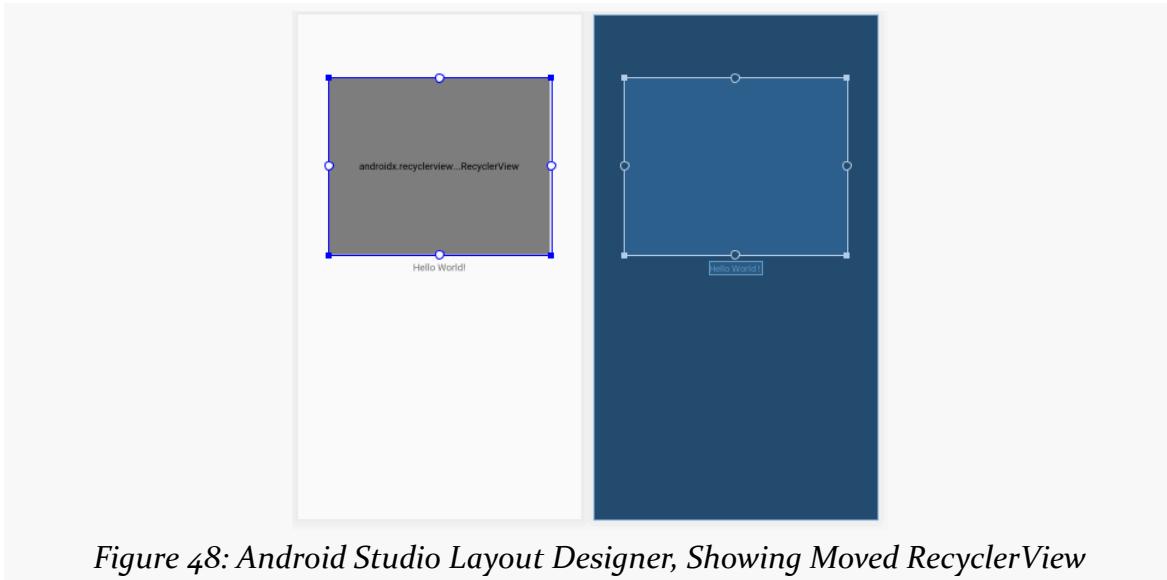
*Figure 47: Android Studio Layout Designer, Showing Smaller RecyclerView*

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

## CONSTRUCTING A LAYOUT

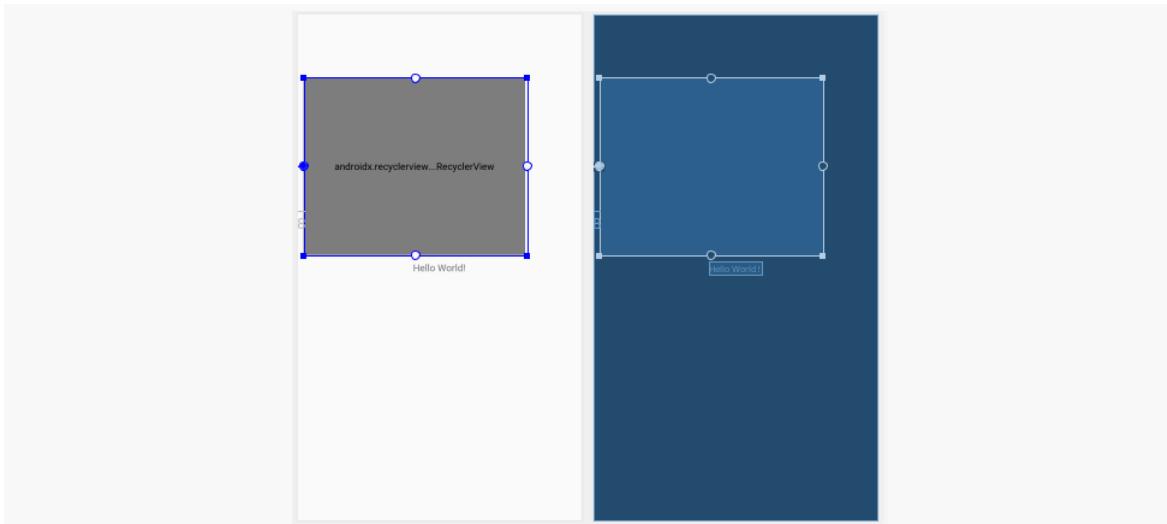
---

maneuver:



*Figure 48: Android Studio Layout Designer, Showing Moved RecyclerView*

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



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

## CONSTRUCTING A LAYOUT

---

Repeat that process on the right side:

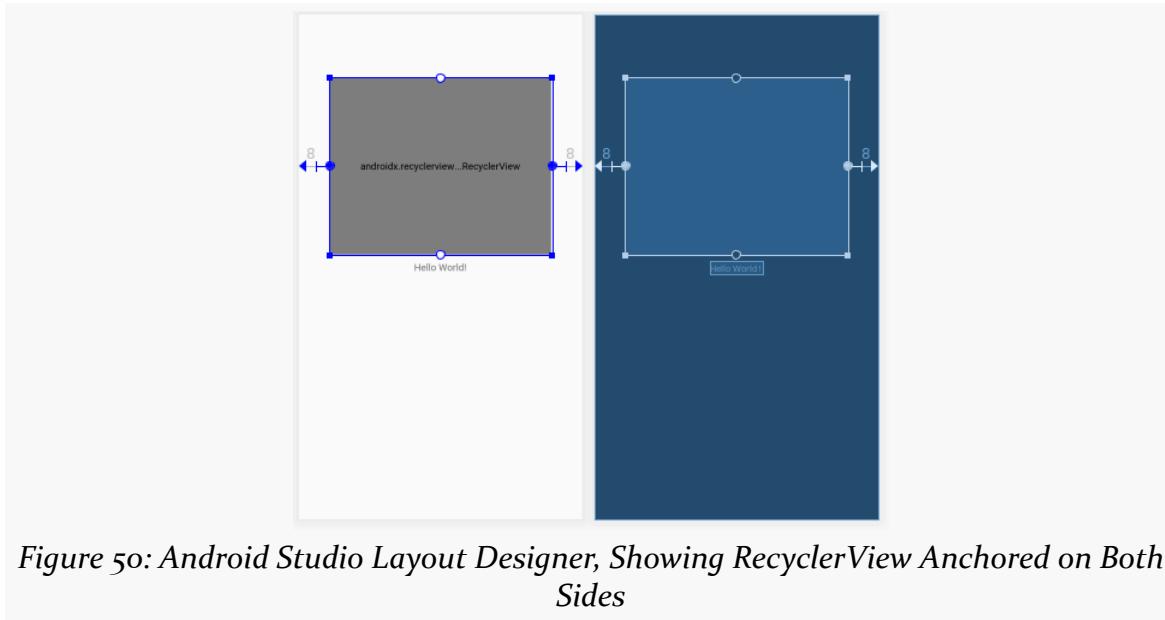


Figure 50: Android Studio Layout Designer, Showing RecyclerView Anchored on Both Sides

Repeat that process on the top side:

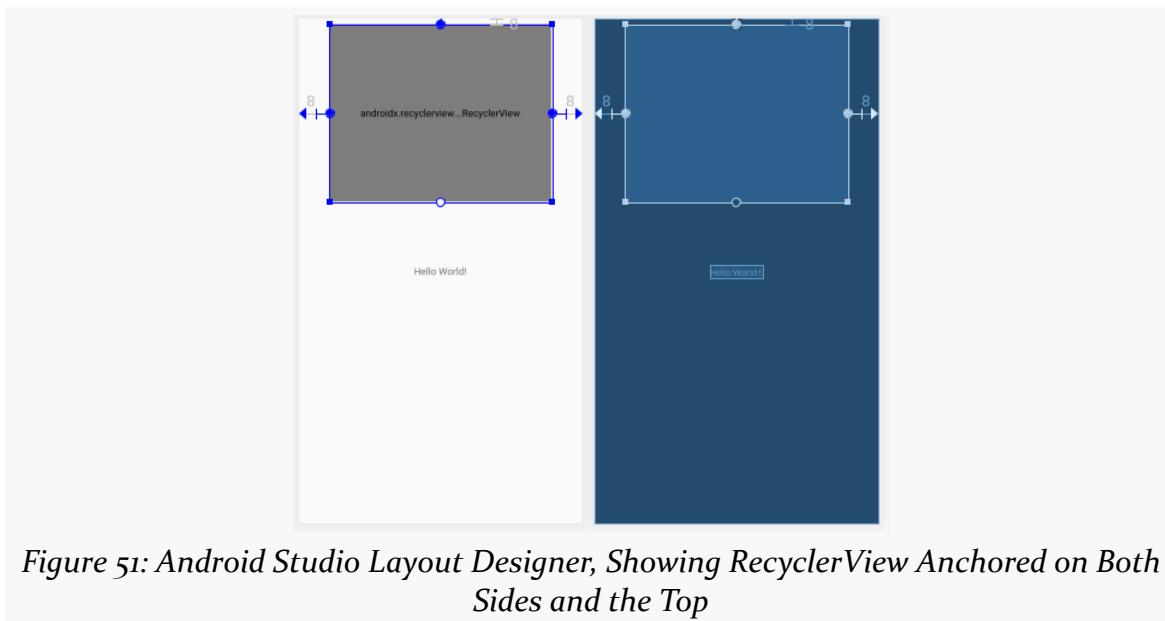


Figure 51: Android Studio Layout Designer, Showing RecyclerView Anchored on Both Sides and the Top

## CONSTRUCTING A LAYOUT

---

Repeat that process on the bottom side:

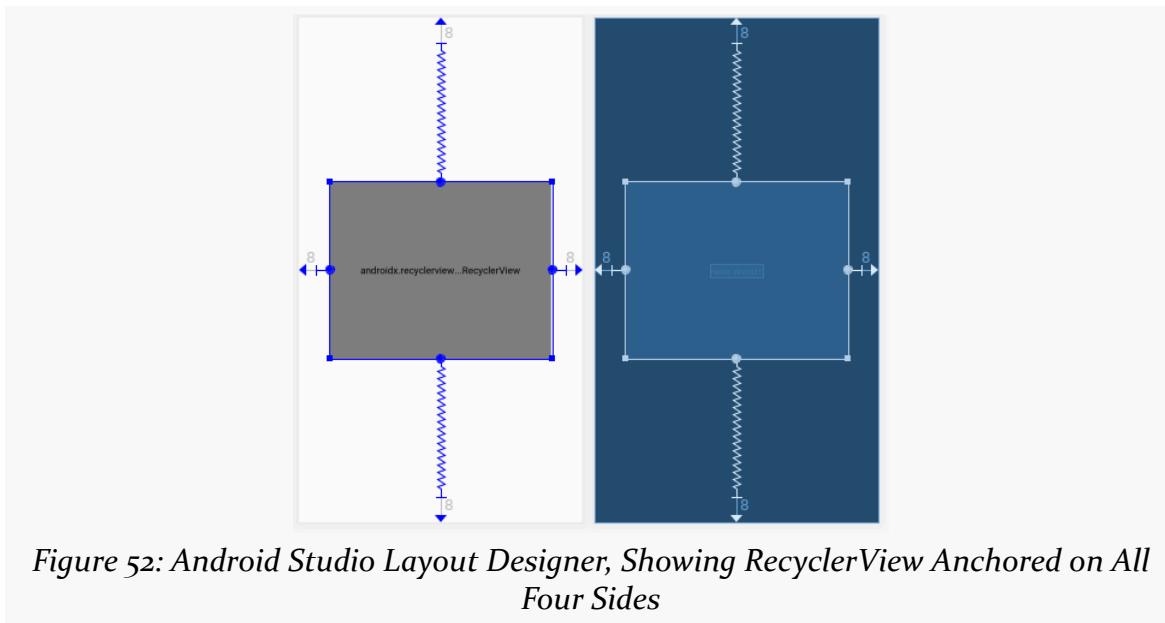


Figure 52: Android Studio Layout Designer, Showing RecyclerView Anchored on All Four Sides

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

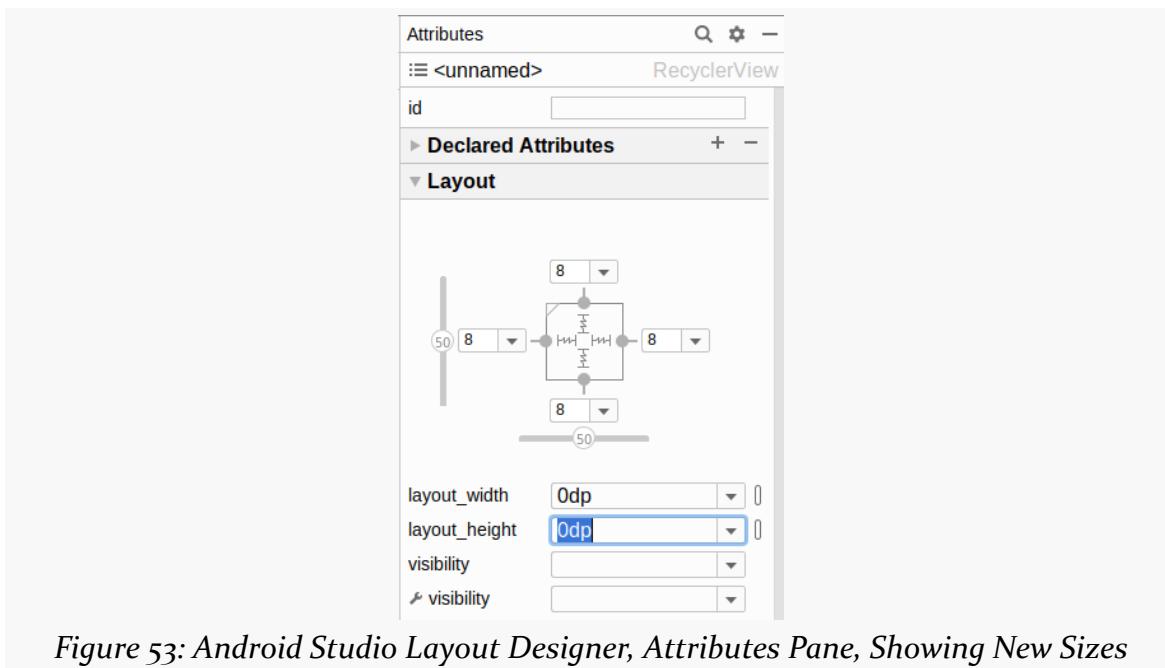
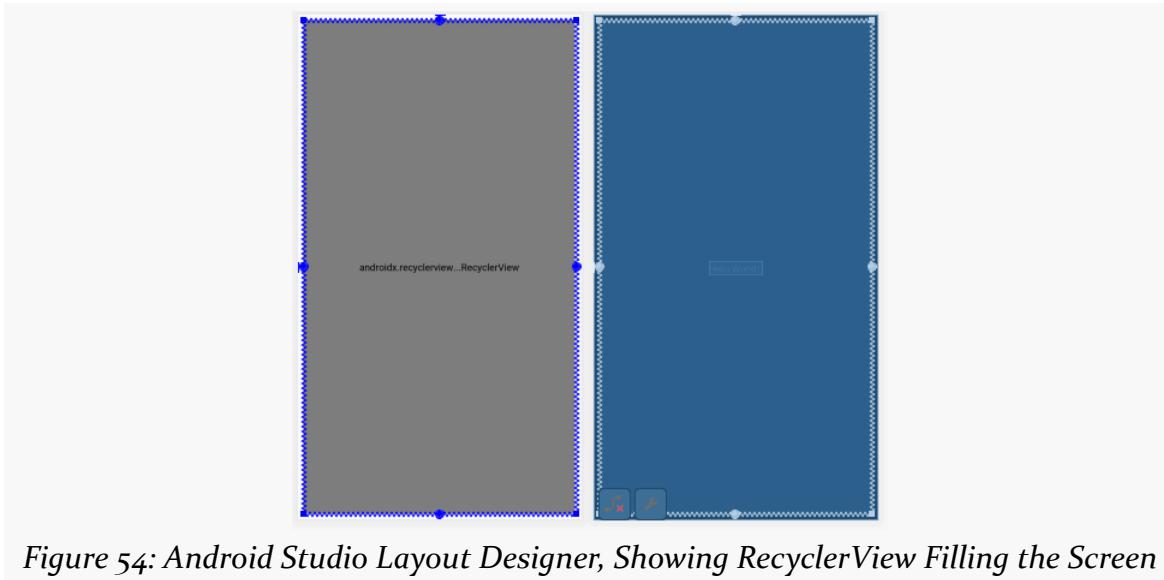


Figure 53: Android Studio Layout Designer, Attributes Pane, Showing New Sizes

## CONSTRUCTING A LAYOUT

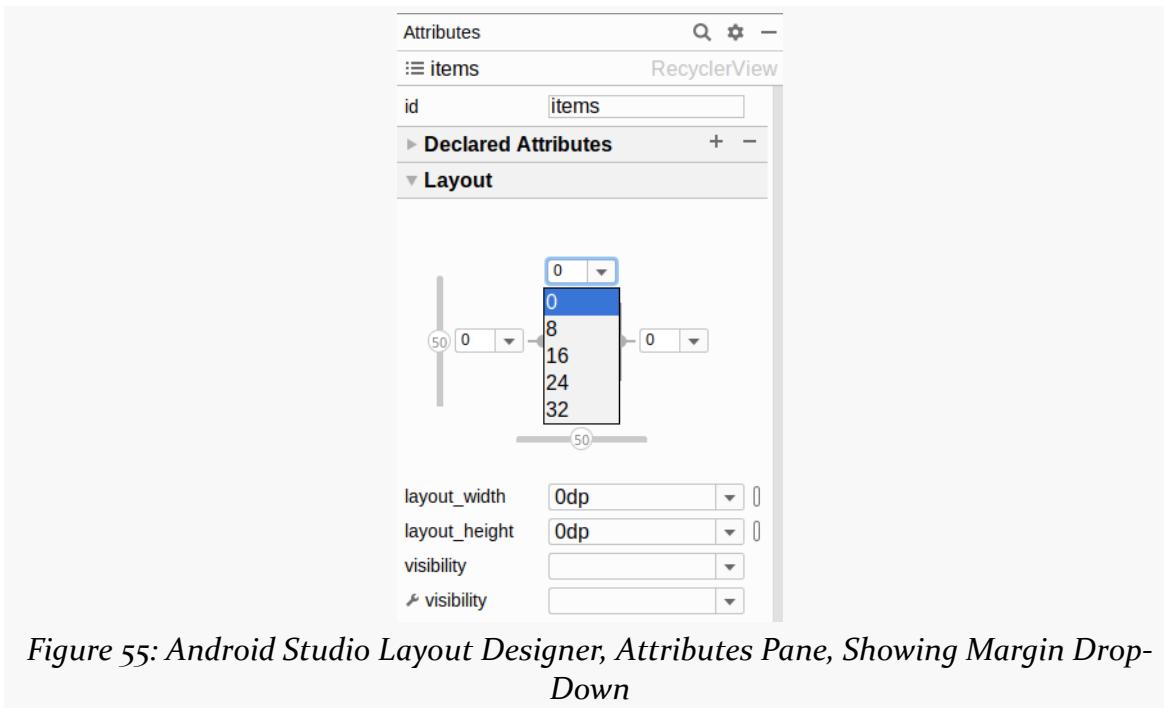
---

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



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

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



## CONSTRUCTING A LAYOUT

---

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

### Step #3: Adjusting the TextView

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



Figure 56: Android Studio Layout Designer, Component Tree Pane

## 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:

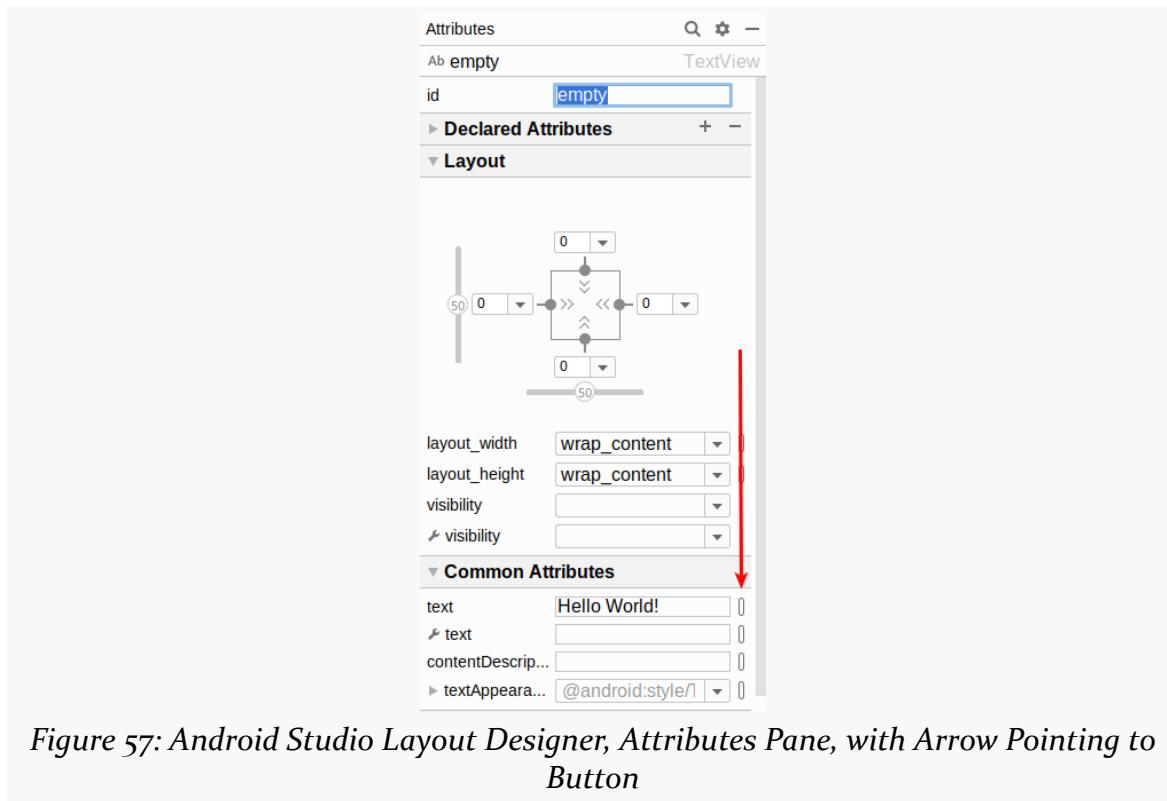


Figure 57: Android Studio Layout Designer, Attributes Pane, with Arrow Pointing to Button

## CONSTRUCTING A LAYOUT

---

This will bring up a dialog showing available string resources:

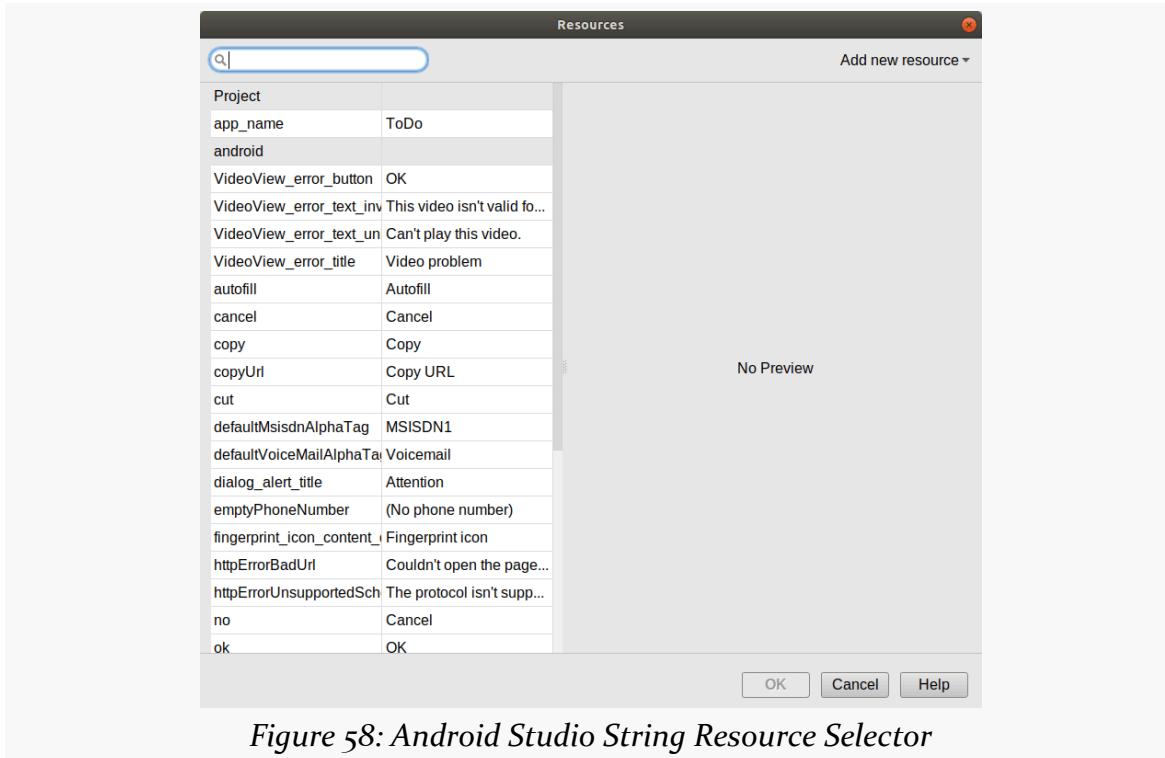


Figure 58: Android Studio String Resource Selector

## CONSTRUCTING A LAYOUT

---

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

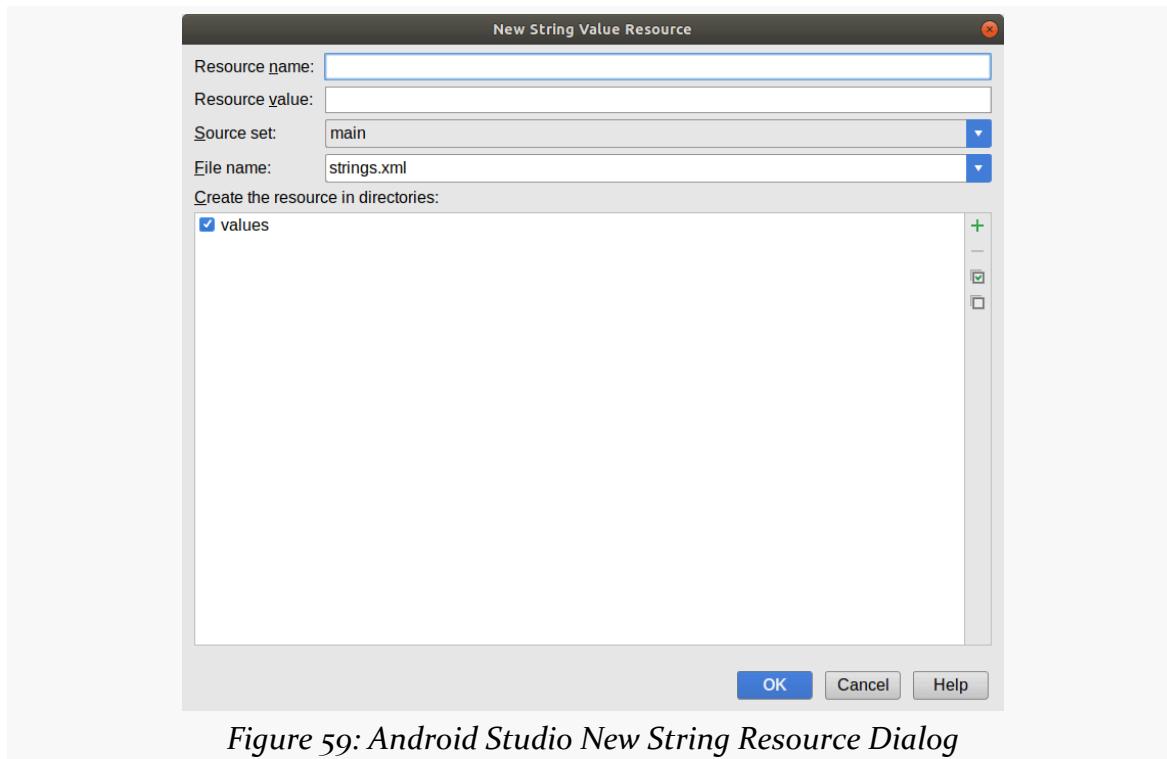


Figure 59: Android Studio New String Resource Dialog

## CONSTRUCTING A LAYOUT

---

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

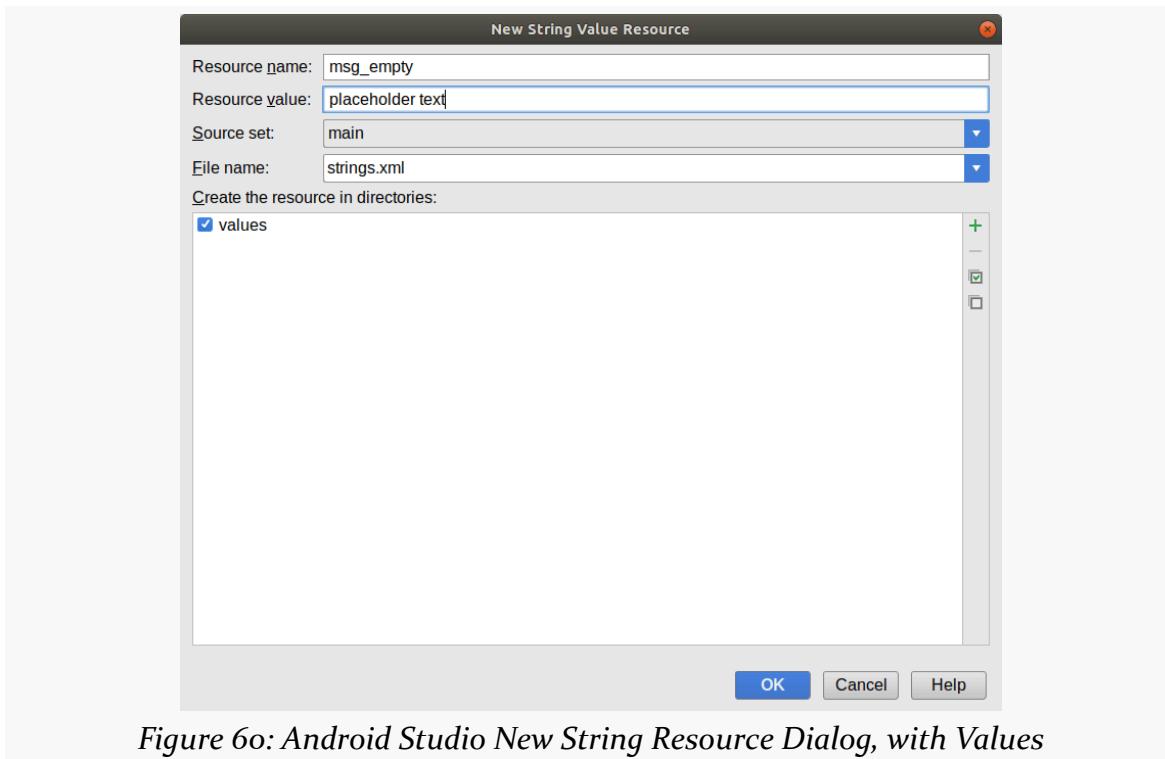


Figure 6o: Android Studio New String Resource Dialog, with Values

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

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

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

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

## CONSTRUCTING A LAYOUT

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

(from [To6-Layout/ToDo/app/src/main/res/layout/activity\\_main.xml](#))

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

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

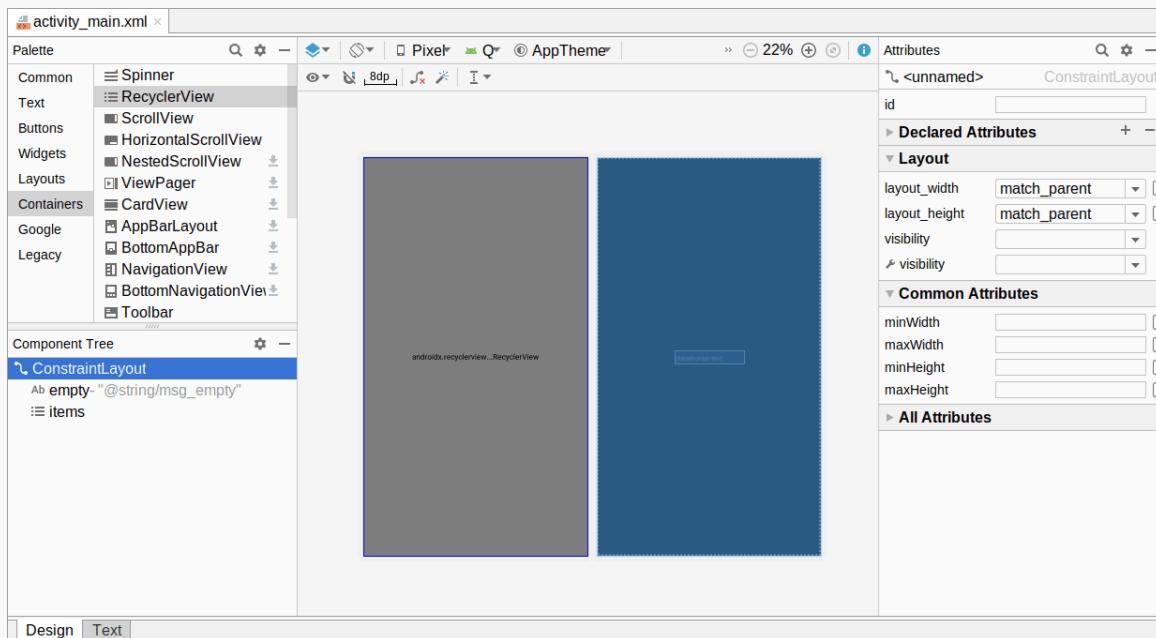


Figure 61: Android Studio Layout Designer, Showing End Result of This Tutorial

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

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

## CONSTRUCTING A LAYOUT

```
        android:text="@string/msg_empty"
        android:textAppearance="?android:attr/textAppearanceMedium"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

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

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

(from [To6-Layout/ToDo/app/src/main/res/layout/activity\\_main.xml](#))

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

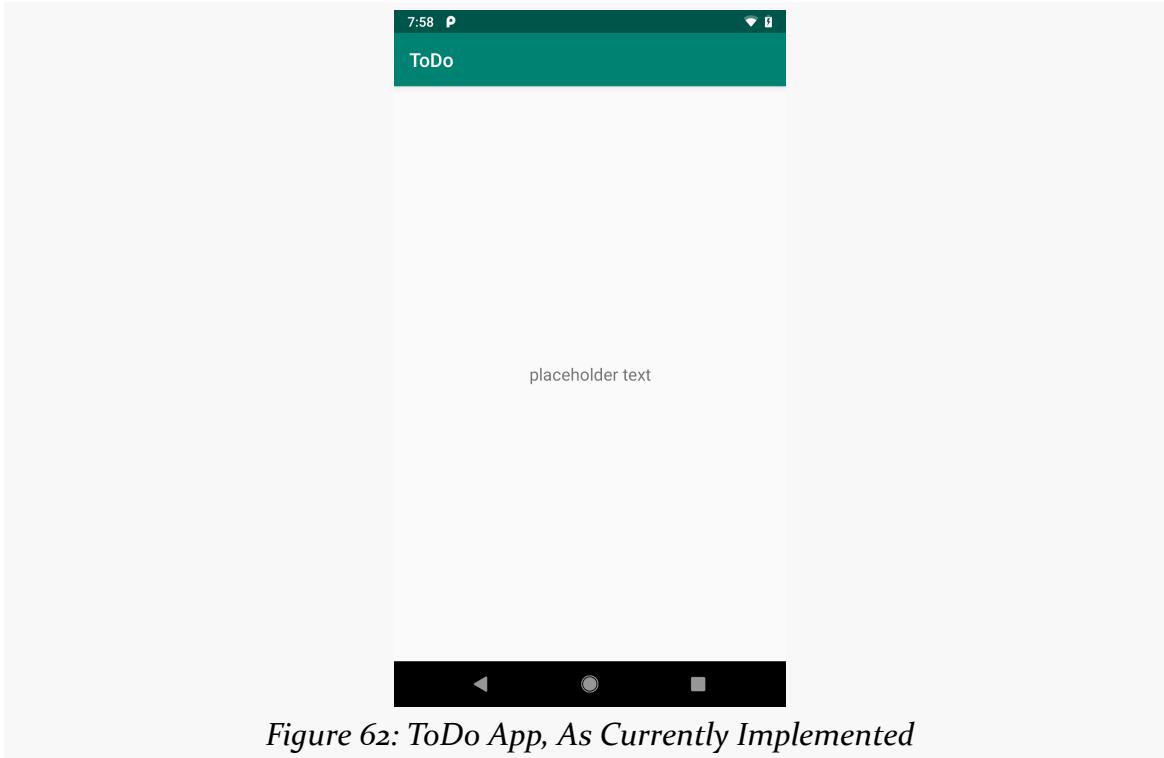


Figure 62: ToDo App, As Currently Implemented

## CONSTRUCTING A LAYOUT

---

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

## 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](#)

# **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](#).

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

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

## SETTING UP THE APP BAR

---

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

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



You can learn more about styles and themes in the "Defining and Using Styles" chapter of [\*Elements of Android Jetpack\*](#)!



You can learn more about Toolbar in the "Configuring the App Bar" chapter of [\*Elements of Android Jetpack\*](#)!

## Step #1: Defining Some Colors

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

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

But, we need to choose some colors.

## SETTING UP THE APP BAR

---

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

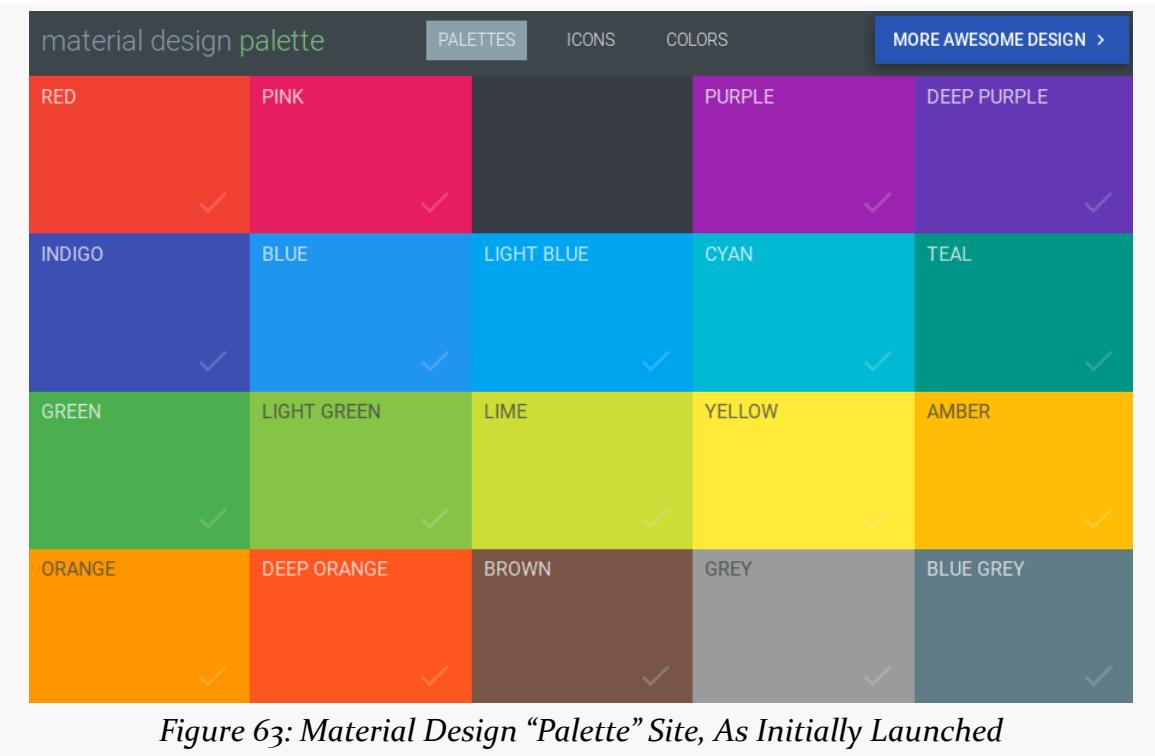


Figure 63: Material Design “Palette” Site, As Initially Launched

## SETTING UP THE APP BAR

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

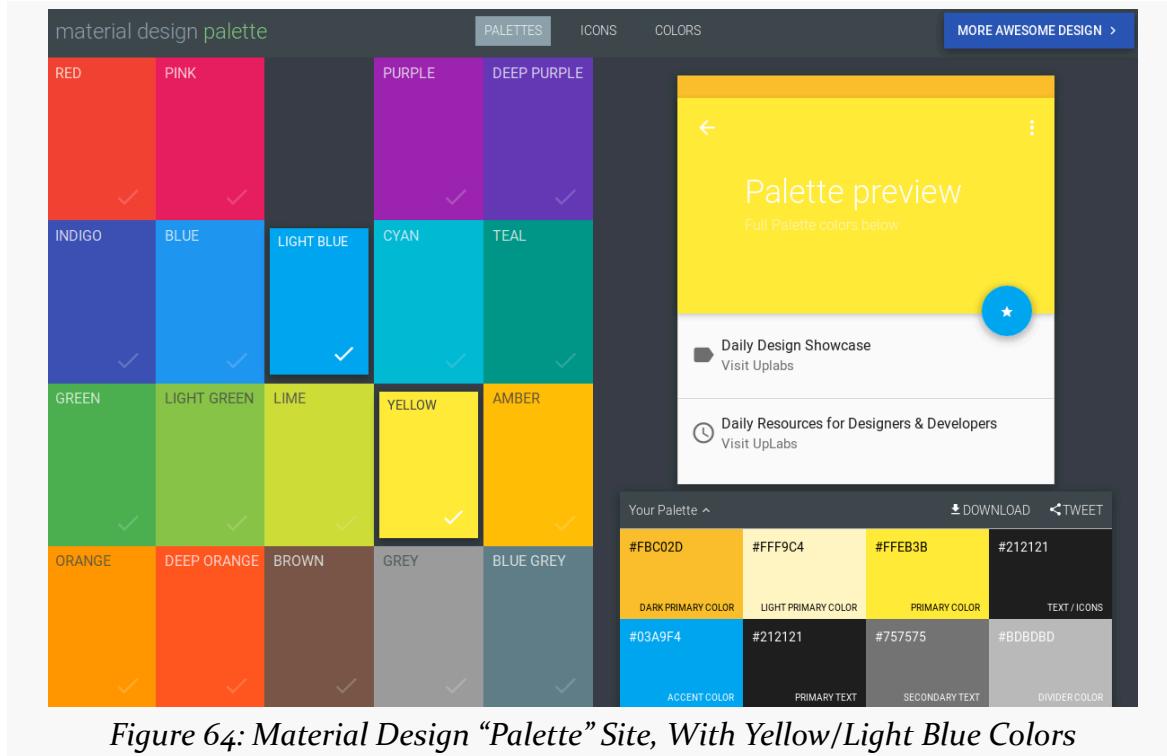


Figure 64: Material Design “Palette” Site, With Yellow/Light Blue Colors

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

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

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

## SETTING UP THE APP BAR

---

```
</resources>
```

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

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

(from [To7-Toolbar/ToDo/app/src/main/res/values/colors.xml](#))

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

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

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

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

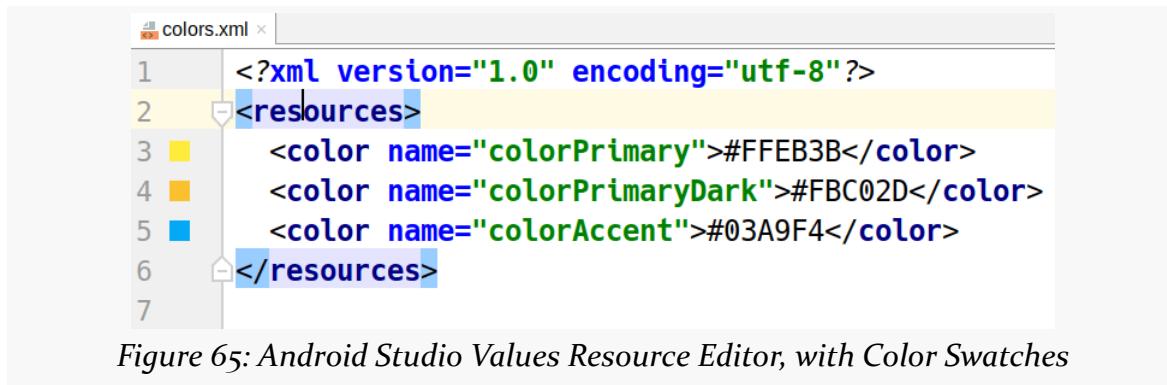


Figure 65: Android Studio Values Resource Editor, with Color Swatches

## 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:

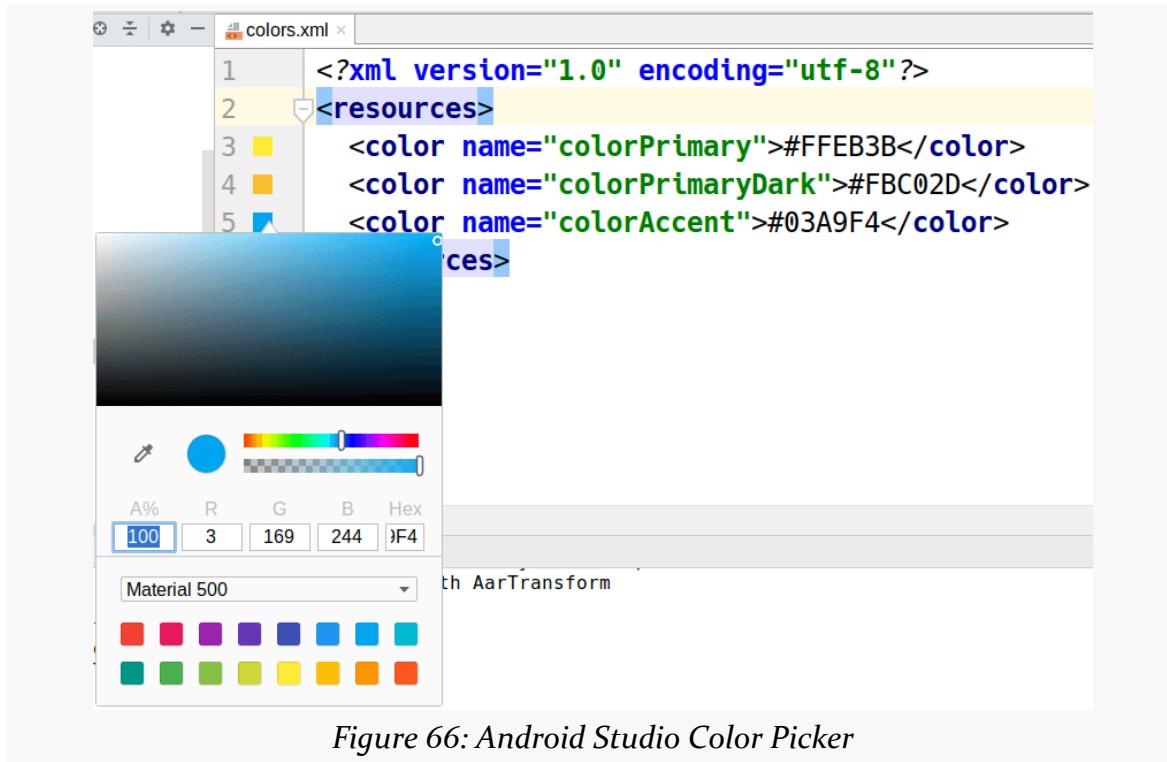


Figure 66: Android Studio Color Picker

## Step #2: Adjusting Our Theme

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

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

```
<resources>

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

## SETTING UP THE APP BAR

---

```
</resources>
```

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

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

There are two problems with our default theme:

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

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

The resulting resource should look like:

```
<resources>

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

</resources>
```

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

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

## SETTING UP THE APP BAR

---

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

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

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

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

</manifest>
```

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

## Step #3: Adding a Toolbar

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

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

## SETTING UP THE APP BAR

---

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

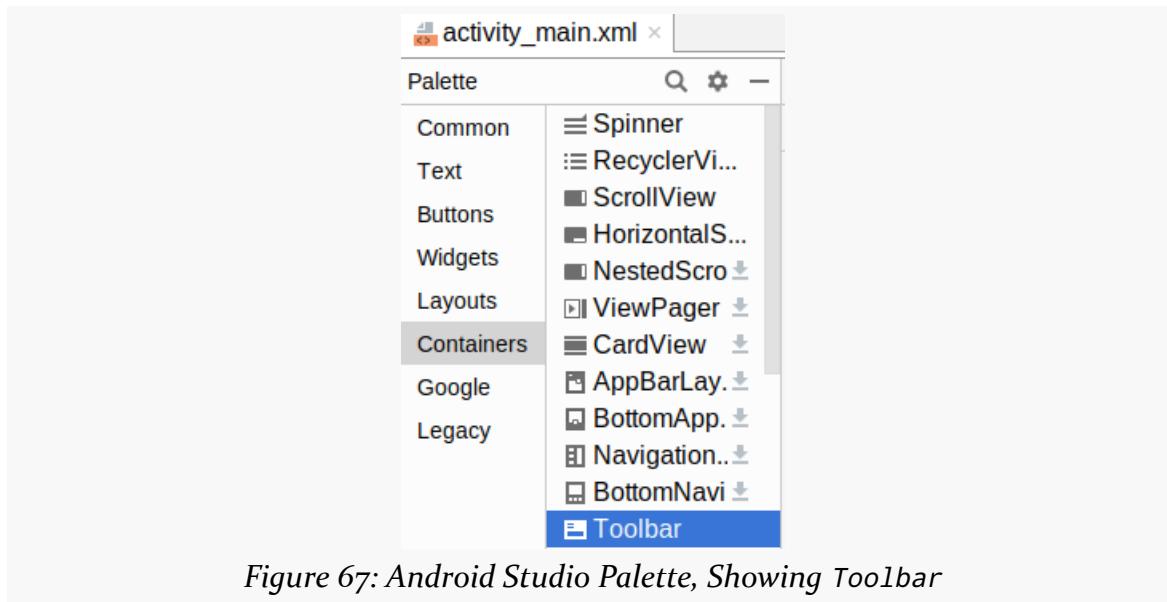


Figure 67: Android Studio Palette, Showing Toolbar

## 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:

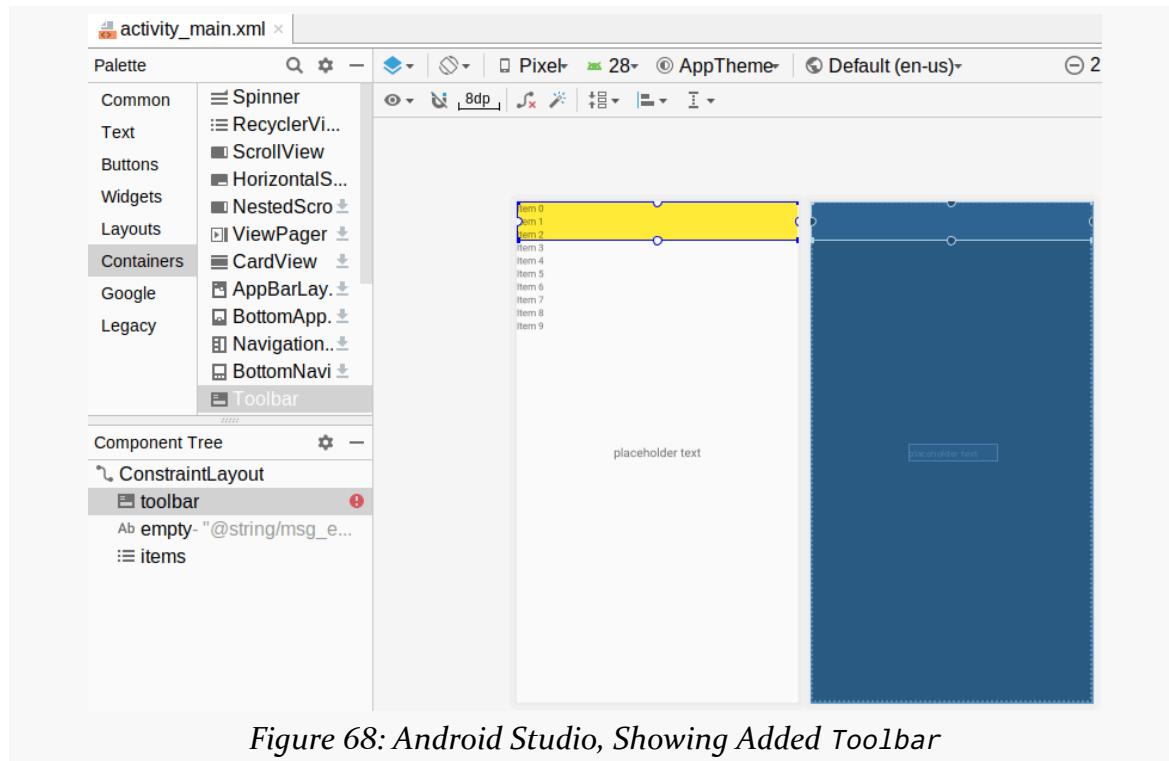


Figure 68: Android Studio, Showing Added Toolbar

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



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

## SETTING UP THE APP BAR

---

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

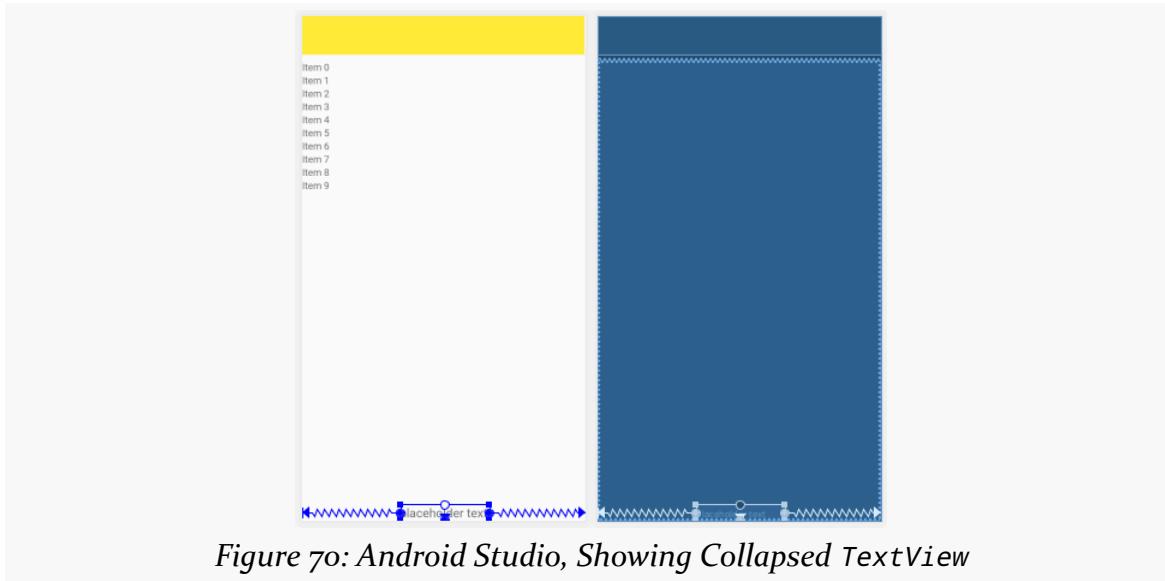


Figure 70: Android Studio, Showing Collapsed TextView

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

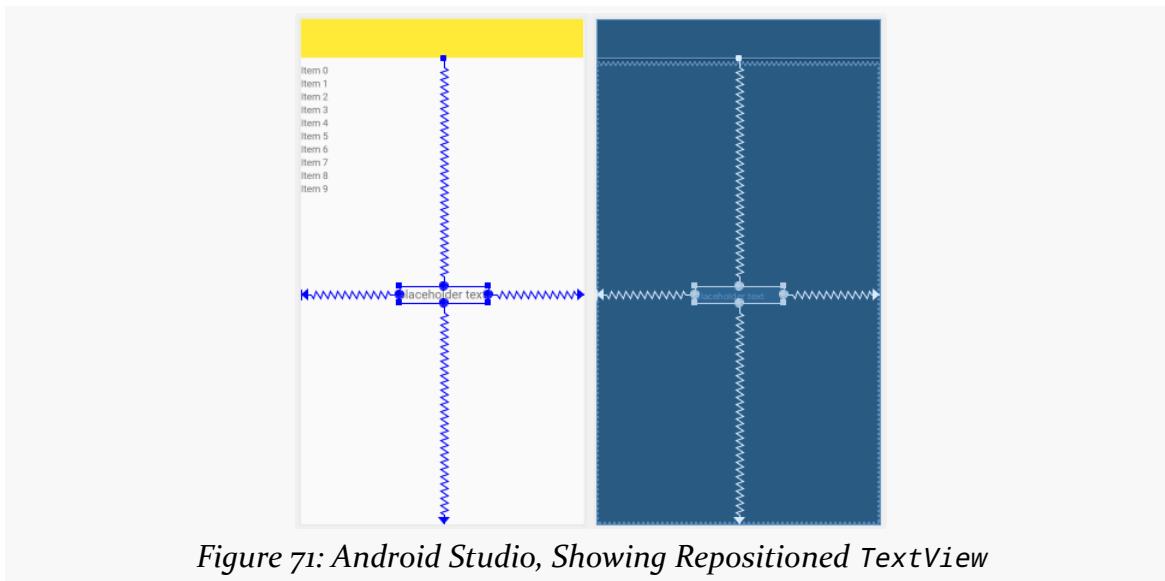


Figure 71: Android Studio, Showing Repositioned TextView

## SETTING UP THE APP BAR

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

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

Then, in the “Attributes” pane:

- Ensure that the ID is set to toolbar (it should be by default)
- Set the layout\_width to match\_constraint (a.k.a., 0dp)
- Clear the margins on the start, top, and end sides

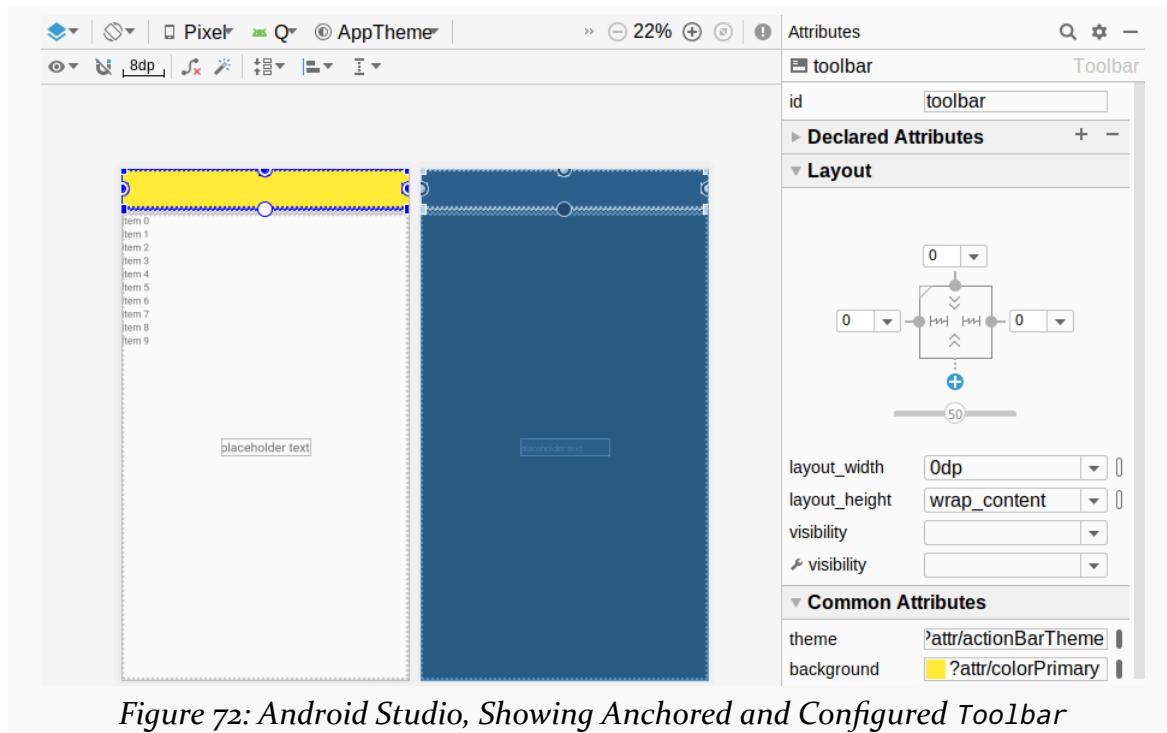


Figure 72: Android Studio, Showing Anchored and Configured Toolbar

In theory, your layout XML should now look like:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
```

## SETTING UP THE APP BAR

---

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

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

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

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

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

(from [To7-Toolbar/ToDo/app/src/main/res/layout/activity\\_main.xml](#))

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

### Step #4: Adding an Icon

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

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

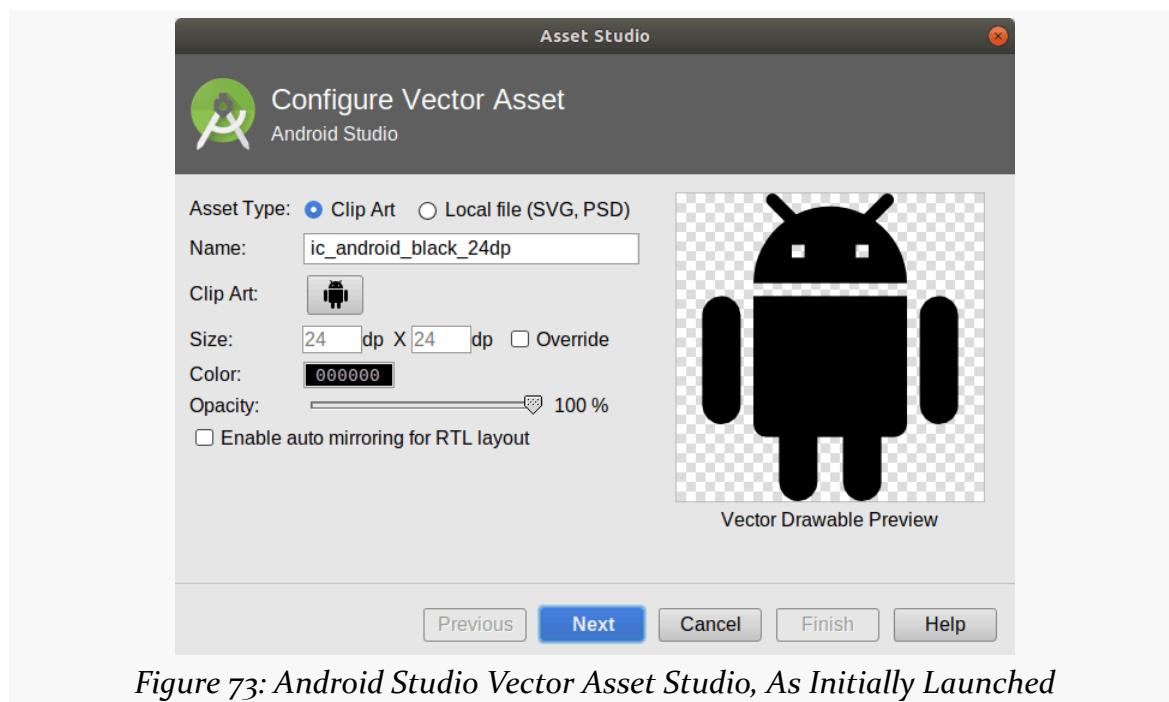


Figure 73: Android Studio Vector Asset Studio, As Initially Launched

## SETTING UP THE APP BAR

---

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

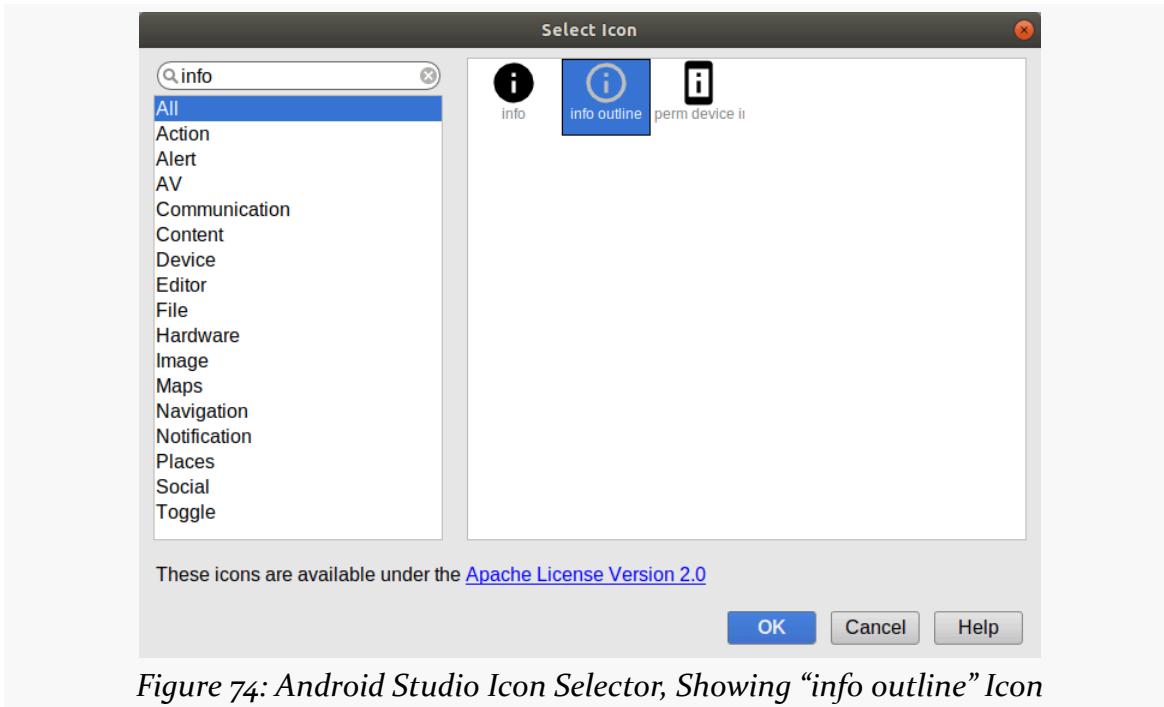


Figure 74: Android Studio Icon Selector, Showing “info outline” Icon

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

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

## Step #5: Defining an Item

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

## SETTING UP THE APP BAR

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

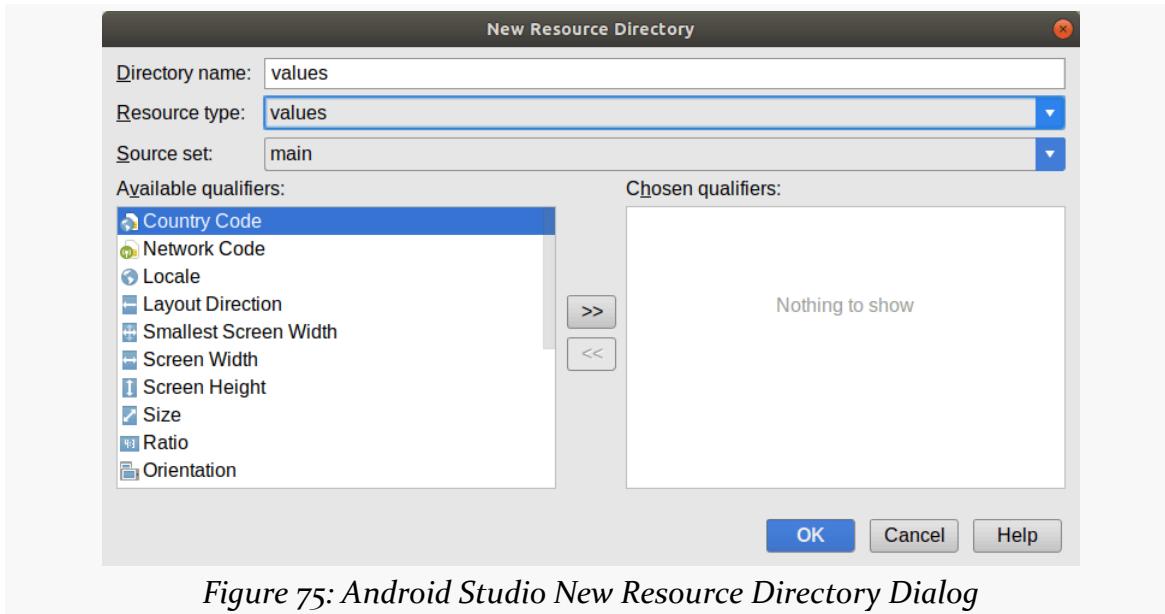


Figure 75: Android Studio New Resource Directory Dialog

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

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

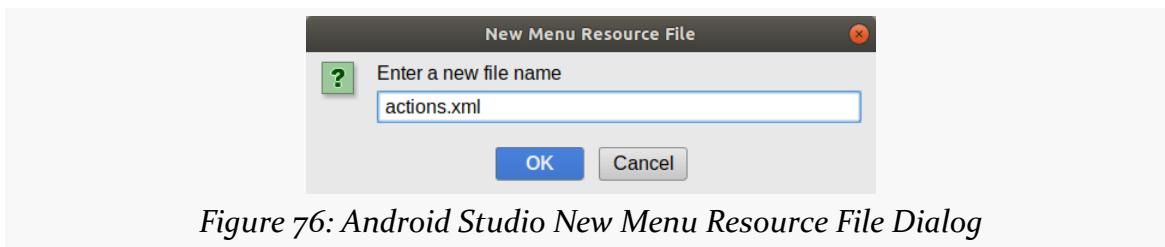


Figure 76: Android Studio New Menu Resource File Dialog

## SETTING UP THE APP BAR

---

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

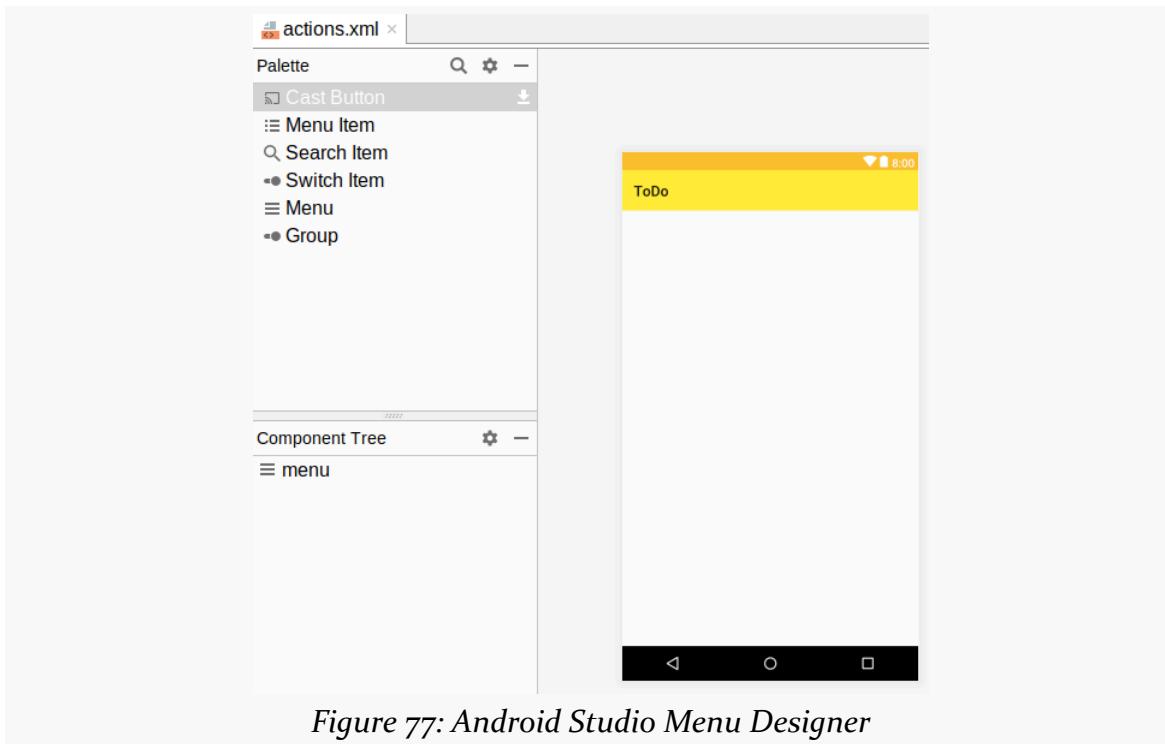


Figure 77: Android Studio Menu Designer

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

## 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:

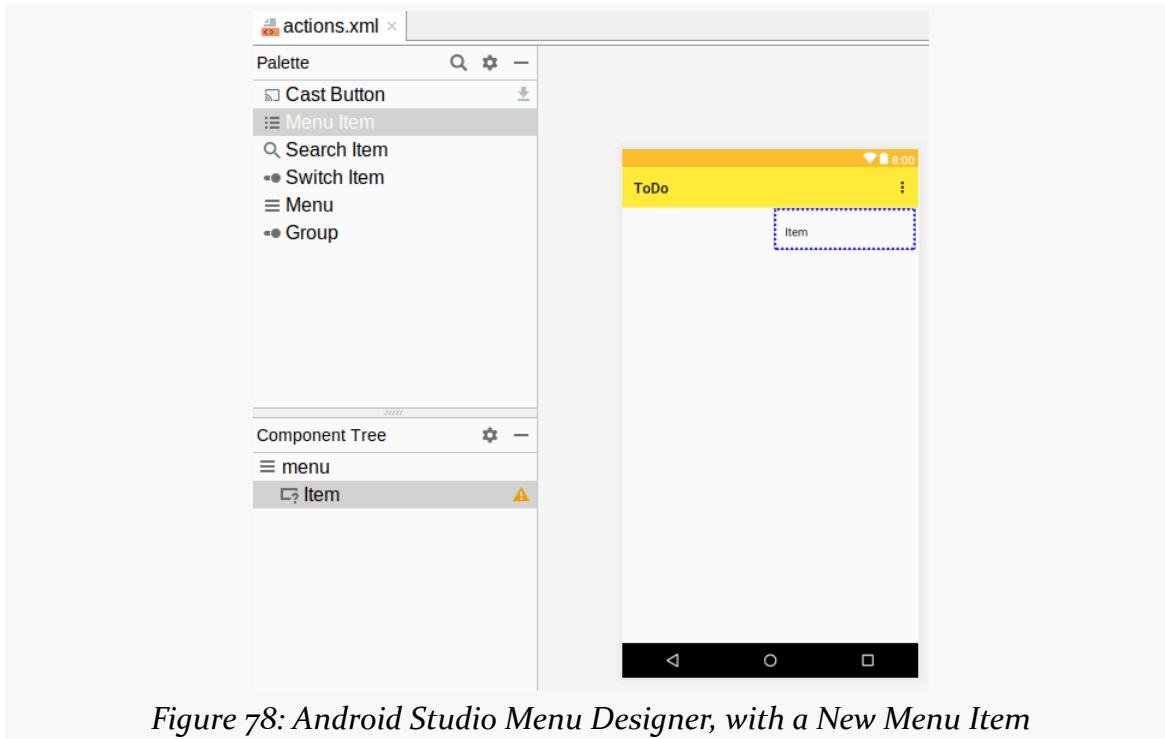


Figure 78: Android Studio Menu Designer, with a New Menu Item

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:

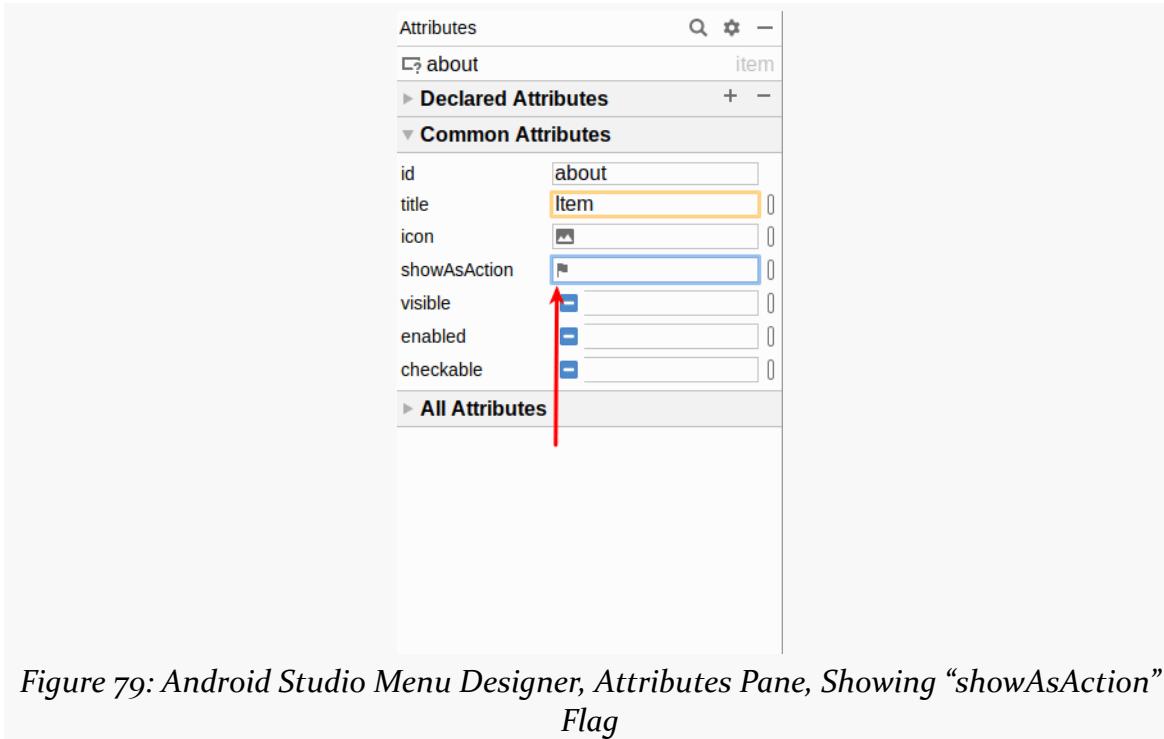


Figure 79: Android Studio Menu Designer, Attributes Pane, Showing “showAsAction” Flag

## SETTING UP THE APP BAR

---

That will fold open a list of available choices:

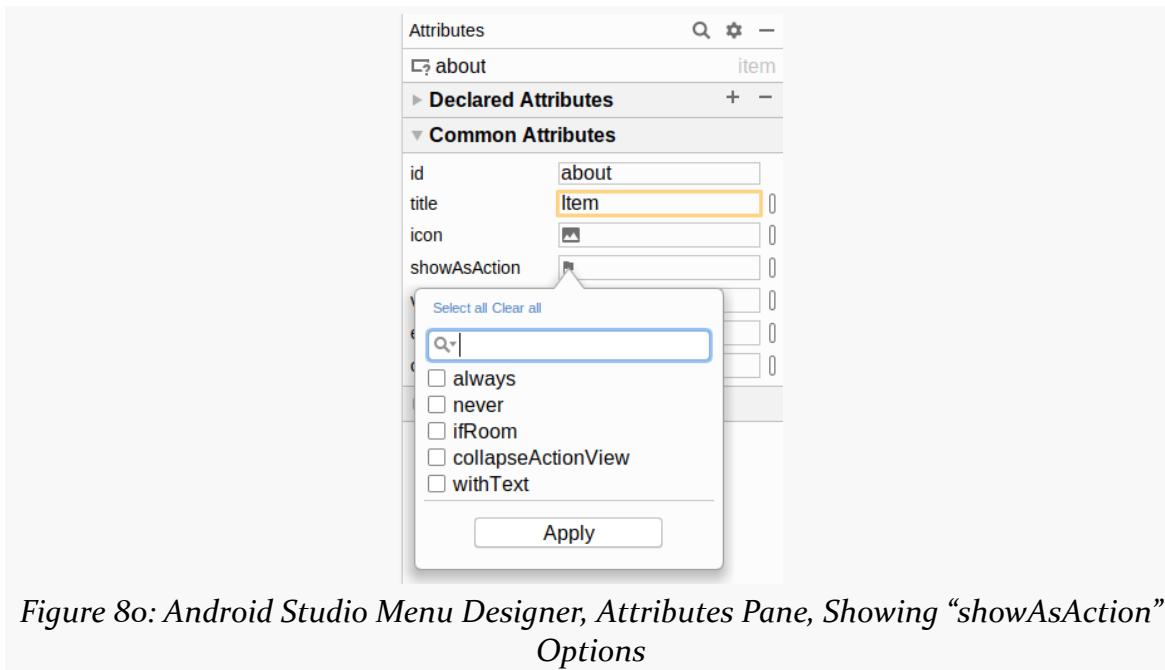


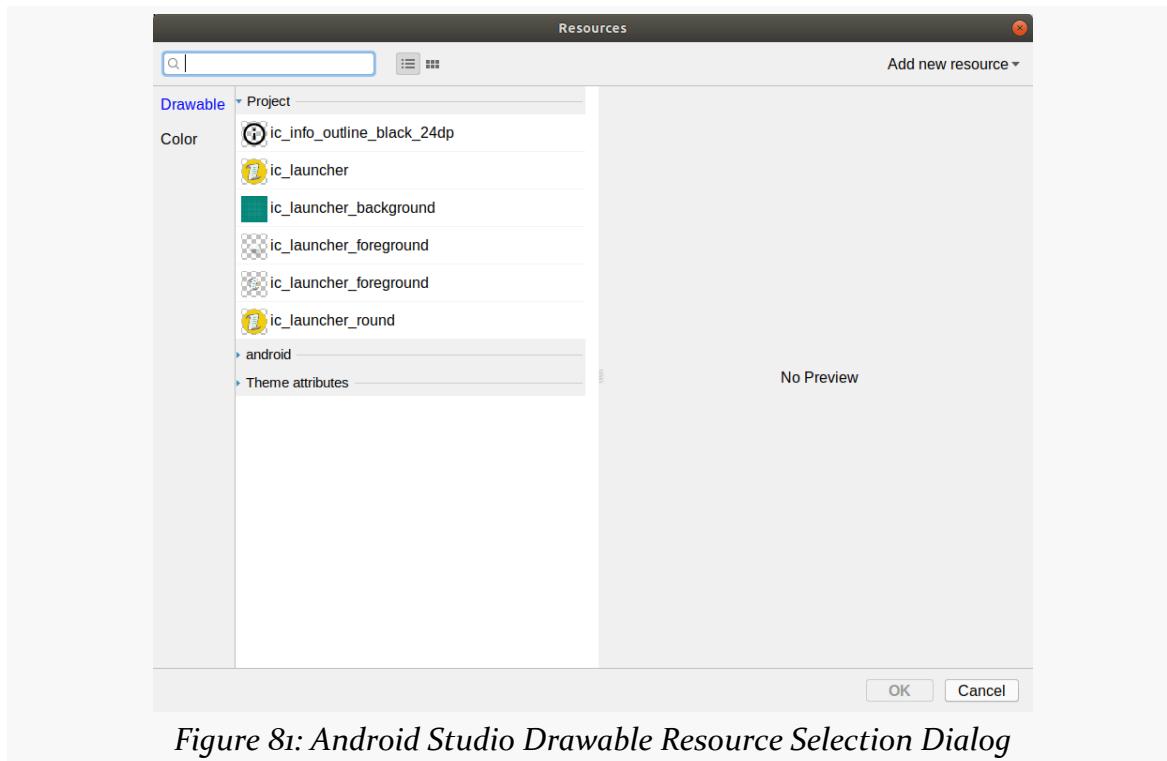
Figure 8o: Android Studio Menu Designer, Attributes Pane, Showing “showAsAction” Options

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. This will bring up a drawable resource selector:



*Figure 81: Android Studio Drawable Resource Selection Dialog*

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

## 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 “Add new resource” > “New string Value” in the drop-down towards the top. In the dialog, fill in `menu_about` as the resource name and “About” as the resource value:

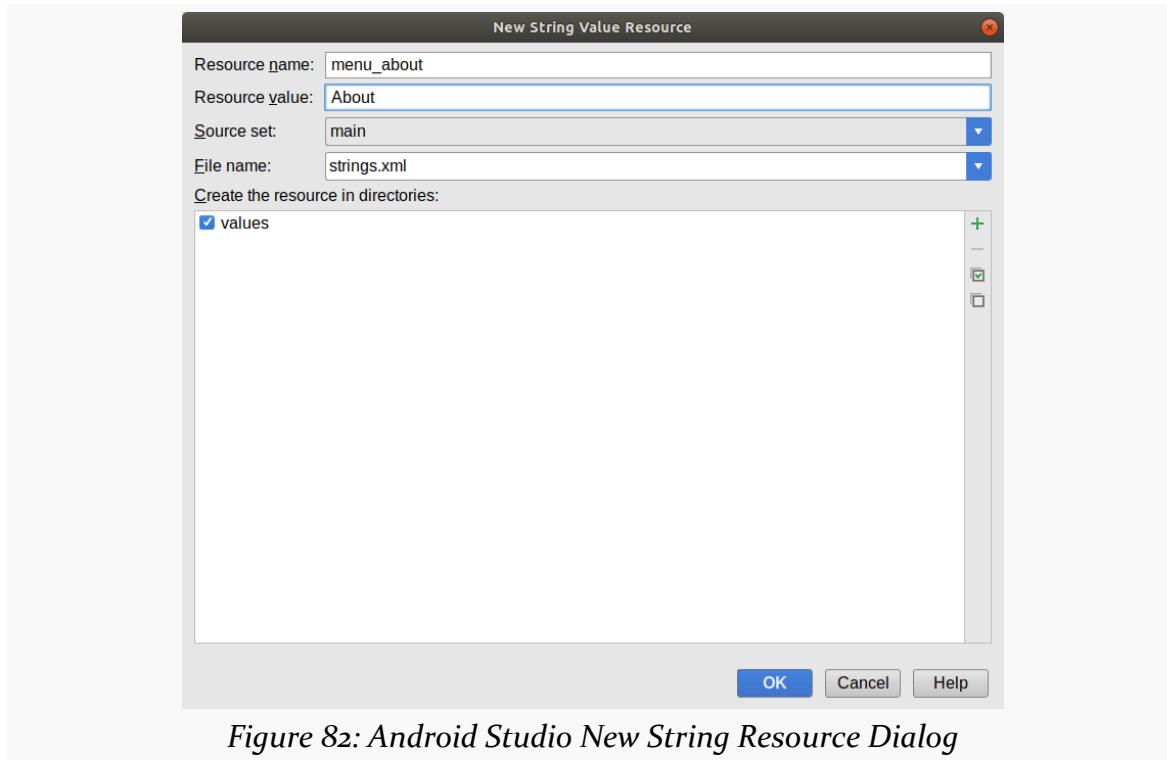


Figure 82: Android Studio New String Resource Dialog

## SETTING UP THE APP BAR

---

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

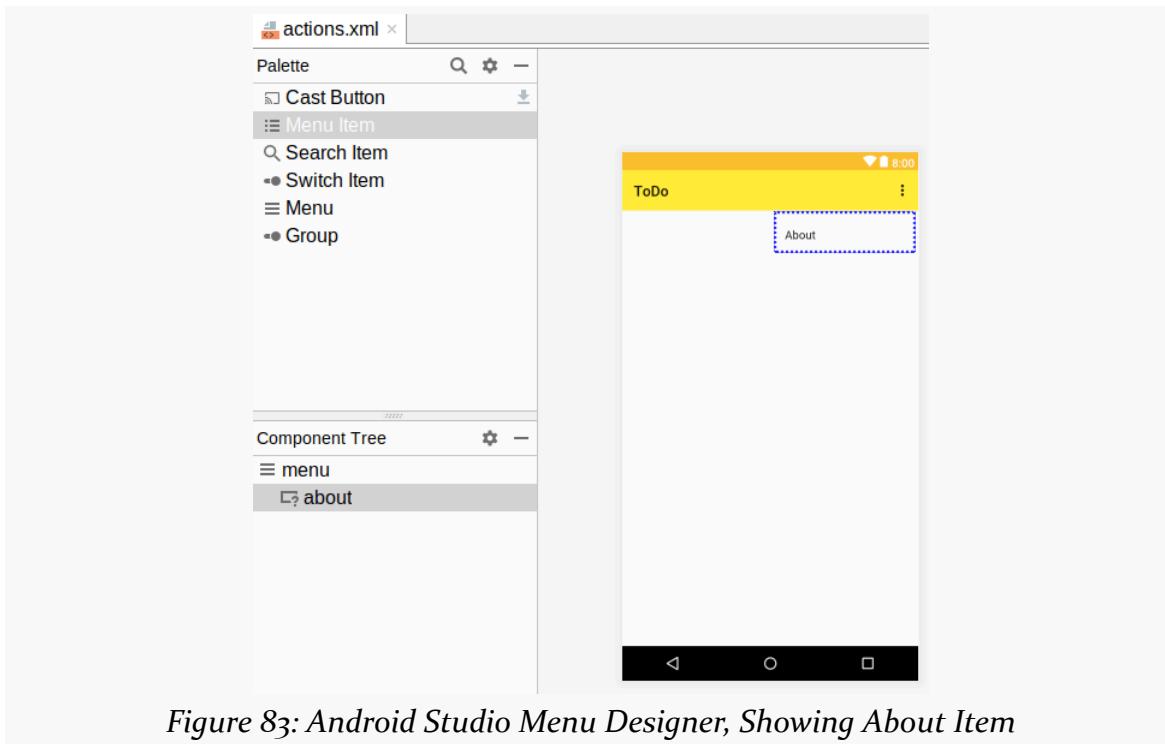


Figure 83: Android Studio Menu Designer, Showing About Item

## Step #6: Enabling Kotlin Synthetic Attributes

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

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

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

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

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

## SETTING UP THE APP BAR

---

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

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

android {
    compileSdkVersion 28

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

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

    androidExtensions {
        experimental = true
    }
}

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

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

## Step #7: Loading Our Options

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

## SETTING UP THE APP BAR

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

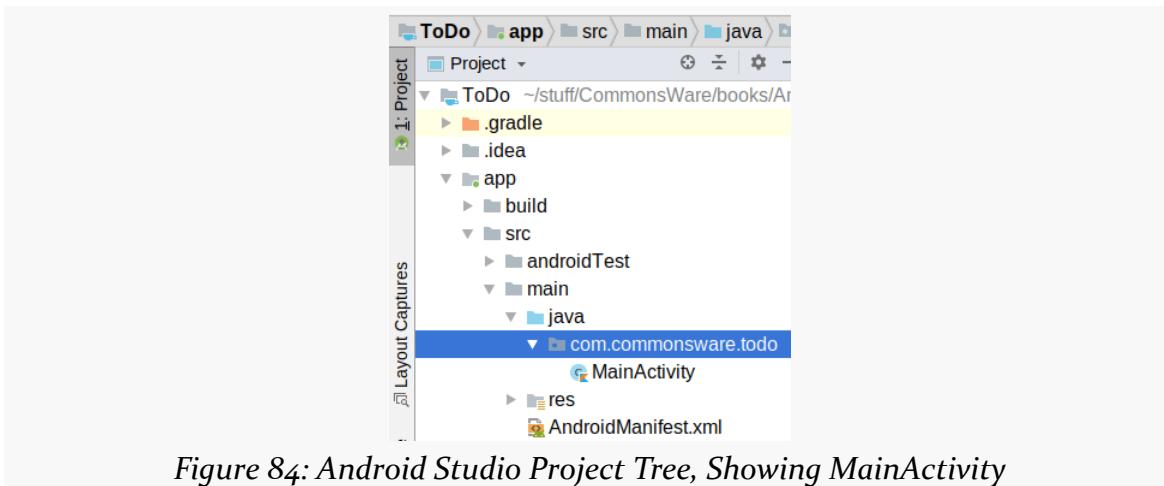


Figure 84: Android Studio Project Tree, Showing MainActivity

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

```
package com.commonware.todo

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

class MainActivity : AppCompatActivity() {

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

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

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

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

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

## SETTING UP THE APP BAR

---

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

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

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

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

Our two lines:

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

### Step 8: Trying It Out

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

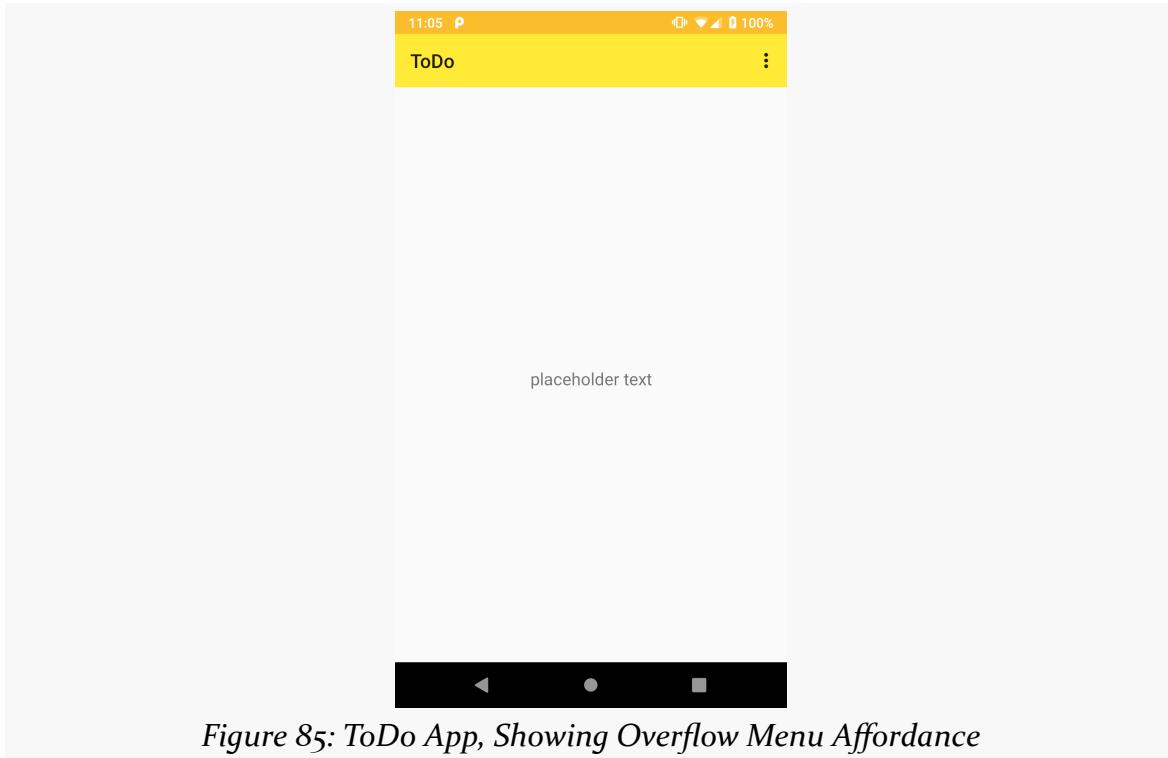
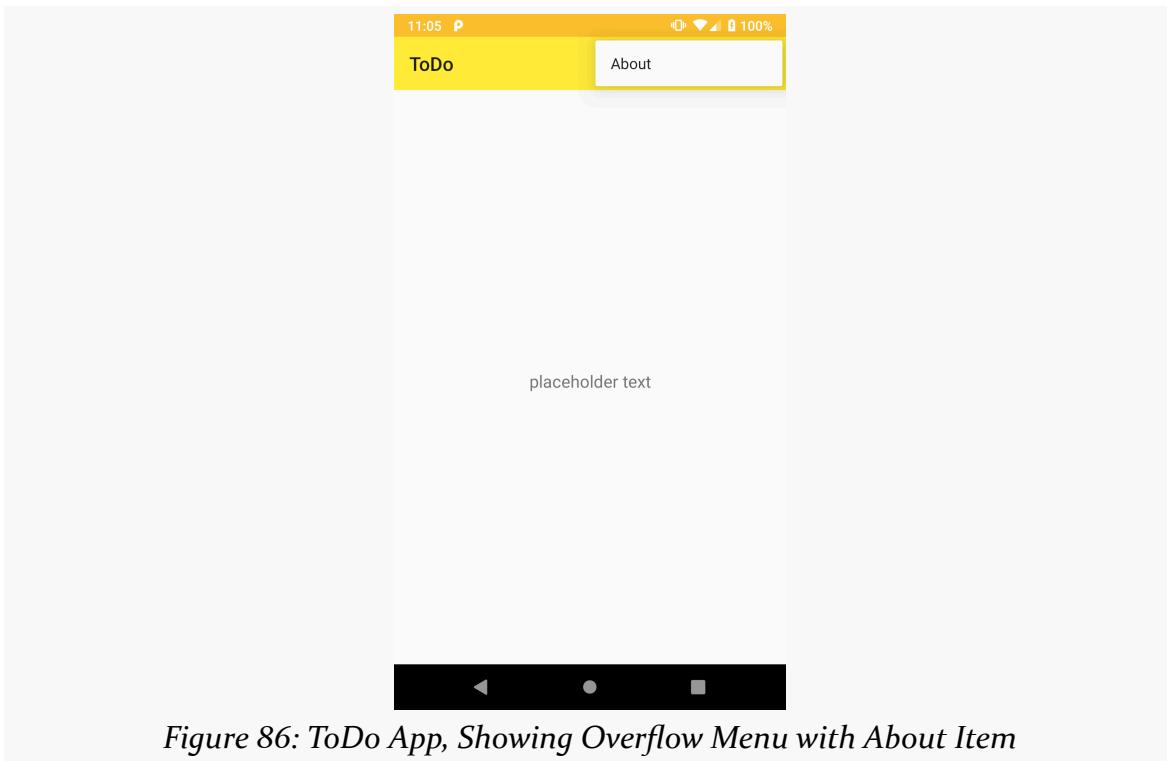


Figure 85: ToDo App, Showing Overflow Menu Affordance

## SETTING UP THE APP BAR

---

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



*Figure 86: ToDo App, Showing Overflow Menu with About Item*

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

## Step #9: Dealing with Crashes

Most likely, you will not need this step.

## SETTING UP THE APP BAR

---

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:

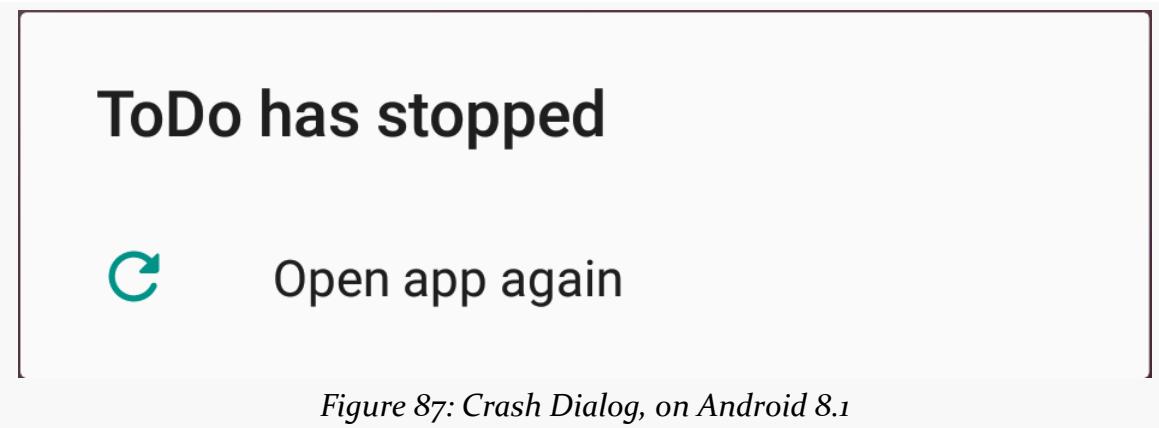


Figure 87: Crash Dialog, on Android 8.1

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:

## SETTING UP THE APP BAR

- 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 88: LogCat, Showing Stack Trace

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

```
8937-8937/com.commonware.todo E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.commonware.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.commonware.todo.MainActivity.onCreateOptionsMenu(MainActivity.java:18)
        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$1.run(PhoneWindow.java:264)
        at android.os.Handler.handleCallback(Handler.java:790)
        at android.os.Handler.dispatchMessage(Handler.java:99)
        at android.os.Looper.loop(Looper.java:164)
        at android.app.ActivityThread.main(ActivityThread.java:6494) <1 internal call>
        at com.android.internal.os.RuntimeInit$MethodAndArgsCaller.run(RuntimeInit.java:438)
        at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:807)
```

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

## SETTING UP THE APP BAR

---

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

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

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

## What We Changed

The book's 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/java/com/commonsware/todo/MainActivity.kt](#)

Licensed solely for use by Patrocinio Rodriguez

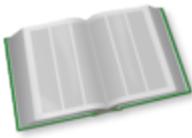
# **Setting Up an Activity**

---

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

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

This is a continuation of the work we did in [the previous tutorial](#). The book’s 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`.

## SETTING UP AN ACTIVITY

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:

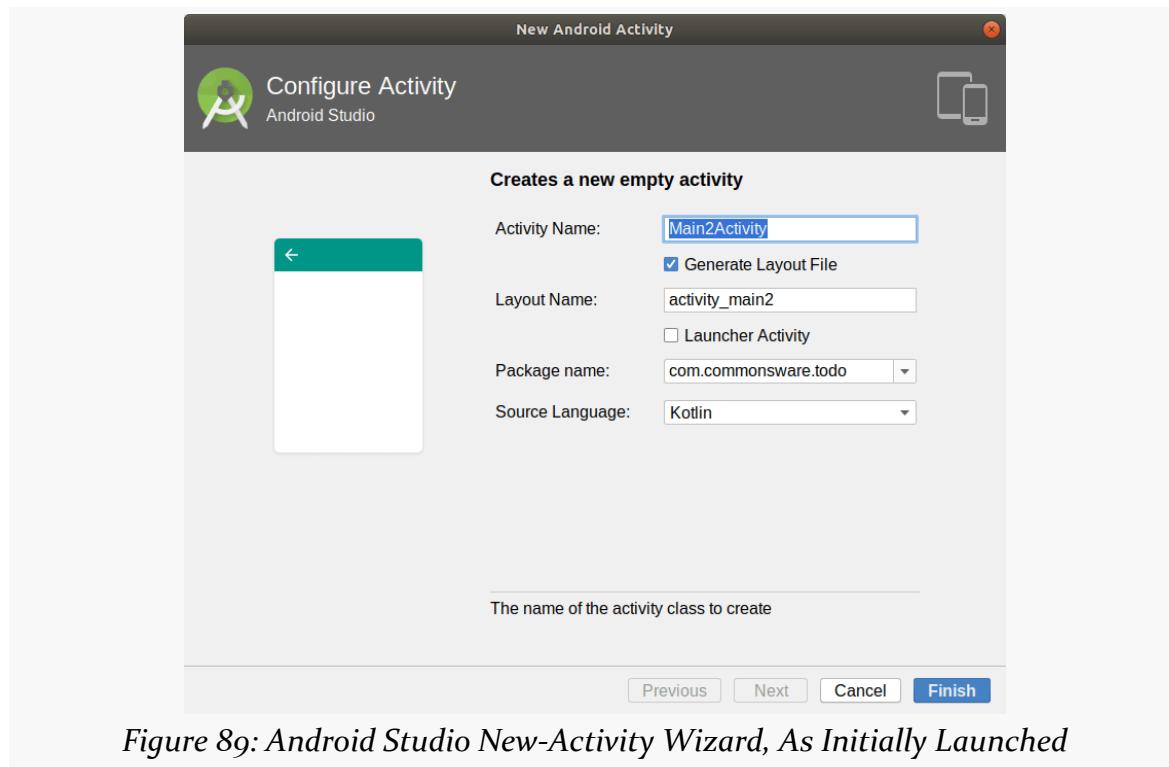


Figure 89: Android Studio New-Activity Wizard, As Initially Launched

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:

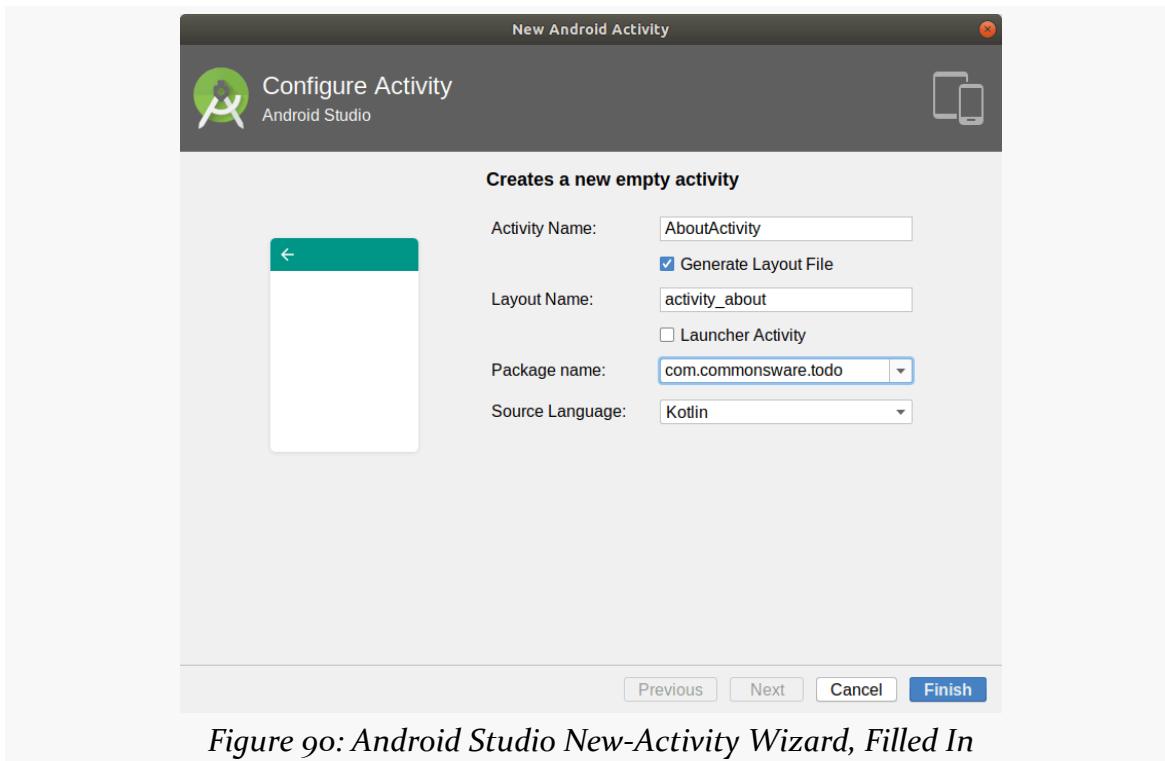


Figure 90: Android Studio New-Activity Wizard, Filled In

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

```
package com.commonsware.todo

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

class AboutActivity : AppCompatActivity() {

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

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

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

## SETTING UP AN ACTIVITY

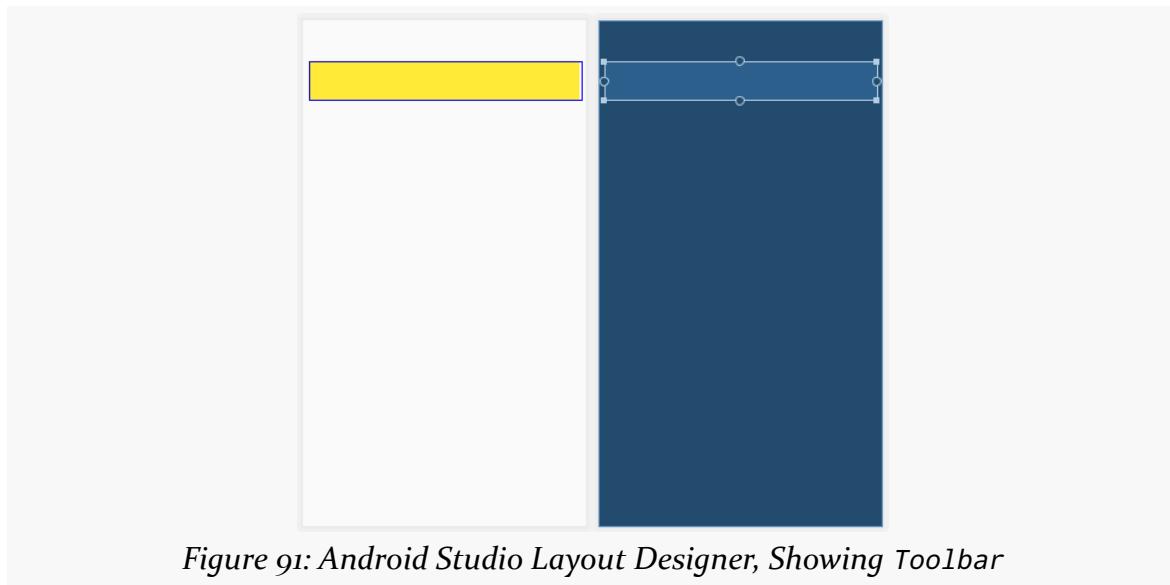
---

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

### 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 91: Android Studio Layout Designer, Showing Toolbar*

## SETTING UP AN ACTIVITY

---

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:

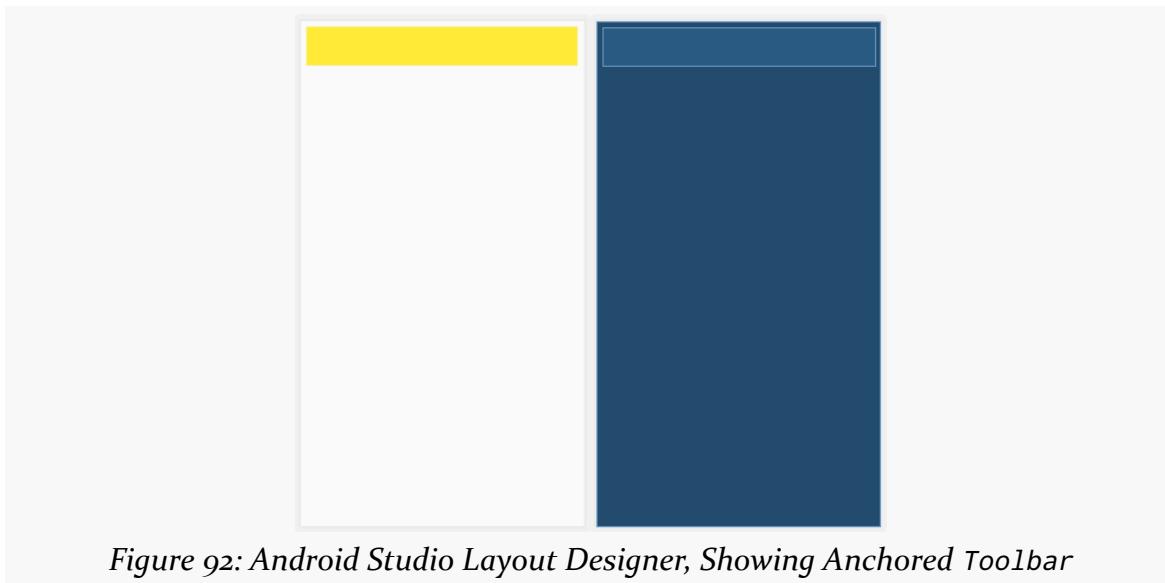


Figure 92: Android Studio Layout Designer, Showing Anchored Toolbar

Then, in the “Attributes” pane, set the layout\_width to be match\_constraint (a.k.a., 0dp), and set the start, top, and end margins to 0dp:

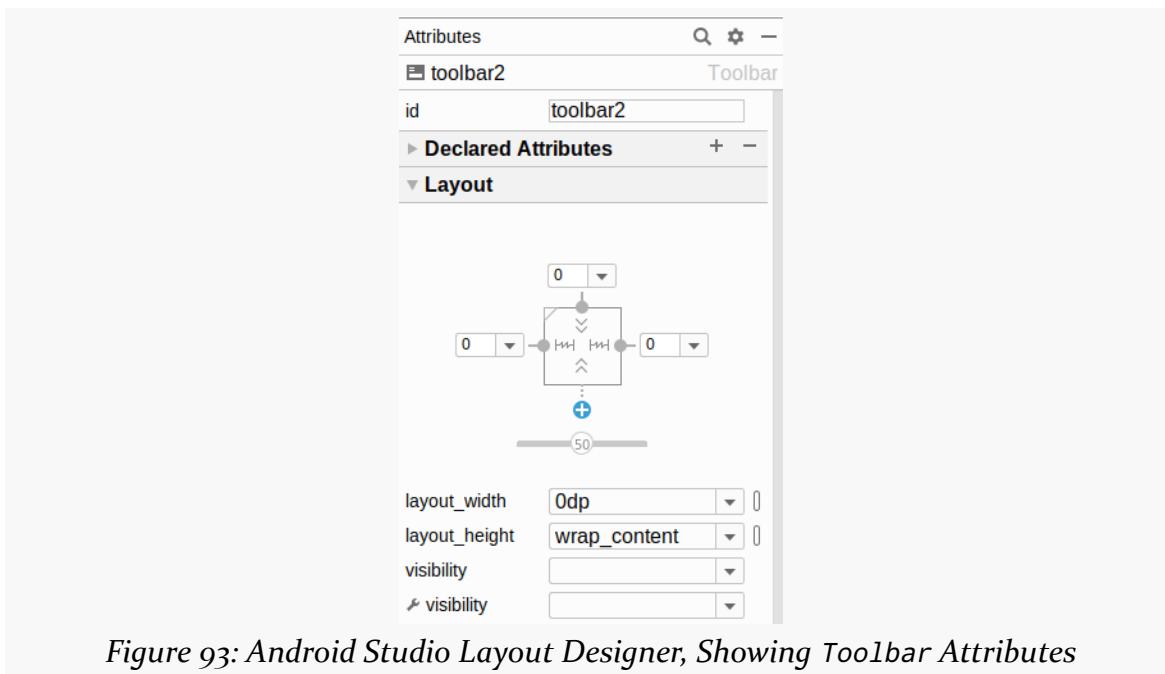


Figure 93: Android Studio Layout Designer, Showing Toolbar Attributes

## SETTING UP AN ACTIVITY

---

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

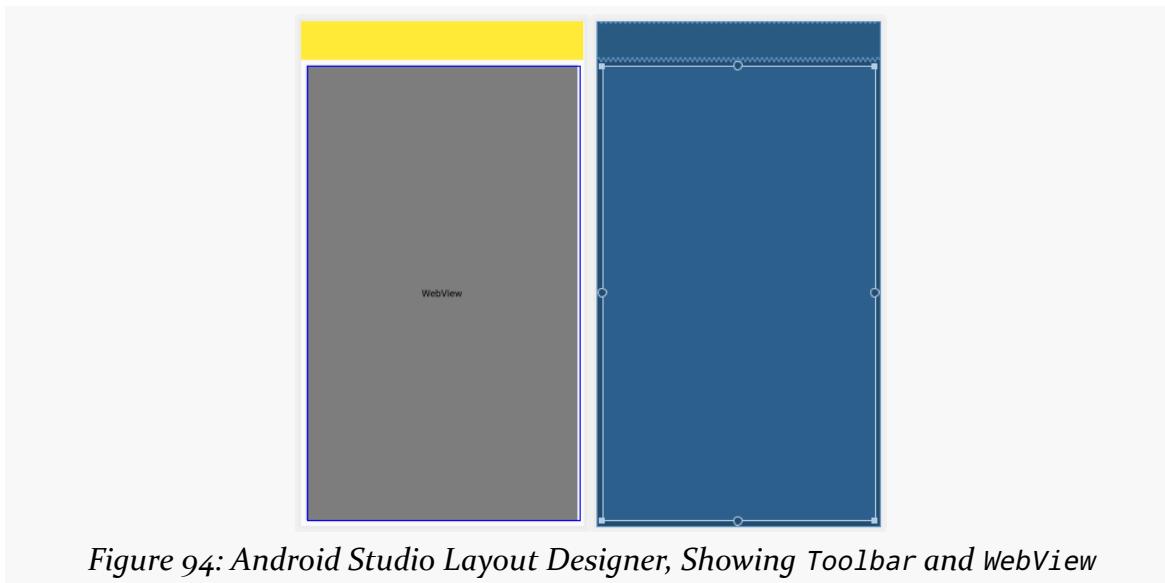


Figure 94: Android Studio Layout Designer, Showing Toolbar and WebView

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

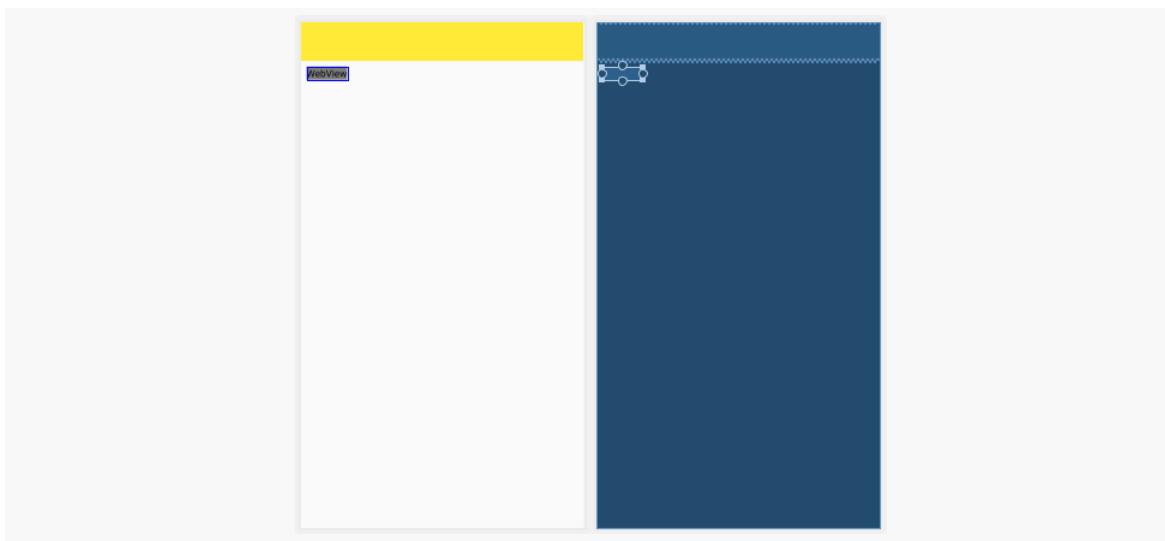


Figure 95: Android Studio Layout Designer, Showing Smaller WebView

## 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`:

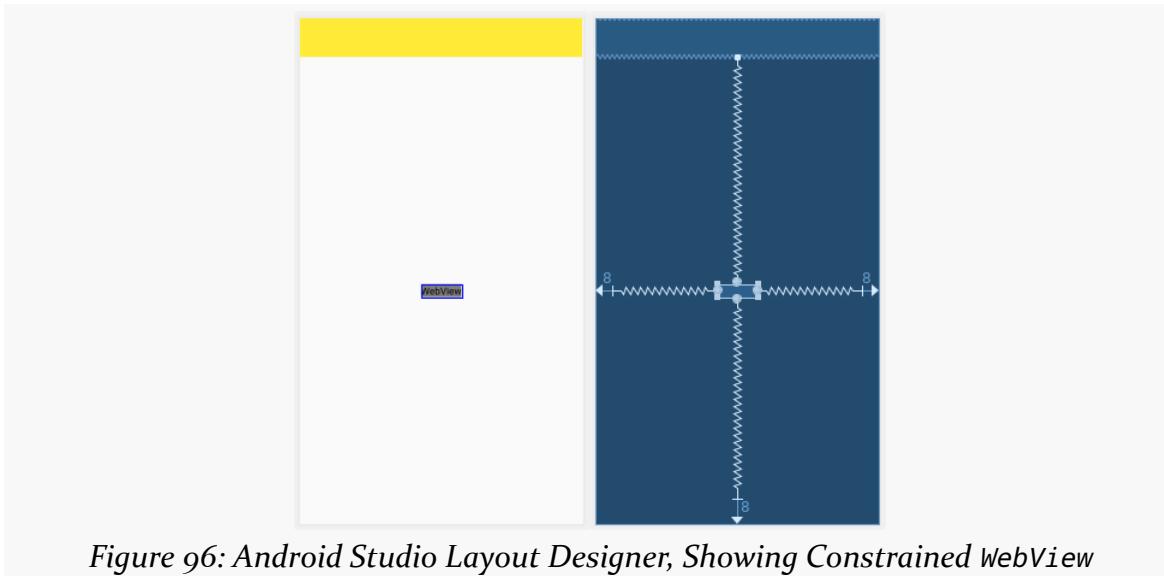
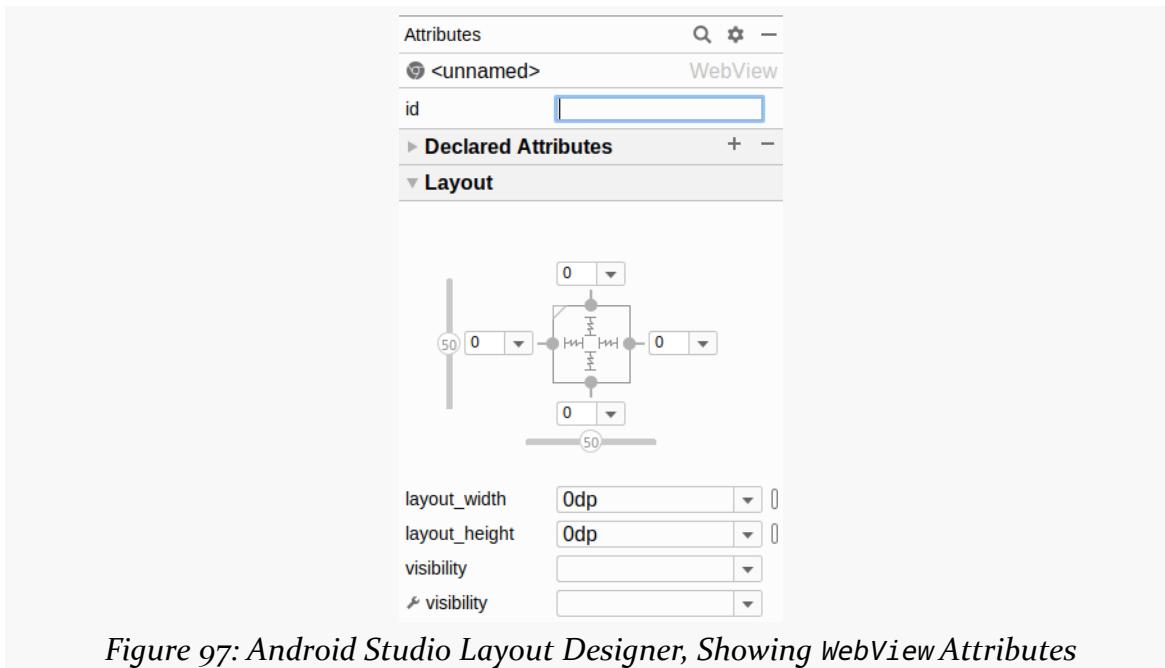


Figure 96: Android Studio Layout Designer, Showing Constrained `WebView`

## SETTING UP AN ACTIVITY

Then, go back to the “Attributes” pane and set the “layout\_width” and “layout\_height” each to `match_constraint` (a.k.a., `0dp`), to have the `WebView` fill all of the available space, and remove the `8dp` of margin from all four sides:



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

At this point, the layout XML should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".AboutActivity">

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

## SETTING UP AN ACTIVITY

---

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

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

(from [To8-Activities/ToDo/app/src/main/res/layout/activity\\_about.xml](#))

## Step #3: Launching Our Activity

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

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

```
package com.commonsware.todo

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

class MainActivity : AppCompatActivity() {

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

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

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

## SETTING UP AN ACTIVITY

---

```
    true  
}  
}  
}
```

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

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

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

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

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

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

## Step #4: Defining Some About Text

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

## 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:

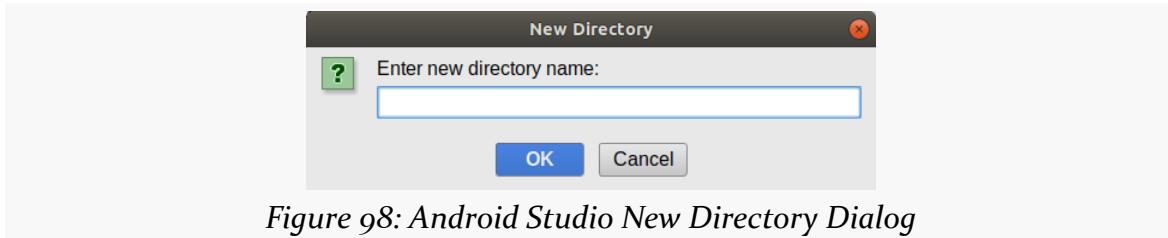


Figure 98: Android Studio New Directory Dialog

Fill in assets and click “OK” to create this directory.

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

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

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

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

<p>No, really — this app is awesome!</p>

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

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

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

## Step #5: Populating the Toolbar and WebView

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

## SETTING UP AN ACTIVITY

---

```
package com.commonsware.todo

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

class AboutActivity : AppCompatActivity() {

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

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

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

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

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

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

## SETTING UP AN ACTIVITY

---

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

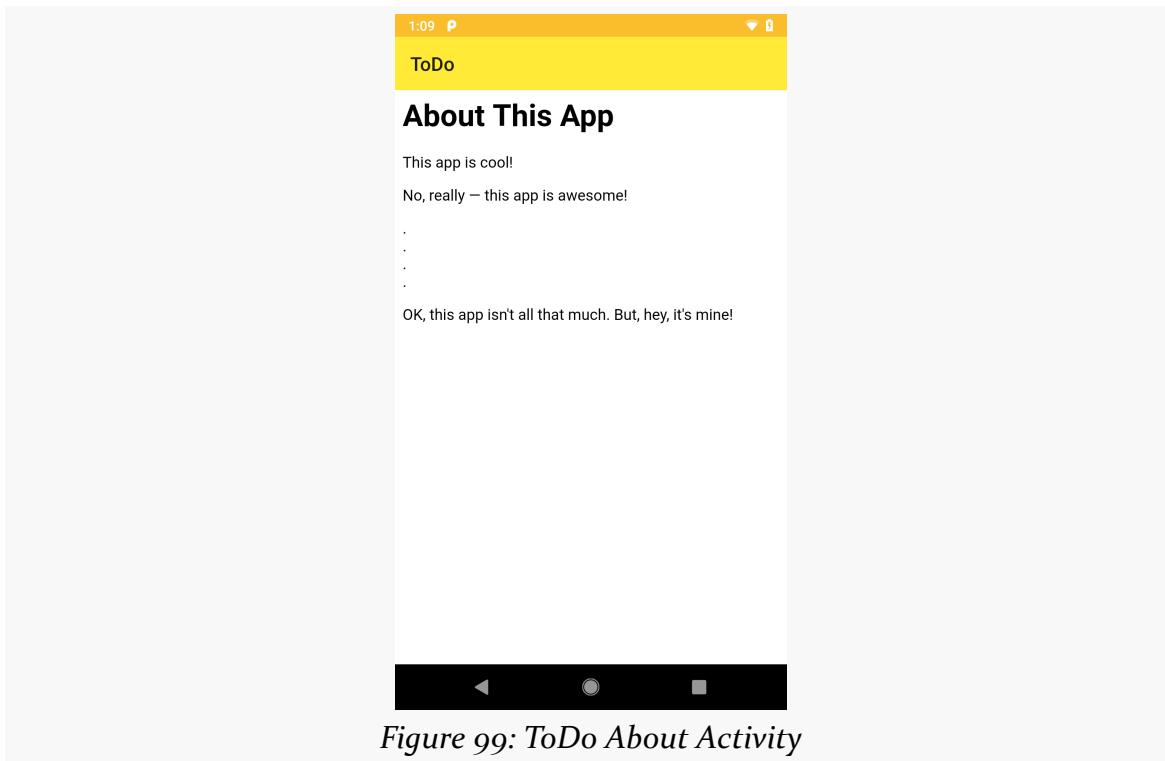


Figure 99: ToDo About Activity

## 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](#)
- [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](#)

Licensed solely for use by Patrocinio Rodriguez

# Integrating Fragments

---

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

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

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

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

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

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

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

## INTEGRATING FRAGMENTS

[completing the work in this tutorial.](#)



You can learn more about fragments in the "Adopting Fragments" chapter of [\*Elements of Android Jetpack\*](#)!

## Step #1: Creating a Fragment

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

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

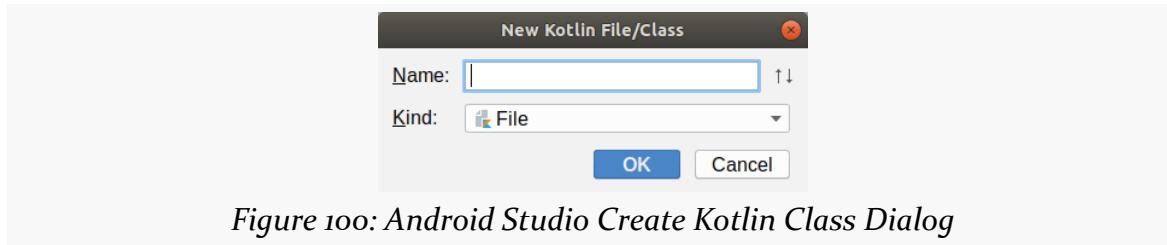


Figure 100: Android Studio Create Kotlin Class Dialog

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

```
package com.commonsware.todo

class RosterListFragment { }
```

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

```
package com.commonsware.todo

import androidx.fragment.app.Fragment

class RosterListFragment : Fragment() { }
```

## INTEGRATING FRAGMENTS

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

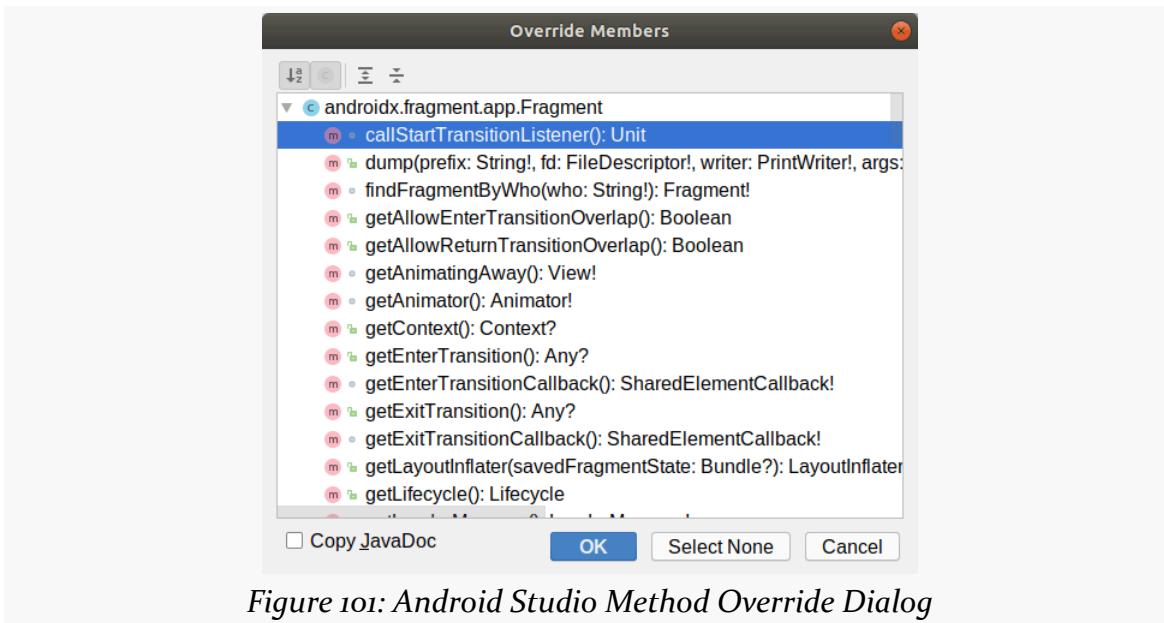


Figure 101: Android Studio Method Override Dialog

## 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:

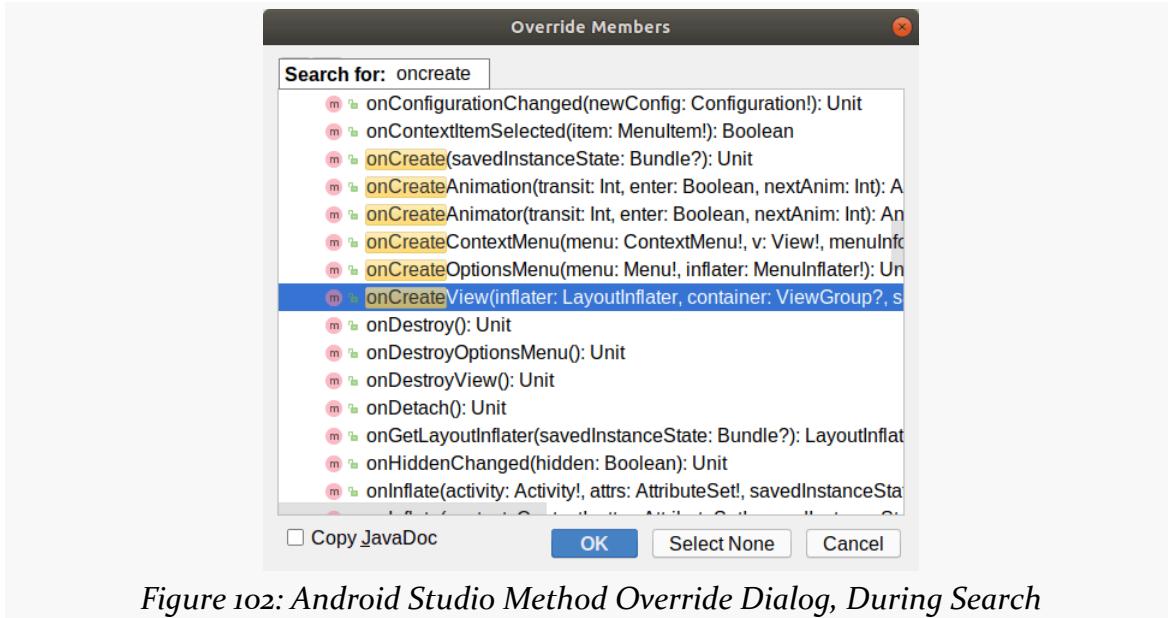


Figure 102: Android Studio Method Override Dialog, During Search

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

```
package com.commonsware.todo

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

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

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

inheriting from Fragment.

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

```
package com.commonsware.todo

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

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

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

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

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

## Step #2: Updating the Toolbar

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

## INTEGRATING FRAGMENTS

---

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

```
package com.commonsware.todo

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

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

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

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

Then, modify `onViewCreated()` to be:

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

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

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

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

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

## INTEGRATING FRAGMENTS

---

inflated UI.

At this point, RosterListFragment should look like:

```
package com.commonsware.todo

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

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

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

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

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

## Step #3: Add the KTX Dependency

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

```
dependencies {
```

## INTEGRATING FRAGMENTS

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

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

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

## Step #4: Displaying the Fragment

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

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

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

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

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

```
package com.commonsware.todo

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

class MainActivity : AppCompatActivity() {

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

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

## INTEGRATING FRAGMENTS

---

```
}
```

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

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

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

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

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

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

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

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

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

## INTEGRATING FRAGMENTS

---

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

## Step #5: Renaming Our Layout Resource

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

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

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

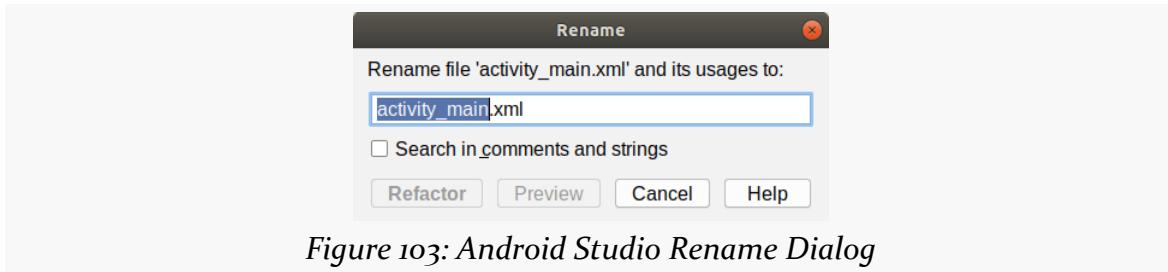


Figure 103: Android Studio Rename Dialog

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

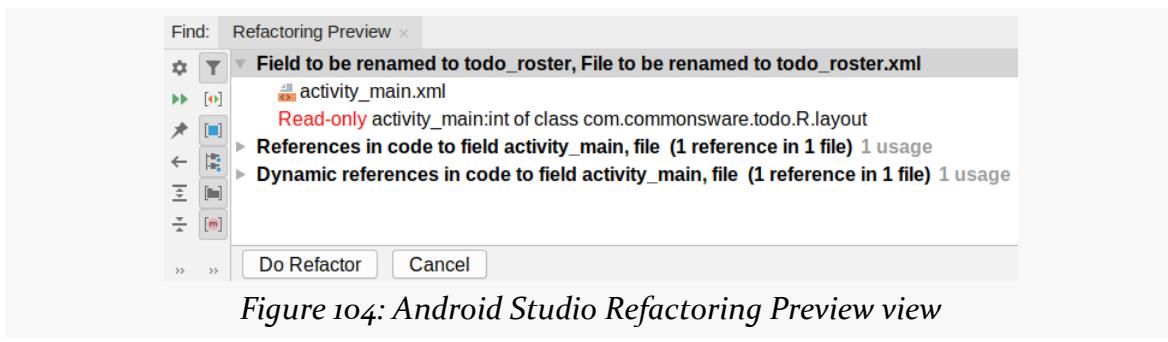


Figure 104: Android Studio Refactoring Preview view

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

Click the “Do Refactor” button towards the bottom of the “Refactoring Preview”

## INTEGRATING FRAGMENTS

---

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

```
package com.commonsware.todo

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

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

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

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

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

            true
        }
    }
}
```

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

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

## What We Changed

The book's 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/layout/todo\\_roster.xml](#)
- [app/src/main/java/com/commonsware/todo/MainActivity.kt](#)

## INTEGRATING FRAGMENTS

---

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

# Defining a Model

---

If we are going to show to-do items in this list, it would help to have some to-do items. That, in turn, means that we need a 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 “Kind” drop-down list. Then click “OK” to create this class. `ToDoModel` should show up in an editor, with an implementation like this:

```
package com.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. This makes them immutable: you do not change a model, but instead replace it with a

## DEFINING A MODEL

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:

A screenshot of the Android Studio code editor. The file is named `ToDoModel.kt`. The code is as follows:

```
1 package com.commonsware.todo
2
3 data class ToDoModel {
```

The word `data` is underlined with a red squiggle, and there is a red squiggle under the opening brace of the class definition. The code editor interface is visible around the code.

Figure 105: Android Studio, Yelling

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

## Step #3: Adding the Constructor

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

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

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

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

## DEFINING A MODEL

---

```
import java.util.*

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

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

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

Of particular note:

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

## What We Changed

The book's 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](#)

Licensed solely for use by Patrocinio Rodriguez

# **Setting Up a Repository**

---

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

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

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

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

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

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



You can learn more about Kotlin and object in the “The object Keyword” chapter of [Elements of Kotlin!](#)

## SETTING UP A REPOSITORY

---

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](#).

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

## Step #1: Adding the Object

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

Right-click over the `com.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 “Object” from the “Kind” drop-down. Then click “OK” to create this file. `ToDoRepository` should show up in an editor, with an implementation like this:

```
package com.commonware.todo

object ToDoRepository {
```

## Step #2: Creating Some Fake Data

At the moment, our repository has no data. We need to fix this, so that we have some to-do items to show in our UI. But 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:

## SETTING UP A REPOSITORY

---

```
package com.commonsware.todo

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

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

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

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

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

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

## What We Changed

The book's 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/ToDoRepository.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# 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`.

Right now, we are going to take a fairly simplistic approach to the problem, having the fragment work directly with the repository. That will work for now, but it is not a great choice. Once we start allowing the user to view and edit to-do items, plus start saving this data in a database, we will need a more sophisticated approach. But, that is a task for the future — today, we will keep it simple.

However, we will explore another feature of the Android ecosystem: the data binding framework. This makes it a bit easier to pour data from objects, such as our `ToDoModel` objects, into UI layouts.

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\*](#)!



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

## Step #1: Adding Data Binding Support

Data binding is easy to enable, but we do have to explicitly enable it. It adds a bit of extra time to the build process, so it is disabled by default.

To enable it, open `app/build.gradle` and add this closure to the `android` closure:

```
dataBinding {  
    enabled = true  
}
```

(from [T12-RecyclerView/ToDo/app/build.gradle](#))

This will give you an `android` closure like:

```
android {  
    compileSdkVersion 28  
  
    defaultConfig {  
        applicationId "com.commonsware.todo"  
        minSdkVersion 21  
        targetSdkVersion 28  
        versionCode 1  
        versionName "1.0"  
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    }  
  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
        }  
    }  
  
    androidExtensions {  
        experimental = true  
    }  
  
    dataBinding {  
        enabled = true  
    }  
}
```

(from [T12-RecyclerView/ToDo/app/build.gradle](#))

## Step #2: Defining a Row Layout

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

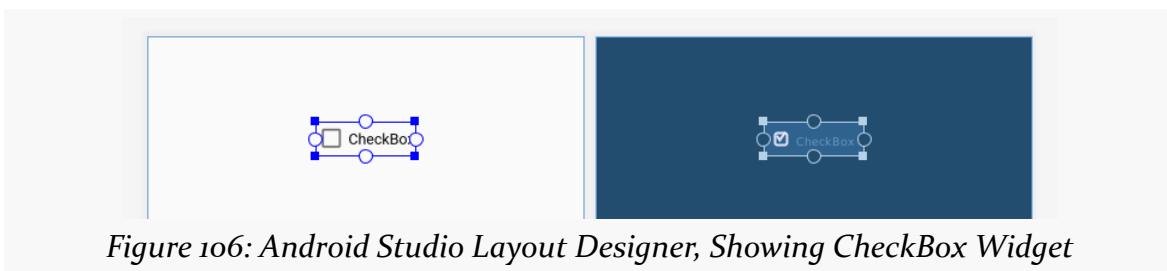
## POPULATING OUR RECYCLERVIEW

---

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.

For now, each row will be a `CheckBox`. This can show the description of the to-do item along with the is-completed status (checked items being those that are completed).

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



*Figure 106: Android Studio Layout Designer, Showing CheckBox Widget*

## POPULATING OUR RECYCLERVIEW

---

Use the round grab handles to drag connections from the CheckBox to the four sides of the ConstraintLayout:

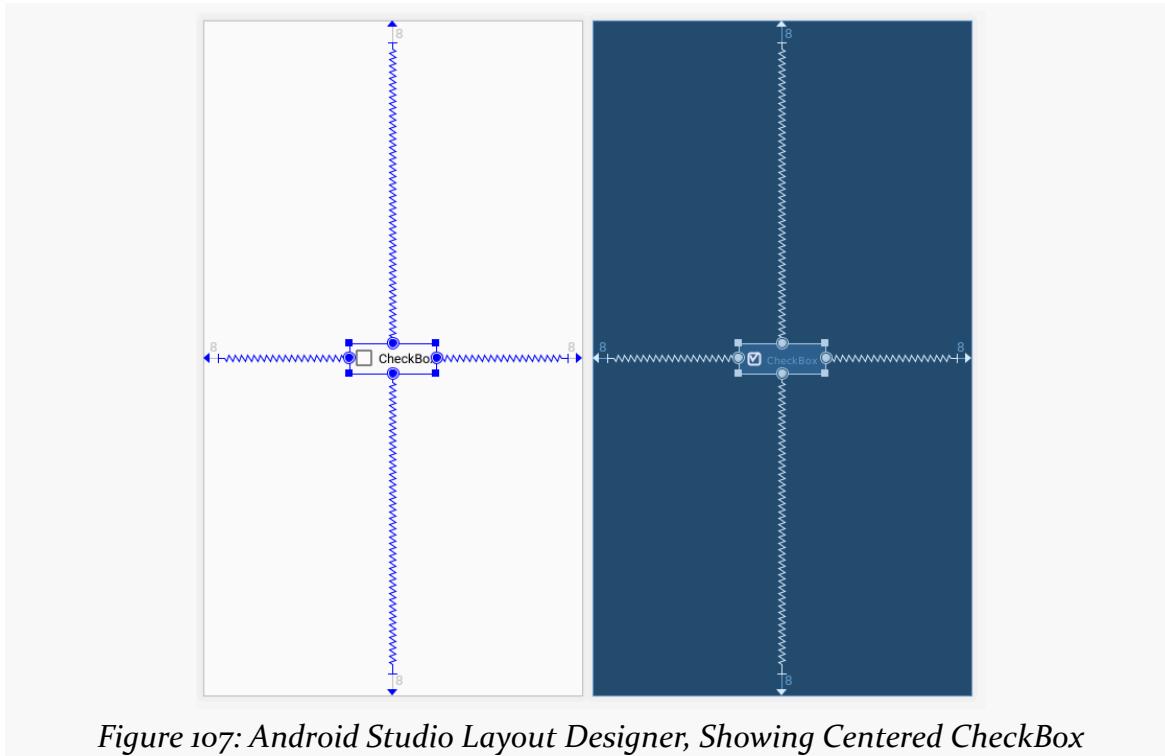


Figure 107: Android Studio Layout Designer, Showing Centered CheckBox

## POPULATING OUR RECYCLERVIEW

---

In the “Attributes” tool, change the layout\_width drop-down to be match\_constraint (a.k.a., 0dp). This will have the CheckBox stretch to fill all available space on the horizontal axis:

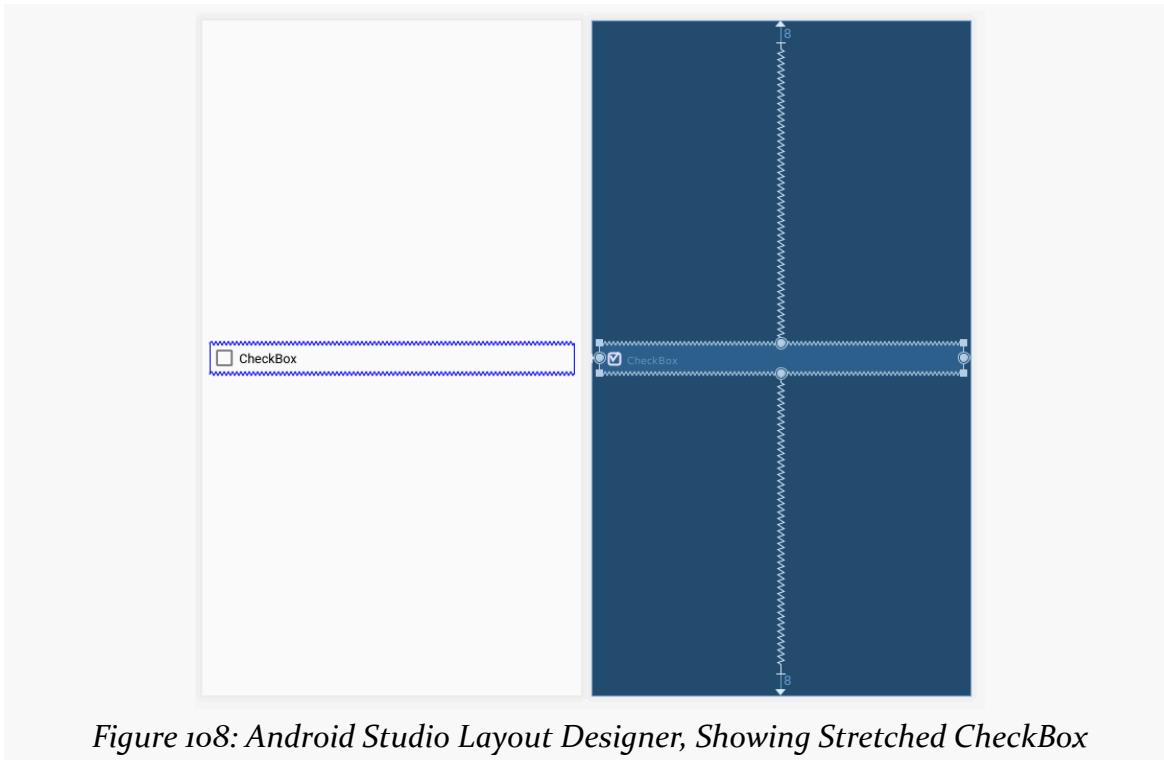


Figure 108: Android Studio Layout Designer, Showing Stretched CheckBox

## POPULATING OUR RECYCLERVIEW

---

Back in the “Attributes” tool, scroll down and open up the “All Attributes” section. This folds open a *long* list of attributes that are available on a CheckBox:

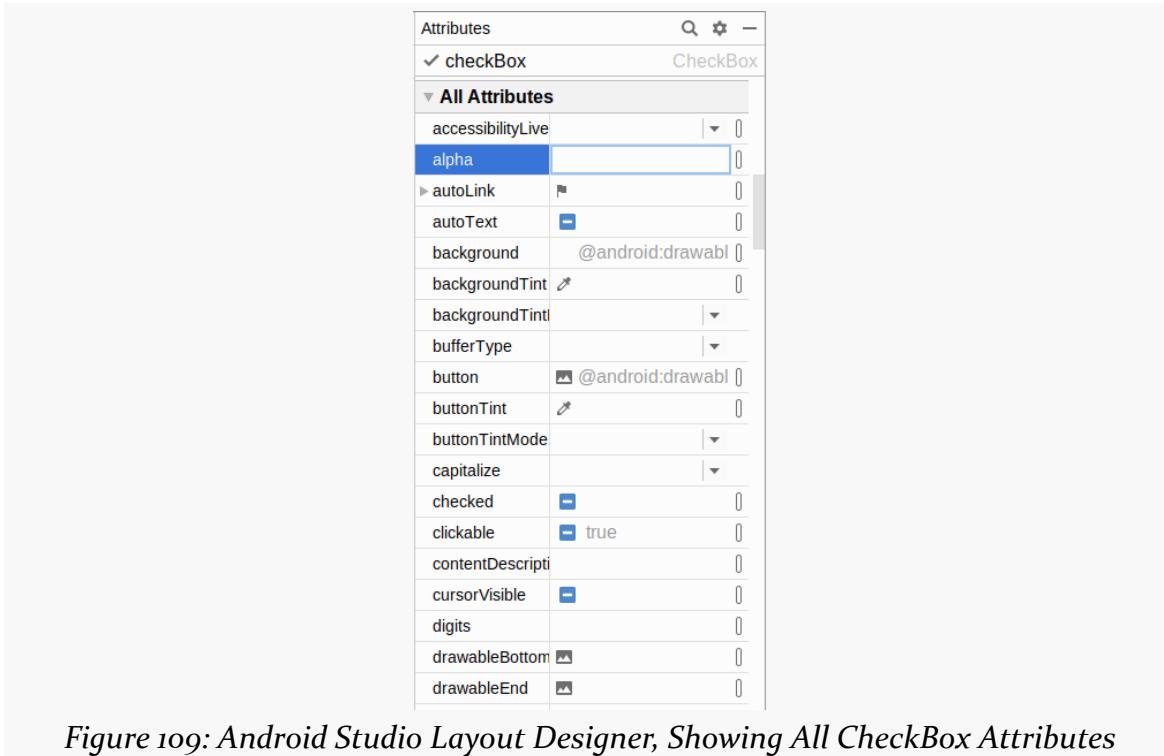


Figure 109: Android Studio Layout Designer, Showing All CheckBox Attributes

Make the following changes:

Attribute Name	New Value
id	desc
ellipsize	end
maxLines	3

(for ellipsize, choose end in the drop-down list)

The latter two attributes say that we will show up to three lines for the description, and if the description exceeds that amount of space, put an ellipsis (...) after whatever fits, at the end.

## POPULATING OUR RECYCLERVIEW

---

Finally, modify `layout_height` on both the `CheckBox` and the `ConstraintLayout` to be `wrap_content`. You can do this by clicking on the `ConstraintLayout` entry in the “Component Tree” view (below the Palette), then changing the `layout_height` in the “Attributes” pane.

We will have some other changes to make later, but this will get us going for now. If you switch to the “Text” sub-tab, the XML of the layout 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="wrap_content">

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

## Step #3: 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

## POPULATING OUR RECYCLERVIEW

---

File/Class” from the context menu. Fill in RosterRowHolder as the “Name” and choose “Class” from the “Kind” drop-down. Then, click “OK” 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 #4: Creating a Stub Adapter

A RecyclerView.ViewHolder is managed by a RecyclerView.Adapter. The Adapter 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.commonsware.todo Java package and choose “New” > “Kotlin File/Class” from the context menu. Fill in RosterAdapter as the “Name” and choose “Class” from the “Kind” drop-down. Then, click “OK” to create a stub Kotlin class.

Then, replace the generated contents with:

```
package com.commonsware.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

## POPULATING OUR RECYCLERVIEW

---

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.

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:

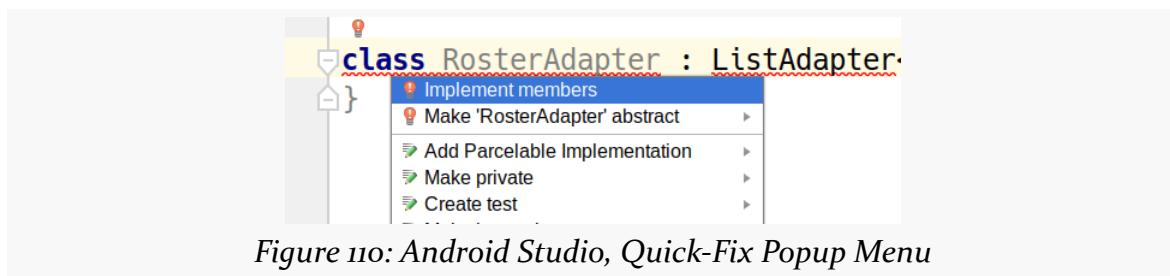


Figure 110: Android Studio, Quick-Fix Popup Menu

## POPULATING OUR RECYCLERVIEW

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

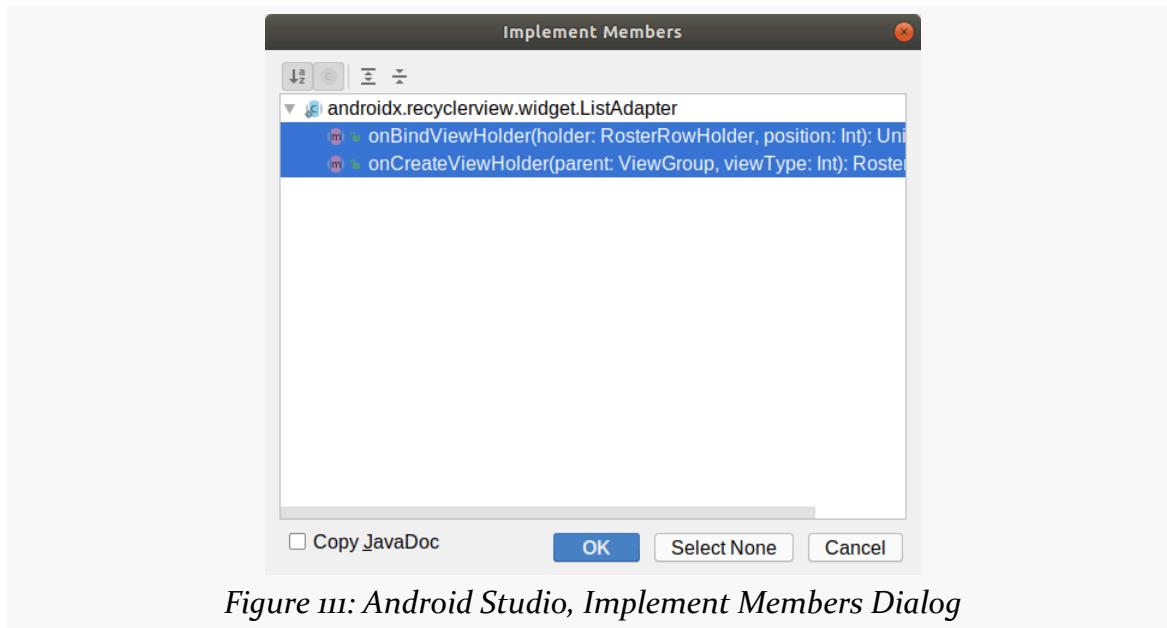


Figure 111: Android Studio, Implement Members Dialog

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 implemented") //To change body of created functions use File | Settings
        // File Templates.
    }

    override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
        TODO("not implemented") //To change body of created functions use File | Settings
        // File Templates.
    }
}
```

## POPULATING OUR RECYCLERVIEW

---

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

## Step #5: 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:

```
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 [T12-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

## POPULATING OUR RECYCLERVIEW

---

means the entire Kotlin source file 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 implemented") //To change body of created functions use File | Settings
        / File Templates.
    }

    override fun onBindViewHolder(holder: RosterRowHolder, position: Int) {
        TODO("not implemented") //To change body of created functions use File | Settings
        / File Templates.
    }
}

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
}
```

We still have two functions to implement on RosterAdapter, which we will fix later in the tutorial.

## Step #6: Adding the Data Binding

We know that we want to show ToDoModel objects in our RecyclerView rows. That means that we can add data binding expressions to our layout resource to be able to pull data from the models into our CheckBox widget.

Add the following to the XML file, after the [?xml version="1.0" encoding="utf-8"?] line (if you have one) and before the <android.support.constraint.ConstraintLayout> element:

## POPULATING OUR RECYCLERVIEW

---

```
<layout>
    <data>
        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />
    </data>
```

You will also need to add `</layout>` to the end of the file.

The `<layout>` element allows us to add some metadata to the layout resource, above and beyond the widgets that we have. The `<data>` element is where we configure metadata for data binding. The `<variable>` element is where we state that we want to bind data from a certain type of object (`com.commonsware.todo.ToDoModel`), and that we will refer to that object by the name `model` elsewhere in the resource.

On the `<CheckBox>` element, replace the existing `android:text` attribute with:

```
    android:text="@{model.description}"
```

(from [T12-RecyclerView/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

The `@{}` notation indicates that this is a data binding expression. Rather than it being some fixed value, we have a snippet of code that will be executed at runtime to fill in this attribute. Specifically, we are using the `description` property on our `ToDoModel` object to fill in the caption of the `CheckBox`.

Then, add the following attribute to the `CheckBox`:

```
    android:checked="@{model.completed}"
```

(from [T12-RecyclerView/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

The `android:checked` attribute on a `CheckBox` indicates whether or not the `CheckBox` is checked. Here, we populate it with another binding expression, using the `isCompleted` property on our `ToDoModel`. Hence, completed items will show up as checked; items not yet completed will show up as unchecked. The binding expression uses `completed` instead of `isCompleted`, as data binding tries to interpret these property references using JavaBean naming rules, so it will convert `completed` to `isCompleted` for us.

At this point, the layout resource XML should look something like this:

## POPULATING OUR RECYCLERVIEW

---

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />
    </data>

    <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">

        <CheckBox
            android:id="@+id/desc"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginBottom="8dp"
            android:layout_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:ellipsize="end"
            android:maxLines="3"
            android:text="@{model.description}"
            android:checked="@{model.completed}"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [T12-RecyclerView/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

## Step #7: Completing the Adapter

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...

## POPULATING OUR RECYCLERVIEW

---

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) {
```

Part of the data binding framework is a code generator, which generates Java classes from our layout resources. In particular, since we created a todo\_row layout resource, the code generator will generate a TodoRowBinding class. This class does a few things:

- It knows its associated layout resource and can work with a LayoutInflater to set up those widgets
- It allows us to assign values for the variables declared in the <variable> elements in the layout
- It provides easy access to named widgets within the layout from our Java code, should we need that (and we will, in future tutorials)

With that in mind, modify onCreateViewHolder() in RosterAdapter to be:

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

(from [T12-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())

However, onCreateViewHolder() will have a compile error, as we are passing a

## POPULATING OUR RECYCLERVIEW

---

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 supply that to the data binding code, to 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 [T12-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.model = model  
    binding.executePendingBindings()  
}
```

(from [T12-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

Our todo\_row layout resource had a <variable> named `model` that takes a `ToDoModel`. The code-generated `TodoRowBinding` class therefore has a property named `model`, which we can fill in using the `model` supplied to `bind()`. We also call `executePendingBindings()`, as we want the data binding framework to evaluate its binding expressions now, not sometime later.

At this point, `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) : ListAdapter<ToDoModel,  
RosterRowHolder>(DiffCallback) {  
    override fun onCreateViewHolder(  
        parent: ViewGroup,  
        viewType: Int  
    ) = RosterRowHolder(TodoRowBinding.inflate(inflater, 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 [T12-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
```

## POPULATING OUR RECYCLERVIEW

---

```
class RosterRowHolder(private val binding: TodoRowBinding) :  
    RecyclerView.ViewHolder(binding.root) {  
  
    fun bind(model: ToDoModel) {  
        binding.model = model  
        binding.executePendingBindings()  
    }  
}
```

(from [T12-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt](#))

## Step #8: Wiring Up the RecyclerView

Now, we can teach RosterListFragment to use our RosterAdapter. Right now, RosterListFragment just sets up our Toolbar:

```
package com.commonsware.todo  
  
import android.content.Intent  
import android.os.Bundle  
import android.view.LayoutInflater  
import android.view.View  
import android.view.ViewGroup  
import androidx.fragment.app.Fragment  
import kotlinx.android.synthetic.main.todo_roster.*  
  
class RosterListFragment : Fragment() {  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? = inflater.inflate(R.layout.todo_roster, container, false)  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
  
        toolbar.title = getString(R.string.app_name)  
        toolbar.inflateMenu(R.menu.actions)  
  
        toolbar.setOnMenuItemClickListener { item ->  
            when (item.itemId) {  
                R.id.about -> startActivity(Intent(activity, AboutActivity::class.java))  
                else -> return@setOnMenuItemClickListener false  
            }  
  
            true  
        }  
    }  
}
```

(from [T11-Repository/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

## POPULATING OUR RECYCLERVIEW

---

Add the following lines to the bottom of the `onViewCreated()` function in `RosterListFragment`:

```
val adapter = RosterAdapter(layoutInflater)

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

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

adapter.submitList(ToDoRepository.items)
empty.visibility = View.GONE
```

(from [T12-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 reaching into the `ToDoRepository` and referencing its `items` property
- Hide the empty widget, by setting its visibility to be `GONE`

This approach of having our fragment work directly with a repository is not great. We will be changing that later in the book, but it will suffice for now.

The overall `RosterListFragment` now should look like:

```
package com.commonsware.todo

import android.content.Intent
import android.os.Bundle
import android.view.LayoutInflater
import android.view.View
```

## POPULATING OUR RECYCLERVIEW

---

```
import android.view.ViewGroup
import androidx.fragment.app.Fragment
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import kotlinx.android.synthetic.main.todo_roster.*
import kotlinx.android.synthetic.main.todo_roster.view.*

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

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

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

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

        val adapter = RosterAdapter(layoutInflater)

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

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

        adapter.submitList(ToDoRepository.items)
        empty.visibility = View.GONE
    }
}
```

(from [T12-RecyclerView/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

## POPULATING OUR RECYCLERVIEW

---

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

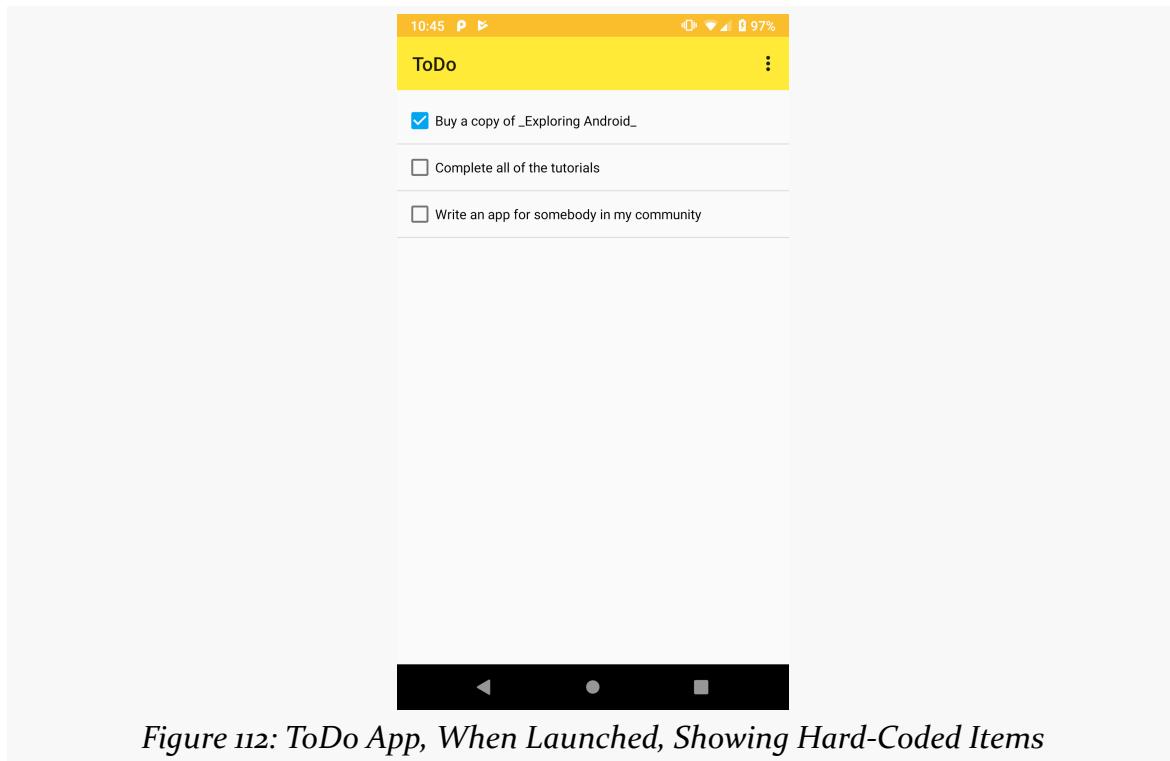


Figure 112: ToDo App, When Launched, Showing Hard-Coded Items

## 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/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](#)

Licensed solely for use by Patrocinio Rodriguez

# 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's 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: Binding Our RosterRowHolder

Somewhere along the line, we need to register an `OnCheckedChangeListener` on the `CheckBox` in each row, so we find out when that checkbox is checked and unchecked. Since we are using the data binding framework, we can let it do that registration, passing control to us when the user checks (or unchecks) the `CheckBox`.

In theory, we could have a data binding expression in the layout that directly updates the model. After all, the `ToDoModel` is what we are binding into the layout, via the `model` variable. In fact, the data binding framework has support for this, through what is known as two-way data binding. However, `ToDoModel` is immutable, so two-way data binding is not an option for us.

Instead, we can add another variable: our `RosterRowHolder`.

To do that, add another `<variable>` element to `res/layout/todo_row.xml`:

```
<variable
```

## TRACKING THE COMPLETION STATUS

---

```
name="holder"
type="com.commonsware.todo.RosterRowHolder" />
```

(from [T13-Completion/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

Then, modify bind() on RosterRowHolder to call setHolder() on the TodoRowBinding:

```
fun bind(model: ToDoModel) {
    binding.model = model
    binding.holder = this
    binding.executePendingBindings()
}
```

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

Initially, this will result in an error, as TodoRowBinding will be reflecting the older edition of your layout resource. From the Android Studio main menu, choose “Build” > “Make Module ‘app’” to update TodoRowBinding from the modified layout resource. When that is completed, the error should go away.

## Step #2: Passing the Event Up the Chain

Eventually, we can find out in our Kotlin code about our model and the fact that the user toggled its completion state. We need to modify our repository to reflect the change in the model state.

We could have the RosterRowHolder do that. After all, right now, we are working directly with the repository in RosterListFragment, and there is little that is stopping us from doing the same thing in RosterRowHolder. 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 RosterListFragment, we may as well have it handle the model modifications as well... for now. Eventually, this will move into a dedicated class for this sort of thing — a “viewmodel” — but we do not have one of those just yet.

However, our RosterRowHolder does not have access to the RosterListFragment. Instead, we need to pass the event up the Kotlin object hierarchy, from the RosterRowHolder through the RosterAdapter to the RosterListFragment.

Modify the constructor of RosterRowHolder to look like:

## TRACKING THE COMPLETION STATUS

---

```
class RosterRowHolder(  
    private val binding: TodoRowBinding,  
    val onRowClick: (ToDoModel) -> Unit  
) : RecyclerView.ViewHolder(binding.root) {
```

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

Here, `onRowClick` is a function type. We are passing some sort of function or lambda expression into `RosterRowHolder` that takes a `ToDoModel` as input and returns nothing (i.e., `Unit`, roughly analogous to `void` in Java).

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: LayoutInflator,  
    private val onRowClick: (ToDoModel) -> Unit  
) : ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
```

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

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

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

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

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

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

```
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 `onRowClick`.

## TRACKING THE COMPLETION STATUS

---

Right now, we have a `TODO()` function call in the lambda expression. In Kotlin, this will crash with an exception when called. Such a crash is a somewhat stronger reminder that we are not yet done, compared to an ordinary `// TODO` comment.

## Step #3: Binding to the Checked Event

The binding expressions that we already have in the `todo_row` layout — on `android:checked` and `android:text` — are setting standard properties of the UI. We can also add binding expressions for some event handlers, routing them to our Java code.

With that in mind, add the following attribute to the `CheckBox` in the `res/layout/todo_row.xml` resource:

```
    android:onCheckedChanged="@{(cb, isChecked) -> holder.onRowClick.invoke(model)}"
```

(from [T13-Completion/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

We are supplying a lambda expression to the binding expression, saying that we want this code to be executed when the user checks or unchecks the `CheckBox`. When the user does toggle the completion status, we:

- Get the `holder` variable
- Get the `onRowClick` property from the `holder`
- Call the function defined by the `onRowClick` property, passing our current `model` variable as the parameter

If this code were Kotlin, we could use `holder.onRowClick(model)`, treating the function type as if it were an actual function. The binding expression syntax [does not support this](#), though, so we have to use the alternative approach, calling `invoke()` to actually “invoke” the function type.

The entire `res/layout/todo_row.xml` resource should now resemble:

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />

        <variable
            name="holder"
```

## TRACKING THE COMPLETION STATUS

---

```
    type="com.commonsware.todo.RosterRowHolder" />
</data>

<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">

    <CheckBox
        android:id="@+id/desc"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:checked="@{model.completed}"
        android:ellipsize="end"
        android:maxLines="3"
        android:onCheckedChanged="@{(cb, isChecked) -> holder.onRowClick.invoke(model)}"
        android:text="@{model.description}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [T13-Completion/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

At this point, 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 #4: Saving the Change

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

Right now, though, `ToDoRepository` is read-only. The models are immutable, and our `items` property is immutable. To be able to change the list contents, we need `items` to be mutable.

## TRACKING THE COMPLETION STATUS

---

To that end, in `ToDoRepository`, make `items` be a `var` instead of a `val`. Then, 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 [T13-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.

At this point, `ToDoRepository` should resemble:

```
package com.commonsware.todo

object 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 [T13-Completion/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

Then, we can call `save()` from our `onRowClick` lambda expression, over in

## TRACKING THE COMPLETION STATUS

---

RosterListFragment:

```
val adapter = RosterAdapter(layoutInflater) { model ->
    ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
}
```

(from [T13-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.

At this point, RosterListFragment should resemble:

```
package com.commonsware.todo

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

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

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

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

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

        val adapter = RosterAdapter(layoutInflater) { model ->
            ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
        }

        view.items.apply {
```

## TRACKING THE COMPLETION STATUS

---

```
setAdapter(adapter)
layoutManager = LinearLayoutManager(context)

addItemDecoration(
    DividerItemDecoration(
        activity,
        DividerItemDecoration.VERTICAL
    )
)

adapter.submitList(ToDoRepository.items)
empty.visibility = View.GONE
}
}
```

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

If you run this revised sample... nothing much seems to change, except that our `TODO()` crash is gone. The UI itself is unaffected, and you cannot readily see the objects in the repository. 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.

## 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](#)

# Preparing for Navigation

---

We need to start adding yet more screens to our app. In particular:

- We need to show the full details of a to-do item to the user, such as the notes
- We need some form for the user to define new to-do items or to modify existing ones

Earlier, when we [set up the about screen](#), we created a second activity. That can work, but Google's vision going forward is for apps to have fewer activities and to use fragments to represent individual screens.

When we [set up RosterListFragment](#), we used a FragmentTransaction to display it. We could do that for the other screens as well. However, Jetpack contains a Navigation component that is designed for managing this sort of inter-screen flow. So, in this tutorial, we will set up the Navigation component and convert MainActivity and RosterListFragment to use it. Then, in upcoming tutorials, we will add new "destinations" to our "navigation graph" to be able to bring up additional fragment-based screens.

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 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:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.3.31'
    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.4.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()

    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

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

In particular, in the `buildscript` closure at the top, you will see:

```
ext.kotlin_version = '1.3.31'
```

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

This sets up a constant, named `kotlin_version`, that is used for multiple Kotlin dependencies. It fills the same sort of role for Kotlin that we want to use for the

## PREPARING FOR NAVIGATION

---

Navigation component.

So, add a similar line right after it:

```
ext.nav_version = '2.0.0'
```

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

We are going to use version 2.0.0 of the Navigation component, and this line sets up that version number.

After making this change, you should get a yellow 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:3.4.1'  
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"  
    // NOTE: Do not place your application dependencies here; they belong  
    // in the individual module build.gradle files  
}
```

(from [T13-Completion/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 [T14-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.

## PREPARING FOR NAVIGATION

---

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

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.3.31'
    ext.nav_version = '2.0.0'

    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.4.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()

    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}
```

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

The yellow 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. Right now, at the top of that file, you should see a few apply plugin statements:

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

## PREPARING FOR NAVIGATION

---

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

Add two more to that list:

```
apply plugin: 'kotlin-kapt'  
apply plugin: 'androidx.navigation.safeargs.kotlin'
```

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

The `kotlin-kapt` plugin allows Kotlin to work with Java annotation processors, the build-time tools that `@Interpret @ThingsThat(lookLike = "These")`. We will be starting to use annotations in our Kotlin code starting in the next tutorial.

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.

You will be tempted by the yellow 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 a 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 [T14-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.

The overall `dependencies` closure in `app/build.gradle` should now resemble:

## PREPARING FOR NAVIGATION

---

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

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

You may now go ahead and click the “Sync Now” link in the yellow banner. Conversely, if for some reason that yellow 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 (activities or 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.

## PREPARING FOR NAVIGATION

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

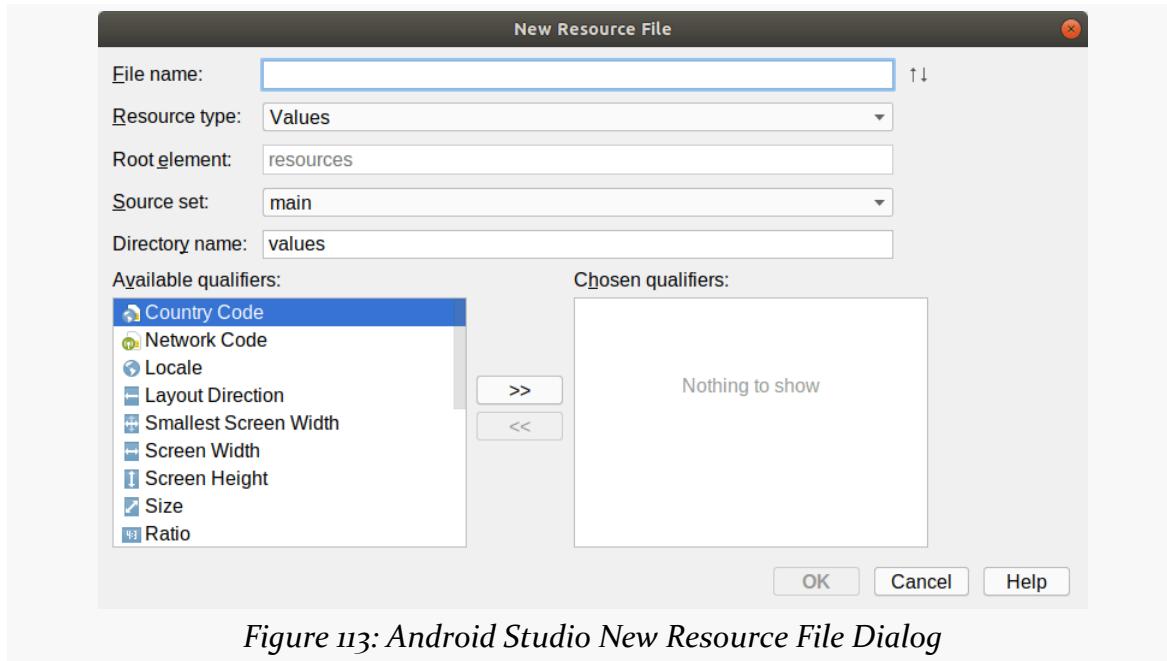


Figure 113: Android Studio New Resource File Dialog

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.

## PREPARING FOR NAVIGATION

As with layout resources, the editor for navigation resources contains two tabs. The “Text” tab has the XML, while the “Design” tab has a graphical designer:

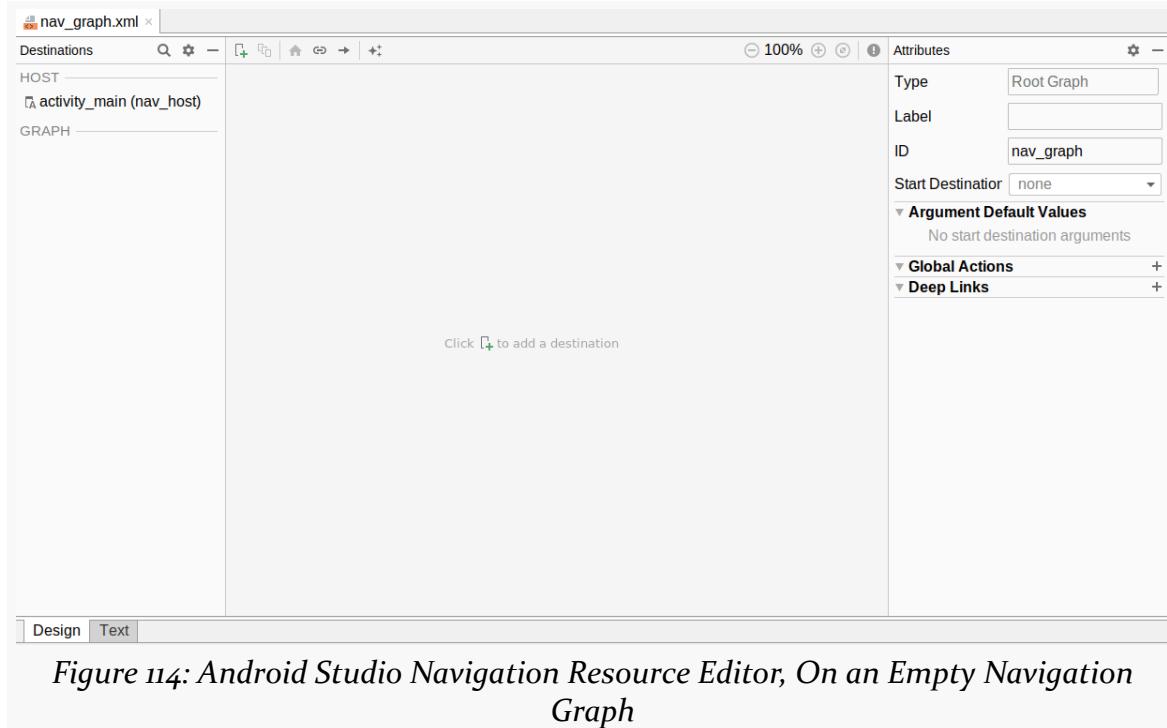


Figure 114: Android Studio Navigation Resource Editor, On an Empty Navigation Graph

If you click over to the Text tab, 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>
```

To add RosterListFragment as our one-and-only screen, in the Design sub-tab, click the toolbar button that looks like a document with a green + sign in the lower-right corner:



Figure 115: Android Studio Navigation Resource Editor Toolbar

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

## PREPARING FOR NAVIGATION

among them:

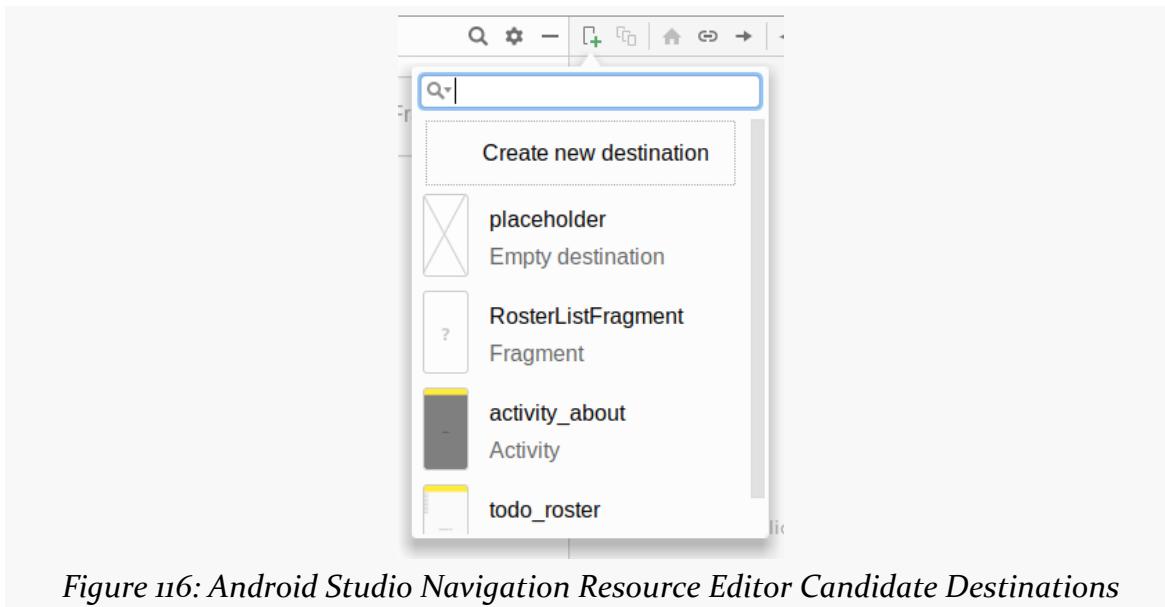


Figure 116: Android Studio Navigation Resource Editor Candidate Destinations

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

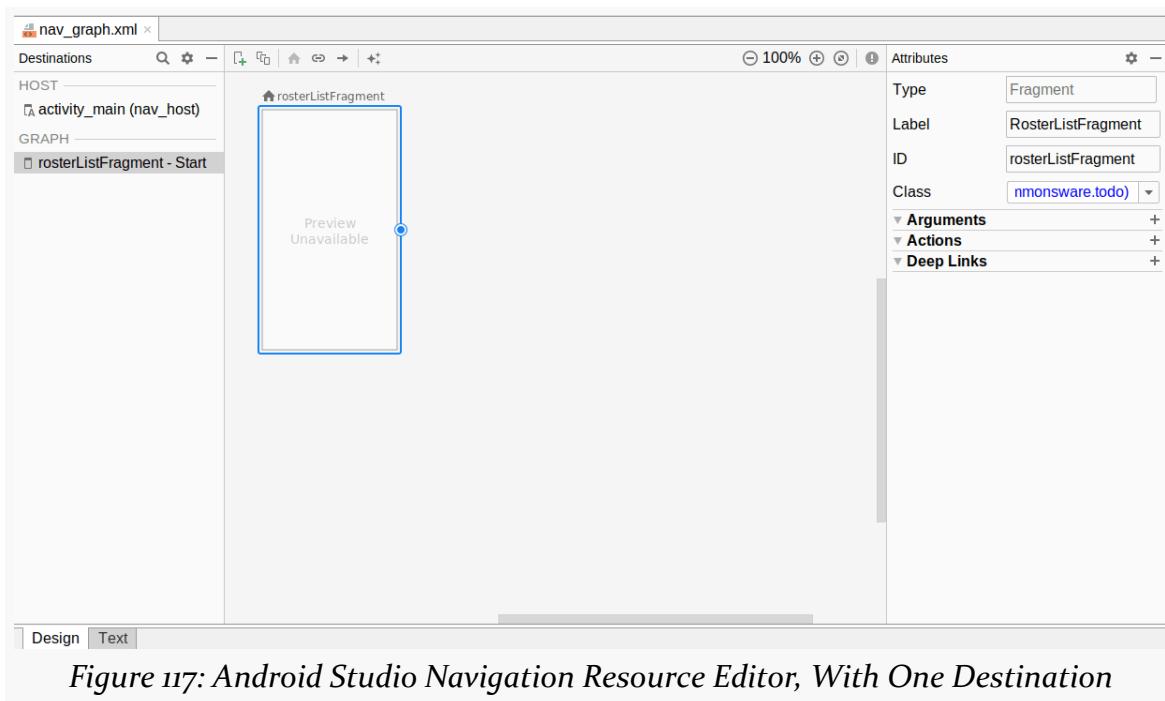


Figure 117: Android Studio Navigation Resource Editor, With One Destination

## PREPARING FOR NAVIGATION

---

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.

The navigation resource 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"
    app:startDestination="@+id/rosterListFragment">

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

(from [T14-Nav/ToDo/app/src/main/res/navigation/nav\\_graph.xml](#))

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 `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.

The Navigation component works best if your `Toolbar` is part of the activity layout, rather than having a separate `Toolbar` for each screen (e.g., our

## PREPARING FOR NAVIGATION

RosterListFragment). So, we will set up this layout resource to have a Toolbar, and we will later remove the Toolbar from our fragment.

With that in mind, drag a Toolbar from the “Containers” category of the “Palette” and drop it into the layout preview:

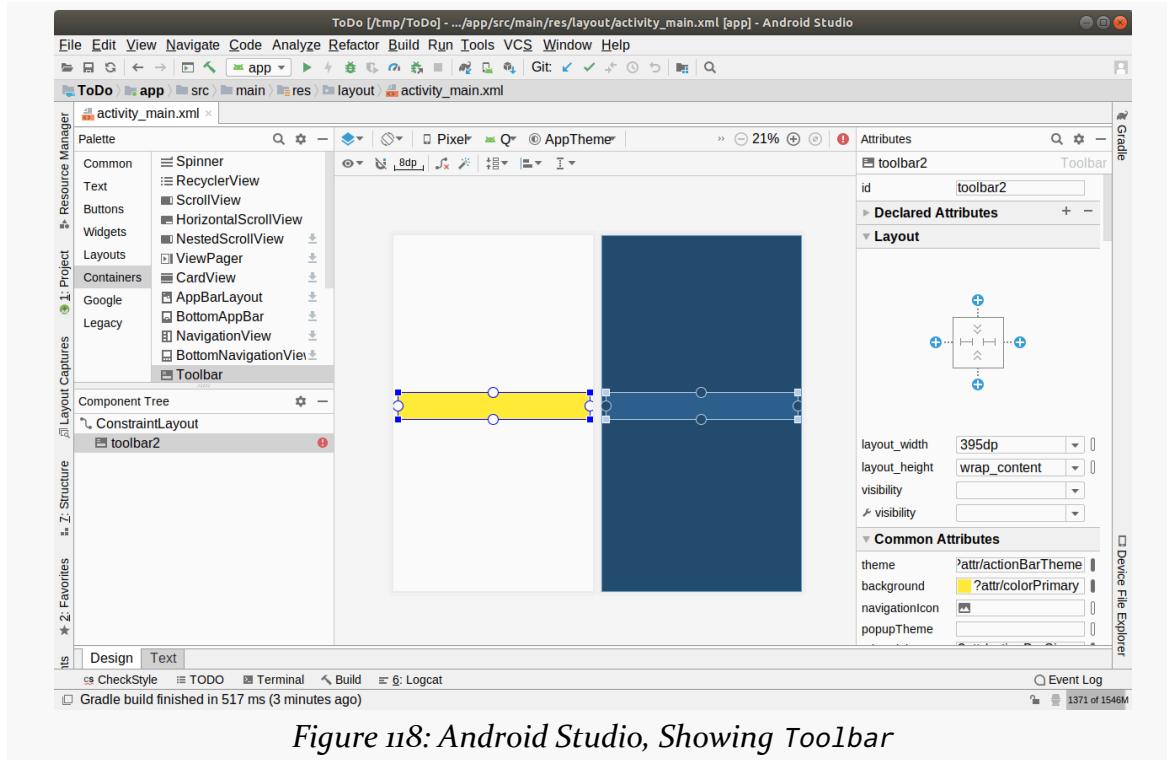


Figure 118: Android Studio, Showing Toolbar

Use the grab handles on the left, top, and right and connect them to the left, top, and right sides of the ConstraintLayout:

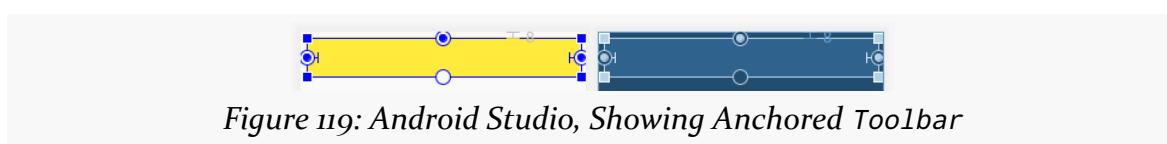


Figure 119: Android Studio, Showing Anchored Toolbar

In the “Attributes” pane:

- Set the “ID” to be toolbar
- Set the width to be match\_constraint (a.k.a., 0dp)
- Set the height to be wrap\_content
- Set the margins on the three anchored sides to be 0

## PREPARING FOR NAVIGATION

---

We also need to add a `<fragment>` element to the layout resource. As the name suggests, this places a fragment into our UI, wherever we size and position it. An `android:name` attribute will indicate what fragment class we want.

Unfortunately, [the drag-and-drop GUI builder does not handle `<fragment>` very well](#). As a result, we need to add this element by hand, directly in the XML.

Over in the “Text” sub-tab, add in this XML element as a child of the `ConstraintLayout` element:

```
<fragment
    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" />
```

(from [T14-Nav/ToDo/app/src/main/res/layout/activity\\_main.xml](#))

This is a “static fragment”. 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.

The overall layout at this point 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="match_parent">

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:background="?attr/colorPrimary"
```

## PREPARING FOR NAVIGATION

---

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

<fragment
    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.constraintlayout.widget.ConstraintLayout>
```

(from [T14-Nav/ToDo/app/src/main/res/layout/activity\\_main.xml](#))

## 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)
}
```

## PREPARING FOR NAVIGATION

---

If you now run the app, it almost works:

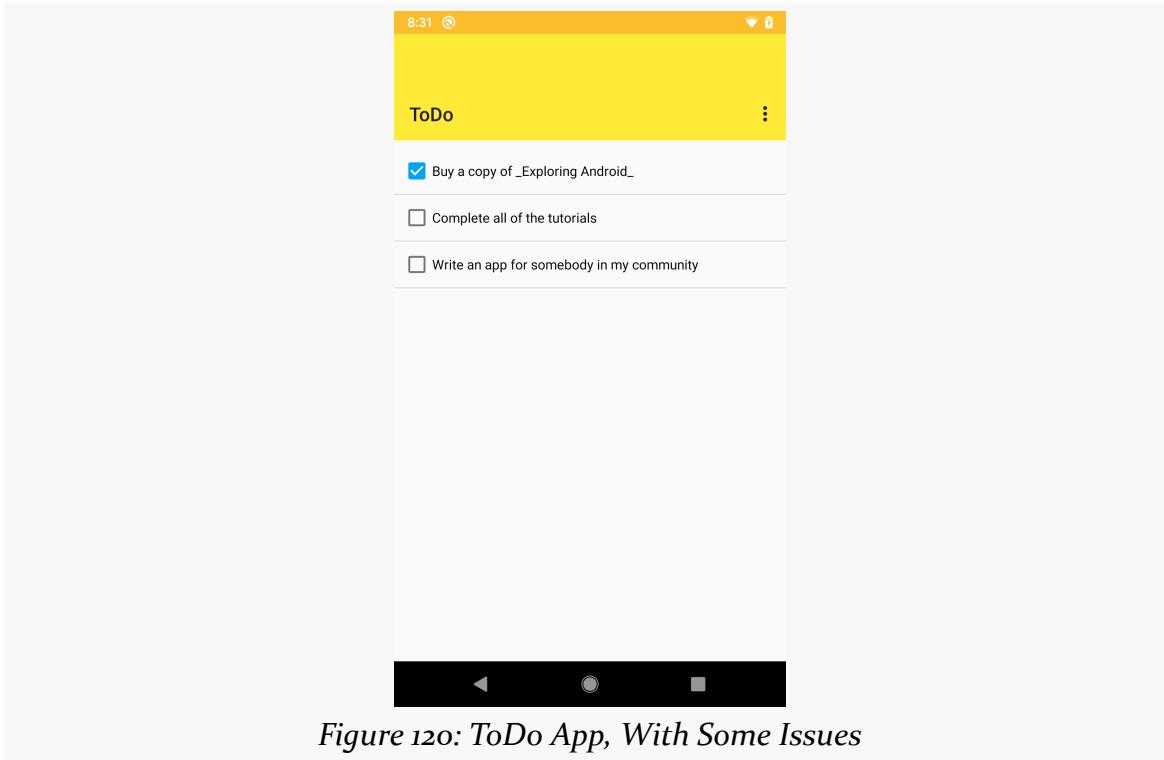


Figure 120: ToDo App, With Some Issues

Of note, our `RosterListFragment` is still being displayed, despite the fact that we just got rid of all the fragment stuff from our activity. That is because the activity's layout has `NavHostFragment`, and we taught `NavHostFragment` to use our `nav_graph` navigation resource. That resource knows about `RosterListFragment` and identifies it as the “start destination”, so `NavHostFragment` creates and displays `RosterListFragment` for us.

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.

## Step #8: Addressing the Toolbar

However, our Toolbar situation is a bit of a mess. Among other things, we have two of them, which is twice as many as we need. So, we should spend a few minutes cleaning that up. Plus, our about screen ought to be available from anywhere in the app, and right now our code for dealing with it is tied to `RosterListFragment`, so we should move that into `MainActivity`.

Start by adding this line to `onCreate()` in `MainActivity`, after the `setContentView()` call:

```
setSupportActionBar(toolbar)
```

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

This tells our activity to treat our Toolbar as the “action bar” for our activity. This is a way that we can allow our other fragments, like `RosterListFragment`, to contribute things to the Toolbar without them having direct access to the Toolbar widget.

Then, add these two functions to `MainActivity`:

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

    return super.onCreateOptionsMenu(menu)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    if (item.itemId == R.id.about) {
        startActivity(Intent(this, AboutActivity::class.java))
        return true
    }

    return super.onOptionsItemSelected(item)
}
```

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

`onCreateOptionsMenu()` will be called so that we can add items to the action bar. Here, we use a `MenuInflater` (obtained via the `menuInflater` property on our activity) and inflate our already-existing actions menu resource. We then chain to the superclass — this is a common pattern with the action bar, so that if we elect to add some superclass between `MainActivity` and `AppCompatActivity`, it too can participate in the action bar.

## PREPARING FOR NAVIGATION

---

`onOptionsItemSelected()` will be called if the user taps on one of our items from `onCreateOptionsMenu()`. Here, we start `AboutActivity` if the user taps on our about menu item. If we do not recognize the menu item, we chain to the superclass, so any items the superclass added in `onCreateOptionsMenu()` can be handled.

Then, delete these lines from `onViewCreated()` in `RosterListFragment`:

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

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

    true
}
```

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

We will be removing the Toolbar, so we need to remove our code that refers to the Toolbar. Plus, most of that stuff was replaced by the `onCreateOptionsMenu()` and `onOptionsItemSelected()` functions that we just added to `MainActivity`.

Then, in the `todo_roster` layout resource, in the “Text” tab, replace `app:layout_constraintTop_toBottomOf="@+id/toolbar"` in both the `empty` and `items` widgets to be `app:layout_constraintTop_toTopOf="parent"`. Also, remove the `android:layout_marginTop="8dp"` attributes from those two widgets. After that, you can remove the `<Toolbar>` element, leaving you with:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

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

    <androidx.recyclerview.widget.RecyclerView
```

## PREPARING FOR NAVIGATION

---

```
    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 [T14-Nav/ToDo/app/src/main/res/layout/todo\\_roster.xml](#))

If you run the app, it works as it did before. We added no new functionality in this tutorial, but we did set up a better foundation for adding new functionality through the rest of this book.

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

Licensed solely for use by Patrocinio Rodriguez

# 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”. Click “OK” 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:

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

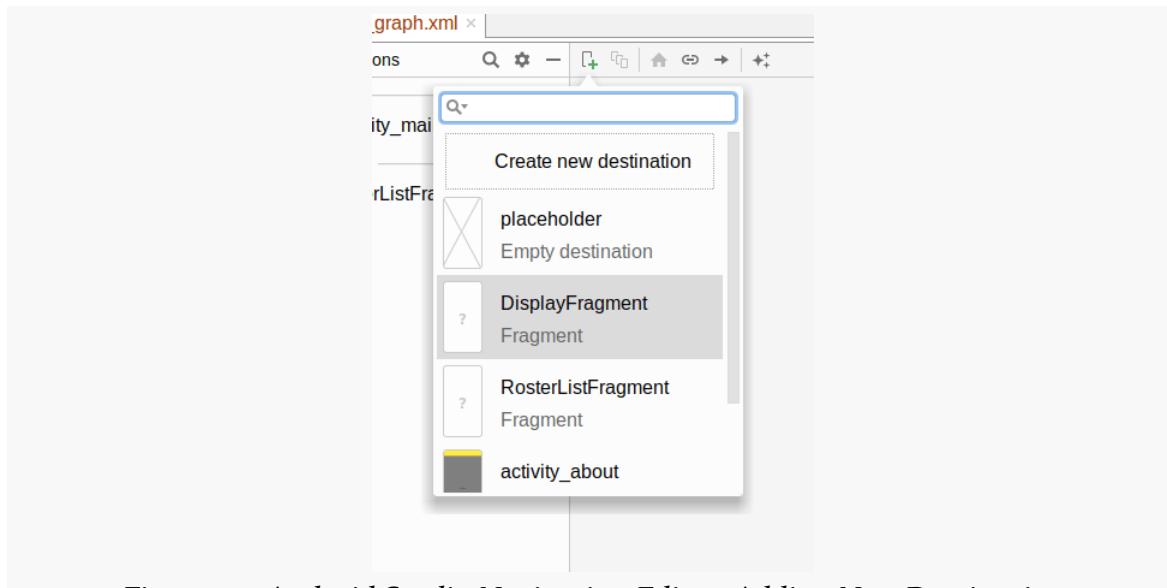
```
import androidx.fragment.app.Fragment

class DisplayFragment : Fragment() {
```

## Step #2: Updating the Navigation Graph

We need to add this new fragment to our navigation graph.

Open up the `res/navigation/nav_graph.xml` resource. In the “Design” sub-tab, 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 121: Android Studio Navigation Editor, Adding New Destination*

## DISPLAYING AN ITEM

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

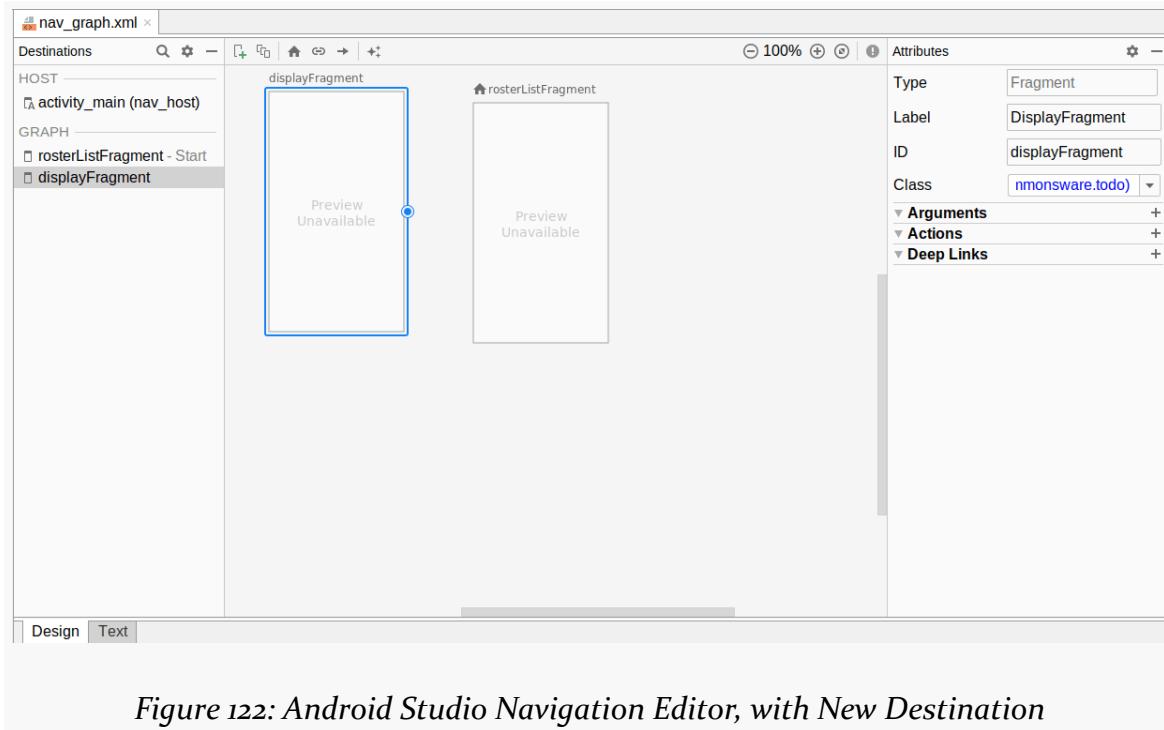


Figure 122: Android Studio Navigation Editor, with New Destination

## DISPLAYING AN ITEM

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

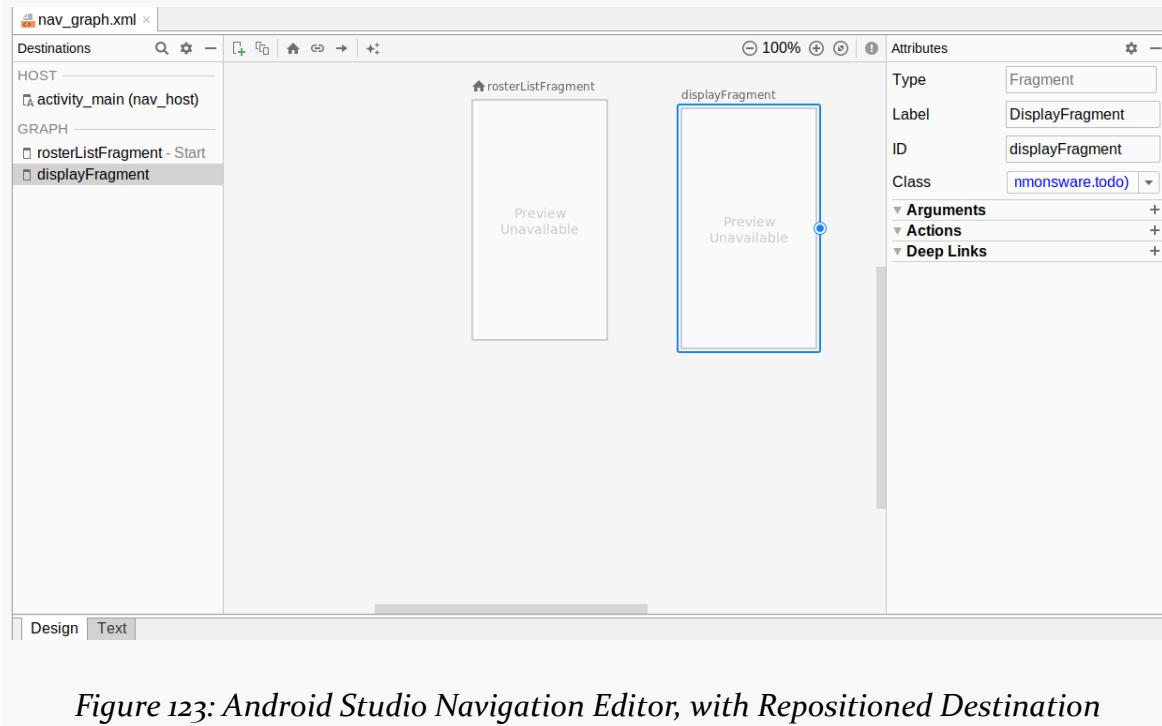
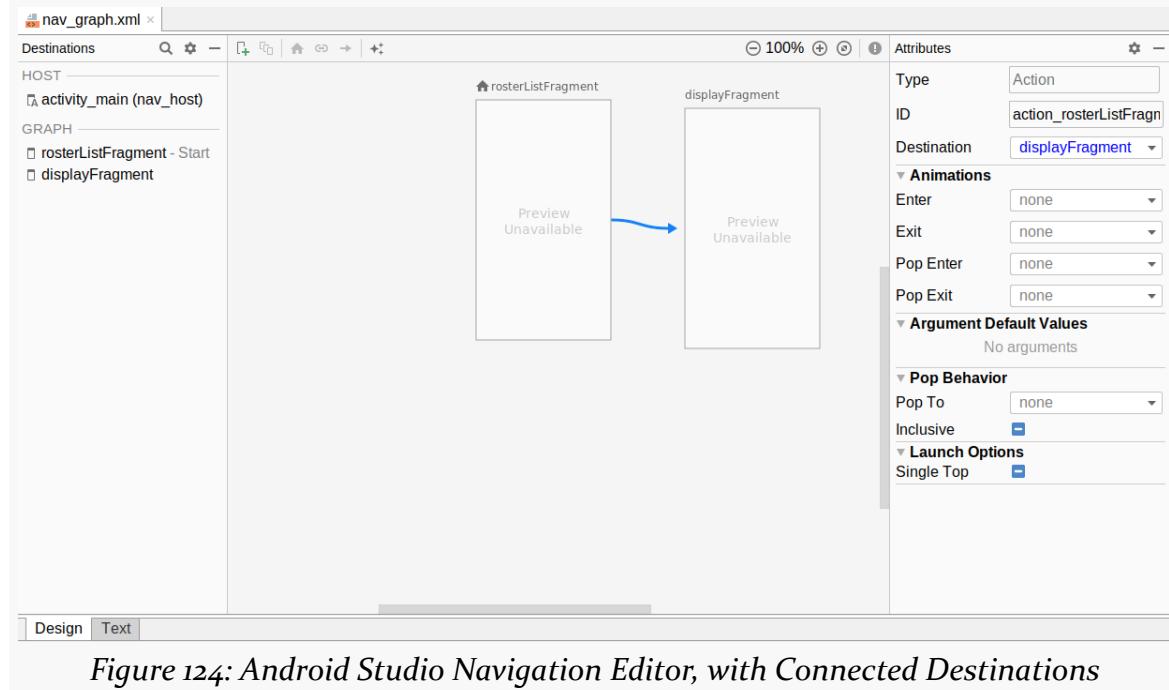


Figure 123: Android Studio Navigation Editor, with Repositioned Destination

## DISPLAYING AN ITEM

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



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"
    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" />
</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.

Right now, though, that will not work in our app.

If you run the app, any you try tapping anywhere on the row, the CheckBox state toggles between checked and unchecked. That is because our CheckBox completely fills the row, and a CheckBox interprets a click anywhere as a trigger to toggle the CheckBox state.

This is a problem.

There is no good way to tell CheckBox to only toggle the state when the user clicks on the box itself, not the text. We do not even have a good way to detect if the user clicked the text instead of the box.

So, we need to reorganize our list row a little bit, using a CheckBox without any text, plus a TextView for the label. Then we can find out when the user clicks on that label, and use that as our trigger to go display that to-do item.

## DISPLAYING AN ITEM

---

Open res/layout/todo\_row.xml in Android Studio. The layout as it stands should resemble:

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />

        <variable
            name="holder"
            type="com.commonsware.todo.RosterRowHolder" />
    </data>

    <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">

        <CheckBox
            android:id="@+id/desc"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginBottom="8dp"
            android:layout_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:checked="@{model.completed}"
            android:ellipsize="end"
            android:maxLines="3"
            android:onCheckedChanged="@{(cb, isChecked) -> holder.onRowClick.invoke(model)}"
            android:text="@{model.description}"
            app:layout_constraintBottom_toBottomOf="parent"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [T14-Nav/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

## DISPLAYING AN ITEM

The CheckBox is set to a width of 0dp, so it fills the screen. Either in the XML or in the Attributes pane, modify the android:layout\_width of the CheckBox to be wrap\_content, and eliminate the constraint tying the end of the CheckBox to the end of its parent (i.e., remove app:layout\_constraintEnd\_toEndOf="parent", or click the blue dot on the end side of the CheckBox). Also, change the ID of the CheckBox to be checkbox (android:id="@+id/checkbox").

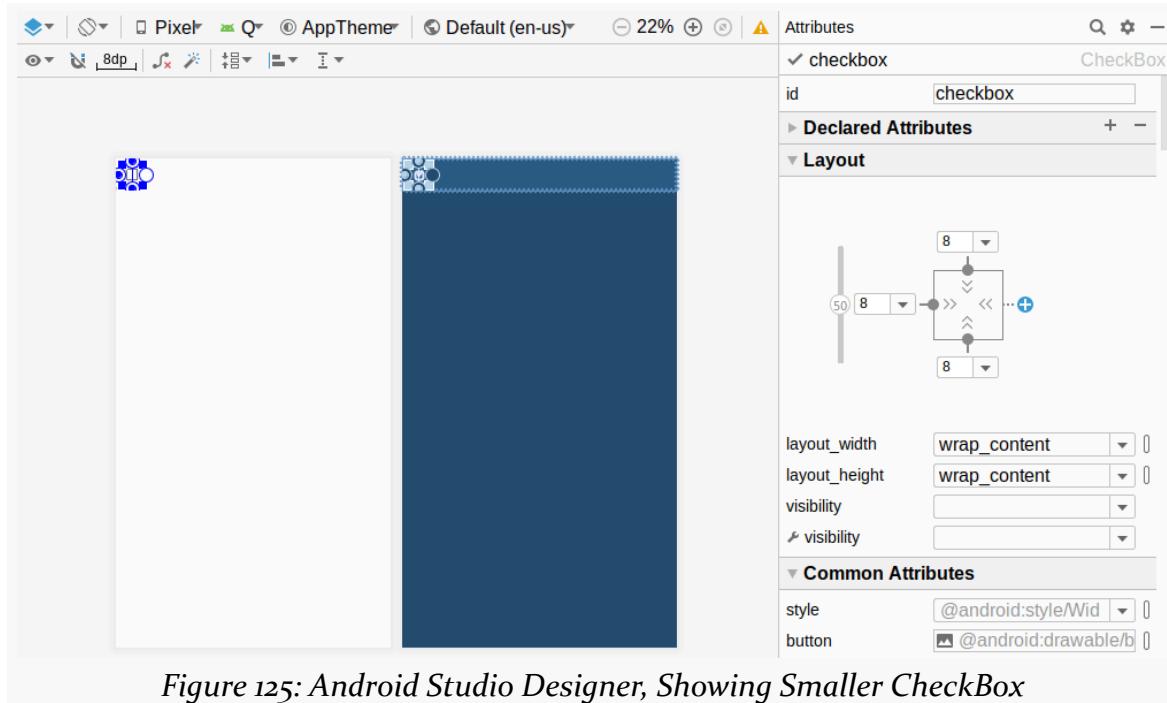


Figure 125: Android Studio Designer, Showing Smaller CheckBox

Then, add a TextView widget to the ConstraintLayout. Using the drag-and-drop editor, add a constraint from the starting edge of the TextView to the end of the CheckBox, and add a constraint from the ending edge of the TextView to the end of its parent:

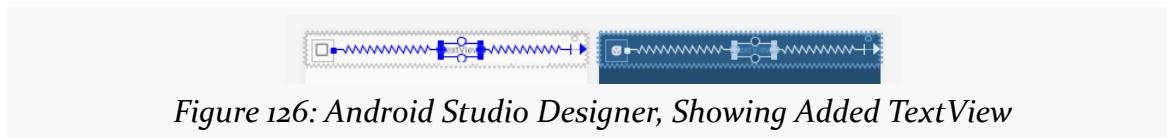


Figure 126: Android Studio Designer, Showing Added TextView

## DISPLAYING AN ITEM

---

Then, set the layout\_width to be match\_constraint (a.k.a., 0dp):

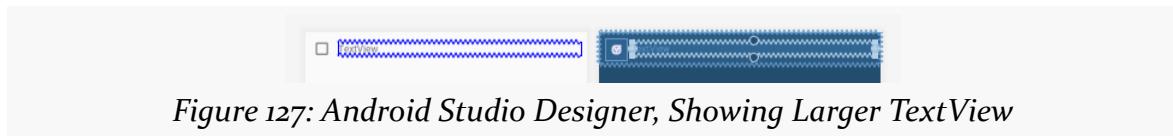


Figure 127: Android Studio Designer, Showing Larger TextView

Then, create constraints to tie the TextView to the top and bottom of its parent:

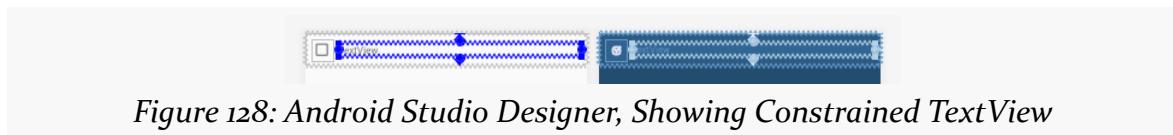


Figure 128: Android Studio Designer, Showing Constrained TextView

Give the TextView an ID of desc (`android:id="@+id/desc"`).

Then, move the `android:text="@{model.description()}"` attribute from the CheckBox to the TextView, so our description shows up on the TextView. Do the same for the `android:ellipsize` and `android:maxLines` attributes.

This should leave you with a layout that resembles:

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.TodoModel" />

        <variable
            name="holder"
            type="com.commonsware.todo.RosterRowHolder" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <CheckBox
            android:id="@+id/checkbox"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginBottom="8dp"
            android:text="@{model.description()}" />

        <TextView
            android:id="@+id/desc"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginTop="8dp"
            android:layout_marginBottom="8dp"
            android:ellipsize="end"
            android:maxLines="2"
            android:text="@{model.description()}" />
    </ConstraintLayout>
</layout>
```

## DISPLAYING AN ITEM

---

```
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:checked="@{model.completed}"
        android:onCheckedChanged="@{(cb, isChecked) -> holder.onClick.invoke(model)}"
        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_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:ellipsize="end"
    android:maxLines="3"
    android:text="@{model.description}"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toEndOf="@+id/checkbox"
    app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

If you run the app, you should see that the rows look more or less as they did before, but nothing happens when you click on the text. The CheckBox state only toggles if you click on the box.

## Step #4: Displaying the (Empty) Fragment

Now that we can get control when the user taps on a row, we need to show that `DisplayFragment`... even if at the moment it will be empty.

We now 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 `onClick` parameters:

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

## DISPLAYING AN ITEM

---

```
) : RecyclerView.ViewHolder(binding.root) {  
    (from T15-Display/ToDo/app/src/main/java/com/commonsware/todo/RosterRowHolder.kt)
```

Next, in `res/layout/todo_row.xml`, change the `android:onCheckedChanged` binding expression in our `CheckBox` to refer to `onCheckboxToggle` instead of `onRowClick`:

```
    android:onCheckedChanged="@{(cb, isChecked) -> holder.onCheckboxToggle.invoke(model)}"  
    (from T15-Display/ToDo/app/src/main/res/layout/todo_row.xml)
```

Then, add this attribute to the `ConstraintLayout`:

```
    android:onClickListener="@{(v) -> holder.onRowClick.invoke(model)}"  
    (from T15-Display/ToDo/app/src/main/res/layout/todo_row.xml)
```

Click events that are not handled by child widgets ripple up the view hierarchy. So, by putting a click listener on the `ConstraintLayout`, 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.

The overall `todo_row` layout should now resemble:

```
<?xml version="1.0" encoding="utf-8"?>  
<layout xmlns:android="http://schemas.android.com/apk/res/android"  
        xmlns:app="http://schemas.android.com/apk/res-auto">  
  
    <data>  
  
        <variable  
            name="model"  
            type="com.commonsware.todo.ToDoModel" />  
  
        <variable  
            name="holder"  
            type="com.commonsware.todo.RosterRowHolder" />  
    </data>  
  
    <androidx.constraintlayout.widget.ConstraintLayout  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:onClickListener="@{(v) -> holder.onRowClick.invoke(model)}">  
  
        <CheckBox  
            android:id="@+id/checkbox"  
            android:layout_width="wrap_content"  
            android:layout_height="wrap_content"  
            android:layout_marginBottom="8dp"  
            android:layout_marginStart="8dp"  
            android:layout_marginTop="8dp"
```

## DISPLAYING AN ITEM

---

```
        android:checked="@{model.completed}"
        android:onCheckedChanged="@{(cb, isChecked) -> holder.onCheckboxToggle.invoke(model)}"
        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_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:ellipsize="end"
        android:maxLines="3"
        android:text="@{model.description}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toEndOf="@+id/checkbox"
        app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_row.xml](#))

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: LayoutInflator,
    private val onCheckboxToggle: (ToDoModel) -> Unit,
    private val onRowClick: (ToDoModel) -> Unit
) : ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
```

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

Overall, this should resemble:

```
package com.commonsware.todo
import android.view.LayoutInflater
```

## DISPLAYING AN ITEM

---

```
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: LayoutInflator,
    private val onCheckboxToggle: (ToDoModel) -> Unit,
    private val onRowClick: (ToDoModel) -> Unit
) : ListAdapter<ToDoModel, RosterRowHolder>(DiffCallback) {
    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ) =
        RosterRowHolder(TodoRowBinding.inflate(inflater, 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 [T15-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`. So, modify the `RosterAdapter` constructor call in `onViewCreated()` of `RosterListFragment` to look like this:

```
val adapter =
    RosterAdapter(
        inflater = layoutInflater,
        onCheckboxToggle = { model ->
            ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
        },
        onRowClick = { }
```

Here, to help us keep track of what lambda expression is for what operation, we are taking advantage of Kotlin's named arguments. We have our original lambda expression assigned to `onCheckboxToggle`, and we have an empty lambda expression for `onRowClick`. At this point, if you run the app, it should work as it did before.

Of course, it would be nice if we actually did something in `onRowClick`.

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.

Finally, modify the `onRowClick` parameter of our `RosterAdapter` constructor call to pass control to this `display()` function:

```
val adapter =
    RosterAdapter(
        inflater = layoutInflater,
        onCheckboxToggle = { model ->
            ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
        },
        onRowClick = { model -> display(model) })
```

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

If you run the sample app now, and you click on one of the to-do items, you will be

## DISPLAYING AN ITEM

---

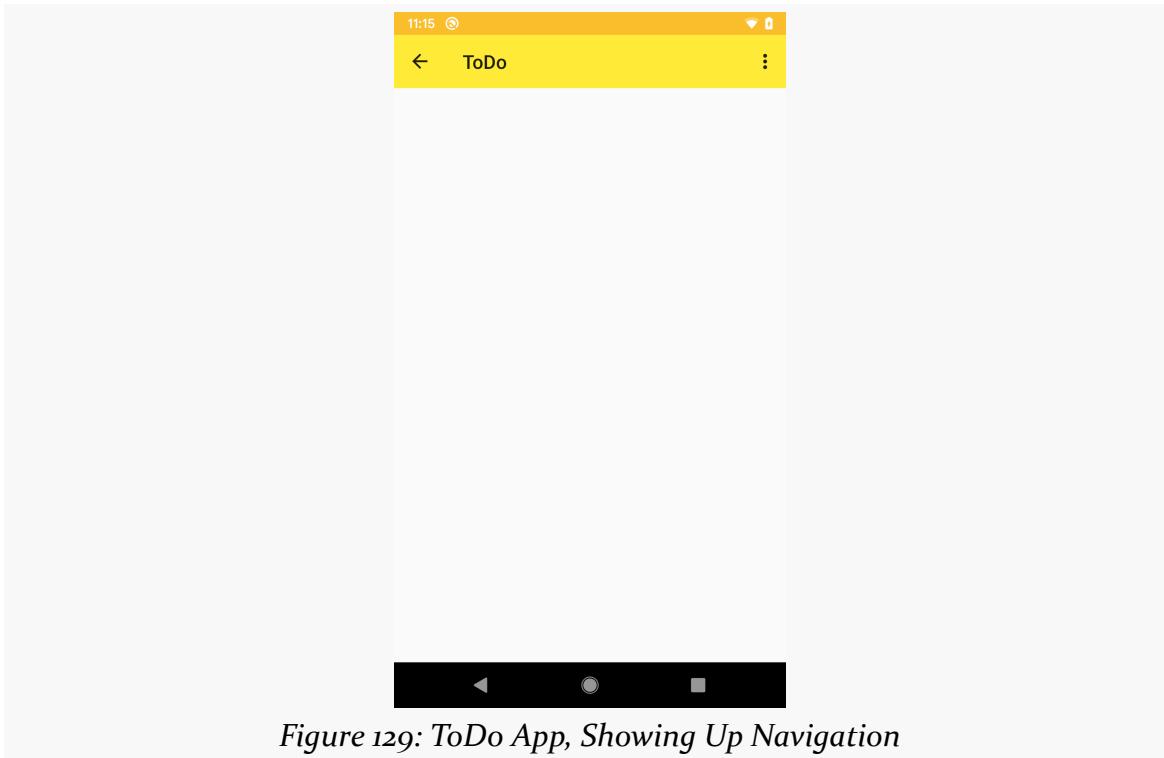
taken to the `DisplayFragment`... which happens to not display anything. We will fix that in the upcoming steps of this tutorial.

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

## Step #5: Teaching Navigation About the Action 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 129: ToDo App, 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 action bar and can add the up navigation

## DISPLAYING AN ITEM

---

icon when needed.

First, in `MainActivity`, add this property:

```
private lateinit var appBarConfiguration: AppBarConfiguration
```

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

An `AppBarConfiguration` is... a configuration for our app bar. In this app, our action bar 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`:

```
findNavController(R.id.nav_host).let { nav ->
    appBarConfiguration = AppBarConfiguration(nav.graph)
    setupActionBarWithNavController(nav, appBarConfiguration)
```

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

In a fragment, when we call `findNavController()`, it knows automatically where to find the `NavHostFragment` — it will be the parent fragment of the child on which we called `findNavController()`. That is not the case in the activity, so we have to pass in the widget ID that we assigned to our `NavHostFragment` (`R.id.nav_host`). 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 action 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 [T15-Display/ToDo/app/src/main/java/com/commonsware/todo/MainActivity.kt](#))

When the user taps that arrow in the action 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.

## DISPLAYING AN ITEM

---

If you run the app, not only will that arrow appear in the action bar when we are viewing the `DisplayFragment`, but tapping it will return you to the `RosterListFragment`

`MainActivity` should now resemble:

```
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.ui.AppBarConfiguration
import androidx.navigation.ui.NavigationUI.navigateUp
import androidx.navigation.ui.setupActionBarWithNavController
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
    private lateinit var appBarConfiguration: AppBarConfiguration

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

        setSupportActionBar(toolbar)

        findNavController(R.id.nav_host).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)
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        if (item.itemId == R.id.about) {
            startActivity(Intent(this, AboutActivity::class.java))
            return true
        }

        return super.onOptionsItemSelected(item)
    }
}
```

## DISPLAYING AN ITEM

---

```
}

override fun onSupportNavigateUp() =
    navigateUp(findNavController(R.id.nav_host), appBarConfiguration)
}
```

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

## Step #6: 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.

## Step #7: Setting Up Data Binding

As with the roster rows, we are going to use data binding to populate the widgets. That means that we need to adjust the layout to contain the `<layout>` root tag and a `<data>` element for our variables. Since we know that we are going to show a `ToDoModel`, we can go ahead and set up that variable as well.

In the Text sub-tab of the `todo_display` editor, modify the layout to look like this:

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent">

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

```
</layout>
```

This sets up a `model` variable, the same as we used in the roster rows.

## Step #8: 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 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 action bar item [in an earlier tutorial](#).

## DISPLAYING AN ITEM

---

There, click the “Clip Art” button and search for check:

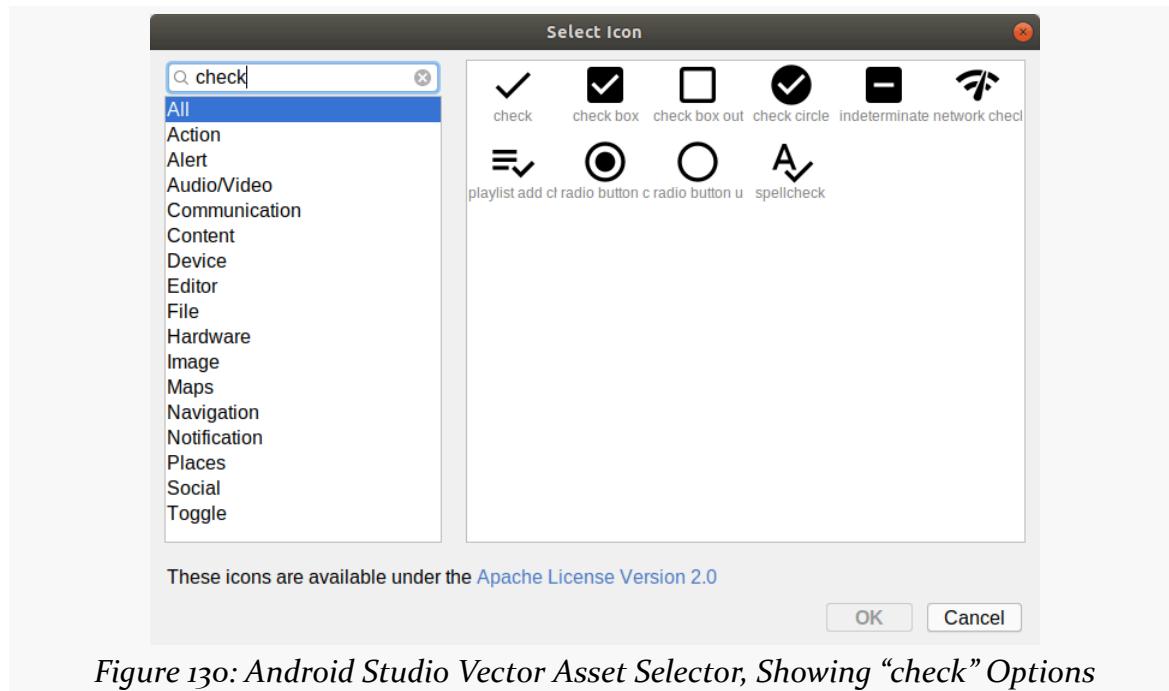


Figure 130: Android Studio Vector Asset Selector, Showing “check” Options

Choose the “check circle” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

## DISPLAYING AN ITEM

---

Then, switch over to the Design sub-tab in the layout editor and drag an ImageView from the “Widgets” category of the “Palette” into the layout. This immediately displays a dialog from which you can choose a resource to display:

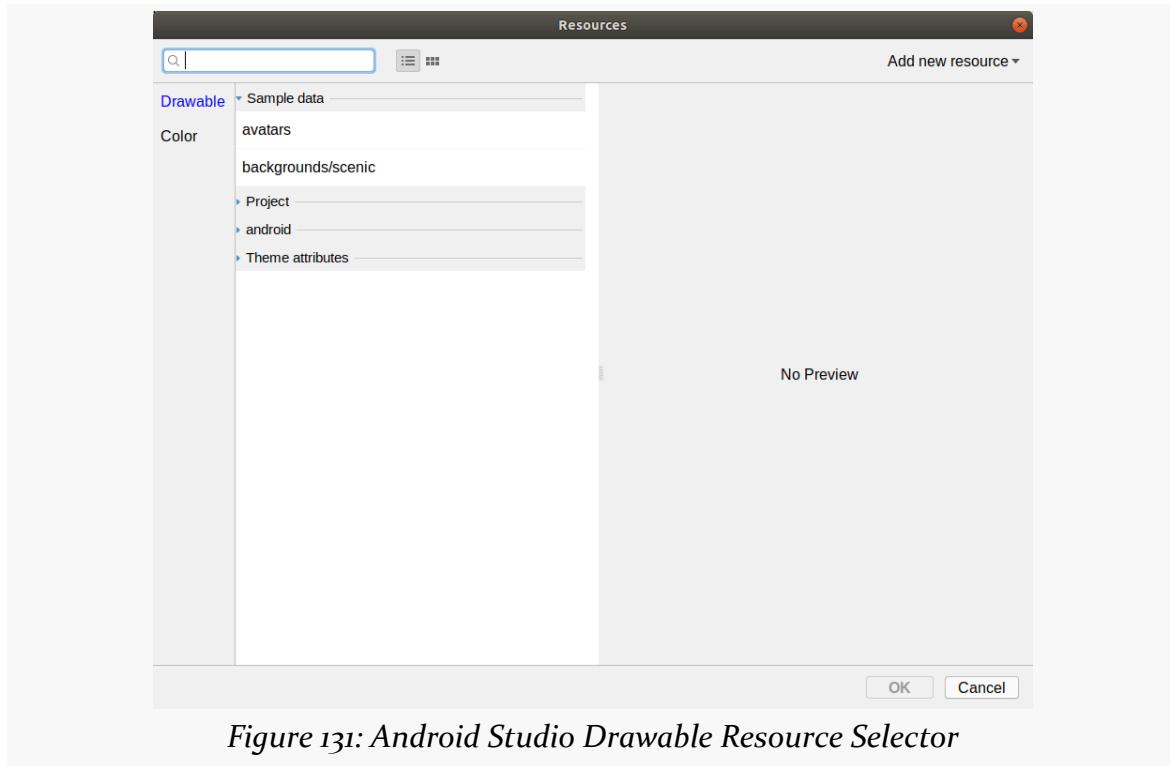


Figure 131: Android Studio Drawable Resource Selector

Open the “Project” category and choose the `ic_check_circle_black_24dp` 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:



Figure 132: Android Studio Layout Designer, Showing Tiny ImageView

In the Attributes pane, give the ImageView an “ID” of `completed`.

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

## DISPLAYING AN ITEM

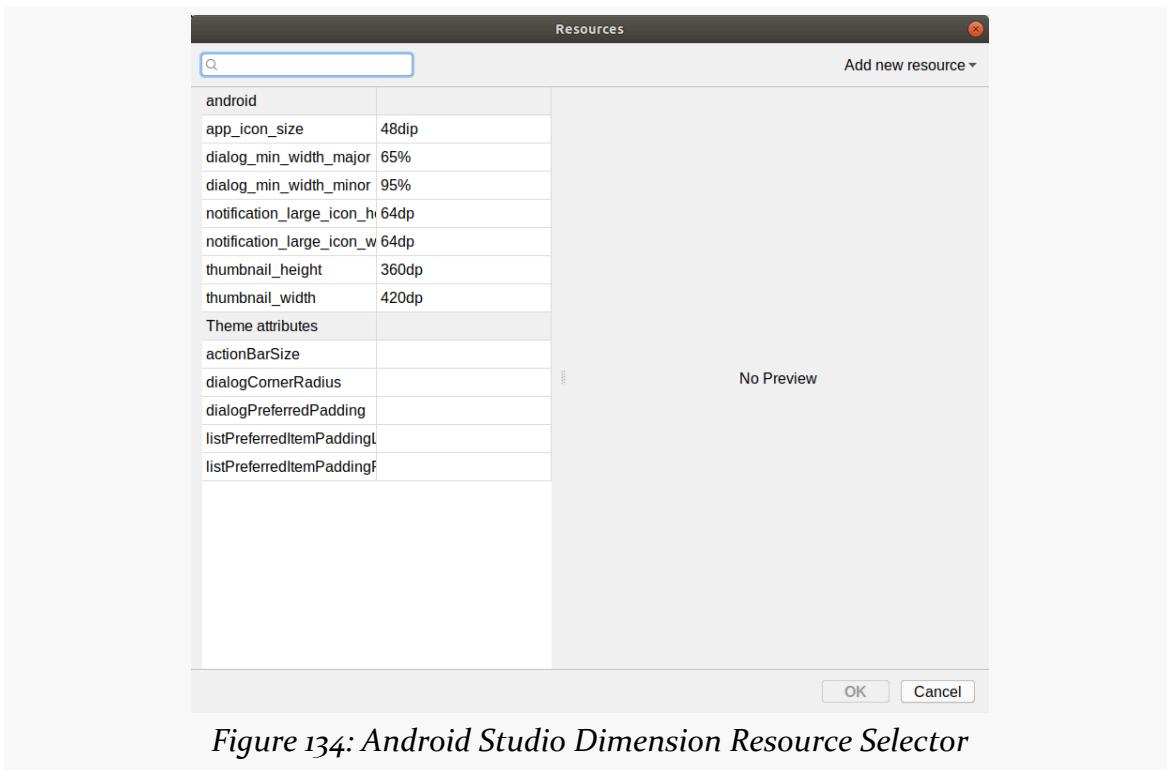
---

larger size, then apply it to both the width and the height.

To do that, click the “O” button to the right of the “layout\_width” drop-down in the Attributes pane:



That brings up a dialog to choose a dimension resource:



## DISPLAYING AN ITEM

---

From the drop-down in the corner, choose “Add new resource” > “New dimension Value...”, to bring up the new-resource dialog:

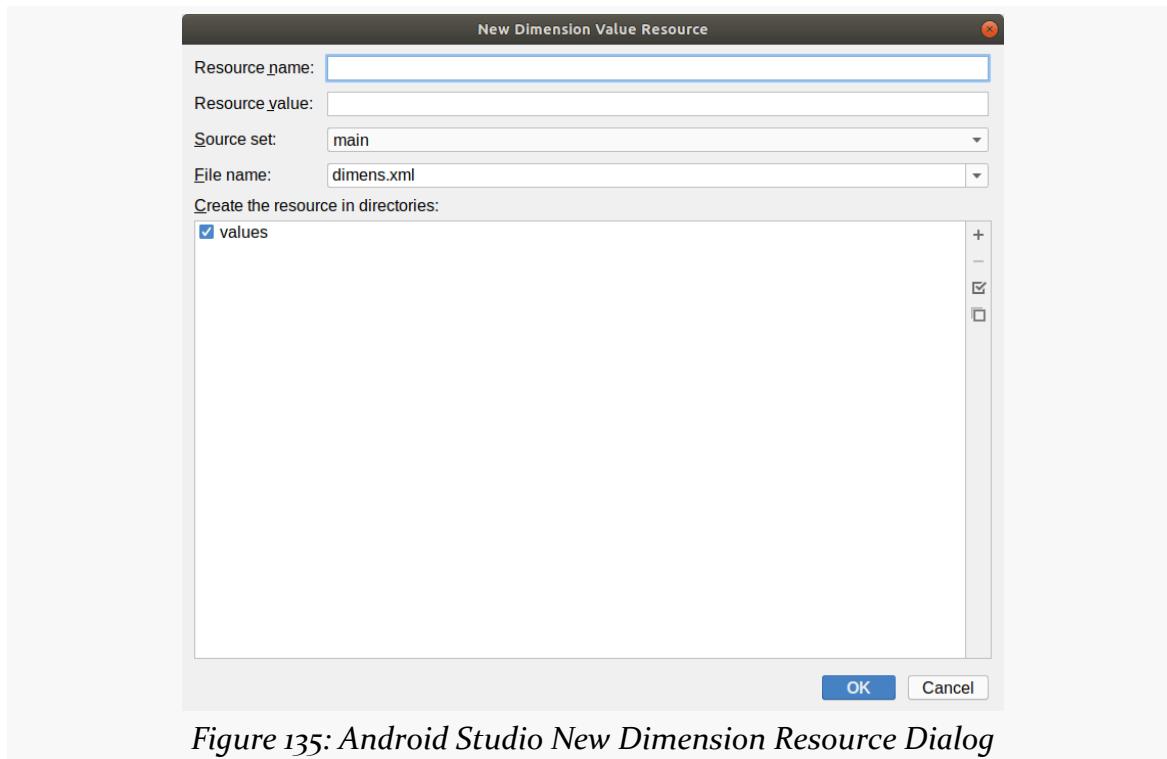


Figure 135: Android Studio New Dimension Resource Dialog

## 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`:

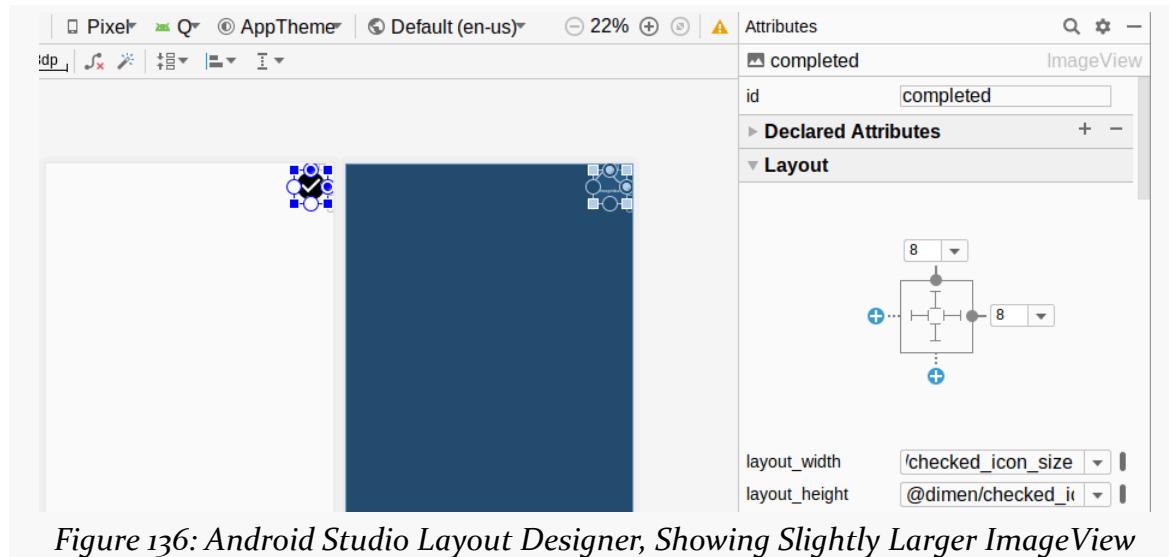


Figure 136: Android Studio Layout Designer, Showing Slightly Larger ImageView

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, click the “O” button next to the “contentDescription” field, to bring up a string resource chooser. From the drop-down in the corner, choose “Add new resource” > “New string Value...”. 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 black. 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, fold open the “All Attributes” section of the “Attributes” pane, then find the “tint” attribute. Click the “O” at the end of that row to bring up a color resource chooser. Open the “Project” category, then double-click on the colorAccent resource, to close up the dialog and apply that tint to the icon:

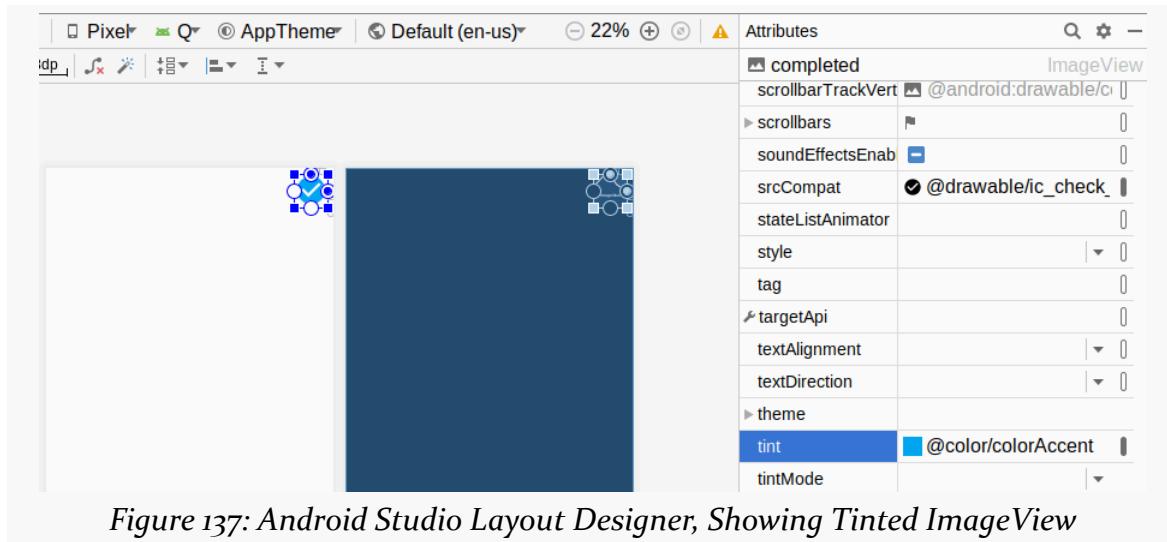


Figure 137: Android Studio Layout Designer, Showing Tinted ImageView

Finally, we only want to show this icon when the to-do item is completed. Otherwise, it should not be visible. To do that, switch over to the “Text” sub-tab of the editor to work with the XML. Add the following attribute to the ImageView:

```
    android:visibility="@{model.completed ? View.VISIBLE : View.GONE}"
```

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_display.xml](#))

This is a binding expression that uses a Java-style ternary expression. We check to see if the ToDoModel is completed — completed in the binding expression language will map to the `isCompleted()` getter method. If it is true, we assign `View.VISIBLE` to the visibility, otherwise we assign `View.GONE`. This implements what we want: show the icon for completed items, and hide it for others.

However, in order to be able to reference View constants, we need to add an `<import>` element to the `<data>` element:

```
<data>  
  
<variable  
    name="viewVisible" type="View" />
```

## DISPLAYING AN ITEM

---

```
    name="model"
    type="com.commonsware.todo.ToDoModel" />

<import type="android.view.View" />
</data>
```

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_display.xml](#))

This works like a Kotlin import statement, indicating where the View symbol comes from.

Your layout XML should now resemble:

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />

        <import type="android.view.View" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        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_marginEnd="8dp"
            android:layout_marginTop="8dp"
            android:contentDescription="@string/is_completed"
            android:tint="@color/colorAccent"
            android:visibility="@{model.completed ? View.VISIBLE : View.GONE}"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:srcCompat="@drawable/ic_check_circle_black_24dp" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

## Step #9: Displaying the Description

Next is the description of the to-do item.

Switch back to the Design sub-tab of the layout resource editor. From the “Text” 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 138: Android Studio Layout Designer, Showing Added `TextView`

Change the “`layout_width`” attribute to `match_constraint` (a.k.a., `0dp`):

Also, give the widget an “ID” of `desc`.

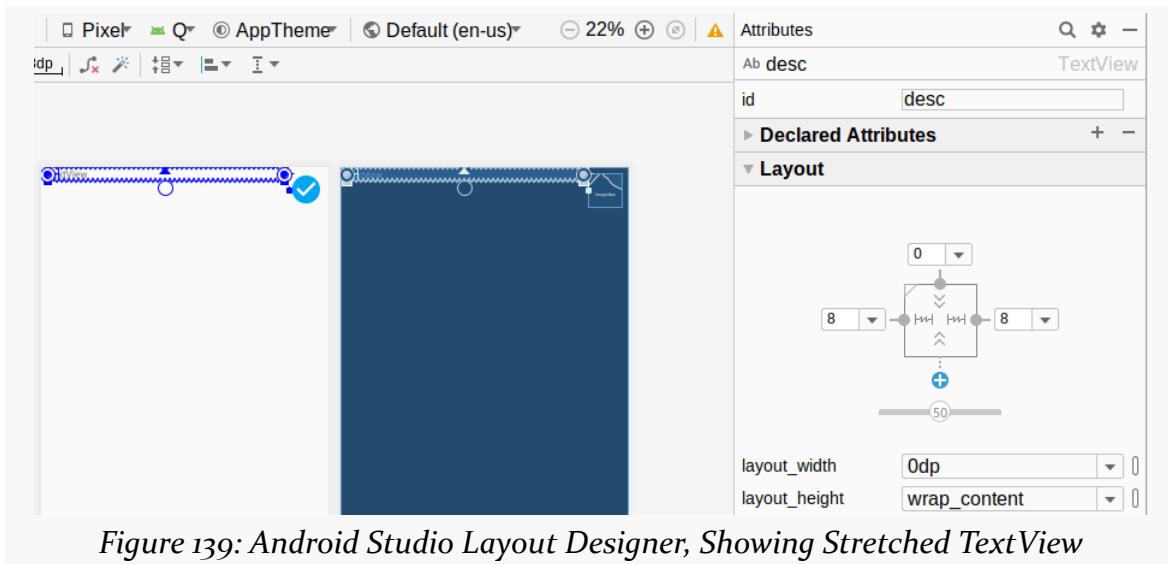


Figure 139: Android Studio Layout Designer, Showing Stretched `TextView`

In the XML, replace the `android:text` attribute with the same binding expression that we used in `todo_row.xml`:

```
    android:text="@{model.description}"
```

## DISPLAYING AN ITEM

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_display.xml](#))

Also add a tools:text attribute:

```
tools:text="This is something to do" />
```

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_display.xml](#))

This gives us something to view in the designer, even though the value at runtime will be coming from our app's data:



Figure 140: Android Studio Layout Designer, Showing Placeholder Text

Back in the “Attributes” pane of the “Design” sub-tab, 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>
```

(from [T15-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 AppTheme resource definition in that file:

```
<item name="textAppearanceHeadline1">@style/HeadlineOneAppearance</item>
```

(from [T15-Display/ToDo/app/src/main/res/values/styles.xml](#))

## DISPLAYING AN ITEM

---

This says “when a widget tries to use `textAppearanceHeadline1`, use this style resource for that”.

Now, back over in the layout designer, we will see that our large bold font is being applied:



Figure 141: Android Studio Layout Designer, Showing Larger, Bolder Placeholder Text

The layout should now resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />

        <import type="android.view.View" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        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_marginEnd="8dp"
            android:layout_marginTop="8dp"
            android:contentDescription="@string/is_completed"
            android:tint="@color/colorAccent"
            android:visibility="@{model.completed ? View.VISIBLE : View.GONE}"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:srcCompat="@drawable/ic_check_circle_black_24dp" />

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

## DISPLAYING AN ITEM

```
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@{model.description}"
    android:textAppearance="?attr/textAppearanceHeadline1"
    app:layout_constraintEnd_toStartOf="@+id/completed"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:text="This is something to do" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

## Step #10: Showing the Created-On Date

The next bit of data to display is the date on which the to-do item was created. This differs from how we handle the description in two ways:

1. We need to provide a label for the date, as otherwise the user may not realize what this date means
2. We need to format the value in a format that the user will understand (versus, say, showing the number of milliseconds since the Unix epoch)

First, let's set up the label. In the "Design" sub-tab, 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 142: Android Studio Layout Designer, Showing Another Added TextView

Give the widget an ID of `labelCreated`.

## DISPLAYING AN ITEM

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 143: 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 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`.

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:

- On the start side of this TextView, to the end side of the `label_created` 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 144: Android Studio Layout Designer, Showing Yet Another TextView

Then, set the “`layout_width`” to `match_constraint` (a.k.a., `0dp`):

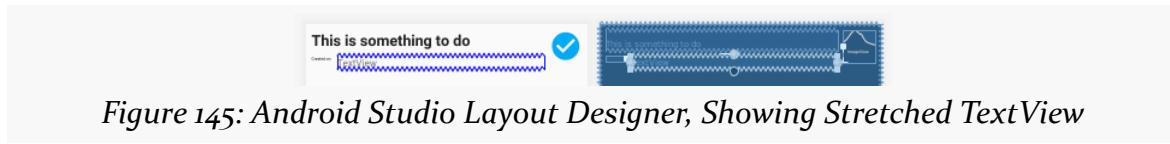


Figure 145: Android Studio Layout Designer, Showing Stretched TextView

## DISPLAYING AN ITEM

---

Give the widget an ID of `createdOn`.

Next, find the `textAppearance` attribute in the “Attributes” pane and set its value to `?attr/textAppearanceHeadline2`, the same as we used for its label.

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 [T15-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 `AppTheme`:

```
<item name="textAppearanceHeadline2">@style/HeadlineTwoAppearance</item>
```

(from [T15-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:

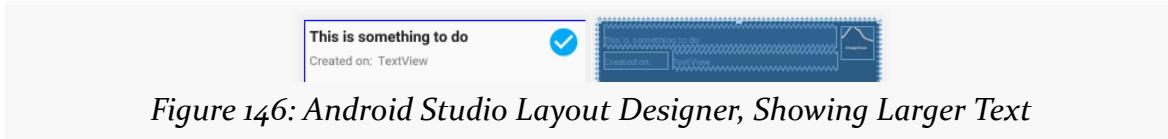


Figure 146: Android Studio Layout Designer, Showing Larger Text

Finally, we need to set the text of this `TextView` to the created-on date. In particular, we need to format the date. Our `ToDoModel` has a `Calendar` object, and we need to convert that into a human-readable string and apply it to this `TextView`.

The best way to do that with data binding is to use a `BindingAdapter`.

Right-click over the `com.commonsware.todo` package and choose “New” > “Kotlin File/Class” from the context menu. Give it a name of `BindingAdapters`, but leave the “Kind” as “File”. Click OK to create the new empty Kotlin file.

## DISPLAYING AN ITEM

---

Then, fill in the following source for it:

```
package com.commonsware.todo

import android.text.format.DateUtils
import android.widget.TextView
import androidx.databinding.BindingAdapter
import java.util.*

@BindingAdapter("formattedDate")
fun TextView.formattedDate(date: Calendar?) {
    date?.let {
        text = DateUtils.getRelativeDateTimeString(
            context,
            date.timeInMillis,
            DateUtils.MINUTE_IN_MILLIS,
            DateUtils.WEEK_IN_MILLIS,
            0
        )
    }
}
```

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

The `@BindingAdapter` annotation can be applied to a Java static method. It so happens that a Kotlin extension function gets converted to a Java static method. So, here, we set up a `formattedDate` adapter for `TextView`. The parameter to our function is a `Calendar?`, so we are saying that if the data binding tools encounter an `app:formattedDate` attribute on a `TextView` (or something extending from `TextView`) and the value of that attribute is a binding expression that returns a `Calendar?`, this function can process that attribute. It formats the `Calendar` using `DateUtils.getRelativeDateTimeString()`, which 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.

Back over in our layout resource, for the `createdOn` `TextView`, replace the existing `android:text` attribute with:

```
app:formattedDate="@{model.createdOn}"
```

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_display.xml](#))

This will trigger our `BindingAdapter` to convert the `createdOn` property of our `ToDoModel` and fill that into our `TextView`.

## DISPLAYING AN ITEM

---

At this point, the XML of the layout resource should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />

        <import type="android.view.View" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        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_marginEnd="8dp"
            android:layout_marginTop="8dp"
            android:contentDescription="@string/is_completed"
            android:tint="@color/colorAccent"
            android:visibility="@{model.completed ? View.VISIBLE : View.GONE}"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:srcCompat="@drawable/ic_check_circle_black_24dp" />

        <TextView
            android:id="@+id/desc"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:text="@{model.description}"
            android:textAppearance="?attr/textAppearanceHeadline1"
            app:layout_constraintEnd_toStartOf="@+id/completed"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            tools:text="This is something to do" />
    
```

## 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_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:formattedDate="@{model.createdOn}"
    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>
</layout>
```

## Step #11: 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 created\_on TextView, and have the other three sides attach to the edges of the ConstraintLayout:

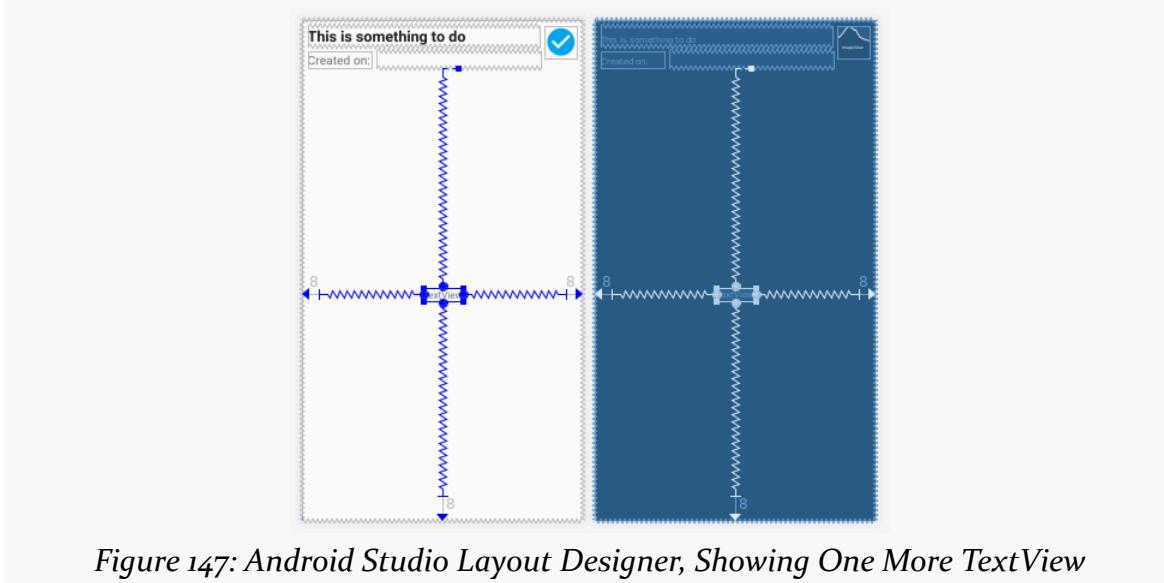


Figure 147: Android Studio Layout Designer, Showing One More TextView

Change both the “layout\_width” and “layout\_height” attributes to `match_constraint`:

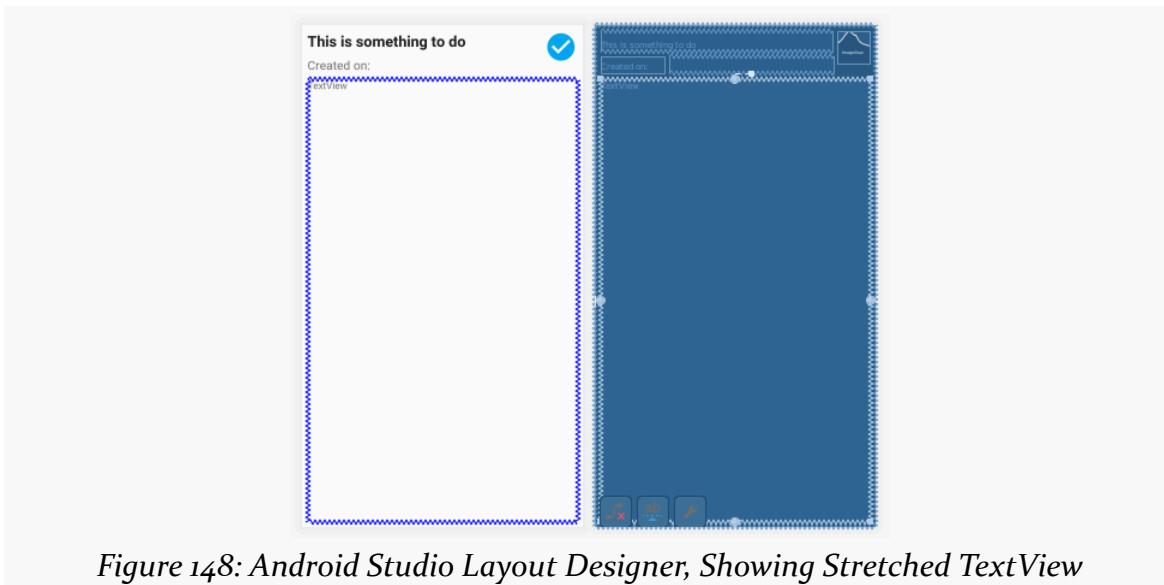


Figure 148: Android Studio Layout Designer, Showing Stretched TextView

## DISPLAYING AN ITEM

---

Give the widget an ID of notes.

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 [T15-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 AppTheme to tie in this new style:

```
<item name="textAppearanceBody1">@style/BodyAppearance</item>
```

(from [T15-Display/ToDo/app/src/main/res/values/styles.xml](#))

Right now, the overall styles.xml resource should look like:

```
<resources>  
  
    <!-- Base application theme. -->  
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">  
        <!-- Customize your theme here. -->  
        <item name="colorPrimary">@color/colorPrimary</item>  
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>  
        <item name="colorAccent">@color/colorAccent</item>  
        <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 [T15-Display/ToDo/app/src/main/res/values/styles.xml](#))

Finally, back in the layout editor, in the "Text" sub-tab, use a binding expression to

## DISPLAYING AN ITEM

---

tie the notes from the ToDoModel to the android:text attribute:

```
    android:text="@{model.notes}"
```

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_display.xml](#))

At this point, your layout XML should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />

        <import type="android.view.View" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        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_marginEnd="8dp"
            android:layout_marginTop="8dp"
            android:contentDescription="@string/is_completed"
            android:tint="@color/colorAccent"
            android:visibility="@{model.completed ? View.VISIBLE : View.GONE}"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintTop_toTopOf="parent"
            app:srcCompat="@drawable/ic_check_circle_black_24dp" />

        <TextView
            android:id="@+id/desc"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:text="@{model.description}"
```

## DISPLAYING AN ITEM

---

```
        android:textAppearance="?attr/textAppearanceHeadline1"
        app:layout_constraintEnd_toStartOf="@+id/completed"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        tools:text="This is something to do" />

<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_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:formattedDate="@{model.createdOn}"
    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_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    android:text="@{model.notes}"
    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>
</layout>
```

(from [T15-Display/ToDo/app/src/main/res/layout/todo\\_display.xml](#))

## Step #12: 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 click on the displayFragment. In the “Attributes” pane, you will see an “Arguments” section with a + icon:

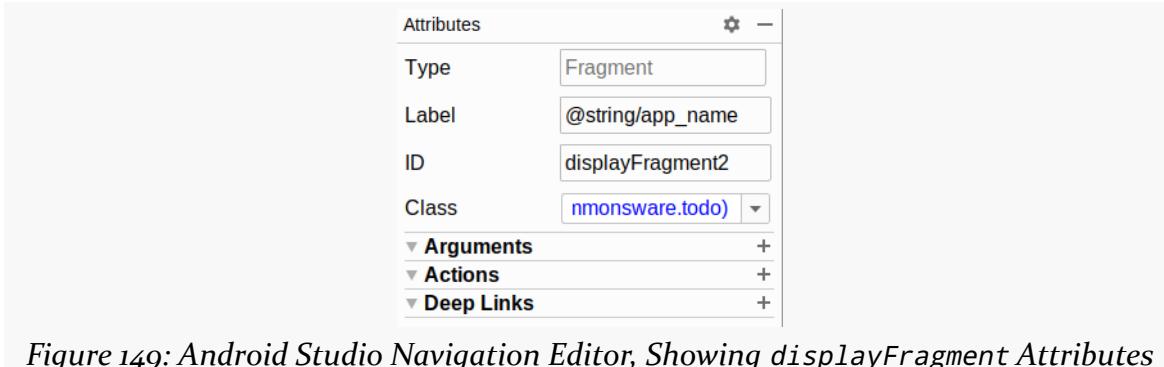


Figure 149: Android Studio Navigation Editor, Showing displayFragment Attributes

Click that + icon in the “Arguments” to open up a dialog to define an argument:

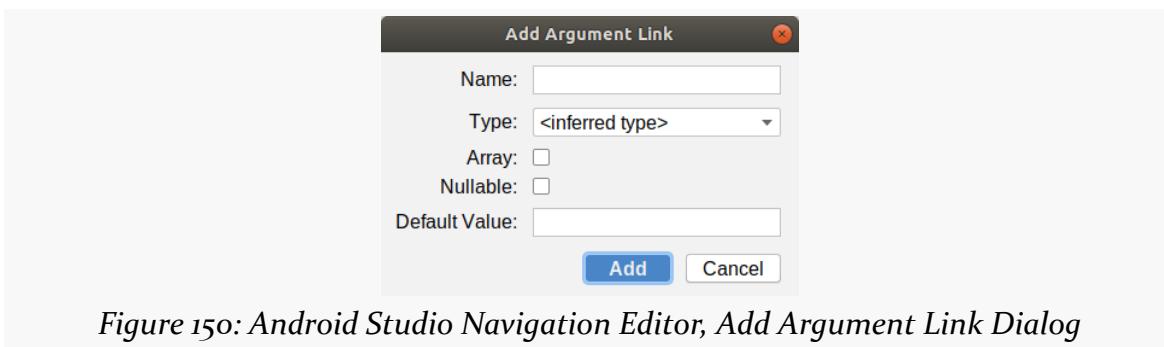


Figure 150: Android Studio Navigation Editor, Add Argument Link Dialog

Fill it in as follows:

## DISPLAYING AN ITEM

Property	Value
Name	modelId
Type	String
Array	unchecked
Nullable	unchecked
Default Value	(leave empty)

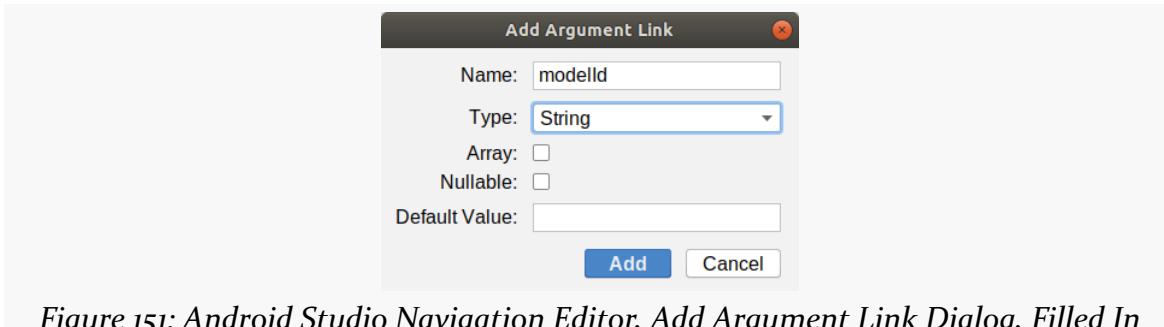


Figure 151: Android Studio Navigation Editor, Add Argument Link Dialog, Filled In

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())
}
```

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

## DISPLAYING AN ITEM

---

RosterListFragment should now 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.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import kotlinx.android.synthetic.main.main_todo_roster.*
import kotlinx.android.synthetic.main.todo_roster.view.*

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

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

        val adapter =
            RosterAdapter(
                inflater = layoutInflater,
                onCheckboxToggle = { model ->
                    ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
                },
                onRowClick = { model -> display(model) })

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

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

    adapter.submitList(ToDoRepository.items)
    empty.visibility = View.GONE
}

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

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

Then, over in DisplayFragment, add this property:

## DISPLAYING AN ITEM

---

```
private val args: DisplayFragmentArgs by navArgs()
```

(from [T15-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 #13: Populating the Layout

Now, we can return to our Kotlin code, specifically the DisplayFragment, and hook it up to this newly-created layout resource.

In DisplayFragment, add a binding field, pointing to our newly-generated TodoDisplayBinding class from our todo\_display layout resource:

```
private lateinit var binding: TodoDisplayBinding
```

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

Then, in DisplayFragment, add an onCreateView() method:

```
override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?
) = TodoDisplayBinding.inflate(inflater, container, false)
    .apply { binding = this }
    .root
```

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

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.

In order to be able to bind the ToDoModel, we need the model object. We can get its ID from args, but we need the full ToDoModel itself. We can do that by adding another method to ToDoRepository, named find():

```
fun find(modelId: String) = items.find { it.id == modelId }
```

## DISPLAYING AN ITEM

---

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

This just iterates over the models to find the one with the ID. Eventually, this will be replaced with a database-backed solution.

Then, back in `DisplayFragment`, add this `onViewCreated()` method:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    binding.model = ToDoRepository.find(args.modelId)
}
```

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

This retrieves the model given its ID and binds it to the layout.

The completed `DisplayFragment` 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.navigation.fragment.navArgs
import com.commonsware.todo.databinding.TodoDisplayBinding

class DisplayFragment : Fragment() {
    private val args: DisplayFragmentArgs by navArgs()
    private lateinit var binding: TodoDisplayBinding

    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?) {
        binding.model = ToDoRepository.find(args.modelId)
    }
}
```

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

At this point, if you run the app, and you click on one of the to-do items in the list,

## DISPLAYING AN ITEM

---

the full details should appear in the `DisplayFragment`:

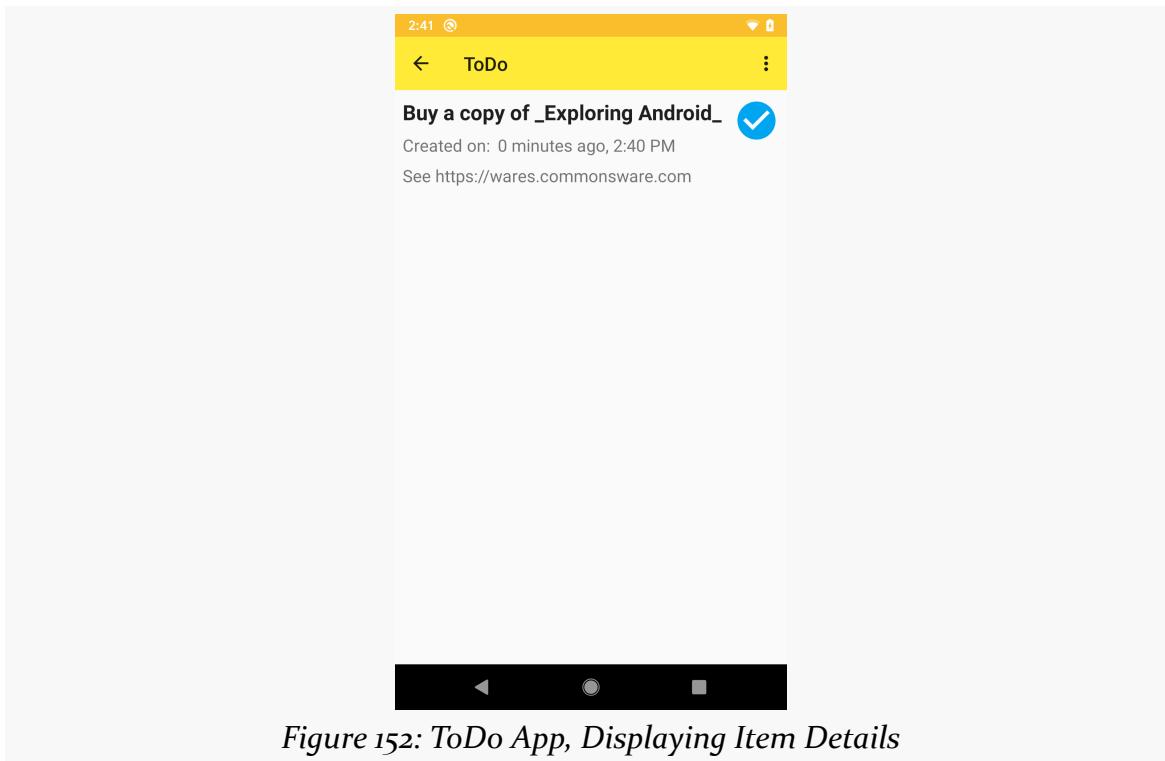


Figure 152: ToDo App, Displaying Item Details

Pressing BACK returns you to the list, as before.

## 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/commonware/todo/DisplayFragment.kt](#)
- [app/src/main/res/navigation/nav\\_graph.xml](#)
- [app/src/main/res/layout/todo\\_row.xml](#)
- [app/src/main/java/com/commonware/todo/RosterRowHolder.kt](#)
- [app/src/main/java/com/commonware/todo/RosterAdapter.kt](#)
- [app/src/main/java/com/commonware/todo/RosterListFragment.kt](#)
- [app/src/main/java/com/commonware/todo/MainActivity.kt](#)
- [app/src/main/res/layout/todo\\_display.xml](#)
- [app/src/main/res/drawable/ic\\_check\\_circle\\_black\\_24dp.xml](#)
- [app/src/main/res/values/styles.xml](#)

## DISPLAYING AN ITEM

---

- [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 an icon to the `DisplayFragment`, 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.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 Java class. For the name, fill in `EditFragment`. For the kind, choose “Class”. Click “OK” to create the class.

That will give you an `EditFragment` that looks like:

```
package com.commonsware.todo

class EditFragment {
```

Then, have it extend the `Fragment` class:

```
package com.commonsware.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:

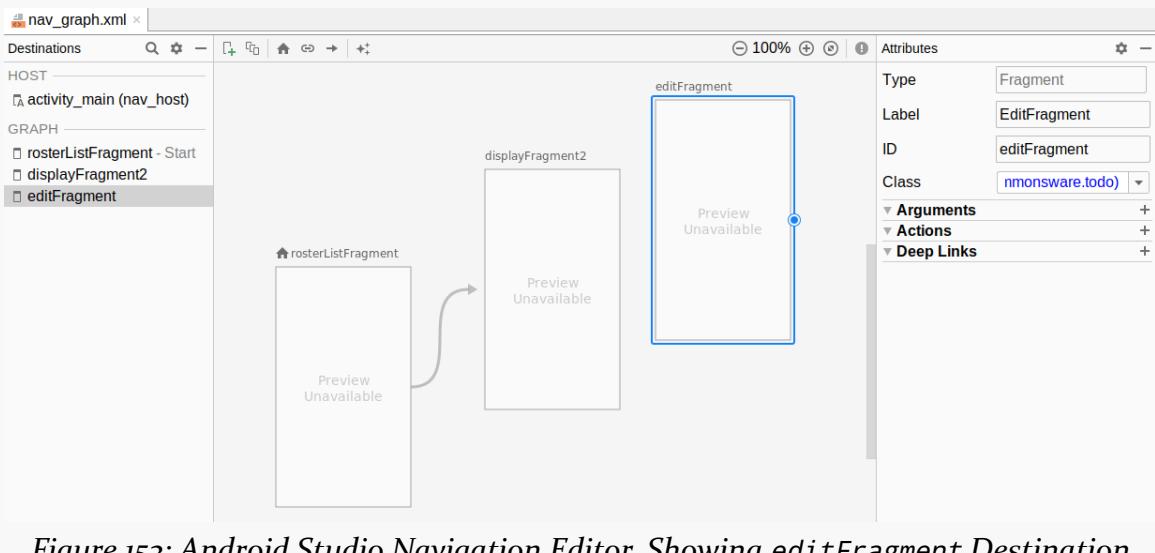


Figure 153: Android Studio Navigation Editor, Showing `editFragment` Destination

Next, click on `displayFragment` and drag an arrow from it to `editFragment`:

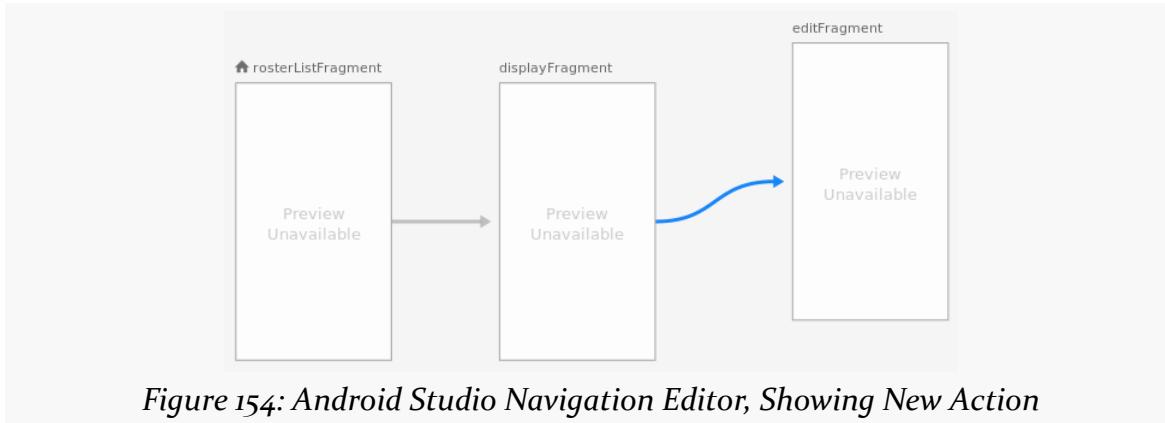


Figure 154: Android Studio Navigation Editor, Showing New Action

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

## EDITING AN ITEM

---

“Add” to add it to the navigation graph.

At this point, the overall 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"
    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" />
        <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 [T16-Edit/ToDo/app/src/main/res/navigation/nav\\_graph.xml](#))

## Step #3: Setting Up a Menu Resource

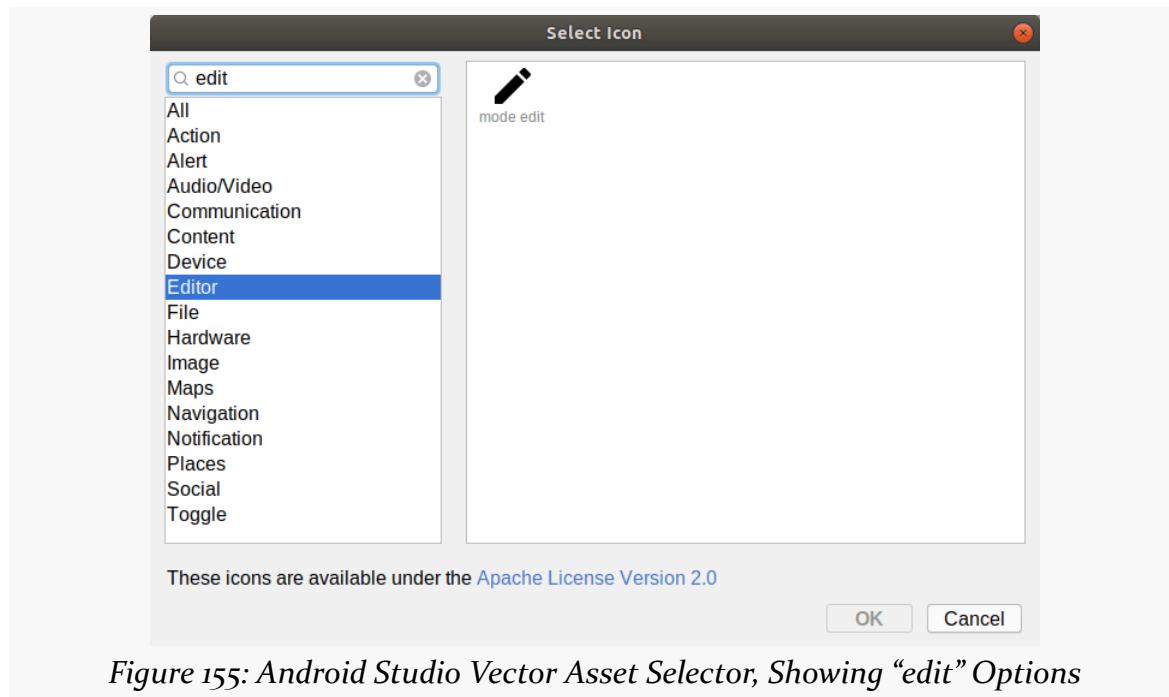
Somewhere, somehow, the user has to be able to get to this fragment. A typical pattern is for there to be an “edit” option somewhere where we are displaying the thing to be edited. In the case of this app, that implies having an “edit” option on the

## EDITING AN ITEM

---

DisplayFragment, and we can do this by adding a action bar 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 155: Android Studio Vector Asset Selector, Showing “edit” Options*

Choose the “mode edit” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

## 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:

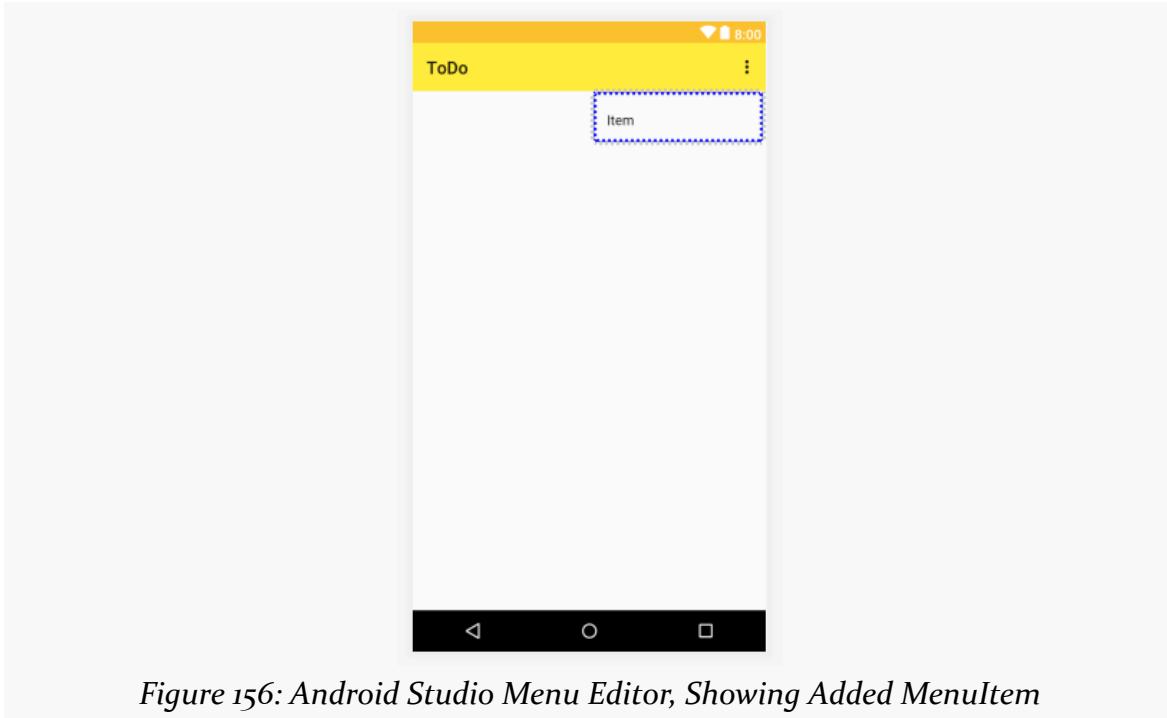


Figure 156: Android Studio Menu Editor, Showing Added MenuItem

## EDITING AN ITEM

In the Attributes pane, fill in edit for the “id”. Then, choose both “ifRoom” and “withText” for the “showAsAction” option:

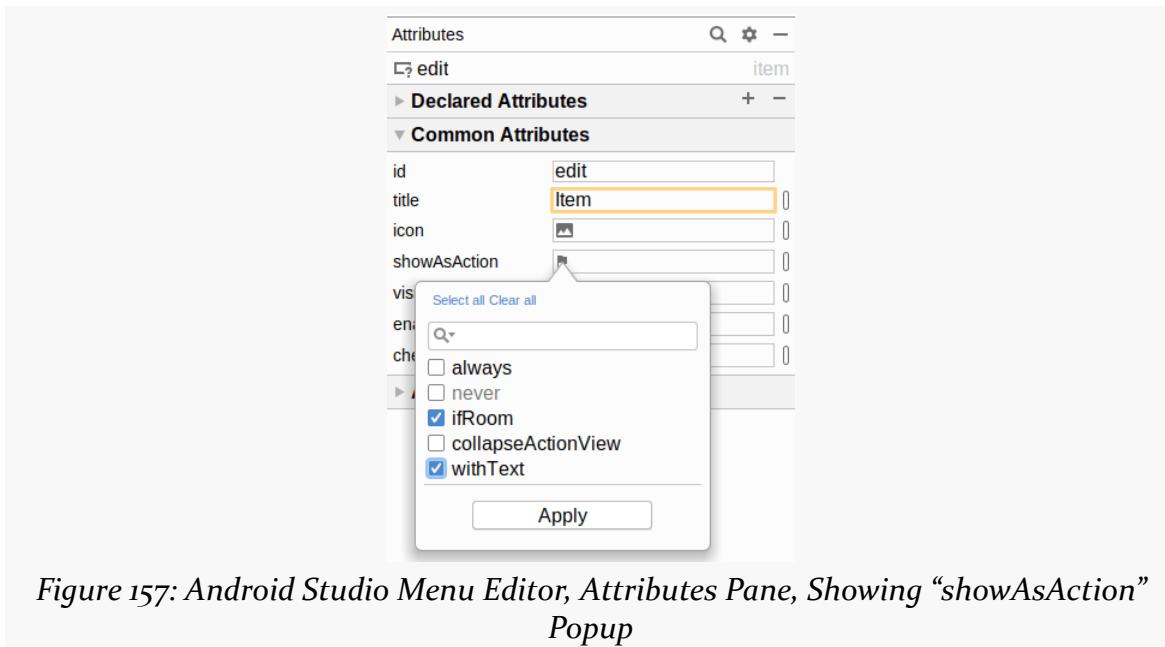


Figure 157: Android Studio Menu Editor, Attributes Pane, Showing “showAsAction” Popup

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_mode_edit_black_24dp` 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_edit` as the resource name and “Edit” as the resource value. Click OK to close the dialog.

## EDITING AN ITEM

---

At this point, your menu editor preview should resemble:

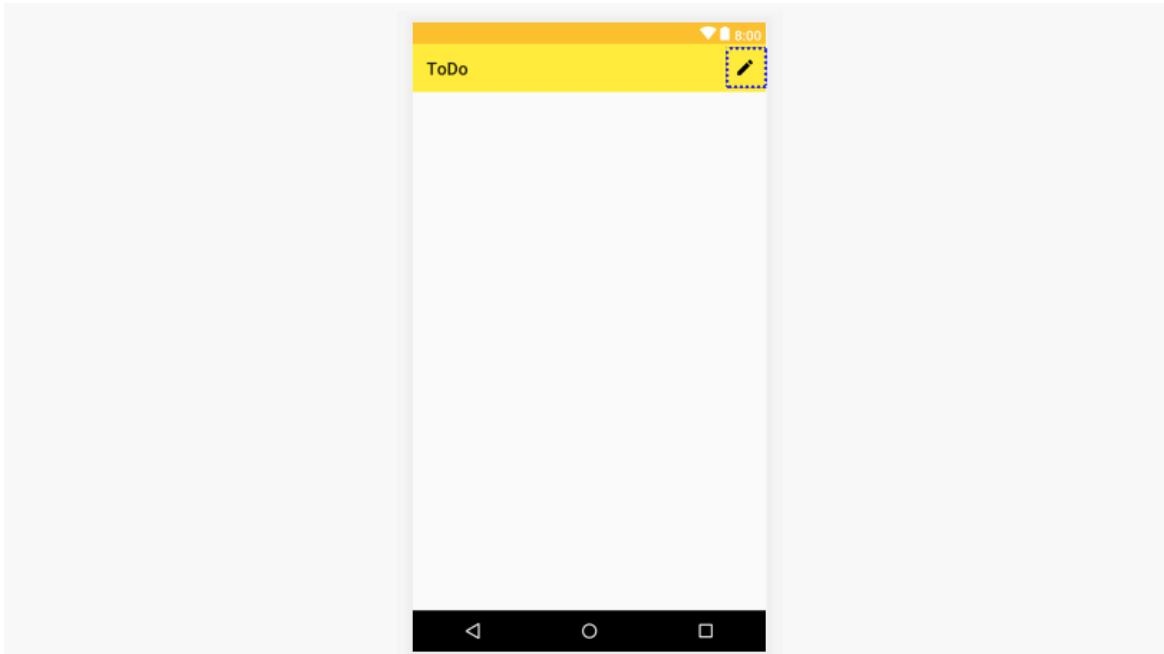


Figure 158: Android Studio Menu Editor, Showing Configured MenuItem

...and the menu resource itself should have this XML in the “Text” sub-tab:

```
<?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_mode_edit_black_24dp"
        android:title="@string/menu_edit"
        app:showAsAction="ifRoom|withText" />
</menu>
```

(from [Ti6-Edit/Todo/app/src/main/res/menu/actions\\_display.xml](#))

## Step #4: Showing the Action Item

We also need to take steps to arrange to show this action bar item on DisplayFragment. [Previously](#), we defined an action bar item that would be available to the entire activity. Now we want one that will be for just this one fragment. The

## EDITING AN ITEM

---

way to do that is to have the fragment itself add this item to the action 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 [T16-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 action bar, via `setHasOptionsMenu(true)`.

Next, add this `onCreateOptionsMenu()` method to `DisplayFragment`:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.actions_display, menu)

    super.onCreateOptionsMenu(menu, inflater)
}
```

(from [T16-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 action bar. Plus, we chain to the superclass, in case the superclass wants to add things to the action bar as well.

## EDITING AN ITEM

---

If you run the app and tap on a to-do item in the list, you should see the new action bar item on the `DisplayFragment`:

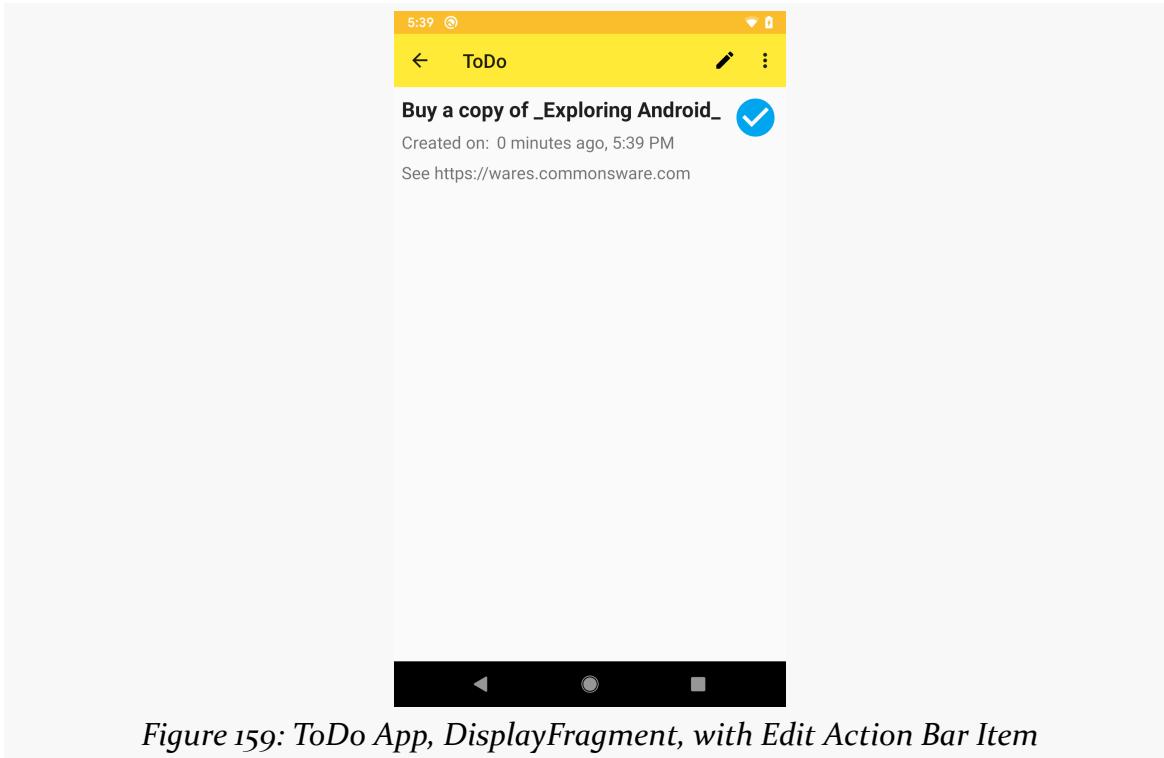


Figure 159: *ToDo App, DisplayFragment, with Edit Action Bar Item*

This is in addition to the overflow menu, which still has our “About” item. By having our activity’s Toolbar serve as the action 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 action 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 [T16-Edit/ToDo/app/src/main/java/com/commonware/todo/DisplayFragment.kt](#))

As we did in `RosterListFragment`, we use `findNavController()` to get the

## EDITING AN ITEM

---

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 [T16-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 click on the “edit” action 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: Setting Up Data Binding

As with the roster rows and DisplayFragment, we are going to use data binding to populate the widgets.

In the Text sub-tab of the todo\_edit editor, modify the layout to look like this:

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

<data>

<variable
    name="model"
    type="com.commonsware.todo.ToDoModel" />
</data>

<androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

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

This sets up a model variable, the same as we used in previous data binding cases.

## Step #8: 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 “Design” sub-tab of the layout editor 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 16o: Android Studio Layout Designer, Showing Added CheckBox

In the Attributes pane, clear out the contents of the “text” attribute, as we just want

## EDITING AN ITEM

---

a bare checkbox, without a caption. Also, give the widget an ID of `isCompleted`.

Then, switch to the “Text” sub-tab and add an `android:checked` attribute with a binding expression, to check the CheckBox based on the completion status of the `ToDoModel`:

```
    android:checked="@{model.completed}"
```

(from [T16-Edit/ToDo/app/src/main/res/layout/todo\\_edit.xml](#))

At this point, the XML should resemble:

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        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"
            android:checked="@{model.completed}"
            app:layout_constraintStart_toStartOf="parent"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

## Step #9: 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.

## EDITING AN ITEM

---

Switch back to the “Design” sub-tab 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 161: Android Studio Layout Designer, Showing Added `EditText`

Next, change the “`layout_width`” attribute in the Attributes pane to `match_constraint`:

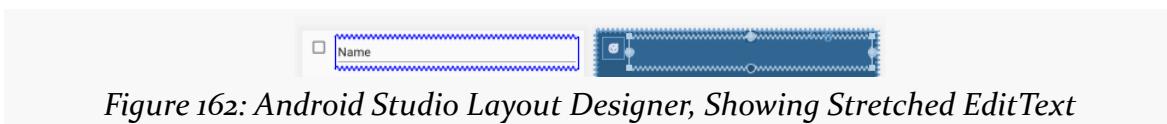


Figure 162: Android Studio Layout Designer, Showing Stretched `EditText`

Then, give the `EditText` an ID of `desc` in the Attributes pane.

If you look closely, you will see that our `CheckBox` is not very well aligned vertically with respect to the `EditText`:

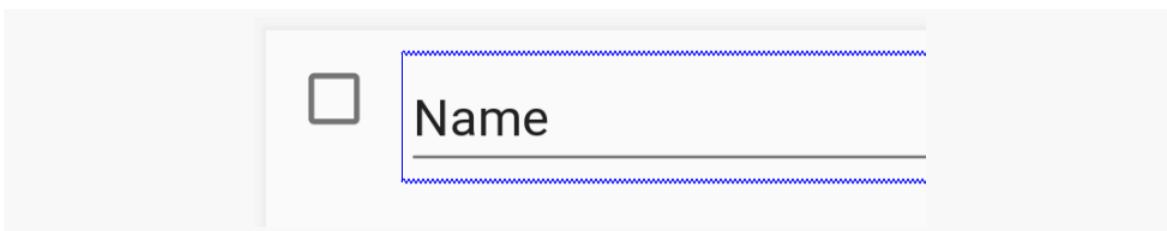


Figure 163: Android Studio Layout Designer, Showing Vertical Alignment Problem

Ideally, it would be vertically centered. To do that:

## EDITING AN ITEM

---

- Remove the constraint tying the top of the CheckBox to the top of the ConstraintLayout, by clicking on the top grab handle
- Drag a fresh constraint from the top of the CheckBox to the top of the EditText
- Create a similar constraint, from the bottom of the CheckBox to the bottom of the EditText

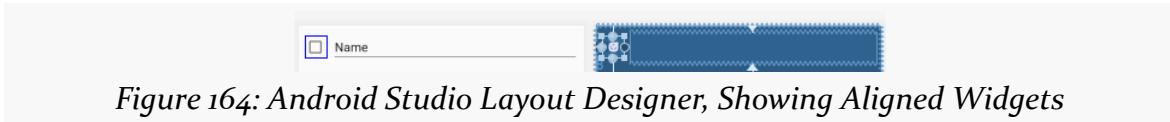


Figure 164: Android Studio Layout Designer, Showing Aligned Widgets

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:

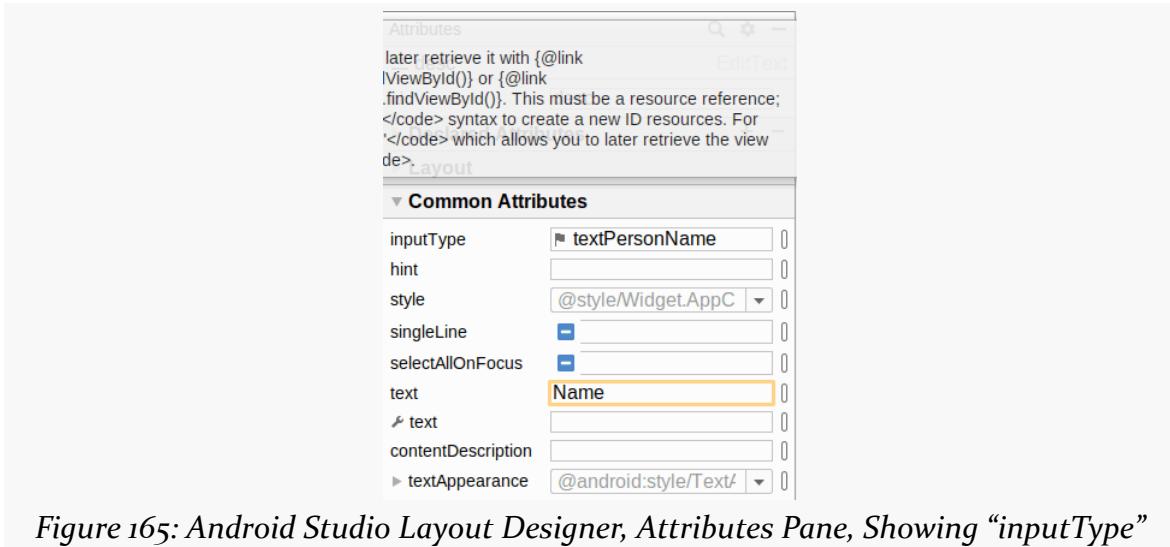


Figure 165: Android Studio Layout Designer, Attributes Pane, Showing “inputType”

## 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`:

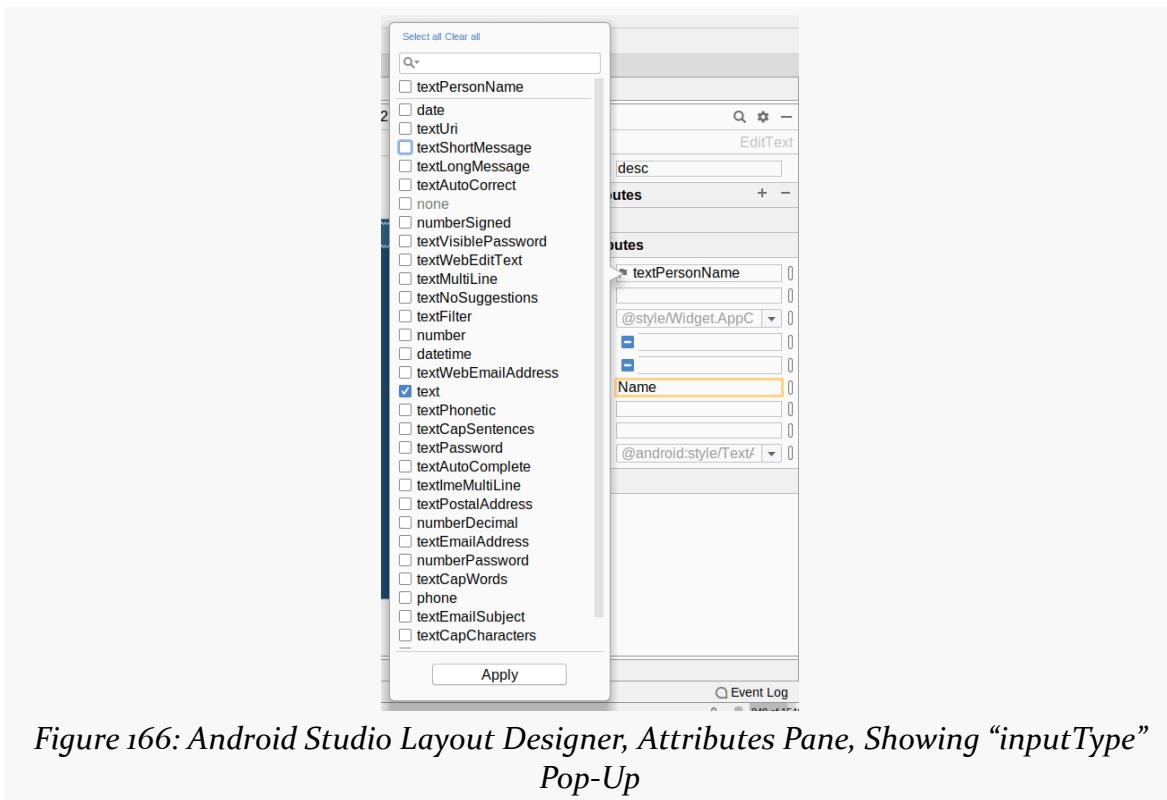


Figure 166: Android Studio Layout Designer, Attributes Pane, Showing “`inputType`” Pop-Up

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, switch to the “Text” sub-tab and replace the existing `android:text` attribute value with a binding expression:

## EDITING AN ITEM

---

```
    android:text="@{model.description}"
```

(from [T16-Edit/ToDo/app/src/main/res/layout/todo\\_edit.xml](#))

At this point, the layout XML should resemble:

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

    <data>

        <variable
            name="model"
            type="com.commonsware.todo.ToDoModel" />
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout
        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:checked="@{model.completed}"
            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_marginEnd="8dp"
            android:layout_marginStart="8dp"
            android:layout_marginTop="8dp"
            android:ems="10"
            android:hint="@string/desc"
            android:inputType="textPersonName"
            android:text="@{model.description}"
            app:layout_constraintEnd_toEndOf="parent"
            app:layout_constraintStart_toEndOf="@+id/isCompleted"
            app:layout_constraintTop_toTopOf="parent" />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

## Step #10: Adding the Notes Field

The other widget is another `EditText`, this time for the notes.

Switch back to the “Design” sub-tab 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:

- 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`

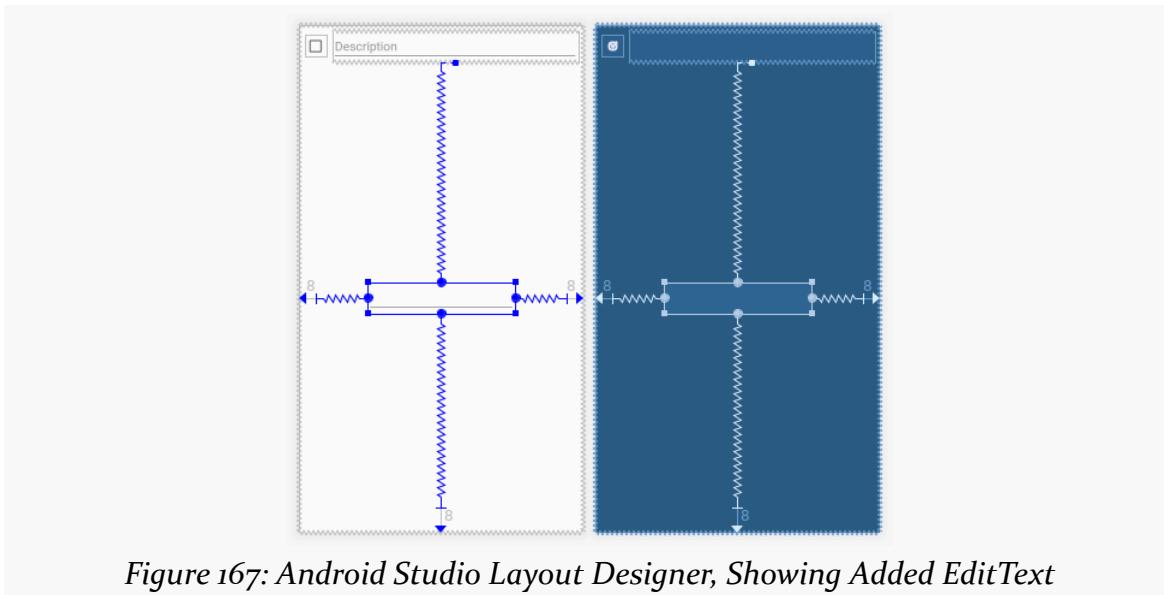


Figure 167: Android Studio Layout Designer, Showing Added `EditText`

## EDITING AN ITEM

---

Then, set both the “layout\_height” and “layout\_width” attributes to `match_constraint`:

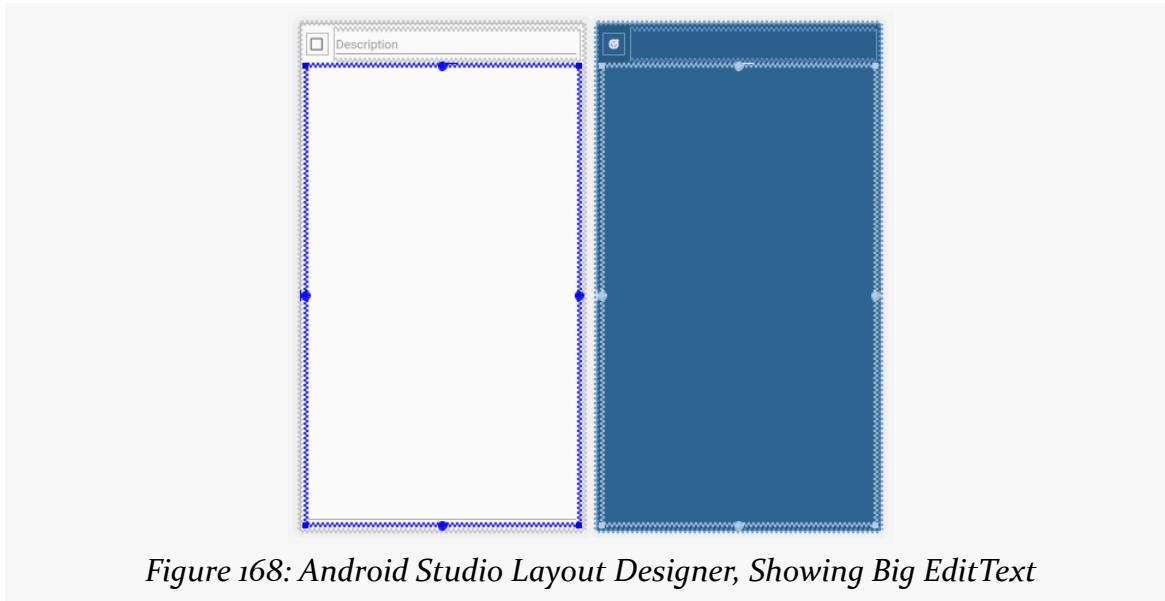


Figure 168: Android Studio Layout Designer, Showing Big EditText

Give the widget an ID of notes.

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.

Finally, switch back to the “Text” sub-tab and add an `android:text` attribute with a binding expression:

```
    android:text="@{model.notes}"
```

(from [T16-Edit/ToDo/app/src/main/res/layout/todo\\_edit.xml](#))

At this point, the layout is complete and should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:tools="http://schemas.android.com/tools"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <data>

        <variable
            name="model"
```

## EDITING AN ITEM

---

```
    type="com.commonsware.todo.ToDoModel" />
</data>

<androidx.constraintlayout.widget.ConstraintLayout
    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:checked="@{model.completed}"
        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_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:ems="10"
        android:hint="@string/desc"
        android:inputType="textPersonName"
        android:text="@{model.description}"
        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_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:ems="10"
        android:gravity="start|top"
        android:hint="@string/notes"
        android:inputType="textMultiLine"
        android:text="@{model.notes}"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
```

## EDITING AN ITEM

---

```
    app:layout_constraintTop_toBottomOf="@+id/desc" />
</androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

(from [T16-Edit/ToDo/app/src/main/res/layout/todo\\_edit.xml](#))

## Step #11: 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 and our navigation arguments:

```
private lateinit var binding: TodoEditBinding
private val args: EditFragmentArgs by navArgs()
```

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

Then, add functions to inflate the binding and bind our model:

```
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?) {
    binding.model = ToDoRepository.find(args.modelId)
}
```

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

The entire 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
```

## EDITING AN ITEM

```
class EditFragment : Fragment() {
    private lateinit var binding: TodoEditBinding
    private val args: EditFragmentArgs by navArgs()

    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?) {
        binding.model = ToDoRepository.find(args.modelId)
    }
}
```

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

If you run the app, click on a to-do item to display it, then click on the “edit” action bar item, you will get a form for modifying the to-do item:

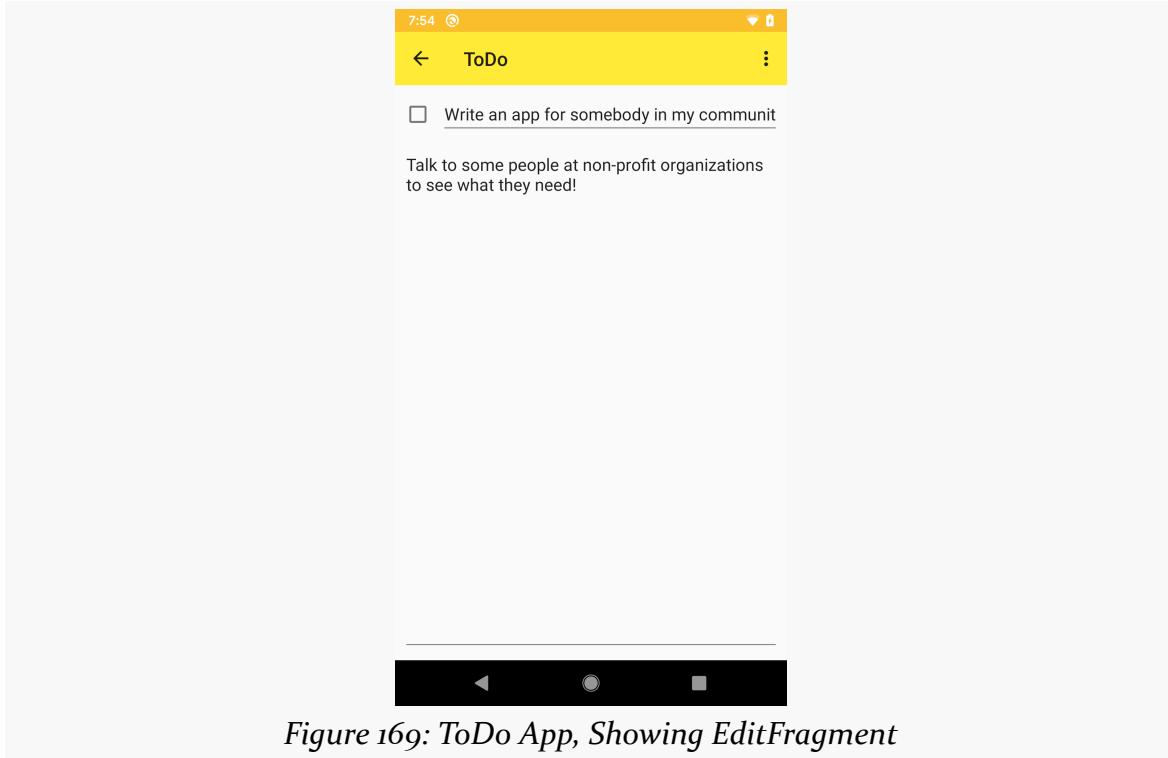


Figure 169: ToDo App, Showing EditFragment

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.

## 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\\_black\\_24dp.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](#)
- [app/src/main/res/drawable/ic\\_check\\_circle\\_black\\_24dp.xml](#)
- [app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# 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 action 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](#).

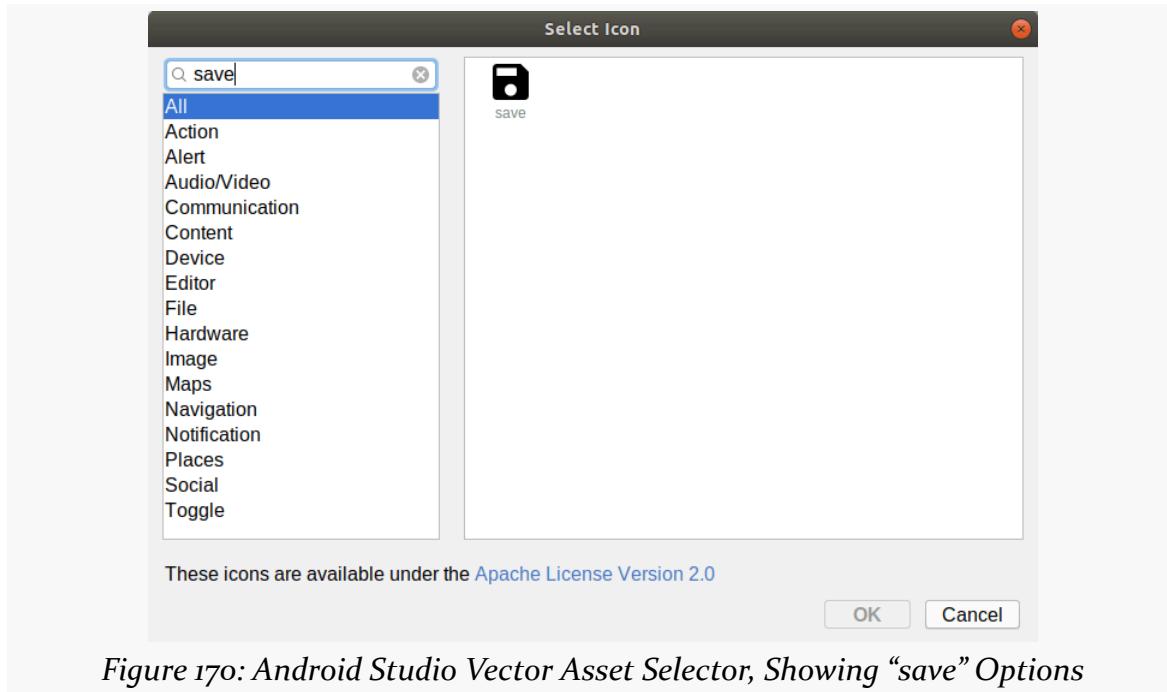
## Step #1: Adding the Action Bar Item

First, let’s set up the Save action 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 170: Android Studio Vector Asset Selector, Showing “save” Options*

Choose the “`save`” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

## SAVING AN ITEM

---

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:

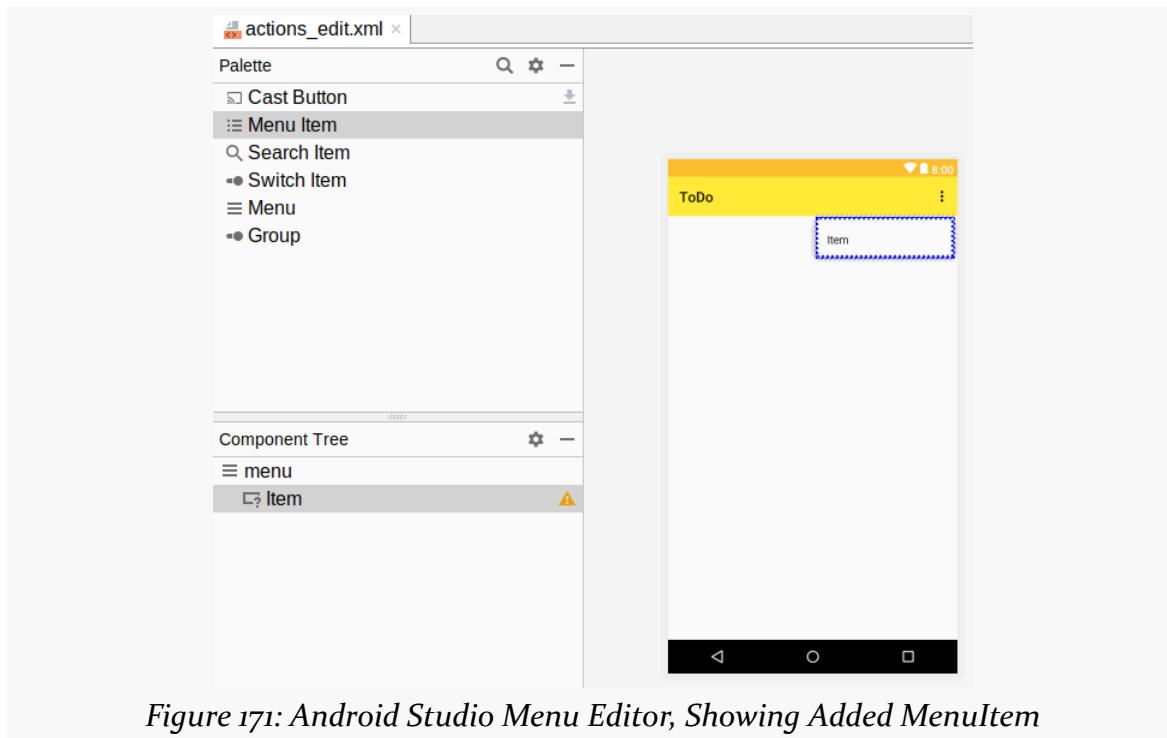


Figure 171: Android Studio Menu Editor, Showing Added MenuItem

In the Attributes pane, fill in `save` 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_save_black_24dp` in the list of drawables, then click `OK` to accept that choice of icon.

## SAVING AN ITEM

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_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 action bar item:

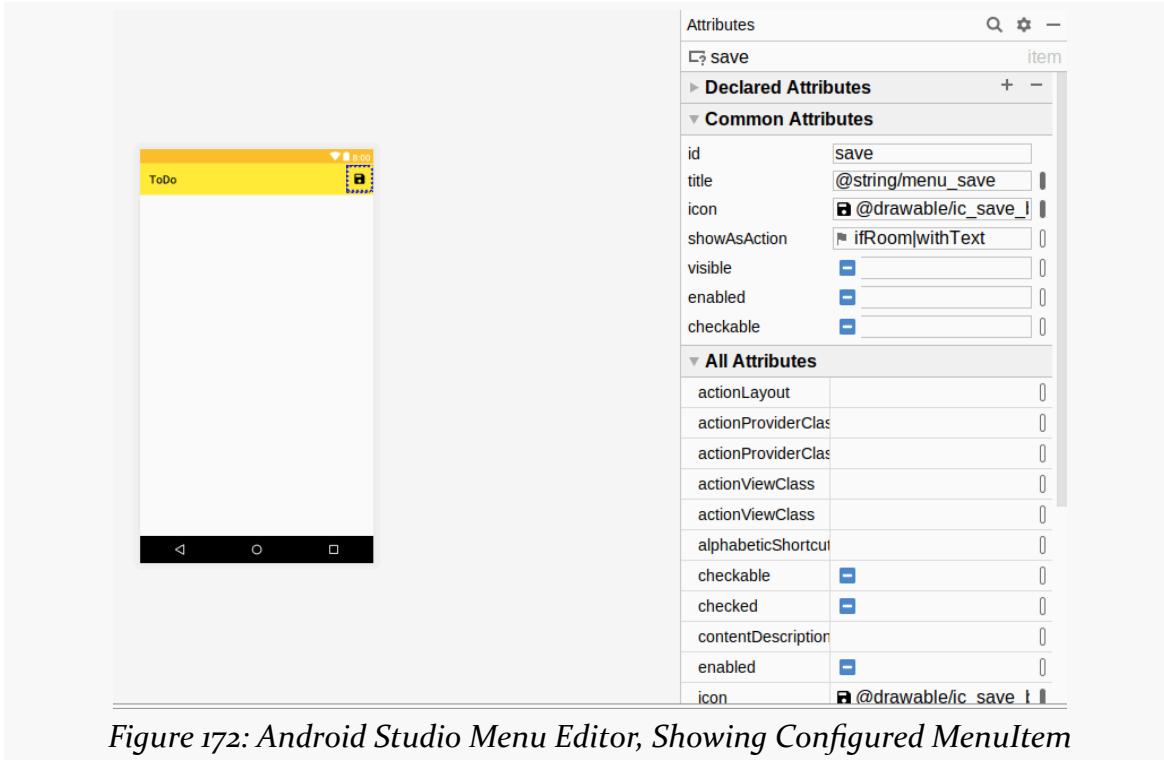


Figure 172: Android Studio Menu Editor, Showing Configured MenuItem

We also need to take steps to arrange to show this action 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 action bar:

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

    setHasOptionsMenu(true)
}
```

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

Next, add this `onCreateOptionsMenu()` function to `EditFragment`, to inflate our

## SAVING AN ITEM

---

newly-created menu resource:

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {  
    inflater.inflate(R.menu.actions_edit, menu)  
  
    super.onCreateOptionsMenu(menu, inflater)  
}
```

(from [T17-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 action bar item on the EditFragment:

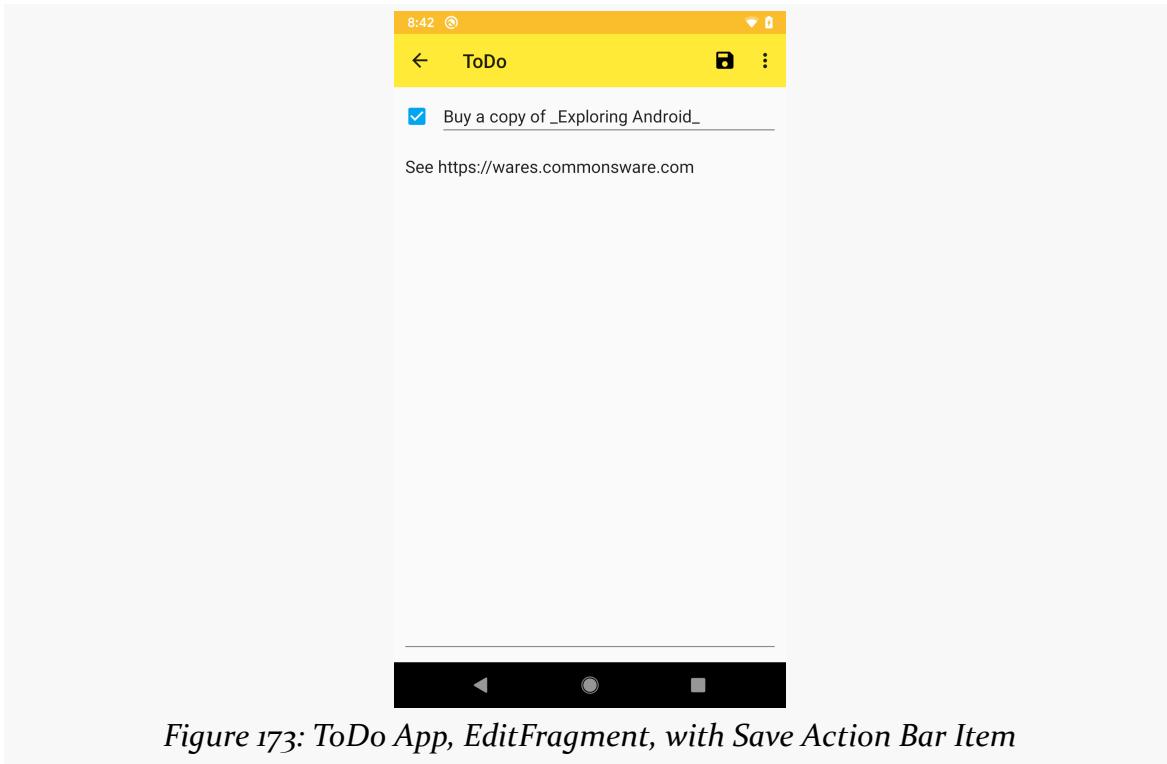


Figure 173: ToDo App, EditFragment, with Save Action Bar Item

## Step #2: Replacing the Item

Now that we have the action 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() {
```

## SAVING AN ITEM

---

```
val edited = binding.model?.copy(  
    description = binding.desc.text.toString(),  
    isCompleted = binding.isCompleted.isChecked,  
    notes = binding.notes.text.toString()  
)  
  
edited?.let { ToDoRepository.save(it) }  
}
```

Here we:

- Retrieve our current ToDoModel from the binding
- Use the copy() function on our data class to create a revised instance of ToDoModel with the data from the form
- Tell the ToDoRepository to replace the existing ToDoModel for this ID with this copy

Then, arrange to call this save() method when the user clicks on the “save” action 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 [T17-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 action bar item... nothing seems to happen. But, if you then press BACK to return to the DisplayFragment, you will see the change that you made to the to-do item.

## Step #3: 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:

## SAVING AN ITEM

---

```
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 action bar.

Then, add a call to navToDisplay() from save():

```
private fun save() {
    val edited = binding.model?.copy(
        description = binding.desc.text.toString(),
        isCompleted = binding.isCompleted.isChecked,
        notes = binding.notes.text.toString()
    )

    edited?.let { ToDoRepository.save(it) }
    navToDisplay()
}
```

(from [T17-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 [T17-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.

## SAVING AN ITEM

---

Then, modify navToDisplay() in MainActivity to call hideKeyboard():

```
private fun navToDisplay() {
    hideKeyboard()
    findNavController().popBackStack()
}
```

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

At this point, 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

class EditFragment : Fragment() {
    private lateinit var binding: TodoEditBinding
    private val args: EditFragmentArgs by navArgs()

    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?) {
        binding.model = ToDoRepository.find(args.modelId)
    }

    override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
        inflater.inflate(R.menu.actions_edit, menu)
    }
}
```

## SAVING AN ITEM

---

```
super.onCreateOptionsMenu(menu, inflater)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.save -> { save(); return true; }
    }

    return super.onOptionsItemSelected(item)
}

private fun save() {
    val edited = binding.model?.copy(
        description = binding.desc.text.toString(),
        isCompleted = binding.isCompleted.isChecked,
        notes = binding.notes.text.toString()
    )

    edited?.let { ToDoRepository.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 [T17-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.

## 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\\_black\\_24dp.xml](#)
- [app/src/main/res/menu/actions\\_edit.xml](#)
- [app/src/main/java/com/commonsware/todo/EditFragment.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 = listOf<ToDoModel>()

(from T18-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`:

```
fun delete(model: ToDoModel) {
    items = items.filter { it.id != model.id }
```

## ADDING AND DELETING ITEMS

---

```
}
```

(from [T18-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:

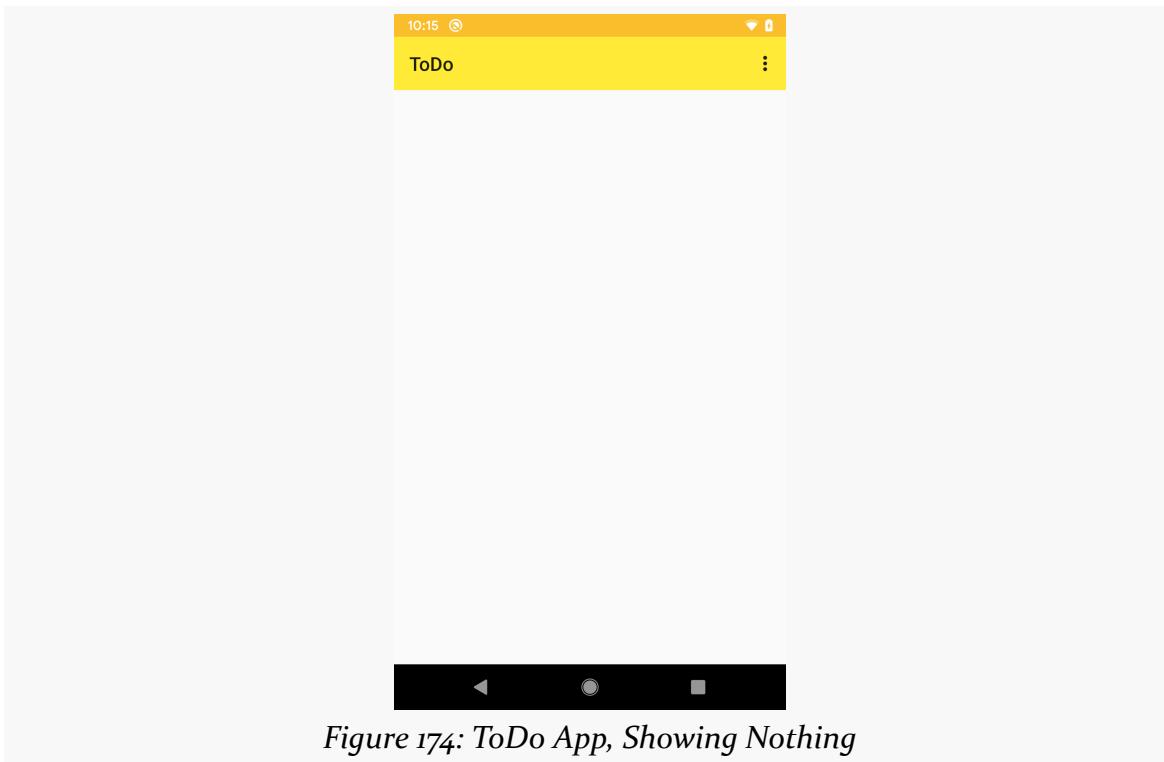


Figure 174: ToDo App, Showing Nothing

## 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.

In `onViewCreated()` of `RosterListFragment`, remove the `empty.visibility = View.GONE` line, leaving you with:

## ADDING AND DELETING ITEMS

---

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

    val adapter =
        RosterAdapter(
            inflater = layoutInflater,
            onCheckboxToggle = { model ->
                ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
            },
            onRowClick = { model -> display(model) }

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

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

    adapter.submitList(ToDoRepository.items)
}
```

## ADDING AND DELETING ITEMS

---

Now when you run the app, you will see... some placeholder text:

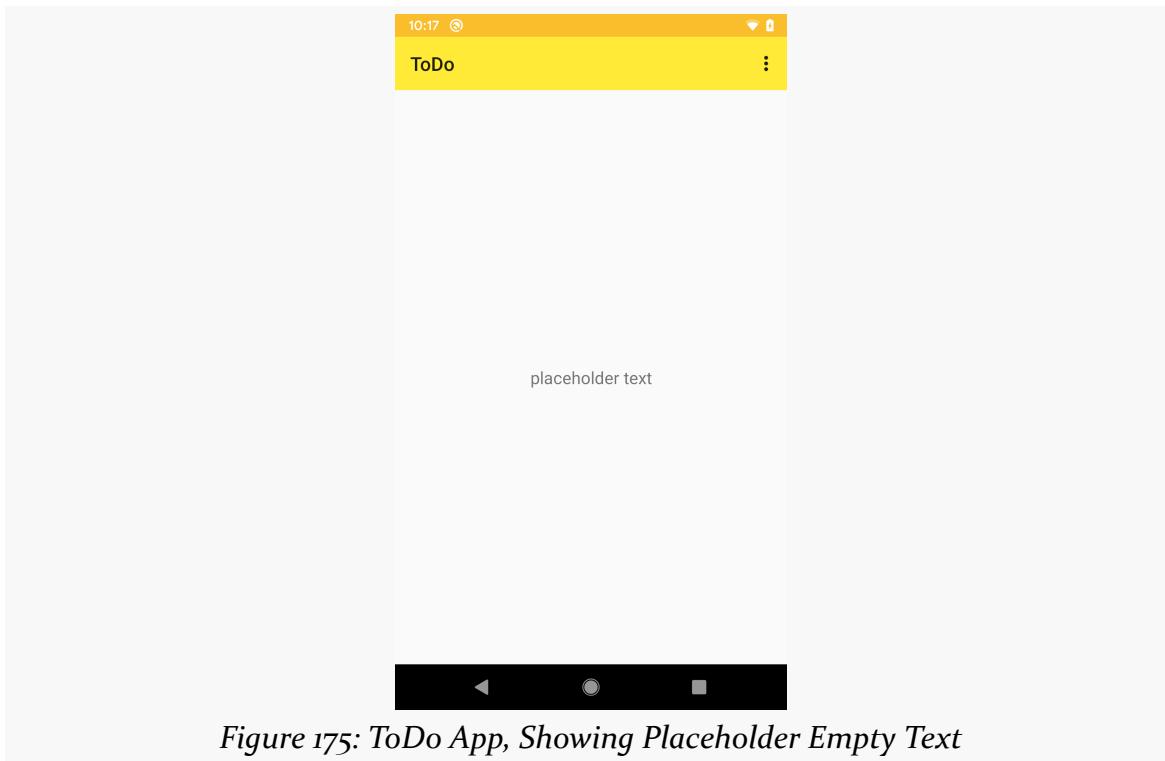


Figure 175: ToDo App, Showing Placeholder Empty Text

We will replace that text with a better message shortly.

## Step #3: Adding an Add Action Bar Item

We need to add another action 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:

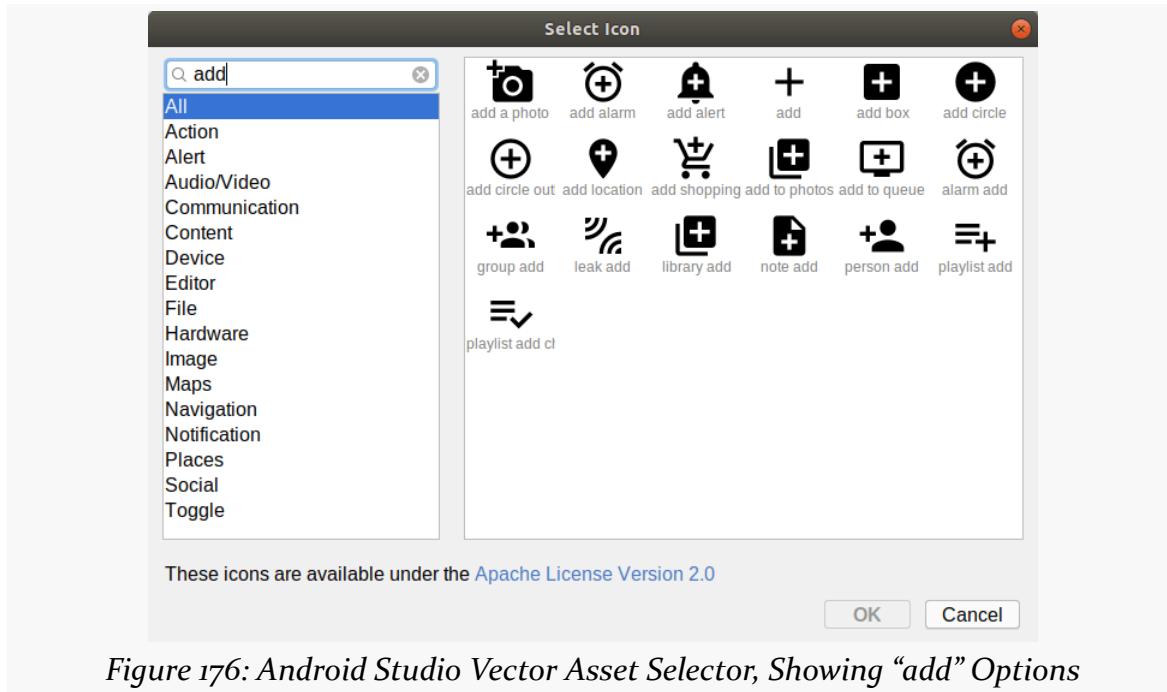


Figure 176: Android Studio Vector Asset Selector, Showing “add” Options

Choose the “add” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

While it feels like we keep adding action bar items, we have never added one directly to the `RosterListFragment`. All previous action 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 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_black_24dp` in the list of drawables, then click OK to accept that choice of icon.

## ADDING AND DELETING 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_add` as the resource name and “Add” as the resource value. Click OK to close the dialog and complete the configuration of this action bar item.

The `actions_roster` 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/add"
        android:icon="@drawable/ic_add_black_24dp"
        android:title="@string/menu_add"
        app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T18-Add/ToDo/app/src/main/res/menu/actions\\_roster.xml](#))

Add this `onCreate()` function to `RosterListFragment`, to indicate that this fragment wishes to participate in the action bar:

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

    setHasOptionsMenu(true)
}
```

(from [T18-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 [T18-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” action bar item, but the empty view text is more useful:

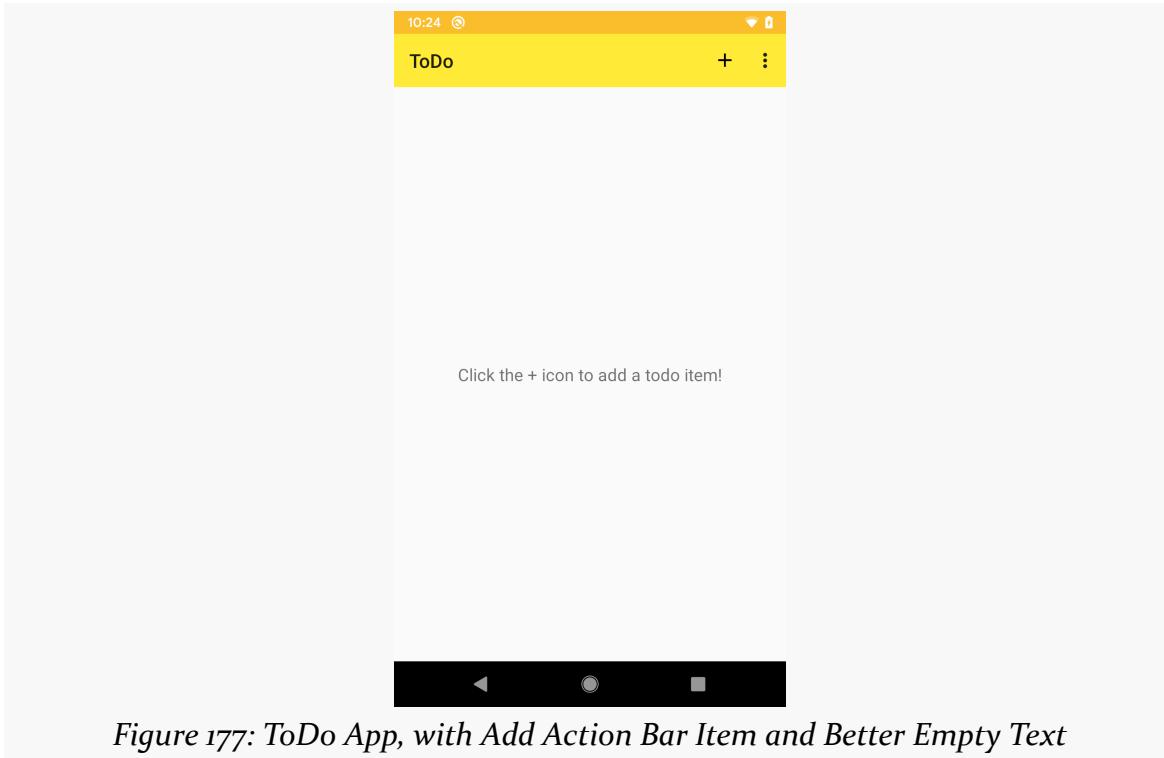


Figure 177: ToDo App, with Add Action Bar Item and Better Empty Text

## Step #4: Launching the EditFragment for Adds

Next, we need to add some logic to do some work when the user taps that “add” action 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:

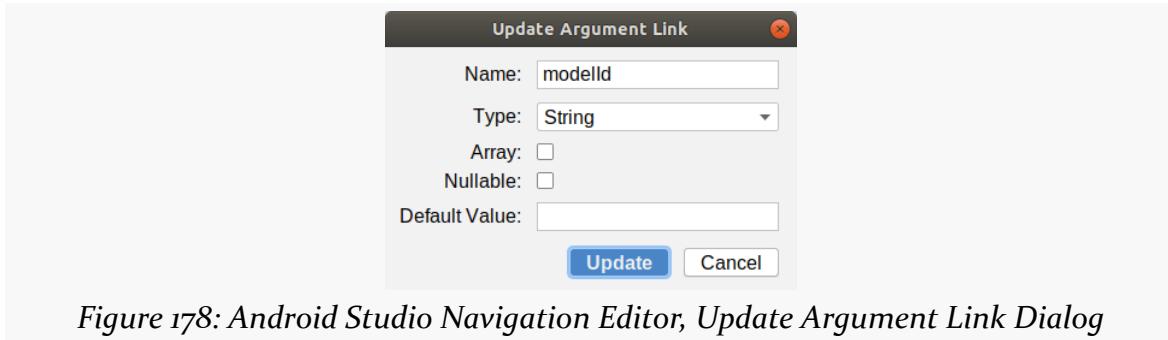


Figure 178: Android Studio Navigation Editor, Update Argument Link Dialog

Check the “Nullable” checkbox, then click Update to close the dialog.

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:

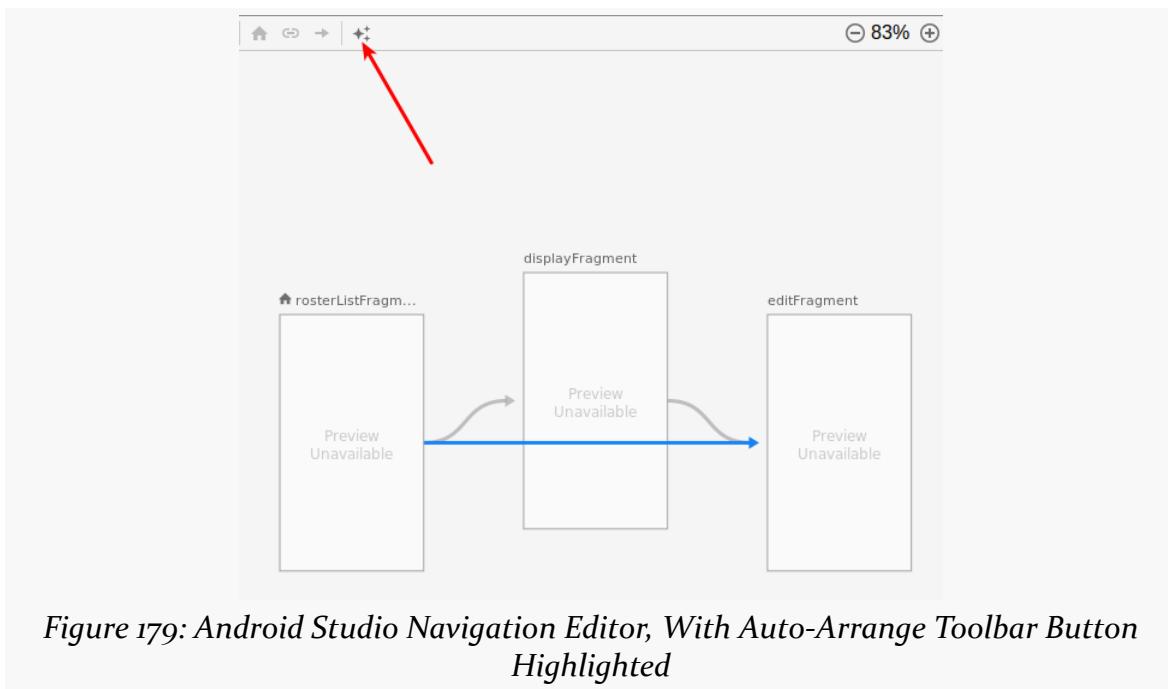


Figure 179: Android Studio Navigation Editor, With Auto-Arrange Toolbar Button Highlighted

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

## ADDING AND DELETING ITEMS

---

the “default value” field, where @null means “no, I really mean null, and not the string “null””.

Then, due to [a bug in the Navigation component](#), switch over to the “Text” sub-tab, find the `createModel` `<action>`, and change its `<argument>` to have `app:nullable="true"` and `app:argType="string"`:

```
<argument
    android:name="modelId"
    android:defaultValue="@null"
    app:argType="string"
    app:nullable="true" />
```

(from [T18-Add/ToDo/app/src/main/res/navigation/nav\\_graph.xml](#))

At this point, the `nav_graph` resource should resemble:

```
<?xml version="1.0" encoding="utf-8"?>
<navigation android:id="@+id/nav_graph"
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    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
                android:name="modelId"
                android:defaultValue="@null"
                app:argType="string"
                app:nullable="true" />
        </action>
    </fragment>
    <fragment
        android:id="@+id/displayFragment"
        android:name="com.commonsware.todo.DisplayFragment"
        android:label="@string/app_name">
        <argument
            android:name="modelId"
            app:argType="string" />
```

## ADDING AND DELETING ITEMS

```
<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"
        app:nullable="true" />
</fragment>
</navigation>
```

(from [Ti8-Add/ToDo/app/src/main/res/navigation/nav\\_graph.xml](#))

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. However, you will find that you now get a compile error in onViewCreated() of EditFragment:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    binding.model = ToDoRepository.find(args.modelId)
}
```

*Figure 18o: Android Studio Kotlin Editor, Showing a Compile Error*

Specifically, we get a Kotlin error, saying that ToDoRepository.find() expects a String, but we are passing in a String?. That is because we changed the editFragment argument to allow null, so the Safe Args code now says that modelId is a String?, not a String.

So, tweak the find() function on ToDoRepository to allow for null, accepting a String? parameter:

```
fun find(modelId: String?) = items.find { it.id == modelId }
```

(from [Ti8-Add/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

At this point, ToDoRepository should resemble:

```
package com.commonsware.todo

object ToDoRepository {
```

## ADDING AND DELETING ITEMS

---

```
var items = listOf<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 delete(model: ToDoModel) {
    items = items.filter { it.id != model.id }
}

fun find(modelId: String?) = items.find { it.id == modelId }
}
```

(from [T18-Add/ToDo/app/src/main/java/com/commonsware/todo/ToDoRepository.kt](#))

Since `id` on `ToDoModel` does not allow `null`, the `find()` call on `items` will not find a match, so our `find()` function will return `null`. Fortunately, the data binding code allows a `null` value for our `model`, so we are OK if `find()` returns `null`.

Next, add this `add()` function to `RosterListFragment`:

```
private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel())
}
```

(from [T18-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

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 [T18-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

If the user clicks the add action bar item, we call the `add()` function.

## ADDING AND DELETING ITEMS

Now, if you run the app and click the “add” action bar item, you should get an empty `EditFragment` form, showing our hints for the description and notes fields:

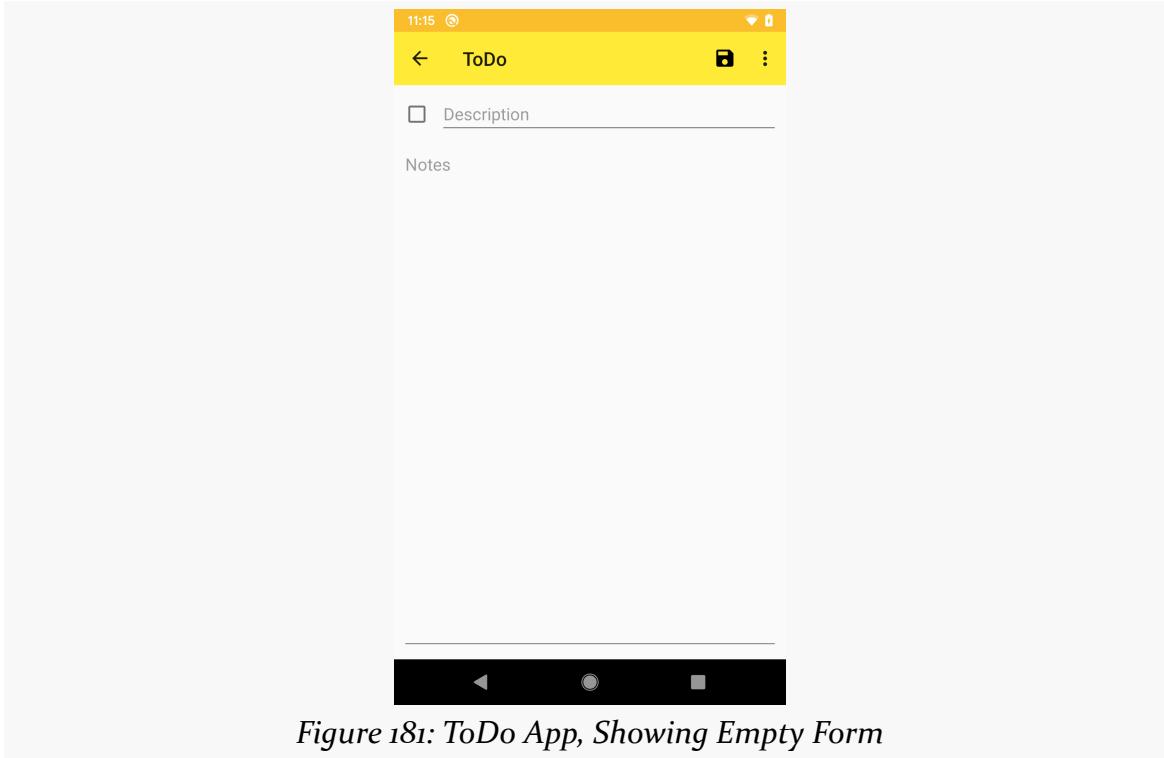


Figure 181: ToDo App, Showing Empty Form

However, clicking the “save” action bar item will cause problems, which we will fix in the next step.

## Step #5: Adjusting Our Save Logic

Our current `save()` function on `EditFragment` assumes that we bound a `ToDoModel` into our `TodoEditBinding` and does not actually do anything otherwise:

```
private fun save() {
    val edited = binding.model?.copy(
        description = binding.desc.text.toString(),
        isCompleted = binding.isCompleted.isChecked,
        notes = binding.notes.text.toString()
    )

    edited?.let { ToDoRepository.save(it) }
    navToDisplay()
}
```

## ADDING AND DELETING ITEMS

---

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

We need to handle a null model now, as that is our signal to create a new ToDoModel.

With that in mind, replace `save()` in `EditFragment` with this implementation:

```
private fun save() {
    val edited = if (binding.model == null) {
        ToDoModel(
            description = binding.desc.text.toString(),
            isCompleted = binding.isCompleted.isChecked,
            notes = binding.notes.text.toString()
        )
    } else {
        binding.model?.copy(
            description = binding.desc.text.toString(),
            isCompleted = binding.isCompleted.isChecked,
            notes = binding.notes.text.toString()
        )
    }

    edited?.let { ToDoRepository.save(it) }
    navToDisplay()
}
```

(from [T18-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

Now, we either create a new `ToDoModel` or a copy of the existing one, and that is what we pass to `ToDoRepository.save()`.

## ADDING AND DELETING ITEMS

---

If you run the sample app, click the “add” action bar item, fill in the form, and click the “save” action bar item, you wind up seeing the list of to-do items... with the empty text still visible:

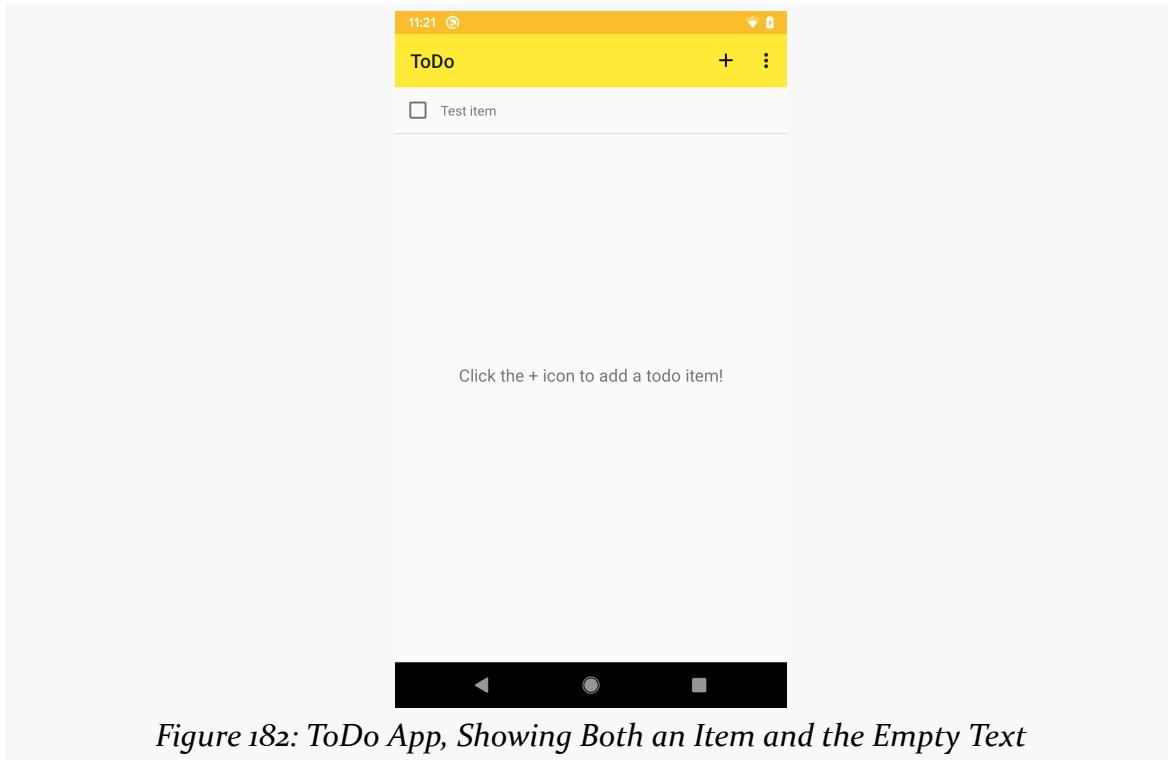


Figure 182: ToDo App, Showing Both an Item and the Empty Text

### Step #6: 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:

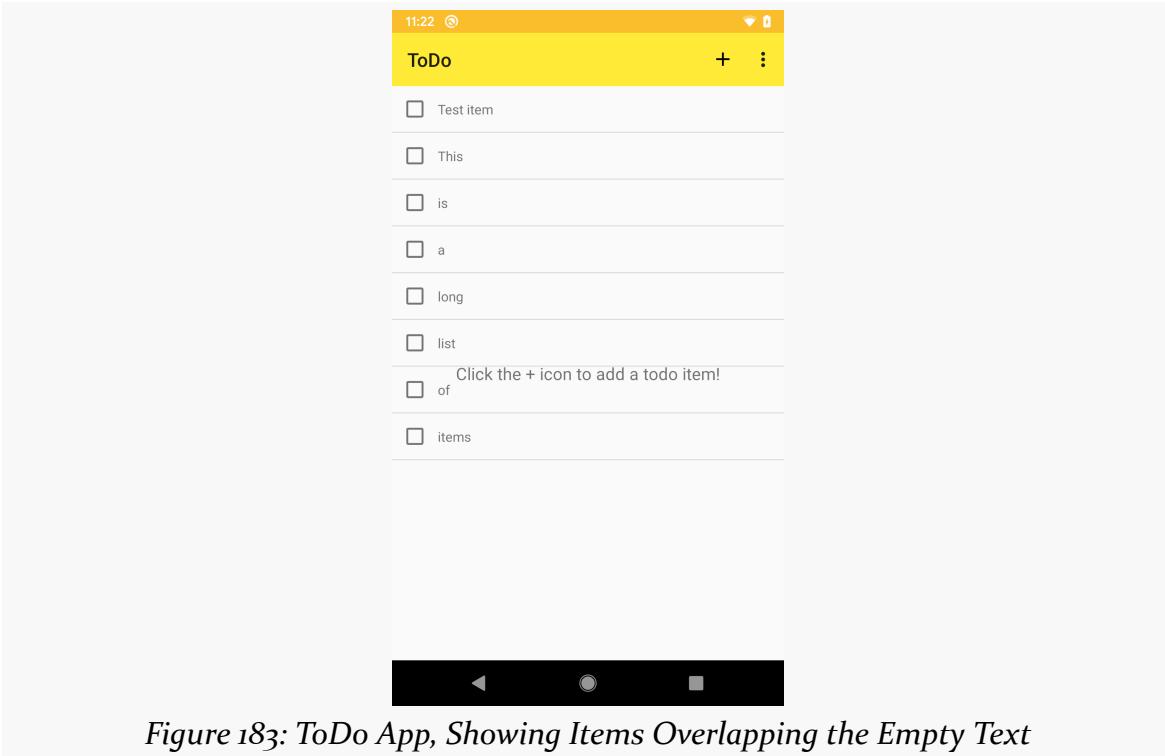


Figure 183: ToDo App, Showing Items Overlapping the Empty Text

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`:

```
empty.visibility = if (ToDoRepository.items.isEmpty()) View.VISIBLE else View.GONE  
(from Ti8-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt)
```

So, we check to see if we have any items, and if we do, we set the empty view to be GONE.

This makes the entire function become:

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

## ADDING AND DELETING ITEMS

---

```
val adapter =  
    RosterAdapter(  
        inflater = layoutInflater,  
        onCheckboxToggle = { model ->  
            ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))  
        },  
        onRowClick = { model -> display(model) })  
  
view.items.apply {  
    setAdapter(adapter)  
    layoutManager = LinearLayoutManager(context)  
  
    addItemDecoration(  
        DividerItemDecoration(  
            activity,  
            DividerItemDecoration.VERTICAL  
        )  
    )  
}  
  
adapter.submitList(ToDoRepository.items)  
empty.visibility = if (ToDoRepository.items.isEmpty()) View.VISIBLE else View.GONE  
}
```

(from [T18-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

And, at this point, `RosterListFragment` should resemble:

```
package com.commonsware.todo  
  
import android.os.Bundle  
import android.view.*  
import androidx.fragment.app.Fragment  
import androidx.navigation.fragment.findNavController  
import androidx.recyclerview.widget.DividerItemDecoration  
import androidx.recyclerview.widget.LinearLayoutManager  
import kotlinx.android.synthetic.main.todo_roster.*  
import kotlinx.android.synthetic.main.todo_roster.view.*  
  
class RosterListFragment : Fragment() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        setHasOptionsMenu(true)  
    }  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? = inflater.inflate(R.layout.todo_roster, container, false)  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
  
        val adapter =  
            RosterAdapter(  
                inflater = layoutInflater,  
                onCheckboxToggle = { model ->
```

## ADDING AND DELETING ITEMS

---

```
ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
},
onRowClick = { model -> display(model) }

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

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

adapter.submitList(ToDoRepository.items)
empty.visibility = if (ToDoRepository.items.isEmpty()) View.VISIBLE else View.GONE
}

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)
}

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

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel())
}
}
```

(from [Tr8-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

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 #7: Adding a Delete Action Bar Item

We have one more action bar item to create, this one to allow the user to delete an item. We will add that to the action bar on EditFragment, so the user can delete the item from there.

## 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 delete:

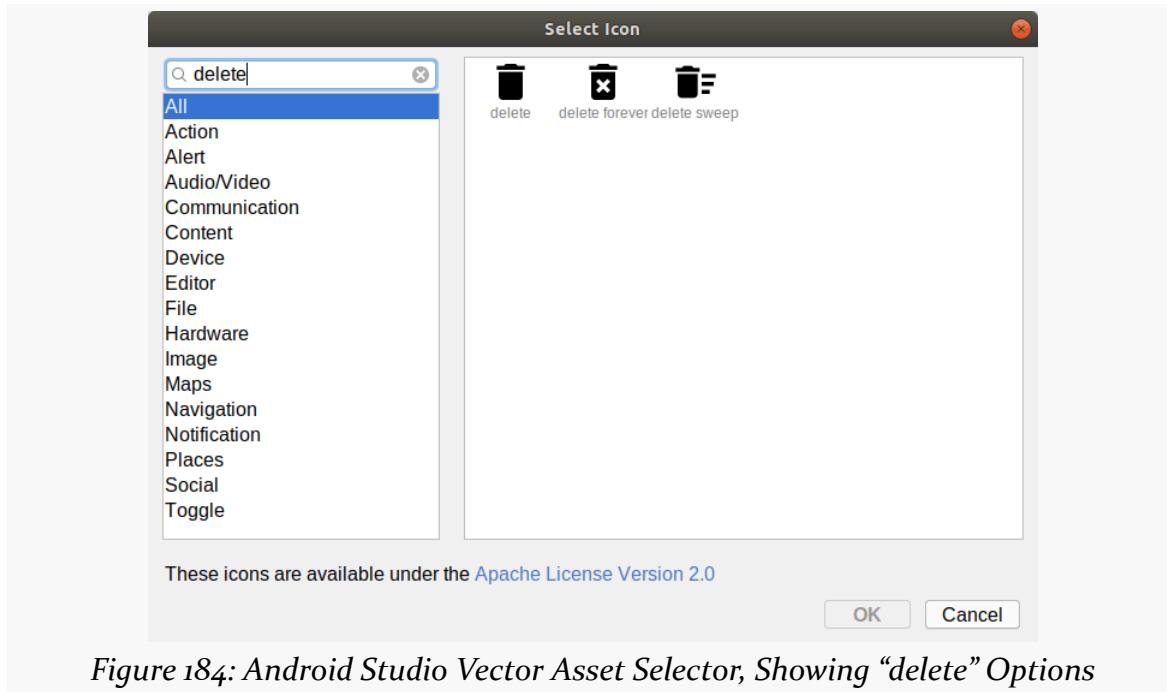


Figure 184: Android Studio Vector Asset Selector, Showing “delete” Options

Choose the “delete” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

Open `res/menu/actions_edit.xml` in the IDE. In the Design sub-tab, drag a 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_black_24dp` 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_delete` as the resource name and “Delete” as the resource value. Click OK to close the dialog, to complete the configuration of this action bar item.

The `actions_edit` menu resource should now resemble:

## ADDING AND DELETING ITEMS

```
<?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/delete"
        android:icon="@drawable/ic_delete_black_24dp"
        android:title="@string/menu_delete"
        app:showAsAction="ifRoom|withText" />
    <item
        android:id="@+id/save"
        android:icon="@drawable/ic_save_black_24dp"
        android:title="@string/menu_save"
        app:showAsAction="ifRoom|withText" />
</menu>
```

(from [T18-Add/ToDo/app/src/main/res/menu/actions\\_edit.xml](#))

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” action bar item:

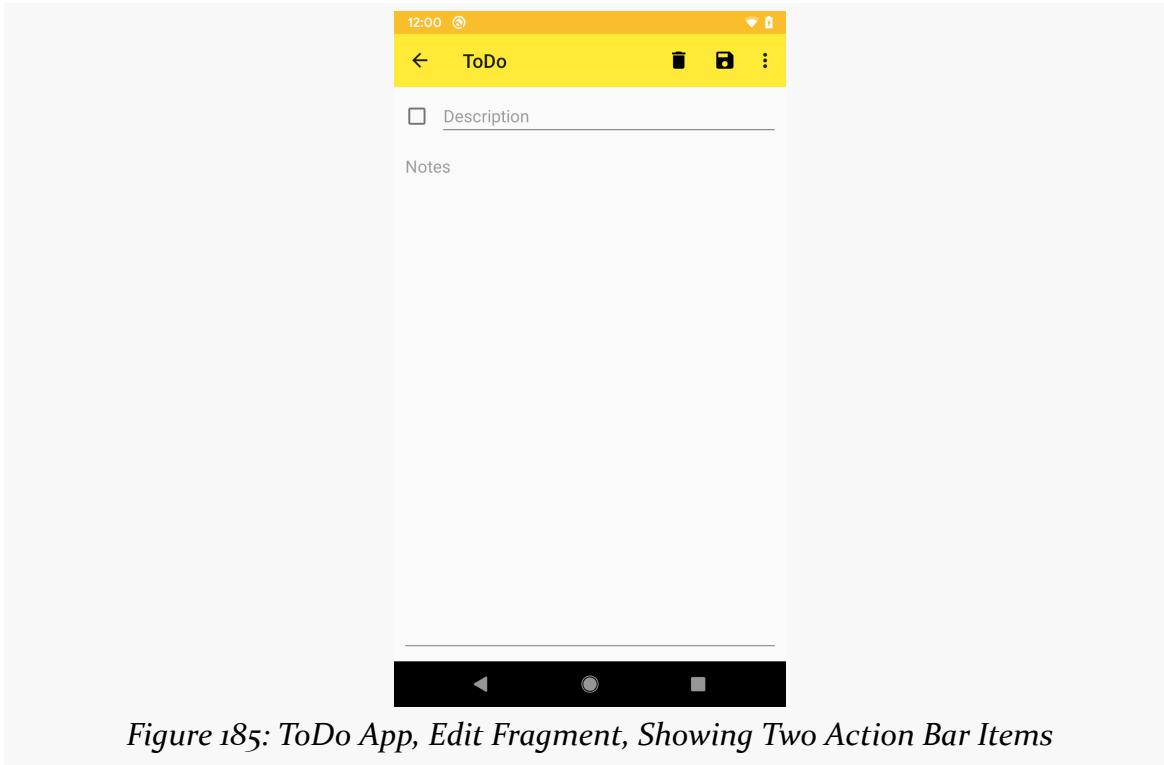


Figure 185: ToDo App, Edit Fragment, Showing Two Action Bar Items

The fact that there is a delete icon for an add operation is... disturbing. We will address that later in this tutorial.

### Step #8: Deleting the Item

Deleting the `ToDoModel` seems fairly straightforward: call `delete()` on the `ToDoRepository`, supplying the model to be deleted.

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 [T18-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() {
    binding.model?.let { ToDoRepository.delete(it) }
    navToList()
}
```

(from [T18-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

This deletes the current model in the binding, plus calls the new `navToList()` function.

Then, add another option to the `when` in `onOptionsItemSelected()` on `EditFragment` to call `delete()` if the user taps the “delete” action bar item:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.save -> { save(); return true; }
        R.id.delete -> { delete(); return true; }
```

## ADDING AND DELETING ITEMS

---

```
    }

    return super.onOptionsItemSelected(item)
}
```

(from [T18-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 action bar item, that newly-added item is deleted, and you return to an empty list.

## Step #9: Fixing the Delete-on-Add Problem

Right now, when you edit an existing to-do item, the “delete” action 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: a call to `setVisible()` 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)
}
```

(from [T18-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.

`EditFragment` overall 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
```

## ADDING AND DELETING ITEMS

---

```
import androidx.navigation.fragment.findNavController
import androidx.navigation.fragment.navArgs
import com.commonsware.todo.databinding.TodoEditBinding

class EditFragment : Fragment() {
    private lateinit var binding: TodoEditBinding
    private val args: EditFragmentArgs by navArgs()

    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?) {
        binding.model = ToDoRepository.find(args.modelId)
    }

    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() {
        val edited = if (binding.model == null) {
            ToDoModel(
                description = binding.desc.text.toString(),
                isCompleted = binding.isCompleted.isChecked,
                notes = binding.notes.text.toString()
        
```

## ADDING AND DELETING ITEMS

---

```
        )
    } else {
        binding.model?.copy(
            description = binding.desc.text.toString(),
            isCompleted = binding.isCompleted.isChecked,
            notes = binding.notes.text.toString()
        )
    }

    edited?.let { ToDoRepository.save(it) }
    navToDisplay()
}

private fun delete() {
    binding.model?.let { ToDoRepository.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 [T18-Add/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

And, if you run the app and go to add a new item, the delete icon does not appear in the action bar, but it will appear if you try to edit an existing item.

# 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/ToDoRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#)
- [app/src/main/res/drawable/ic\\_add\\_black\\_24dp.xml](#)
- [app/src/main/res/menu/actions\\_roster.xml](#)
- [app/src/main/res/drawable/ic\\_delete\\_black\\_24dp.xml](#)
- [app/src/main/res/menu/actions\\_edit.xml](#)
- [app/src/main/java/com/commonsware/todo/EditFragment.kt](#)

# **Interlude: So, What's Wrong?**

---

We're done! Right?

Right?!?

.

.

Well, OK, we're 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 a lot of 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.

## Testability

As we start adding this sort of asynchronous persistence, we really ought to write some tests. It can be tricky to get all of this working properly. Plus, “testing by wandering around” our UI can miss bugs, because humans get tired. Ideally, we would have some automated tests that we can run with the click of a button (or two), and those tests can confirm that what we wrote does what we want, despite any changes that we made.

However, before we can get into writing tests, we need to think about testability itself.

Right now, our UI and our repository are fairly tightly coupled. Our fragments directly call functions on the repository singleton. As a result, the only way to test the fragments requires us to work with the real repository. That may be OK for some tests, but it may make it difficult to test certain scenarios. For example, when we persist data, things may go wrong (out of disk space, network connectivity failure, etc.), triggering an exception. Ideally, we could test our fragments with some sort of

---

## INTERLUDE: SO, WHAT'S WRONG?

---

mock implementation of the repository, so we can have that mock throw exceptions, so we can see if our fragments handle those exceptions properly.

Without some amount of separation between the “front end” (activity and fragments) and “back end” (repository and eventual data storage code), testing each in isolation will be troublesome. So, we should try to address that.

## Configuration Changes

Asynchronous work will also start to give us problems with respect to configuration changes.

In Android, an app will undergo a “configuration change” when something about the environment has changed in ways that cause Android to think that we may need other resources. The most common configuration change is a screen rotation, as we may need different layout resources to make effective use of a landscape screen instead of a portrait screen. However, there are many other configuration changes, including:

- Changes in “screen” size, particularly on Chrome OS’ freeform multi-window mode or Android 7.0’s split-screen feature
- Changes in the user’s chosen device language(s)
- Changes in ambient light, perhaps triggering a switch “night mode”

Android’s default behavior on a configuration change is to destroy and recreate our activity and fragments, so we load the correct resources. We need to make sure that we do not lose data or have to re-load it from storage just because the user switched from portrait to landscape.

We also need to take into account how asynchronous operations work with respect to configuration changes. For example, suppose the user clicks our “save” action bar item. While we are saving the data, the user rotates the screen, thereby replacing our activity and its `EditFragment`. We still need to know when the save is complete or if there is some sort of error... even though the fragment that *requested* the data to be saved is different from the current fragment.

## Messy UI Architecture

Right now, our fragments are doing everything:

- Asking for the data from the repository

## INTERLUDE: SO, WHAT'S WRONG?

---

- Binding that data to the UI
- Processing events from the UI (menus, CheckBox toggles, etc.)

That on its own might not be all that bad. But by the time we start adding in all of the above changes, our fragments might get too complicated... and this is a fairly simple app. There are *much* more elaborate apps out there, with more complex user interfaces and more complicated data storage scenarios.

Part of the way that developers deal with complexity is to divide the code, delegating different bits of work to different classes dedicated for that work. This helps keep each class relatively compact, but it then requires us to coordinate the communications between these classes.

There are various UI architecture patterns that describe how to divide the code and how communications between them should work. These tend to get named using abbreviations like “MVP” and “MVVM”. Ideally, our app would adopt some sort of consistent UI architecture.

## 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-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.

## INTERLUDE: SO, WHAT'S WRONG?

---

### 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.

### Testability: Dependency Injection via Koin

Many apps use “dependency injection” or “service locator” patterns for connecting different pieces of the app. For example, rather than our fragments directly talking to `ToDoRepository`, we could:

- Have an interface representing our repository’s API
- Have our fragments ask a dependency injection system to get an implementation of that interface
- For the real app, have an implementation of that interface be used that works with Room to save the data
- For tests, inject mock implementations of that interface that can simulate test scenarios, such as running out of disk space

Specifically, we will use [Koin](#), a popular lightweight implementation of the “dependency injection”/“service locator” pattern for Kotlin.

### Configuration Changes: ViewModel/LiveData

The typical solution nowadays for keeping track of data across configuration changes is to use `ViewModel` from the Jetpack components. The contents of a `ViewModel` are retained across configuration changes, so our portrait fragment and a later landscape fragment will share a common `ViewModel` and the data that it holds.

A popular way for a `ViewModel` to deliver data updates and asynchronous results to activities and fragments is through `LiveData`. The big advantage of `LiveData` is that it is aware that activities and fragments might come and go, whereas something like Kotlin coroutines do not. So, while we will use coroutines for some of our asynchronous work, we will deliver the results of that work to the UI via `LiveData`. And, for loading data from Room, we will use `LiveData` directly, as that will help

## **INTERLUDE: SO, WHAT'S WRONG?**

---

ensure that things like `RosterListFragment` find out about changes to the data made elsewhere in the app.

## **UI Architecture: Unidirectional Data Flow**

One of the up-and-coming approaches for UI architectures is called “unidirectional data flow”. In short, it means that data flows through a loop in our app:

- The user taps on something on the screen, which...
- Causes us to load some data from the database, which...
- Delivers data for our UI to show on the screen, which...
- Gives the user other things to tap on, and so on

This approach is sometimes called “model-view-intent” (MVI) or “what Redux does in Web apps”. We will adopt a variation on this approach for our fragments.

---

## **Phase Two: Asynchronous Architecture**

---

Licensed solely for use by Patrocinio Rodriguez

# **Injecting 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.

However, right now, our UI layer is very dependent on there being a `ToDoRepository` object that can handle that work.

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 injection” 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 fragments should not be declaring that a 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.

## INJECTING OUR DEPENDENCIES

---

Part of the problem with dependency injection in Android is that the historically dominant solution — Dagger — is very complex and has difficult-to-understand documentation.

Kotlin opened up new opportunities for simplifying dependency injection. One of the more popular Kotlin dependency injection 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.

So, in this chapter, we will integrate Koin and have our fragments get their ToDoRepository from Koin instead of working with some singleton directly.

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:

```
ext {  
    koin_version = "1.0.2"  
}
```

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

This is equivalent to:

```
ext.koin_version = "1.0.2"
```

## INJECTING OUR DEPENDENCIES

---

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.

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

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.3.31'
    ext.nav_version = '2.0.0'

    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.4.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
        google()
        jcenter()

    }
}

task clean(type: Delete) {
    delete rootProject.buildDir
}

ext {
    koin_version = "1.0.2"
}
```

(from [T19-DI/ToDo/build.gradle](#))

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

```
implementation "org.koin:koin-core:$koin_version"
implementation "org.koin:koin-android:$koin_version"
```

(from [T19-DI/ToDo/app/build.gradle](#))

These pull in two Koin dependencies: the core engine (`koin-core`) and the Koin for Android library (`koin-android`).

We will add a third Koin dependency in a later tutorial, but for now, these are all

## INJECTING OUR DEPENDENCIES

---

that we need.

At this point, your app/build.gradle file should resemble:

```
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply plugin: 'kotlin-android-extensions'
apply plugin: 'kotlin-kapt'
apply plugin: 'androidx.navigation.safeargs.kotlin'

android {
    compileSdkVersion 28

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

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

    androidExtensions {
        experimental = true
    }

    dataBinding {
        enabled = true
    }
}

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

(from [T19-DI/ToDo/app/build.gradle](#))

Android Studio should be asking you to “Sync Now” in a yellow 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” in the “Kind” drop-down list. Click OK, and you will get an empty ToDoApp class.

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

```
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/AppTheme">
```

(from [T19-DI/ToDo/app/src/main/AndroidManifest.xml](#))

## INJECTING OUR DEPENDENCIES

---

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

At this point, the manifest should look something like:

```
<?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: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/AppTheme">
    <activity android:name=".AboutActivity"></activity>
    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

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

## Step #3: Converting Our Repository

Right now, ToDoRepository is a Kotlin object, meaning that we are stuck with this one implementation. To make effective use of Koin for testing, we are going to want to replace ToDoRepository with a test (“mock”) implementation for some tests, and for that, we need ToDoRepository to be a class.

So, change the object keyword to class on the ToDoRepository declaration:

## INJECTING OUR DEPENDENCIES

---

```
class ToDoRepository {
```

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

This means your entire ToDoRepository should be:

```
package com.commonsware.todo

class ToDoRepository {
    var items = listOf<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 delete(model: ToDoModel) {
        items = items.filter { it.id != model.id }
    }

    fun find(modelId: String?) = items.find { it.id == modelId }
}
```

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

This broke all of our code that was using the ToDoRepository singleton object, and we will fix that code later in this tutorial.

## Step #4: Defining Our Module

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

Back in ToDoApp, add this property:

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

(from [Tig-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:

## INJECTING OUR DEPENDENCIES

---

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

(from [Tig-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 injection. 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 injection 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(this, listOf(koinModule))
}
```

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

`startKoin()` is another extension function that will need to be imported:

```
import org.koin.android.ext.android.startKoin
```

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

As the name suggests, `startKoin()` starts the Koin dependency injection engine. We can supply a list of modules for Koin to use. Since we only have one module, we wrap it in a list using the Kotlin `listOf()` global function, then pass that to `startKoin()`.

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.

At this point, `ToDoApp` should look like:

```
package com.commonsware.todo

import android.app.Application
import org.koin.android.ext.android.startKoin
```

## INJECTING OUR DEPENDENCIES

---

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

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

    override fun onCreate() {
        super.onCreate()

        startKoin(this, listOf(koinModule))
    }
}
```

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

## Step #5: Injecting Our Repository

All three of our fragments are using `ToDoRepository`. Right now, they try to do so by calling functions on a `ToDoRepository` object. That no longer works, as `ToDoRepository` is now a class. We need to make some minor adjustments to the fragments to have them get and use a `ToDoRepository` object supplied by Koin.

First, we can fix `RosterListFragment`. Add this property to it:

```
private val repo: ToDoRepository by inject()
```

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Here, `inject()` is another extension function that will need to be imported:

```
import org.koin.android.ext.android.inject
```

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

`inject()` is available for activities, fragments, and some other Android classes, and other versions of injection are available for other types of objects. Here, `inject()` sees that we are looking for a `ToDoRepository`, and it asks Koin to supply one. The `by inject()` syntax indicates that what `inject()` creating is a “delegate” for our Kotlin property. When we go to access this `repo` property, in reality, the delegate will handle that work for us.

Now, we can change the three lines that referred to the `ToDoRepository` singleton and replace them with references to `repo`. As it turns out, in this fragment, all three

## INJECTING OUR DEPENDENCIES

---

of those references are in onViewCreated():

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

    val adapter =
        RosterAdapter(
            inflater = layoutInflater,
            onCheckboxToggle = { model ->
                ToDoRepository.save(model.copy(isCompleted = !model.isCompleted))
            },
            onRowClick = { model -> display(model) })

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

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

    adapter.submitList(ToDoRepository.items)
    empty.visibility = if (ToDoRepository.items.isEmpty()) View.VISIBLE else View.GONE
}
```

(from [Ti8-Add/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Replace that with:

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

    val adapter =
        RosterAdapter(
            inflater = layoutInflater,
            onCheckboxToggle = { model ->
                repo.save(model.copy(isCompleted = !model.isCompleted))
            },
            onRowClick = { model -> display(model) })

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

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

## INJECTING OUR DEPENDENCIES

---

```
)  
}  
  
    adapter.submitList(repo.items)  
    empty.visibility = if (repo.items.isEmpty()) View.VISIBLE else View.GONE  
}
```

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

This is the same as before, just with ToDoRepository replaced with repo.

Now, our RosterListFragment is no longer locked into using a particular instance of ToDoRepository — it will use whatever instance Koin gives us.

At this point, RosterListFragment should look like:

```
package com.commonsware.todo  
  
import android.os.Bundle  
import android.view.*  
import androidx.fragment.app.Fragment  
import androidx.navigation.fragment.findNavController  
import androidx.recyclerview.widget.DividerItemDecoration  
import androidx.recyclerview.widget.LinearLayoutManager  
import kotlinx.android.synthetic.main.todo_roster.*  
import kotlinx.android.synthetic.main.todo_roster.view.*  
import org.koin.android.ext.android.inject  
  
class RosterListFragment : Fragment() {  
    private val repo: ToDoRepository by inject()  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        setHasOptionsMenu(true)  
    }  
  
    override fun onCreateView(  
        inflater: LayoutInflater,  
        container: ViewGroup?,  
        savedInstanceState: Bundle?  
    ): View? = inflater.inflate(R.layout.todo_roster, container, false)  
  
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
        super.onViewCreated(view, savedInstanceState)  
  
        val adapter =  
            RosterAdapter(  
                inflater = layoutInflater,  
                onCheckboxToggle = { model ->  
                    repo.save(model.copy(isCompleted = !model.isCompleted))  
                },  
                onRowClick = { model -> display(model) })  
  
        view.items.apply {
```

## INJECTING OUR DEPENDENCIES

---

```
setAdapter(adapter)
layoutManager = LinearLayoutManager(context)

addItemDecoration(
    DividerItemDecoration(
        activity,
        DividerItemDecoration.VERTICAL
    )
)

adapter.submitList(repo.items)
empty.visibility = if (repo.items.isEmpty()) View.VISIBLE else View.GONE
}

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)
}

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

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel())
}
}
```

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/RosterListFragment.kt](#))

Next, add the same repo declaration to DisplayFragment:

```
private val repo: ToDoRepository by inject()
```

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

Then change onViewCreated() in DisplayFragment to use repo instead of ToDoRepository:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    binding.model = repo.find(args.modelId)
}
```

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

## INJECTING OUR DEPENDENCIES

---

DisplayFragment should now resemble:

```
package com.commonsware.todo

import android.os.Bundle
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.android.ext.android.inject

class DisplayFragment : Fragment() {
    private val args: DisplayFragmentArgs by navArgs()
    private lateinit var binding: TodoDisplayBinding
    private val repo: ToDoRepository by inject()

    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?) {
        binding.model = repo.find(args.modelId)
    }

    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 [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/DisplayFragment.kt](#))

Next, add the same repo declaration to EditFragment:

## INJECTING OUR DEPENDENCIES

```
private val repo: ToDoRepository by inject()
```

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

We have three references to `ToDoRepository` to fix, scattered among three different functions. So, let's use search-and-replace.

While in the `EditFragment` editor, press `Ctrl-R` (or `Option-R` on macOS), or choose “Edit” > “Find” > “Replace” from the Android Studio main menu. This will open a small panel at the top of the editor:

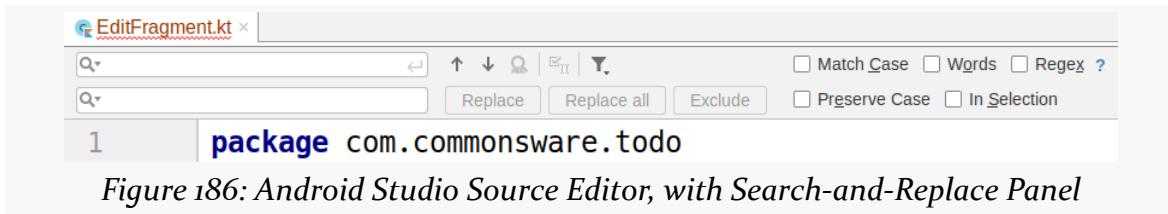


Figure 186: Android Studio Source Editor, with Search-and-Replace Panel

The top field with the magnifying glass represents the search expression, and the bottom field with the magnifying glass represents the text to use as a replacement. This tool supports case matching, regular expressions, and all sorts of other features... but we just need a simple search-and-replace here.

So, fill in `ToDoRepository` in the search field and `repo` in the replace field:

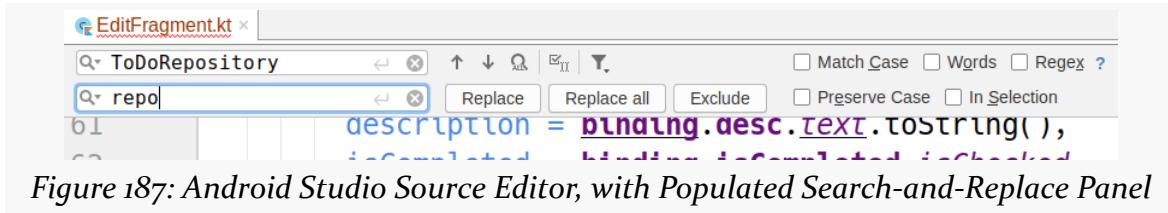


Figure 187: Android Studio Source Editor, with Populated Search-and-Replace Panel

One of the `ToDoRepository` occurrences should now be highlighted. If it is the one that we just added, in the `repo` declaration, skip over it by clicking the down-arrow button to the right of the search field. Once one of the other `ToDoRepository` references is highlighted, click the “Replace” button to replace it with `repo` and skip to the next reference. Do this until all three of the `ToDoRepository` references that we want to be replaced with `repo` have been replaced, leaving just the one in the `repo` declaration line.

In other words, change `EditFragment` to look like:

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

## INJECTING OUR DEPENDENCIES

---

```
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.android.ext.android.inject

class EditFragment : Fragment() {
    private lateinit var binding: TodoEditBinding
    private val args: EditFragmentArgs by navArgs()
    private val repo: ToDoRepository by inject()

    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?) {
        binding.model = repo.find(args.modelId)
    }

    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)
    }
}
```

## INJECTING OUR DEPENDENCIES

---

```
}

private fun save() {
    val edited = if (binding.model == null) {
        ToDoModel(
            description = binding.desc.text.toString(),
            isCompleted = binding.isCompleted.isChecked,
            notes = binding.notes.text.toString()
        )
    } else {
        binding.model?.copy(
            description = binding.desc.text.toString(),
            isCompleted = binding.isCompleted.isChecked,
            notes = binding.notes.text.toString()
        )
    }
}

edited?.let { repo.save(it) }
navToDisplay()
}

private fun delete() {
    binding.model?.let { repo.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
        )
    }
}
}
```

## INJECTING OUR DEPENDENCIES

---

(from [T19-DI/ToDo/app/src/main/java/com/commonsware/todo/EditFragment.kt](#))

When you are done, pressing `Esc` will close up the search-and-replace panel to get it out of your way.

At this point, if you run the app, you should see... absolutely no changes. That will be a common theme for the next few tutorials: we are changing some of the implementation details, but we are not changing things that the user will see.

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

Licensed solely for use by Patrocinio Rodriguez

# 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 click OK.

Do that again, but this time fill in `ui.display`. This creates a package (`ui`) and a sub-package (`display`) in one shot.

## REFACTORING OUR CODE

---

Do that again, but this time fill in ui.edit. Once you type in the ui part, you will get an error indicating that you already have that package:

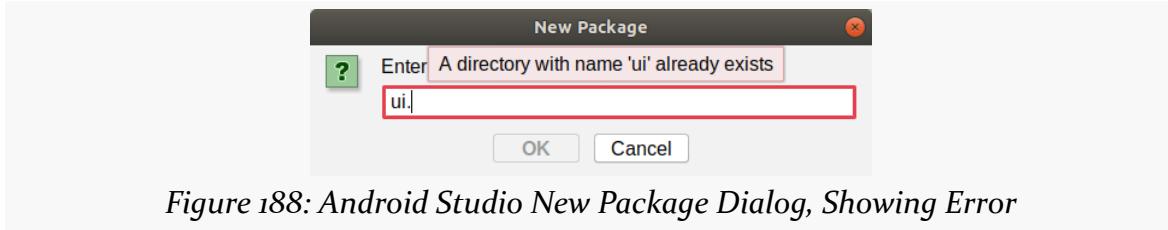


Figure 188: Android Studio New Package Dialog, Showing Error

But, if you keep typing in the rest, since ui.edit does not exist, the error goes away, and you can create the package.

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 of that, a peer of the display and edit ones.

Finally, do that again, but this time add a util package to com.commonsware.todo.ui.

## REFACTORING OUR CODE

At this point, your project tree should resemble:

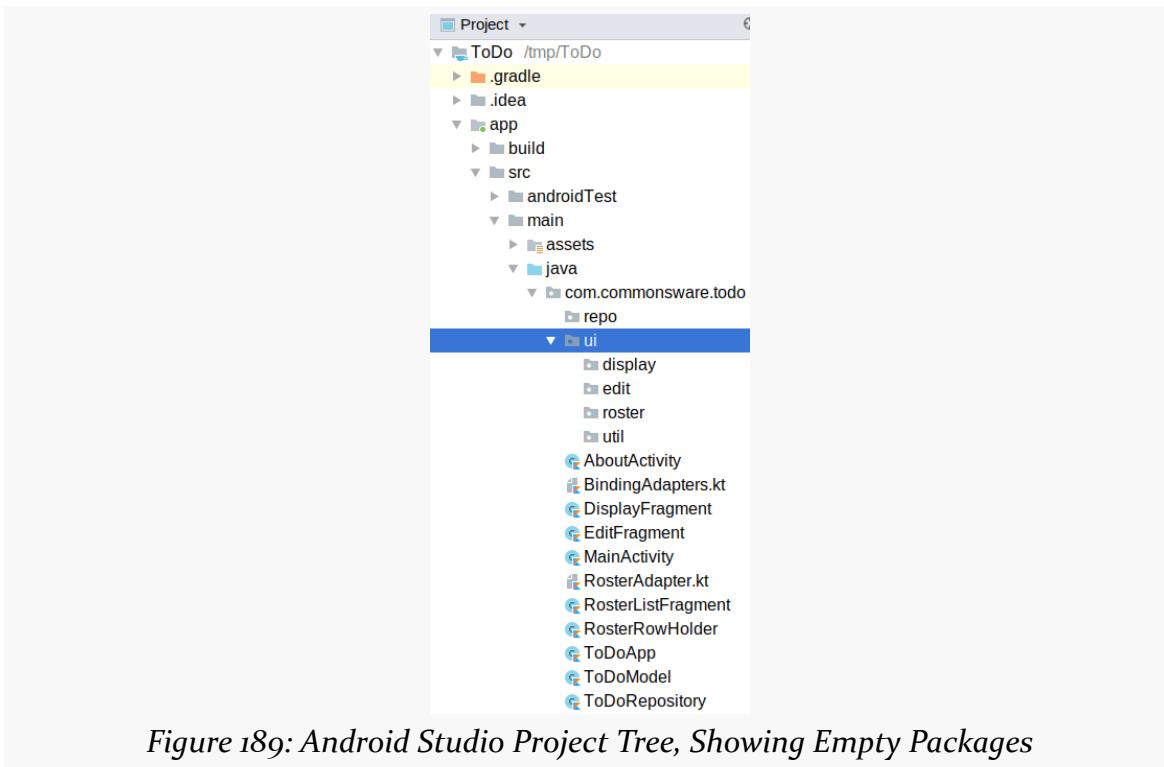


Figure 189: Android Studio Project Tree, Showing Empty Packages

## 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, data binding references, the manifest, and related stuff.

Start by dragging `ToDoRepository` and `ToDoModel` to the `repo` package. You can use the `Ctrl` key while clicking on classes in the project tree to select more than one class. You can also use the `Shift` key to select a range of classes. Or, move them one at a time, if you prefer.

## REFACTORING OUR CODE

When you drop the class(es) in the repo package, a “Move” dialog will appear:

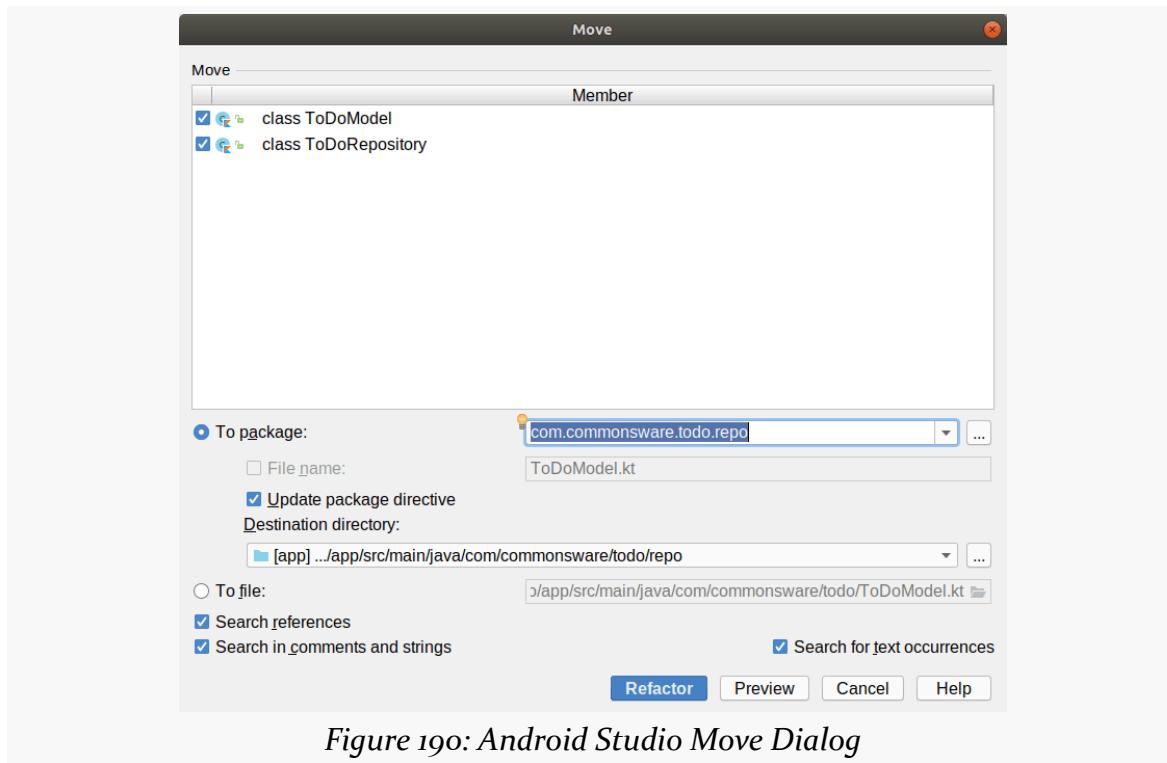


Figure 190: Android Studio Move Dialog

This confirms that we are going to move these classes to the designated package. The checkboxes towards the bottom of the dialog indicate where Android Studio will look for these classes, in order to change that code to point to the new package (if needed).

## REFACTORING OUR CODE

Click “Refactor”, and in a moment, your classes will now be in the `repo` package:

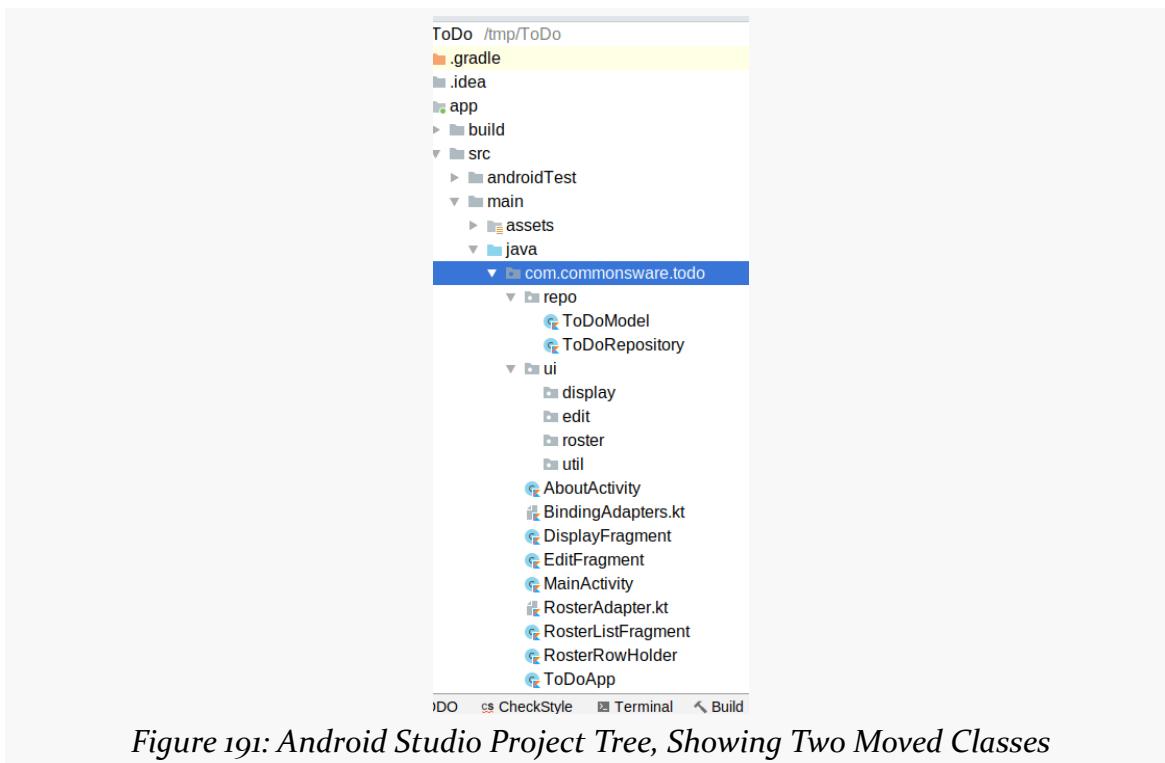


Figure 191: Android Studio Project Tree, Showing Two Moved Classes

Next, move `RosterAdapter`, `RosterListFragment`, and `RosterRowHolder` to the `roster` package. This time, if you select all three and try moving them, you get a different “Move” dialog:

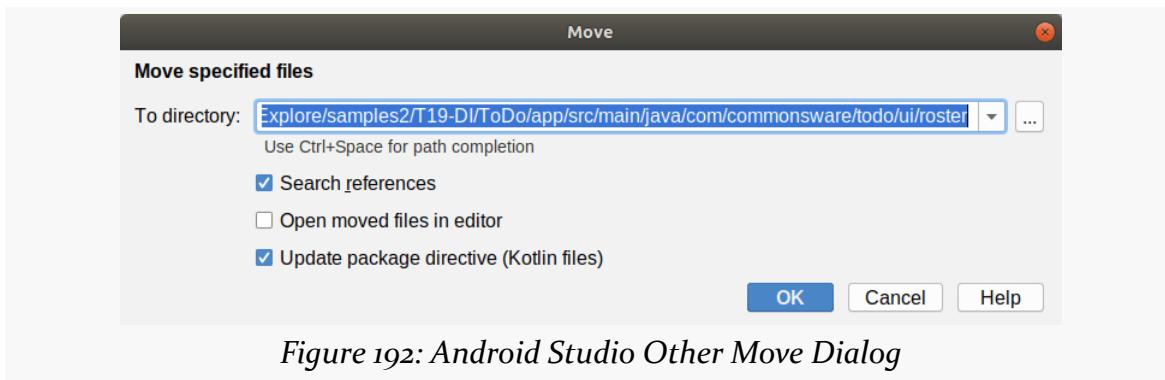


Figure 192: Android Studio Other Move Dialog

This is because one of our chosen files — `RosterAdapter` — contains more than just a single class. For some reason, that causes Android Studio to treat the move differently.

## REFACTORING OUR CODE

---

Click the “OK” button to complete moving those classes into roster.

Then, move some of the remaining classes to the new packages:

Class(es)	Package
AboutActivity, MainActivity	ui
BindingAdapters	util
DisplayFragment	display
EditFragment	edit

In the end, only ToDoApp should be directly in com.commonware.todo, with everything else in one of the sub-packages off of com.commonware.todo.

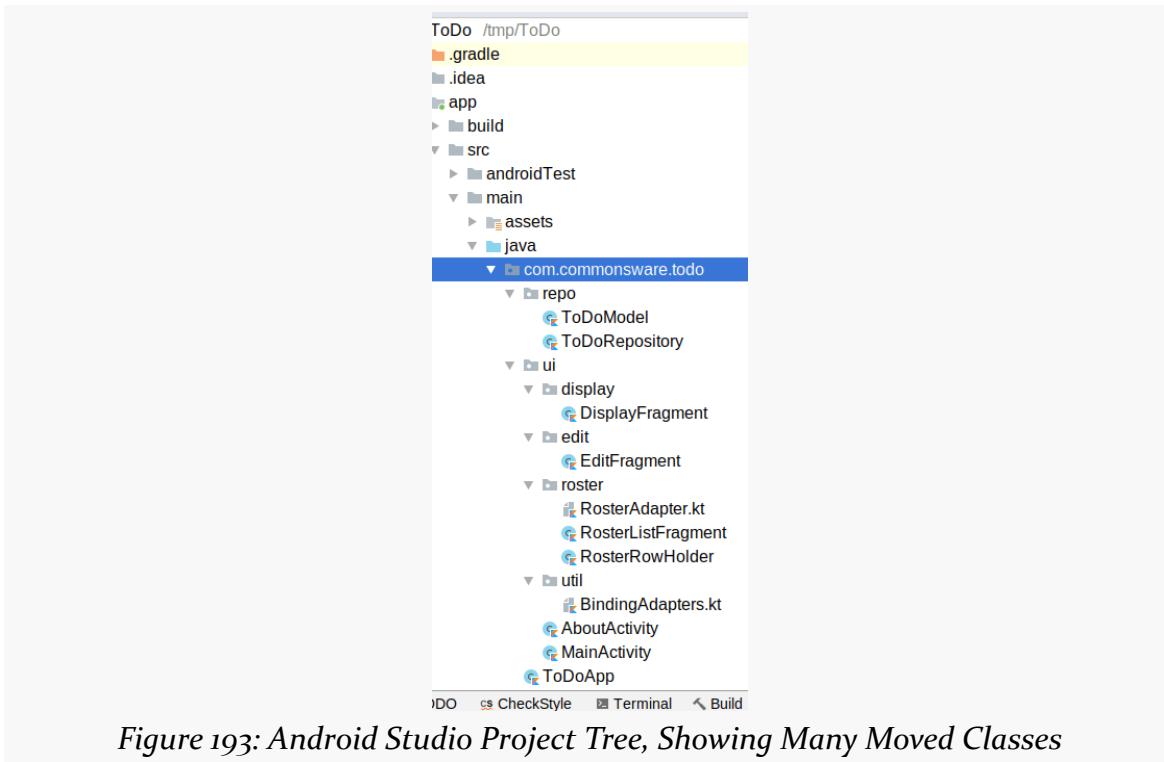


Figure 193: Android Studio Project Tree, Showing Many Moved Classes

However, if you try building the app, you will find that you have some compile errors. That is because the refactoring work messes up the classes related to

## REFACTORING OUR CODE

---

navigation, adding import statements for the old locations of those classes.

To fix this, we just need to remove the erroneous import statements. Remove these two from `DisplayFragment`:

```
import com.commonsware.todo.DisplayFragmentArgs  
import com.commonsware.todo.DisplayFragmentDirections
```

Then remove this one from `EditFragment`:

```
import com.commonsware.todo.EditFragmentArgs
```

Finally, remove this one from `RosterListFragment`:

```
import com.commonsware.todo.RosterListFragmentDirections
```

Now, if you run the app, everything should work as it did before the refactoring.

## 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.

Licensed solely for use by Patrocinio Rodriguez

# Incorporating a ViewModel

---

The Architecture Components library 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 it works fairly nicely, which is why we will use it here.

So, in this tutorial, we will set up a basic `ViewModel` for `RosterListFragment`. Later in the book, we will add `ViewModel` implementations for the other two fragments.



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: Adding the Dependencies

As it turns out, Koin has special support for the AndroidX ViewModel class. Not only can we inject things like ToDoRepository, but we can inject ViewModel implementations into our fragments. Once again, this helps with testing, as we may want to test our fragments with mock ViewModel implementations for some tests.

However, that requires another Koin dependency.

So, in app/build.gradle, add this line to the dependencies closure:

```
implementation "org.koin:koin-androidx-viewmodel:$koin_version"
```

(from [T21-ViewModel/ToDo/app/build.gradle](#))

The entire dependencies closure should now look like:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.fragment:fragment-ktx:1.0.0'  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "org.koin:koin-core:$koin_version"  
    implementation "org.koin:koin-android:$koin_version"  
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test:runner:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'  
}
```

(from [T21-ViewModel/ToDo/app/build.gradle](#))

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

## Step #2: Creating a Stub ViewModel

There are two base classes that we can choose from: `ViewModel` and `AndroidViewModel`. The latter is for cases where we need a Context, that “god object” in Android that provides access to all sorts of Android-specific things. For our case, `ViewModel` is all that we need.

So, once again, we create a new Kotlin class. However, we now have our new set of sub-packages to consider. The `ViewModel` that we will be creating is for `RosterListFragment`, so it makes sense to put that in the `roster` package.

Right-click over the `com.commonsware.todo.ui.roster` 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 click OK to create the empty `RosterMotor` class.

(the reason why this is being called a “motor” will be explained more in [an upcoming tutorial](#))

Then, modify `RosterMotor` to extend from `ViewModel`:

```
package com.commonsware.todo.ui.roster

import androidx.lifecycle.ViewModel

class RosterMotor : ViewModel()
```

## Step #3: 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

## INCORPORATING A VIEWMODEL

---

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.ui.roster

import androidx.lifecycle.ViewModel
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository

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

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

(from [T21-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/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
- Expose the save() function, passing it along to the repository

Both of those will wind up changing a fair bit later on, as we start to move towards an asynchronous API, but they will suffice for now.

## Step #4: 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()) }
}
```

## INCORPORATING A VIEWMODEL

---

(from [T21-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”.

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.

At this point, `ToDoApp` overall should resemble:

```
package com.commonsware.todo

import android.app.Application
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.ui.roster.RosterMotor
import org.koin.android.ext.android.startKoin
import org.koin.androidx.viewmodel.ext.koin.viewModel
import org.koin.dsl.module.module

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

    override fun onCreate() {
        super.onCreate()

        startKoin(this, listOf(koinModule))
    }
}
```

(from [T21-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

## Step #5: Injecting the Motor

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

Right now, we have a `repo` property in `RosterListFragment`:

## INCORPORATING A VIEWMODEL

---

```
private val repo: ToDoRepository by inject()
```

(from [T2o-Refactor/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Replace that with a `motor` property:

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

(from [T2i-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Before, we used an `inject()` Koin extension function to get our `ToDoRepository`. `viewModel()` is another Koin extension function, one specifically designed to get AndroidX `ViewModel` objects from Koin. 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.

`onViewCreated()` has three references to `repo` that now can get replaced with `motor`:

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

    val adapter =
        RosterAdapter(
            inflater = layoutInflater,
            onCheckboxToggle = { model ->
                motor.save(model.copy(isCompleted = !model.isCompleted))
            },
            onRowClick = { model -> display(model) }

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

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

## INCORPORATING A VIEWMODEL

---

```
        )
    }
}

adapter.submitList(motor.items)
empty.visibility = if (motor.items.isEmpty()) View.VISIBLE else View.GONE
}
```

(from [T2i-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

So the RosterListFragment, when this work is done, should look like:

```
package com.commonsware.todo.ui.roster

import android.os.Bundle
import android.view.*
import androidx.fragment.app.Fragment
import androidx.navigation.fragment.findNavController
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import com.commonsware.todo.R
import com.commonsware.todo.repo.ToDoModel
import kotlinx.android.synthetic.main.todo_roster.*
import kotlinx.android.synthetic.main.todo_roster.view.*
import org.koin.androidx.viewmodel.ext.android.viewModel

class RosterListFragment : Fragment() {
    private val motor: RosterMotor by viewModel()

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

        setHasOptionsMenu(true)
    }

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

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

        val adapter =
            RosterAdapter(
                inflater = layoutInflater,
                onCheckboxToggle = { model ->
```

## INCORPORATING A VIEWMODEL

---

```
        motor.save(model.copy(isCompleted = !model.isCompleted))
    },
    onRowClick = { model -> display(model) }

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

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

adapter.submitList(motor.items)
empty.visibility = if (motor.items.isEmpty()) View.VISIBLE else View.GONE
}

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)
}

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

private fun add() {
    findNavController().navigate(RosterListFragmentDirections.createModel())
}
}
```

(from [T21-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

## INCORPORATING A VIEWMODEL

---

If you compile and run the app, it might seem like it works. However, there is one crucial flaw: when we save or delete a `ToDoModel` in `EditFragment`, `RosterListFragment` does not change to reflect the changed list of model objects. Instead, the list remains as it was. Since we are starting with an empty list, that means it appears as though we cannot create new to-do items.

The reason is that the `items` property of `RosterMotor` is holding onto the `items` list of the `ToDoRepository` *at the time the RosterMotor instance was created*. So, even though `ToDoRepository` is updating its list, `RosterMotor` does not know about that. Previously, when `RosterListFragment` was accessing `ToDoRepository` directly, we did not have this problem.

One fix would be to replace the `items` property with a `getItems()` function:

```
fun getItems() = repo.items
```

Each call to `getItems()` would get the now-current list from the repository, and this problem would go away.

Right now, with our list of to-do items being held in memory, this solution would be fine. However, we do not want to be constantly querying a database or making server calls when the data has not changed. What we really want is for `RosterMotor` to hold onto the list of items but have changes to that list *pushed* to it by the repository. The repository knows when the data changes, after all. That way, the motor does not have to keep asking the repository for the items — it asks once in the beginning and then gets updates delivered to it as needed. This will minimize the number of times that we need to query the database or connect to the server to get the items.

Our [next tutorial](#) will set that up, using what is known as `LiveData`.

## 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/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# Making Our Repository Live

---

When the device undergoes a configuration change — such as the user rotating their device — visible activities and fragments get destroyed and recreated on the fly. This allows them to rebuild their UI for the new configuration, such as switching to different layouts optimized for the new screen orientation.

Our `ViewModel` objects will help us hold onto data across those configuration changes, so we do not lose information despite our fragments being destroyed and recreated. This happens mostly automatically, which is one of the benefits of `ViewModel`.

However, soon we will want to start putting our to-do items in a database, and we will want to do that I/O on background threads. Now, we have a problem: what happens if we start I/O in one fragment and complete it in another? For example, what if we start loading our items for the `RosterListFragment`, but we undergo a configuration change before that load finishes?

The Jetpack solution for this is `LiveData`. It is a simple data holder with an associated set of observers, who are notified when the data changes. The key is that `LiveData` knows that its observers have lifecycles, such as being created and destroyed. `LiveData` not only delivers data changes to currently-registered observers, but it delivers the latest data to new observers when they are ready. Our `ViewModel` objects can expose our repository-held data via `LiveData`, and our fragments can seamlessly get the initial data and updates as needed. Along the way, this will fix the problem from the preceding tutorial, where changes that we made in `EditFragment` are no longer visible to `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 the Dependency

First, we need another dependency, one that contains the LiveData classes and interfaces. Add this line to the dependencies closure of the app/build.gradle file:

```
implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"
```

(from [T22-LiveData/ToDo/app/build.gradle](#))

The overall dependencies closure should now resemble:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.fragment:fragment-ktx:1.0.0'  
    implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "org.koin:koin-core:$koin_version"  
    implementation "org.koin:koin-android:$koin_version"  
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test:runner:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'  
}
```

(from [T22-LiveData/ToDo/app/build.gradle](#))

## Step #2: Making Our Items Live

Next, we need our ToDoRepository to start thinking in terms of LiveData. Eventually, it will get LiveData from our database, courtesy of Room. For now, though, it will hold its own LiveData.

To that end, replace the ToDoRepository implementation that you have with this:

```
package com.commonsware.todo.repo  
  
import androidx.lifecycle.LiveData  
import androidx.lifecycle.MutableLiveData
```

## MAKING OUR REPOSITORY LIVE

---

```
class ToDoRepository {
    private val _items =
        MutableLiveData<List<ToDoModel>>().apply { value = listOf() }
    val items: LiveData<List<ToDoModel>> = _items

    fun save(model: ToDoModel) {
        _items.value = if (current().any { it.id == model.id }) {
            current().map { if (it.id == model.id) model else it }
        } else {
            current() + model
        }
    }

    fun delete(model: ToDoModel) {
        _items.value = current().filter { it.id != model.id }
    }

    fun find(modelId: String?) = current().find { it.id == modelId }

    private fun current() = _items.value!!
}
```

(from [T22-LiveData/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

Our RosterMotor uses the `items` property to get at the list of to-do items. That used to be a simple list of `ToDoModel` objects, but now it is a `LiveData` of the list of `ToDoModel` objects.

However, the real object is `_items`. This is a `MutableLiveData`, which is a subclass of `LiveData`. `MutableLiveData` is designed for classes, like `ToDoRepository`, that are creating new `LiveData` objects. Many implementations of `LiveData` are really `MutableLiveData` objects. However, we do not need consumers of this `_items` object to have all of the capabilities of `MutableLiveData`. So, the typical pattern is to keep the `MutableLiveData` as a `private` object to its maintainer, and expose the object as a `LiveData` type. In Kotlin, we can do that by having the `items` property be typed as `LiveData` but point to the underlying `MutableLiveData` object.

`MutableLiveData` has a `value` property. You can read it to get the latest value, and you can update it to replace that value with another. When you set `value`, any registered observers will find out about the changes. Previously, `items` was a `var`, and we would replace the `items` with a new list that reflects any changes (e.g., newly added items in `save()`). Now, `items` is a `val`, and we set `value` on `items` to reflect changes.

Internally, `save()`, `delete()`, and `find()` all need the current list of items. `LiveData`

## MAKING OUR REPOSITORY LIVE

---

is somewhat annoying to use from Kotlin, as it starts with a null value. That means we need to deal with nullable types when working with LiveData, even if in practice we do not intend to have a null value. The way ToDoRepository handles this is:

- Initialize `_items` right away, by setting the value to an empty list via `listOf()`. This way, we know for certain that `_items` always has a value.
- Use the `!!` operator inside a `current()` function to give us the current value without the nullable type

`save()`, `delete()`, and `find()` all do what they did before. They just use `current()` as the list. In addition, `save()` and `delete()` set the value on `_items` when they want to replace the list with a revised copy.

So, the public API of `ToDoRepository` is mostly the same, except that `items` is now a `LiveData`.

## Step #3: Observing Our Items

The only direct user of `items` is `RosterMotor`, and it simply exposes it as part of its API:

```
package com.commonsware.todo.ui.roster

import androidx.lifecycle.ViewModel
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository

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

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

(from [T22-LiveData/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

As a result, we do not need to make any changes there.

The only direct user of `RosterMotor` is `RosterListFragment`. There, in `onViewCreated()`, we have some lines that try to work with `items` as a list:

```
adapter.submitList(motor.items)
empty.visibility = if (motor.items.isEmpty()) View.VISIBLE else View.GONE
```

(from [T21-ViewModel/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

## MAKING OUR REPOSITORY LIVE

---

Replace those lines with:

```
motor.items.observe(this, Observer { items ->
    adapter.submitList(items)
    empty.visibility = if (items.isEmpty()) View.VISIBLE else View.GONE
})
```

(from [T22-LiveData/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

The main thing that a consumer of a `LiveData` does is call `observe()`. This says “give me the current value and all updates, please!”. The first parameter — in this case, the `RosterListFragment` itself — is a `LifecycleOwner`, which is an interface that tells interested parties about its lifecycle (create, destroy, etc.). `LiveData` knows how to use a `LifecycleOwner` to stop sending us updates after we no longer need them. The second parameter is an implementation of the `Observer` interface. In Kotlin, that can be a simple lambda expression, taking our data as a parameter. So, the `items` parameter in the lambda expression is our list of to-do items. However, now it is not only the *initial* list of to-do items, but any *changed* editions of the list that are published by the `ToDoRepository`.

So, our `Observer` 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.

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

Licensed solely for use by Patrocinio Rodriguez

# **Using a Unidirectional Data Flow**

---

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. 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 `RosterMotor` keep track just of the `items` list, we will have it emit `RosterViewState` objects. Right now, these will just wrap the `items`, but we can add more properties to `RosterViewState` as we add more features with data to be tracked.

## USING A UNIDIRECTIONAL DATA FLOW

This idea of a “motor” and 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:

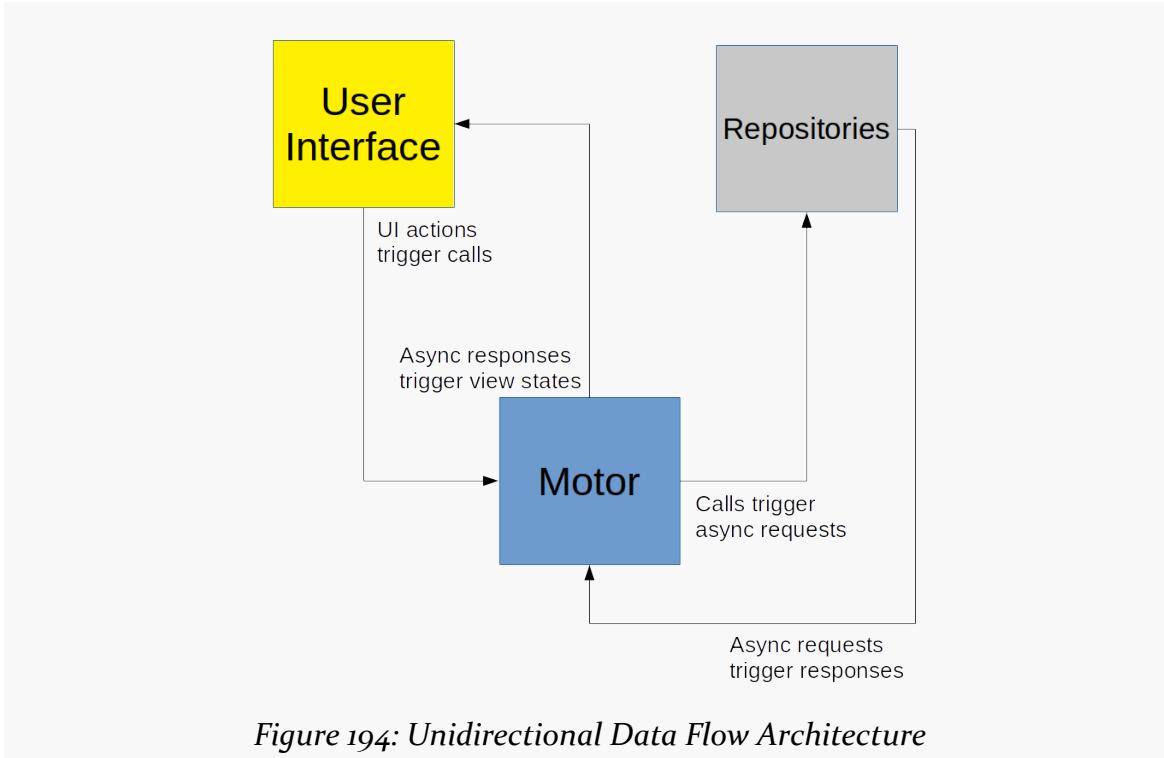


Figure 194: Unidirectional Data Flow Architecture

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 View State

First, 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:

```
class RosterViewState(  
    val items: List<ToDoModel> = listOf()
```

## USING A UNIDIRECTIONAL DATA FLOW

---

```
)
```

(from [T23-ViewState/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: Mapping View States

From `ToDoRepository`, we have a `LiveData` emitting our to-do item lists. We now want to have `RosterMotor` emit `RosterViewState` objects instead. And, since we still have configuration changes and future asynchronous work to consider, we want to use `LiveData` to emit those `RosterViewState` objects. In effect, we want to somehow convert a `LiveData` of our to-do items into a `LiveData` of our `RosterViewState`.

For that, we have `Transformations.map()`.

In `RosterMotor`, replace the `items` property with:

```
val states: LiveData<RosterViewState> = Transformations.map(repo.items) { RosterViewState(it) }
```

(from [T23-ViewState/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

`Transformations.map()` takes a `LiveData` and a lambda expression. That lambda expression will be invoked each time the `LiveData` gets a new value, and the job of the lambda expression is to convert the input value (list of to-do items) into the desired output value (`RosterViewState`). `Transformations.map()` returns a *new* `LiveData` object that emits the objects returned by the lambda expression. So, in our case, `states` is a `LiveData` of `RosterViewState`.

You might wonder why we need the return type. In other words, you might think that we could just use:

```
val states = Transformations.map(repo.items) { RosterViewState(it) }
```

This would work, but we get a warning in Android Studio, complaining that it does not know whether `Transformations.map()` might return `null` or not. As it turns out, `Transformations.map()` will not return `null`. By adding the explicit type declaration, we eliminate the Android Studio complaint.

At this point, the `RosterMotor.kt` file, with both classes, should look like:

```
package com.commonsware.todo.ui.roster
```

## USING A UNIDIRECTIONAL DATA FLOW

---

```
import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository

class RosterViewState(
    val items: List<ToDoModel> = listOf()
)

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
    val states: LiveData<RosterViewState> = Transformations.map(repo.items) { RosterViewState(it) }

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

(from [T23-ViewState/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

## Step #3: Consuming View States

Our RosterListFragment now will complain that it no longer has an `items` property on our `RosterMotor`. So, change the `LiveData` consumption block to look like this:

```
motor.states.observe(this, Observer { state ->
    adapter.submitList(state.items)
    empty.visibility = if (state.items.isEmpty()) View.VISIBLE else View.GONE
})
```

(from [T23-ViewState/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Now, we are observing the `states` `LiveData`, and we are getting our `items` list out of the `RosterViewState` that we get from the `Observer` (referred to as `state`). Otherwise, this is the same as what we had before. And, if you run the app, you should see no functional changes.

The `onViewCreated()` function for `RosterListFragment` should now look like:

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

    val adapter =
        RosterAdapter(
            inflater = layoutInflater,
            onCheckboxToggle = { model ->
                motor.save(model.copy(isCompleted = !model.isCompleted))
            },
            onRowClick = { model -> display(model) }
}
```

## USING A UNIDIRECTIONAL DATA FLOW

```
view.items.apply {
    setAdapter(adapter)
    layoutManager = LinearLayoutManager(context)

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

motor.states.observe(this, Observer { state ->
    adapter.submitList(state.items)
    empty.visibility = if (state.items.isEmpty()) View.VISIBLE else View.GONE
})
}
```

(from [T23-ViewState/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/roster/RosterMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# Extending the Architecture

---

So far, we have only updated `RosterListFragment` to use the `ViewModel-and-LiveData` architecture. In this tutorial, we will employ that same approach for `DisplayFragment` and `EditFragment`.

One advantage here is that both share a common requirement: getting a particular `ToDoModel` given its ID. This will allow us to share a common `ViewModel` implementation.

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: Making Single Items Live

First, we should make `forId()` in `ToDoRepository` work using `LiveData`, just as we have done for `items`. Eventually, our `forId()` call will be making a database query, and we want to do that on a background thread, so `LiveData` will help with that.

However, right now, our to-do items are not yet in a database, so we will need some other way to offer up a `LiveData`. In our case, we can once again use `Transformations.map()`, this time to implement a filtering operation, to return a `LiveData` of a single `ToDoModel`.

To that end, change `find()` on `ToDoRepository` to be:

```
fun find(modelId: String?): LiveData<ToDoModel> =  
    Transformations.map(items) {  
        it.find { model -> model.id == modelId }  
    }
```

## EXTENDING THE ARCHITECTURE

---

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

`items` is a `LiveData` of our list of to-do items. Our mapping lambda expression uses `find()` to find the first model whose ID matches the one passed into our own `find()` function. This means that our `find()` function returns a `LiveData` of `ToDoModel?`, as there are two scenarios in which our mapping lambda expression will not find a matching `ToDoModel`:

- The `modelId` might be `null`, such as when we start `EditFragment` to create a new to-do item
- The `modelId` might not reflect a currently-available item (e.g., it was deleted)

At this point, `ToDoRepository` should resemble:

```
package com.commonsware.todo.repo

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.Transformations

class ToDoRepository {
    private val _items =
        MutableLiveData<List<ToDoModel>>().apply { value = listOf() }
    val items: LiveData<List<ToDoModel>> = _items

    fun save(model: ToDoModel) {
        _items.value = if (current().any { it.id == model.id }) {
            current().map { if (it.id == model.id) model else it }
        } else {
            current() + model
        }
    }

    fun delete(model: ToDoModel) {
        _items.value = current().filter { it.id != model.id }
    }

    fun find(modelId: String?): LiveData<ToDoModel> =
        Transformations.map(items) {
            it.find { model -> model.id == modelId }
        }

    private fun current() = _items.value!!
}
```

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

## EXTENDING THE ARCHITECTURE

---

This change broke all our uses of `find()` on `ToDoRepository`, but we will be fixing them in the rest of the tutorial.

## Step #2: 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.

Since we can share this class between two fragments, we should not put it either the display or the edit packages. Instead, we can put it in the parent `ui` package, alongside our activities.

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 `SingleModelMotor`, then choose “Class” for the “Kind”. Then click OK to create the empty `SingleModelMotor` class.

Then, replace its contents with:

```
import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import com.commonware.todo.repo.ToDoModel
import com.commonware.todo.repo.ToDoRepository

class SingleModelState(
    val item: ToDoModel? = null
)

class SingleModelMotor(repo: ToDoRepository, modelId: String) : ViewModel() {
    val states: LiveData<SingleModelState> =
        Transformations.map(repo.find(modelId)) { SingleModelState(it) }
}
```

This has the same basic structure as does `RosterMotor` and `RosterViewState`. The difference in the view-state is that our state is just a single `ToDoModel`... or possibly `null`, if we cannot find a model matching the requested ID.

Our `SingleModelMotor` class takes two constructor parameters: our `ToDoRepository` and the ID of the model that we want. We use both of those to set up the `states` property, using `find()` on the `ToDoRepository` to get a `LiveData` for our desired item and using that to construct the `SingleModelState`.

## EXTENDING THE ARCHITECTURE

---

We also need to teach Koin about this ViewModel implementation. So, in ToDoApp, add this line to the koinModule configuration:

```
viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
```

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This is a bit different than the viewModel line that we have for RosterMotor:

```
viewModel { RosterMotor(get()) }
```

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This time, not only do we need our ToDoRepository but also the modelId. While we can get the ToDoRepository from Koin itself, we need to get the modelId from whoever wants to use a SingleModelMotor. The way that we do that in Koin is to add a parameter list to the lambda expression that creates our ViewModel and include modelId in that parameter list. In the next step, we will see how we pass in that value when injecting our motor.

At this point, ToDoApp should resemble:

```
package com.commonsware.todo

import android.app.Application
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.ui.SingleModelMotor
import com.commonsware.todo.ui.roster.RosterMotor
import org.koin.android.ext.android.startKoin
import org.koin.androidx.viewmodel.ext.koin.viewModel
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(this, listOf(koinModule))
    }
}
```

## EXTENDING THE ARCHITECTURE

---

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

### Step #3: Displaying with a Motor

Now, we can have `DisplayFragment` use this new `ViewModel`.

First, replace the `repo` property with this new `motor` property:

```
private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }
```

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/display/DisplayFragment.kt](#))

This is similar to how we injected the motor into `RosterListFragment`. The difference is the lambda expression, where we use a `parametersOf()` function to wrap up the `modelId` that we get from our args. These parameters wind up as parameters to the lambda expression that creates the `SingleModelMotor`, and that is how `SingleModelMotor` determines the ID of the item that we want.

Then, replace the current `onViewCreated()` function with this:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    motor.states.observe(this, Observer { state -> binding.model = state.item } )
}
```

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/display/DisplayFragment.kt](#))

As we did with `RosterListFragment`, this has us observe the states from our motor and pass our `ToDoModel` to the data binding framework, so our widgets can be populated.

At this point, `DisplayFragment` should resemble:

```
package com.commonsware.todo.ui.display

import android.os.Bundle
import android.view.*
import androidx.fragment.app.Fragment
import androidx.lifecycle.Observer
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 org.koin.androidx.viewmodel.ext.android.viewModel
import org.koin.core.parameter.parametersOf

class DisplayFragment : Fragment() {
    private val args: DisplayFragmentArgs by navArgs()
    private lateinit var binding: TodoDisplayBinding
```

## EXTENDING THE ARCHITECTURE

---

```
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.states.observe(this, Observer { state -> binding.model = state.item } )
}

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 [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/display/DisplayFragment.kt](#))

## Step #4: Making Yet Another Motor

EditFragment also needs a ViewModel that gets our model by ID, but it also needs save() and delete() functions.

So, let's add those to SingleModelMotor:

```
fun save(model: ToDoModel) = repo.save(model)

fun delete(model: ToDoModel) = repo.delete(model)
```

## EXTENDING THE ARCHITECTURE

---

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

These just delegate their calls to our `ToDoRepository`.

However, this will result in a compile error, as right now, `repo` is just a constructor parameter, so it is not available to these functions.

We also have another problem with the constructor: the one that we set up earlier does not allow for a `null` `modelId` value. This will be an issue with `EditFragment`, as it may not have a `modelId`, if we are to be creating a new to-do item.

To fix those problems, change the constructor to turn `repo` into a property and allow `modelId` to be `null`:

```
class SingleModelMotor(private val repo: ToDoRepository, modelId: String?) : ViewModel() {
```

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

At this point, `SingleModelMotor.kt` should look like:

```
package com.commonsware.todo.ui

import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository

class SingleModelState(
    val item: ToDoModel? = null
)

class SingleModelMotor(private val repo: ToDoRepository, modelId: String?) : ViewModel() {
    val states: LiveData<SingleModelState> =
        Transformations.map(repo.find(modelId)) { SingleModelState(it) }

    fun save(model: ToDoModel) = repo.save(model)

    fun delete(model: ToDoModel) = repo.delete(model)
}
```

(from [T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

## Step #5: Editing with a Motor

Now, we can update `EditFragment` to use the revised `SingleModelMotor`.

As we did with `DisplayFragment`, replace the `repo` property with a `motor` property:

## EXTENDING THE ARCHITECTURE

---

```
private val motor: SingleModelMotor by viewModel { parametersOf(args.modelId) }  
(from T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt)
```

And replace the `onViewCreated()` function with one that observes our states:

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  
    motor.states.observe(this, Observer { state -> binding.model = state.item } )  
}  
(from T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt)
```

Finally, modify `save()` and `delete()` to call the corresponding `save()` and `delete()` functions on `motor` instead of `repo`:

```
private fun save() {  
    val edited = if (binding.model == null) {  
        ToDoModel(  
            description = binding.desc.text.toString(),  
            isCompleted = binding.isCompleted.isChecked,  
            notes = binding.notes.text.toString()  
    )  
    } else {  
        binding.model?.copy(  
            description = binding.desc.text.toString(),  
            isCompleted = binding.isCompleted.isChecked,  
            notes = binding.notes.text.toString()  
    )  
    }  
  
    edited?.let { motor.save(it) }  
    navToDisplay()  
}  
  
private fun delete() {  
    binding.model?.let { motor.delete(it) }  
    navToList()  
}
```

```
(from T24-Arch/ToDo/app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt)
```

And, at this point, if you run the app, it should work as it did before.

## What We Changed

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

## EXTENDING THE ARCHITECTURE

---

- [app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/display/DisplayFragment.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/edit/EditFragment.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# **Switching to Coroutines**

---

In `ToDoRepository`, `items` and `find()` are set up to support asynchronous operations, courtesy of `LiveData`. `save()` and `delete()` are not, and we should do something about that.

“Something”, in this case, involves 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.

Right now, multiple threads will *not* be involved, as we do not need background threads for updating an in-memory list. However, when we move our to-do items into a database, we will need background threads, and coroutines become far more important.

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 Dependency**

Coroutines are described as being part of Kotlin. In reality, most of what makes up “coroutines” comes from a library. We will need that library to use coroutines. However, rather than request that library directly, we are going to pull in another Jetpack library, one that has its own dependency on coroutines, and so will bring in the coroutines library to our app.

Add this line to the dependencies closure of `app/build.gradle`:

## SWITCHING TO COROUTINES

```
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"
```

(from [T25-Coroutines/ToDo/app/build.gradle](#))

`lifecycle-viewmodel-ktx` is a set of Kotlin extension functions for `ViewModel` and related classes. Among those are some extension functions that help us use coroutines in our viewmodels.

Note that this is an alpha version of a library. Normally, we try to avoid these, unless we want to experiment with cutting-edge features of Jetpack. In this case, Google is being very slow about pushing this library to production, and it seems to be working well. So, we will use the alpha library, and hope for the best.

At this point, the overall dependencies closure should look like:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.fragment:fragment-ktx:1.0.0'  
    implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"  
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "org.koin:koin-core:$koin_version"  
    implementation "org.koin:koin-android:$koin_version"  
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test:runner:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'  
}
```

(from [T25-Coroutines/ToDo/app/build.gradle](#))

## Step #2: Suspending Our Functions

Next, in `ToDoRepository`, add the `suspend` keyword to the front of the `save()` and `delete()` functions:

```
suspend fun save(model: ToDoModel) {  
    _items.value = if (current().any { it.id == model.id }) {  
        current().map { if (it.id == model.id) model else it }  
    } else {
```

## SWITCHING TO COROUTINES

```
    current() + model
}
}

suspend fun delete(model: ToDoModel) {
    _items.value = current().filter { it.id != model.id }
}
```

(from [T25-Coroutines/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

This keyword indicates that these functions may use a background thread, in such a fashion that callers may need to “suspend” their work waiting for these functions to return.

Right now, though, Android Studio will be annoyed with you, marking those keywords in gray:

```
suspend fun save(model: ToDoModel) {
    _items.value = if (current().any { it.id == model.id })
        current().map { if (it.id == model.id) model else it }
    } else {
        current() + model
    }
}

suspend fun delete(model: ToDoModel) {
    _items.value = current().filter { it.id != model.id }
}
```

Figure 195: Android Studio, Complaining About Pointless suspend

That is because there is no sign that we are actually doing anything with a background thread. In this case, Android Studio is correct, and this suspend keyword is pointless. However, we are simply setting things up for later use, and there is no particular harm in marking these functions as suspend functions right now.

## Step #3: Launching Those Functions

Next, open up `SingleModelMotor`. You will find that Android Studio is *really* annoyed with you, showing some errors:

```
17 ↗ fun save(model: ToDoModel) = repo.save(model)
18
19 ↗ fun delete(model: ToDoModel) = repo.delete(model)
```

Figure 196: Android Studio, Complaining About Calling suspend

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 this, replace the `save()` and `delete()` functions in `SingleModelMotor` with these:

```
fun save(model: ToDoModel) {
    viewModelScope.launch(Dispatchers.Main) {
        repo.save(model)
    }
}

fun delete(model: ToDoModel) {
    viewModelScope.launch(Dispatchers.Main) {
        repo.delete(model)
    }
}
```

(from [T25-Coroutines/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.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 blocked as needed to wait for the suspend work to complete”.

A `CoroutineScope` is a container, of sorts, for these sorts of coroutine-based bits of work. `viewModelScope` is an extension function supplied by `lifecycle-viewmodel-ktx`, to give us a `CoroutineScope` associated with our `ViewModel`.

## SWITCHING TO COROUTINES

---

We are passing two things to launch. The big thing is the lambda expression indicating what we want “launched”, which is our code that calls suspend functions. We also pass Dispatchers.Main, which says “for the bits of this work *not* in suspend functions, do that work on the Android main application thread”. Later, if we want to do something that affects our UI after we save or delete the model, we could just have those statements after our repo.save() and repo.delete() calls, and those statements would be executed on the main application thread.

If you now look at RosterMotor, you will see that its save() function suffers from the same problem as did the one in SingleModelMotor. So, change its save() function to be:

```
fun save(model: ToDoModel) {
    viewModelScope.launch(Dispatchers.Main) {
        repo.save(model)
    }
}
```

(from [T25-Coroutines/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

If you run the app now, once again, it should work without any changes.

At this point, while we have introduced coroutines, they are not very useful. In particular, ToDoRepository has suspend functions that do not actually arrange to do anything on a background thread. We will be addressing that a bit later, when we introduce Room and start storing our to-do items in a database.

## 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/repo/ToDoRepository.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#)

Licensed solely for use by Patrocinio Rodriguez

---

---

## **Phase Three: Testing This Thing**

---

Licensed solely for use by Patrocinio Rodriguez

# **Testing Our Repository**

---

We think that our ToDoRepository 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 "Test Tour" 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 OUR REPOSITORY

---

examine your project in Android Studio, you will see that there are three directories in app/src/: androidTest/, main/, and test/:



Figure 197: 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.InstrumentationRegistry
import androidx.test.runner.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)
class ExampleInstrumentedTest {
    @Test
    fun useAppContext() {
        // Context of the app under test.
        val applicationContext = InstrumentationRegistry.getTargetContext()
```

## TESTING OUR REPOSITORY

---

```
        assertEquals("com.commonsware.todo", appContext.packageName)
    }
}
```

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)
    }
}
```

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.

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.

## TESTING OUR REPOSITORY

---

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`:

```
testImplementation "androidx.arch.core:core-testing:2.0.0"
testImplementation "org.amshove.kluent:kluent-android:1.49"
testImplementation "org.mockito:mockito-inline:2.21.0"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.1.0"
testImplementation 'com.jraska.livedata:testing-ktx:1.1.0'
```

## TESTING OUR REPOSITORY

---

(from [T26-Tests/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.

The org.amshove.kluent:kluent-android library is [Kluent](#), a library for helping us write tests that read a bit more like human-language sentences.

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. We will use Mockito in [the next tutorial](#).

The com.jraska.livedata:testing-ktx library is [a LiveData testing library](#), to make it easier for us to test LiveData from unit tests.

At this point, your dependencies closure should resemble:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.fragment:fragment-ktx:1.0.0'  
    implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"  
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "org.koin:koin-core:$koin_version"  
    implementation "org.koin:koin-android:$koin_version"  
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"  
    testImplementation 'junit:junit:4.12'  
    testImplementation "androidx.arch.core:core-testing:2.0.0"  
    testImplementation "org.amshove.kluent:kluent-android:1.49"  
    testImplementation "org.mockito:mockito-inline:2.21.0"  
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.1.0"  
    testImplementation 'com.jraska.livedata:testing-ktx:1.1.0'  
    androidTestImplementation 'androidx.test:runner:1.1.1'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'  
}
```

(from [T26-Tests/ToDo/app/build.gradle](#))

## 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 `ToDoRepository`, we should rename it to something like `ToDoRepositoryTest`. Typically, a unit test class focuses on testing one main project class, so `ToDoRepositoryTest` would focus on testing `ToDoRepository`.

Also, typically, the test class resides in the same package as is the class that it is testing. So, just as `ToDoRepository` is in `com.commonsware.todo.repo`, so should `ToDoRepositoryTest`. However, we do not have a `repo` 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 `repo` for the new package name, then click “OK” to create the sub-package.

## TESTING OUR REPOSITORY

Next, drag and drop the ExampleUnitTest class from its current location (inside com.commonware.todo) into this new repo sub-package. As before when we moved around classes, this will bring up a “Move” dialog:

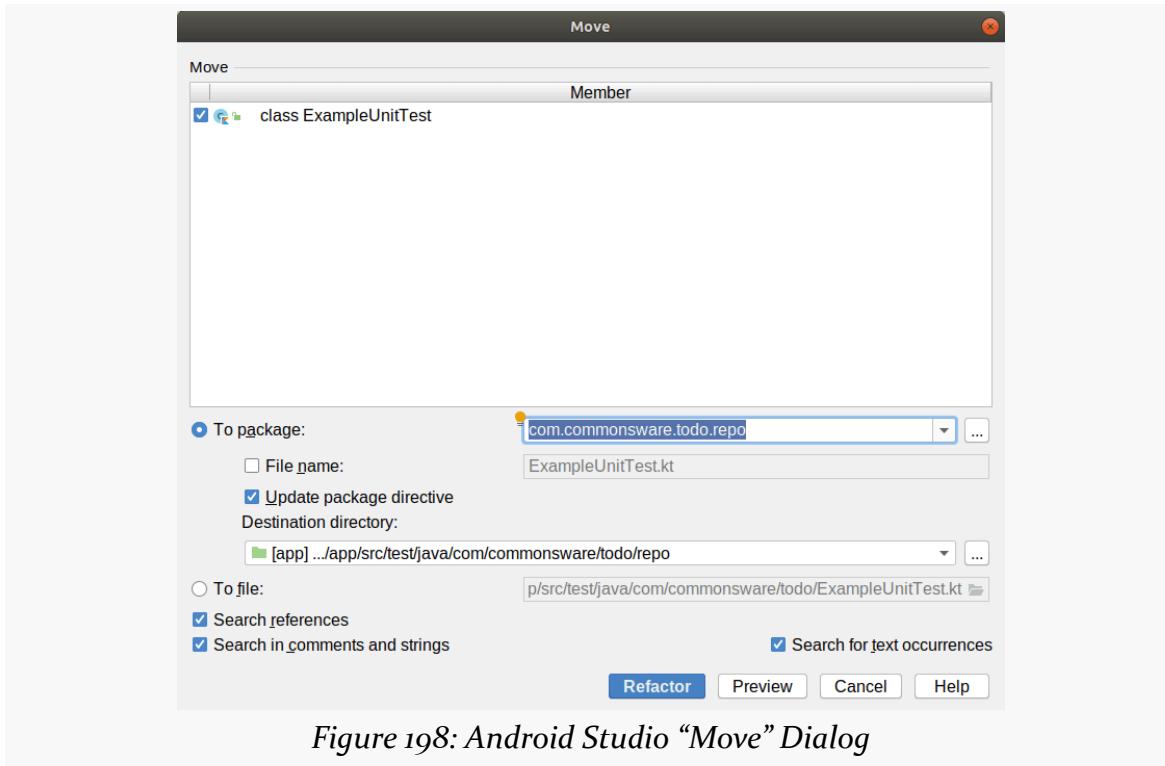


Figure 198: Android Studio “Move” Dialog

Just click the “Refactor” button to complete the move.

Then, right-click over the newly-moved ExampleUnitTest and choose “Refactor” > “Rename” from the context menu. In the “Rename” dialog, fill in ToDoRepositoryTest as the new name:

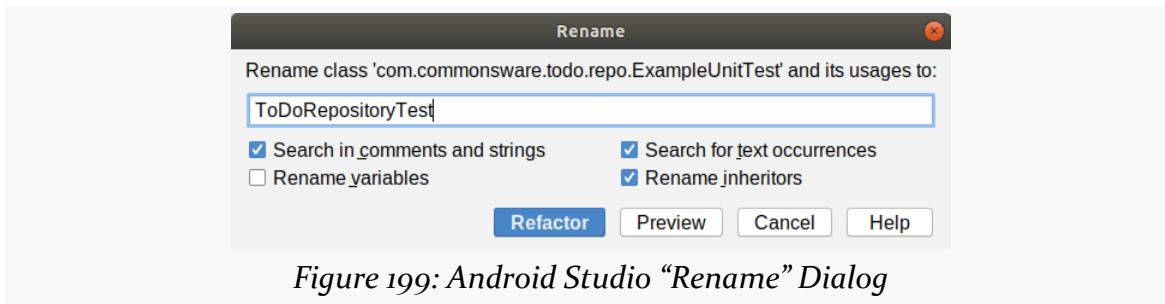


Figure 199: Android Studio “Rename” Dialog

Then click “Refactor”. This will rename both the file and the Kotlin class, so we now

## TESTING OUR REPOSITORY

have a `ToDoRepositoryTest`.

## Step #5: Running the Stub Unit Test

You have a variety of ways to run the unit test. The simplest ones come from green triangle “run” icons in the gutter of the editor:

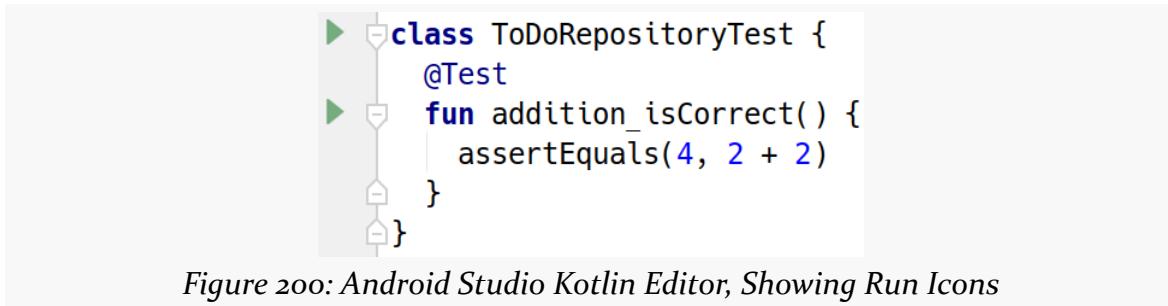


Figure 200: Android Studio Kotlin Editor, Showing Run Icons

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 `ToDoRepositoryTest`, that class’ test functions will be run, and the “Run” tool window will open in Android Studio to show you the results:



Figure 201: 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, the author apologizes for not adding “SPOILER ALERT” before the preceding sentence)

## TESTING OUR REPOSITORY

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 loop indicating that the previous test passed:

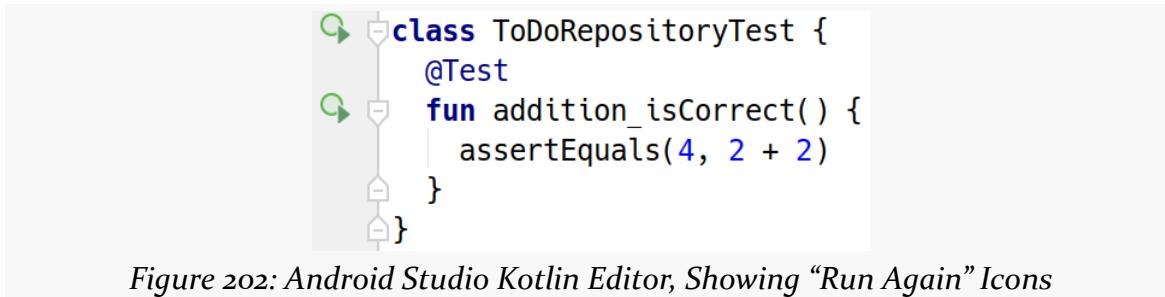


Figure 202: Android Studio Kotlin Editor, Showing “Run Again” Icons

If you change the `assertEquals()` call to be `assertEquals(5, 2 + 2)` and run the test again, you will see that it fails:

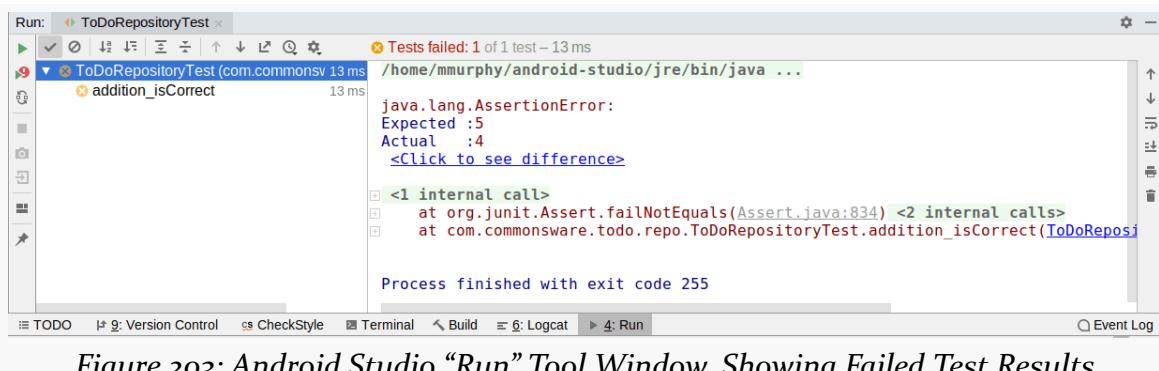


Figure 203: Android Studio “Run” Tool Window, Showing Failed Test Results

The yellow icon indicates that the test failed due to a failed assertion. And, at this point, our editor icons are now a red loop, indicating that the previous run of the test failed:

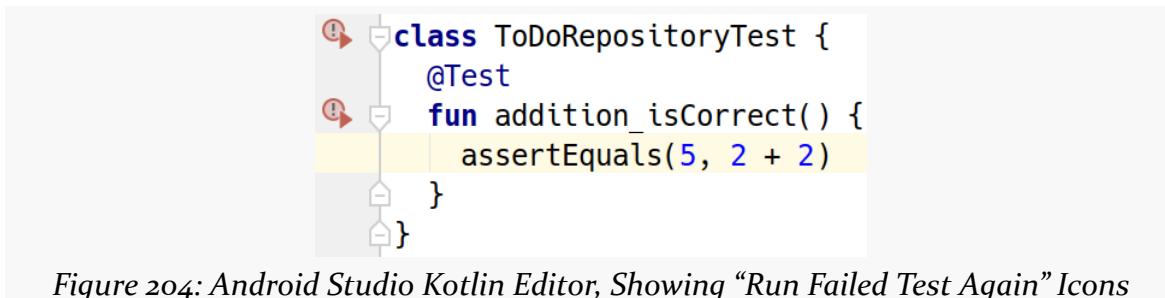


Figure 204: Android Studio Kotlin Editor, Showing “Run Failed Test Again” Icons

If you change the `assertEquals()` call to be `assertEquals(4, 2 / 0)` and run the

## TESTING OUR REPOSITORY

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 205: Android Studio “Run” Tool Window, Showing Crashed Test Results

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: Writing and Running Our First Test

Now, we can start putting in test logic for testing `ToDoRepository` itself. 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.

Replace the `ToDoRepositoryTest` implementation with:

```
package com.commonsw.todo.repo

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import com.jraska.livedata.test
import kotlinx.coroutines.runBlocking
import org.amshove.kluent.shouldBeEmpty
import org.amshove.kluent.shouldContainSame
import org.amshove.kluent.shouldEqual
import org.junit.Before
import org.junit.Rule
import org.junit.Test

class ToDoRepositoryTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    private lateinit var underTest: ToDoRepository
```

## TESTING OUR REPOSITORY

---

```
@Before
fun `setUp`() {
    underTest = ToDoRepository()
}

@Test
fun `can add items`() {
    val testModel = ToDoModel("test model")

    underTest.apply {
        items.test().value().shouldBeEmpty()

        runBlocking { save(testModel) }

        items.test().value() shouldContainSame listOf(testModel)

        find(testModel.id).test().value() shouldEqual testModel
    }
}
}
```

There is quite a bit to explain in these few lines of code.

```
@get:Rule
val instantTaskExecutorRule = InstantTaskExecutorRule()
```

In JUnit, a Rule is a standard way to package reusable bits of test logic, particularly related to common test configuration. The androidx.arch.core:core-testing library that we added earlier gives us a Rule called InstantTaskExecutorRule. This rule configures the threading behavior of the Jetpack components, such that things like LiveData and MutableLiveData do all their work on the current thread, rather than special threads like Android's main application thread. 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”. InstantTaskExecutorRule therefore lets us test LiveData with unit tests, by having it no longer depend upon Android's main application thread.

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.

## TESTING OUR REPOSITORY

---

```
private lateinit var underTest: ToDoRepository

@Before
fun setUp() {
    underTest = ToDoRepository()
}
```

We then have an `underTest` property for our `ToDoRepository`. `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 `ToDoRepository` instance. Ideally, we would just use a `val` and initialize our `ToDoRepository` that way, skipping this `setUp()` function. Unfortunately, our `InstantTaskExecutorRule` will not have had a chance to do its work yet, and the `MutableLiveData` in our `ToDoRepository` will crash at runtime with an error when it tries to work with Android’s main application thread. So, we are forced to use this approach, so the `InstantTaskExecutorRule` can fix up the threading before we try creating a `ToDoRepository` instance.

```
@Test
fun `can modify items`() {
```

As noted previously, `@Test` annotates test functions. Your test class can have other functions in it, but the ones annotated with `@Test` will be the “entry points” and will be executed by JUnit as part of running your tests.

Normally, our function names are written in `lowerCamelCase()`. Kotlin has an interesting feature that allows function names to have whitespace, punctuation, etc. in them, by wrapping the function name in backticks. Frequently, this is awkward, so this is not a commonly-used feature of Kotlin. However, it does get used a fair bit in testing, as it means that our test output reads more normally (e.g., `can add items` instead of `canAddItems`).

```
val testModel = ToDoModel("test model")

underTest.apply {
    items.test().value().shouldBeEmpty()
```

`items` is a `LiveData` property. `test().value()` comes from the `com.jraska.livedata:testing-ktx` library. `test()` registers an observer of the `LiveData` for us and gives us a blocking `value()` call to wait until a value has been handed over to the `LiveData`. For our current `items` implementation, we could get away with just `items.value!!`, but that does not work for all situations, as we will

## TESTING OUR REPOSITORY

---

see shortly. The net is that `items.test().value()` gives us the `List` of to-do items presently in our repository.

`shouldBeEmpty()` is the first of the assertion functions that we will be using from Kluent. It simply tests to see whether the collection is empty or not, failing the test if the collection has something in it. Right now, since we have done nothing with the `ToDoRepository` since we created it, it should be empty.

```
runBlocking { save(testModel) }
```

Our `save()` function on our `ToDoRepository` is a suspend function. We cannot just run it directly — the Kotlin compiler will refuse. We have to run it inside of some context that indicates how the function will be called with respect to threads.

`runBlocking` is the most crude option for this, as it just says “run the function on the current thread, please”. For production code, this is rarely what we want, as if we wanted a blocking call, we would have skipped the suspend entirely. But, for *test* code, blocking a test for a moment to get the result is perfectly fine and it greatly simplifies the tests. So, this line calls `save()` and will not continue until that work is completed.

```
items.test().value() shouldContainSame listOf(testModel)
```

`shouldContainSame()` is the second Kluent assertion function that we are using. It validates that the collection on the left has the same elements as does the collection on the right. In this case, we are confirming that our repository now has our test `ToDoModel`.

```
find(testModel.id).test().value() shouldEqual testModel
```

`find()` on our `ToDoRepository` returns a `LiveData`, specifically one created using `Transformations.map()`. `Transformations.map()` creates a “cold” `LiveData` — it will not do anything unless some observer starts observing it. If we tried using `value!!` to retrieve the value of the `LiveData`, we would get `null`, as there is no observer of the `LiveData` to trigger any actual `Transformers.map()` work. `test().value()`, by contrast, registers an observer and gives us the first value that it sees, which is enough to trigger the actual `Transformers.map()` work.

Observing the `LiveData` will trigger our observer to be invoked with the results of our `find` operation. There, we use `shouldEqual`, another Kluent assertion function, to compare the `ToDoModel` from `find()` with the test model that we added earlier, to ensure that we get that model back.

## TESTING OUR REPOSITORY

---

If you run this test, you will see that it succeeds.

## Step #7: 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 `can modify items`() {
    val testModel = ToDoModel("test model")
    val replacement = testModel.copy(notes = "This is the replacement")

    underTest.apply {
        items.test().value().shouldBeEmpty()

        runBlocking { save(testModel) }

        items.test().value() shouldContainSame listOf(testModel)

        runBlocking { save(replacement) }

        items.test().value() shouldContainSame listOf(replacement)
    }
}

@Test
fun `can remove items`() {
    val testModel = ToDoModel("test model")

    underTest.apply {
        items.test().value().shouldBeEmpty()

        runBlocking { save(testModel) }

        items.test().value() shouldContainSame listOf(testModel)

        runBlocking { delete(testModel) }

        items.test().value().shouldBeEmpty()
    }
}
```

This uses all the same techniques that the first test function did. The `can modify`

## TESTING OUR REPOSITORY

---

`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.

At this point, the `ToDoRepositoryTest` class should resemble:

```
package com.commonsware.todo.repo

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import com.jraska.livedata.test
import kotlinx.coroutines.runBlocking
import org.amshove.kluent.shouldBeEmpty
import org.amshove.kluent.shouldContainSame
import org.amshove.kluent.shouldEqual
import org.junit.Before
import org.junit.Rule
import org.junit.Test

class ToDoRepositoryTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    private lateinit var underTest: ToDoRepository

    @Before
    fun setUp() {
        underTest = ToDoRepository()
    }

    @Test
    fun `can add items`() {
        val testModel = ToDoModel("test model")
```

## TESTING OUR REPOSITORY

---

```
underTest.apply {
    items.test().value().shouldBeEmpty()

    runBlocking { save(testModel) }

    items.test().value() shouldContainSame listOf(testModel)

    find(testModel.id).test().value() shouldEqual testModel
}

}

@Test
fun `can modify items`() {
    val testModel = ToDoModel("test model")
    val replacement = testModel.copy(notes = "This is the replacement")

    underTest.apply {
        items.test().value().shouldBeEmpty()

        runBlocking { save(testModel) }

        items.test().value() shouldContainSame listOf(testModel)

        runBlocking { save(replacement) }

        items.test().value() shouldContainSame listOf(replacement)
    }
}

@Test
fun `can remove items`() {
    val testModel = ToDoModel("test model")

    underTest.apply {
        items.test().value().shouldBeEmpty()

        runBlocking { save(testModel) }

        items.test().value() shouldContainSame listOf(testModel)

        runBlocking { delete(testModel) }

        items.test().value().shouldBeEmpty()
    }
}
}
```

(from [T26-Tests/ToDo/app/src/test/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](#)

Licensed solely for use by Patrocinio Rodriguez

# **Testing a Motor**

---

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. Nor will you be interested in writing enough tests to achieve 100% test coverage. This is a set of tutorials in a book, not an app that you are building for your manager or for customers. We need to be realistic.

However, this tutorial still has you add another unit test. In principle, this is to help our test coverage. In reality, it is to demonstrate the use of a mock.

As was noted in [the previous tutorial](#), we use mocks for two main things.

One is for creating a fake instance of some object, one that we teach how to respond 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)

## TESTING A MOTOR

---

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.

So, in this tutorial, we will test `SingleModelMotor`, using a mock `ToDoRepository`, to see how these things work.

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 Another Unit Test Class

Since we are testing a different class (`SingleModelMotor`) than we did before (`ToDoRepository`), we should set up a dedicated test class for it. In addition, `SingleModelMotor` is in `com.commonsware.todo.ui`, not `com.commonsware.todo.repo`, so we will need another package.

However, we cannot right-click over the `com.commonsware.todo` package anymore, as Android Studio does not show it — it only shows `com.commonsware.todo.repo`.

So, in the test source set, right-click over the `java` directory and choose “New” > “Package” from the context menu. Fill in `com.commonsware.todo.ui` for the new package name, then click “OK” to create this package.

Then, right-click over the newly-created `ui` sub-package and choose “New” > “Kotlin File/Class” from the context menu. Fill in `SingleModelMotorTest` as the name, and choose “Class” for the “Kind”. Click “OK” to create the empty class.

## Step #2: Tweaking Our Motor

Before we work on the test though, take a look at our current implementation of `SingleModelMotor`:

```
package com.commonsware.todo.ui

import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.Dispatchers
```

## TESTING A MOTOR

---

```
import kotlinx.coroutines.launch

class SingleModelState(
    val item: ToDoModel? = null
)

class SingleModelMotor(private val repo: ToDoRepository, modelId: String?) : ViewModel() {
    val states: LiveData<SingleModelState> =
        Transformations.map(repo.find(modelId)) { SingleModelState(it) }

    fun save(model: ToDoModel) {
        viewModelScope.launch(Dispatchers.Main) {
            repo.save(model)
        }
    }

    fun delete(model: ToDoModel) {
        viewModelScope.launch(Dispatchers.Main) {
            repo.delete(model)
        }
    }
}
```

(from [T26-Tests/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

The `save()` and `delete()` functions each pass `Dispatchers.Main` to `launch()`. This indicates that we want the results of the `save()` and `delete()` work to be delivered to us on the Android main application thread.

This presents a problem with unit tests, though: we do not have an Android main application thread. Using `Dispatchers.Main` in the production code is fine, and it is fine in instrumented tests, but we cannot use it in unit tests.

With that in mind, modify the `SingleModelMotor` constructor to take a third parameter:

```
class SingleModelMotor(
    private val repo: ToDoRepository,
    modelId: String?,
    private val uiContext: CoroutineContext = Dispatchers.Main
) : ViewModel()
```

(from [T27-MotorTests/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

The new `uiContext` parameter is a `CoroutineContext`. We give `uiContext` a default value of `Dispatchers.Main`, so our existing code can still pass two parameters.

Then, change `save()` and `delete()` to pass `uiContext` to `launch()`, instead of directly referencing `Dispatchers.Main`:

## TESTING A MOTOR

---

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

fun delete(model: ToDoModel) {
    viewModelScope.launch(uiContext) {
        repo.delete(model)
    }
}
```

(from [T27-MotorTests/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

Now, our existing code works as it did before, but we have the flexibility of passing a different value for uiContext from our unit tests.

At this point, SingleModelMotor should look like:

```
package com.commonsware.todo.ui

import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlin.coroutines.CoroutineContext

class SingleModelState(
    val item: ToDoModel? = null
)

class SingleModelMotor(
    private val repo: ToDoRepository,
    modelId: String?,
    private val uiContext: CoroutineContext = Dispatchers.Main
) : ViewModel() {
    val states: LiveData<SingleModelState> =
        Transformations.map(repo.find(modelId)) { SingleModelState(it) }

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

## TESTING A MOTOR

---

```
}

fun delete(model: ToDoModel) {
    viewModelScope.launch(uiContext) {
        repo.delete(model)
    }
}
```

(from [T27-MotorTests/Todo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

## Step #3: Setting Up a Mock Repository

Next, change SingleModelMotorTest to be:

```
package com.commonsware.todo.ui

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import androidx.lifecycle.MutableLiveData
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.Dispatchers
import org.amshove.kluent.When
import org.amshove.kluent.calling
import org.amshove.kluent.itReturns
import org.amshove.kluent.mock
import org.junit.Before
import org.junit.Rule

class SingleModelMotorTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    private val testModel = ToDoModel("this is a test")
    private val repo = mock(ToDoRepository::class)
    private lateinit var underTest: SingleModelMotor

    @Before
    fun setUp() {
        When calling repo.find(testModel.id) itReturns MutableLiveData<ToDoModel>().apply
        { value = testModel }

        underTest = SingleModelMotor(repo, testModel.id, Dispatchers.Default)
    }
}
```

## TESTING A MOTOR

---

Some elements of this, such as the `instantTaskExecutorRule`, are the same as they were in `ToDoRepositoryTest`.

The first difference of note is the `repo` property. Here, we use a `mock()` extension function from Kluent to set up a mock implementation of `ToDoRepository`. This function, in turn, wraps Mockito's mock engine. 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. Right now, that `mock()` function name probably has a red under-squiggle, indicating that there is an error — we will fix that shortly.

In our `@Before`-annotated `setUp()` function, we teach our mock how to respond to a `find()` call:

```
When calling repo.find(testModel.id) itReturns MutableLiveData<ToDoModel>().apply  
{ value = testModel }
```

This `When calling ... itReturns ...` syntax comes from Kluent, as it tries to make it easy for us to describe how a mock should behave. Here, we are saying:

- When something calls `find()` on the mock, and the passed-in parameter to `find()` is this particular ID value...
- ...then return a `MutableLiveData` wrapped around our `testModel` object

Our `SingleModelMotor` then uses that mock `ToDoRepository`. When it calls `find()` and passes in our desired model ID, the mock will return the `MutableLiveData` that we declared in the `When calling ... itReturns ...` line. Also note that our `SingleModelMotor` constructor call passes in `Dispatchers.Default`, which is a default `CoroutineContext` that we can use in our unit test as an alternative to the Android-specific `Dispatchers.Main`.

As noted earlier, though, we have an error: the compiler is unhappy with `mock()`. The error hint tooltip should be something like this:



Figure 206: Android Studio Error Hint

Android Studio uses Java 1.6 bytecodes by default, and we need Java 1.8 bytecodes for

## TESTING A MOTOR

---

Mockito mocks to work.

To fix this, open up `app/build.gradle` and add this block of code to the bottom of the `android` closure:

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

This teaches Android Studio to use Java 1.8 bytecodes instead of Java 1.6 bytecodes. After you click the “Sync Now” link in the yellow banner that appears, the error reported on `mock()` should go away.

Your overall `android` closure in `app/build.gradle` should now resemble:

```
android {  
    compileSdkVersion 28  
  
    defaultConfig {  
        applicationId "com.commonsware.todo"  
        minSdkVersion 21  
        targetSdkVersion 28  
        versionCode 1  
        versionName "1.0"  
        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"  
    }  
  
    buildTypes {  
        release {  
            minifyEnabled false  
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'  
        }  
    }  
  
    androidExtensions {  
        experimental = true  
    }  
  
    dataBinding {  
        enabled = true  
    }  
  
    compileOptions {  
        sourceCompatibility JavaVersion.VERSION_1_8  
        targetCompatibility JavaVersion.VERSION_1_8  
    }  
}
```

(from [T27-MotorTests/ToDo/app/build.gradle](#))

## Step #4: Adding a Test Function

Now, we can start testing SingleModelMotor.

Add this test function to SingleModelMotorTest:

```
@Test  
fun `initial state`() {  
    underTest.states.observeForever { it.item shouldEqual testModel }  
}
```

states is a LiveData that depends upon the LiveData returned by find(). In particular, states uses Transformations.map(), and it depends upon the results of find(), which is also created by Transformations.map(). So, as with find() tests in ToDoRepositoryTest, we need to use observeForever to test the behavior of states. Here, we confirm that our initial state's item is our test ToDoModel.

If you run this test function, it should succeed.

## Step #5: 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)  
  
    runBlocking { Verify on repo that repo.save(replacement) was called }  
  
    underTest.delete(replacement)  
  
    runBlocking { Verify on repo that repo.delete(replacement) was called }  
}
```

Here, we are confirming that we can save and delete properly. To do this, we need to determine if save() and delete() on the motor are calling save() and delete() on the repository.

## TESTING A MOTOR

---

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.

The `Verify on ... that ... was called` syntax once again comes from Kluent. The first `...` is the mock that we want to examine, and the second `...` is the specific function invocation that we want to see if it occurred, complete with parameter. If those functions were called with the designated parameter, the test succeeds; otherwise, the test would fail with an assertion pointing out which call was missed.

If you run all the tests on `SingleModelMotorTest`, they should all succeed.

## 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/SingleModelMotor.kt](#)
- [app/build.gradle](#)
- [app/src/test/java/com/commonsware/todo/ui/SingleModelMotorTest.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# Testing a UI

On the one hand, UI 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
- We will be manipulating widgets as much, if not more, than we will be working with ordinary properties in our Kotlin classes

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: 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 RosterListFragment, we should rename it to RosterListFragmentTest. And, since RosterListFragment is in the ui.roster sub-package, we should have the test class mimic that.

In the androidTest source set, right click over the com.commonsware.todo package and choose “New” > “Package” from the context menu. Fill in ui.roster for the name, then click “OK” to make this sub-package.

Then, drag-and-drop the ExampleInstrumentedTest into this new ui.roster 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 RosterListFragmentTest as the replacement name, and click “Refactor” to make the change.

## Step #2: Running the Stub Instrumented Test

Just as our unit tests use green triangle “Run” icons in the gutter, so do our instrumented tests. The difference, when you click on one, is that you get the same sort of “where do you want to run this?” dialog that you do when you run the app itself.

So, if you click the run icon next to the RosterListFragmentTest class, and you choose your desired device or emulator, the test will run. You will see the same sort of “Run” output pane as you did for unit tests, and you will see that this test passes.

Of course, we are going to get rid of this test entirely shortly, so the fact that it passes just means that Google did a good job with the stub test implementation.

## Step #3: Adding Some Instrumented Test Dependencies

Right now, our dependencies closure has two androidTestImplementation statements:

## TESTING A UI

---

```
androidTestImplementation 'androidx.test:runner:1.1.1'  
androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'
```

(from [T27-MotorTests/ToDo/app/build.gradle](#))

Replace those with these lines:

```
androidTestImplementation 'androidx.test.espresso:espresso-contrib:3.1.1'  
androidTestImplementation "androidx.arch.core:core-testing:2.0.0"  
androidTestImplementation 'androidx.test.ext:junit:1.1.0'
```

(from [T28-Espresso/ToDo/app/build.gradle](#))

In this case, the dependencies that we are adding will pull in the two that we replaced via transitive dependencies.

The `androidx.test.espresso:espresso-contrib` library pulls in the Espresso testing code. In particular, `espresso-contrib` contains some utility classes related to testing `RecyclerView`, which we will want for our test of `RosterListFragment`.

The `androidx.arch.core:core-testing` library is the same one that we used in unit testing, giving us `InstantTaskExecutorRule`. We need to include it twice to make it available both for instrumented tests (`androidTestImplementation`) and for unit tests (`testImplementation`).

The `androidx.test.ext:junit` library will give us an `ActivityScenario` class that we will use for setting up our test.

At this point, your dependencies closure should resemble:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.fragment:fragment-ktx:1.0.0'  
    implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"  
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "org.koin:koin-core:$koin_version"  
    implementation "org.koin:koin-android:$koin_version"  
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"  
    testImplementation 'junit:junit:4.12'
```

## TESTING A UI

```
testImplementation "androidx.arch.core:core-testing:2.0.0"
testImplementation "org.amshove.kluent:kluent-android:1.49"
testImplementation "org.mockito:mockito-inline:2.21.0"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.1.0"
testImplementation 'com.jraska.livedata:testing-ktx:1.1.0'
androidTestImplementation 'androidx.test.espresso:espresso-contrib:3.1.1'
androidTestImplementation "androidx.arch.core:core-testing:2.0.0"
androidTestImplementation 'androidx.test.ext:junit:1.1.0'
}
```

(from [T28-Espresso/ToDo/app/build.gradle](#))

## Step #4: Initializing Our Repository

Next, replace the current implementation of `RosterListFragmentTest` with this:

```
package com.commonsware.todo.ui.roster

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import androidx.test.ext.junit.runners.AndroidJUnit4
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.runBlocking
import org.junit.Before
import org.junit.Rule
import org.junit.runner.RunWith
import org.koin.dsl.module.module
import org.koin.standalone.StandAloneContext.loadKoinModules

@RunWith(AndroidJUnit4::class)
class RosterListFragmentTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    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() {
        repo = ToDoRepository()

        loadKoinModules(module {
            single(override = true) { repo }
        })
    }
}
```

## TESTING A UI

---

```
    })
    runBlocking { items.forEach { repo.save(it) } }
}
}
```

Some of this, such as the `InstantTaskExecutorRule`, is the same as what we have in the unit test classes. Some things, however, are new to this instrumented test.

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.

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 one that Koin normally would return with a fresh instance. `loadKoinModules()` works in conjunction with the `override = true` option that we use on `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 #5: 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)))
```

## TESTING A UI

---

```
}
```

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:

- `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 `Matcher`. `Matcher` comes from a library called [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 these “matchers” that, in the end, can perform

## TESTING A UI

---

some inspections of objects and return a boolean indicating whether the objects matched expectations or not. The `matches()` `ViewAssertion` fails the test if the `Matcher` returns `false`.

`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.

## 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/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#)

Licensed solely for use by Patrocinio Rodriguez

---

## **Phase Four: Being a Little Bit Persistent**

---

Licensed solely for use by Patrocinio Rodriguez

# Getting a Room

---

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 need 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.

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.

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:

## GETTING A ROOM

---

```
room_version = "2.1.0-alpha04"
```

(from [T29-Room/ToDo/build.gradle](#))

This will make the entire closure be:

```
ext {  
    koin_version = "1.0.2"  
    room_version = "2.1.0-alpha04"  
}
```

(from [T29-Room/ToDo/build.gradle](#))

Here, we are using an alpha version of Room. That is necessary at the present time to be able to use Room with coroutines, as Google is *very* slowly moving Room along.

Then, in `app/build.gradle`, add three new dependencies that reference that version constant:

```
implementation "androidx.room:room-runtime:$room_version"  
implementation "androidx.room:room-coroutines:$room_version"  
kapt "androidx.room:room-compiler:$room_version"
```

(from [T29-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-coroutines` dependency adds support for Room doing database I/O using coroutines.

At this point, your overall dependencies closure should look like:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.fragment:fragment-ktx:1.0.0'  
    implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"  
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "org.koin:koin-core:$koin_version"
```

## GETTING A ROOM

```
implementation "org.koin:koin-android:$koin_version"
implementation "org.koin:koin-androidx-viewmodel:$koin_version"
implementation "androidx.room:room-runtime:$room_version"
implementation "androidx.room:room-coroutines:$room_version"
kapt "androidx.room:room-compiler:$room_version"
testImplementation 'junit:junit:4.12'
testImplementation "androidx.arch.core:core-testing:2.0.0"
testImplementation "org.amshove.kluent:kluent-android:1.49"
testImplementation "org.mockito:mockito-inline:2.21.0"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.1.0"
testImplementation 'com.jraska.livedata:testing-ktx:1.1.0'
androidTestImplementation 'androidx.test.espresso:espresso-contrib:3.1.1'
androidTestImplementation "androidx.arch.core:core-testing:2.0.0"
androidTestImplementation 'androidx.test.ext:junit:1.1.0'
}
```

(from [T29-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” in the “Kind” drop-down. Click “OK” 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
```

## GETTING A ROOM

---

```
import androidx.room.Entity
import androidx.room.Index
import androidx.room.PrimaryKey
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @field:PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Calendar = Calendar.getInstance(),
    val isCompleted: Boolean = false
) { }
```

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 two things:

1. We want the underlying table name to be `todos`, as opposed to the default, which is the same as the class name (`ToDoEntity`)
2. We want to create an index on our `id` column, since that is our primary key, and so we may be referencing that column a lot

Room knows that `id` is our primary key because we have the `@PrimaryKey` annotation on its property (or, more accurately, on the underlying Java field, courtesy of the `@field:PrimaryKey` form of the annotation). Room wants us to declare some primary key, typically via that `@PrimaryKey` 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, add this nested interface:

```
@Dao
interface Store {
    @Query("SELECT * FROM todos")
    fun all(): LiveData<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String): LiveData<ToDoEntity>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(vararg entities: ToDoEntity)

    @Delete
    suspend fun delete(vararg entities: ToDoEntity)
}
```

(from [T29-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).

## GETTING A ROOM

---

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 LiveData. This has two key effects:

1. Room will perform the queries on a background thread and post the results to the LiveData 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 LiveData, if we do other database operations that affect the todos table, Room will automatically deliver a new result to those observers via the LiveData. 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 —

## GETTING A ROOM

---

you can have @Insert or @Delete functions that accept a single entity, a List of entities, etc.

The entire ToDoEntity class should now look like:

```
package com.commonsware.todo.repo

import androidx.lifecycle.LiveData
import androidx.room.*
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
    val description: String,
    @field:PrimaryKey
    val id: String = UUID.randomUUID().toString(),
    val notes: String = "",
    val createdOn: Calendar = Calendar.getInstance(),
    val isCompleted: Boolean = false
) {
    @Dao
    interface Store {
        @Query("SELECT * FROM todos")
        fun all(): LiveData<List<ToDoEntity>>

        @Query("SELECT * FROM todos WHERE id = :modelId")
        fun find(modelId: String): LiveData<ToDoEntity>

        @Insert(onConflict = OnConflictStrategy.REPLACE)
        suspend fun save(vararg entities: ToDoEntity)

        @Delete
        suspend fun delete(vararg entities: ToDoEntity)
    }
}
```

(from [T29-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

## Step #4: Adding a Database (And Some Type Converters)

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.

## GETTING A ROOM

---

Which means... we need another Kotlin class! Again!

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 `ToDoDatabase`, and set the “Kind” to “Class”. Then, click “OK” to 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-

## GETTING A ROOM

---

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 `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.

## 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 a build error:

```
> Task :app:kaptDebugKotlin FAILED
e: T29-Room/ToDo/app/build/tmp/kapt3/stubs/debug/com/commonsware/todo/repo/
ToDoEntity.java:16: error: Cannot figure out how to save this field into database.
You can consider adding a type converter for it.
    private final java.util.Calendar createdOn = null;
```

The problem is that Room does not know what to do with a `Calendar` 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 `Calendar` 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.commonsware.todo.repo` package in the `java/` directory and choose “New” > “Kotlin File/Class” from the context menu. For the name, fill in `TypeTransmogrifier` and choose “Class” for the “Kind”. Click “OK” to create the

## GETTING A ROOM

---

class, giving you:

```
package com.commonsware.todo.repo

class 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.util.*

class TypeTransmogrifier {
    @TypeConverter
    fun fromCalendar(date: Calendar?): Long? = date?.timeInMillis

    @TypeConverter
    fun toCalendar(millisSinceEpoch: Long?): Calendar? = millisSinceEpoch?.let {
        Calendar.getInstance().apply { timeInMillis = millisSinceEpoch }
    }
}
```

(from [T29-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 `Calendar` objects into `Long` objects, using the time-since-the-Unix-epoch methods on `Calendar`.

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.

## GETTING A ROOM

---

At this point, 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 [T29-Room/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#))

## 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 [T29-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.

ToDoApp now should resemble:

```
package com.commonsware.todo

import android.app.Application
import com.commonsware.todo.repo.ToDoDatabase
```

## GETTING A ROOM

---

```
import com.commonsware.todo.repo.ToDoRepository
import com.commonsware.todo.ui.SingleModelMotor
import com.commonsware.todo.ui.roster.RosterMotor
import org.koin.android.ext.android.startKoin
import org.koin.android.ext.koin.androidContext
import org.koin.androidx.viewmodel.ext.koin.viewModel
import org.koin.dsl.module.module

class ToDoApp : Application() {
    private val koinModule = module {
        single { ToDoDatabase.newInstance(androidContext()) }
        single { ToDoRepository() }
        viewModel { RosterMotor(get()) }
        viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
    }

    override fun onCreate() {
        super.onCreate()

        startKoin(this, listOf(koinModule))
    }
}
```

(from [T29-Room/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Right now, we are not actually using this database anywhere... but we will handle that in the next tutorial.

## 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/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](#)

# Integrating Room Into the Repository

Having a ToDoDatabase is nice. Having our ToDoRepository *use* that ToDoDatabase would be even nicer, as then we would start saving our to-do items to a database, so they would not vanish every time our process is terminated. So, in this tutorial, we will do just that: modify ToDoRepository – and its clients — to work with ToDoDatabase.

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: Getting a Database

First, we need to have our ToDoRepository get access to a ToDoEntity.Store, so that it can work with our ToDoEntity objects.

Update the ToDoRepository to add a constructor parameter:

```
class ToDoRepository(private val store: ToDoEntity.Store) {  
    (from T3o-RoomRepo/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt)
```

Then, in ToDoApp, change the ToDoRepository line in koinModule to be:

```
single {  
    val db: ToDoDatabase = get()  
  
    ToDoRepository(db.todoStore())  
}  
  
(from T3o-RoomRepo/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt)
```

## INTEGRATING ROOM INTO THE REPOSITORY

---

Here, we use `get()` to retrieve our singleton `ToDoDatabase` instance, then call `todoStore()` on it to get the `ToDoEntity.Store`.

This broke our tests. Specifically, `ToDoRepositoryTest` and `RosterListFragmentTest` now have compile errors, as we are not passing in a `ToDoEntity.Store` to our calls to the `ToDoRepository` constructor. We will fix those later in this tutorial.

## Step #2: Fixing the CRUD

Now, we need to have the `ToDoRepository` really use the `ToDoEntity.Store`, rather than just hold onto it.

However, we have a slight issue. `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
    )
}
```

(from [T3o-RoomRepo/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.

## INTEGRATING ROOM INTO THE REPOSITORY

---

Then, replace the contents of ToDoRepository with the following:

```
package com.commonsware.todo.repo

import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations

class ToDoRepository(private val store: ToDoEntity.Store) {
    val items: LiveData<List<ToDoModel>> =
        Transformations.map(store.all()) { all -> all.map { it.toModel() } }

    fun find(id: String): LiveData<ToDoModel> =
        Transformations.map(store.find(id)) { it.toModel() }

    suspend fun save(model: ToDoModel) {
        store.save(ToDoEntity(model))
    }

    suspend fun delete(model: ToDoModel) {
        store.delete(ToDoEntity(model))
    }
}
```

(from [T3o-RoomRepo/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

This replaces all of our manual LiveData work with calls to the ToDoEntity.Store, and delegates our save() and delete() calls to their corresponding ones on ToDoEntity.Store. We use the constructor and toModel() function that we added to ToDoEntity to map from our models to our entities and vice versa.

If you try building the app, though, you will find that we have a problem in SingleModelMotor. Previously, ToDoRepository accepted a null value for the ID, as it was easy enough for ToDoRepository to handle that. However, Room will not want a nullable id value for ToDoEntity.Store and its find() function, as the id in the database can never be null. Somewhere, we need to detect that we have a null value for the id and not bother trying to query the database. The best spot for that is in SingleModelMotor itself, as it is the class that gets the null ID value in the first place.

Modify the states property in SingleModelMotor to be:

```
val states: LiveData<SingleModelState> =
    modelId?.let { Transformations.map(repo.find(modelId)) { SingleModelState(it) } }
    ?: MutableLiveData<SingleModelState>().apply { value = SingleModelState(null) }
```

(from [T3o-RoomRepo/ToDo/app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#))

## INTEGRATING ROOM INTO THE REPOSITORY

---

If we have a non-null modelId, we use the repository as we did before, using the model that we get back in our SingleModelState. If, on the other hand, we have a null value for modelId, we can skip the repository entirely and return a MutableLiveData pre-populated with a SingleModelState that has our null model. Consumers of states then do not care whether our SingleModelState is based on a database call or not — they just react to the changing states as they arrive.

At this point, if you try out the app, it should mostly work as before... except that it will remember your to-do items, even if you terminate your process (e.g., swipe the app off of the overview screen).

The tests, however, need a bit more work.

## Step #3: Fixing the Instrumented Test

Let's start by fixing RosterListFragmentTest, as it is a bit simpler to adjust.

First, remove these lines:

```
@get:Rule  
val instantTaskExecutorRule = InstantTaskExecutorRule()
```

We will be using Room, and Room is not compatible with this rule. Besides, we will no longer need it.

The other thing that we need to do is to pass a ToDoEntity.Store to the ToDoRepository constructor. We could get one from Koin, but that will be the real database. Ideally, in a test, you do not want to save things to disk or the network, as that will slow things down. It will also make it more difficult to reset the environment to the state that it was in before the test, so you can run your tests with consistent conditions.

We could use a mock here, but since this is an instrumented test and we have the full power of Room, we can use a different approach: a memory-only SQLite database.

In ToDoDatabase, add this function to the companion object:

```
fun newTestInstance(context: Context) =  
    Room.inMemoryDatabaseBuilder(context, ToDoDatabase::class.java).build()
```

## INTEGRATING ROOM INTO THE REPOSITORY

---

(from [T3o-RoomRepo/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#))

`Room.databaseBuilder()` is for a real persistent database.

`Room.inMemoryDatabaseBuilder()` is for an in-memory database. Whatever you do with this database will only be held in RAM, and that memory will be released when we are done with the database.

At this point, `ToDoDatabase` should contain:

```
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()

        fun newTestInstance(context: Context) =
            Room.inMemoryDatabaseBuilder(context, ToDoDatabase::class.java).build()
    }
}
```

(from [T3o-RoomRepo/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#))

Then, change the `setUp()` function in `RosterListFragmentTest` to be:

```
@Before
fun setUp() {
    val context = InstrumentationRegistry.getInstrumentation().targetContext
    val db = ToDoDatabase.newTestInstance(context)

    repo = ToDoRepository(db.todoStore())

    loadKoinModules(module {
        single(override = true) { repo }
    })
}
```

## INTEGRATING ROOM INTO THE REPOSITORY

---

```
    runBlocking { items.forEach { repo.save(it) } }
```

(from [T30-RoomRepo/ToDo/app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

Here, we get an in-memory database using our new `newTestInstance()` function. For the context, we use the somewhat cumbersome `InstrumentationRegistry.getInstrumentation().targetContext` syntax, to give us a Context object associated with the code being tested.

If you run the revised test, it should work as it did before.

## Step #4: Fixing the Unit Test

The biggest headache when unit testing Room comes from the core premise of unit tests: we are running on your development machine, not on Android. Not only do methods like `Room.inMemoryDatabaseBuilder()` want a Context (which we do not have), but the Room implementation by default will try to use a framework copy of SQLite (which we do not have). So, we cannot use Room very easily in our unit tests.

One solution would be to move `ToDoRepositoryTest` from `test` to `androidTest` and make it be an instrumented test. Then, we could test it much as we did in `RosterListFragmentTest`. For an app like this, with few tests, we could get away with that.

The other solution is to create some sort of mock implementation of `ToDoEntity.Store`. It would need to be “real enough” that `ToDoRepository` could work with it, so using a Mockito/Kluent mock might be troublesome. On the other hand, the API of `ToDoEntity.Store` is fairly limited, so we can create an in-memory implementation backed by a `List`, much as `ToDoRepository` itself was implemented using a `List`, before we swapped in the `ToDoEntity.Store` code.

With that in mind, add this class inside `ToDoRepositoryTest`:

```
class TestStore : ToDoEntity.Store {
    private val _items =
        MutableLiveData<List<ToDoEntity>>().apply { value = listOf() }

    override fun all(): LiveData<List<ToDoEntity>> = _items

    override suspend fun save(vararg entities: ToDoEntity) {
        entities.forEach { entity ->
```

## INTEGRATING ROOM INTO THE REPOSITORY

---

```
_items.value = if (current().any { it.id == entity.id }) {
    current().map { if (it.id == entity.id) entity else it }
} else {
    current() + entity
}
}

override suspend fun delete(vararg entities: ToDoEntity) {
    entities.forEach { entity ->
        _items.value = current().filter { it.id != entity.id }
    }
}

override fun find(modelId: String): LiveData<ToDoEntity> =
    Transformations.map(_items) {
        it.find { model -> model.id == modelId }
    }

private fun current() = _items.value!!
}
```

(from [T3o-RoomRepo/ToDo/app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

This uses the same basic code as we used to have in `ToDoRepository`, just tweaked for the `ToDoEntity.Store` API, such as working with `ToDoEntity` objects instead of `ToDoModel` objects.

Then, change the `setUp()` function to use this new `TestStore`:

```
@Before
fun setUp() {
    underTest = ToDoRepository(TestStore())
}
```

(from [T3o-RoomRepo/ToDo/app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

If you run this test, it should pass as it did before.

## 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/ToDoRepository.kt](#)

## INTEGRATING ROOM INTO THE REPOSITORY

---

- [app/src/main/java/com/commonsware/todo/ToDoApp.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#)
- [app/src/main/java/com/commonsware/todo/ui/SingleModelMotor.kt](#)
- [app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#)
- [app/src/main/java/com/commonsware/todo/repo/ToDoDatabase.kt](#)
- [app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#)

# 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](#).

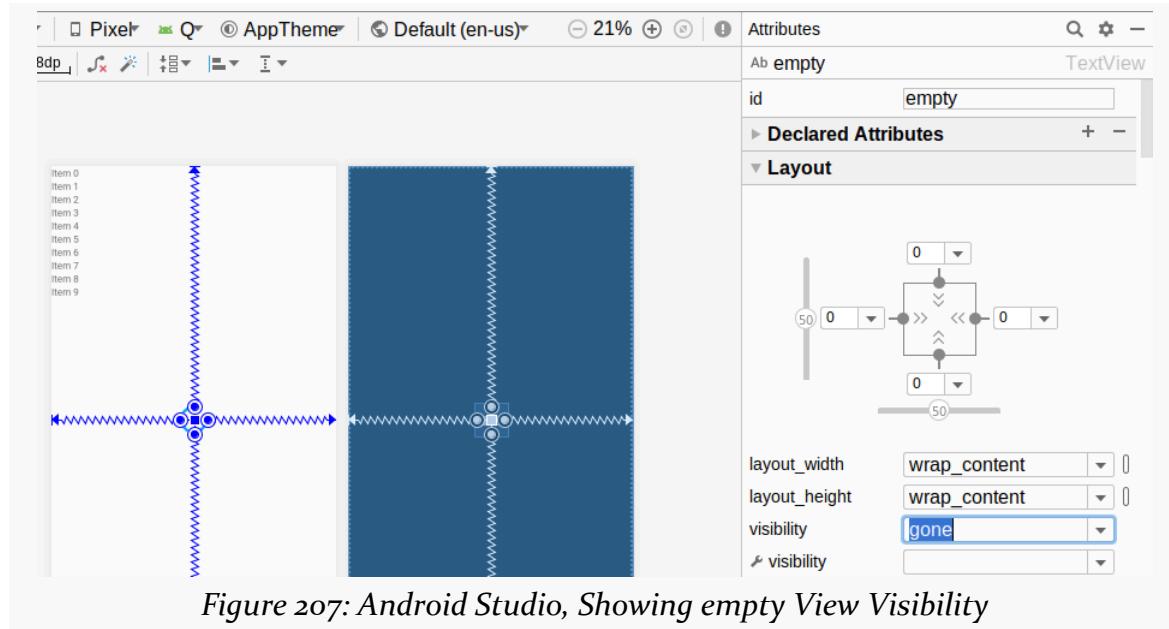
## Step #1: Adjusting Our Layout

We need to make a couple of changes to the layout used by `RosterListFragment`.

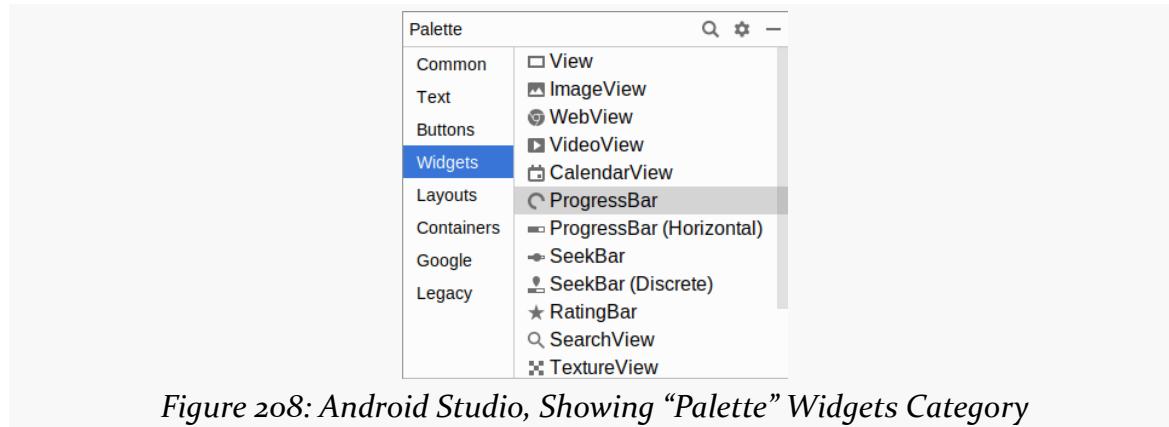
Open `res/layout/todo_roster.xml` in the IDE. Click on the empty `TextView` in the

## TRACKING OUR LOAD STATUS

“Component Tree”. In the list of all attributes, find the `visibility` attribute, and set it to `gone`:



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:



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

## TRACKING OUR LOAD STATUS

---

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 sometimes get a constraint that ties the ProgressBar to the items RecyclerView, instead of to the parent ConstraintLayout, such as:

```
<ProgressBar  
    android:id="@+id/progressBar"  
    style="?android:attr/progressBarStyle"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginBottom="8dp"  
    android:layout_marginEnd="8dp"  
    android:layout_marginStart="8dp"  
    android:layout_marginTop="8dp"  
    app:layout_constraintBottom_toBottomOf="@+id/items"  
    app:layout_constraintEnd_toEndOf="parent"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toTopOf="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.

Your resulting layout 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">
```

## TRACKING OUR LOAD STATUS

---

```
<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"
    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.recyclerview.widget.RecyclerView>
</androidx.constraintlayout.widget.ConstraintLayout>
```

(from [T31-Load/ToDo/app/src/main/res/layout/todo\\_roster.xml](#))

## Step #2: 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.

As it turns out, our observer will not receive a RosterViewState until the data is loaded. So, as soon as we get a RosterViewState, we can hide the ProgressBar.

## TRACKING OUR LOAD STATUS

---

So, change the observer in the `onViewCreated()` function of `RosterListFragment` to be:

```
motor.states.observe(this, Observer { state ->
    adapter.submitList(state.items)
    empty.visibility = if (state.items.isEmpty()) View.VISIBLE else View.GONE
    loading.visibility = View.GONE
})
```

(from [T31-Load/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

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`, the simplest solution is to comment out the observer code from the preceding code listing. Then, we will not react to any `RosterViewState`, so the UI remains in its initial state, where our `ProgressBar` is visible. Just be sure to uncomment that code when you are done.

## 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/RosterListFragment.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# 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: Updating Our Queries

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. However, while we are working with the `ToDoEntity.Store`, let's add one more thing: sorting the items by description. Right now, the items come back unsorted, which is not ideal.

With that in mind, change the SQL in the `@Query` annotation for the `all()` function to add `ORDER BY description` at the end:

```
@Query("SELECT * FROM todos ORDER BY description")
fun all(): LiveData<List<ToDoEntity>>
```

(from [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

Then, add this `filtered()` function:

```
@Query("SELECT * FROM todos WHERE isCompleted = :isCompleted ORDER BY description")
fun filtered(isCompleted: Boolean): LiveData<List<ToDoEntity>>
```

(from [T32-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 `LiveData`.

At this point, `ToDoEntity.Store` should look like:

```
@Dao
interface Store {
    @Query("SELECT * FROM todos ORDER BY description")
    fun all(): LiveData<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE isCompleted = :isCompleted ORDER BY description")
    fun filtered(isCompleted: Boolean): LiveData<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId ORDER BY description")
    fun find(modelId: String): LiveData<ToDoEntity>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(vararg entities: ToDoEntity)

    @Delete
    suspend fun delete(vararg entities: ToDoEntity)
}
```

(from [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt](#))

## 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 [T32-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`:

```
private fun filteredEntities(filterMode: FilterMode) = when (filterMode) {
    FilterMode.ALL -> store.all()
    FilterMode.OUTSTANDING -> store.filtered(isCompleted = false)
    FilterMode.COMPLETED -> store.filtered(isCompleted = true)
}
```

(from [T32-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` property in `ToDoRepository` with this similar `items()` function:

```
fun items(filterMode: FilterMode = FilterMode.ALL): LiveData<List<ToDoModel>> =
    Transformations.map(filteredEntities(filterMode)) { all -> all.map { it.toModel() } }
```

## FILTERING OUR ITEMS

---

(from [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

As with the former items property, we use Transformations.map() to convert a LiveData of entities to a LiveData of models. However, this time, we use the new filteredEntities() function to get the LiveData of entities. To help simplify our test code — which is now broken, a problem we will need to address — 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).

At this point, ToDoRepository.kt should resemble:

```
package com.commonsware.todo.repo

import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations

enum class FilterMode { ALL, OUTSTANDING, COMPLETED }

class ToDoRepository(private val store: ToDoEntity.Store) {
    fun items(filterMode: FilterMode = FilterMode.ALL): LiveData<List<ToDoModel>> =
        Transformations.map(filteredEntities(filterMode)) { all -> all.map { it.toModel() } }

    fun find(id: String): LiveData<ToDoModel> =
        Transformations.map(store.find(id)) { it.toModel() }

    suspend fun save(model: ToDoModel) {
        store.save(ToDoEntity(model))
    }

    suspend fun delete(model: ToDoModel) {
        store.delete(ToDoEntity(model))
    }

    private fun filteredEntities(filterMode: FilterMode) = when (filterMode) {
        FilterMode.ALL -> store.all()
        FilterMode.OUTSTANDING -> store.filtered(isCompleted = false)
        FilterMode.COMPLETED -> store.filtered(isCompleted = true)
    }
}
```

(from [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

## Step #4: Augmenting Our Motor

RosterMotor now no longer compiles, as it relied on the items property on ToDoRepository that we replaced with the items() function. Plus, we need to have a way for the the RosterListFragment to request a particular FileMode.

First, add a FileMode property to RosterViewState:

## FILTERING OUR ITEMS

---

```
class RosterViewState(  
    val items: List<ToDoModel> = listOf(),  
    val filterMode: FilterMode = FilterMode.ALL  
)
```

(from [T32-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 = MediatorLiveData<RosterViewState>()  
    val states: LiveData<RosterViewState> = _states  
    private var lastSource: LiveData<List<ToDoModel>>? = null  
  
    init {  
        load(FilterMode.ALL)  
    }  
  
    fun load(filterMode: FilterMode) {  
        lastSource?.let { _states.removeSource(it) }  
  
        val items = repo.items(filterMode)  
  
        _states.addSource(items) { models ->  
            _states.value = RosterViewState(models, filterMode)  
        }  
  
        lastSource = items  
    }  
  
    fun save(model: ToDoModel) {  
        viewModelScope.launch(Dispatchers.Main) {  
            repo.save(model)  
        }  
    }  
}
```

(from [T32-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:

## FILTERING OUR ITEMS

---

```
val states: LiveData<RosterViewState> =  
    Transformations.map(repo.items) { RosterViewState(it) }
```

That is because we always loaded all of the to-do items.

We could have replaced it with:

```
val states: LiveData<RosterViewState> =  
    Transformations.map(repo.items()) { RosterViewState(it) }
```

This would work... 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* `LiveData` than the one we had originally. That highlights a limitation of `Transformations.map()`: it can only map *one* `LiveData`. 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 `LiveData` supplying the `RosterViewState` objects, rather than having to know to subscribe to different `LiveData` objects at different times for different reasons. So, we have one or more `LiveData` objects with our items, and we want to funnel them all into a single `LiveData` of `RosterViewState` objects.

The solution for that is `MediatorLiveData`.

`MediatorLiveData` is what underlies `Transformations.map()`. `MediatorLiveData` can observe one or several other `LiveData` objects. When values change in those `LiveData` objects, `MediatorLiveData` will invoke a lambda expression that you provide. There, you can convert the value to whatever you need and update the `MediatorLiveData` itself based on that change. You can add and remove `LiveData` objects from the `MediatorLiveData` whenever you need.

So, what `RosterMotor` is doing is using a `MediatorLiveData` as the stable `LiveData` that `RosterListFragment` observes. As we change filter modes, we swap in a new `LiveData` source for the `MediatorLiveData`.

With all that in mind...

- `_states` is our `MediatorLiveData`, and it is `private`
- Since we do not want `RosterListFragment` to know the implementation

## FILTERING OUR ITEMS

---

- details, states is a plain LiveData property that happens to point to the MediatorLiveData in \_states
- lastSource is the last LiveData that we retrieved from ToDoRepository via a call to items() (or null when we start out)
  - load() will remove lastSource from the MediatorLiveData, make a fresh call to items() to get the items based on the new FilterMode, add that LiveData to the MediatorLiveData (with a lambda to convert the list of models into a RosterViewState), and hold onto that LiveData in lastSource
  - init {} triggers our initial load() call, to retrieve ALL items

Overall, RosterMotor.kt should now be:

```
package com.commonsware.todo.ui.roster

import androidx.lifecycle.LiveData
import androidx.lifecycle.MediatorLiveData
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.Dispatchers
import kotlinx.coroutines.launch

class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val filterMode: FilterMode = FilterMode.ALL
)

class RosterMotor(private val repo: ToDoRepository): ViewModel() {
    private val _states = MediatorLiveData<RosterViewState>()
    val states: LiveData<RosterViewState> = _states
    private var lastSource: LiveData<List<ToDoModel>>? = null

    init {
        load(FilterMode.ALL)
    }

    fun load(filterMode: FilterMode) {
        lastSource?.let { _states.removeSource(it) }

        val items = repo.items(filterMode)

        _states.addSource(items) { models ->
            _states.value = RosterViewState(models, filterMode)
        }
    }
}
```

## FILTERING OUR ITEMS

---

```
    lastSource = items
}

fun save(model: ToDoModel) {
    viewModelScope.launch(Dispatchers.Main) {
        repo.save(model)
    }
}
```

(from [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

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: Repairing Our Test

However, ToDoRepositoryTest will no longer compile at this point, due to the changes that we made to ToDoRepository.

First, change all of the references to the `items` property on `underTest` to be calls to the `items()` function. You do not need to provide any parameters to `items()` — the default value of `FilterMode.ALL` is what our current test code needs.

We also now need to implement `filtered()` on `TestStore`, as that class no longer fully implements the `ToDoEntity.Store` interface. So, add this function to `TestStore`:

```
override fun filtered(isCompleted: Boolean) =
    MutableLiveData<List<ToDoEntity>>()
    .apply { value = current().filter { it.isCompleted == isCompleted } }
```

(from [T32-Filter/ToDo/app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

Here, we:

- Get the current list of items
- Use Kotlin's `filter()` function to find the subset of items matching the desired completion status
- Wrap that in a `MutableLiveData` and return it

If you run the test, it should work.

## FILTERING OUR ITEMS

---

At this point, `ToDoRepositoryTest` should resemble:

```
package com.commonsware.todo.repo

import androidx.arch.core.executor.testing.InstantTaskExecutorRule
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.Transformations
import com.jraska.livedata.test
import kotlinx.coroutines.runBlocking
import org.amshove.kluent.shouldBeEmpty
import org.amshove.kluent.shouldContainSame
import org.amshove.kluent.shouldEqual
import org.junit.Before
import org.junit.Rule
import org.junit.Test

class ToDoRepositoryTest {
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    private lateinit var underTest: ToDoRepository

    @Before
    fun setUp() {
        underTest = ToDoRepository(TestStore())
    }

    @Test
    fun `can add items`() {
        val testModel = ToDoModel("test model")

        underTest.apply {
            items().test().value().shouldBeEmpty()

            runBlocking { save(testModel) }

            items().test().value() shouldContainSame listOf(testModel)

            find(testModel.id).test().value() shouldEqual testModel
        }
    }

    @Test
    fun `can modify items`() {
        val testModel = ToDoModel("test model")
        val replacement = testModel.copy(notes = "This is the replacement")
```

## FILTERING OUR ITEMS

---

```
underTest.apply {
    items().test().value().shouldBeEmpty()

    runBlocking { save(testModel) }

    items().test().value() shouldContainSame listOf(testModel)

    runBlocking { save(replacement) }

    items().test().value() shouldContainSame listOf(replacement)
}

}

@Test
fun `can remove items`() {
    val testModel = ToDoModel("test model")

    underTest.apply {
        items().test().value().shouldBeEmpty()

        runBlocking { save(testModel) }

        items().test().value() shouldContainSame listOf(testModel)

        runBlocking { delete(testModel) }

        items().test().value().shouldBeEmpty()
    }
}

class TestStore : ToDoEntity.Store {
    private val _items =
        MutableLiveData<List<ToDoEntity>>().apply { value = listOf() }

    override fun all(): LiveData<List<ToDoEntity>> = _items

    override fun filtered(isCompleted: Boolean) =
        MutableLiveData<List<ToDoEntity>>()
            .apply { value = current().filter { it.isCompleted == isCompleted } }

    override suspend fun save(vararg entities: ToDoEntity) {
        entities.forEach { entity ->
            _items.value = if (current().any { it.id == entity.id }) {
                current().map { if (it.id == entity.id) entity else it }
            } else {
                current() + entity
            }
        }
    }
}
```

## FILTERING OUR ITEMS

---

```
    }
}

override suspend fun delete(vararg entities: ToDoEntity) {
    entities.forEach { entity ->
        _items.value = current().filter { it.id != entity.id }
    }
}

override fun find(modelId: String): LiveData<ToDoEntity> =
    Transformations.map(_items) {
        it.find { model -> model.id == modelId }
    }

private fun current() = _items.value!!
}
```

(from [T32-Filter/ToDo/app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt](#))

## Step #6: Adding a Checkable Submenu

We have added quite a few action 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 action 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:

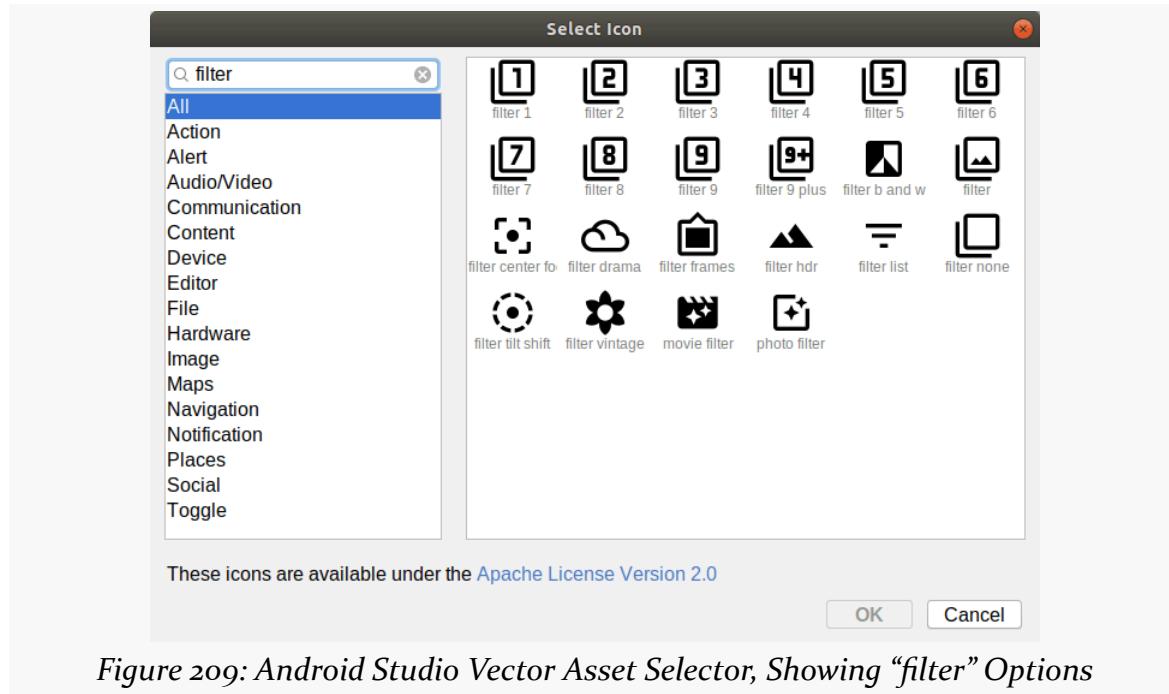


Figure 209: Android Studio Vector Asset Selector, Showing “filter” Options

Choose the “filter list” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

## FILTERING OUR ITEMS

Next, open up the `res/menu/actions_roster.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 “add” item:

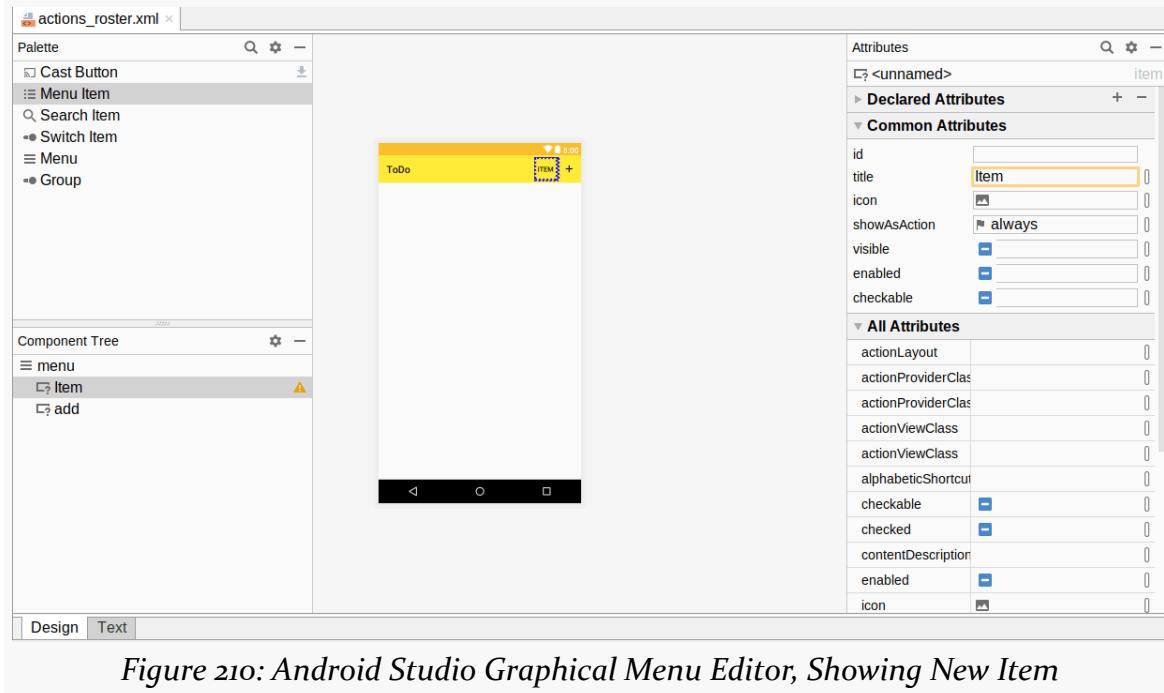


Figure 210: Android Studio Graphical Menu Editor, Showing New Item

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_list_black_24dp` 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 action bar item:

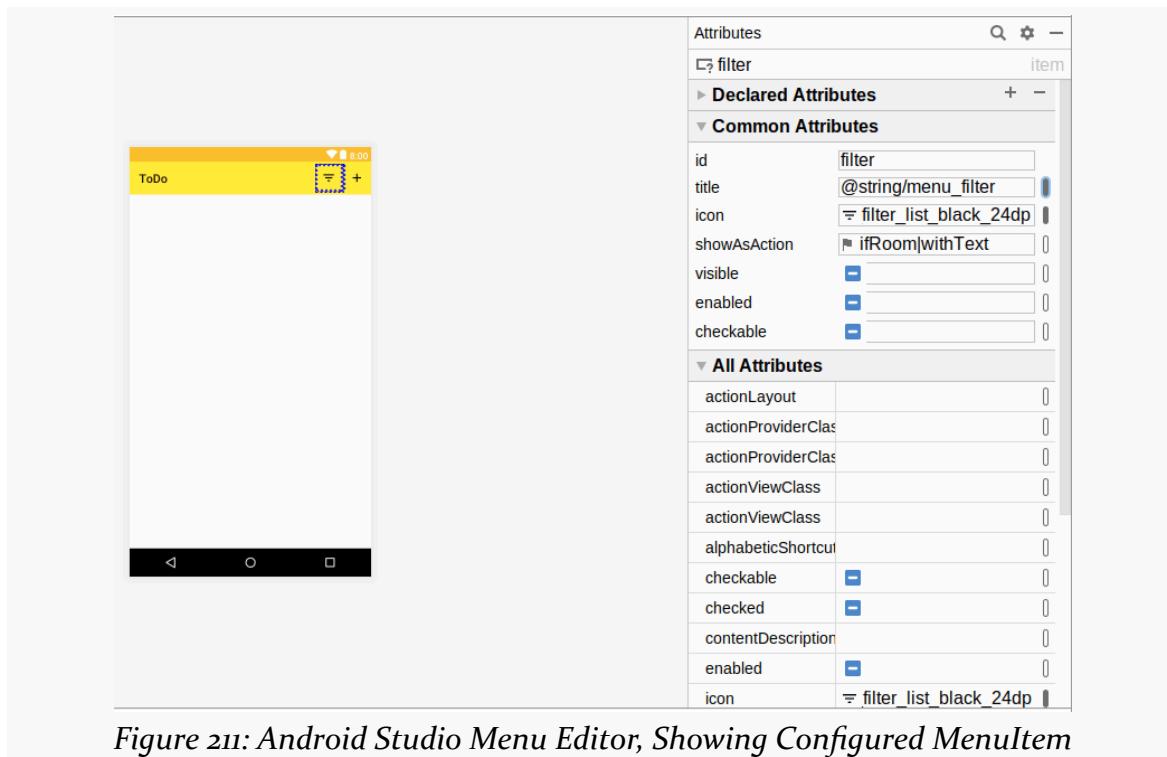


Figure 211: Android Studio Menu Editor, Showing Configured MenuItem

That gives the user something to click on, but now we need to set up a submenu. Unfortunately, at this point, the drag-and-drop functionality of the menu editor has a bug — we cannot create a submenu this way.

Instead, switch over to the “Text” sub-tab and add a `<menu>` element as a child of the “filter” `<item>` element manually:

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

    <item
        android:id="@+id/filter"
        android:icon="@drawable/ic_filter_list_black_24dp"
        android:showAsAction="ifRoom|withText"
        android:title="@string/menu_filter">
```

## FILTERING OUR ITEMS

```
<menu>  
    </menu>  
    </item>  
    <item  
        android:id="@+id/add"  
        android:icon="@drawable/ic_add_black_24dp"  
        android:showAsAction="ifRoom|withText"  
        android:title="@string/menu_add" />  
</menu>
```

If you then switch back to the “Design” sub-tab, you should see the “menu” entry below our “filter” item in the Component Tree, though you may have to expand the tree yourself:

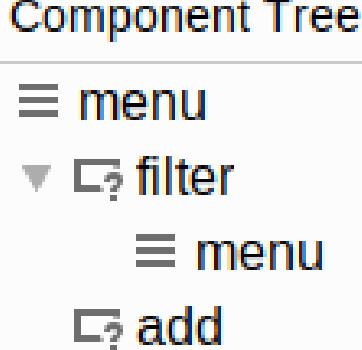


Figure 212: Android Studio Menu Editor, Showing New Submenu

Then, from the Palette, drag a “Group” into the new “menu” in the Component Tree:

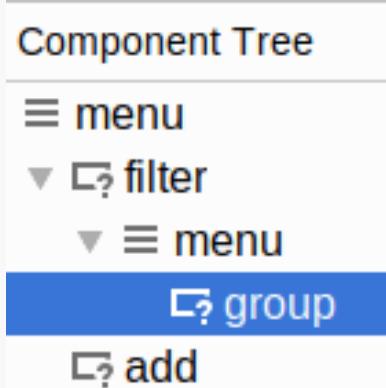


Figure 213: Android Studio Menu Editor, Showing New Group

## FILTERING OUR ITEMS

---

In the Attributes pane, give the group an ID of filter\_group and set the “checkableBehavior” to “single”.

Then, from the Palette, drag a “Menu Item” into the new group in the Component Tree:

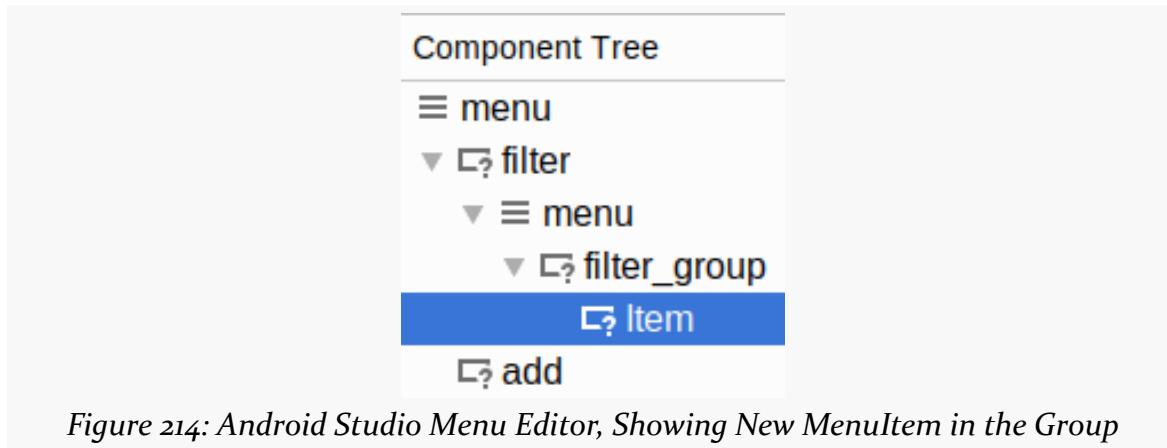


Figure 214: Android Studio Menu Editor, Showing New MenuItem in the Group

Drag two more “Menu Item” 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 “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”.

At this point, in the “Text” sub-tab, your menu resource XML should resemble:

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

## FILTERING OUR ITEMS

---

```
<item
    android:id="@+id/filter"
    android:icon="@drawable/ic_filter_list_black_24dp"
    android:showAsAction="ifRoom|withText"
    android:title="@string/menu_filter">
    <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_black_24dp"
    android:showAsAction="ifRoom|withText"
    android:title="@string/menu_add" />
</menu>
```

(from [ToDo/app/src/main/res/menu/actions\\_roster.xml](#))

## FILTERING OUR ITEMS

---

And, if you run your app, you should see the new filter action bar item. Clicking it will expose the submenu, although clicking on the submenu items will have no effect.

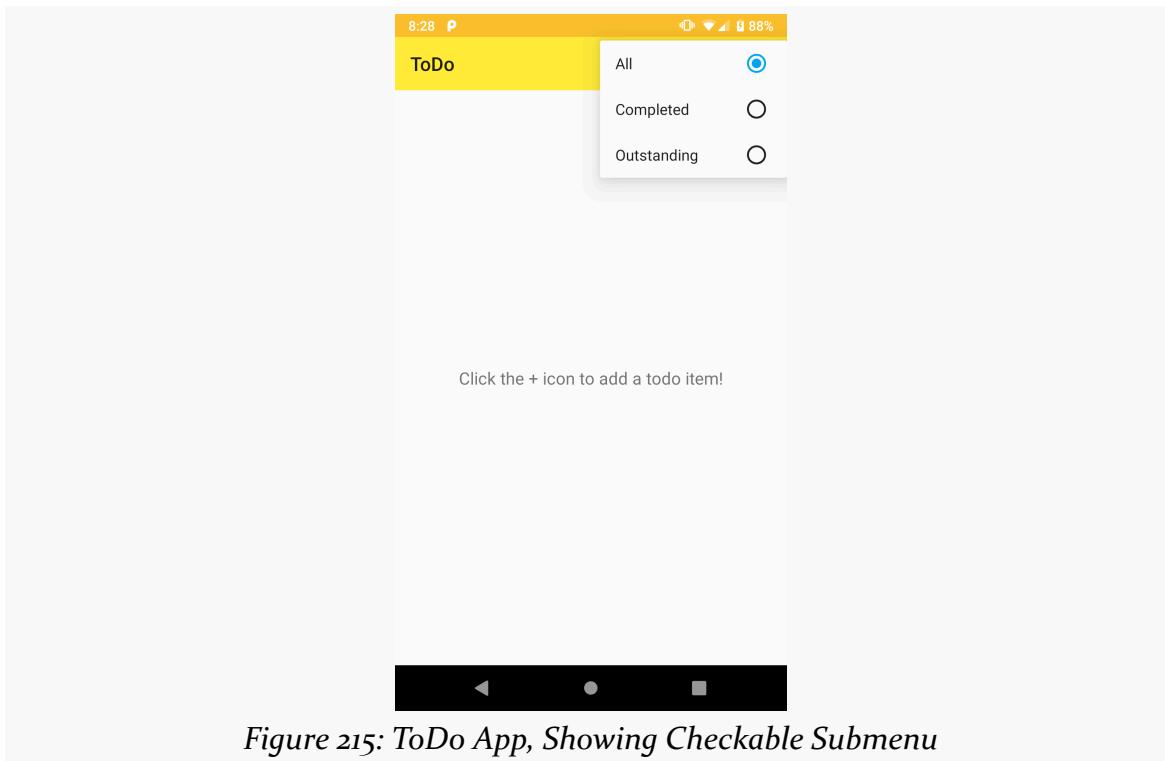


Figure 215: ToDo App, Showing Checkable Submenu

## Step #7: Getting Control on Filter Choices

In particular, clicking on the submenu items does not even change their checked state. Even though our submenu looks like a group of radio buttons, it does not automatically behave like one. 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
        }
    }
}
```

## FILTERING OUR ITEMS

---

```
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 [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

`onOptionsItemSelected()` will get called when the user clicks on the checkable sub-menu 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
- “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 #8: 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

## FILTERING OUR ITEMS

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 [T32-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)

Next, open `res/layout/todo_roster.xml` in the IDE. Click on our empty `TextView`. In the “All Attributes” section of the “Attributes” pane, fold open the “padding” options:

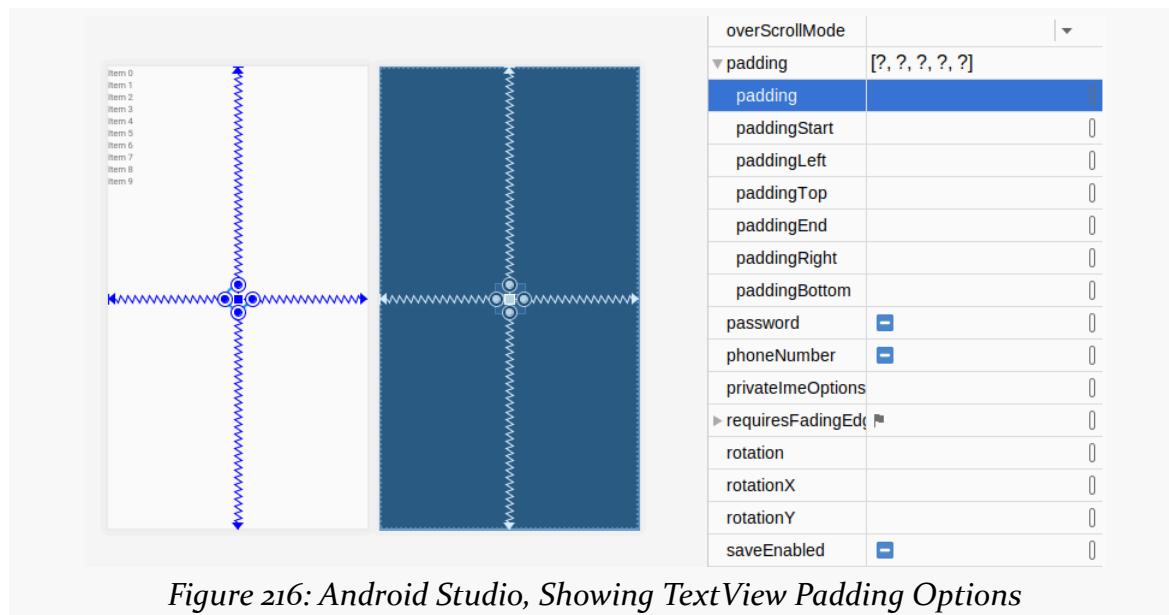
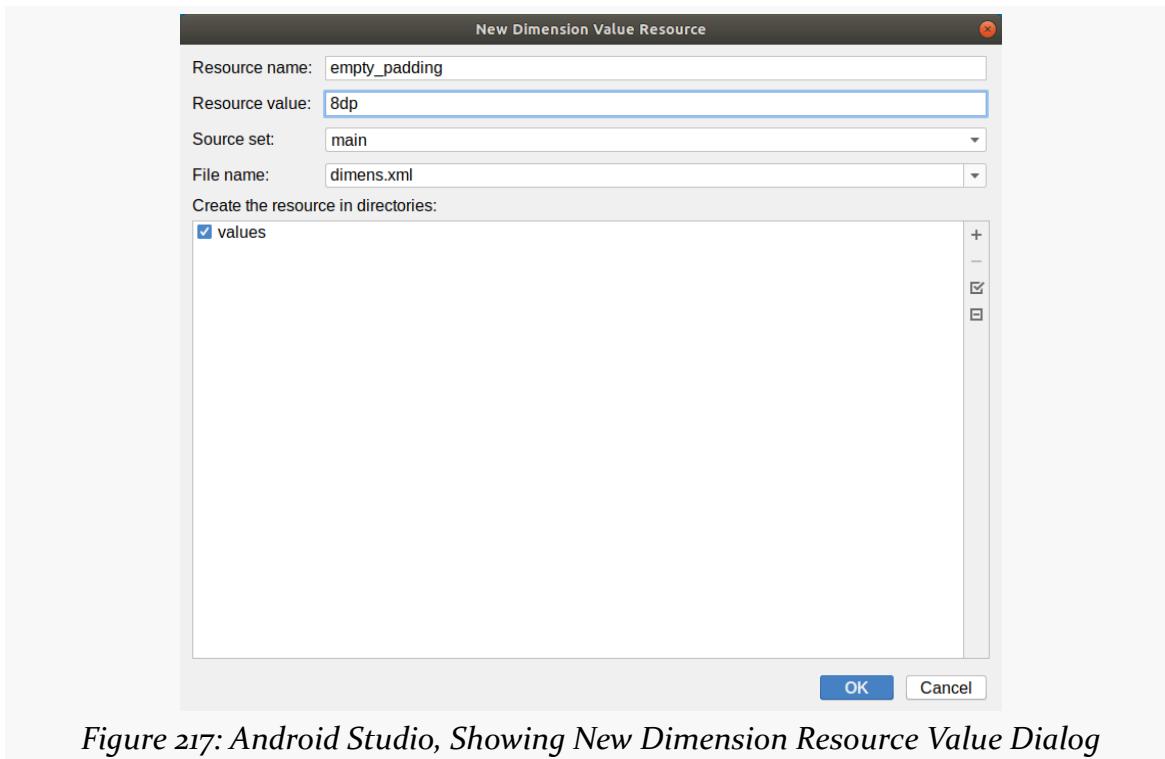


Figure 216: Android Studio, Showing TextView Padding Options

## FILTERING OUR ITEMS

---

Click the “O” button next to the “all” entry in the “Padding” options. In there, create a new dimension resource, named `empty_padding`, with a value of `8dp`:



*Figure 217: Android Studio, Showing New Dimension Resource Value Dialog*

Click OK to close up the dialogs and assign that dimension resource to the padding for all four sides. This will prevent our empty message from running all the way to the edges of the screen.

## FILTERING OUR ITEMS

Then, open the “gravity” options:

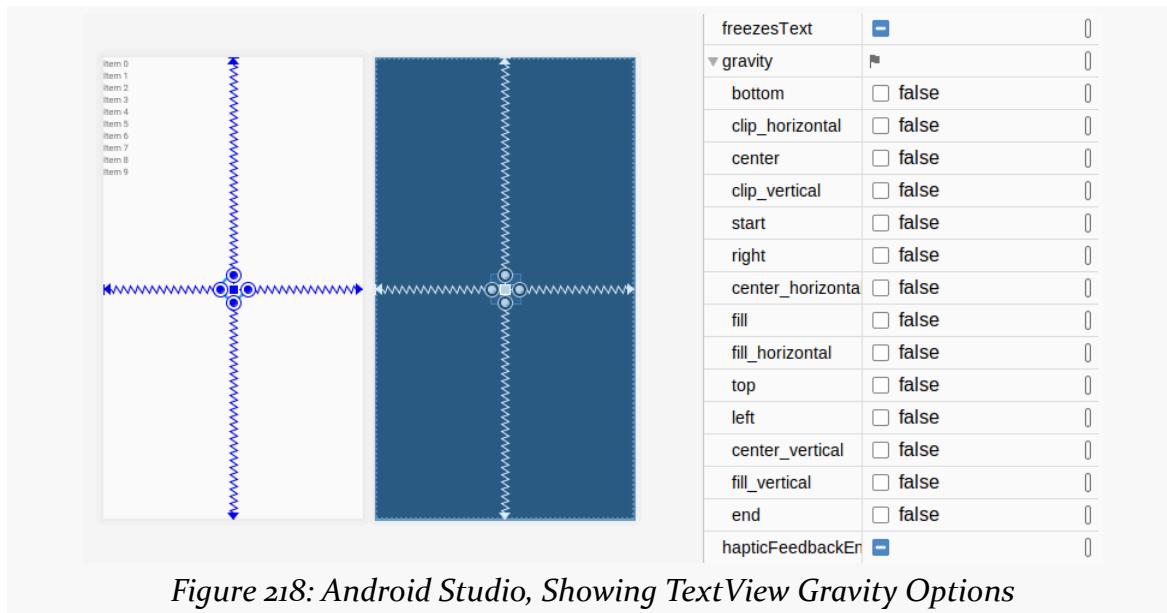


Figure 218: Android Studio, Showing TextView Gravity Options

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 RosterState observer in onViewCreated() to be:

```
motor.states.observe(this, Observer { state ->
    adapter.submitList(state.items)

    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
    }

    loading.visibility = View.GONE
})
```

## FILTERING OUR ITEMS

---

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:

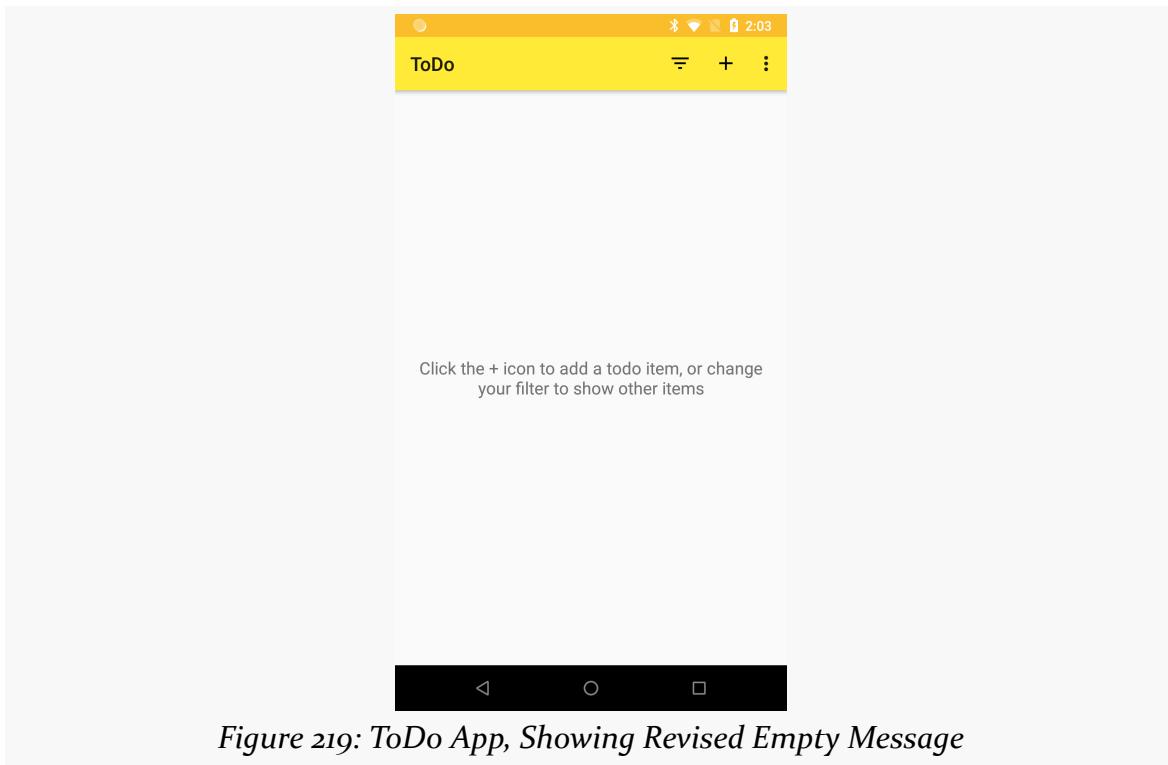


Figure 219: ToDo App, Showing Revised Empty Message

## Step #9: 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.

## FILTERING OUR ITEMS

---

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 [T32-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))
    }

    motor.states.value?.let { menuMap[it.filterMode]?.isChecked = true }

    super.onCreateOptionsMenu(menu, inflater)
}
```

(from [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Here, we:

- 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

Then, add this line to the bottom of the RosterViewState observer that we set up in onViewCreated():

```
menuMap[state.filterMode]?.isChecked = true
```

## FILTERING OUR ITEMS

---

(from [T32-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.

The overall observer now should look like:

```
motor.states.observe(this, Observer { state ->
    adapter.submitList(state.items)

    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
    }

    loading.visibility = View.GONE
    menuMap[state.filterMode]?.isChecked = true
})
}
```

(from [T32-Filter/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

And, if you run the app, you should see that the filtering applied to the list matches the checked MenuItem, even after some navigation.

## 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/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.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](#)

## FILTERING OUR ITEMS

---

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

---

## **Additional Features**

---

Licensed solely for use by Patrocinio Rodriguez

# 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.

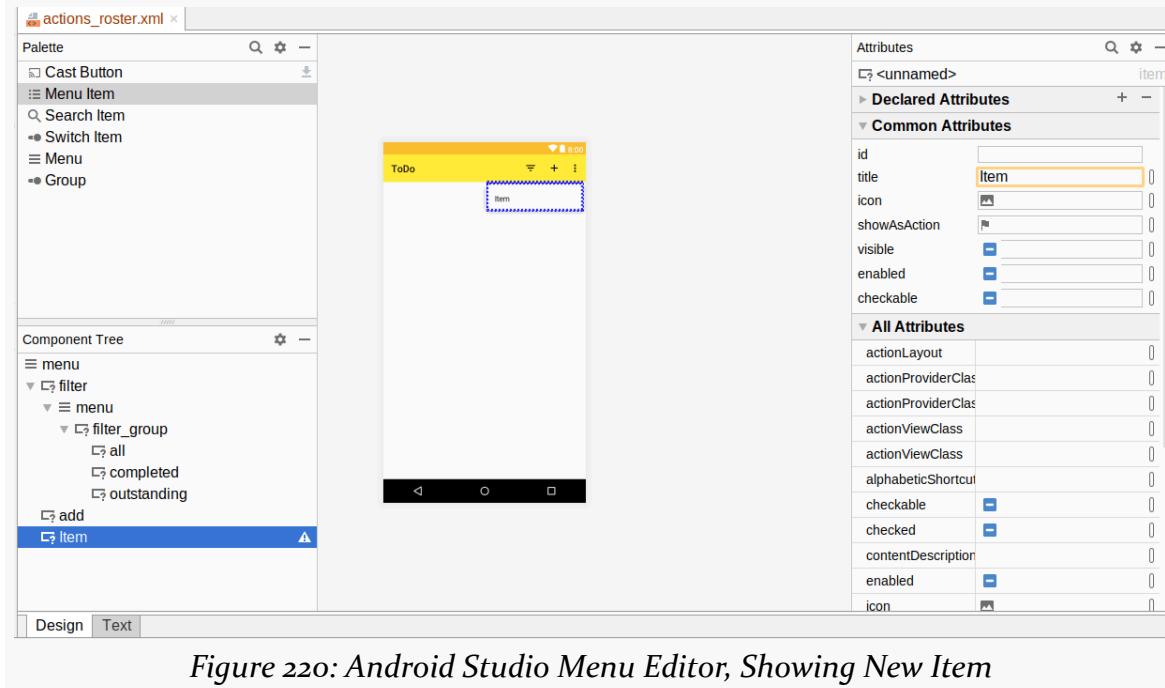
This is a continuation of the work we did in [the previous tutorial](#). The book's GitLab repository contains [the results of the previous tutorial](#) as well as [the results of completing the work in this tutorial](#).

## Step #1: Adding a Save Action Bar Item

It's time for another action bar item! This time, though, it uses an existing icon and string resource, as we already have a "save" action bar item in the `EditFragment`. We will reuse that for a "save" action bar item in the `RosterListFragment`.

## GENERATING A REPORT

Open up the `res/menu/actions_roster.xml` resource file, and switch to the “Design” sub-tab. Drag an “Item” from the “Palette” view into the Component Tree, slotting it after the existing “add” 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_black_24dp` 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:

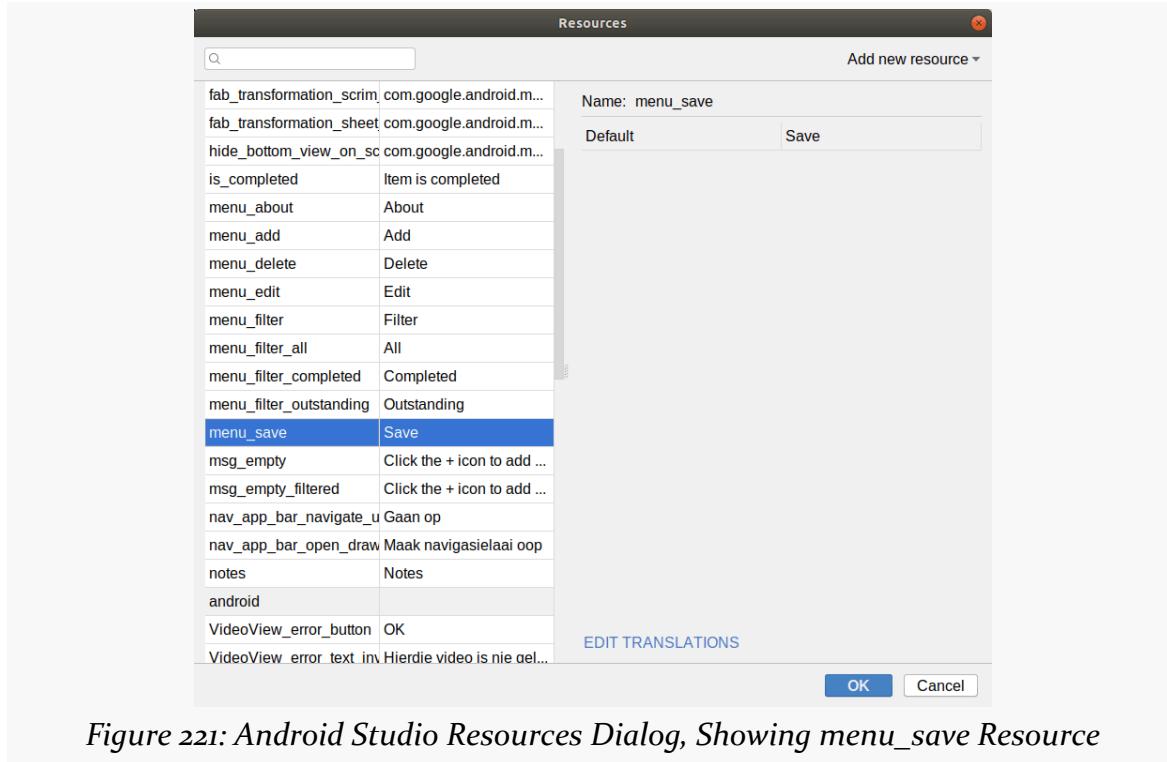


Figure 221: Android Studio Resources Dialog, Showing `menu_save` Resource

Double-click on it to choose it and complete configuration of our action 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 `ACTION_CREATE_DOCUMENT`, an Intent action that, in Android, fills a role akin to the “file ‘save as’” dialogs that you see in desktop operating systems.

`ACTION_CREATE_DOCUMENT` works with `startActivityForResult()`, where we will be given the location that the user chose after the user chooses it. `startActivityForResult()` takes an `int` to identify this request from others. So, add this constant to `RosterListFragment`:

## GENERATING A REPORT

```
private const val REQUEST_SAVE = 1337
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Then, add this saveReport() function to RosterListFragment:

```
private fun saveReport() {
    val intent = Intent(Intent.ACTION_CREATE_DOCUMENT)
        .addCategory(Intent.CATEGORY_OPENABLE)
        .setType("text/html")

    startActivityForResult(intent, REQUEST_SAVE)
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Here, we create an ACTION\_CREATE\_DOCUMENT Intent, with CATEGORY\_OPENABLE, so we are sure to get a location on which we can open a stream to write our report to. We indicate that the MIME type of the resulting content will be text/html, the MIME type for Web pages. Then, we call startActivityForResult() to make the request.

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 [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

Your overall onOptionsItemSelected() function should resemble:

```
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
        }
    }
}
```

## GENERATING A REPORT

```
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)
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

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:

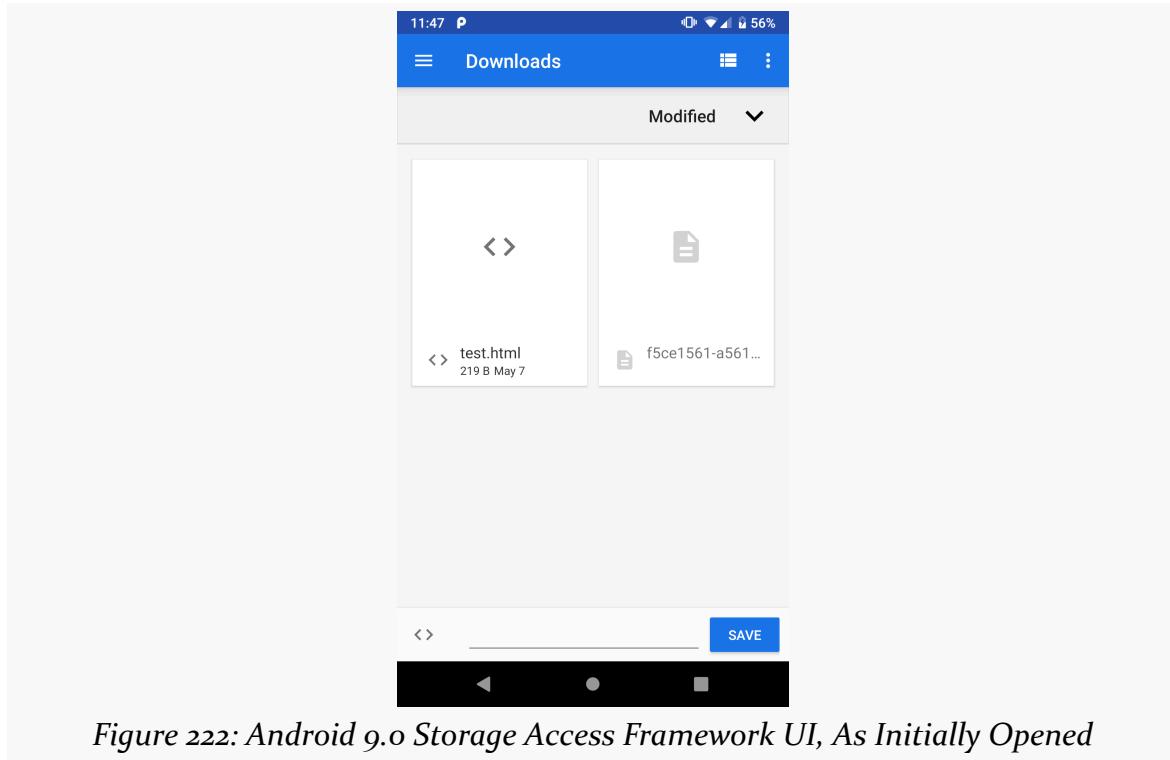


Figure 222: Android 9.0 Storage Access Framework UI, As Initially Opened

## GENERATING A REPORT

The user can choose “Show internal storage” from the overflow menu to add more options of where to save the content:

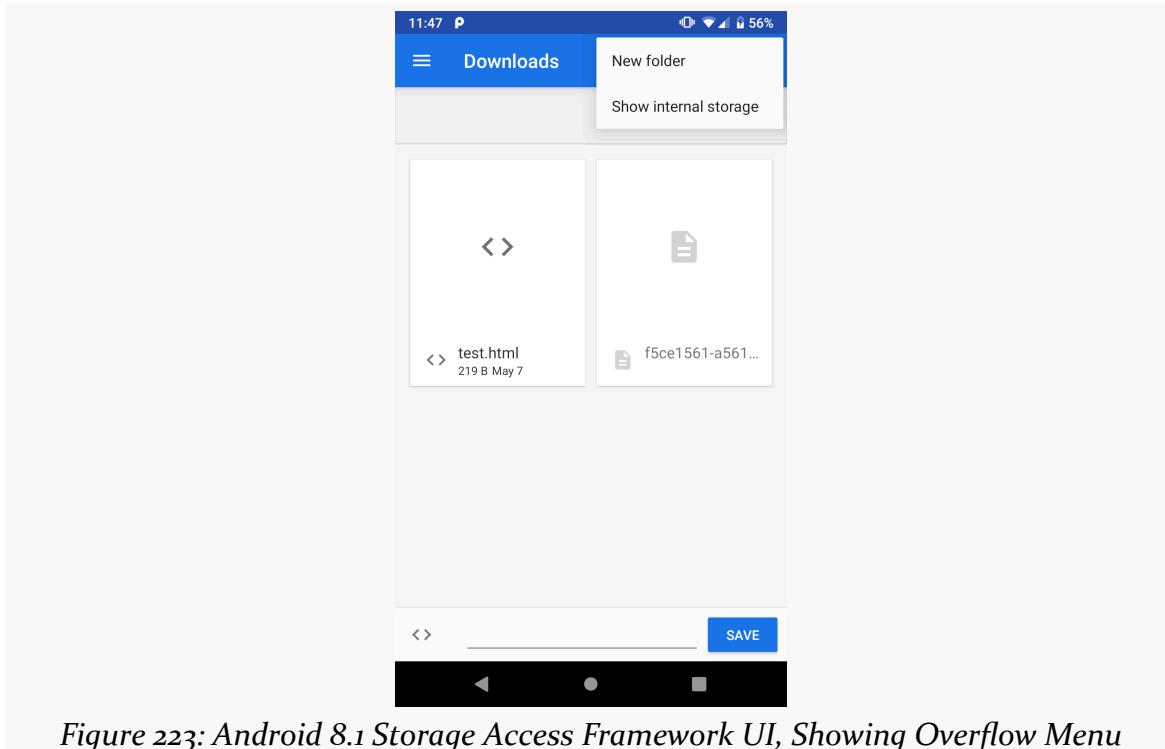


Figure 223: Android 8.1 Storage Access Framework UI, Showing Overflow Menu

Later, we will add an `onActivityResult()` method to `RosterListFragment`, to find out where the user chose to save the Web page. But, we need some code to create the Web page in the first place, which we will get to later in this tutorial.

## GENERATING A REPORT

If your device happened to show the “Save” item in the overflow, you might have noticed that “About” appeared before “Save”:

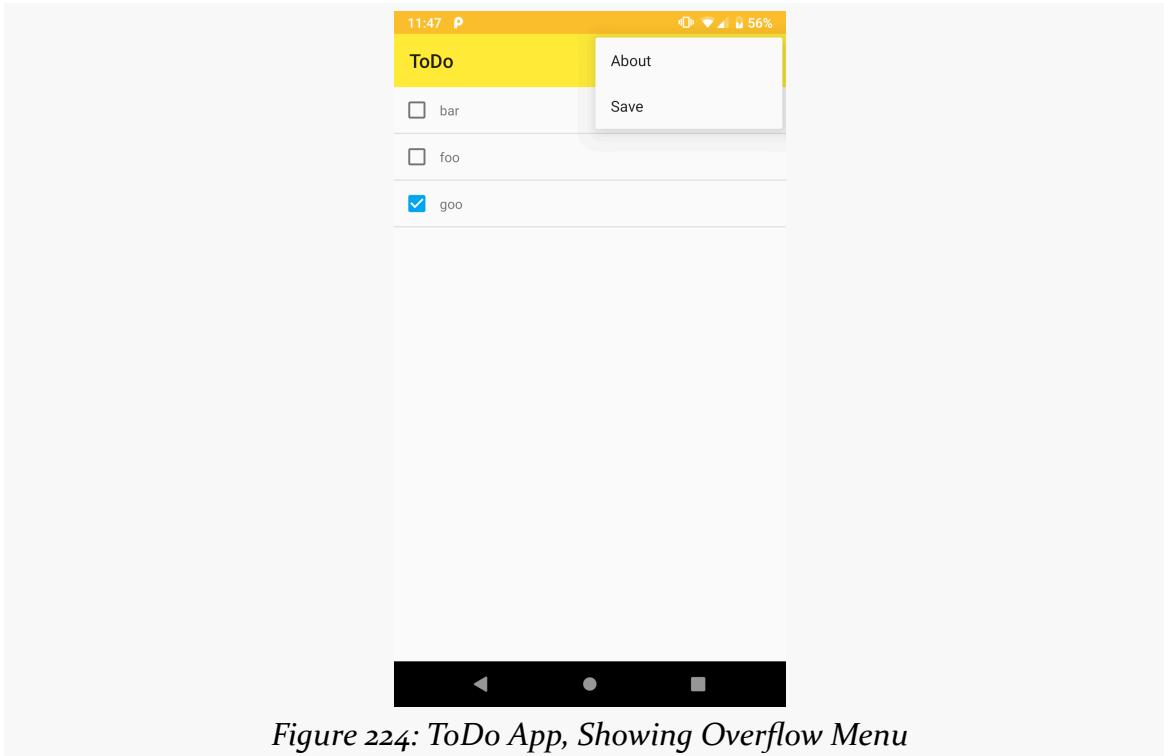


Figure 224: ToDo App, Showing Overflow Menu

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”:

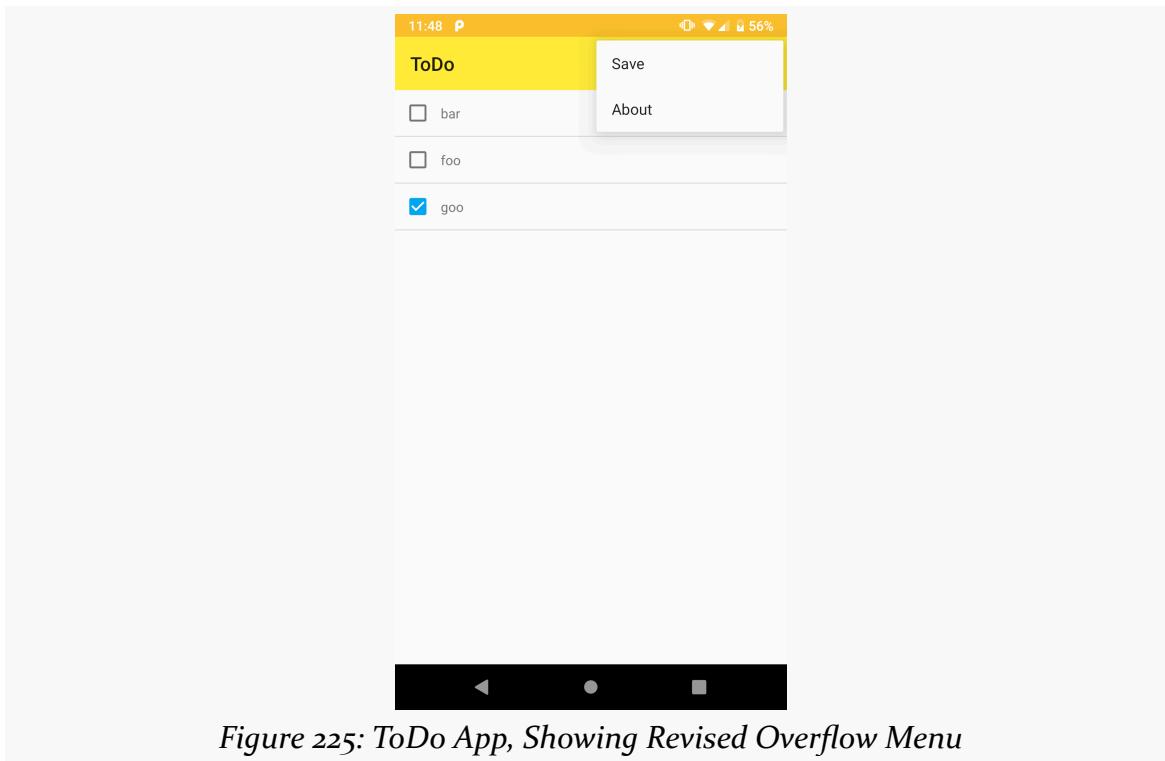


Figure 225: ToDo App, Showing Revised Overflow Menu

## 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 [T33-Report/ToDo/app/build.gradle](#))

Your overall dependencies closure should now look like:

```
dependencies {  
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'
```

## GENERATING A REPORT

```
implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
implementation 'androidx.recyclerview:recyclerview:1.0.0'
implementation 'androidx.fragment:fragment-ktx:1.0.0'
implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"
implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
implementation "org.koin:koin-core:$koin_version"
implementation "org.koin:koin-android:$koin_version"
implementation "org.koin:koin-androidx-viewmodel:$koin_version"
implementation "androidx.room:room-runtime:$room_version"
implementation "androidx.room:room-coroutines:$room_version"
implementation "com.github.jknack:handlebars:4.1.2"
kapt "androidx.room:room-compiler:$room_version"
testImplementation 'junit:junit:4.12'
testImplementation "androidx.arch.core:core-testing:2.0.0"
testImplementation "org.amshove.kluent:kluent-android:1.49"
testImplementation "org.mockito:mockito-inline:2.21.0"
testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.1.0"
testImplementation 'com.jraska.livedata:testing-ktx:1.1.0'
androidTestImplementation 'androidx.test.espresso:espresso-contrib:3.1.1'
androidTestImplementation "androidx.arch.core:core-testing:2.0.0"
androidTestImplementation 'androidx.test.ext:junit:1.1.0'
}
```

(from [T33-Report/ToDo/app/build.gradle](#))

Then, in ToDoApp, add this single to our koinModule:

```
single {
    Handlebars().apply {
        registerHelper("dateFormat", Helper<Calendar> { value, _ ->
            DateUtils
                .getRelativeDateTimeString(
                    androidContext(), value.timeInMillis,
                    DateUtils.MINUTE_IN_MILLIS, DateUtils.WEEK_IN_MILLIS, 0
                )
        })
    }
}
```

(from [T33-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

## GENERATING A REPORT

a dateFormat helper that takes a java.util.Calendar 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.commonware.todo package in the java/ directory, choose "New" > "Package" from the context menu, fill in report for the package name, and click "OK". This will create a com.commonware.todo.report package.

Then, right-click over the new com.commonware.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". Click "OK" to create the class, giving you:

```
package com.commonware.todo.report

class RosterReport {
```

Next, open up res/values/strings.xml and add this odd-looking string resource:

```
<string name="report_template"><![CDATA[<h1>To-Do Items</h1>
{{#this}}
<h2>{{description}}</h2>
<p>{{#isCompleted}}<b>COMPLETED</b> &mdash; {{/isCompleted}}Created on: {{dateFormat createdOn}}</p>
<p>{{notes}}</p>
{{/this}}
]]></string>
```

(from [T33-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

## GENERATING A REPORT

---

- {{description}} and {{notes}} pull values out of our models
- {{#isCompleted}} and {{/isCompleted}} represent the beginning and ending markers of a conditional section, which will only be included if isCompleted is true
- {{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 and our Handlebars object:

```
class RosterReport(private val context: Context, engine: Handlebars) {  
    (from T33-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))  
    (from T33-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 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()) }  
    (from T33-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 ACTION\_CREATE\_DOCUMENT request to our RosterReport and some implementation of ReportWriter.

What we get from ACTION\_CREATE\_DOCUMENT 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

## GENERATING A REPORT

file to an entry in Google Drive. A ContentResolver has an `openOutputStream()` method that will work with any Uri returned by `ACTION_CREATE_DOCUMENT`, 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) {
        OutputStreamWriter(context.contentResolver.openOutputStream(doc)).use { osw ->
            osw.write(template.apply(content))
            osw.flush()
        }
    }
}
```

(from [T33-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 (`withContext(Dispatchers.IO)`), 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
- Call `apply()` on our template to have it generate the HTML for our models
- Write that to the `OutputStreamWriter`

At this point, `RosterReport` 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.Dispatchers
import kotlinx.coroutines.withContext
import java.io.OutputStreamWriter

class RosterReport(private val context: Context, engine: Handlebars) {
    private val template =
        engine.compileInline(context.getString(R.string.report_template))

    suspend fun generate(content: List<ToDoModel>, doc: Uri) {
        withContext(Dispatchers.IO) {
            OutputStreamWriter(context.contentResolver.openOutputStream(doc)).use { osw ->
                osw.write(template.apply(content))
                osw.flush()
            }
        }
    }
}
```

## GENERATING A REPORT

---

```
    }
}
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/report/RosterReport.kt](#))

## 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 [T33-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 [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Overall, ToDoApp should now 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
import com.commonsware.todo.ui.roster.RosterMotor
import com.github.jknack.handlebars.Handlebars
import com.github.jknack.handlebars.Helper
import org.koin.android.ext.android.startKoin
import org.koin.android.ext.koin.androidContext
import org.koin.androidx.viewmodel.ext.koin.viewModel
import org.koin.dsl.module.module
import java.util.*

class ToDoApp : Application() {
    private val koinModule = module {
        single { ToDoDatabase.newInstance(androidContext()) }
```

## GENERATING A REPORT

```
single {
    val db: ToDoDatabase = get()

    ToDoRepository(db.todoStore())
}

single {
    Handlebars().apply {
        registerHelper("dateFormat", Helper<Calendar> { value, _ ->
            DateUtils
                .getRelativeDateTimeString(
                    androidContext(), value.timeInMillis,
                    DateUtils.MINUTE_IN_MILLIS, DateUtils.WEEK_IN_MILLIS, 0
                )
        })
    }
}

single { RosterReport(androidContext(), get()) }
viewModel { RosterMotor(get(), get()) }
viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }

}

override fun onCreate() {
    super.onCreate()

    startKoin(this, listOf(koinModule))
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Then, add this function to RosterMotor:

```
fun saveReport(doc: Uri) {
    viewModelScope.launch(Dispatchers.Main) {
        _states.value?.let { report.generate(it.items, doc) }
    }
}
```

Here, if we have a list of items in our RosterViewState, we launch() our generate() coroutine, supplying that list of items and the Uri identifying where we want the report to be written.

Then, in RosterListFragment, add this function:

```
override fun onActivityResult(
    requestCode: Int,
    resultCode: Int,
```

## GENERATING A REPORT

```
    data: Intent?  
    ) {  
        if (requestCode == REQUEST_SAVE) {  
            if (resultCode == Activity.RESULT_OK) {  
                data?.data?.let { motor.saveReport(it) }  
            }  
        }  
    }  
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This is the counterpart to our `startActivityForResult()` function call. `onActivityResult()` is where we get the result of our `ACTION_CREATE_DOCUMENT` call. The “result” of `startActivityForResult()` is another Intent object, created by the activity that we started. The way `ACTION_CREATE_DOCUMENT` works is that the user’s chosen Uri will be in the “data” facet of that Intent. So, if this was for our original request (`REQUEST_SAVE`) and if the user did not cancel out of the “save as” UI (`Activity.RESULT_OK`), we get that Uri and call `saveReport()` on our motor. The net result is that if the user chooses “Save”, we generate the report to the location that the user chooses.

## 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. Since we are using copy constructors to create new viewstates from older ones, our Uri would remain part of the viewstate until we expressly get rid of it. That is doable, but somewhat messy.

Another pattern is to consider these sorts of one-time events to be separate things that can be emitted by our motor via a separate `LiveData`. And, as part of that, we

## GENERATING A REPORT

---

can arrange to ensure that the events are only consumed once.

To that end, right-click over the `com.commonware.todo.util` package, and choose “New” > “Kotlin File/Class” from the context menu. Fill in “Event” as the filename, but this time choose “File” for the “Kind”. Then, click “OK” to create a nearly-empty `Event.kt` file.

Next, replace that nearly-empty file with the following:

```
package com.commonware.todo.ui.util

import androidx.lifecycle.Observer

class Event<out T>(private val content: T) {
    private var hasBeenHandled = false

    fun handle(handler: (T) -> Unit) {
        if (!hasBeenHandled) {
            hasBeenHandled = true
            handler(content)
        }
    }
}

class EventObserver<T>(private val handler: (T) -> Unit) : Observer<Event<T>> {
    override fun onChanged(value: Event<T>?) {
        value?.handle(handler)
    }
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonware/todo/ui/util/Event.kt](#))

`Event` is a wrapper around some piece of data (`content`) of some arbitrary type (`T`). The `handle()` function accepts a function type (e.g., a lambda expression) that gives you the content... but only if you have not called `handle()` previously on this instance of `Event`. This ensures that you only handle this event once.

`EventObserver` is an `Observer` that delegates to `handle()` on the `Event`, to make it easy to consume a `LiveData` of an `Event` type. There are a few variations on this “single live event” pattern available; this is merely one approach to the problem.

Next, in `RosterMotor.kt`, add the following as a peer of `RosterViewState` and `RosterMotor`:

```
sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
```

## GENERATING A REPORT

```
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))



You can learn more about Kotlin and sealed class in the "Enums and Sealed Classes" chapter of [Elements of Kotlin!](#)!

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, we have a Nav sealed class for representing all of our navigation requests, and 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 = MutableLiveData<Event<Nav>>()
val navEvents: LiveData<Event<Nav>> = _navEvents
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

This sets up another LiveData, this time for a Nav request, wrapped in an Event. As before, we have a MutableLiveData for internal use, and we expose it to the RosterListFragment as a plain LiveData, to hide the implementation details.

Next, modify saveReport() in RosterMotor to be:

```
fun saveReport(doc: Uri) {
    viewModelScope.launch(Dispatchers.Main) {
        _states.value?.let { report.generate(it.items, doc) }
        _navEvents.postValue(Event(Nav.ViewReport(doc)))
    }
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Here, once the report has been saved, we post a ViewReport request to our MutableLiveData.

At this point, RosterMotor.kt should resemble:

## GENERATING A REPORT

---

```
package com.commonsware.todo.ui.roster

import android.net.Uri
import androidx.lifecycle.*
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 com.commonsware.todo.ui.util.Event
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    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 = MediatorLiveData<RosterViewState>()
    val states: LiveData<RosterViewState> = _states
    private var lastSource: LiveData<List<ToDoModel>>? = null
    private val _navEvents = MutableLiveData<Event<Nav>>()
    val navEvents: LiveData<Event<Nav>> = _navEvents

    init {
        load(FilterMode.ALL)
    }

    fun load(filterMode: FilterMode) {
        lastSource?.let { _states.removeSource(it) }

        val items = repo.items(filterMode)

        _states.addSource(items) { models ->
            _states.value = RosterViewState(models, filterMode)
        }

        lastSource = items
    }

    fun save(model: ToDoModel) {
        viewModelScope.launch(Dispatchers.Main) {
            repo.save(model)
        }
    }

    fun saveReport(doc: Uri) {
        viewModelScope.launch(Dispatchers.Main) {
            _states.value?.let { report.generate(it.items, doc) }
            _navEvents.postValue(Event(Nav.ViewReport(doc)))
        }
    }
}
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

## GENERATING A REPORT

Then, in RosterListFragment, add this viewReport() function:

```
private fun viewReport(uri: Uri) {
    val i = Intent(Intent.ACTION_VIEW, uri)
        .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)

    try {
        startActivity(i)
    } catch (e: ActivityNotFoundException) {
        Toast.makeText(activity, R.string.msg_saved, Toast.LENGTH_LONG).show()
    }
}
```

(from [T33-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 startActivity() to bring up the Web browser (or whatever). If there is no suitable activity, we catch the ActivityNotFoundException and just show a Toast, where a Toast is a little pop-up bubble containing our requested text. That text is defined as a string resource, so you will need to add another entry to res/values/strings.xml for it:

```
<string name="msg_saved">Your Web page is saved!</string>
```

(from [T33-Report/ToDo/app/src/main/res/values/strings.xml](#))

Finally, in RosterListFragment, towards the bottom of onViewCreated(), add the following:

```
motor.navEvents.observe(this, EventObserver { nav ->
    when (nav) {
        is Nav.ViewReport -> viewReport(nav.doc)
    }
})
```

(from [T33-Report/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

This gets our new LiveData from the motor and sets up an EventObserver for it. When the motor posts a ViewReport request, we call viewReport()... but only the first time for a given request. If the user rotates the screen or otherwise triggers a configuration change, while our LiveData will still have our Event, it will be marked as having been handled, and our EventObserver lambda expression will

## GENERATING A REPORT

---

not be called again.

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.

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

# Sharing the Report

---

We have the HTML form of our to-do list, and, on many 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 action 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 Action Bar Item

Right now, it may seem like Android app development is just a series of action 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 action bar items.

That being said... we need to create another action 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:

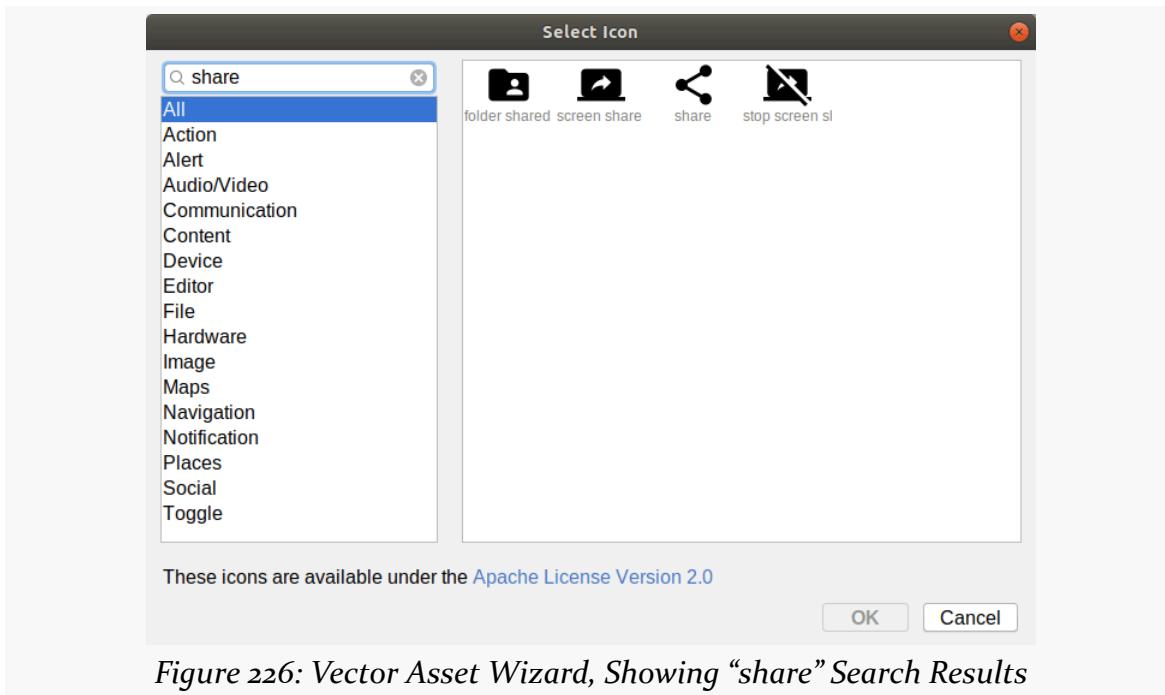


Figure 226: Vector Asset Wizard, Showing “share” Search Results

Choose the “share” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

## SHARING THE REPORT

Open up the `res/menu/actions_roster.xml` resource file, and switch to the “Design” sub-tab. Drag an “Item” from the “Palette” view into the Component Tree, slotting it after the existing “save” item:

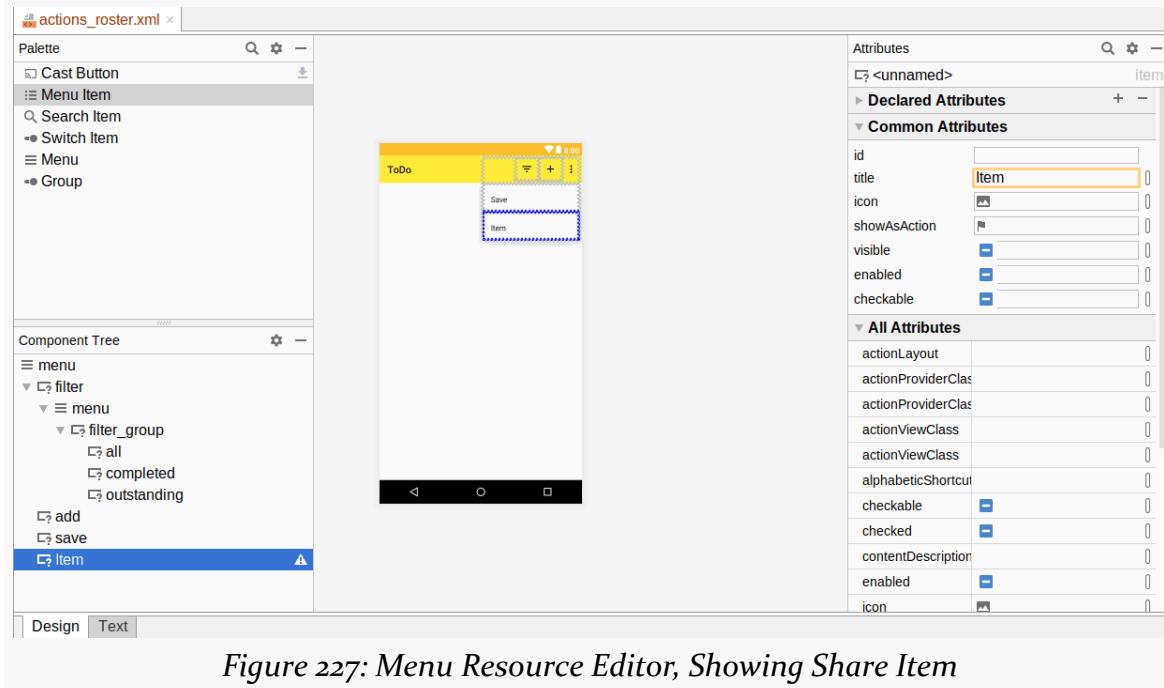


Figure 227: Menu Resource Editor, Showing Share Item

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_save_black_24dp` 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 action bar item.

## Step #2: Adding FileProvider

For the purposes of a “share” action 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 an IDE. The provider node under the activity node is selected, highlighting the android:authorities attribute. The code is as follows:

```
        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" />
        <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
```

Below the code, there are tabs for "Text" and "Merged Manifest".

Figure 228: 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

## SHARING THE REPORT

create an XML file with instructions, a bit reminiscent of how you configure a Web server to say what directories it should serve.

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:

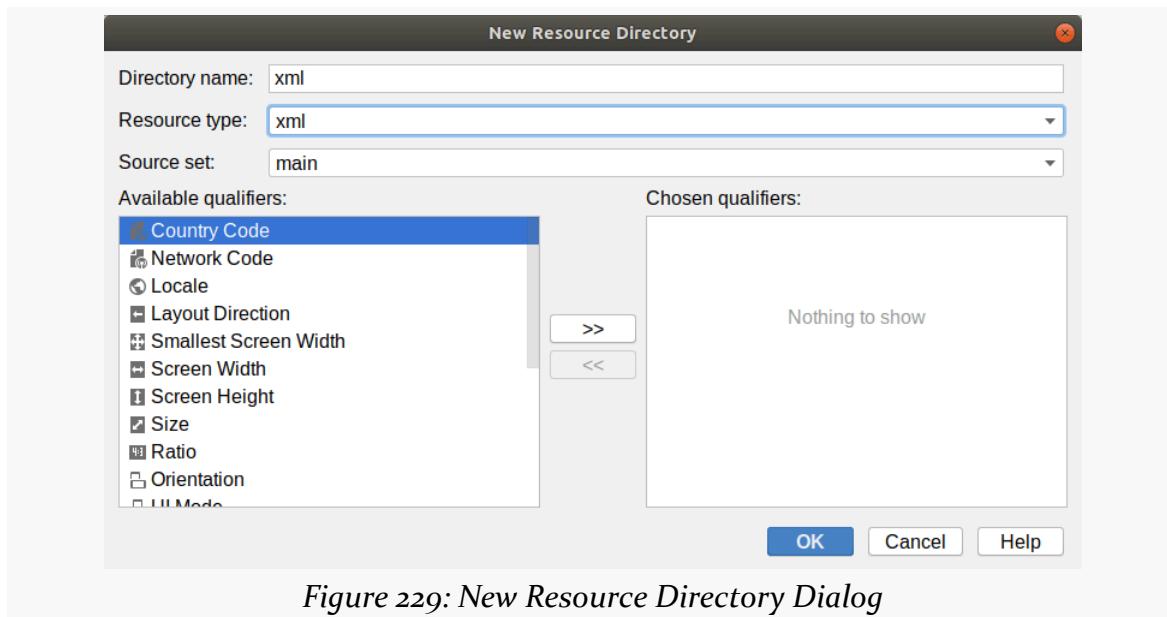


Figure 229: New Resource Directory Dialog

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:

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

## SHARING THE REPORT

```
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <cache-path name="shared" path="shared" />
</paths>
```

(from [T34-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 [T34-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.

The overall `AndroidManifest.xml` file should now look like:

## SHARING THE REPORT

```
<?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: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/AppTheme">
    <activity android:name=".ui.AboutActivity"></activity>
    <activity android:name=".ui.MainActivity">
      <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 [T34-Share/ToDo/app/src/main/AndroidManifest.xml](#))

## 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

## SHARING THE REPORT

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(Dispatchers.Main) {
        saveForSharing()
    }
}

private suspend fun saveForSharing() {
    withContext(Dispatchers.IO) {
        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.postValue(Event(Nav.ShareReport(doc)))
    }
}
```

(from [T34-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 couple of compile errors. One is that `AUTHORITY` is undefined. This needs to match the value that we have in the `android:authorities` attribute in the `<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 [T34-Share/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

## SHARING THE REPORT

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.

The other 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 [T34-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.

## 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.

As before, we will need to let the user know about that. So, open res/values/strings.xml and add a new string resource, named msg\_share\_fail, with a string like “Nothing knows how to share a Web page!”:

```
<string name="msg_share_fail">Nothing knows how to share a Web page!</string>
```

(from [T34-Share/ToDo/app/src/main/res/values/strings.xml](#))

## SHARING THE REPORT

Next, in RosterListFragment, add this shareReport() function:

```
private fun shareReport(doc: Uri) {
    val i = Intent(Intent.ACTION_SEND)
        .setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION)
        .setType("text/html")
        .putExtra(Intent.EXTRA_STREAM, doc)

    try {
        startActivity(i)
    } catch (e: ActivityNotFoundException) {
        Toast.makeText(activity, R.string.msg_share_fail, Toast.LENGTH_LONG)
            .show()
    }
}
```

(from [T34-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 extra rather than in the main portion of the Intent the way our ACTION\_VIEW Intent was set up.

Finally, in onViewCreated() of RosterListFragment, modify the navEvents EventObserver configuration to look like:

```
motor.navEvents.observe(this, EventObserver { nav ->
    when (nav) {
        is Nav.ViewReport -> viewReport(nav.doc)
        is Nav.ShareReport -> shareReport(nav.doc)
    }
})
```

(from [T34-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. And, if not, you should get the Toast showing that there were no options available for you.

## What We Changed

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

## SHARING THE REPORT

---

- [app/src/main/res/drawable/ic\\_share\\_black\\_24dp.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.

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 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:1.0.0"
```

## COLLECTING A PREFERENCE

(from [T35-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 "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"  
    implementation 'androidx.appcompat:appcompat:1.0.2'  
    implementation 'androidx.core:core-ktx:1.0.2'  
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'  
    implementation 'androidx.recyclerview:recyclerview:1.0.0'  
    implementation 'androidx.fragment:fragment-ktx:1.0.0'  
    implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"  
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"  
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"  
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"  
    implementation "androidx.preference:preference-ktx:1.0.0"  
    implementation "org.koin:koin-core:$koin_version"  
    implementation "org.koin:koin-android:$koin_version"  
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"  
    implementation "androidx.room:room-runtime:$room_version"  
    implementation "androidx.room:room-coroutines:$room_version"  
    implementation "com.github.jknack:handlebars:4.1.2"  
    kapt "androidx.room:room-compiler:$room_version"  
    testImplementation 'junit:junit:4.12'  
    testImplementation "androidx.arch.core:core-testing:2.0.0"  
    testImplementation "org.amshove.kluent:kluent-android:1.49"  
    testImplementation "org.mockito:mockito-inline:2.21.0"  
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.1.0"  
    testImplementation 'com.jraska.livedata:testing-ktx:1.1.0'  
    androidTestImplementation 'androidx.test.espresso:espresso-contrib:3.1.1'  
    androidTestImplementation "androidx.arch.core:core-testing:2.0.0"  
    androidTestImplementation 'androidx.test.ext:junit:1.1.0'  
}
```

(from [T35-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

## COLLECTING A PREFERENCE

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 `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:

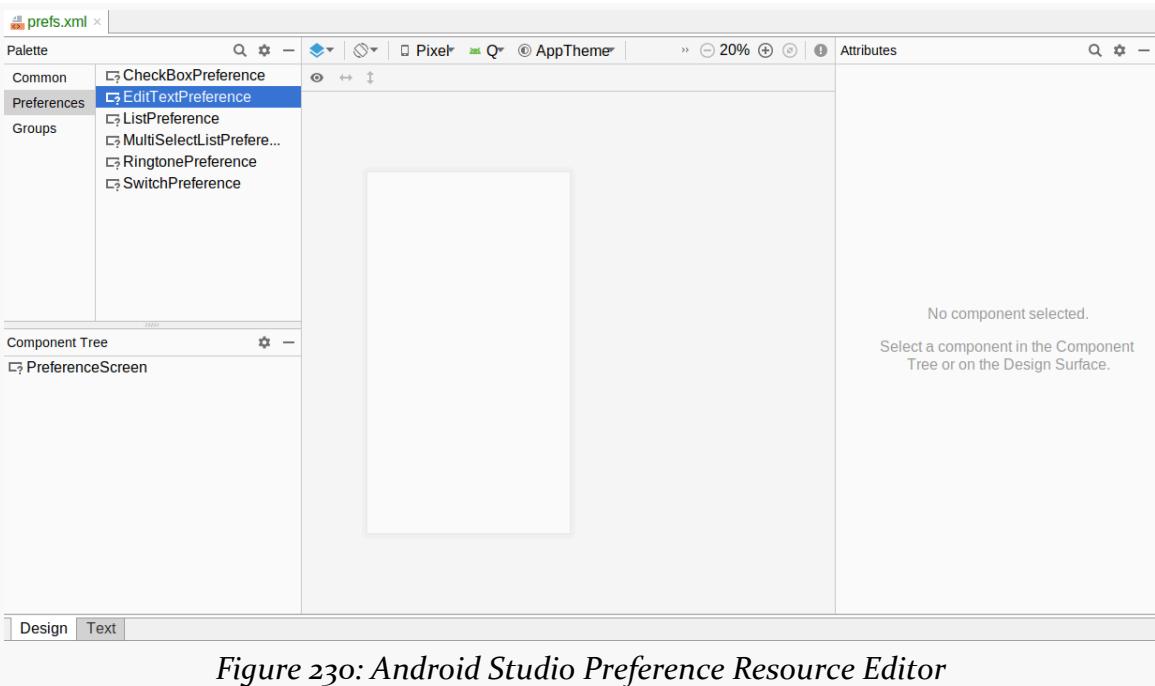


Figure 230: Android Studio Preference Resource Editor

Unfortunately, the Android Studio preference resource editor does not work all that well. So, switch over to the “Text” sub-tab of the 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 [T35-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="settings">Settings</string>
<string name="web_service_url_key">webServiceUrl</string>
<string name="web_service_url_default">https://commonsware.com/AndExplore/items.json</string>
```

(from [T35-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.commonsware.todo.ui package in the java/ directory, choose “New” > “Package” from the context menu, fill in prefs for the package name, and click “OK”. This will create a com.commonsware.todo.ui.prefs package.

Then, right-click over the new com.commonsware.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”. Click “OK” to create the class, giving you:

```
package com.commonsware.todo.ui.prefs

class PrefsFragment {
```

Finally, replace that stub class with:

```
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 [T35-Prefs/ToDo/app/src/main/java/com/commonsware/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 display PrefsFragment when requested.

## COLLECTING A PREFERENCE

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:

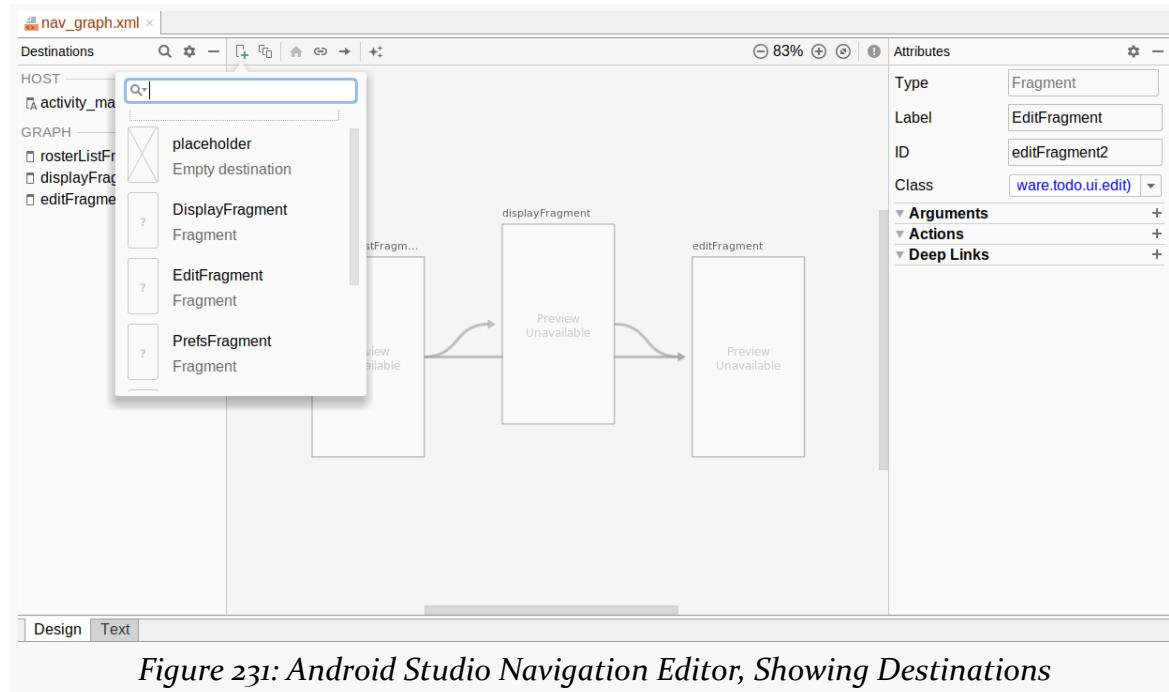


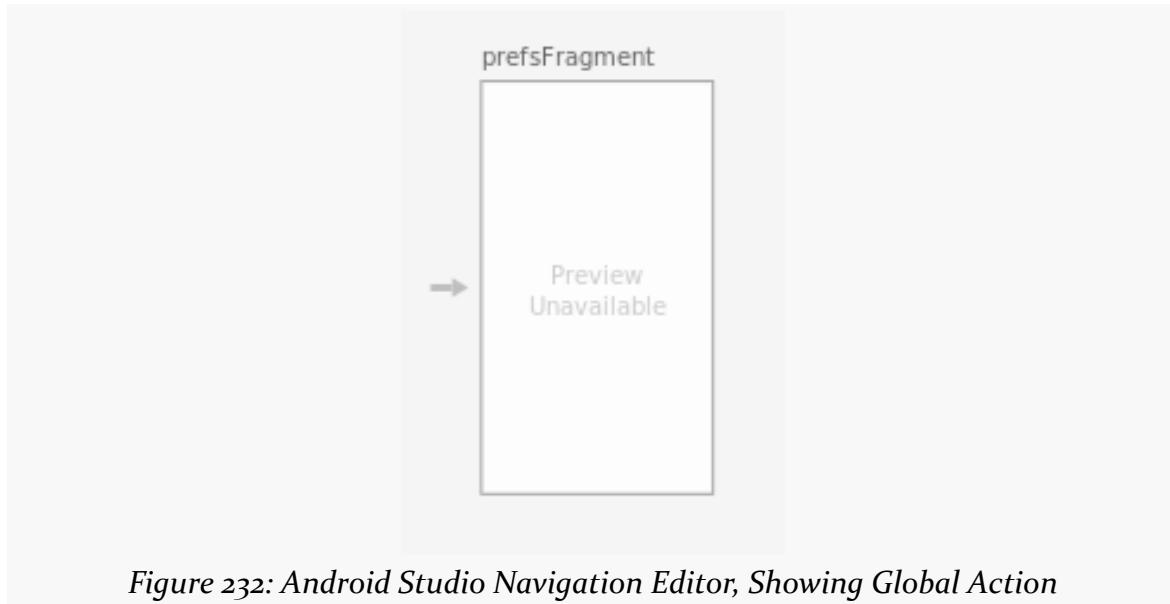
Figure 231: Android Studio Navigation Editor, Showing Destinations

Click on `PrefsFragment`, then drag its tile to some clean spot in the diagram.

## COLLECTING A PREFERENCE

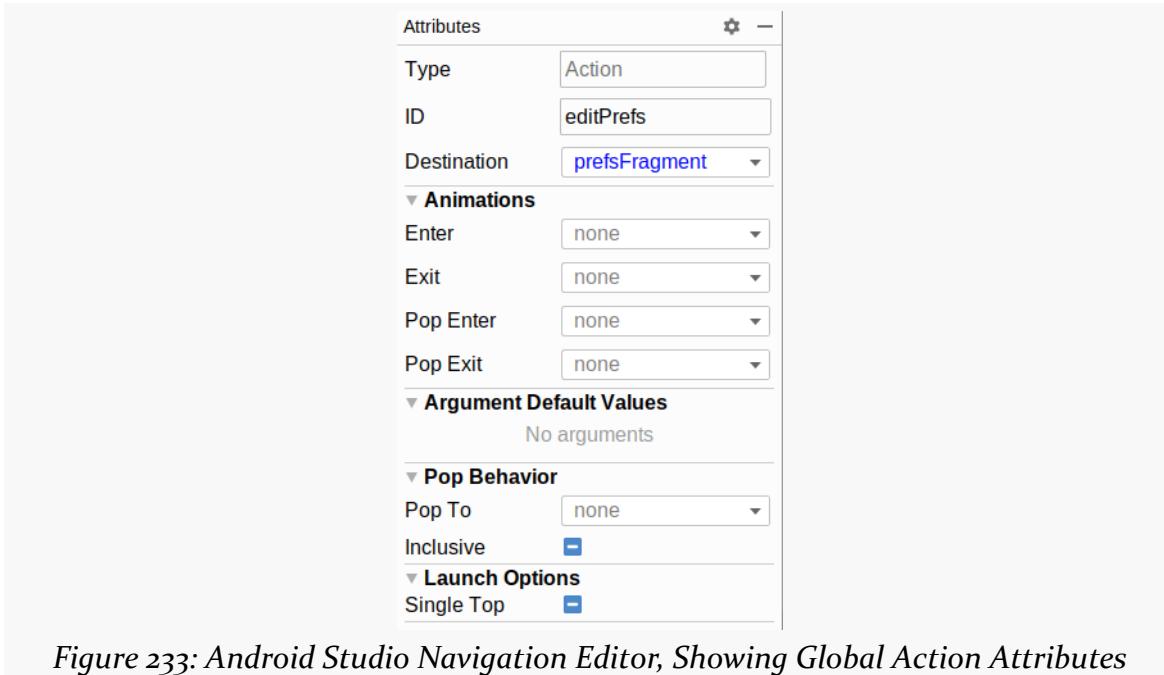
---

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:



## 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:



## Step #5: Navigating to Our Preference Screen

Now, we need to arrange to display that PrefsFragment. The first step is yet another action bar item, because we *love* action 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`:

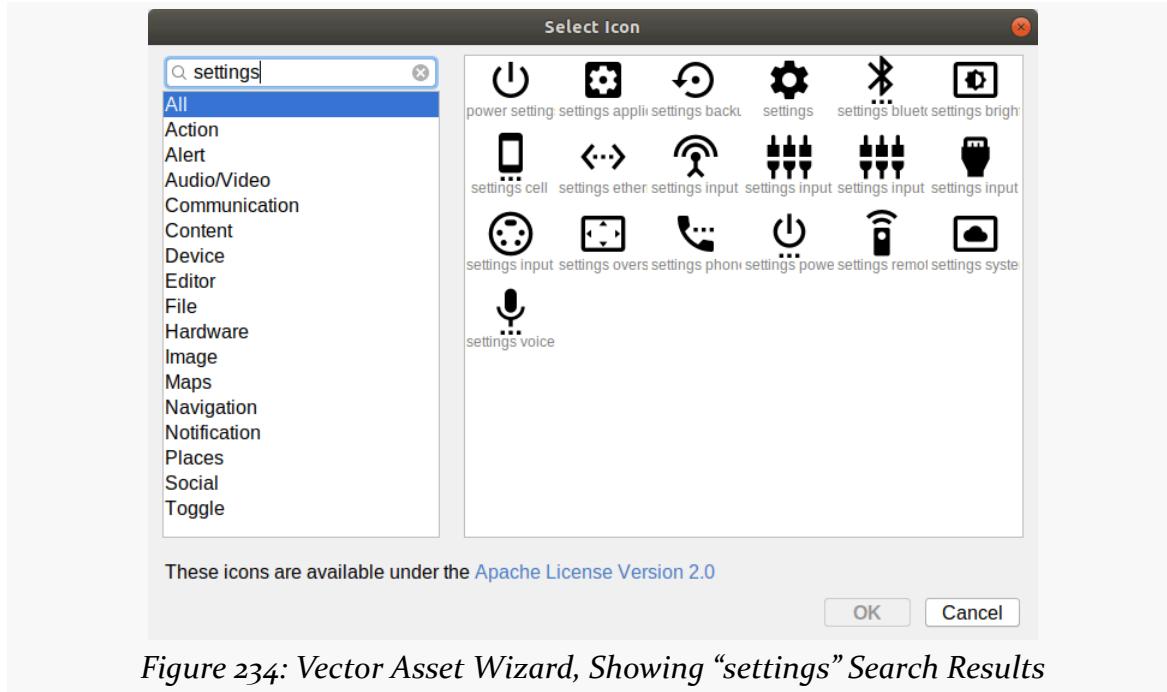


Figure 234: Vector Asset Wizard, Showing “settings” Search Results

Choose the “settings” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

Open up the `res/menu/actions.xml` resource file, and switch to the “Design” sub-tab. Drag an “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 a drawable resource selector. Click on `ic_settings_black_24dp` 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 `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.

## COLLECTING A PREFERENCE

---

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 [T35-Prefs/ToDo/app/src/main/java/com/commonsware/todo/ui/MainActivity.kt](#))

This switches us to a `when` instead of an `if` and 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 action bar item:

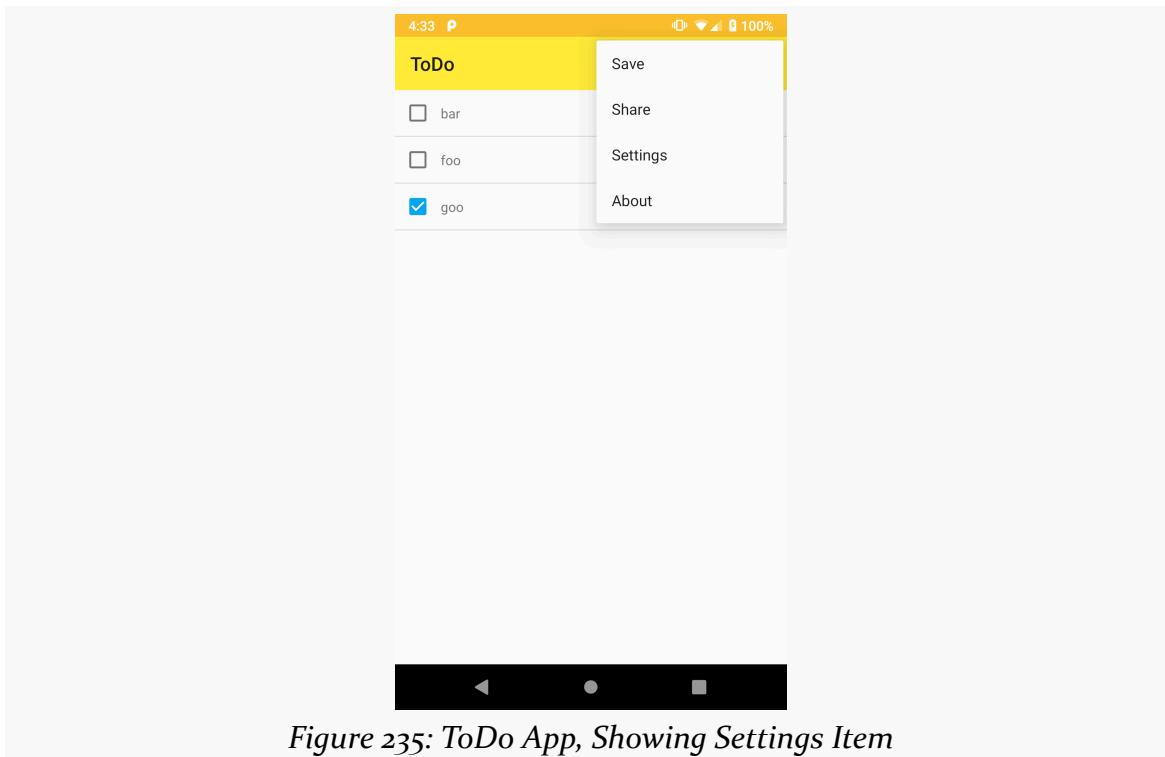
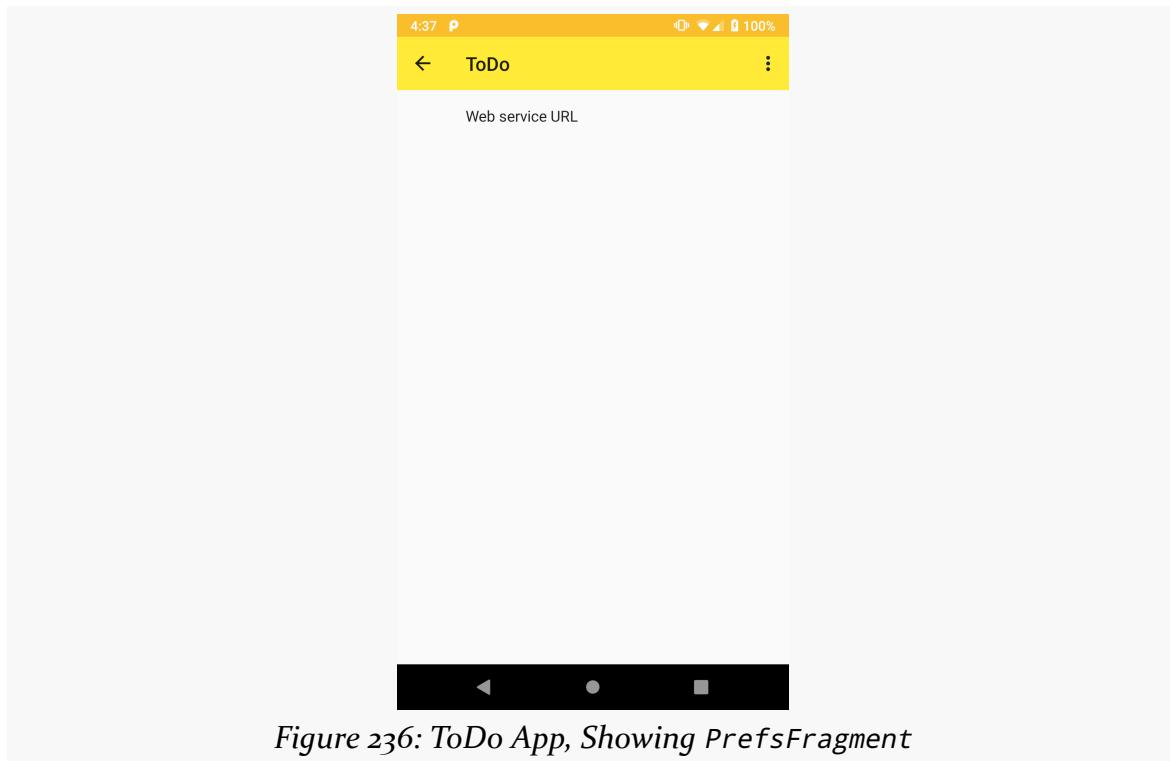


Figure 235: ToDo App, Showing Settings Item

## COLLECTING A PREFERENCE

---

Clicking that will bring up the fairly boring `PrefsFragment`:



*Figure 236: 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:

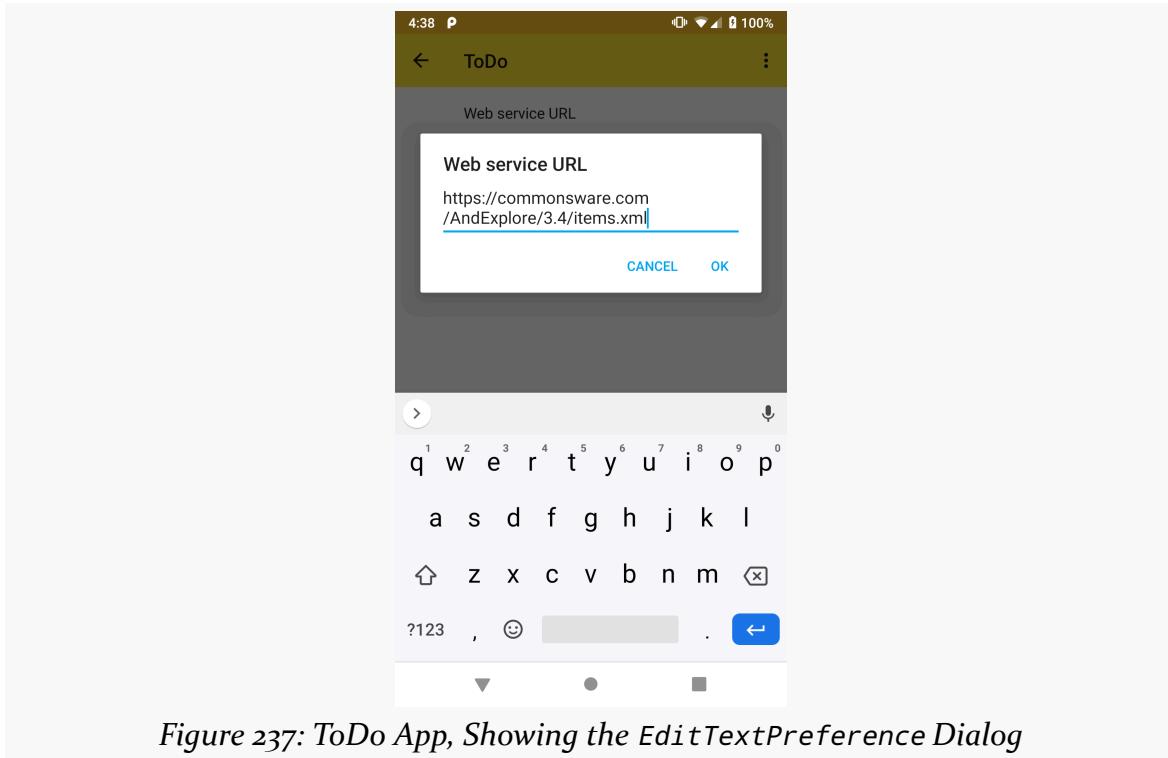


Figure 237: ToDo App, Showing the `EditTextPreference` Dialog

Right now, leave the value alone — just click BACK a few times to exit back to the main screen.

## 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\\_black\\_24dp.xml](#)
- [app/src/main/res/menu/actions.xml](#)
- [app/src/main/java/com/commonsware/todo/ui/MainActivity.kt](#)

Licensed solely for use by Patrocinio Rodriguez

# 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 `HttpURLConnection`, 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 team that

## CONTACTING A WEB SERVICE

created OkHttp

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:3.14.2"
implementation "com.squareup.moshi:moshi:$moshi_version"
kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"
```

(from [T36-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 = "1.8.0"
```

(from [T36-Internet/ToDo/build.gradle](#))

At this point, the top-level `build.gradle` file should look a bit like:

```
// Top-level build file where you can add configuration options common to all sub-projects/modules.

buildscript {
    ext.kotlin_version = '1.3.31'
    ext.nav_version = '2.0.0'

    repositories {
        google()
        jcenter()

    }
    dependencies {
        classpath 'com.android.tools.build:gradle:3.4.1'
        classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        classpath "androidx.navigation:navigation-safe-args-gradle-plugin:$nav_version"
        // NOTE: Do not place your application dependencies here; they belong
        // in the individual module build.gradle files
    }
}

allprojects {
    repositories {
```

## CONTACTING A WEB SERVICE

```
        google()
        jcenter()
    }

task clean(type: Delete) {
    delete rootProject.buildDir
}

ext {
    koin_version = "1.0.2"
    moshi_version = "1.8.0"
    room_version = "2.1.0-alpha04"
}
```

(from [T36-Internet/ToDo/build.gradle](#))

...and the dependencies closure of app/build.gradle should contain:

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation 'androidx.appcompat:appcompat:1.0.2'
    implementation 'androidx.core:core-ktx:1.0.2'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.recyclerview:recyclerview:1.0.0'
    implementation 'androidx.fragment:fragment-ktx:1.0.0'
    implementation "androidx.lifecycle:lifecycle-livedata:2.0.0"
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-beta01"
    implementation "androidx.navigation:navigation-fragment-ktx:$nav_version"
    implementation "androidx.navigation:navigation-ui-ktx:$nav_version"
    implementation "androidx.preference:preference-ktx:1.0.0"
    implementation "org.koin:koin-core:$koin_version"
    implementation "org.koin:koin-android:$koin_version"
    implementation "org.koin:koin-androidx-viewmodel:$koin_version"
    implementation "androidx.room:room-runtime:$room_version"
    implementation "androidx.room:room-coroutines:$room_version"
    implementation "com.github.jknack:handlebars:4.1.2"
    implementation "com.squareup.okhttp3:okhttp:3.14.2"
    implementation "com.squareup.moshi:moshi:$moshi_version"
    kapt "com.squareup.moshi:moshi-kotlin-codegen:$moshi_version"
    kapt "androidx.room:room-compiler:$room_version"
    testImplementation 'junit:junit:4.12'
    testImplementation "androidx.arch.core:core-testing:2.0.0"
    testImplementation "org.amshove.kluent:kluent-android:1.49"
    testImplementation "org.mockito:mockito-inline:2.21.0"
    testImplementation "com.nhaarman.mockitokotlin2:mockito-kotlin:2.1.0"
    testImplementation 'com.jraska.livedata:testing-ktx:1.1.0'
    androidTestImplementation 'androidx.test.espresso:espresso-contrib:3.1.1'
    androidTestImplementation "androidx.arch.core:core-testing:2.0.0"
    androidTestImplementation 'androidx.test.ext:junit:1.1.0'
}
```

## CONTACTING A WEB SERVICE

---

(from [T36-Internet/ToDo/app/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 [T36-Internet/ToDo/app/src/main/AndroidManifest.xml](#))

At this point, the manifest overall should resemble:

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

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

## CONTACTING A WEB SERVICE

```
<activity android:name=".ui.MainActivity">
    <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 [T36-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.

## CONTACTING A WEB SERVICE

---

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”. Click “OK” to create the class. Then, replace the class contents with:

```
@JsonClass(generateAdapter = true)
data class ToDoServerItem(
    val description: String,
    val id: String,
    val completed: Boolean,
    val notes: String,
    @Json(name = "created_on") val createdOn: Calendar
)
```

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 `Calendar` and knows nothing about how to take a value like “`2019-05-22`” and convert it into a `Calendar`. For that, we need to create an adapter class.

So, below `ToDoServerItem` in the same file, add this code:

```
@SuppressLint("SimpleDateFormat")
private val FORMATTER = SimpleDateFormat("yyyy-MM-dd")
```

## CONTACTING A WEB SERVICE

```
class MoshiCalendarAdapter {
    @ToJson
    fun toJson(date: Calendar) = FORMATTER.format(date.time)

    @FromJson
    fun fromJson(dateString: String): Calendar =
        Calendar.getInstance().apply { time = FORMATTER.parse(dateString) }
}
```

(from [T36-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 `SimpleDateFormat` to convert a `Calendar` 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.commonsware.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”. Click “OK” to create the class. Then, replace the class contents with:

```
package com.commonsware.todo.repo

import com.squareup.moshi.JsonAdapter
import com.squareup.moshi.Moshi
import com.squareup.moshi.Types
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import okhttp3.OkHttpClient
import okhttp3.Request
import java.io.IOException

class ToDoRemoteDataSource(private val ok: OkHttpClient) {
```

## CONTACTING A WEB SERVICE

```
private val moshi = Moshi.Builder().add(MoshiCalendarAdapter()).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 [T36-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRemoteDataSource.kt](#))

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).

## CONTACTING A WEB SERVICE

---

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 `MoshiCalendarAdapter()` to the `Moshi` instance.

By and large, Moshi is a series of adapter classes. Some we write ourselves, such as `MoshiCalendarAdapter()`. 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 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()) }
```

## CONTACTING A WEB SERVICE

(from [T36-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 remoteDataSource:  
ToDoRemoteDataSource) {
```

(from [T36-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:

```
single {  
    val db: ToDoDatabase = get()  
  
    ToDoRepository(db.todoStore(), get())  
}
```

(from [T36-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 [T36-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 [T36-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

## CONTACTING A WEB SERVICE

---

- 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 stuff instead of `import`.

To use `importItems()`, we need a way to map `ToDoServerItem` objects to `ToDoEntity` objects. So, add this `toEntity()` function to `ToDoServerItem`:

```
fun toEntity(): ToDoEntity {
    return ToDoEntity(
        id = id,
        description = description,
        isCompleted = completed,
        notes = notes,
        createdOn = createdOn
    )
}
```

(from [T36-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.

At this point, `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.text.SimpleDateFormat
import java.util.*

@JsonClass(generateAdapter = true)
data class ToDoServerItem(
    val description: String,
    val id: String,
    val completed: Boolean,
    val notes: String,
    @Json(name = "created_on") val createdOn: Calendar
)
```

## CONTACTING A WEB SERVICE

```
fun toEntity(): ToDoEntity {
    return ToDoEntity(
        id = id,
        description = description,
        isCompleted = completed,
        notes = notes,
        createdOn = createdOn
    )
}

@SuppressLint("SimpleDateFormat")
private val FORMATTER = SimpleDateFormat("yyyy-MM-dd")

class MoshiCalendarAdapter {
    @ToJson
    fun toJson(date: Calendar) = FORMATTER.format(date.time)

    @FromJson
    fun fromJson(dateString: String): Calendar =
        Calendar.getInstance().apply { time = FORMATTER.parse(dateString) }
}
```

(from [T36-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoServerItem.kt](#))

Now, we can glue all of this together. Add this `importItems()` function to `ToDoRepository`:

```
suspend fun importItems(url: String) {
    store.importItems(remoteDataSource.load(url).map { it.toEntity() })
}
```

(from [T36-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`.

## CONTACTING A WEB SERVICE

---

So, now our repository knows how to get items and import them into our database.

At this point, ToDoRepository should resemble:

```
package com.commonsware.todo.repo

import androidx.lifecycle.LiveData
import androidx.lifecycle.Transformations

enum class FilterMode { ALL, OUTSTANDING, COMPLETED }

class ToDoRepository(private val store: ToDoEntity.Store, private val remoteDataSource:
ToDoRemoteDataSource) {
    fun items(filterMode: FilterMode = FilterMode.ALL): LiveData<List<ToDoModel>> =
        Transformations.map(filteredEntities(filterMode)) { all -> all.map { it.toModel() } }

    fun find(id: String): LiveData<ToDoModel> =
        Transformations.map(store.find(id)) { it.toModel() }

    suspend fun save(model: ToDoModel) {
        store.save(ToDoEntity(model))
    }

    suspend fun delete(model: ToDoModel) {
        store.delete(ToDoEntity(model))
    }

    suspend fun importItems(url: String) {
        store.importItems(remoteDataSource.load(url).map { it.toEntity() })
    }

    private fun filteredEntities(filterMode: FilterMode) = when (filterMode) {
        FilterMode.ALL -> store.all()
        FilterMode.OUTSTANDING -> store.filtered(isCompleted = false)
        FilterMode.COMPLETED -> store.filtered(isCompleted = true)
    }
}
```

(from [T36-Internet/ToDo/app/src/main/java/com/commonsware/todo/repo/ToDoRepository.kt](#))

## 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(), ToDoRemoteDataSource(OkHttpClient()))
```

(from [T36-Internet/ToDo/app/src/androidTest/java/com/commonsware/todo/ui/roster/RosterListFragmentTest.kt](#))

## CONTACTING A WEB SERVICE

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. At this point, if you run `RosterListFragmentTest`, it should pass.

Then, in `ToDoRepositoryTest` (in the test/ source set), add a new property for a mock implementation of `ToDoRemoteDataSource`:

```
private val remoteDataSource = mock(ToDoRemoteDataSource::class)  
(from T36-Internet/ToDo/app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt)
```

This is similar to how we mocked the `ToDoRepository` in `SingleModelMotorTest`.

Then, modify `setUp()` in `ToDoRepositoryTest` to pass that mock to our `ToDoRepository` constructor:

```
@Before  
fun setUp() {  
    underTest = ToDoRepository(TestStore(), remoteDataSource)  
}
```

We also have an error in our `TestStore`, as it is not implementing our `importItems()` function that we added to `ToDoEntity.Store`. So, add this function to `TestStore`:

```
override fun filtered(isCompleted: Boolean) =  
    MutableLiveData<List<ToDoEntity>>()  
    .apply { value = current().filter { it.isCompleted == isCompleted } }  
(from T36-Internet/ToDo/app/src/test/java/com/commonsware/todo/repo/ToDoRepositoryTest.kt)
```

This will filter the entities list to just those where `none()` of the `current()` items has the same ID, then will add those to the `current()` items to form the new list of entities.

And, if you run `ToDoRepositoryTest` now, it should pass. That is because we are not testing the import process here either. That is mostly because the function that we need to mock — `load()` — is a suspend function, and the mocking libraries have limited support for mocking such functions.

## 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”. Click “OK” 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 [T36-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 [T36-Internet/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

This will make a `PrefsRepository` available to other components via dependency injection.

## 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 prefs: PrefsRepository,  
    private val report: RosterReport,
```

## CONTACTING A WEB SERVICE

---

```
private val context: Context  
) : ViewModel() {
```

(from [T36-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(), get(), get()) }
```

(from [T36-Internet/Todo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

At this point, ToDoApp should resemble:

```
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.commonsware.todo.ui.roster.RosterMotor  
import com.github.jknack.handlebars.Handlebars  
import com.github.jknack.handlebars.Helper  
import okhttp3.OkHttpClient  
import org.koin.android.ext.android.startKoin  
import org.koin.android.ext.koin.androidContext  
import org.koin.androidx.viewmodel.ext.koin.viewModel  
import org.koin.dsl.module.module  
import java.util.*  
  
class ToDoApp : Application() {  
    private val koinModule = module {  
        single { ToDoDatabase.newInstance(androidContext()) }  
        single {  
            val db: ToDoDatabase = get()  
  
            ToDoRepository(db.todoStore(), get())  
        }  
        single {  
            Handlebars().apply {  
                registerHelper("dateFormat", Helper<Calendar> { value, _ ->  
                    DateUtils  
                        .getRelativeDateTimeString(  
                            value.time, DateUtils.FORMAT_ABBREV_RELATIVE  
                        )  
                })  
            }  
        }  
    }  
}
```

## CONTACTING A WEB SERVICE

```
        androidContext(), value.timeInMillis,
        DateUtils.MINUTE_IN_MILLIS, DateUtils.WEEK_IN_MILLIS, 0
    )
}
}
single { RosterReport(androidContext(), get()) }
single { OkHttpClient.Builder().build() }
single { ToDoRemoteDataSource(get()) }
single { PrefsRepository(androidContext()) }
viewModel { RosterMotor(get(), get(), get(), get()) }
viewModel { (modelId: String) -> SingleModelMotor(get(), modelId) }
}

override fun onCreate() {
    super.onCreate()

    startKoin(this, listOf(koinModule))
}
}
```

(from [T36-Internet/ToDo/app/src/main/java/com/commonsware/todo/ToDoApp.kt](#))

Then, add this function to RosterMotor:

```
fun importItems() {
    viewModelScope.launch(Dispatchers.Main) {
        repo.importItems(prefs.loadWebServiceUrl())
    }
}
```

(from [T36-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.

At this point, RosterMotor, should look like:

```
package com.commonsware.todo.ui.roster

import android.content.Context
import android.net.Uri
import androidx.core.content.FileProvider
import androidx.lifecycle.*
```

## CONTACTING A WEB SERVICE

---

```
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.util.Event
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch
import kotlinx.coroutines.withContext
import java.io.File

class RosterViewState(
    val items: List<ToDoModel> = listOf(),
    val filterMode: FilterMode = FilterMode.ALL
)

sealed class Nav {
    data class ViewReport(val doc: Uri) : Nav()
    data class ShareReport(val doc: Uri) : Nav()
}

private const val AUTHORITY = BuildConfig.APPLICATION_ID + ".provider"

class RosterMotor(
    private val repo: ToDoRepository,
    private val prefs: PrefsRepository,
    private val report: RosterReport,
    private val context: Context
) : ViewModel() {
    private val _states = MediatorLiveData<RosterViewState>()
    val states: LiveData<RosterViewState> = _states
    private var lastSource: LiveData<List<ToDoModel>>? = null
    private val _navEvents = MutableLiveData<Event<Nav>>()
    val navEvents: LiveData<Event<Nav>> = _navEvents

    init {
        load(FilterMode.ALL)
    }

    fun load(filterMode: FilterMode) {
        lastSource?.let { _states.removeSource(it) }

        val items = repo.items(filterMode)

        _states.addSource(items) { models ->
            _states.value = RosterViewState(models, filterMode)
        }
    }
}
```

## CONTACTING A WEB SERVICE

```
    lastSource = items
}

fun save(model: ToDoModel) {
    viewModelScope.launch(Dispatchers.Main) {
        repo.save(model)
    }
}

fun saveReport(doc: Uri) {
    viewModelScope.launch(Dispatchers.Main) {
        _states.value?.let { report.generate(it.items, doc) }
        _navEvents.postValue(Event(Nav.ViewReport(doc)))
    }
}

fun shareReport() {
    viewModelScope.launch(Dispatchers.Main) {
        saveForSharing()
    }
}

fun importItems() {
    viewModelScope.launch(Dispatchers.Main) {
        repo.importItems(prefs.loadWebServiceUrl())
    }
}

private suspend fun saveForSharing() {
    withContext(Dispatchers.IO) {
        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.postValue(Event(Nav.ShareReport(doc)))
    }
}
}
```

(from [T36-Internet/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt](#))

Now, we need to arrange to call that `importItems()` function... triggered by another action bar item.

Right-click over `res/drawable/` in the project tree and choose “New” > “Vector

## CONTACTING A WEB SERVICE

Asset” from the context menu. This brings up the Vector Asset Wizard. There, click the “Icon” button and search for download. Choose the “file download” icon and click “OK” to close up the icon selector. Then, click “Next” and “Finish” to close up the wizard and set up our icon.

Open up the `res/menu/actions_roster.xml` resource file, and switch to the “Design” sub-tab. 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_file_download_black_24dp` 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 [T36-Internet/ToDo/app/src/main/java/com/commonsware/todo/ui/roster/RosterListFragment.kt](#))

At this point, the overall `onOptionsItemSelected()` function should contain:

```
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 -> {
    }
}
```

## CONTACTING A WEB SERVICE

---

```
        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)
}
```

(from [T36-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” action 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.

However, you might run into problems. For example, your device may be in airplane mode or is otherwise unable to access the CommonsWare server hosting this JSON. Or perhaps the CommonsWare server is down for maintenance, or the author of this book screwed up and deleted that JSON file. In those sorts of cases, `importItems()` will throw an exception... which we are not handling. In a future edition of this book, we will catch that exception, show an error dialog, and allow the user to retry the import operation.

## 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/test/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\\_file\\_download\\_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](#)

Licensed solely for use by Patrocinio Rodriguez