Growing the Test Suite

"To succeed, planning alone is insufficient. One must improvise as well."

--Isaac Asimov, Foundation Series

Writing tests is fun! This may seem contrary to the first sentence in Chapter 1, *Writing the First Test*, of this book. However, once we solve the difficult problems such as test stability, test data, and the framework design, writing a new test case is the most gratifying experience one can have in our field.

Once we have a good grasp of Selenium and are ready to grow our test suite, we will face some new challenges. Questions such as what test needs to be written next or what CI tool to choose from will naturally come up. In this chapter, we will discuss the following topics about the long-term growth and maintenance of the test suite:

- Strategies for writing test suites
- Different types of tests
- Different types of test suites
- Continuous Integration
- Testing in multiple browsers
- Selenium Grid
- Managing build nodes
- Build node virtualization
- Frequently Asked Questions

Strategies for writing test suites

A common question during a job interview for test automation is "How do you plan to build the test suite?" When I was new to software test automation, I would answer that 99 percent coverage is critical. After reality had a chance to catch up, it became apparent that such high coverage is impossible due to obvious time constraints.

Instead of having 100 percent test coverage, the best that can be done is to prioritize the growth of the test suites. In this section, we will discuss the order in which test suits should be built. As we build our test suites, some tests will cross multiple boundaries, which is perfectly normal. However, it is best to have a way to group certain tests together so they can be executed individually. For example the smoke test suite is a subset of the regression suite, but we need ability to execute it without having to run the regression suite.

NOTE

All of the strategies listed are in order from highest priority to lowest, but they are not mutually exclusive.

Different types of tests

Before we dive into the different Selenium test suites, let's define several types of automated tests. This will help us understand where Selenium tests belong in the development cycle. The following definitions are commonly used to describe a type of an individual test; however, they are slightly redefined with Selenium bias:

- **Unit test**: This is, by definition, the smallest test unit. This type of a test is written to test an individual object and even individual methods of said object. Unit testing is highly important because it prevents bugs from creeping in at the lowest level. These tests rarely use any production-like test data and often solely rely on mock data. Since unit tests are at a low level of the application, Selenium tests are not applicable here.

  ### NOTE

  Low level is a phrase commonly used to describe code that has a low level of abstraction. Likewise, high level describes code with a high level of abstraction. For example, a method that adds 1 and 1 would be described as low level and a method that registers a new user in the database is a high-level method.

- **Integration test**: This consists of several modules of code put together, allowing them to pass data between each other. The purpose of this type of test is to make sure that all modules integrate and work with each other. In terms of Selenium, integration might be checking that the store module of our website can pass the product information into the cart module. The tests that run in CI after every commit to test only our application and stubbing all third-party services is considered integration build.

  ### NOTE

  Integration tests are sometimes referred to as functional tests, since they test the functionality of an application.

- **End-to-end**: This is the highest-level of test. This type of test is executed in production or a production-like environment, such as staging. Similar to integration tests, an end-to-end test tries to verify that all of the components, including third-party services, can communicate well with each other.

  ### NOTE

  End-to-end tests are sometimes referred to as **Verification and Validation**(**V&V**)

The majority of Selenium tests will fall into the integration category. By blocking as much instability caused by test data and third-party dependencies as possible, our tests can concentrate on testing only one piece of functionality at the time. However, a well-written test that is properly hermetically sealed should be able to run in both integration and end-to-end environments.

NOTE

See *Hermetic Test Pattern* in Chapter 3, *Refactoring Tests*, for more information.

The smoke test suite

Smoke testing is a very common and popular concept in the quality assurance world. The idea is to plug in the new code, let it run, and see whether it runs or catches on fire. Out of all the test suites, a smoke suite will by far be the smallest in size, since it needs to give a close to instantaneous pass or fail verdict. This test suite is best used in the first few minutes after new code is deployed to any environment. Use this small test suite to make sure that the production environment is up and running.

Tests in the smoke test suite should look for the following:

- **Running application**: Does the website load or does it give a 500 error? This is by far the simplest question and could be answered by navigating to several key pages, such as the home page or the online store.
- **Database connection**: Database issues happen more often than anyone cares to admit. After the deployment of the new code, we realize that the database was not properly migrated. Test should do several *read only* checks against the database, such as log in with an existing user.
- **Abnormal amount of exception**: This question is a little bit more involving than others. The starting point is to make sure no page returns an error code when it should not. It can evolve into dumping the JavaScript console logs to check whether new JavaScript errors start to appear.

Smoke test suite should almost be like a feather in a boxing fight. It should touch the application without leaving a single dent or scratch. We should keep the following in mind:

- **Avoid writing to a database if you cannot clean it up after**: It is normal to register new users or make purchases on a staging environment. However, this is typically a bad idea in production, since it is difficult to clean up the test data after the test is complete. To stay on the safe side, the test should only perform actions that read from the database, never write to it.
- **Don't test too much**: We want to have an answer about the state of the environment as fast as possible. Leave the more extensive testing to other test suites.

The money path suite

Money path is one or several core key pathway through our application. In the case of an online store, it is the ability to add items to the cart and receive payment information. In an inventory management system, it's the ability to retrieve and update current inventory. Noncritical functionality, such as updating user's email preferences, is to be left out of this test suite.

Money path and smoke test strategies can have multiple tests in common; however, tests that write to the database in the money path suite should probably not be included in the Smoke Test suite.

The money path suite should answer the following question: is the customer prevented from giving us money? This is by far the most important part of this suite. Every single test in this test suite should aim to answer that question, if it's not it should be moved to another test suite.

New feature growth strategy

Smoke test suite and money path suites are the top priority when writing tests. However, those test suites are relatively small and will go into maintenance mode pretty quickly. After they are finished, we will spend the majority of our time in this mode. The idea of the new feature strategy is to keep up with the development of the application. As a new feature is added, we add a new test. This strategy does not try to write tests for an already existing functionality such as regression strategy.

By far, this new feature strategy is most effective when the test writer is embedded in the development team.

TIP

Some of the teams I've personally worked on, the developers themselves were responsible for the creation of new tests as the application got new features. This gave an up-to-date Selenium test support for all new features and gave the QA team a starting point to add and improve the said tests. This setup has been extremely successful.

Being part of the team on a daily basis and seeing the direction of development is an important part in keeping up with new features. The classic over-the-wall approach to quality assurance will not work well because by the time the test developer starts writing tests for newly delivered feature, the development team has moved on to new tasks.

NOTE

The over-the-wall testing approach consists of the Development and QA teams being completely separate. After a new set of features has been added to the application, the new build is given to the QA team to test. The QA members are not involved in daily testing as each commit happens.

When writing tests for new features, the tests should concentrate on answering the following questions:

- Is this the most important and critical feature to be tested?

  There are new features that enhance the application slightly and there are critical features. When pressed for time, as we always are, the new tests should aim to test mission critical features and leave the enhancements to the regression suite.

- Are the new tests useful right away?

  As soon as a new test is written, even for an unfinished feature, it should be added to CI. Features that are in active development have the maximum instances of instability and bugs by far. Having the new tests added in step with new code and running on every commit provides a good foundation for stability.

Bug-driven growth strategy

Bug-driven and new feature strategies are extremely compatible with each other. The new feature strategy concentrates on adding a new test for every new feature. The bug-driven strategy concentrates on adding a new test for every bug discovered and fixed.

Every new release of the application comes with a list of new features or bug fixes, and more often than not, both at the same time. Most people who will be testing the new release of the application will concentrate on testing the new features, while giving the bug fixes a cursory glance. Having an automated test case for every bug fixed in the current build is a great safety net. Furthermore, sometimes when a new release branch is being created in the **Version Control System** (**VCS**), bug fixes are sometimes overwritten or reverted by accident. A single test might prevent an emergency deploy!

TIP

An accidental revert of a bug fix may not be the most common occurrence on a team familiar with their VCS tool; when it does happen, it happens at the worst possible time.

The regression suite

Regression tests are all of the tests in our suite that test features developed in the past; this is not to be confused with features actively being developed. Following the previously-described strategies, we will add new tests to the regression test suite.

Sadly, more often than not, our teams will always be too understaffed to write new tests against already existing features. When pressed for time, we should always concentrate on new features, since this part of the code will prove to be most unstable. As new bugs are discovered in older code and feature set, the regression suite will slowly grow. However, spending too much time writing tests for sections of code that were not touched in years will probably prove to be a waste of time.

The 99 percent coverage suite

A test suite that covers 99 percent of new and existing features in an application is the dream of every tester. However, it often proves to be nothing but a pipe dream. Unless we are on the team writing tests for the space station or a nuclear power plant, we will never have enough resources to test everything. Thus, for most automated test creators, this strategy is not only wasteful but can be extremely harmful.

Any piece of the application that has not been touched in a long time and has not had any bugs in that time is unlikely to randomly start producing bugs. Writing a test for that code may be harmful because it takes time away from writing a test for new code that more than likely will break. However, if the said old code starts to be updated with a bug fix or a new feature, it's a good idea to write tests for it.

On the other hand, if you are in a very fortunate position where you can afford to keep updating the test suite, bringing it closer and closer to full coverage, consider yourself extremely lucky and keep going!

One last thought about adding new tests to our application before we move on, is that it is always better to have a smaller test suite that is reliable and is executed often, than to have a large test suite that fails randomly and makes everyone on the team lose faith in its usefulness.

TIP

As soon as we start to add any test to our suite, it's a good time to start thinking about CI.