17 Ways to Scale Selenium Test Automation for the Enterprise

By Matt Heusser for Tricentis

Selenium and its mobile cousin, Appium, are tools of choice for many organizations trying to get started with test automation. The tools are free and open, coming with drivers for most popular programming languages. That means the tests are created as code, likely in the same programming language and version control branch as the production code. Getting started with Selenium is straightforward for programmers. A mid-level programmer can download the libraries, install the application, and have a sample check up in a day or less.

Then the real work begins.

There is a big difference between a) automating login and making sure the page displays "Hello, Matthew" and an account number, and b) creating an automation suite that broadly covers the entire application. Even if an organization commits to this work, the test suite will grow linearly as development progresses. A single team that adds five minutes to the test suite per week will have a two-hour test run at the end of year. Ten teams that add a single minute to the test suite per week will have a four-hour run at the end of the year—and that doesn't account for the build, setting up the servers, and creating the data. Six months into the project, the development group could experience automation delay that actually slows down delivery…and that assumes they have not abandoned the automation effort entirely.[1]

The point at which a test suite becomes expensive and brittle, or the "inflection point" can be hard to identify. One day, the team is getting value from simple Selenium tests … a few months later, the tests are just "something we always do" and are slowing down the project.

"Plain Old Selenium Tests", or POSTs, might be a good fit for small projects. For example, a one-time, one-team digital marketing effort that lasts six months might have manual setup and a single test suite. However, for a larger, multi-team, multi-year effort, Selenium alone is simply not enterprise ready. Fortunately, it can be made so, with technology and process support.

This paper outlines 17 ways to scale Selenium in the enterprise.

The Methods

The classic and straightforward answer to automation delay is to run tests in parallel. Seeing the specter of automation delay, most groups run to running tests in parallel. This makes sense on a superficial level. After all, if you have one hundred automated tests that take ten to twenty minutes each, you can simply run them on a hundred cloud-based computers, and have the entire suite run in twenty minutes. That is a 7575% improvement over the 25 hours it would take to run them one after the other. However, in order to run them at the same time, each of those one hundred tests will need to not mix test data. The company may need to develop expertise in

[1] By automation delay, I mean that the process of waiting for a new build, installing it, running the tests, seeing failures, "greening" the run to account of false failures, and re-running, can delay the release significantly. In some cases, those false failures can erode overall confidence in the test program.

cloud development and even orchestration, as the Selenium tests themselves may need to run on different servers.

All of this leads us to 5 general categories of scaling:

- Setup & Structure
- Process
- Architecture
- Parallelism
- Enterprise Support

For teams to be successful, they will likely need to borrow from every category.

Setup & Structure

Earlier, I mentioned automation delay—that Selenium tests get slower over time. They also impact productivity. "Out of the box," POSTs run on the desktop of the tester. The common pattern is to run them over lunch, or else lose productivity. POSTs also require setup; the tester needs to configure the test server to have the correct version and the right information.

These techniques allow the team to scale up Selenium without slowing it down.

Automated Build and Provisioning

Most IT groups today have a build process that generates a "build"; it might even run some unit tests. Provisioning means creating an actual test server (that is, a web server) for any given build on demand. In some cases, you may also want to create and load a test database. This can also be done through a command-line tool.

Test Data Management

To consider test data management in action, imagine a test on an ecommerce site that searches for parkas. The expect result is "Showing 1-4 of 4 Results", along with specific results. If test is a copy of production, then shortly after the company discontinues a parka, the test will "break." Managing the test data, then, is having a known set of test data that will generate the same web pages, same search results, and so on. One common strategy is to have a database that is large enough to be realistic, but small enough to be quickly imported as part of the setup. This database can live in version control. As the team adds features, programmers can load, or import the test database. That makes the expected results consistent.

Test Account Management

This is similar to test data management. It may be possible to create a separate isolated account on each test run and create information that only that account can see. The alternative, re-using test accounts on the same database, may "miss" things like one-time events in the account setup process that stop working. Having the data

isolated might simplify test data management. Instead of having to re-create the entire database on demand, programmers can "just" load a standard set of information to this account.

Tests Structured in Suites, Tagged by Purpose

If the development group is organized in "teams of teams," then the lowest unit, the individual team, probably has a specific feature or set of features it is responsible for. A change to a feature is likely to break many tests, which need to be corrected before the test suite can run. Instead of checking the code into a master branch to see what fails, run it locally, at least against the tests that are related. To do this, you will want to "tag" tests and run tests with that tags. With a complex tagging structure, it may be possible to run all the tests for a given feature on a given browser, or on a specific platform, such as the iPad. Eventually, if the architecture changes so you can deploy by feature, having these tags will help "break the monolith." Checking them by team will allow earlier, more relevant testing.

Service Virtualization

Sometimes the test infrastructure is too large to simulate. There are too many APIs, too many dependencies. Instead of creating a "test world" that includes test versions of all the subsystems, provision a single subsystem, the mock or stub out the dependencies. More on this in the *Infrastructure* section.

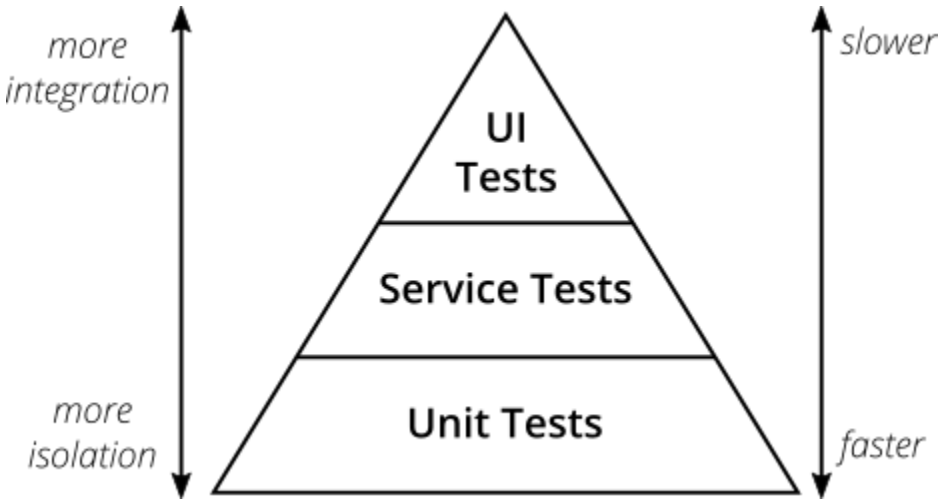
Process

Run different tests by team

Earlier, I mentioned structuring the tests with tags. This is about structuring the work so that each team has a set of tests that they can maintain. Ideally, once the tests run, the team will be able to deploy their own code to their own deploy point. This may require changes in infrastructure.

Focus on the whole pyramid

When the application looks like a black-box, and the role division between test and programming is wide, it can be tempting to start out by testing the GUI. That is where Selenium lives: at the top of the test tooling pyramid. However, GUI tests are also the most brittle tests, the ones with the largest delay and that are the hardest to debug. Not having low-levels tests mean the software will fail more often (meaning more automated tests to cover, increasing the automation delay problem I mentioned earlier). Teams will want to use good code practices and lower-level tests to make sure that very few problems are caught at the GUI level. To do that, teams should measure first time quality.



The test tool pyramid as described by Martin Fowler

Measure and improve first time quality

Look at the number of failures that code changes create, especially in features outside the team. That is, a change in feature A, maintained by team Alpha, breaks the code in feature F, maintained by team Foxtrot. Track them, and do the spade work with retrospectives to improve first time quality.

Mind the maintenance gap

Track how much time is spent "fixing" tests that are only "broken" because the software changed, thus the old expectations for behavior are no longer valid. Find the root causes and reduce them when possible.

All of these process changes are designed to limit and prevent automation delay. In other words, they keep the gap between code checked into version control and tested features as small as possible.

Architecture

Build the complete pipeline first, then add tests

It's tempting to show progress by creating tests first, and doing all the process and architecture work later. Automaton Delay, after all, will take months to catch up, and executives like to watch screens that magically click on their own. Instead, go the other way: Start with one single test, that runs end-to-end, unattended. That is, a build/deploy process that creates a real test server and runs one simple test, either in the cloud or at least on a build machine.

Isolate components

If the entire application is defined as a monolith and the build is slow, then testing will need to be exhaustive, which will mean more time and more tests. However, if each team has one or more deploy points that can deploy in isolation, then only one feature can break at a time. Organizing the tests around these features means a smaller test set to run. These components can correspond to single web pages, APIs, or services like login, tagging, search, creating a profile, and so on. Designing a mobile application as isolated components can be particularly challenging, but the tools to do this are emerging.

Extend tests with a broader tool

It is tempting to build a test system entirely out of open-source tools. After all, Selenium and Jenkins are free, and the build scripts are essential elements of the development process. Once you add service virtualization, components, tagging, provisioning, and reporting, the infrastructure is essentially a software engineering project of its own. Scheduling, running, and reporting tests, along with "adding" tests, turning them on and off and integrating the reporting along with the human test effort, is much more than a traditional continuous integration server was designed for.

Plan to build a tool to coordinate testing, or buy one. This tool will be the "information hub" for the test effort, and squarely fits in the domain of DevOps.

Parallelism

Once the team has the setup, tools, process, and architecture in place, it may be time to run tests in parallel (e.g., run many tests at the same time). This generally requires some dedicated machines to run multiple browsers at the same time. Coordinating the running and reporting of those machines adds a new layer of complexity to the test effort. Here are some of the common solutions I have seen to accelerate testing in parallel while reducing the risk.

Consider running the grid in a public cloud

If the production code is already using a private cloud (e.g., with something like Kubernetes or OpenStack), it might be possible to re-use that tooling to organize an internal grid. Even then, having to run on multiple browsers and operating systems could be challenging. It may be easier to re-use an existing public cloud or cloud offering. Even that can introduce security challenges; build a proof of concept before committing to a solution.

Create a test runner

This is a tool that takes a tag (or combination of them), perhaps a branch or build, and a platform (browser, operating system) and runs the tests, reporting to a single reporting source. Each test run may need a unique ID to report back to. That reporting source could be a database. It is likely the runner and reporting system are part of the larger tool described under the Architecture section.

Consider multiple test web servers

Most of the discussion about running tests in parallel is about running the tests against a single server. In some cases, it will be impossible to separate those tests. One option is to create different test servers, test databases, and test API systems.

Enterprise Support

Visualize and scale reporting and analysis

With an investment like enterprise Selenium, management will ask new questions. For example, they might want to know how long the tests are taking to run, how many are failing, how the failing is changing over time, are the failures the same problem over and over again or new problems? Are we "missing" tests or do we have too many? Dashboards with test runs over time, separated by tag, combined with the ability to drill-in, can help management make decisions on what to invest in. They can also help with debugging, fixing, identifying problems and taking corrective action.

Purchase commercial support or training

Learning to scale Selenium is more than learning to use a code library; it is learning to integrate that code library, along with the patterns that make it scalable. At the very least, develop internal expertise and roll out tooling slowly, expecting to either fund a new testing role, or for development to slow as the team learns how to use a new tool.

A Path Forward: Getting Serious About Scaling

As mentioned before, the main categories of setup, architecture, process and support are intertwined. Rolling out test tooling to a large organization is going to be a learning process, as the obstacles will be continuously evolving. The core question is usually less "what is the roadmap?" and more "what should we do next?". In other words, what is the single most valuable investment for keeping the program going?

After doing this work with hundreds of teams doing informal analysis, I have found some trends. Most organizations will need to invest in an infrastructure/automation role; larger organizations will want a team. This role will analyze the software itself to see how easy it will be to run in parallel, and what pieces described above need to be put in place first. That may include collaborating with Software Reliability/Operations to create monitors. It may mean working with software engineering to break the monolith and build different release endpoints, or to move toward a microservices architecture.

Expect to iterate and revise this strategy. Plan to roll it out to a single team, then grow it with other teams. Compared to what the team builds in two to five years, the original roadmap will likely look as if it is scrawled on the back of a napkin. That is normal. It is okay. It means the team is learning, adapting, and growing.

"The important thing is to get started" may be a common saying. However, many failures can be traced back to lack of setup, architecture, process, and support.

Yes, please, start. You will never get anywhere unless you start.

Just be sure to start *smart.*

About the Author

Managing Consultant at Excelon Development, Matt Heusser is a former board member of the Association for Software Testing and the 2014 CAST Keynote Speaker. The 2014 recipient of the Most Influential Agile Test Professional Personal Award (MIATPPA) award, Matt is also the creator of the Lean Software Delivery family of methods and probably best known for his writing.