The Ultimate Guide to Automation Testing

Tools & Best Practices for Test Automation

Traditionally, during regression testing, a manual tester will take an existing test case procedure and execute it step by step. This can be time-consuming and also error-prone since it is done by hand.

Because of the reasons above, to save time, many companies try to take their manual test cases and convert them to an automated test case. An automated test tool then executes the test steps automatically without human intervention.

Sounds easy, but there are many pitfalls teams encounter when starting their test automation journey.

In this tutoral I will address the most common automation issues folks face and hopefully help you begin your automation project off right.

What Is Automation Testing?

Functional automation testing uses tools designed specifically for automation to emulate a user interacting with an application and verifying test steps using programming assertions.

Many folks also call these tests automation “checks” or automated checking.

This distinction is made to remind testers that automation and the implementation of automation is a checker—it doesn’t replace your testing strategy. An automated test is also, in a sense, dumb in that it can test only what you tell it to test; if you don’t assert it, it doesn’t get checked.

Also, it’s important to remember that “automation” does not apply to just user interface (UI) end-to-end tests. In fact, I would say that you get more benefit from a lower-level automated test, like a unit test, than you do from a big bloated test suite of end-to-end tests.

Before we take a more in-depth look at automated testing, let’s touch on some problems with manual testing.

Issues With Manual Testing

There are many reasons why having a testing strategy that relies heavily on just manual testing causes issues. Here are a few:

Automation testing can help.

But one of the first hurdles you might face when introducing automated testing to your organization is the false belief that automation can replace all your testers and tests with automation.

Here's the deal:

Automation Testing Does Not Replace Testers.

Some people assume because a test activity is automated, that means it replaces human testers. In fact, however, it's the opposite. Automated tests are great for running tests precisely and quickly, but they in no way replace human testers.

Automation tests are also great for running the same steps over and over again, but they don't think.

"*Computers are complements to humans, not substitutes. The most valuable businesses of the coming decades will be built by entrepreneurs who seek to empower people rather than try to make them obsolete.*"

Although we can agree that automation testing does not replace other testing activities, with today's software development environment and continuous integration practices, it is critical and cannot be ignored.

Why Is Automation Testing Necessary?

Practices like continuous integration and delivery require tests that run quickly and reliably. Lots of manual tests will stop your ability to achieve velocity with your software development.

I'd go so far as to say that in today's modern development environment, we couldn't succeed without automation.

Although the main reason teams try to create automated tests is to save the company both time and money, it's also important to give developers quick feedback so that when they check in code, they are alerted as soon as possible that the change they checked in broke something.

Some other reasons for automated testing are:

While these are good reasons for automation, many folks fail to factor in the amount of time and money it takes to maintain sizeable automated test suites.

Automation Testing Considerations

Since automated tests usually rely on programming languages for their creation, automation becomes a full-blown development effort. What you are doing is developing a piece of software to test another piece of software.

Automation testing is difficult and complicated, just like most other development software projects. It also presents many of the same issues other software programs do. Treating your automated code just like your development code is essential. Follow the same processes and best practices you would use for any other software development project.

To learn more, John Sonmez covers many of these best practices in his awesome Pluralsight course Creating an Automated Testing Framework with Selenium.

Automation Testing Pitfalls

Teams often claim that automation testing “doesn’t work.” But this attitude is usually caused by poorly designed test automation more than anything else.

If you keep the issues listed below in mind as you create your test automation framework, you can avoid many of these automation pitfalls ahead of time.

Many issues are caused by setting unrealistic goals, like, for example, having a goal of reaching 100% UI automated testing coverage. Teams often believe that automation tests will find more new defects, so they have a false sense of security. Your automation is only as good as your tests.

Teams also underestimate the amount of time it takes to maintain automation. They’ll often create large, end-to-end tests, but tests should be atomic so that when they fail, you know why.

Several other common issues that teams face are:

Automation Is a Team Effort

For automation to be successful in an organization, you need to educate everyone on the team as to what the expectation should be for your testing. It’s also critical that you create a whole team approach to your automation efforts, meaning regardless of a person’s role on the team, development and testing (and ultimately, delivering a feature) requires a total team effort.

Testing shouldn’t be an activity that is done only at the end of a sprint by a designated tester.

Quality needs to be baked into the software from the beginning, not after the fact. The only way to do this is to have everyone working toward making the application under development as testable as possible. This takes the whole team working together to deliver a quality product.

Many failures I see with automation are not caused by technical issues, but rather by a company’s cultural issues.

Remember—automation is a time-consuming test activity. You want to use it only when it makes sense. Other testing activities, like exploratory-type work, should be encouraged.

Automation is just one of many types of test activities that can be used by testers.

At this point, another common question I’m frequently asked is, “Which tests should be automated?”

Tests That Should Be Automated

The biggest problem I usually see is that teams start off trying to automate everything. The problem is that not everything is automatable. When planning which test cases to automate, you should look for tests that are deterministic, don’t need human interaction, are hard to test manually, and need to run more than once.

You should also seek to automate any manual process that will save engineers time (not necessarily an official “testing” process), along with tests that focus on the money or risk areas of your application.

Other tests that are useful to automate are unit tests, as well as tests that run against different data sets, focus on critical paths of your application, need to run against multiple builds and browsers, and are used for load/stress testing.

The more repetitive the execution is, the better candidate a test is for automation testing. However, every situation is different.

Ultimately, you should consider using automation for any activity that saves your team time. It doesn’t have to be a pure testing activity; you can leverage automation to help reduce any time-consuming activity found anywhere in the software development lifecycle.

At this point, some of you may be asking, “What is the return on investment (ROI) of test automation?”

ROI: The Cost of Test Automation

Determining the ROI of your automation testing efforts can be tricky. Here is a common calculation some folks use to get a rough estimate of their test automation costs. This can also help you decide whether a test case is even worth automating as opposed to testing it manually.

Automation Cost = how much the tools cost + how much the labor costs to create an automated test + how much it costs to maintain the automated tests

Consequently, if your automation cost calculation is lower than the manual execution cost of the test, it’s an indicator that automation is a good choice.

Moreover, ROI quickly adds up with each re-run of your automated test suite.

Because it’s critical that you get a good return on your test automation investment, there are some things you shouldn’t automate.

What Shouldn’t Be Automated?

There are exceptions to everything, of course, but in general, you may not want to automate the following test case scenarios:

In addition to what not to automate, another element of a successful automation project is having an automation framework.

What Is an Automation Testing Framework?

I like to break down an automation testing framework into specific areas of concern, or what I call the “four Ps” of an automation framework: people, planning, process, and performance.

The first one is the people aspect of a test automation framework.

People

As we already covered earlier, you want to make sure that you have set the expectations of your managers and team about your automation strategy. To help ensure that your automation is collaborative and a whole team effort, I recommend that you include automation on your sprint team’s definition of done.

The next stage is the planning piece of your automation framework.

Planning

Before writing one line of code, always check to see if there is an existing library or tool you can use before inventing your own. Break your automation framework into abstraction layers so that if anything changes, you just need to make the change in one place. Using established automation testing design patterns like the ones we cover later in this post should be part of the planning stage.

Separating your tests from your framework will also help when you have to make changes to your framework.

Also, plan on making your methods and utilities reusable to avoid code duplication. Making your test and code as readable as possible (they should read like English) will go a long way to prevent confusion and code duplication.

Finally, when planning your test, it's important to be aware of what test data your tests need. Many times, tests run against different environments that might not have the data you expect, so make sure to have a test data management strategy in place. Including support for mocking and stubbing in your framework can also help with some test data issues.

Process

Having a process in place that holds team members accountable for automation is another vital piece of a framework. Since automation is just like any other development project, make sure to use the same process and best practices that developers already follow, like using version control and performing code reviews on all automated tests.

Performance

Always start your automation framework with the end in mind. Tests need to be reliable and maintainable. They also should run as fast as possible.

Along these lines, make sure that your developers are creating unique IDs for each element that you will have to interact with within your tests. Doing this will help you avoid resorting to lousy automation practices like relying on a coordinate-based way to communicate with an element.

Another top killer of test automation script performance is the failure to use proper synchronization/wait points in your tests. Too many hard-coded waits will slow down your test suite. Use the preferred explicit wait method for synchronization.

As you write your test scripts, think about how they would perform if you had to run in parallel. Thinking about possible parallel issues beforehand will avoid problems when you start to scale the running of your test suite again on a grid or a cloud-based service like Sauce Labs.

To ensure that your tests are as preformatted as possible, refactor slow or poorly written code whenever possible. Including reporting and logging in your framework will help you quickly identify poorly running code.

To make sure that teams follow all these guidelines, determine a strategy for training and retraining your framework users.

That's not all …

Automation Testing Design Patterns

It's a given that your applications are going to change over time. And since you know change is going to happen, you should start off right from the beginning using best practices or design patterns. Doing so will make your automation more repeatable and maintainable.

Here are some common automation testing design patterns that many teams use to help them create more reliable test automation.

Page Objects

One popular strategy to use when creating your test automation is to model the behavior on your application. Creating simple page objects that model the pieces of your software that you are testing against can do this.

So, for example, you would write a page object for login or a page object for a homepage. Following this approach correctly makes use of the single responsibility principle.

If anything changes—say, an element ID—you just need to go to one place to make the change and all of the tests that use the page object will automatically pick up the changes without you doing anything else. The test code needs to be updated in only one place.

Page objects also hide the technical details about HTML fields and CSS classes behind methods that have easy-to-understand names. Being mindful when naming your methods has the extra benefit of helping to create a nice readable test application programming interface (API) that a less technical automation engineer can quickly start using for automation.

Presenter First

Presenter First is a modification of the model-view-controller (MVC) way of organizing code and development behaviors to create completely tested software using a test-driven development (TDD) approach.

He mentioned that if you draw out the MVC pattern as blocks and arrows, you can see that the view, which is your UI, has well-defined channels of communication with the model and the controller. If you can replace those during runtime with models and controllers that your test creates and controls, then there is no reason why you can't just test that the UI behaves in the way that you want.

You can also set your model and controller to mimic all sorts of odd behaviors, like a network going down.

Test Automation Process

Here are some high-level good automation practices you should follow in more detail:

Atomic Test

An atomic test is a strategy for ensuring that each test is entirely self-contained. That is, it should not depend on the outcome of other tests to establish its state, and another test should not affect its success or failure in any way.

Also, when an automated test fails, you need to know why. Having a well-named atomic test that tests only one thing will help you quickly identify what broke if your test fails.

Furthermore, you should endeavor to get feedback to your developers as quickly as possible, and the best way to do that is with a fast, well-named test.

This is also critical if you plan on running your automation test in parallel in a Selenium Grid.

Test Sizing

Test size matters because tests need to run quickly.

At this point, many people visualize a traditional test pyramid, which has unit tests as its base, integration tests in the middle, and graphical user interface (GUI) tests at the top.

But I think more in terms of test size. By test size, I'm referring to tests that are faster than others.

While I understand the need to run UI tests if you have to create one, make it as fast as possible.

Readability

A quick point on test code readability—did you know that developers spend more time reading code than actually writing it?

It is rare that the person who wrote code will also be the one that needs to modify it. Even worse, how many times have you written code only to come back to it months later and have no idea

what it is doing?

Since, as we mentioned, automation code is software development, you should create your test code with the reader of the code in mind—not the computer.

This will help not only to make your test more maintainable, but also will help ensure that you do not duplicate code because you didn't know what an existing piece of code was doing.

This might seem like a minor issue, but ignore readability of your automation test at your peril.

Testability

Testability needs to be baked into our applications right from the start. As a regular part of sprint planning, developers should be thinking about how they can make their application code more testable. They can do this by providing things like unique element IDs for their application fields and APIs to help create hooks into their application(s) that can be used in their automated tests.

They should also be thinking about how any code changes they make to the application are going to impact existing automated tests, and plan accordingly.

If you don't do this, you're not going to be successful with automation for very long.

Remember, you can't automate that which is not testable.

Stable Environment

Without a stable test environment that is always in a known state, it will be tough for your teams to make progress with their automation efforts.

Tests failing due to environmental issues rather than actual application issues will cause your teams to lose confidence in your test feedback quickly.

Once teams start ignoring automation results, your test efforts become useless.

How To Pick an Automation Testing Tool

There is no "correct" test tool for automation testing. Ultimately, it all depends on your team's unique skill set.

I always recommend that you run a two-week proof of concept (POC) for each tool that you are considering and include your team's feedback in the process before committing to a tool.

The first place to start is to look at the product roadmap and make sure the tools you select will handle future features and technologies. To avoid future compatibility issues and tedious framework refactoring due to false notions, don't skip this step.

Next, you should evaluate the cost, including maintenance. If you plan on having your whole team help out with the automation effort, make sure to use a tool that leverages the same tools and languages your developers use.