Luke Reckard
February 4th, 2020
Rooio

Test Coverage

Our code coverage has never been great, and this metric report reveals that. However, this is due to a variety of factors that contribute to such a low coverage report. On my end, this has been over two months of researching testing in Android to typically no avail. In my opinion, it is a huge area of concern because any block of code that is not tested becomes a liability in the application when it is released to customers, but at the same time seems to be pointless as most of our current tests that "cover" code really just verify that a new activity has started.

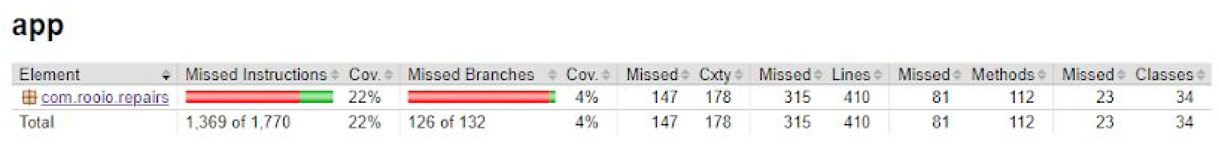
Referring to the latter statement, most of our classes in Java are activities that handle the UI and calls to the API, leaving little room for unit testing as there is barely any logic that needs to be tested or edge cases that occur. It feels hollow when testing a click button that switches pages as a test which behind the scenes is just a listener with a two line UI test (or "instrumented test" in Android). There is a different issue with the API testing, though. Mocking the Android Volley library takes a lot of work and research, and although it may be worth it to make sure that our side of connection is not causing an issue, the API and UI are so tightly coupled that any backend mocking that occurs would need to be verified through a UI test as well. This is something that Android keeps very separate as unit and instrumented tests are not supposed to cross over. It also poses the question if it is even possible to test this aspect of our application, and on top of that in a fast and efficient way as instrumented tests require an emulator and take a lot of time to run.

This leads to the documentation of Android testing. Recently, Google acknowledged the concerns of Android developers feeling overwhelmed by the numerous testing frameworks and claimed to have consolidated that into the new testing libraries AndroidX. Unfortunately, Google clearly does not hire enough technical writers (or good ones), as the documentation for any of the testing frameworks, old or new. becomes a rabbit hole that is easy to get swept up in. A lot of information that is found is contradicting, not clearly explained, definitely outdated and deprecated, or a poorly written example that has errors in it. This becomes a headache for the developer as it remains difficult to just test a simple activity and button listeners in Android and produce a representative code coverage report.

Now for code coverage reports, it seems like Android has a lot of options, but it quickly becomes apparent that you pick your poison. For starters, on any unit test it is possible to run a coverage report that shows uncovered lines and percentages within Android Studio. However, this becomes an issue when 80% of our classes are Activity classes that have mostly UI components involved and cannot be tested without a running context. The fix, seemingly, is to use JaCoCo, or Java Code Coverage, which initially does not unify the two different types of tests but after a lot of head banging is able to create a coverage report. This comes with flaws, as it takes a long time to run both types of tests, does not appreciate a mix of Kotlin and Java files, and does not like running on the emulator in our CI. Researching a fix, I found Robolectric which is now built into AndroidX and can run UI tests on the JVM, without an emulator. Unit tests with UI components that were run through Android Studio pass in seconds, but Gradle on the command line does not appreciate the ingenuity of such a framework and fails all Robolectric tests, thus blocking a chance at a unified code coverage report.

This is basically a long winded analysis of why our code coverage is the way it is just due to technical complications. On top of figuring out how to test such a behemoth of a problem, this information then has to be relayed to teammates and thus lies the problem that prevents us from even practicing TDD.

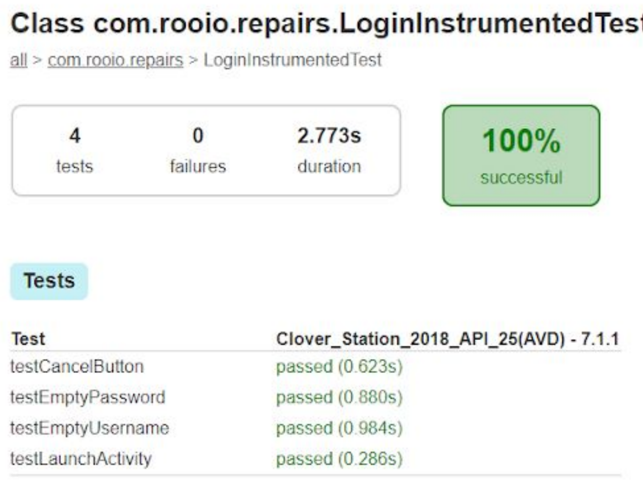
Looking below, you can see a code coverage report of the current state of our application. As mentioned above, Kotlin classes have magically disappeared even though they have UI components covered through instrumented tests. There are 315 Java code lines missed out of 410, and although it is only 22% of our application's Java, this metric is not accurate based off the many more lines of code that exist in Kotlin.



app

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.rooio.repairs		22%		4%	147	178	315	410	81	112	23	34
Total	1,369 of 1,770	22%	126 of 132	4%	147	178	315	410	81	112	23	34

Here is an example of the Login tests that were not included but passed.



Class com.rooio.repairs.LoginInstrumentedTest

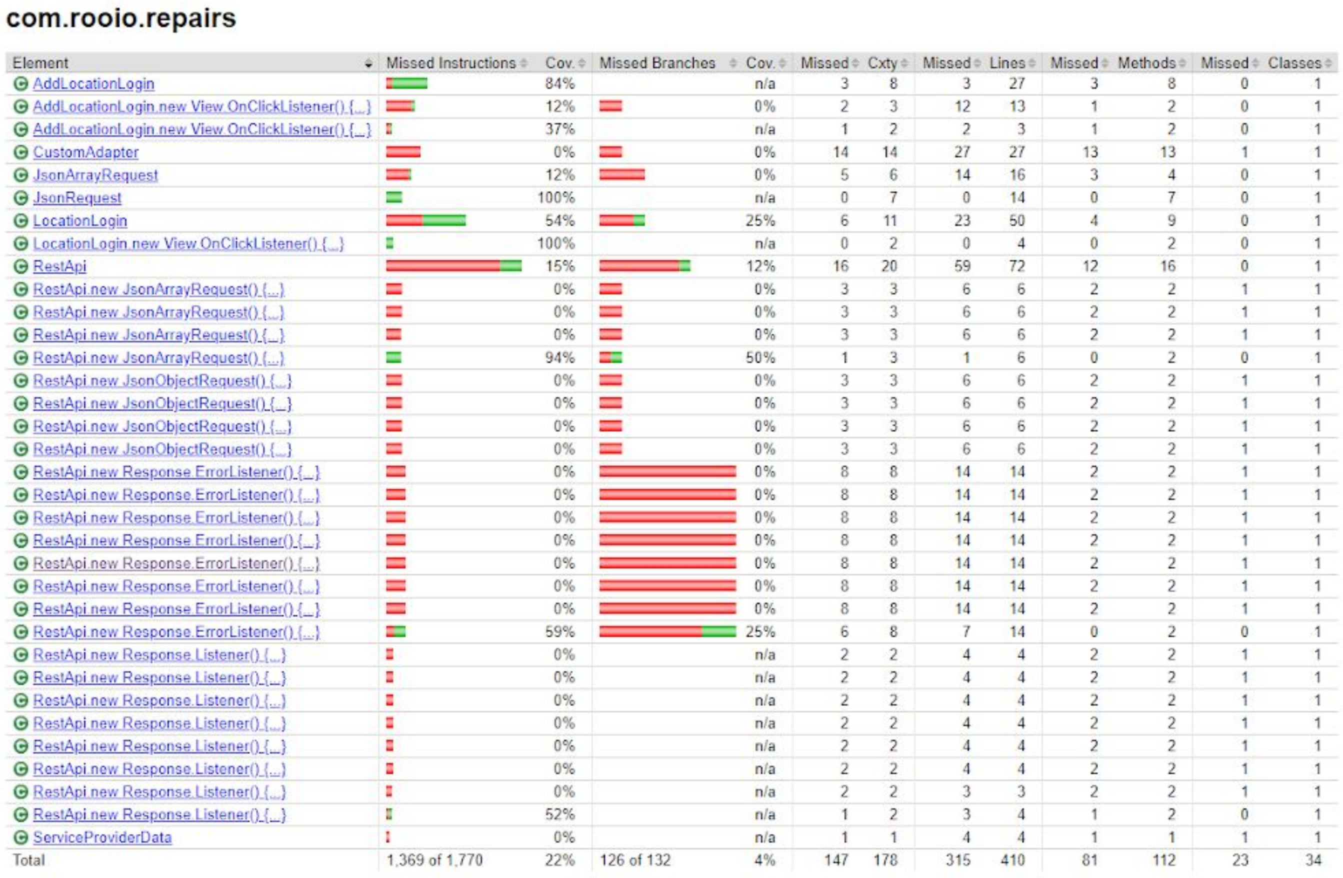
all > com.rooio.repairs > LoginInstrumentedTest

4 tests | 0 failures | 2.773s duration | 100% successful

Tests

Test	Clover_Station_2018_API_25(AVD) - 7.1.1
testCancelButton	passed (0.623s)
testEmptyPassword	passed (0.880s)
testEmptyUsername	passed (0.984s)
testLaunchActivity	passed (0.286s)

And here is a closer look at the missed lines for each class. JaCoCo unfortunately reports lambdas and onClickListeners as classes which can become overwhelming, especially in the case of RestApi (our class that utilizes Volley to send and receive JSON Requests through listeners). This also shows how some coverage was achieved for the RestApi, but solely because of some test attempting to send a request and receiving an error that needs to be refactored.



com.rooio.repairs

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
AddLocationLogin		84%		n/a	3	8	3	27	3	8	0	1
AddLocationLogin.new View.OnClickListener() {...}		12%		0%	2	3	12	13	1	2	0	1
AddLocationLogin.new View.OnClickListener() {...}		37%		n/a	1	2	2	3	1	2	0	1
CustomAdapter		0%		0%	14	14	27	27	13	13	1	1
JsonArrayRequest		12%		0%	5	6	14	16	3	4	0	1
JsonRequest		100%		n/a	0	7	0	14	0	7	0	1
LocationLogin		54%		25%	6	11	23	50	4	9	0	1
LocationLogin.new View.OnClickListener() {...}		100%		n/a	0	2	0	4	0	2	0	1
RestApi		15%		12%	16	20	59	72	12	16	0	1
RestApi.new JsonArrayRequest() {...}		0%		0%	3	3	6	6	2	2	1	1
RestApi.new JsonArrayRequest() {...}		0%		0%	3	3	6	6	2	2	1	1
RestApi.new JsonArrayRequest() {...}		0%		0%	3	3	6	6	2	2	1	1
RestApi.new JsonArrayRequest() {...}		94%		50%	1	3	1	6	0	2	0	1
RestApi.new JsonObjectRequest() {...}		0%		0%	3	3	6	6	2	2	1	1
RestApi.new JsonObjectRequest() {...}		0%		0%	3	3	6	6	2	2	1	1
RestApi.new JsonObjectRequest() {...}		0%		0%	3	3	6	6	2	2	1	1
RestApi.new JsonObjectRequest() {...}		0%		0%	3	3	6	6	2	2	1	1
RestApi.new Response.ErrorListener() {...}		0%		0%	8	8	14	14	2	2	1	1
RestApi.new Response.ErrorListener() {...}		0%		0%	8	8	14	14	2	2	1	1
RestApi.new Response.ErrorListener() {...}		0%		0%	8	8	14	14	2	2	1	1
RestApi.new Response.ErrorListener() {...}		0%		0%	8	8	14	14	2	2	1	1
RestApi.new Response.ErrorListener() {...}		0%		0%	8	8	14	14	2	2	1	1
RestApi.new Response.ErrorListener() {...}		0%		0%	8	8	14	14	2	2	1	1
RestApi.new Response.ErrorListener() {...}		0%		0%	8	8	14	14	2	2	1	1
RestApi.new Response.ErrorListener() {...}		59%		25%	6	8	7	14	0	2	0	1
RestApi.new Response.Listener() {...}		0%		n/a	2	2	4	4	2	2	1	1
RestApi.new Response.Listener() {...}		0%		n/a	2	2	4	4	2	2	1	1
RestApi.new Response.Listener() {...}		0%		n/a	2	2	4	4	2	2	1	1
RestApi.new Response.Listener() {...}		0%		n/a	2	2	4	4	2	2	1	1
RestApi.new Response.Listener() {...}		0%		n/a	2	2	4	4	2	2	1	1
RestApi.new Response.Listener() {...}		0%		n/a	2	2	4	4	2	2	1	1
RestApi.new Response.Listener() {...}		0%		n/a	2	2	3	3	2	2	1	1
RestApi.new Response.Listener() {...}		52%		n/a	1	2	3	4	1	2	0	1
ServiceProviderData		0%		n/a	1	1	4	4	1	1	1	1
Total	1,369 of 1,770	22%	126 of 132	4%	147	178	315	410	81	112	23	34

There is a lot that can be worked on in terms of untangling the web of testing and refactoring.