GERGELY OROSZ

A primer on automated mobile testing

Insights for teams looking to tailor their test suites and build apps that just work.

PART OF
ISSUE 18
AUGUST 2021
Mobile

We rely on mobile apps to check the weather, pay our bills, message loved ones, arrange transportation, work, exercise, relax, and so much more. Naturally, we expect them to "just work" on our phones. Anything less is a disappointment at best and a lost customer at worst.

For engineers working together to build an app, ensuring it just works is no mean feat. There are millions of people using thousands of combinations of devices and operating systems every day, and manually testing each use case is all but impossible. We need to automate the process as much as we can.

However, automated testing can be just as challenging as manual validation. It involves complex thinking and codebases, coordinating across sizable mobile engineering teams, and hunting for issues across a staggering number of variables. It can be similarly daunting to determine which automated tests are most worthwhile. Here, we'll dig into the types of automated mobile tests many large teams use, the challenges you'll face at scale, and considerations for creating an automated testing strategy.

ALSO IN THIS ISSUE

The building blocks of scale

A discussion of the fundamentals of flexible, extensible mobile teams.

Unit tests

Unit tests exercise an isolated component, the unit. Most mobile unit tests exercise the behavior of a class or method. They're the workhorses of automated tests, the simplest and the easiest to understand and write, and they're a dependable testing method used across the software engineering stack, not just in mobile development.

The most important characteristics of unit tests are:

- **Speed:** They execute quickly, require minimal setup, and use lightweight mocks.
- **Size:** They're atomic and test as little as possible, which keeps them fast and makes them easy to debug.
- **Reliability:** They're deterministic, reliable, and don't depend on local configuration or regional settings to produce consistent results.

At scale, large apps can have thousands of unit tests. A discussion on the Mobile Native Foundation GitHub forum started in March 2021 revealed that Shopify on Android has over 8,000 unit tests, Airbnb on iOS has close to 10,000, and the Nordstrom iOS app has nearly 30,000. At Uber, where I was an engineering manager and mobile engineer, we unit tested all code that was non-visual business logic—basically, all codebase classes that were not views, presenters, or builders.

As a codebase grows, however, unit testing classes with many dependencies can slow down. Introducing dependency injection to your codebase makes its unit testable class dependencies explicit and the architecture of the application more modular.

Integration tests exercise multiple components, validating how they work together. They can help validate complex pieces of business logic across multiple classes, and you can use the same frameworks you employ for unit testing on iOS and Android to write them. For example, if your app has functionality to add a credit card, which utilizes different classes for validating the credit card number format and the expiry date, an integration test could exercise them in tandem. Integration testing usually results in more verbose tests, since they have to set up more components up front. However, many integration tests can be replaced with multiple unit tests, which are usually more atomic and easier to maintain.

Ultimately, the larger the codebase, the more teams stand to benefit from unit testing. These tests help validate code changes and force you to separate concerns while reducing accidental regressions and serving as a safety net when refactoring parts of the codebase. It's table stakes for any complex mobile codebase, allowing engineering teams to iterate quickly, keeping regressions low and confidence high.

UI tests

UI tests are the closest automated simulation to a user performing a set of actions on an app. This makes them particularly powerful, offering valuable insight into how people actually interface with a developer's hard work. However, UI tests are also the slowest and most fragile type of test, as the tooling still lags behind platforms like web apps in terms of reliability and performance.

In a UI test, the app is spun up on a simulator or a device, which can then be tested and validated via UI automation. On iOS, Apple provides UI testing out of the box, which involves a manual process to record and replay the user's actions. This works fine when you have few test cases, but for apps with more than a dozen test cases, or when engineers need to collaborate on writing and maintaining UI tests, you may want to engineer a more robust solution. Capital One, for example, uses the Robot Pattern (which involves structuring tests with "testing robots") to test its flagship app, while Wantedly's team utilizes the Page Object Pattern approach (which entails using page objects for better test readability). Android, meanwhile, natively supports UI testing with Espresso and the UI Automator frameworks, which are both powerful solutions that generally scale well for larger apps.

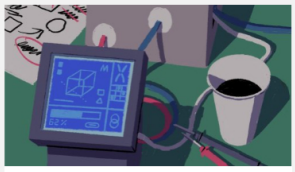
UI tests tend to be more prone to failure and more difficult to debug than other types of tests. When a UI test fails, it can be hard to determine whether the root cause is an issue with the tool, the test, or a nondeterministic race condition such as network and touch events firing close together. All this makes UI tests the most expensive to maintain.

End-to-end tests are UI tests that communicate with server endpoints. When you're not mocking any of the network responses, you're simulating the app by approximating a user's experience on their phone. One of the biggest downsides is added latency waiting for the network, like having the server return 4xx or 5xx responses or error messages. Plus, some scenarios can be difficult to simulate—ones that require multiple steps to put the user account into a rare state, for example, or to trigger an edge case. Some teams opt for UI tests that aren't fully end-to-end but simulate the network layer instead. (When mocking the networking layer for UI tests, you'll have to implement the network mocking yourself or use tools like Mocker on iOS or OkHttp's MockWebServer functionality on Android.) This can speed up the tests and make testing edge cases with special network responses easier.

Many larger teams are hesitant to use too many UI or end-to-end tests because of their spotty reliability. They're also time-consuming to write, and because they're slow to run, some teams don't execute them on each pull request. Plus, as the number of tests increases, so does the time required to execute them. It's unusual for companies to run more than 10 to 30 UI tests before a merge. At Uber, for instance, we had only a handful of UI tests for many years. Similarly, Airbnb, Robinhood, and Shopify have invested heavily in unit and screenshot testing, but have only a small number of UI tests (less than 20). Spotify, XING, and Avito, in contrast, each run more than 500 before merging changes into main.

> A stable and relatively quick-to-execute UI test suite is a major competitive advantage.

Despite the challenges, a stable and relatively quick-to-execute UI test suite is a major competitive advantage because it comes closest to simulating people actually using your app. Try prototyping UI tooling and creating your own if you can afford to, as Avito did with its Mixbox framework. Just don't underestimate the engineering effort required to build and maintain a UI testing suite. Evaluate the balance between cost and benefit, and if you're putting in more than you're getting out, consider cutting your losses and investing instead in a more robust beta testing, phased rollout, and monitoring and alerting process.



READ MORE IN ISSUE 10

Testing

This issue looks at how software testing impacts our collaboration, innovation, and code.

Snapshot tests

Snapshot tests compare a screen's layout or a UI element with a reference image of the expected result. They're a valuable, and increasingly popular, middle ground between unit and UI tests. They're easy to write, run faster than UI tests, and are very visual to debug, since you just need to compare two images. These tests can help catch visual regressions in the development life cycle, which isn't possible with unit or UI tests. To create and run snapshot tests, you'll need to use a snapshot testing framework such as iOSSnapshotTestCase or SwiftSnapshotTesting on iOS or Screenshot Tests for Android, Shot, KotlinSnapshot, or Testify on Android.

Airbnb is a heavy user of snapshot tests—its iOS app is exercised with close to 30,000 such tests, compared to about 10,000 unit tests. The company found these tests particularly useful when adding support for Apple's Dynamic Type feature, which scales fonts automatically. Developers were able to generate a series of snapshot tests, then fix layout issues they observed in these snapshots.

At Uber, we made extensive use of snapshot tests on iOS, and so do apps like Shopify and XING. I was skeptical of snapshot testing before joining Uber, but I've since seen teams spot small visual regressions before they made it to production and use the approach successfully for localization testing. It's worth a try, particularly if first-class UI and UX is of paramount importance to your product.

Automated testing at scale

Selecting the right testing strategy for your organization can seem like an intimidating task. In truth, there's no one "correct" answer—every team must weigh the benefits and trade-offs and opt for a sensible balance, taking into account factors like the app's use case, team setup, and technical and business constraints. To identify the approach that works best for your team

technical and business constraints. To identify the approach that works best for your team, consider experimenting with the tooling available for screenshot and UI testing and prototyping a few tests before committing to one path.

If you get to the scale of having thousands of unit or screenshot tests, you may face issues with tooling, test stability, and lengthy test execution. Seek inspiration in approaches that have worked for companies at similar scale. For example, Avito maintains more than 700 end-to-end UI tests for iOS, and it built and open-sourced its own end-to-end UI testing framework to do so. Along the way, it found it didn't need to invest as heavily in unit testing as teams without an extensive UI test suite.

The more tests you have, the more sluggish your test suite will become. Speeding it up can bring great rewards as your organization scales, assuming there are obvious slowdowns you can eliminate, such as mocking dependencies and long-running operations not core to the test. Profiling your tests and identifying ways to cut the runtime is a useful first step toward speeding up your test suite. Splitting up your suite to run on parallel agents is another valuable approach. While theoretically simple, this can be tough to do in practice. Implementing parallelization of tests from scratch is a complex effort, so you'll likely benefit from using frameworks like Flank or Spoon that help accelerate test velocity.

If you work on a codebase with more than 20 mobile engineers making daily changes, you may need to optimize further and create more custom solutions. At Uber, we had over 100 iOS and Android engineers contributing to the same codebase. The mobile developer experience team built a submit queue that distilled builds into parallel parts in an effort to accelerate the average build time while keeping the main branch green. The submit queue did probabilistic modeling on the build queue, determining and prioritizing changes that were likely to pass. This solution was complex, but it allowed Uber to keep the main branch green at scale.

Does investing time and resources into automated testing guarantee quality and app stability? Not if done in isolation. (Sorry to burst your bubble.) You'll still need to invest in some manual testing, exploratory testing, and a beta program, as well as monitoring app metrics in production. But a strong, thoughtful approach to test automation will enable you to move fast without breaking too many things. Choose the automation strategy that best suits your needs and balances the benefits and challenges, and you'll be able to scale your app and your team with greater speed and certainty.

ABOUT THE AUTHOR

Gergely Orosz is a former engineering manager and mobile engineer at Uber and the author of the books *Building Mobile Apps at Scale* and *Growing as a Mobile Engineer*.

@GergelyOrosz

TOPICS

Guides & Best Practices

Scaling & Growth

CONTINUE READING

10 Testing

MYRA AWODEY AND KARIN TSAI

The process: Launching Duolingo's Arabic language course

How the language-learning app used staggered releases, dogfooding, and a culture of A/B testing everything to ship its first Arabic course for English speakers.

18 Mobile

KAMILAH TAYLOR

Ready, set, multi-platform

Tips for picking your stack, shoring up your foundations, and scaling with grace.

18 Mobile

INCREMENT STAFF

Mobile development at scale

Engineering leaders at adidas Runtastic, Eventbrite, and Citymapper discuss app performance, how mobile fits into their org structures, and native versus cross-platform development.

POOJA BHAUMIK

Ask an expert: How can mobile teams best use feature flags?

CreatorStack's Pooja Bhaumik shares how to reap the benefits—without the codebase bloat.

INCREMENT STAFF

On-call at any size

We take a close look at how to make on-call work at any scale, sharing industry best practices that apply to companies at any size, from tiny startups in garages to companies the size of Amazon, Facebook, and Google.

PATRICK MCKENZIE

An engineer's guide to cloud capacity planning

If you're a small company with big dreams for the future, one of the biggest advantages cloud infrastructure providers have over traditional provisioning systems is the flexibility they offer you to adjust the resources your application uses.

8 Internationalization

CHRIS NICCOLI AND LACEY BUTLER

Beyond translation

A look at how Microsoft's Cloud+AI division aligns customer-centered conversations with localization—at scale.

8 Internationalization

ALLIE BROWNE

Making mobile global

This primer builds upon the basics, presenting key considerations on internationalization and localization for mobile developers.

10 Testing

TAMMY BUTOW

Tests from the crypt

Think of chaos engineering as unit testing your monitoring and alerting—or as exorcising your haunted house.

EXPLORE TOPICS

Learn Something New · Scaling & Growth · Ask an Expert · Interviews & Surveys · Guides & Best Practices · Essays & Opinion · Workplace & Culture

ALL ISSUES

ISSUE 19 NOVEMBER 2021 Planning
ISSUE 18 AUGUST 2021 Mobile
ISSUE 17 MAY 2021 Containers
ISSUE 16 FEBRUARY 2021 Reliability
ISSUE 15 NOVEMBER 2020 Remote
ISSUE 14 AUGUST 2020 APIs
ISSUE 13 MAY 2020 Frontend
ISSUE 12 FEBRUARY 2020 Software Architecture
ISSUE 11 NOVEMBER 2019 Teams
ISSUE 10 AUGUST 2019 Testing
ISSUE 9 MAY 2019 Open Source
ISSUE 8 FEBRUARY 2019 Internationalization
ISSUE 7 OCTOBER 2018 Security
ISSUE 6 AUGUST 2018 Documentation
ISSUE 5 APRIL 2018 Programming Languages
ISSUE 4 FEBRUARY 2018 Energy & Environment
ISSUE 3 OCTOBER 2017 Development
ISSUE 2 JULY 2017 Cloud
ISSUE 1 APRIL 2017 On-Call