AT*SQA

Test Automation: Tools & Solutions Micro-Credential

Syllabus

Table of Contents

Test Automation - Tools & Solutions

3 Overview
5 Acquisition
12 Test Environments
20 Types of Tools
23 Tool Sources
25 Frameworks
27 Techniques
30 Examples - Tools and targets
36 Conclusion

Overview

This micro-credential describes the various tools and methods used to achieve successful test automation.

Not every test automation effort is tool-based. Some organizations choose to apply pure scripting or coding to implement test automation. For that reason, some use the term “Test Automation Solution” so as not to assume tool usage. In this syllabus, we use the term “test automation tool” with the understanding that the tool may be a fully-contained software application (e.g., Playwright and Cypress), a coding language (e.g., Python), external input (e.g., database, Excel, etc.), or all of these.

LEARNING OBJECTIVES FOR TEST AUTOMATION: TOOLS & SOLUTIONS

Overview

Aquisition

Summarize the different perspectives and goals for test automation
Explain the goals of a proof of concept exercise
Summarize the cost factors for a test automation tool and effort
Summarize the common licensing models for test automation tools
Summarize the factors to consider when calculating ROI

Test Environments

Explain the purpose and need for services to be mocked or virtualized
Summarize the types of environments that may be used for test automation
Summarize the components of a test automation solution
Explain how integrations may affect test automation

Types of Tools

Summarize the pros and cons of different types of tools

Tool Sources

Compare the different sources of tools

Frameworks

Explain how to design a test automation system for maintenance and re-usability

Techniques

Summarize the characteristics of good test automation frameworks
Summarize the framework approaches

Examples - Tools and Targets

Compare LowCode, NoCode, RPA and Code First tools
Summarize the uses of the programming languages in test automation
Compare the different test automation framework tools

Acquisition

Tool acquisition may be one of the most challenging aspects of test automation. There are many test tools and solutions on the market, some better than others. There is also the trap of believing that just because you have acquired a tool you can be successful with it.

Even in the case of free and open source tools, an acquisition effort is involved. For example, security approvals may be required, and technical assistance in obtaining the tool may be needed. In addition, licensing terms need to be closely examined.

In this section, we cover how to compute and justify the return on investment of a test automation solution and how to perform an adequate evaluation and proof of concept of the tool. It is also very important to consider how to get buy-in from the entire organization to use a particular tool or solution. Buy-in is needed from the larger project team, including the developers, testers, project managers and other stakeholders to ensure the understanding and acceptance of the cost to purchase and implement the tool. Buy-in is needed from the wider organization because the tool should be usable across multiple, if not all, projects. Acquiring different tools for different projects tends to be inefficient and creates an unnecessary learning curve unless there is a compelling reason such as unique technical concerns associated with a certain project.

Getting buy-in (Developers, Testers, Operations, Management)

An important part of the tool acquisition process is getting others in the organization to approve of and support the tool choice. One way to do this is to involve stakeholders in the evaluation.

Different stakeholders have different perspectives and goals for test automation. For example, developers often want test automation to integrate with tools they already use for unit testing and CI/CD. Testers want effective and sustainable tools to build regression test suites and reduce the manual testing load. Management wants faster testing with lower cost and high-quality deliverables.

It may be difficult to get approval and budget for a test automation effort in a project-focused environment where budget is allocated for the development of the project, but the support is handled under a different budget. In this case, it may be difficult to get approval for the money to develop the test automation since the project development sees little benefit from it and the support group, who would see the most benefit, doesn't have any project money allocated for automation.

It's important for all stakeholders (including project managers) to agree on the direction to avoid "shelfware," where the tool is acquired and installed but seldom used. Continuous engagement of all relevant stakeholders throughout the acquisition and implementation process can achieve this .

Evaluation / POC

It is vital to do a complete evaluation of any tool or solution candidate to determine the degree of fit.

Evaluations are typically short in nature and are often based on demonstrations and information provided by the vendor.

A Proof of Concept (PoC) is much more involved and is intended to test how well a tool candidate can handle test automation tasks in a given environment with actual applications to be tested. A POC may take days or weeks to perform.

Working with the tool vendor during the PoC can help to identify solutions for problem areas (e.g. accessing an image within a table) and can also provide an opportunity to assess the support you get from the vendor. Any complicated software application will have areas that are difficult to automate. Having support ready to help can save time and can help build a valid PoC.

FEATURES, CAPABILITIES, AND LIMITATIONS

Features are what the tool can do functionally and how it behaves non-functionally (such as how usable it is).

Capabilities are what the tool is able to do beyond the feature set. For example, a tool might list integration with other tools as a capability. It is not unusual to need to have the test automation tool integrate with the test management tool to ensure reporting is consistent across manual and automated testing.

Limitations are very important to identify as they might rule out a tool in your evaluation. This is best learned early in the evaluation process. For example, a tool may not interact well with a particular platform, which is a limitation. Some tools are designed for web applications but cannot support desktop applications. Others specialize in mobile applications. Ideally, the selected tool will work effectively on all the required environments. This is an important consideration to include in your evaluation criteria.

COST

Tool purchase and licensing costs can be obtained early in the evaluation effort and can be based upon:

- Number of people developing tests
- Number of people executing tests
- Number of tests performed in a given period of time
- Number of concurrent execution streams (agents)
- Number of devices (real and/or simulated), browsers, operating systems, versions, etc.

Other costs of ownership include:

- Training
- Upgrades
- Maintenance
- Add-ons and plug-ins

The availability and type of training may depend on the base knowledge of the testers. For example, some tools require good programming knowledge where others are designed to require only business domain knowledge. It may not be

realistic to assume manual testers will be able to learn programming (or will want to) in the timeframe available for training.

2.2.3 LICENSING

Licensing models can take many forms and are constantly changing. It is imperative to fully understand the license of any tool under consideration, as there could be hidden costs, limitations, and requirements. Some of the open source models, such as GPL, are very restrictive. Licensing isn't necessarily just a purchase consideration; long-term ownership and usage may influence the type of licensing that is acceptable.

The most common licensing models include:

Open source

- **Permissive** – Minimal restrictions with the most flexibility. Users can freely use, modify, integrate, and distribute the software without requiring changes to be open sourced.
- **Copyleft -** Requires modified works that incorporate open source code also be open sourced under the same terms. An example is the GNU General Public License (GPL).
- **Weak copyleft** – Adds some flexibility. An example is the Mozilla Public License.
- **Public Domain** – No restrictions at all. Creative Commons is one example. In another example, people have created entire test frameworks based on Selenium IDE and WebDriver and sold them commercially with no obligation to the Selenium project.

Commercial

- **Node-locked** – The tool can be used only on one specified machine.
- **Perpetual** – The license is based on a one-time fee.
- **Floating** – The software can be used between machines with a limit on how many users can concurrently use the tool.
- **Maintenance** – This covers changes that inevitably will be needed to stay current with the tool.
- **Elastic** – combines "pay as you go" and subscription models where you lease a certain base-level of usage rights, but can expand users (or virtual users) on demand if needed.

This model is commonly used with performance testing tools.

- **Concurrent** – X number of users can use the tool at a given time.
- **Subscription** – Pay by the month or year based on the number of users. May include updates to the tool, similar to a maintenance license.
- **Freeware and Freemium** – Free to use to a certain extent, often with restrictions on key features. To gain the more valuable features, a paid license is required. One example of this is SoapUI which is used for API testing. There is a free version and a paid "Pro" version. Both are downloadable apps. This model is also seen in SaaS tools such as Blazemeter for performance testing which allows a certain number of executions and virtual users per month.

Justifying ROI of Test Automation

An essential activity in tool acquisition is projecting when a positive Return on Investment (ROI) of test automation should be achieved. ROI is basically when the value of positive benefits of test automation starts to exceed the cost of the tools and other related costs such as personnel, environments, and process modifications.

ROI projections are estimates based on information at the time of the projection. These projections are often over-optimistic due to:

- Underestimating the complexity of the tests to be automated
- Underestimating the effort needed to create and maintain the automated tests
- Not considering the new skills needed to implement and apply the tool
- Unforeseen costs, such as if the vendor increases licensing costs
- Technical problems encountered in using the tool that require significant effort to

resolve, such as unforeseen tool limitations, test environment complexities (such as lack of integration, environment creation and maintenance), missing or highly dynamic object locators.

Investment costs can include:

- Tool licensing
- Obtaining external expertise in implementing and using test automation
- Training for the existing team in tool usage and test automation practices
- Hiring new personnel with test automation expertise
- In the case of free and open source tools, the total cost of ownership (licensing, configuration, and maintenance)
- In the case of “homegrown” tools, the cost of software development, such as labor and ongoing maintenance
- Integration with other tools in use
- Creation of automated tests and the framework
- Creation and maintenance of the test environment and test data

The return value can include:

- Reduced time needed for testing, which may be seen as more cycles of testing performed in a given period of time
- Increased test coverage, such as using a more comprehensive range of input data
- More accurate and repeatable testing
- Removal of the burden of repetitive manual testing so testers can focus on complex and more mentally demanding tests that cannot be automated
- Intangible benefits such as less drudgery and career growth opportunities
- Earlier testing, such as in-line CI/CD tests
- Higher quality and reduced regressions
- Better automated and objective reporting
- Improved time to market for product changes due to the faster testing cycles

Notably absent from the above list are the reduction of testers needed and higher numbers of defects found. Test automation does not replace testers; it enables them to test better where tools can’t. Also, test automation is very confirmatory when it comes to finding defects. In fact, it is

problematic if too many automated tests fail. This is one reason why test automation is best applied in regression testing and other kinds of tests, such as CI/CD, where there is high repeatability.

However, keep in mind that application changes can impact locators which can ripple and cause all tests to fail due to not being able to find the object specified in the tests. This can be prevented by the use of reliable locators other than XPath or CSS.

Essentially, positive ROI has been achieved once it can be shown that the value seen in test automation outweighs the cost. A requirement for knowing if ROI has been achieved is being able to measure the above items. An example of this is the metric known as Equivalent Manual Testing Effort (EMTE). For example, if manual testing of a test suite requires four hours per execution, and the test suite is executed 10 times per week, manual testing can be expected to take 40 hours to execute per week (EMTE). If all the tests in the suite are automated and each test run now takes one hour to perform and evaluate, the automated time is 10 hours for a savings of 30 hours of EMTE each week.

It is important to note that the EMTE metric does not include test development and maintenance. EMTE is just one way to show the value of test automation.

Test Environments

A primary consideration for the selection, implementation and use of test tools, specifically test automation tools, is that of test environments. Test tools that are used to create, execute, and maintain test automation can be considered part of the larger test environment in which they operate, along with other components such as test data and the applications under test.

Setup, Configuration and Maintenance

The creation and ongoing maintenance of a test environment involves many tasks. This topic describes some of the top considerations.

SECURITY (USER CREDENTIALS AND PERMISSIONS)

Once a test environment has been established, the proper security credentials and permissions must be established. These credentials include those needed for human testers, virtual users (such as those established for test user accounts in an application), devices, test databases, files, and other assets. Unfortunately, security access is often an afterthought that can delay further test automation progress. Security setup usually requires setting up the proper roles and users within those roles so that the access is controlled and testable. Creating test users can be a maintenance issue if the test environment is frequently refreshed from the production environment. Developing test automation scripts that create the necessary test users and roles may be a worthwhile effort.

Service Virtualization

Service virtualization or service mocking is used to create a test service that is used instead of the real service. This may be needed because the real service is not available (such as a service that is used to validate a credit card number), or because the service has not yet been developed (as in a CI/CD pipeline). There are two primary methods for creating virtual or mocked services.

Virtualized services can be created with service virtualization tools which will either analyze the code or monitor a live service to determine what are valid transactions. These are then mimicked back when the virtualized service is called. For example, you can run the tool for a set of credit card validations using both valid and invalid cards. These cards are then used in testing and the responses from the virtualized service are the same as would be retrieved from the real service.

Mocked services are usually created by coding a stubbed service. They are often quite simple and will usually just return a positive result. In the credit card example, the mocked service might always return “valid” for any card number it receives.

The purpose of creating these mocked or virtualized services is to allow test automation, to be developed and run even before the real services are in place. This allows automation to be developed sooner and to be effective early in the CI/CD pipeline. As the real services are created, the mocks are removed and the APIs have already been tested.

Service virtualization is a technique to enable virtual delivery of services which are deployed, accessed and managed remotely. An example of this would be a date and time conversion service that is accessed remotely and which resides on a virtual server. For purposes of test automation, knowing how virtual services are accessed is a major part of environment definition. Timing issues can be a problem as delays can result in failed transactions .

This process is known as synchronization and can apply to mock services as well. These services may be local and may not be fully implemented. Some people deal with synchronization using wait times, however, care must be taken not to increase wait times too much as this incurs a penalty in tests that run too long. The impact is that tests often

need tuning in terms of synchronization. This is part of test script maintenance and is best done when isolated to one or a few test scripts.

ONGOING MAINTENANCE

Just as test automation maintenance is ongoing, so is test environment maintenance. This includes updating and/or restoring test data for new test conditions, resetting test data after a period of testing, or syncing date-based test data to current processing dates. It is often a part of the test automation suites to have specific data generation or manipulation scripts.

Backing up the test environment is a consideration. If restoring the environment is desirable rather than building it from scratch, periodic backups should be performed. Outages in test environments can have significant impact on project schedules.

Multiple Environments

Test automation may span multiple environments depending on the level of testing and type of testing being performed. When test automation is used at the system integration level, multiple system environments are likely to be needed. It is not unusual for SaaS systems to have a “test” environment, but more than one environment may be needed (such as one for UAT, one for test automation development, one for performance testing, etc.). These environments can be expensive to procure and may require additional configuration to integrate these with other test environments. This requires time for initial setup and maintenance and must be planned early in a project.

UNIT TEST

Unit test environments are typically controlled by the developers and often vary greatly depending on the type of application being developed (web, mobile, etc.). In general, in a CI/CD process, the test automation will be expected to run in these developer-managed environments to provide

continuous testing for the build and integration pipeline.

TEST/SYSTEMS INTEGRATION TEST

These environments house entire systems, and sometimes multiple systems. A systems administrator or multiple administrators often control these environments. SaaS systems are usually managed by a vendor who may charge for system administration services and may control access in a way that limits what test automation can access (such as a database to verify test results).

UAT

While User Acceptance Testing is often a manual activity, some aspects of it can be automated. A frequent goal of UAT test environments is to mimic a live environment as closely as possible, realizing that exact replication is typically not possible due to the dynamic and sensitive nature of the test data. In addition, live systems may have databases that are too large to replicate in a test environment. When UAT must be run with multiple test cycles, creating and maintaining test data with test automation may significantly reduce the need for manual setup for each cycle of testing.

When large systems are being implemented, there is often a data migration process happening concurrently with testing. This makes it difficult to control the data as it will change with each data import. Being able to maintain controlled test data, either by creation or manipulation, is a requirement for the smooth progression of testing. This is particularly important in UAT testing when users are not necessarily data-aware and will be upset when the data changes between test cycles.

STAGING

The staging environment is typically the final environment before an application is released to production. Like UAT, this environment may closely replicate the live environment. However, if live data is used, it is essential to avoid unintentional actions against actual customers or users, so care must be taken. For example, an application that sends notifications to users could inadvertently send test notices to actual customers if controls are not in place to prevent such events.

Anytime production data is used, it is important to understand how it will be used and to whom it will be exposed. For example, credit card information cannot be shared with testers or appear in testing reports. Mocked data, either anonymized real data or fake data, should be used at all times to avoid the potential impact to personal information.

LIVE PRODUCTION

While testing in production has inherent risks, there may be times when test automation is used to detect problems in production. The main risks to be considered and mitigated are:

- Inadvertent data updates or destruction
- Degradation of performance
- Inadvertent access or exposure of personal information

Types of Environments

There are many types of test environments. In this syllabus, we address three of the broadest categories.

ON-PREM

These environments reside on servers under physical control of an organization, usually "on premises". The organization is responsible for all aspects of establishing, configuring and maintaining the environment, including hardware, networks, operating systems, test data, applications under test, and test automation tools including their assets. There is usually more control over these environments and more flexibility for usage as the system administrators are part of the organization. These administrators must be included in any planning and design of test automation frameworks to ensure the environments will be available as needed.

CLOUD

Cloud environments reside on servers often controlled by third-party companies. The exception are private and hybrid cloud environments, which are still under control of the organization.

Cloud test environments offer the advantages of lower cost of ownership and scalability upon demand. However, the configuration, control and maintenance of the test environment is still the responsibility of the organization.

Care must be taken to understand the costs associated with whichever cloud service is being used. There have been cases of organizations incurring very high costs unknowingly. Test automation makes this easy to do, particularly on systems that automatically scale based on usage.

For more information on cloud test environments, please see the Cloud Testing syllabus at https://atsqa.org/micro-credentials/testing-using-the-cloud

DISTRIBUTED

Distributed test environments are under the control of an organization, but the platforms are in multiple locations. An example is mobile testing performed on real devices around the world.

Components

HARDWARE

Regardless of the type of environment (such as cloud, on-prem, and distributed), hardware is the foundation for everything. Hardware encompasses servers, desktop computers, notebook computers, mobile devices (phones, tablets, etc.), peripherals (hard drives, monitors, keyboards, mice, etc.), and special purpose devices (barcode scanners, etc.).

It is important to ensure hardware version compatibility and correctness with the tools being used and the applications under test. Where unique test environments are required, it's also important to confirm availability and access settings for these components.

TEST DATA

Test data supplies test tools with needed input to support test conditions in test scripts. In some situations, the test data may be stored in a separate file or database to support data-driven testing. Alternatively, test data may be supplied in the test script itself. If production data is to be used for testing, anonymization may be required to protect personal information. Anonymization may be a non-trivial exercise requiring planning and the availability of special skills and tools.

APPLICATIONS UNDER TEST

The application to be tested can take many forms, such as:

- Web pages and web-based applications
- Mobile apps
- Desktop apps

Regardless of the type and form of the application under test, configuration management (CM) is essential to ensure the correctness of the application version in relation to test scripts, operating systems, test data, and co-existing applications. Failure to perform CM leads to invalid or incorrect tests, including false positives and false negatives.

Integration

Test automation must take into consideration integration. This integration can be between components, systems, environments, databases, and external entities. Integrations that are not available or not reliable in the test environment may have to be mocked to allow testing to proceed. The earlier the testing occurs in the pipeline, the more likely the integrations will not be available.

SERVICES

In this context, services are small, self-contained pieces of code to perform very specific functions. Sometimes, these are called “microservices”. These are commonly found in the cloud and accessed as web services using API test tools such as SoapUI or Postman. However, some test automation tools such as OpenTest can also perform APIs testing. Like the integrations they

may support, the services may not be available for testing when the application under test is tested. Service virtualization and mocking are often used to simulate the services that are not available. Tools are available that can create mock or virtual services based on specifications, the code, or by recording and replaying transactions. See Section 3.1.2 for more information on Service Virtualization.

BETWEEN TOOLS

A common concern with test tools is whether or not they will work together in a larger picture. For example, test management tools should integrate with test design and test execution tools to manage test cases and collect metrics about the tests. Integration and interoperability of tools should be part of the evaluation criteria for tools under consideration to ensure the tool suite will provide the necessary capabilities. If the tools do not integrate out of the box, additional software may have to be developed to create an interface that will support transferring data between the tools. This sometimes requires manual intervention, but that can create significant overhead and complexity for the testers when testing is occurring.

Types of Tools

Test automation tools and solutions can take many forms. There are pros and cons to each of these tool types, and the best fit for a particular need depends on the evaluation outcome and proof-of-concept (PoC). The pros and cons shown here are generic and may or may not apply to a particular environment or project.

No code

These tools promote the ability to create test automation by either recording a test session or dragging and dropping test actions into a sequential order of execution. The tool determines if a test passes or fails with no additional assertions needed.

Pros:

- Easy to use
- Quick way to build scripts
- Workflows can be built from building blocks, allowing good flexibility
- Starting point for learning test automation

Cons:

- Can be expensive
- Scripts are difficult to maintain and are usually just re-recorded when changes occur
- Integration with other tools is not usually included
- May not work with more complex applications or environments
- Don't allow coding to work with complex or unusual code

Low code

These tools are often mentioned as a variant of No Code tools. The difference is that Low Code makes provision for coding as needed. Common reasons for needing to work with code might include complex conditional logic, dealing with GUI objects that do not respond as expected, and adding diagnostic messaging to assist in troubleshooting.

The tool determines if a test passes or fails with no additional assertions needed, but additional assertions can be created depending on the nature of the item being automated and the nature of the test.

Pros:

- More flexible than No Code
- Provides a good basis for both technical and non-technical testers to use the tool effectively
- Some coding is allowed, so complex applications can be automated
- Objects that cannot be immediately recognized by the tool can be coded to work
- Can be a starting point for automating complex tests

Cons:

- Tend to be expensive, although not as much as the No Code tools
- Coding skills may be required to use the tool effectively

Robotic Process Automation (RPA)

RPA tools automate business tasks such as data entry. While not a test tool per se, they can be used to create test automation scripts. Verifying results is performed as a separate activity, as this is not a typical feature of RPA tools.

Pros:

- Because objects are not individually identified, the tools can work on objects that other tools cannot recognize

Cons:

- Because these tools were not designed to be test tools, there is no inherent comparison or reporting capability.

Some have used test automation tools to perform RPA. For example, a No Code or Low Code tool can record and playback functional actions for the purpose of performing them but not for testing them.

Full programming

Full programming can be used to create test automation in a variety of ways using:

- Existing tools that are script-based with no recording ability. Examples would be Selenium WebDriver, Junit, and OpenTest.
- Coding languages such as Python, Ruby and JavaScript

Pros:

- Complete flexibility to do anything needed
- Maintenance depends on coding stills and architecture

Cons:

- Programming skills required, almost to the level of a developer
- Maintenance can be costly, depending on architecture

Visual (AppliTools, Sikuli)

Visual test automation tools focus on the appearance of the object under test to detect any differences from a previous test. This can help identify defects such as incorrect GUI object appearance. These tools make use of visual images captured in one test to evaluate a subsequent test.

These tools can also inspect an object under test to determine if the items shown are correct. For example, a tester might use a visual tool to verify a specific text or other item shown on a GUI.

Pros:

- Easy to use and apply
- Minimal setup needed
- Finds visual defects not otherwise detected by functional test automation tools

Cons:

- Limited to visual differences
- Costs vary from free to moderately expensive

Tool Sources

Another helpful way to categorize tools is by their source. The most common sources are commercial and open source. However, some organizations prefer to build their own tools, frameworks, and scripts using languages such as Python, Ruby, and JavaScript.

Commercial tools

Commercial tools are sold by vendors with the intent of making a profit. These tools can vary widely in functionality, cost, and licensing models. Plus, you don't always get what you pay for with some tools.

However, there are some benefits that are possible with commercial tools, such as readily accessible support and staying current with new hardware and operating systems.

With commercial software, there is a persistent risk known as the "vendor risk." You never know when a vendor will go out of business, sell the tool to another vendor, drop support for a tool, or take other unexpected and uncontrollable actions. If commercial tools are the preferred choice, research the quality and satisfaction with the vendor. Automation is a long term investment and creating a dependency on a tool that may not exist next year is not a good decision.

Open source tools

With the rise of open source software licensing, test automation tools started to become more diverse and accessible due to the zero-dollar initial price. This was significant, as a typical commercial test automation tool can cost around $5,000 USD per seat license, annually.

Many people who have acquired and implemented open source tools have learned that the "free" price has a much higher cost in implementation and maintenance. There is also a "vendor risk" with open source. You never know when an open source tool project will go dormant, die completely, or be acquired by a commercial vendor and put under a new licensing model.

Another concern about open source is the potential for security vulnerabilities. While test automation tools typically do not carry the same level of risk as applications that allow external access, security releases and patches are frequent, as with commercial software. Some organizations will not allow the use of open source tools due to the perceived risk.

Homegrown tools

Some organizations like to own and control much of what they do. This is often seen in software companies that have the technical talent to build internal tools.

For other organizations, the decision to build in-house tools is based on whether or not they want to commit to the entire software development life cycle. For most organizations, it makes more financial sense to license or buy a tool as long-term maintenance costs can exceed the initial development cost of the tool.

However, there are exceptional cases where no other tool options are available and tools must be developed in-house. In this case, budget and time must be allocated for the development and support of the tool, for the life of the tool.

An option for homegrown tooling is to build a custom framework based on open source tools and use programming languages to implement test scripts. An example would be to build a framework in an IDE such as Eclipse or IntelliJ, use Java as the scripting language, and use Selenium WebDriver along with tools such as TestNG or Junit.

There is also the possibility of automating some tests using only code written in Python and executed at the command line level or by a script that executes at pre-determined times. Any time combinations of tools are used, the complexity of the test environment is increased.

Frameworks

There are different ways to define test automation frameworks. In this syllabus, we take the technical view that sees a framework as a way to house, control, execute, and maintain all test automation assets.

DESIGNING/BUILDING FOR MAINTENANCE (SUSTAINABILITY)

A primary goal in designing and building a framework is to facilitate sustainability and ease of maintainability.

This requires planning and thought toward future changes and growth. Making decisions about the framework that can have significant negative impacts in the future is easy. Examples include deciding on a particular locator approach. When designing a framework, one locator approach might appear to be the best, but in future projects, developers might not use the same kind of locator in their applications. Ideally a framework is designed to support multiple projects and to be extensible as needed. Long-term usage should be assumed.

BUILDING GOOD REPORTING AND ANALYTICS CAPTURE

A key aspect of a test automation framework is reporting test results in ways that are highly visible to everyone who needs the information and with minimal manual intervention. A best practice is to have a dashboard that is automatically updated. A test automation dashboard should show not only pass/fail results but also run times and test coverage.

RE-USABILITY AND MODULARITY

A good way to enhance maintainability in test automation is to reduce or eliminate redundancy in test scripting. This can be achieved by defining test scripts as functions that can be invoked each time they are needed in other scripts. This practice is described in further detail in the “Test Automation Implementation” micro-credential.

Common naming of scripts and modules, good commenting, and other coding practices will help to increase the maintainability of the framework. Defining these practices prior to commencing building the framework will help to set the guidelines.

Techniques

In order to be effective, test automation tools require some type of framework. Simply recording or scripting a test may achieve a short-term goal, but for building sustainable test automation that is efficient, scalable, and maintainable, a good framework is needed.

A test automation framework is a set of tools, guidelines, and best practices that help automate the software testing process. It provides a structured approach to writing, executing, and managing tests, and aims to make the testing process more efficient.

The characteristics of good test automation frameworks include the following:

- **Defines how tests are organized**

The framework defines how tests are structured, where they're stored, and how they interact with the system being tested.

- **Automates repetitive tasks**

The framework can automate repetitive aspects of software testing, such as menu behavior and login/signup/checkout flows. These repetitive tasks are captured in reusable code modules that can be shared across many test scripts.

- **Catches and fixes problems early**

The framework can help catch and fix problems earlier in the development workflow by allowing developers to create early test automation.

- **Ensures consistency and maintainability**

The framework provides a structured way to organize and execute test scripts, ensuring consistency, maintainability, and efficiency.

Common framework approaches include the following:

Data-driven

Data-driven test automation can reduce the number of test scripts and, therefore, the level of maintenance. The concept is to keep the test data in an external source such as an Excel file or CSV file. The data files contain input data values and expected results.

Each time a new instance of data is needed, the test script reads the external file to obtain the test data. The “Test Automation How To” micro-credential describes this practice in further detail.

Keyword / Activity-driven

Keyword-driven test automation is based on the concept that certain functions can be identified in a single word (keyword) and invoked each time that keyword is encountered in a test script. The “Test Automation How To” micro-credential describes this practice in further detail.

CREATING BUILDING BLOCKS

Keyword-driven testing is often combined with modular scripting because of the natural connection. For example, a keyword might be “login”. The associated script would also be named “login.” The “login” script could then read data from an external test data file to get the credentials for the test user. In this example, keyword-driven testing also applies modular test script design and data-driven testing.

ALLOWING ASSEMBLY BY NON-PROGRAMMERS

One of the challenges in test automation is getting manual testers involved. Some testers simply do not want to be automatons, and others may lack the essential skills and attributes needed for test automation.

The keyword-driven framework allows test designers to contribute test cases, test data, and other input without needing to know anything about test scripting. This is done through a GUI or direct access to the keyword and/or data files.

BDD and Test Automation

Behavior-driven development (BDD) is a way to apply “test-first” principles at a functional level. The BDD construct is:

Given that (a condition exists)
When (something occurs)
Then (something should result)

The BDD construct is more outcome-based than a user story. The Gherkin language is designed to implement BDD, but programming is required for the implementation.

AI-Generated

Artificial Intelligence (AI) is becoming an increasingly popular feature in test tools. AI can augment tests that are already automated. However, anytime AI is used to create test collateral, it must be reviewed for correctness. AI may not be able to derive the expected results, so augmentation is often required to create the acceptance criteria for a script.

AI-Driven

AI is also used in test automation to make decisions during test execution and to repair broken tests (“self-healing”). This application of AI can anticipate problems that might occur during test execution and avoid failures that could cause a test script or entire sets of tests to end abnormally. For example, if an object has been renamed, AI may be able to detect the change and update the script accordingly to use the new name. While this can save significant time, particularly with a changing UI, human intervention is advised to ensure AI has made the correct decisions.

Examples - Tools and Targets

The tool landscape is constantly changing as new tools come to market, existing tools are updated, and sometimes existing tools either change ownership or disappear altogether.

This syllabus is not intended to endorse or diminish any particular tool. The examples shown here represent commonly used test automation tools and are not exhaustive.

Low code/No code Tools

Tool	Supported Platforms	Vendor	Strengths
Selenium IDE	Web only Chrome, Firefox, Edge	Open Source	Free Record/ playback
Katalon Recorder	Web only Chrome, Firefox, Edge	Katalon (Based on Selenium IDE)	Free Record/ playback
Katalon Suite	Web, Mobile, API, Desktop	Katalon	Data-driven testing Mobile testing
Test Complete	Web, Desktop	Smart Bear	Drag and drop, Data-driven, keyword-driven
Tosca	Web, Mobile, Enterprise	Tricentis	Cloud-based, AI
Ranorex	Desktop, Web, Mobile	Ranorex	Reputation, Test Design
UFT	Desktop, Web, Mobile, Mainframe	OpenText	Many people with UFT Expereince

Robotic Process Automation (RPA) Tools

Tool	Platforms Supported	Vendor	Strengths
UI Path	Desktop, Web	UI Path	Designed for RPA

Code-First Tools

Tool	Platforms Supported	Vendor	Strengths
Selenium WebDriver	Web only Languages - Java, Python, C#	Open Source	Web Testing
Cypress	Web only	Open Source	Web Testing Totally encapsulated – no plugins or external components needed
Playwright	Desktop, Web, Mobile (emulators)	Open Source	Ease and variety of language understanding
XUnit Family	Desktop, Web, Mobile (emulators)	Open Source	Used mainly during component testing
OpenTest	Web, Mobile, API	Open Source	Uses YAML as primary scripting language with JavaScript as a second option.

Languages

Python – Very popular language (#1 in the world), especially among testers. It is an interpreted language with no compiler needed for execution. Developed in the late 80s.

Java - A general-purpose object-oriented programming language intended to let programmers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need to recompile. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. Java is the #3 most popular language behind C++ (#2).

JavaScript - Often abbreviated as JS, is a programming language and core technology of the Web, alongside HTML and CSS. 99% of websites use JavaScript on the client side for webpage behavior.

C# - intended to be suitable for writing applications for both hosted and embedded systems, ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions.

Ruby - an interpreted, high-level, general-purpose programming language. It was designed with an emphasis on programming productivity and simplicity. In Ruby, everything is an object, including primitive data types. It was developed in the mid-1990s by Yukihiro “Matz” Matsumoto in Japan.

API Testing

Tool	Platforms Supported	Vendor	Strengths
Postman	Windows, Mac	Postman	Nice UI
SoapUI	Windows, Mac	Smart Bear, with Free open source version available	Handles SOAP and REST, Performance and Security test features
SwaggerHub	Windows, Mac	SmartBear	Tests API functionality and performance
OpenTest	Windows, Mac	OpenSource	Handles Web, Mobile and API testing

Mobile Testing

Tool	Platforms Supported	Vendor	Strengths
Appium	Windows, Mac	OpenSource	Widely used
Katalon Studio	Windows, Mac	Katalon	Ease of use
OpenTest	Windows, Mac	OpenSource	Handles Web, Mobile and API testing

Frameworks

Some tools claim to include frameworks, but since there are so many ways to define frameworks, specific tools are not mentioned. It is best to refer to the previous section on frameworks to determine your own needs and solutions.

ROBOT FRAMEWORK

“Robot Framework is an open source automation framework for test automation and robotic process automation (RPA). It is supported by the Robot Framework Foundation and is widely used in the industry.

Robot Framework’s human-friendly and versatile syntax uses keywords and supports extending through libraries in Python, Java, and other languages. It integrates with other tools for comprehensive automation without licensing fees, bolstered by a rich community with hundreds of 3rd party libraries.”

TESTNG

“TestNG is a testing framework inspired from JUnit and NUnit but introducing some new functionalities that make it more powerful and easier to use, such as:

- Annotations
- Run your tests in arbitrarily big thread pools with various policies available (all methods in their own thread, one thread per test class, etc.)
- Test that your code is multithread safe
- Flexible test configuration
- Support for data-driven testing (with @ DataProvider)
- Support for parameters
- Powerful execution model (no more TestSuite)
- Supported by a variety of tools and plug-ins (Eclipse, IDEA, Maven, etc.)
- Embeds BeanShell for further flexibility
- Default JDK functions for runtime and logging (no dependencies)
- Dependent methods for application server testing

TestNG is designed to cover all categories of tests: unit, functional, end-to-end, integration, etc.”

JUNIT5

JUnit 5 is the current generation of the JUnit testing framework, which provides a modern foundation for developer-side testing on the Java Virtual Machine (JVM). This includes focusing on Java 8 and above, as well as enabling many different styles of testing.

Unlike previous versions of JUnit, JUnit 5 is composed of several different modules from three different sub-projects.

JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage

The JUnit Platform serves as a foundation for launching testing frameworks on the JVM. It also defines the TestEngine API for developing a testing framework that runs on the platform.

Furthermore, the platform provides a Console Launcher to launch the platform from the command line and the JUnit Platform Suite Engine for running a custom test suite using one or more test engines on the platform.

First-class support for the JUnit Platform also exists in popular IDEs (IntelliJ IDEA, Eclipse, NetBeans, and Visual Studio Code) and build tools (see Gradle, Maven, and Ant).

JUnit Jupiter combines the programming and extension models for writing tests and extensions in JUnit 5. The Jupiter subproject provides a TestEngine for running Jupiter-based tests on the platform.

JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform. It requires JUnit 4.12 or later to be present on the class path or module path.

Conclusion

In this micro-credential syllabus we have covered the high points of the many test automation options available ranging from open source to commercial. We have examined the pros and cons of the test automation tools, along with the risks and benefits.

Test automation is a project that requires definition of needs (requirements), design, coding, testing and maintenance. In fact, maintenance comprises the largest effort and expense of most test automation efforts.

The test tool landscape is constantly changing. You are encouraged the verify the tool information presented in this syllabus for any changes.

Even with the risks and challenges, many organizations have been successful in implementing and growing their test automation efforts to be an integral part of their overall testing efforts. By following the information in this syllabus and the other related micro-credentials, you too can be successful in test automation.

Sources:
- https://robotframework.org/
- https://testng.org/

www.atsqa.org