I'm not robot

reCAPTCHA

Continue

Androidx fragment testing

Robolectric is intended to be fully compatible with android's official testing library since version 4.0. This way, we encourage you to try out this new API and provide feedback. At some point the Robolectric equivalent will be rejected and removed. Using the AndroidX Test API reduces the cognative load for you as a developer, with just one set of APIs to learn the same Android concept, no matter if you're writing Robolectric testing or instrumentation testing. In addition it will make your test more portable and compatible with our future plans. TestRunner It is now possible to use AndroidX test runners in Robolectric tests. If you need a custom test runner at this time, please check the new configuration and plugin API and let us know if there are any missing extension points you need. Robolectric @RunWith(RobolectricTestRunner::class) public class SandwichTest { } AndroidX Test @RunWith(AndroidJUnit4::class) public class SandwichTest { } Applications Because most Android code is centered around Context, getting the context of your app is a typical task for most tests. Robolectric @Before fun setUp() { val app = RuntimeEnvironment.application as ExampleApplication app.setLocationProvider(mockLocationProvider) } This can be immediately replaced with ApplicationProvider. You might want to import it statically for readability. AndroidX Test import androidx.test.core.app.ApplicationProvider.getApplicationContext @Before fun setUp() { val app = getApplicationContext<LocationTrackerApplication>() app.setLocationProvider(mockLocationProvider) } Robolectric activities are provided robolectric.setupActivity() for a rough-grain usecase where you only need the launched activity to be resumed and ready and visible for users to interact. Robolectric also provides Robolectric.buildActivity() which returns an ActivityController that allows developers to step through the Activity lifecycle. This has proven problematic because it requires developers to fully understand valid lifecycle transitions and possible valid circumstances. Using Activity in an invalid state has unfined behavior and can cause compatibility issues while running on different Android test runtimes or when upgrading to a newer version of Robolectric. ActivityScenario provides replacements for both of these usecas, but places tigher restrictions around lifecycle transitions, i.e. that invalid or incomplete transitions are not possible. If you want a Rules-based equivalent, please use ActivityScenarioRule instead. Robolectric import org.robolectric.Robolectric.buildActivity class LocationTrackerActivityTest { @Test location { // GIVEN val controller = buildActivity<LocationTrackerActivity>().setup() // WHEN controller.pause().stop() // THEN assertThat(controller.get().locationListener).isNull() }</LocationTrackerActivity> </LocationTrackerApplication> </LocationTrackerApplication> Android X Test import androidx.lifecycle.Lifecycle import androidx.test.core.app.ActivityScenario.launch class LocationTrackerActivityTest { @Test fun locationListenerShouldBeUnregisteredInCreatedState() { // GIVEN val scenario = launchActivity<LocationTrackerActivity>() // WHEN scenario.moveToState(Lifecycle.Lifecycle.. State.CREATED) // THEN scenario.onActivity { activity -> assertThat(activity.locationListener).isNull() } Notes that in Robolectric because ui test and event loops run on the same thread, synchronization is not an issue but ActivityScenario.onActivity provides a secure way to access activities, if you need to, that will be guaranteed to be compatible with our future plans. Views Robolectric has a very limited API for Interaction View; most often test authors can only use Android SDK APIs like Activity.findViewById() that are safe because Robolectric tests don't have to worry about syncing between the test thread and the UI. Espresso is a library of matching and interaction display options for instrumentation tests. Since Robolectric 4.0, Api Espresso is now supported in Robolectric tests. import androidx.test.espresso.espresso.onView @RunWith(AndroidJUnit4::class) class AddContactActivityTest { @Test fun inputTextShouldBeRetainedAfterActivityRecreation() { // GIVEN val contactName = Test User val scenario = launchActivity<AddContactActivity>() // WHEN // Enter contact name onView(withId (R.id.contact_name_text)).perform(type)t(contactName)) // Destroy and recreate the Event scenario.recreate() // THEN // Check the name of the retained contact. onView(withId(R.id.contact_name_text)).check(matches(withText(contactName))) } } Robolectric Fragment provides APIs such as SupportFragmentUtil and SupportFragmentController that offer rough and smooth control of the Fragment lifecycle, which requires developers to have a full understanding of lifecycle transitions and valid final circumstances, makes it easy to shoot yourself in the foot for the same reasons as with the Activity above. AndroidX Test provides FragmentScenario, which offers APIS to securely create your fragments under testing and propel them through valid transitions. @RunWith(AndroidJUnit4::class) class FragmentTest { @Test fun testEventFragment() { val fragmentArgs = Bundle() val factory = MyFragmentFactory() val scenario = launchFragmentInContainer<MyFragment>(fragmentArgs, factory) onView(withId(R.id.text)).check(matches(withText(Hello World!))) } Read more vile test fragments here: developer.android.com/training/basics/fragments/testing this article was updated November 29, 2020. Fragments are citizens of many Android apps. When the fragment class was introduced as part of the Android SDK, many people hesitated to in Android apps, especially after a complicated life cycle. Many apps use different approaches: Multiple Activities</MyFragment> </AddContactActivity > </LocationTrackerActivity> </LocationTrackerActivity > One Activity with multiple fragments vs. a mixed approach (multiple Activities and multiple Fragments. Currently, many problems with Fragments have been fixed in the AndroidX version of the fragment and approach with one Activity and many fragments have become increasingly popular. When we look at the Jetpack Navigation components, the main idea is to use some fragments and the Jetpack Navigation components help navigate between them. Testing Android apps right from the start wasn't an easy task, but then, the UiAutomator and Espresso frameworks were introduced that simplified UI testing and many developers, and QA engineers started making a lot of UI test cases, but from the very beginning all UI test cases were tested for applications; as a result, we have several End to End tests. The test is slow, fragile and unstable. Nowadays, more and more people prefer the test component in isolation. Let's talk today about testing Android fragments in isolation and speeding up the Espresso test case. Configure the project that I want to start with the project set up for the fragment testing library. This library can be used with Local and Instrumentation tests. We need robolectric libraries for local testing and devices or emulators to run Instrumentation test cases. Let's add dependencies for local tests and instrumentation. dependencies { ... // Fragment implementation androidx.fragment:fragment-ktx:1.2.5 debugImplementation androidx.fragment:fragment-testing:1.3.0-beta01 // Robolectric testImplementation org.robolectric:robolectric:4.4 // Espresso and tests utilities for local and instrumentation tests testImplementation androidx.test.ext:unit:1.1.2 testImplementation androidx.test:runner:1.3.0 testImplementation androidx.test:core:1.3.0 testImplementation androidx.test:1.3.0 testImplementation androidx.test.espresso:espresso-core:3.3.0 testImplementation androidx.test.espresso:espresso espresso-intents:3.3.0 androidTestImplementation androidx.test.ext:junit:1.1.2 androidTestImplementation androidx.test:runner:1.3.0 androidTestImplementation androidx.test:core:1.3.0 androidTestImplementation androidx.test:rules:1.3.0 androidTestImplementation androidx.test.espresso:espresso-core:3.3.0 androidTestImplementation androidx.test.espresso:espresso-intents:3.3.0 } Note: The fragment-testing library uses a different version than the fragment-ktx one because we use the androidx.test:core:1.3.0, and the fragment-ktx uses an old version of the androidx.test:core library. Once the fragment-ktx library is updated, we can use similar versions for fragment-ktx and fragment-testing libraries. We can also create shared tests that can be run as Local and Instrumentation tests (read more here: Code sharing between local testing and instrumentation). Test fragments by local testing and instrumentation can be run on Android Android Robolectric's library can help him. (We have to use Robolectric 4.0+). The test is fast; we can use the Espresso syntax to make such a test case, but on the other hand, we cannot run these test cases on different devices. From time to time, you can face situations when the app doesn't work correctly on different devices. In this case, the INSTRUMENTATION UI test can be useful. Instrumentation test cases require devices or emulation, and they are slow, but we can run them on different devices simultaneously. However, often you can face situations when all instrumentation test cases are written in the End to End approach. This means we always start the test from the home screen or application, such as the Login or Splash screen. As a result, our End-To-End test case is slow because we have to spend time in navigation to the necessary screen, and only after that, we can start verifying the screen. E2E test case: Add notes Fortunately, we can verify each fragment in isolation, and speed up instrumentation testing. Use the case I proposed to create a test case for a fragment of Add notes from the MapNotes app. These fragments allow us to add records that are connected to the current latitude and longitude. So, let's look at the possible UI of the fragment: the ADD button is disabled by default; The ADD button is enabled when note text exists. Let's visualize the app screen: MapNotes - Home screen Test the fragments I want to start by testing fragments without fragment testing libraries. This can be useful if you are not using fragments from the androidx.fragment package. First of all, we must start the Activity and after that, add Fragments to it. FragmentTestActivity is a simple Activity with containers for Fragments. @RunWith (AndroidJUnit4::class) class AddNoteFragmentTest { @Before fun setUp() { ActivityScenario.launch(FragmentTestActivity::class.java) } ... } The first step of each test case where we want to test fragments in isolation is to add the necessary fragments to this Activity. (SetFragment is a custom method that adds the required Fragments to an Activity.) @RunWith(AndroidJUnit4::class) class AddNoteFragmentTest { @Before fun setUp() { ActivityScenario.launch(FragmentTestActivity::class.java) .onActivity { it.setFragment(AddNoteFragment()) } } ... } Finally, we can add a test case: @RunWith(AndroidJUnit4::class) class AddNoteFragmentTest { private val testNoteText = test note @Before fun setUp() { ActivityScenario.launch(FragmentTestActivity::class.java) .onActivity { it.setFragment(AddNoteFragment()) } } @Test fun shouldDisplayNoteHintForANewNote() { .check(matches(withHint(R.string.add_note_hint))) } @Test fun shouldChangeAddButtonEnableAfterChangingNoteText() { onView(withId(R.id.add)) .check(matches(not(isEnabled()))) onView(withId(R.id.note)) onView(withId(R.id.note)) onView(withId(R.id.add)) .check(matches(isEnabled())) } With this approach, you can test fragments in isolation without any problems, but you need to manually create empty Activities to attach the required Fragments. So, let's take a look at the new approach of testing fragments in isolation with the androidx.fragment:fragment-testing library. The launchFragmentInContainer method allows us to launch fragments containing UI. After that, your fragment will be attached to the Activity root view. @RunWith (AndroidJUnit4::class) class AddNoteFragmentTest { @Before fun setUp() { // detailed version launchFragmentInContainer<AddNoteFragment>(fragmentArgs = null, // Bundle themeResId = R.style.AppTheme, initialsState = Lifecycle.State.RESUMED, // Lifecycle.State factory = null // FragmentFactory) // or // short version launchFragmentInContainer<AddNoteFragment>(temaResId = R.style.AppTheme) } ... } If we want to use custom FragmentFactory to initialize our fragments, we can use factory parameters. We can customize the Fragment parameter using the fragmentArgs parameter. @RunWith(AndroidJUnit4::class) class AddNoteFragmentTest { @Before fun setUp() { val arguments = Bundle().also { it.putString(ARGUMENT_KEY, ARGUMENT_VALUE) } launchFragmentInContainer<AddNoteFragment>(fragmentArgs = argument, temaResId = R.style.AppTheme) } } } } In addition, we can run fragments under the necessary circumstances. Let's imagine that we want to test the fragment in a ON_START state: @RunWith(AndroidJUnit4::class) class AddNoteFragmentTest { @Test fun shouldChangeSomethingWhenFragmentOnPauseState() { launchFragmentInContainer<AddNoteFragment>(initialsState = Lifecycle.State.STARTED, themeResId = R.style.AppTheme) // or val = launchFragmentInContainer<AddNoteFragment>() scenario.moveToState(State.STARTED) } } } Note: We can test non-graphical fragments with the launchFragment method. The fragment will be attached to an empty Activity. launchFragment<AddNoteFragment>(fragmentArgs = null, // Bundle themeResId = R.style.AppTheme, factory = null // FragmentFactory) Instrumentation UI tested the launchFragmentInContainer method creating an empty FragmentActivity with the necessary fragments under the hood. As you can see, this approach is similar to the previous approach, which was popular before because we had activity with the necessary fragments. So, let's discuss the following scenario with the test case: the ADD button is disabled by default; Enter the test record into the EditText record; The ADD button is turned on. @RunWith(AndroidJUnit4::class) class AddNoteFragmentEspressoTest { private val testNoteText = test note @Before fun setUp() { loadKoinModules(listOf(testAppModule))FragmentInContainer<AddNoteFragment>() } menyenangkan harusChangeAddButtonEnableAfterChangingNoteText() { onView(withId(R.id.add)) .check(not(isEnabled())</AddNoteFragment> </AddNoteFragment> </AddNoteFragment> </AddNoteFragment> </AddNoteFragment> </AddNoteFragment> </AddNoteFragment> </AddNoteFragment> .perform(replaceText(testNoteText)) onView(withId(R.id.add)) .check(matches(isEnabled())) } } As you can see, we can avoid ActivityTestRule in the test suite and have more possibilities, such as: Test fragments under the necessary circumstances; Customize bundles for fragments; Clean API to test fragments in isolation. Local UI testing The main benefit of this approach is that we can execute the same test case with Robolectric without changes. I recommend checking the Code share between local tests and instrumentation that describes the test case storage approach with the possibility to run it as a local test case and instrumentation. Fragment testing libraries allow us to test graphical and non-graphic fragments in isolation without additional preparation. Of course, fragments will be attached to the Test activity. This tool allows us to test fragments with local test cases and instrumentation. We can use Robolectric for local tests. So, let's try comparing the execution time of the same scenario for local test cases and instrumentation. We spent about 18 seconds on local testing and 30 seconds on instrumentation testing because we had to install 2 apk to an external device. When the test is complete, we'll also need to copy the test report from an external device to your work station. However, instrumentation testing can be executed on different devices simultaneously. If you have problems on various devices, I recommend using an instrumentation test. In general, I recommend using a shared test module where you can save a test case with the possibility of running it as a local test and instrumentation. Local tests can be executed in pipelines, and you can use instrumentation testing on different devices before release). Resource Resources