Test Automation: Transitions Micro-Credential

Syllabus

Table of Contents

Test Automation - Transitions

Introduction

The purpose of this micro-credential is to describe the steps needed to make the transition from a manual tester to one who understands and can effectively develop test automation.

LEARNING OBJECTIVES FOR TEST AUTOMATION: TRANSITIONS

Introduction

Understanding the Differences Between Manual and Automated Testing
(K2) Summarize the differences between manual and automated testing

Becoming More Technical
(K2) Explain the skills required for a test automation engineer and how those compare to the skills for a manual tester

Gaining Tool Knowledge
(K2) Explain the recommended approach for selecting a test automation tool for learning

Gaining Coding Skills
(K2) Summarize the capabilities and features of the various languages used in test automation

Gaining Knowledge of Platforms and Environments
(K2) Explain how environments and platforms influence tool selection

Testing APIs
(K2) Explain how service virtualization can be used in API testing

Working with Frameworks
(K2) Summarize the capabilities of testing frameworks

Building Reporting Skills
(K2) Explain the types of reporting commonly used for test automation

Understanding Which Features are Good Candidates for Test Automation
(K2) Summarize the characteristics of good candidates for test automation

Understanding Other Things Test Automation Can Do
(K2) Summarize the steps recommended for building test automation knowledge

Understanding the Differences Between Manual and Automated Testing

There are key differences between manual and automated testing. To get started, you must understand the differences to know why certain approaches are needed.

Manual testing is more forgiving, which means that if the test fails, you can explore why and try again. In contrast, sometimes, just a very small change can cause an entire automated test run to fail. In manual testing, you get a chance to simply try the test again.

In manual testing, you can modify the test data and/or conditions "on the fly," which means you can try different variations of the test and make a judgment on whether or not the test passed or failed. If needed, you can re-run the test and verify results with subject matter experts (SMEs) to get their opinion of the test results and whether your manual test needs to be modified.

With an automated test, evaluation starts when the failure has occurred. From that point, the analysis needs to be conducted to determine the cause of the failure.

A common misconception about automated testing is that it can replace manual testers. While test automation can ease the burden of repetitive testing, there will always be a role for manual testing because the tools do not understand all the nuances in an application. In fact, before a test can be automated, it must first be understood and tested manually. For this reason, all test automation engineers must do some amount of manual testing.

Manual testing is quick, can be "fire and forget" which means a manual test can be performed quickly and the outcomes seen immediately. Ideally, such a test would be planned, analyzed, designed, and implemented in a test management tool. As compared to an automated test, a manual test can be created faster. It might take longer, however, to execute than an automated test that has already been created.

Automated tests do exactly what you program them to do. There is no room for judgements to occur. Just as in the software you are testing, an automated test runs as programmed and determines pass/fail based on pre-defined parameters.

When a test fails, you start troubleshooting. In manual testing, a failure can be quickly assessed. Depending on the assessment, a defect report may be written or the test may be adjusted (or the environment or data) and the test is executed again. Failures can occur because a test was executed incorrectly. With manual testing, that's easily corrected with a re-execution. With test automation, failures can also be due to failures in the way the test has been coded, which may cause the test to fail erroneously.

A failure with an automated test may cause the entire test run to fail. In manual testing, you can often work around a failed test and complete other tests that pass. This is not always true in test automation. One failure might cause an entire test run to stop.

Test automation is always particularly prone to failure when changes occur. For example, an object locator may have changed, or a navigation may be altered with an extra mouse click required. This doesn't affect manual testing but is likely to cause an automated test to fail. The object under test may actually be correct, just changed.

Automated testing takes time and technical skills. Test automation is more than just translating a manual test script into code. Some manual tests may be very difficult to code due to the setup needed or the validation required. Some manual tests focus only on the positive path. These "passing tests" (confirmatory) verify that the software works per the requirements, but don't test for how it handles error conditions, invalid data, etc. These types of tests are more suitable as smoke tests than for a rigorous test of the application.

Budgets must be justified. Even when using open-source tools, test automation can be costly. There must be money allocated for building and maintaining the automation, plus training of the people creating and maintaining it. In the short-term, manual testing is almost always less expensive. It's in the multiple executions and relief of tiresome, repetitive work that the value of test automation is realized.

Training is needed. While this is also true of manual testing, test automation has unique training requirements due to the constant march/influx of tools and features in the tools that must be monitored and understood. Training is essential to keep up with the changes, as well as the functionality in tool(s) you are using. Trying to create test automation without the proper training is a recipe for failure.

Sophisticated design is needed to reduce maintenance costs. If tests are simply created with no thought given to sustainability with frameworks, then you will eventually need to deal with a maintenance nightmare. Careful framework design, as discussed in Part 2 of this micro-credential, is essential and designing a good framework requires training and experience.

Updates are needed. As the software under test grows and gets more features, the automation can become stale, and the scope of coverage is reduced. Growing the test automation is much more than just keeping up with maintenance changes. It is easy for the test automation to suffer from growing weaker over time, with all tests passing but no meaningful testing as a result. This happens when the focus of the testing is on the parts of the code that are the most mature and most stable.

Becoming More Technical

All of this means that testers who want to become test automation engineers must become more technical in nature. The skills needed include:

Analytical skills, which are all about studying the items to be tested and identifying what should and should not be tested. In addition, this includes understanding which features should and should not be automated. Technically anything can be automated, but not everything is feasible to automate due to complexity, high level of change, low expected yield, etc. Determining the best use of the limited budget for automation is critically important for the long-term success of an automation program.

In test automation, especially at the outset, prioritization of tests to be automated and executed is needed. This ensures that the automation effort is put into the highest reward areas. Prioritization is based on risk and other criteria, such as expected areas that are likely to change and will need more testing, the need for refactoring of automated tests, risky areas that are hard to test manually, and the feedback from retrospectives regarding ways to improve the test automation. Other criteria for test prioritization include usage criteria, such as which areas of the application are used the most by users, and defect trends, both past and present.

Estimation skills in test automation are very important. This includes estimating:

- development of test automation assets (e.g., framework, test scripts, reporting)
- maintaining test automation assets and environments
- the time needed to execute tests
- the cost of tools and environments
- the time needed for documentation of test automation along with the basis of estimation

Critical thinking to design tests that are more than just confirmatory, based on:

- Boundary Value Analysis
- Decision Tables
- State Transitions

Critical thinking is also needed for understanding the balance between confirmatory tests, exploratory tests, and rigorous tests. It is also important to be able to assess where automation fits in the software development lifecycle. This includes understanding how test automation is applied in unit testing, system testing, system integration testing, and acceptance testing. In a DevOps environment, this includes determining how continuous testing will be implemented into the pipeline.

Problem-Solving Skills to deal with the never-ending challenges presented by test automation that fails for obscure reasons. In many ways, test automation is one big problem-solving activity! Troubleshooting can be complex and often requires a good understanding of the software under test as well as the test automation.

Gaining Tool Knowledge

The tool landscape is always evolving for test automation. A test automation engineer should know about the major tools used in test automation in order to select the best tool for the work at hand.

Popular tools include:

- Selenium (both IDE and WebDriver)
- Playwright
- Cypress
- Katalon
- Test Complete
- Tosca
- Ranorex
- Postman

Some of these are open source while others are licensed and may require a considerable outlay for the purchase of the necessary licenses.

Most manual testers who want to learn automation need to start by learning one of the tools. When choosing a tool to learn first, consider:

- Tools in use in your organization
- Tools available for you to obtain on your own with minimal expense (Selenium IDE and WebDriver), Playwright, Katalon, Cypress, etc.
- Tools that are easily learned (Selenium IDE, Katalon, Playwright, Postman)
- Tools for which skills are sought in the marketplace

Even some of the more expensive tools may have free online learning available. It pays to be realistic though. There's no reason to spend time learning to use a tool that your organization will never be willing to buy due to cost.

Gaining Coding Skills

While there are no-code and low-code tools, they have limited capabilities for most automation efforts. It is a realistic expectation that programming skills will be required. At the outset, you may only be able to learn one language. However, learning to code in one language can help prepare you for others.

Popular languages for test automation include:

- Java for Selenium WebDriver and Junit
- C# for Selenium WebDriver
- Python for coding tests independent of a tool
- Ruby for coding tests independent of a tool
- JavaScript for Playwright and Cypress

Where to Start with Coding Languages

There are many programming languages, and the starting point depends on your own needs and whether a particular tool is already in use within your organization. If you are learning on your own with no existing tool set, perhaps the best place to start in language learning is JavaScript, as it is also used in one of the more popular and freely available test tools, Playwright.

However, Python is the most popular coding language for testers and comes in handy when coding utilities and tests outside of a particular tool set. If you are new to test automation, Python is a great place to start. It is easy to learn, has a large online community, and is versatile enough for various testing needs. Once you have a good understanding of the basics, you can then explore other languages based on your specific needs.

Factors to consider when choosing a language:

- Your experience level: Python is generally considered the easiest to learn, while Java and C# have steeper learning curves.
- The type of testing you will be doing: JavaScript is best for front-end web testing, while other languages are more versatile.
- The tools and frameworks you will be using: Make sure the language you choose integrates well with the tools you need.
- The needs of your team or organization: If your team already uses a specific language, it might be best to start there.

If your team is using Selenium WebDriver, you will probably want to learn Java or C#, whichever is in use, realizing that other languages, such as Python and Ruby, can also be used with Selenium WebDriver. The language you choose to learn first depends a lot on which one is currently in use within your organization. It's much easier to learn a programming language when you can see it in real use rather than just in training models.

Keep in mind that the above list of languages can change, so new training is needed to keep up with those changes.

Building Logical Thinking Skills

When dealing with code, the ability to think in logical and repeatable patterns is a must-have attribute and skill. Since test automation is "software testing software," you must learn how to think in terms of code constructs that are efficient.

Working with flowcharts and control flow graphs are two ways to gain this skill.

For some, this is a very challenging skill to obtain. Keep in mind that test automation is not for everyone, but as mentioned earlier, there are tools that are "low code" and "no code" that allow usage without heavy coding skills. Also, frameworks such as data-driven and keyword-driven allow testers without coding skills to interact with the framework by providing test data based on test

conditions. API testing is another example of test automation that can be achieved with minimal coding. These can often be the intermediate step between pure manual testing and pure test automation development.

Finally, remember that while technical skills come in handy for testers, career advancement in testing does not require technical skills. In actuality, human skills are often needed more than technical skills. Excellent analysis skills are the basis for all test development.

Gaining Knowledge of Platforms & Environments

Platforms and environments comprise the foundation of everything else: tools, architecture, data and so forth. The most common operating systems include Mac OS, Windows, and Linux. These comprise the context within which you will be working. These are typically decided by an organization based on their needs and preferences. It is important to remember that not all tools will work on all platforms.

Within the environment are browsers and devices to be tested. The variations and combinations are practically endless, so decisions must be made based on those browsers and devices most commonly used by the user audience. Decisions around these items can have major implications for testing. For example, consider the requirements for testing iOS and Android devices. Each have particular platform requirements. iOS, in particular requires the Mac OS to test using XCode.

Another key decision point, particularly in mobile application testing, is whether testing will be done on premise or in the cloud. The cloud offers the ability to test various devices and browsers without the need for specific platforms and environments. Kobiton is an example of one such cloud-based test automation service that can test real devices in the cloud.

Testing APIs

API testing is a great place to start learning test automation due to tool ease of use and simplicity of testing. APIs have a limited number of activities they can perform, which limits the amount of testing required. The API testing is also very focused and often can ignore the processing before and after the use of the API. One of the most popular tools for API testing, Postman, is available for free to individual users. API testing allows you to learn the basics of test automation without interfacing with a user interface. It also allows you to learn how to interact with mock objects.

API testing and service virtualization are two closely related concepts that are required for ensuring the quality and reliability of modern software applications. Service virtualization is a technique that simulates the behavior of dependent systems or services that are not yet available or are difficult to access during testing. This is a similar technique to using mock objects. This allows testers to perform comprehensive testing of the application under test without being blocked by external dependencies such as interfacing modules that are not yet ready for testing. Service virtualization creates a virtual version of the dependent system, service, or object, mimicking its behavior and responses. This allows testers to test various scenarios, including error conditions and performance bottlenecks, early in the testing cycle.

*For a more in-depth discussion, see the AT*SQA micro-credential course on API Testing.*

Understanding Database Functionality

Part of the test automation journey involves learning databases and how they work. A test automation engineer should have well-developed SQL skills. For example, when creating test data, an SQL query might be used to get an initial set of data from a database table. Care must be exercised when working with databases so as not to accidentally destroy or corrupt data, which is one of the risks of using production data in testing.

Working with Frameworks

Getting into test automation will likely not require that you be able to build a framework, but rather, be able to work within the framework you have been given. Test automation design frameworks were covered in Part 2 of the test automation micro-credential, as data-driven and keyword-driven frameworks were described. Part 3 of the test automation micro-credential discussed two other frameworks: JUnit and TestNG. It's worth revisiting those briefly.

JUnit and TestNG are two popular testing frameworks in the Java environment, which makes them limited to that environment. They can be used with tools such as Selenium WebDriver as a way to manage test execution and convey test results.

JUnit has a longer history and is the more commonly used of the two frameworks. It is relatively simple and easy to use and is commonly incorporated into DevOps pipelines to provide continuous and early testing. JUnit is commonly used by developers as a framework to support the development and execution of their unit tests.

There are test automation experts with strong opinions and preferences toward either JUnit or TestNG, depending on the needs at hand. As a tester making the transition to test automation, you will likely not have to decide between JUnit and TestNG. That is an architectural decision and is often made in concert between the architecture team, the developers and the test automation engineers.

Working With an IDE

If you will be working with tools that are code-intensive, you will need to learn how to work with an Integrated Development Environment (IDE). An IDE provides the ability, among other things, to:

- Edit code
- Organize projects
- Debug code
- Interact with the console

Some common IDEs are:

- IntelliJ Idea
- Visual Studio
- Eclipse
- XCode
- Android Studio

Building Reporting Skills

In test automation, as in all of testing, reporting is vitally important to accurately understanding test results. Dashboards are a great way to visualize the results of a test automation run. Many tools have built-in dashboards, which are convenient and reduce the implementation effort. Setup and configuration is usually required, but is also relatively straightforward.

Some tools, particularly some open-source tools, require a separate dashboard to be implemented. Commonly used dashboards include:

- Allure
- ReportPortal
- RobotFramework
- Tesults
- TestReport.io

Nearly all test management tools offer dashboard functionality to report automated test runs as well as manual test results. These tools include:

- QTest (by Tricentis)
- Jira (Xray, Zephyr Scale)
- PractiTest
- Azure DevOps

Some people have found that building their own dashboards in a spreadsheet can help fill gaps where commercial and open-source dashboards may be lacking. However, such homegrown dashboards can come with development and maintenance costs.

Gaining SDLC Understanding

As in all of testing, understanding the Software Development Lifecycle (SDLC) you are working in is vital to know when to:

- Plan test automation efforts
- Design test automation frameworks
- Prioritize which features will be automated first
- Determine the level of test automation
- Implement test automation
- Grow test automation
- Maintain test automation

As with other items discussed, if your organization is already using test automation, your priority is to understand the SDLC and know which activities have already been performed, which activities you are expected to perform, and when you are expected to perform them.

Examples of SDLCs are sequential (waterfall, v-model), iterative, and incremental (Agile and DevOps).

Understanding Which Features are Good Candidates for Test Automation

As a test analyst transitioning to test automation engineer, it is important to know which parts of an application are good candidates for test automation. Some helpful criteria to consider include:

- Features that are very predictable in their behavior
- Features that are stable and unlikely to change substantially
- Features that are high risk to the business and frequently used
- Features that must be included in regression testing

Part of this understanding comes from learning how to work with others (e.g., test analysts, developers, and users) to better understand the nature of items to automate. This requires a proactive effort on the part of the test automation engineer to communicate with others. There is normally an expectation that test automation engineers can communicate on a more technical level than test analysts. For example, this is needed when implementing automation into the DevOps pipeline or developing scripts to generate and manage test data.

Understanding Other Things Test Automation Can Do

Test automation has many uses beyond testing. Automation can be applied to test-related activities such as building and managing test data, resetting and creating test environments, and even applied to business functions, such as data entry. These are all value-added activities that can help justify the cost of test automation by reducing manual work and improving accuracy and speed.

Getting Started in Test Automation

The following are helpful in getting started in test automation.

1. **Get training in the concepts, tool(s), and methods needed**

There are options such as:

- Certifications (ISTQB, AT*SQA)
- Self-study
- Vendor courses (some are free, such as Cypress)
- Independent courses (range from free to paid), such as Udemy

2. Understand the concepts of test automation

These concepts were covered in Part 1 of this test automation micro-credential and form an essential foundation of why we do what we do. Without this understanding of basic test automation concepts, it is risky to start performing test automation activities without understanding their purpose and why they are needed. These concepts include:

Start slow and grow over time, as simple tests can provide significant leverage and show early benefits. Building your test suite slowly and measuring rates of change allows early successes and a proof of concept.

Strive for sustainability. Automate tests that are less likely to change and limit the number of tests to a manageable and maintainable level. Use techniques already discussed, such as modular scripting, data-driven testing, and keyword-driven testing, to help reduce the maintenance burden.

The use of “no-code” and “low-code” tools, as opposed to “code-first” tools, may also provide leverage in maintenance, especially those that offer self-healing functionality when a script fails. As AI becomes embedded within the tools, the self-healing and analysis capabilities of the tools improves significantly.

3. Start with low-code/no-code tools

This allows you to create test automation without the initial need to code. You can see quick results. One of the best ways to dive into test automation is to do it! While this may not be the best way to implement a large test automation project, it is a great way to create some small scripts and learn the basics.

4. Learn to program

Any language will do to start! See the discussion above for some practical tips.

5. **Find a mentor to help**

Having a real-life person to review your work and provide feedback is invaluable.

6. **Move to code-first tools to gain deeper knowledge**

An alternative is to code some utility programs, such as for test data maintenance.

7. **Build a portfolio of work to demonstrate skills**

This is a great help when trying to advance your career either within your existing organization or when looking for a new job.

8. **Continue to grow**

This means staying current with tool and technology trends. For example, Selenium is always changing. New tools are constantly being released into the marketplace, and other tools may decline in use. You will always be more hirable if you are familiar with the current and most popular tools. A good place to watch for tool trends is to check the Gartner Group annual reports.

www.atsqa.org