Extracted from:

Kotlin and Android Development featuring Jetpack

Build Better, Safer Android Apps

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Kotlin and Android Development

featuring Jetpack

Build Better, Safer Android Apps

Michael Fazio

edited by Michael Swaine

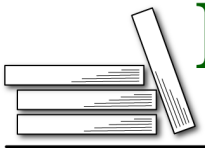
Kotlin and Android Development featuring Jetpack

Build Better, Safer Android Apps

Michael Fazio

The Pragmatic Bookshelf

Raleigh, North Carolina



For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-815-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—June 2021

Add UI Tests

While this chapter does indeed focus on the UI and user interaction with the app, things are thankfully going to look pretty similar to last chapter. This includes the test format (setup, run, assert) and the dependencies. The androidTestImplmentation dependencies look like this:

```
androidTestImplementation "junit:junit:$junit_version"
androidTestImplementation "androidx.test:core-ktx:$test_core_version"
androidTestImplementation "androidx.test.ext:junit-ktx:$test_ext_version"
androidTestImplementation
  "androidx.test.espresso:espresso-core:$espresso_version"
androidTestImplementation
  "org.jetbrains.kotlinx:kotlinx-coroutines-test:$coroutines_version"
```

The only new dependency here is for Espresso, Jetpack's UI testing library. Espresso simulates user interactions with an app, then runs assertions on those actions in a succinct, readable syntax. Under the hood, Espresso is using the Hamcrest library to help with matching elements—in fact, a number of the pieces we see with Espresso UI tests are Hamcrest Matcher objects.

Rather than spending time explaining more about Espresso and Hamcrest, let's get a simple test in place for the PickPlayersFragment, namely a test to see if the Play Game FAB exists.

Add PickPlayersFragmentTests

Create the PickPlayersFragmentTests class inside the androidTest folder (same place we had PennyDropDaoTests) and annotate the class with @RunWith(AndroidJUnit4::class). Then we're going to add a JUnit rule as we did before, but this time it's ActivityScenarioRule. This rule tells the tests to create and launch an Activity before each test, which in our case is MainActivity. The initial class setup looks like this:

```
@RunWith(AndroidJUnit4::class)
class PickPlayersFragmentTests {

  @get:Rule
  var activityScenarioRule = activityScenarioRule<MainActivity>()

  // We've got more coming up soon.
}
```

Adding that Rule effectively gives us an @Before function without needing to write any extra code. We do need to do a bit more before each test is run, though, since the app starts on the Game screen. The goToPickPlayersFragment() does what we need, taking us over to the PickPlayersFragment. We can use the

activityScenarioRule to get the current activity, then reference the activity to navigate:

```
@Before
fun goToPickPlayersFragment() {
  activityScenarioRule.scenario.onActivity { activity ->
      activity
        .findNavController(R.id.containerFragment)
        .navigate(R.id.pickPlayersFragment)
  }
}
```

Now we can add our test, which makes sure it can find the Play Game FAB (the round button with the Play icon) on the Pick Players screen. The way this works with Espresso is that we find the item by ID, then check() that some condition is true—in our case, checking the visibility of the button via the isDisplayed() function. The code looks like this:

```
@Test
fun testFindFab() {
  onView(withId(R.id.buttonPlayGame)).check(matches(isDisplayed()))
}
```

The onView() function finds a view based on the criteria sent into it, which in our case is the ID of the FAB. Once we have the view, we then check() that the view matches() the condition of being displayed. This is the general flow for Espresso; we get a View via one or more ViewMatcher objects, then optionally perform some kind of ViewAction, then finally confirm one or more ViewAssertion objects.

Espresso Cheat Sheet

The Android Developers team created an awesome cheat sheet for Espresso functions and common test pieces you'll be using. I *highly* recommend you have it up when you're writing Espresso tests. That's just what I did when I was writing the tests you'll see here in our chapter.

The cheat sheet can be found at https://link.mfazio.dev/espresso-cheat-sheet. They also have a PDF version of the sheet in case you want to save a copy for later or need to print it out to put on your wall.

Now that we've got the basics down and know the general structure, let's move on to something more complicated—adding a couple of players, then starting a game.

Test Adding Players to a Game

Next up is the testAddingNamedPlayers() function. With this test, we're going to have Espresso type some names into player fields, close out the keyboard, then click the Play Game button. Once we're on the Game screen, we're going to verify a few things to make sure our player names were entered correctly. Note that we're not checking *everything* on the Game screen, as we'll save that for the GameFragmentTests later in the chapter.

Typing in a player's name is a two-step process; we need to find the <EditText> element, then perform() a typeText() ViewAction with some kind of String value. We can find the <EditText> element by using the allOf() Matcher, which takes a variable number of inputs and says if they all are true.

In this case, we want the ID of the <include> tag and the R.id.edit_text_player_name resource value. More specifically, we use the withParent() matcher around the latter ID to find that element, since R.id.edit_text_player_name is a child view inside the <include> view. That leaves us with a matcher that looks like this:

```
onView(
  allOf(
    withId(R.id.edit_text_player_name),
    withParent(withId(parentId))
  )
).perform(typeText(text))
```

Typing a player's name in is something we'll do in multiple tests across multiple test classes, so it makes sense to have a TestHelpers.kt file in our androidTest folder with a few handy functions. This particular function is uncreatively named typeInPlayerName(parentId: Int, text: String), and we'll be using it a bunch. We'll see two more helper functions in the rest of the chapter.

Espresso Is for Developers

As you may be able to tell from the examples so far, Espresso is intended to be used by developers who are familiar with an app's code. Accessing elements by ID would be nigh impossible without being able to read the code, and while you *can* access view components via text, it's far less useful, and text values like that are prone to errors.

The test will type in two players' names, then close the keyboard and click the Play Game button. The closeSoftKeyboard() piece is critical here because otherwise the test runner won't be able to see the Play Game button to click it, and the test will fail. The first two-thirds of our test (setup + action) look like this:

```
@Test
fun testAddingNamedPlayers() {
  typeInPlayerName(R.id.mainPlayer, "Michael")
  typeInPlayerName(R.id.player2, "Emily")
  closeSoftKeyboard()

  onView(withId(R.id.buttonPlayGame)).perform(click())

  // Verifying things in a bit.
}
```

The typeInPlayerName() function, which lives in the TestHelpers.kt file (if you haven't created that file yet, do so now), can be referenced here since it's in the same package as the PickPlayersFragmentTests. Also, there's no need to create a class just to hold a few helpful functions, we can instead put them into a file and use them directly. Even if they live in a separate package, that still works; we just need to import that package in our file.

You *could* run this test right now, and I'm betting it would succeed since we're not yet asserting anything, though it could still fail if there's an issue with the test code itself.

We're going to add three assertions, all of which look similar to what we did in findTestFab(). We want to make sure the current player's name is Michael (or whatever you sent in for player one in the test) and that both entered players are in the standings text box with ten pennies (since the game has started but no actions have been taken).

With all three assertions, we use the withText() ViewMatcher to verify the displayed text is accurate. The current player name assertion is particularly familiar:

```
onView(
  withId(R.id.textCurrentPlayerName)
).check(
  matches(
    withText("Michael")
  )
)
```

Hopefully the pieces of this assertion are clearer being split out like that. We're doing the same thing with the current standings, but since we're performing two checks on the same View, we can use the allOf() object Matcher to check both items at once. Each assertion will use containsString() to make sure the players and coin values are at least somewhere in the standings text:

```
onView(withId(R.id.textCurrentStandingsInfo)).check(
  matches(
    allOf(
      withText(containsString("Michael - 10 pennies")),
```

```
      withText(containsString("Emily - 10 pennies"))
    )
  )
)
```

Now that the full test is in place, go ahead and run it to make sure it works. It *should*, but if not, check the test output console to see what hints it gives you as to why things aren't working right now. Once it's ready to go, it's time for test number three, adding a third player.

Test Adding a Third Player

This test adds an extra step to our previous test, as we need to enable the third player to be in the game. In the previous test, both players are included by default (since you have to have two people in a game of Penny Drop for it to be interesting), but now we want to get that third player in there.

We do so by clicking the check box to the left of the Player Name input, which turns out to be a very similar process to typing into that Player Name input. We find the parent element (the player row), then get the check box and perform() a click(). As we'll be reusing this functionality again as well, it should also go into TestHelpers.kt:

```
fun clickPlayerCheckbox(parentId: Int) {
  onView(
    allOf(
      withId(R.id.checkbox_player_active),
      withParent(withId(parentId))
    )
  ).perform(click())
}
```

Now we do the same process as we did before but with the additional checkbox click, another typing command, and one more bit of validation:

```
@Test
fun testAddingThreeNamedPlayers() {
  typeInPlayerName(R.id.mainPlayer, "Michael")
  typeInPlayerName(R.id.player2, "Emily")
➤ clickPlayerCheckbox(R.id.player3)
➤ typeInPlayerName(R.id.player3, "Hazel")
  closeSoftKeyboard()
  onView(withId(R.id.buttonPlayGame)).perform(click())
  onView(withId(R.id.textCurrentPlayerName))
    .check(matches(withText("Michael")))
  onView(withId(R.id.textCurrentStandingsInfo)).check(
```

```
      matches(
        allOf(
          withText(containsString("Michael - 10 pennies")),
          withText(containsString("Emily - 10 pennies")),
          withText(containsString("Hazel - 10 pennies"))
        )
      )
    )
  }
```

Run this test as well, and once it's working, we can get one more test created in this class.

Test Adding an AI Player

We're going to once again add a third player, but this time it's one of our AI players in that slot. This means instead of typing in the third player's name, we need to hit the Player/AI <SwitchCompat>, open the AI name <Spinner>, and click one of the items.

The first two steps are done in a similar way to how we click the FAB, but finding data in a <Spinner> requires use of the onData() function. Here, instead of looking at views in our layout, we're instead looking at data in some kind of list. This is generally used with <RecyclerView> lists, but we can also use it with our <Spinner>.

We call onData() and look for AI types, then we pick the one at position 3 (or whichever AI we want) and click it. Once we're into the next screen, we do our checks as we did before with the third check being for the selected AI, in this case Fearful Fred. Here's the full version of that test:

```
@Test
fun testAddingThirdAIPlayer() {
  typeInPlayerName(R.id.mainPlayer, "Michael")
  typeInPlayerName(R.id.player2, "Emily")

  closeSoftKeyboard()

  clickPlayerCheckbox(R.id.player3)

  onView(
    allOf(
      withId(R.id.switch_player_type),
      withParent(withId(R.id.player3))
    )
  ).perform(click())

  onView(
    allOf(
      withId(R.id.spinner_ai_name),
      withParent(withId(R.id.player3))
```

```
    )
).perform(click())

//AI Position #3 is Fearful Fred
//Also, note the use of backticks with the `is` function
onData(`is`(instanceOf(AI::class.java))).atPosition(3).perform(click())

onView(withId(R.id.buttonPlayGame)).perform(click())

onView(withId(R.id.textCurrentPlayerName))
  .check(matches(withText("Michael")))
```

```
    onView(withId(R.id.textCurrentStandingsInfo)).check(
      matches(
        allOf(
          withText(containsString("Michael - 10 pennies")),
          withText(containsString("Emily - 10 pennies")),
          withText(containsString("Fearful Fred - 10 pennies"))
        )
      )
    )
  }
```

We're all set now with testing PickPlayersFragment! Certainly, we could add more tests of value here, but we at least know we can add both human and AI players to a game and things appear to initialize correctly. I say "appear" since we don't know for sure what's going on in the database and something could be funky, but that's why we wrote database and ViewModel tests last chapter.

Next up is the GameFragmentTests class, where we'll make sure slots start as they should and everything is updated correctly after a roll.