Testing strategies #6

keith started this conversation in General

keith on Mar 1, 2021 Maintainer

Comentário original em inglês - Traduzir para português

I'm curious to hear from folks about their general testing strategies, automated or manual, and what has been the most useful over time for your apps.

↑ 95 110

21 comments · 57 replies

Oldest Newest Top

keith on Mar 1, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

Lyft currently has:

1. Lots of unit tests, something we're still trying to foster the culture around, but coverage has improved significantly over time. At this point we still run all affected tests (defined by the reverse dependency tree of the files in your change) on changes, but I'm hoping at some point we can improve PR CI time by breaking out some amount of realistic breakage potential and split out some pre-submit / post-submit tests instead.
2. Few snapshot tests, these are in a POC phase. I'm curious what folks have found to be the best workflow around these, and if they're worth the trouble. I've heard some success stories from folks who spun up a web service for diffing them and easily accepting new snapshots, but this UX seems like the most difficult productivity hurdle.
3. Few UI tests, we have a bunch of these now, mostly written by our QA teams. They aren't ever blocking on PRs but folks can run them manually if they know they could break them given their changes. I'm curious here what folks are doing to continue to scale these, and how they triage breakages, who tracks down the core issues, etc.
4. Manual tests, we block releases on manual testing at this point. We're hoping that this is reduced over time as we backfill more of our UI test cases though.

↑ 7 31 6 replies

Show 1 previous reply

keith on Mar 2, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

Yea I guess at this point that's how ours work, we have some "smoke" tests that are release blocking and everything else that's not. Teams run them on their own schedules or manually, and get notifications of failures wherever they specifically choose. This is working well for us at this point, although I do think sometimes failures languish, but that might be a necessary trade-off.

dgupta1-gd on Mar 4, 2021

Comentário original em inglês - Traduzir para português

Regarding point 3, in GD Mobile app, we have smoke tests/E-2-E tests for the core features that are run daily/nightly against the nightly builds and throws the success/failure on slack and email to the whole team. QA is the owner of those and looks into any investigations if needed. These tests do not block PR merges and are not run in CI pipeline, but have the capability to be run on a local PR by any developer.

BenziAhamed on Mar 5, 2021

Comentário original em inglês - Traduzir para português

> At this point we still run all affected tests (defined by the reverse dependency tree of the files in your change) on changes

Curious, how do you folks figure this reverse dependency tree and get the list of affected tests?

gsavit on Mar 5, 2021 edited

Comentário original em inglês - Traduzir para português

Curious about point 4! It's important that it's blocking, but that can present a bunch of coordination challenges and overhead. How do you run this final QA? Any process or tools you'd call out? How does the team get on the same page about the status of that blocking step?

tinder-maxwellelli... on Mar 8, 2021

Comentário original em inglês - Traduzir para português

@BenziAhamed There are open source options for determining the affected set of tests, https://github.com/Tinder/bazel-diff (disclaimer I do work on that project)

1

Write a reply

dflems on Mar 1, 2021

Comentário original em inglês - Traduzir para português

We (Spotify) have a lot of unit and integration tests (32k at the time of writing, not including the tests in our shared C++ codebase). A lot of them still run in app-hosted test bundles for legacy reasons, which we're slowly fixing. These tests are hard to cache for that reason.

We're also playing around with snapshot testing. It's been super helpful for validating views in different rendering modes (dynamic font sizing, RTL layout, etc). It's currently only in our shared UI toolkit library and we're storing them in `git-lfs` (via artifactory). Despite being fairly young internally, the total size of all the images is getting a little unwieldy (1600 images). It's a lot to force people to check out if they're not planning on running all of the tests. Might prefer a high-availability service that downloads/caches them on-demand instead, even if there might be the occasional network flakiness from it. We haven't been bitten yet by an Xcode upgrade that forced us to re-run all the snapshots. Bound to happen sooner or later. Will cross that bridge when we get there.

We have ~500 XCUI tests that are mostly automated replacements for what would otherwise be manual end-to-end testing. Mostly non-blocking except for a very small suite of stable smoke tests that verify that the UI testing framework is happy pre-submit.

↑ 5 16 4 replies

Category
General

Labels
None yet

39 participants
and others

Notifications
Subscribe
You're not receiving notifications from this thread.

keith on Mar 2, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

What's the UX for folks updating snapshots if they're not manually running them all?

Abeansits on Mar 2, 2021 edited

Comentário original em inglês - Traduzir para português

Thank you for sharing. Also curious to hear who writes and maintains the XCUI tests?

5

bielikb on Mar 3, 2021

Comentário original em inglês - Traduzir para português

Re large blobs/images.
Have you tried to play around partial clones/sparse checkouts?
This could improve the devex when dealing with (mono)repos with huge amounts of data.
Theres pretty interesting github universe talk on this topic, you might want to check it out:

https://githubuniverse.com/Optimize-your-monorepo-experience/

4

raphaelguye on Nov 26, 2021

Comentário original em inglês - Traduzir para português

your XCUI tests ar running on which environment? Do you have a special testing environment or you can deal with an existing environment (preprod, int, dev or even prod) ?
If you don't have a test environment, how do you prepare / clean the data without disturbing too much this environment?

Write a reply

gergelyorosz on Mar 2, 2021 edited

Comentário original em inglês - Traduzir para português

As of Oct 2020 (when I left), Uber had:

1. Unit tests, lots of them. All business logic would be covered with these.
2. Many snapshot tests on iOS. I cannot remember the exact reason why it was iOS only, but that's where we ended up with. In general, the teams liked it, but a problem has been how the snapshots are stored in the repo, and they slow down cloning.
3. UI tests, the number fluctuating over the years. Flakyness and maintenance have been issues.
4. Sanity tests (manual tests) defined by the teams, and executed by a testing team.
5. Exploratory testing done by contractors, trying to "break" the app with unexpected ways of testing.
6. Security testing: one-offs, with specialists / vendors. I'm mentioning this as this becomes increasingly important with testing.

4 21 10 replies

Show 5 previous replies

thedavidharris on Mar 5, 2021

Comentário original em inglês - Traduzir para português

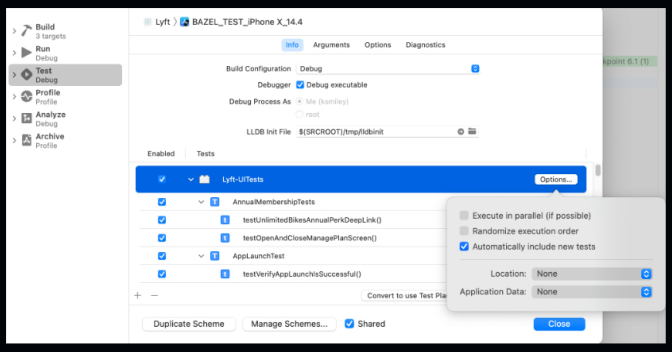
@tinder-maxwellelliott was there any way to actually get that parallelism though without dropping to xctool/Buck/Bazel? We converted all our tests to logic tests in the hopes we could figure out it, and it is still a bit more performant regardless, but never figured out the parallelism piece while we were still dealing with xcodebuild.

Something like `swift test --parallel` but for iOS tests on xcodebuild would be amazing. From when I asked some Apple engineers at WWDC on this they basically said they were only looking to better support running parallel sims for an existing module, not running modules in parallel.

keith on Mar 5, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

I think this works with xcodebuild today too right? You can check some boxes in the scheme for this:



thedavidharris on Mar 6, 2021

Comentário original em inglês - Traduzir para português

That works if all your tests are in the same bundle, but if they are in different bundles you can't run the bundles in parallel as described in https://github.com/facebookarchive/xctool#parallelizing-test-runs or https://github.com/uber/ios-snapshot-test-case/blob/master/docs/LibraryVsApplicationTestBundles.md#library-tests.

Which I'm wondering now if there's potential improvement in just combining all of our test bundles into one mega test bundle we then run in parallel?

keith on Mar 8, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

Ah ok by bundle. That's definitely a benefit we get from bazel. I've actually still considered the mega test bundle approach with bazel to avoid multiple simulator overhead, and in general the setup and teardown time of those (lots of bazel specifics that cause that to be kinda slow), but I think it would make a few things harder if you analyze the results and want to do something with them per bundle.

Also depending on how you cleanup after each test bundle I guess you could get some unfortunate pollution there.

thedavidharris on Mar 8, 2021

Comentário original em inglês - Traduzir para português

Definitely would make analysis harder and as you mentioned, you couldn't do dependency analysis as easy based on the rulekeys or anything.

Write a reply

birwin93 on Mar 2, 2021 edited

Comentário original em inglês - Traduzir para português

At Robinhood

1. Mostly unit tests
2. ~400 snapshot tests. We don't use anything special here, except a custom script that handles uploading/downloading references images from S3. We've had a few tests that randomly started failing on machines that use xcode 12.4, but we've confirmed that if they are recorded on a machine using xcode 12.4 it fixes the issue. No other issues in the last ~2 years though.
3. We have 10-15 diff/PR blocking E2E UI tests with plans to grow this to closer to 50+ over the next few months. We're investing in this heavily right now. We built a record/replay service and have a job that re-records all tests each morning, then only uses replayed traffic as diff/PR blockers. We're currently heads down on some stability work, but early results have been surprisingly positive.
4. We're experimenting with hermetic UI tests. On iOS side we've built a low level socket connection to allow the XCUITest runner and application to send messages to each other. We also launch a different UIApplication that doesn't load the full app and instead just allows us to load a single feature module to a specific screen. No networking and all tests data is mocked/preloaded/cached.

Before each release we run the full test suite + have a much larger manual QA regression test plan. We have a small time working on rethinking our testing strategy and find a more scalable way to automate more of our manual QA tests. I'll definitely have a lot more to share here over the next few months, especially around E2E and hermetic UI testing

3 13 6 replies

Show 1 previous reply

birwin93 on Mar 2, 2021 edited

Comentário original em inglês - Traduzir para português

> this is for updating network traffic payloads or recording something else?

It's a server version of OkReplay. We run our UI tests in record mode each day, where all the requests proxy through the service to our dev environments, and record traffic into tapes. We then commit the tapes to source control, then when we submit diffs, we upload the tapes to the service which then replays the traffic when running the tests. We've built some logic into the record/replay engine itself to help handle edge cases we've seen, especially around polling.

This effort is still in the early stages, but we've seen some really promising results with ~15 tests and focusing to try and expand this suite to way more.

> did you investigate any of the other frameworks in this space? I think I've heard of some other companies writing something similar, but I guess the alternative is going down the earlgrey path?

We tried using earlgrey a few years back. I think it was the early days of earlgrey2. Ran into issues just getting the tests to run and at the time wasn't sure if the additional functionality was worth it, so we went with XCUITest. It's definitely something I've been looking into recently, but still not sure if the effort to migrate all our tests would be worth it. XCUITest seems to be constantly improving and it might be easier to solve some of these one-off problems with specific tooling rather than use a whole new testing framework.

gsavit on Mar 5, 2021

Comentário original em inglês - Traduzir para português

> Before each release we run the full test suite + have a much larger manual QA regression test plan.

Anything you can share on how the manual regression works, in terms of tools/process/coordination? How is the status of this regression surfaced to the team?

birwin93 on Mar 12, 2021

Comentário original em inglês - Traduzir para português

> Anything you can share on how the manual regression works, in terms of tools/process/coordination? How is the status of this regression surfaced to the team?

Yeah absolutely!

- We cut builds every Friday night. QA has an oncall that goes through the test plans Monday and sometimes it bleeds into Tuesday.
- All test plans are kept in TestRail. Most teams have a QA engineer who defines test plans.
- If any regressions are found, the iOS/Android/Web oncall is notified in a dedicated slack channel

2

tirodkar on Oct 5, 2021

Comentário original em inglês - Traduzir para português

@birwin93 we've updated a lot in EarlGrey 2.0 and it's in GA. The source is at head all the time and all the functionality we previously had (and more) is present in open-source.

jazz-mobility on Nov 30, 2022

Comentário original em inglês - Traduzir para português

I used EarlGrey 2.0 while writing UITests for Delivery Hero consumer apps. (2019), and it works faster and more reliably due to resource synchronization.
My challenge working with the EarlGrey tests was slow updates to the framework, which caused us to get stuck with the old Xcode versions.

Write a reply

PeqNP on Mar 2, 2021

Comentário original em inglês - Traduzir para português

Nordstrom:

1. ~29,800 unit tests with a 100% pass rate -- otherwise code can not be pushed to `master`. However, we can experience flakiness with tests that use `BrightFutures` -- as there's no way to fake them. There are also difficulties ensuring the test state isn't polluted by previous test runs within the same spec file. This prevents us from running tests in random order.

2. ~1,500 UI tests, using a thin wrapper around XCUITest API. These are non-blocking, but take a while to run. We have never had a 100% pass rate with UI tests due to timing issues, network stubs that aren't updated when a service changes, or the UI test steps weren't updated when a change was made to the system. We also record scenarios and store network stubs in a separate repository resulting in thousands of network stubs. It is a challenge to update test stubs when a network service changes. It can also be difficult to find the respective test that needs to be updated when a feature is changed.
3. CI which runs unit tests and UI tests at regular intervals. Build breakages or unit test run failures require that it be repaired before new code can be pushed to `master`.
4. We have a handful of manual tests that are ran only when necessary (and possibly a handful of others that don't fall into this category). For example, logic was changed in the module that deals with notifications.

In a perfect world, I would like to see much faster UI test runs, which is somewhat achievable in unit test land when we use KIF. It should be easy to create and maintain UI tests. Having some type of flow diagram of the entire app to understand which tests are affected by a change you make in the UI would greatly increase our productivity. For example, when you make a change in a VC it would highlight the parts of the flow that are affected. Lastly, a better testing grammar that is easy to read, write, organize, provides facilities to configure the app launch *by test* (feature flags, configuring signed in user, etc.), and doesn't require several seconds to initialize before a test can be ran.

↑ 4 ☺ 👍 13 🚀 4 1 reply

alexs555 on Dec 2, 2022

Comentário original em inglês - Traduzir para português

Sorry for necroposting, but what tool do you use in order to run 1500 UI tests?) Default `xcodebuild test-without-building` will take a lot of time on a single machine, so we had to build a special tool for this task in order to parallize test run across multiple mac minies. But was your choice?

☺ 👍 1

Write a reply

lwise on Mar 2, 2021 edited

Comentário original em inglês - Traduzir para português

Shopify Mobile (Android):

1. All tests are run on CI which is required to pass to merge a PR. CI is painfully slow (~25 mins), so most developers run relevant tests manually before opening a PR. Our builds are split up into many pipelines/modules; otherwise times would be way longer.
2. ~8400 unit tests with ~100% stability.
3. ~2300 screenshot tests with ~100% stability (except for one module with around 5 notoriously flaky orientation change tests). We use our own tool for this. Having so many screenshots was a pain point when re-theming the entire app, but hasn't been otherwise. If a developer is writing a new feature or intentionally changing the UI of a screen they will need to re-record the screenshot before merging. Screenshot diffs can be viewed in the PR with GitHub.
4. We have ~20 automation/end-to-end tests at about 98% stability. These are our most flaky tests, but also have been extremely important for catching production-only issues.

Our main pain points have been how long it takes CI to run (~11 minutes to build APK and ~15 minutes of testing) as well as failure of our BuildKite testing infrastructure in general (timeouts, network errors, etc). These infrastructure issues, the few flaky orientation tests, and legitimate failures bring our master stability down to 85%, which we'd like to improve.

↑ 1 ☺ 👍 13 1 reply

maxlinsenbard-qz on Mar 2, 2021

Comentário original em inglês - Traduzir para português

> All tests are run on CI which is required to pass to merge a PR. CI is painfully slow (~25 mins)

Ditto for us at Quizlet (iOS, at least). What's helped lighten that load a bit is enabling GitHub auto merge capabilities on the repo so that you can initialize a merge on CI Check completion versus having to wait nearly 30m for everything to complete. Great for those PRs with a lot of refactors/change requests!

> ~8400 unit tests with ~100% stability.

Living the dream 😮

> These infrastructure issues, the few flaky orientation tests, and legitimate failures bring our master stability down to 85%, which we'd like to improve.

It's actually super refreshing seeing other orgs out there with the same issues (as annoying as it may be). We're using a UI testing infrastructure called KIF and it's been a double edged sword of helping keep UI tests fast, but seemingly at the cost of stability since nearly 100% of our "flaky" failures in this library are solved when doing a `Rerun from failed` on CircleCI.

☺ 👍 3 ❤ 1 👀 2

Write a reply

thedavidharris on Mar 2, 2021 edited

Comentário original em inglês - Traduzir para português

At Ford for our FordPass/Lincoln Way iOS apps:

1. Heavily unit test based, somewhere around 20,000 across all our modules.
2. Lots of snapshot tests that run with our unit tests, just off of https://github.com/pointfreeco/swift-snapshot-testing.
3. Coverage is good, but there's a lot of variation we're continually adding tests for. We do TDD pretty heavily, but our apps have to deal with lots of regional and product variation based on the account so always things we need to be adding in, so our unit test numbers consistently grow.
4. Performance is decent, with a lot of optimization just within Xcode and xcodebuild it takes ~24 minutes to do a full build and run on ephemeral build nodes. No stability issues, we control for flakiness and asynchronous things pretty well.
5. No UI tests, for reasons I'll note below. Lots of manual testing. There's a certain level of mocking we can do around our connected vehicle networks, but we do have to do a fair amount of end-to-end tests with actual hardware over all sorts of protocols to communicate between app and vehicles including bluetooth and USB.

Challenges we've had and are looking to improve:

- Running tests on a per-module basis for each PR based on the dependency graph as **@keith** described. We feel this should be doable soon and have quite a bit of work to do on cleaning our dependency graphy so it's more horizontal to get more of these benefits.
- Running module tests one-by-one is really slow. None of our tests use a test host (we mostly avoid UIControl event loop issues by using this gist: https://gist.github.com/thedavidharris/3f3ddd5304cf23c89a959434027ee139), so in theory can run headless in parallel, and we did have this working temporarily when we were building our project with Buck, but had to abandon Buck for some internal reasons and haven't found a good way to do this with just xcodebuild.
- We mostly only test snapshot tests with English localization, and have to put in scripts to update all our snapshot tests whenever our CI pulls in new strings. We'd really like to figure out a good way to pseudolocalize this or use dev hints instead of translated values that are a bit more stable.
- UI tests ended up being so much more trouble than value, as we were having to maintain tons of server mocks, flakiness, and just general strategic questions on the right way to use UI tests. As mentioned above, they just leave too many questions: they are flaky, it's not necessarily easy to turn into a bug report to explain what went wrong and who is responsible, etc. We'll probably look to use these just to automate some of our core flows, but more as just a helper to relieve some manual end-to-end pressure.

↑ 3 ☺ 👍 4 4 replies

keith on Mar 3, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

> had to abandon Buck for some internal reasons

Separate from the original topic, but I'd love to hear more about this!

1

thedavidharris on Mar 3, 2021 edited

Comentário original em inglês - Traduzir para português

> Separate from the original topic, but I'd love to hear more about this!

Might be worth starting a build system discussion, I see a number of folks/companies here using Bazel and Buck (we had Buck fully working and some early POCs with Bazel).

At a real high level there were are some small technical things where we have some internal tooling that only works on xcodebuild or required some level of custom linking scripts that we couldn't or didn't want to port over. We've actually solved most of those, but haven't revisited it and migrated to https://github.com/tuist/tuist to do project generation and better manage all our modules and still have things work with xcodebuild.

Really the kicker was the dev experience though. We felt that for our specific team, we weren't going to be able to invest in dev tooling enough to make the Buck/Bazel experience frictionless and painless enough that it would be a net positive to the dev team once we considered the workflow changes and the comfort zone that they had. Outside of our engineering managers and tech leads we didn't feel that we had the expertise to really maintain this across our teams that are pretty large and distrubuted across the whole globe and various levels of team and individual experience.

Across both of those reasons though we had some difficulties in getting build caching setup efficiently especially across all those machines in the five countries we have dev teams in, and just had really large size of built artifacts that didn't play nice with CircleCI at the time (we probably could've fixed but just never investigated), so in lieu of not being able to get the build caching, we made a call just to go with declarative project generation but built on xcodebuild.

7

zachgrayio on Mar 5, 2021

Comentário original em inglês - Traduzir para português

Also **very** interested in this, and yeah a new thread about build systems seems inevitable, though it might be hard to keep that one on the rails given the differing opinions and preferences on the matter -- stackoverflow closes down "opinion-based" questions pretty quick and that's what comes to mind here.

That said, I'm sure there is a way to have such a conversation constructively here with a little bit of consideration.

zachgrayio on Mar 5, 2021

Comentário original em inglês - Traduzir para português

Ill-advised as it may be, I started #41 on the topic 😅

Write a reply

erikkerber on Mar 3, 2021 edited

Comentário original em inglês - Traduzir para português

Target

- **10,000+ unit tests.** It feels extremely rare that these fail with code changes after they are originally authored, though sometimes build errors in test suites offer some signal something might have gone awry. In a recent conversation I had with a few at Target, we came to the conclusion that the vast majority of the value we get from unit tests happens when code is first written – helping a developer flush out the design and edge cases. They didn't seem to do much in terms of protecting against regressions though.
- **Snapshot tests** we PoC'd and put on ice a few months later. The staffing levels we had on our platform team at the time couldn't really invest what it would take to make the DevX for managing test images reasonable, similar to what others have pointed out. It's a bit of a shame because I always felt just like "just as a picture is worth 1,000 words, a screenshot comp is worth 1,000 assertions". In other words, snapshot tests account for what I feel is the biggest challenge with...
- **UI Tests.** We've had about 4 distinct pushes towards leveraging UI tests over the last ~7 years, each time finding it difficult to land in a sweet spot of them being both useful and maintainable. From my experience, the most value comes from just asserting that a flow can execute to completion without crashing or not finding an element, which can be great for smoke tests. Because UI tests have an exponentially slower feedback loop than unit tests, and because they are much harder to debug, ownership and maintenance is a nut we have yet to crack.
- **Beta Testing** through TestFlight (maxed out 10k users) paired with a crash/error monitoring tool. We release to our beta group immediately after a release branch is made and that build runs on TF for ~6-7 days before we go to the App Store. We've found the TestFlight feedback extremely useful, and have worked to "democratize" it out of App Store Connect via various ASC/Fastlane tools so not everyone needs to go fishing for an ASC account (and deal with the slow as dog ASC website). *This is where we detect most of our crash and network-based issues before production.*
- **Manual Testing** managed through a tool that stores and organizes manual test cases. Culturally, we're at an inflection point as to whether we continue this practice or not – as the time and cost for manual testing (and sometimes manual testers) is significant and can be a bottleneck.

4 15 3 replies

gsavit on Mar 5, 2021

Comentário original em inglês - Traduzir para português

Impressive Beta practice compared to many other orgs - can you share any process you use to help formalize it? For example, who's in charge of that 6-7 day "soak" and is there visibility into where things stand during that period? Can you describe the "democratization" setup a bit?

erikkerber on Mar 5, 2021 edited

Comentário original em inglês - Traduzir para português

Rough TestFlight Process

We operate with a "release captain" model, which is really just a point person who handles everything that has yet to be automated. That person, on a pretty regular schedule, will do something essentially like:

Noon on a Tuesday:

1. Make a release branch
2. Bump the marketing version of the main branch
3. Submit the resulting release branch to TestFlight

Friday by or around end-of-day:

1. If there are any cherry picks to the release branch, submit the latest release build again to TestFlight
2. Do a quick check for that version on whatever monitoring tool you have
3. Quickly glance over TF feedback (see below)

Monday AM:

> **NOTE** *At this point, the release branch will have "soaked" for about a week*

1. Cursory check of monitoring systems for stability regressions on the latest TF release
2. Release to 100% (we find that TestFlight **@10k** provides statistically significant adoption to skip phased rollouts). Our SLOs aim for

4-9's, and if stability is below that number we make a call based on severity if we release or hold.

Apple allowed for automatic updates on TestFlight as of ~November and it is now *extremely* effective at getting sessions on a build in short time. I think out of the 10k testers we have, it's pretty normal to see ~100,000 sessions in a week. To keep this rate high, it's important to automate the removal of inactive testers (2 months or so without usage).

TestFlight Feedback "Democratization"

At any big company – especially any big company that is more than just an app – accessing the data behind App Store Connect is a game in managing nearly infinite passively interested parties (devs, product, QA, sales, marketing, leadership, etc...) . This is especially true because Apple does not provide any sort of SAML/Federated Identity type of login, so unless you want to somehow manage thousands of ASC users (!), you'll have pull information out of ASC to make it available to more people at your company.

We've built (well, prototyped) an internal website that presents information about TestFlight builds (feedback, sessions, installs, crashes) to anyone granted access within Target. This is not only open to far more people than those that have ASC accounts, it's also a lot faster – caching feedback and avoiding the 10-second App Store Connect spinners.

2

gsavit on Mar 8, 2021

Comentário original em inglês - Traduzir para português

This is *super* helpful, and impressive, thanks so much for sharing! The difficulty of ASC access/visibility for all the different stakeholders resonates especially - have experienced this on teams, and now working on a project that aims to address it (and various other release-related pain points)...

Write a reply

noahsmartin on Mar 3, 2021 edited

Comentário original em inglês - Traduzir para português

Snapshot tests were critical for Dynamic Type adoption when I worked at Airbnb. Months before enabling the feature on the App Store I added a variant to our Happo pipeline and developers started fixing large font issues in their features. This was much more scalable than when I was manually testing the app and asking feature developers to fix UI components. I'm sure the team will have more to say about this at the upcoming mobile tech talk.

Manual regression testing on release builds was also helpful to catch any remaining features that didn't have screenshot coverage. I asked the QA team to test larger font sizes ~30% of the time, since that was the percentage of users with a larger than default size.

I'm also a fan of build products testing. Things like verifying that all the linked libraries are correct, size of files make sense, and all the right data is included (bitcode, extensions, entitlements). I did this manually at Airbnb when we were transitioning build systems.

Once we shipped an update that was accidentally built with the wrong version of Xcode causing some major issues. I definitely wished we had build product tests automated then! Haven't heard about a company doing this on a large scale yet, but it looks like Jerry Marino had a similar idea: http://jerrymarino.com/2018/09/22/ios-build-validation.html I'm also adding some of these features into Emerge.

3 9

0 replies

Write a reply

fdiaz on Mar 3, 2021 edited

Comentário original em inglês - Traduzir para português

Airbnb (iOS) as of today (March 2021) has (building on what Noah mentioned):

- **Unit tests:** ~10k unit tests that run on every PR. We bundle all of our unit tests into one big target for performance reasons. Today they run at ~15 mins in CI. We use vanilla XCTest and also support Quick and Nimble.
- **Snapshot tests:** ~30k screenshots testing that run on every PR. These are split into Screens (full UIViewControllers) screenshots and components (UIViews) screenshots. We run these with multiple variants (iPhone / iPad / a11y - i.e. extra large font). They take ~30 mins to run on CI. We upload these screenshots to AWS and upload them to a 3rd party service called Happo. This service compares the screenshots and gives us a nice interface for our developers to accept or reject when a screenshot is different. This check blocks PRs unless all the screenshots are the same or the developer manually accepted the changes. We have had some problems with flakiness. A common problem that we've solved at the infrastructure level was using `Date()` in a UI component which changes every day / month and thus the screenshot changes too. We do see some flakiness sometimes with font rendering or 1 px changes and whenever Xcode updates too but IMHO the volume of these is pretty small compared to the value we get from this. **@thedrick** gave a talk that goes into more detail here (starts at 16:30).
- **UI Tests:** We use *UI Tests* to refer to a test that extensively mocks its dependencies and tests the interaction of one screen with its components or the interaction between multiple screens. We have 0 tests today but we're investing in this area in 2021.
- **E2E tests:** We use *E2E tests* to refer to tests that mimic the end-user experience. That is, they start from the entry point of the app, they login and go through a complete flow. They have little to no mocking on the client side and they hit real endpoints in a close to production environment. We've attempted a couple of times to have these types of tests in the past but they've have been disabled due to either flakiness or being hard to keep up to date when the product changes. We have 0 tests today but we're investing in this area in 2021.
- **Manual testing:** We have a team that tests each beta version of the app before we release to the app store (usually weekly) and lets us know of any issues. They can raise release blocking issues.

5 15 1

8 replies

Show 3 previous replies

wswebcreation on Mar 19, 2021

Comentário original em inglês - Traduzir para português

Love to hear those challenges you have with Appium **@bigyelow** , can you elaborate a bit more about them

ffittschen on Apr 8, 2021

Comentário original em inglês - Traduzir para português

@fdiaz I'd be really interested in the talk about snapshot testing! Do you know when the video will be available?

fdiaz on Apr 8, 2021 edited

Comentário original em inglês - Traduzir para português

@ffittschen updated the original post with the link, but for convenience here it is too (16:30).

1

tinder-maxwellelli... on Jul 9, 2021

Comentário original em inglês - Traduzir para português

@fdiaz I am interested in moving snapshot test comparisons out of the main blocking PR builds. I have a hunch that 99% of these will never fail and when they do, it will be by the author themselves or during an Xcode migration. The big upside here is not running any hosted tests in PRs, which will drastically reduce build times.

What is a sound strategy for this? I remember at Uber we had a ticketing system where after the fact you would be assigned a ticket for you to view the snapshot changes, if you did not want those then you needed to make another PR

fdiaz on Jul 9, 2021

Comentário original em inglês - Traduzir para português

@tinder-maxwellelliott what we currently do is that we show the diff to the PR creator *at PR time* and then they can accept or reject the changes. Each developer is free to accept any changes they see, but for any change in a PR we notify the owner of the module that a diff is happening in that PR too, so they're aware that a PR has changes to a screenshot they own.

We have struggled with CI times when taking screenshots too but we've reduced this time by splitting all the screenshot into different CI jobs (we have a *very* modularized codebase so we can split them by modules) so we can take them in parallel and we're considering using multiple simulators in parallel too to speed it even further.

IMHO having these be post-merge might result in other issues. e.g. people ignoring screenshots, merging changes that you're not aware are causing regressions, etc. This is why we ended with this middle ground where we have a CI job that prevents people from merging until they accept the changes, which has been quite well for us.

Write a reply

lachudra87 on Mar 4, 2021 edited

Comentário original em inglês - Traduzir para português

At Circle K (Android) we develop many apps and approach to tests was changing over time. So our legacy projects are a little bit less "tested" and requires more manual tests before releases. But general approach in projects is:

- **UnitTests** - run on each PR on our CI, hard to give numbers but coverage varies from 20% up to 80% in some newer projects. We use junit tests with roboelectric for some android related code
- **UITests** - compiled on each PR, run nightly each day, covers UI behavior same as business use cases. We're fans of UI testing so we've got a lot of tests written with robot pattern. And we use barista as abstraction layer over espresso
- **Appium tests** - testers create and maintain their own test suites, mostly to reduce time of regression before releases. So we've some kind of double check in automatic tests
- **Manual testing** - some processes requires manual testing, especially for e2e tests

1 2 0 replies

Write a reply

alexs555 on Mar 5, 2021 edited

Comentário original em inglês - Traduzir para português

at Avito (ios)

1. **Unit tests**, we have only 2000 of them, but it's okay for us, we don't have much logic in our app, we tend to move all business logic to backend, while app's job is to parse network responses and visualize them. We focus heavily on low-level UI testing and only cover some obvious cases with unit tests like parsing, something that is easier to cover with unit tests or is better to cover with unit tests.

2. **Component tests.** These are low-level gray box UI tests that test mainly 1 screen (rarely 2 or more screens) with all network and other external services stubbed, they are quite stable and quick. These tests are also compact, easier to write and debug and cover more cases compared to black box UI tests, which usually test a scenario, even with multiple parameters, but not every possible case in the scenario. Our tools play the main role here, they give us possibility to write and run these tests. We have just over 1000 of component tests (this number is growing steadily) that we run on every pull request as a blocking build. The build lasts for approximately 15 minutes.

3. **E2E UI tests.** These are high-level black box UI tests. For a long period of time we relied heavily on these tests and had a bunch of tips & tricks on how to run them as a blocking build on each pull request, how to fight with flakiness. But several months ago we decided to enforce Component test culture in Avito. So now, we run all E2E tests every night to ensure that all business scenarios still work well. Also we run them before every release (we have weekly releases). Now we have ~750 e2e tests, which we run on 4 simulators with different iOS versions and screen sizes, so we run ~2500 e2e every night and on each release. It takes ~40 minutes to run them all, that is also due to our tools, I'm going to describe them below.

4. **Manual testing.** We have a weekly release cycle, and quite strict release policy. Every monday we do a branch cut and feature freeze, and manual testers have 1 day to make regress testing. Because of many automated e2e tests mentioned above, we have only a few manual not-automated scenarios that have to be checked.

5. **Tools.** All magic is here. We have our own OSS testing framework named Mixbox that enables us to write e2e and component tests. It is built on top of XCUI and provides much wider functionality. We run our tests with our OSS test runner Emcee, it can run all tests in parallel using all 75 Mac minis that we self host and support in our datacenter. We launch from 2 to 4 simulators on each machine, so around 150 tests can be executed simultaneously. This allows us to run many tests on every pull request as a blocking build. Usually we run around ~150k tests per day on our CI.

We would be glad to encourage you to try out our tools to extend your testing strategies beyond the traditional ways of doing them.

6 12 16 0 replies

Write a reply

pedrovgs on Mar 10, 2021 edited

Comentário original em inglês - Traduzir para português

Hi all! I'm Pedro Gómez a Spanish Software Engineer working from Spain for a company named Karumi 😃

I can't share the specific numbers of our latest projects because I work on a software development studio where we develop different apps and I can't be counting all the tests of the different projects were I've been working on for Android and iOS. However, I can share our testing approach and tools in case it is useful for you.

🛠️ **Android Tools**

- JUnit.
- MockK or Mockito.
- MockWebServer.
- Robolectric.
- Espresso.

- Compose UI testing.
- Shot for screenshot testing.
- KotlinSnapshot.
- Property-based testing with Kotlin test.
- Hilt as our dependency injector.
- GitHub actions for CI & CD.

🛠️ **iOS Tools**

- XCTest.
- Nimble.
- Cuckoo.
- OHHTTPStubs.
- KIF.
- Swift snapshot testing for screenshot testing.
- Property-based testing with SwiftTest.
- Service locators to replace dependency injectors.
- GitHub actions for CI & CD.

🧪 **Testing approach**

Our team splits our app into different layers in order to use different testing strategies:

- **UI Layer:** From the Activity/Fragment/ViewController or any view to the facade of the business logic. Tested using screenshot testing. Interacting with the UI using Espresso o the Compose UI testing tool. The facade of the domain is replaced with test doubles thanks to our mocking libraries and the usage of a service locator or Hilt as a dependency injector. This is the most important detail when talking about how to reduce the scope of our tests. Screenshots are taken with Swift Snapshot Testing or Shot (a library we created that is open source). When possible, we use robolectric to speed up the execution of our test. However, Jetpack Compose code or screenhsot tests can't use robolectric.

We do not add unit tests for our view models or presenters when using MVP or MVVM. That's an implementation detail of our UI layer. This UI testing strategy let us change any implementation detail in our UI layer and we think it is very convenient. We only add a "unit" or "integration" test to the view model or the presenter when there is a specific detail we want to test we can't cover with the already mentioned strategy.

- **Domain Layer:** From the facade of the business logic to the ports specification. Plain old JUnit and XCTest coverage using test doubles to provide implementations for our ports. Depending on the code to cover, we use
- **Adapters Layer:** Storage related code is tested using robolectric on Android and XCTest on iOS. When testing the HTTP layer, we use OHHTTPStubs and MockWebServer to ensure our part of the contract is implemented properly.
- **End to end tests:** When we have access to a QA team, we help them to implement end-to-end scenarios. But these tests are written by the QA team and not the development team.

Sometimes, when we start working on an already implemented app there is no place for unit tests because the code is not testable at all so we use XCTest and Espresso + screenshots and OHTTPStubs or MockWebServer instead of plain old mocks. This allows us to add coverage to the feature before modifying any part of the existing code.

🍵 **Testing pyramid?**

In our apps, most of the business logic is implemented on server-side. That's why our domain is mostly a bunch of lines of code requesting data from the network layer and returning this information to the UI. However, our UI is complex and contains a lot of details we should check from our automated tests. That's why we think the testing pyramid is the best model for our needs. The amount of tests we write per feature is equivalent to the number of different states we have on the UI layer, the number of decisions we make on the domain layer, and the way we request and save data from different datasources. More than a pyramid, we would be talking about a cuboid or a cylinder.

😟 **Pain points**

The main pain points we are facing right now are:

- Deploying apps for the QA team in order to let them manually test any feature without building the branch is really slow. Release builds for iOS are really slow compared to the Android build.
- Android emulators on GitHub actions or any other CI as a service are slow and not reliable at all. Having a frozen emulator while running our tests was quite common until we found there are some emulators working better than others depending on the CI platform. We are using GitHub actions now, but for the last years, we've been using Bitrise, Travis, and Circle CI.
- Compose doesn't work with robolectric and this makes us write a lot of instrumentation tests that could be replaced with robolectric ones. This is really handy when testing navigation between screens. Since we started using Jetpack Compose we can't use this strategy anymore and we need to use instrumentation tests.
- We always need to keep in mind screenshot tests should take animations and async code into account. This is not always easy for some developers and the APIs to synchronize the production app from the tests code is not always trivial to implement.
- Compose screenshots depend on hardware rendering and generates screenshots with just a bunch of different pixels. This forced us to build a tolerance feature in our screenshot testing tool.
- We miss a nice API for parametric tests on XCTest and JUnit a lot.
- Compose doesn't let use render 2 components and take 2 screenshots from a test. To do this we need to start another activity where we can place our composable component and then take the screenshot. This makes our test suite really slow because we depend on the activity lifecycle to write our tests.
- Property-based testing may be really difficult to use for some developers and we only use it for some specific scenarios if the team has some experience with it. If the team has no experience we provide some training when we have time so they can learn how to do it.
- Updating the device we use for testing forces us to record all the screenshots again just because the antialiasing strategy used on the newest iOS version is different or something like that xD

⭐ **Wish list**

We are a small studio, most of the QA teams in some of the companies mentioned here are bigger than our team so there are some nice to have I'd like to implement in future projects if we have resources available:

- Use mutation testing 😃
- Run our test suite on a cluster full of devices. We already provide support for this feature, but we don't have resources to create the cluster right now.
- Adding screenshot tests using a device matrix. We only use 1 device as a reference for our screenshots but it could be great if we could add more devices to our test suite.
- Add tests related to accessibility. We respect the basic accessibility rules, but I'd love to add coverage to ensure the app is implemented properly from the accessibility viewpoint.
- Stop using GitHub actions and migrate to a custom Jenkins with custom resources.
- Add coverage for different device configurations like languages, dark mode, etc.

➕ **Extra resources**

For years, we've been working on open-source repositories we use in our training to show the way we write every type of test using different testing strategies. This is a list of some of the most interesting ones:

- Testing kata about screenshot testing for Android.
- Testing kata about screenshot testing for iOS.
- Testing kata about UI testing for Android.
- Testing kata about UI testing for iOS.
- Property-based testing for Android.
- Property-based testing for iOS.
- Stubbing HTTP for Android.
- Stubbing HTTP for iOS.

JoeSSS on Mar 13, 2021 edited ···

At XING we do a lot of unit, Espresso, Snapshot, but also a lot of e2e tests. As was noted several times in this thread, achieving a working e2e suite is quite hard, but we did not give up :) We have around 3000 tests on both iOS and Android and they run partially in each pull request + fully on a daily basis. We had to build a lot of tooling around it to measure instability, parallel the test runs in a way that each chunk has equal size, "smart" re-run strategies, where when developers need to re-run tests in the PR it will re-run only failing from the past run if the commit did not change. All the measures we took, help us to have a more or less healthy suite but we still trying to improve the monitoring that should say our developers and testers of what is broken and why faster, so they can react and unblock their PRs quicker.

↑ 1 ☺ 👍 1 ❤ 2 0 replies

Write a reply

---

yousefhamza on Mar 18, 2021

Comentário original em inglês - Traduzir para português

Instabug has:

1. ~2000 unit tests, cover mainly the very edge cases that would be infeasible to write them as UI cases. (2016-2018) we have used Kiwi but it has made it too easy to write ineffective tests (too much stubbing) so we have been working on removing it (90% done as of now) and the quality of our tests definitely got much higher. Another part that we are still struggling with some tests is flakiness and we are using test-center to mitigate this for now and not block our PRs for too long.
2. ~200 UI tests, these are usually like "smoke tests" they cover all of the important scenarios we have in the SDK and any support issues we had that we can write a UI test for to make sure it never happens again. Recently we have started using SBTUITestTunnel for asserting on network requests in UI tests and our experience is positive so far.
3. ~20 integration tests, these are unit tests that focus on testing that data are being processed and *sent* to the API properly, it has a mocked API that we can retrieve the data that has been sent it for the assertion
4. We have been working on an internal tool for randomized tests, where we give it some actions and checks, then it mixes and matches them and runs for a set period of time.

↑ 10 ☺ 👍 12 ❤ 12 🚀 8 👀 7 0 replies

Write a reply

---

sachinvas-halodoc on May 6, 2021 edited

Comentário original em inglês - Traduzir para português

Halodoc

1. ~ 4000 Unit Test cases and we measure region coverage (iOS), and branch coverage (Android). We have set up our own Jenkin pipeline which would report the region coverage (iOS) or branch coverage (Android) and upload the metrics to Kibana and break the builds if it doesn't meet the expectation of 98% of testable code. We do omit UI code. To calculate UT metrics we rely on jacoco for Android and our own custom script for iOS.
2. ~ 400 test cases in our automation pipeline consisting of UI tests. We have distributed codebase and UI tests mainly focus on testing the specific functionalities which are classified into multiple SDKs. This is mainly owned by our SDETs/QA team, updating it regularly as the new features get released and maintaining it.
3. AWS Device Farm, provides the ability to run UI tests on real devices with different configurations and OS versions.
4. Manual Testing for newer features and failed UI test cases along with reporting the NFR metrics like AppLaunch, Size, Network, and Memory usage as part of our release checklist.
5. We have automated the pen-testing framework capable of performing static, dynamic(Android only), and malware analysis for iOS & Android using MobSF.
6. We are exploring the benefits of Kotlin Multi-Platform in our UI tests on Android and iOS. Creating common SDK between the platform and providing the flexibility of extending it in platform-specific UI testing.

↑ 2 ☺ 👍 16 🎉 3 🚀 7 1 reply

piannaf on May 6, 2021

Comentário original em inglês - Traduzir para português

Very interested in learning more about 6 since KMP usage generally shies away from UI. It's definitely useful to scale unit (and some integration) testing and reduce the risk of test-case divergence across platforms. What's your approach to leveraging KMP for UI testing?

☺

Write a reply

---

swiftyfinch on Jun 8, 2021

Comentário original em inglês - Traduzir para português

Hi! I wonder how are you saving snapshots? We used `git-lfs` with **Bitbucket**, but now we are moving to **GitLab** and looking for another solution. Any suggestions?

↑ 1 ☺ 5 replies

keith on Jun 8, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

We decided to use S3 for this instead so we upload and download from there

☺

swiftyfinch on Jun 8, 2021

Comentário original em inglês - Traduzir para português

Thanks, **@keith**! Do you use some utilities for this? How do you compare the old and new snapshots visually?

☺

keith on Jun 8, 2021 Maintainer Author

Comentário original em inglês - Traduzir para português

@codeman9 could probably provide more details, but for iOS we pull the image diff from the xcresult bundle and show that

☺ 👍 1

fdiaz on Jun 15, 2021

Comentário original em inglês - Traduzir para português

We also use S3. We use Happo to compare the screenshots. **@thedrick** gave a talk that goes into more detail here (starts at 16:30).

☺ 👍 2

swiftyfinch on Jun 15, 2021

Comentário original em inglês - Traduzir para português

@fdiaz Thanks for sharing!

Write a reply

JPinlac on Oct 18, 2021

Comentário original em inglês - Traduzir para português

It seems like getting UITests to be effective at scale is quite the challenge. Can anybody speak to their physical device device testing strategy? Have you found it useful to execute UITests over a representative sample of physical devices or does the simulator suffice?

↑ 1

1 reply

keith on Oct 19, 2021 **Maintainer** **Author**

Comentário original em inglês - Traduzir para português

We've considered physical device testing a bit for iOS but have punted on it since we don't think for our use case we'll find meaningfully different issues.

On android we use firebase test lab for it to get a better spread of devices to test on

Write a reply

haocuisc on Nov 16, 2021

Comentário original em inglês - Traduzir para português

At Snap/Snapchat:

We have evolved quite a bit in our testing strategies over the years:

Around 2014-2017, we invested heavily in UI tests as a way to provide coverage that's close to the customer experience. We adopted a platform-agnostic solution with Calabash to cover all the UI testing needs with the goal of having test owners write the test once, and it'll work on both iOS and Android. This tool was mostly adopted by QA engineers to write and expand the tests, but as the feature sets on Snapchat grow with the engineering workforce, we faced scalability challenges, especially when iOS and Android feature sets diverged (some features shipped on iOS first, vice versa), maintainability/onboarding new engineers became a problem as well since calabash is written in Ruby and is not native to the development environment.

In the recent years (2018 onwards), we've shifted from Calabash to native frameworks (XCUITest for iOS and Espresso/UIAutomator for Android), as we gradually shift quality/test ownership from QA to Devs. This change really helped to exponentially scale the number of contributors and test writers. We've also beefed up our internal device lab to support all the internal testing needs on real devices. However, the investment here is still very top heavy (focusing on end-to-end UI tests), and getting coverage as early as possible (in the CI pipeline), and we've continuously battled against unreliable/flaky tests in the system as well as keeping CI pipeline fast and reliable.

To combat these challenges, in 2020, we started a program to re-shape our testing strategy to be more of like Test Pyramid shape than a Test Hourglass (yes, we're very top heavy currently), here's the rough breakdown for both iOS and Android code bases:

1. Unit Test - we have some coverage here (~20k+ tests), more on Android than iOS. Few challenges around this area are getting teams to think about writing unit testable code, along with unit tests, as part of the feature development process. Retrofitting coverage, especially around legacy code, is very challenging and risky. We treat code coverage % as a guidance rather than a hard goal, and what is NOT covered is more interesting than what is covered.
2. Snapshot/Screenshot tests (~50 tests) - we are just starting to invest in this space and don't have much coverage here at all. Most of the coverage are in the e2e tests. We're hoping to convert some of the existing e2e tests (that are testing more on the unit-level UI code) to be covered in this layer.
3. Hosted app/Feature app testing (~600 tests) - as we evolve the Snapchat app and modularize the code base to be more isolated and managed by different feature teams, we are gradually unlocking this layer of testing where we have a simple lightweight app that boots up the feature modules isolated from the other feature dependencies. We're just starting to ramp up coverage in this area as well to reduce our dependency against the e2e tests. This also allows feature teams to implement their features directly on the feature app and iterate faster.
4. E2E tests - as mentioned earlier, majority of our test coverage is in this layer (~6k tests). These tests mostly run on real devices in our in house device lab. We are currently looking at sharding some of the executions to simulators/emulators, and we're looking into ways to reduce the number of tests in this layer through educating teams and providing the framework support on alternative ways to get these coverage (items 1-3 above)
5. Manual tests - We have a team of QA engineers tackling the manual tests. There are some efforts here as well to reduce the number of manual tests through automation.

Our goal is to detect bugs as early as possible and block these bugs from merging into main branch. This puts a lot of pressure on making sure the CI pipeline has both the right amount of coverage & is able to execute fast.

Happy to discuss more and learn from you all on the best practices around testing strategies 😃

↑ 3 👍 6

6 replies

Show 1 previous reply

piannaf on Nov 17, 2021

Comentário original em inglês - Traduzir para português

Great write-up! Thank you

> especially when iOS and Android feature sets diverged (some features shipped on iOS first, vice versa)

Is this something product prioritizes deliberately, or would product like the features on both platforms at the same time but a side effect of team/architecture/platform/process issues make it faster to ship the feature on one platform vs the other?

haocuisc on Nov 18, 2021 edited

Comentário original em inglês - Traduzir para português

> ~6k e2e tests sounds great! What tool do use to shard these tests over apple machines? Would you consider to try out our open source test runner - Emceem that does this task perfectly? (we support only simulators after research that investments on supporting the farm of real devices are bigger than profit for us) In Avito we have several thousands of UI tests that are sharded over 70 macminies.

We built some in-house tool to shard all UI tests. Tests are executed fully in parallel across all available devices/simulators, there's thousands of simulators and hundreds of real devices for iOS, and ~1k+ real devices for Android.
Thanks for the pointer for https://github.com/avito-tech/Emcee, i'll check it out!

haocuisc on Nov 18, 2021

Comentário original em inglês - Traduzir para português

> Great write-up! Thank you
>
> > especially when iOS and Android feature sets diverged (some features shipped on iOS first, vice versa)
>
> Is this something product prioritizes deliberately, or would product like the features on both platforms at the same time but a side effect of team/architecture/platform/process issues make it faster to ship the feature on one platform vs the other?

Yes, in the early days we tried to keep everything in sync, but there are some architecture/platform differences, such as hardware dependnecy differences for iOS and Android, IAP differences, etc. that cause divergence in features over time.

2

alexs555 on Nov 18, 2021 edited

Comentário original em inglês - Traduzir para português

> We built some in-house tool to shard all UI tests. there's thousands of simulators and hundreds of real devices for iOS

How many mac computers do you use? Is this a single solution both for iOS and Android? This tool definetly is worth writing a medium article)

haocuisc on Nov 18, 2021

Comentário original em inglês - Traduzir para português

> We built some in-house tool to shard all UI tests. there's thousands of simulators and hundreds of real devices for iOS
>
> How many mac computers do you use? Is this a single solution both for iOS and Android? This tool definetly is worth writing a medium article)

I don't know have the numbers, but we have a separate team maintaining these physical hardware - we share a pool of mac machines with build team, and android stuff are mainly on linux boxes.

Write a reply

raphaelguye on Nov 26, 2021

Comentário original em inglês - Traduzir para português

I would like to open a discussion related to e2e tests.

On which environment are you running them? Closely to the production environment would be the best, but is it an environment dedicated for testing or is it an environment also used by developers, stakeholder, etc..?

If you are running e2e tests on an existing environment (let's say preprod or dev), how can you ensure that you will not break that environment or making it unusable due to performance issues for example?

Let's say you have a test in charge of creating a new item. The assertion will be to see it in the catalog : Will you remove the item at the end of the test? Otherwise your database will just increase every time the tests are run.

4 4

1 reply

alexs555 on Dec 1, 2022 edited

Comentário original em inglês - Traduzir para português

Hi! I'd like to describe how we do it in **@avito-tech** - the biggest classified service in the world.

Several prerequisites:

1. We have an environment, that we call - staging. Actually it is real copy of prod (every backend microservice when is deployed on prod is deployed in staging simultaneously). Every night all databases (we follow the rule: one microservice -1 db on backend) are wiped, but several of them are created with some anonymized data (small footprint from prod) in order to make possible tests for searching items)
2. We have special tool - we call it "resource manager" - it is an api service that can create almost any resources for making possible to create hermetic tests (i.e - in the beginning of every test we create a user in a special state, an item in a special state, so on). At night all these extra items will be wiped by schedule.

back to mobile:

At iOS, We have 600+ e2e tests, that are running over this staging env every night over three lates ios versions supported. We have 100+ mac minies and a super tool Emcee - that we created for our needs, but now that we offer for other companies, that parallizes all these 1800 tests over 100 macminies with 3-5 iOS simulators launched on each of them, making it possible to run all nightly tests during an hour or so instead of days if we were using default Apple tools(only single machine with multiple simulators)

and, finally, we use this suite of e2e tests before every weekly mobile release (we call this stage - regression testing) and every developer has an opportunity to run all suite over his branch with significant changes

2 1

Write a reply

sversov on Jan 10 edited

Comentário original em inglês - Traduzir para português

@ NatWest BanklineMobile (IOS)

1. We have ~4k unit tests, run for each PR as part of PR Checker pipeline. Blocking PR if test are not green.
2. ~1k of Snapshot test. We using Snapshot tests for all small reusable components, and also for Dynamic Font accessibility testing. Running for every PR as part of PR Checker pipeline. Blocking PR if test are not green.
3. ~100 UI tests. Using native XCUITest, all tests are written by developers. UI Tests executed nightly. Broken UI test fixed by those who's PR broke the tests. Flaky UI tests are retired 4 times before reporting failure status.
4. We also have Appium regression pack that is maintained by QA team, and run ad-hock and as part of release train.

0

0 replies

Write a reply

Write Preview

Write a comment

Attach files by dragging & dropping, selecting or pasting them.

Comment

Remember, contributions to this repository should follow its code of conduct.