Alliance for Qualification

Released Version 1.0

12th April 2023

A4Q Selenium 4 Tester Foundation

Syllabus

Copyright Notice

Revision History

Version	Date	Remark
0.6	29.10.2022	Initial Merged layout
0.7	15.10.2022	Internally reviewed document
0.8	15.01.2023	TP reviewed document
0.9	26.03.2023	Beta version document
1.0	12.04.2023	Release version

Table of Contents

Acknowledgements

This document was formally released by A4Q GmbH on 12.04.2023.

0 Introduction

0.1 Purpose of this syllabus

This syllabus presents the business outcomes, learning objectives, and concepts underlying the Selenium Tester Foundation training and certification.

0.2 Examinable objectives and cognitive levels of knowledge

Learning objectives support the business outcomes and are used to create the certified A4Q Selenium Tester Foundation exams. That is, the candidate may be asked to recall, show understanding, apply knowledge, and analyze a scenario mentioned in any of the six chapters.

The knowledge levels of the specific learning objectives are shown at the beginning of each chapter, and classified as follows:

- K1: remember
- K2: understand
- K3: apply
- K4: analyze

0.3 The A4Q Selenium Tester Foundation exam

The A4Q Selenium Tester Foundation exam will be based on this syllabus. Answers to exam questions may require the use of material based on more than one section of this syllabus. All sections of the syllabus are examinable, except for the Introduction and Appendices. Standards, books, and ISTQB® syllabi may be included as references, but their content is not examinable, beyond what is summarized in this syllabus itself from such standards, books, and ISTQB® syllabi.
The exam shall be comprised of 40 multiple-choice questions. Each correct answer has a value of one point. A score of at least 65% (that is, 26 or more questions answered correctly) is required to pass the exam. The time allowed to take the exam is 60 minutes. If the candidate's native language is not the examination language, the candidate may be allowed an extra 25% (15 minutes) time.

0.4 Accreditation

Training providers wishing to offer training for the A4Q Selenium Tester Foundation should be accredited by A4Q and use the official A4Q Selenium Tester Foundation training materials.

0.5 Level of Detail

This syllabus consists of:

- General instructional objectives describing the intention of the certification
- A list of terms that students must be able to recall.
- Learning objectives for each knowledge area, describing the cognitive learning outcome to be achieved
- A description of the key concepts, including references to sources such as accepted literature or standards

The syllabus content is not a description of the entire knowledge area of Selenium automated testing; it reflects the level of detail to be covered in this foundation level training courses. This syllabus does not contain any specific learning objectives related to any particular software development lifecycle or method, but it does discuss how these concepts may apply in various software development lifecycles.

0.6 How this syllabus is organized

There are six chapters with examinable content. The top-level heading for each chapter specifies the time for the chapter; timing is not provided below chapter level. For the A4Q Selenium Tester Foundation training course, the syllabus requires a minimum of 18.25 hours of instruction, distributed across the six chapters and the practical exercise as follows:

Chapter 1: Introduction to Test Automation – 175 minutes
Chapter 2: Automation Web Technologies – 250 minutes
Chapter 3: Selenium Automation Tools – 155 minutes
Chapter 4: Using Selenium WebDriver – 150 minutes
Chapter 5: Implementation of Test Automation in an organization – 80 minutes
Chapter 6: Adapting a Selenium TAS – 45 minutes
Practical: Java or Python – 240 minutes

0.7 Business outcomes (BOs)

After this certificate, candidates are expected to

1) Have an understanding of Selenium automation tools
2) Have an understanding of the different TAS that can be implemented using Selenium tools
3) Have an understanding of the underlying web technologies and how they are used on Selenium
4) Adopt the best practices when onboarding Selenium automation projects
5) Understand how to maximize the return on investment for the automation projects

0.8 Acronyms

Acronym	Meaning
SUT	System Under Test
API	Application Programming Interface
CSS	Cascading Style Sheets
EMTE	Equivalent Manual Test Effort
gTAA	Generic Test Automation Architecture
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
JDBC	Java Database Connectivity
JSON	JavaScript Object Notation
OS	Operating System
POC	Proof Of Concept
REST	Representational State Transfer
RPA	Robotic Process Automation
SDLC	Software Development Life Cycle
SOAP	Simple Object Access Protocol
TAE	Test Automation Engineer
TAF	Test Automation Framework
TAM	Test Automation Manager
TAS	Test Automation Solution
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
XML	eXtensible Markup Language

1 Introduction to Test Automation – 175 minutes

Keywords

Test automation solution, generic test automation architecture, test automation architecture, test automation framework, Application Programming Interface, component testing, integration testing, system testing, acceptance testing, capture and playback tool, off the shelf TAS, custom TAS, page pattern automation framework, data driven automation framework, keyword driven automation framework

Learning Objectives

STF1-1 (K2) Understand how automation testing can be applied to different test levels

STF1-2 (K2) Understand the components of gTAA and how to build a TAS

STF1-3 (K1) Remember industry-wide expectations from a TAS

STF1-4 (K2) Understand the relationship between manual and automated tests

STF1-5 (K2) Explain the benefits and limitations of automation testing

STF1-6 (K2) Understand the different test automation solutions

1.1 Test automation in a nutshell

Test automation is the ability to use a machine to run test scripts on the SUT without the need to continually be monitored or required interventions from a human software tester. In its narrowest sense, test automation is the use of software to perform or support test activities, e.g., test management, test design, test execution and results checking. It is a misconception that test automation happens only on the graphical user interface level.

A test automation solution (TAS) can interact with the SUT at multiple levels through the use of adaptors. For example, interactions can occur on the API level, database level, protocol level like ‘http,’ OS level, webservice level, etc.

Nevertheless, for test automation to be possible, the TAS must be able to exhibit the following properties on the SUT:

1) Observability: ability for the TAS to read the state of the SUT and its components
2) Controllability: ability for the TAS to change the state of the SUT and its components

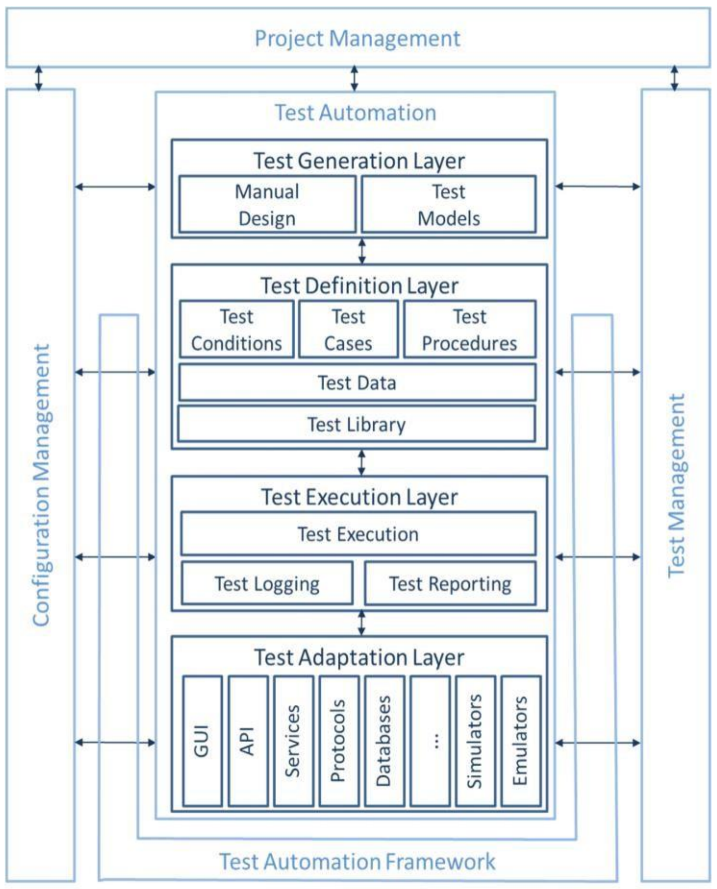
Test automation has a defined layered architecture which is termed as the gTAA (generic Test Automation Architecture). The gTAA is a representation of the layers, components, and interfaces of a test automation architecture, allowing for a structured and modular approach to implementing test automation.

The gTAA essentially consists of:

- The test generation layer:
  - This layer handles how test scripts will be created. Tests can be automatically generated by tools. For example, using test models in model-based testing environments or it can also be created manually by software test engineers.
- The test definition layer:
  - This layer will encapsulate how test conditions, test cases and test procedures are handled by the TAS. It needs to interface with test libraries and test data to convert test scripts into executable tests. So, the test definition layer handles the test implementation in the TAS.
- The test execution layer:
  - The test execution layer will allow the executable tests to be carried out on the SUT. This layer exerts the observability and controllability of the TAS. This layer encapsulates test execution, test logging and test reporting.

The test adaptation layer:
  - This layer provides an abstraction of the adaptors that support interaction with the various interfaces, components, and configurations of the SUT. This layer may encapsulate adapters for interaction on the GUI level, protocol level, simulator level etc.

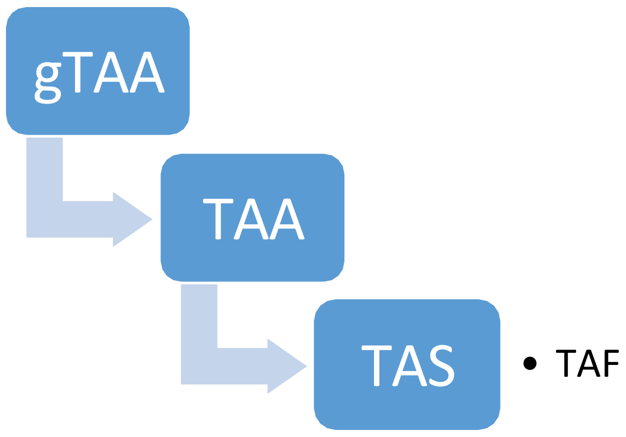
Below is the gTAA layered architecture:



(Source: ISTQB® Test Automation Engineer Syllabus V1.0)

There may also be interfaces from the TAS towards configuration management tools, test management tools and project management tools. These tools may support the correct functioning of the different layers. The test automation framework (TAF) consists of the test harness and test libraries that will support the TAS in activity. For instance, the Selenium automation engine combined with test libraries will constitute the test automation framework.

The gTAA is a blueprint to constitute a TAA (Test Automation Architecture) based on which the TAS will be conceived supported by the TAF.



Although test automation is mostly viewed as a tool for test execution, test data creation and test result analysis can also be automated. Test data can be generated automatically during run time to be random or unique (e.g., using timestamps). The evaluation of pass/fail status of tests should also be part of test automation.

Test automation requires design, test creation, and maintenance at different levels of testing. This may include the environment in which the tests will be executed, the tools to be used, the code libraries which supply functionality, the test scripts, test harnesses and the logging & reporting capabilities to evaluate the test results. Depending on the tools used, monitoring, and controlling the execution of the tests may be a combination of manual and automated processes.

Objectives of test automation may include:

- Improving the efficiency of testing.
- Reducing the cost of each test run
- Performing tests that are not manually possible.
- Reducing test execution time
- Increasing the frequency with which tests can be run
- Providing wider functional coverage
- Test Automation at different levels

1.1.1 Test automation at different test levels

Tests are categorized according to where they occur in the SDLC or the level of detail they include. Component testing, integration testing, system testing, and acceptance testing are the basic four test levels of software testing. The objective of having test levels is to organize software testing in the test process and group test artifacts by test level. Test artifacts could be specification documents, test cases, test basis, etc. Each test level has its own set of test objectives and its own set of expected type of defects. Therefore, the test levels help in identifying test coverage gaps and ensure that the development lifecycle phases are reconciled.

Component testing using test automation is a way to implement the "preventive" and "corrective" aspects of verification and validation. When developing unitary system modules, component testing is the implementation of early testing and helps in preventing defects escaping to higher levels. The tests here can therefore provide a lot of value if implemented through test automation. Developers and TAEs will be able to get quick feedback on the quality of the developed module(s) before moving to the next step. Implementation of component automated tests does not require the setup of additional infrastructural resources. It can be easily done on the local machines being used by the developers and the tests can be implemented using the IDE itself integrated with some automation tools.

Once the separate components or modules have been developed and unit-tested, they must be connected to each other to form the system. Component integration testing can thus start. At this level, the interfaces and interactions between components are tested. The adaptation layer of the gTAA facilitates the component integration testing by using various adaptors that will interact with different interfaces of the SUT. For example, database adapter can be used to execute automated checks to test the interaction of a component to the database.

The component integration tests ensure that the correct modules are invoked and that the right data is transferred through the interfaces at the right time, in addition to ensuring that the components functionally perform well together.

System test starts after component integration tests. The objective of system test is to verify a fully integrated system by checking if functional flows within the system are operational. An assessment of the completeness of the implemented system requirements is carried out in system test. The functional system tests of the browser-based application can be scripted using Selenium which saves a lot of time during test execution. Automation tests are normally the most prevalent in this level. At this stage, there are test packs created to support regression tests and smoke tests.
In system integration testing, the interfaces that may exist in an ecosystem of software applications are tested. An ecosystem of software applications includes software and the associated hardware that share interfaces beyond the system boundary to carry out an operation. This may be in terms of an API call towards a shared webservice or a database request towards a resource beyond the system boundary. The

main objective of this test level is to check compatibility and conformance of interfaces between shared resources. One typical way to implement test automation on this level is to write scripts that test all external calls or interfaces made by the system that goes outside its boundaries. This gives confidence in the reliability and conformance of the interfaces. These tests require the adaptation layer of the gTAA to be implemented effectively.

Acceptance testing is often the last test level carried out to evaluate satisfaction of user needs, requirements, and business processes. It assesses the alignment of functional and non-functional requirements with the business workflow. These tests can be scripted on automation tools by test analysts working on behalf of the end users.

Similar to system testing, acceptance testing often focuses on how a system or product will behave and function when used by its intended users in a real or simulated operational environment. The fundamental goal is to increase user confidence in the system's ability to satisfy criteria, cater to needs, and carry out business procedures with the least amount of complexity, expense, and risk. Using a remotely controlled Web browser, Selenium can simulate user interactions with the SUT to automate acceptance testing (or functional testing). Furthermore, performance tests are done by executing some Selenium written tests simulating different users hitting a particular function on the web app and retrieving some useful metrics. Additionally, Selenium Grid (see section 3.2.3) has significantly increased the scalability of performance testing.

1.1.2 Test automation tools & utilities

An application can be tested manually or automatically using specialized tools. In manual testing, a test analyst assumes the role of an end user to verify the proper implementation of the functionalities. Manual testing is hence prone to errors as the effectiveness of the tests depends heavily on the test analyst’s experience and attentiveness. Manual testing is hence a time-consuming and demanding activity. These problems can be solved by automated testing. Test analysts can design repetitive and reusable test scenarios using automation tools. These test cases can then be executed as frequently as required. Additionally, as an application naturally grows to include more features, it becomes more complex and manual testing alone becomes unsustainable. Automating the repetitive tests such as regression test suites, becomes the only viable solution to keep the long-term cost of software testing reasonable.

Automation tools can be grouped in different categories:

1) **Capture / playback tool**

   The capture / playback tool allows the software test analyst to record the actions undertaken on the SUT as part of the manual test. The interactions, recorded in a linear script, can then be played repeatedly. Nevertheless, the capture and playback solution for automation testing is not highly scalable and the cost of maintaining the tests can grow on the long term. The Selenium IDE (see section 3.2.1) is an example of the capture / playback tool.

2) **Off-the-self TAS**

   There exists some off-the-shelf test automation solutions that can be directly implemented onto projects. These test automation solutions are made from a TAA that give some flexibility and adaptability of the TAS to many projects. These solutions nevertheless will not have in-built functionalities to support all projects. They are built in a generic way to provide a quick implementation on the project, and they may have some limitations. Compared to capture / playback tools, an off-the self TAS is more adaptable and provides lower costs for test script maintenance. The Selenium project has some off-the-shelf test automation solutions that are maintained by the community (See section 3.1.1 for examples).

3) **Custom made TAS**

   Custom TAS is an automation solution that is developed from scratch or from an open-source TAS. It consists of a specifically coded TAS so that it suits all the needs of a specific project. Given that it is custom made, the TAS will be made to fit all the needs of the project and can therefore be of more value than off-the-self TAS for the project.

1.1.3 Test automation in the industry

Numerous automation tools have been developed to satisfy the rising demand for test automation in the software industry. The current test automation strategies are constantly being improved by researchers and developers. A shift from manual testing to heavy automation testing for repetitive tasks can be seen across the industry. Increasing aspects of test processes are being automated using a variety of methods and tools. Tools for record and playback, keyword-driven TAS, strategies for exploring event flows, and model-based methodologies are all constantly changing with a higher level of test automation.

Another crucial element in this area is the optimization of this resource-intensive activity. When running bigger test suites, dependencies between separate tests can result in deadlock situations and such dependencies can be managed by restructuring of the test suites and test cases. The use of UML models and the use of model-based testing methodologies are also both essential components of the automated testing framework in the software industry. Using these models, automation tools can generate and execute tests on an SUT.

Development and testing activities are gradually becoming a single integrated and simplified process due to the streamlining of software test automation procedures. The adoption of the agile mindset in the SDLC has led to test automation being implemented in a way to create value early in the testing chain. CI/CD frameworks also allowed test automation to be encapsulated in the integration and deployment pipelines.

This helps save time and resources by ensuring that businesses invest in the proper software tools that match the budget and fulfill their operational requirements.

With the advent of test automation, the industry expectation has been growing ever since. Below is a non-exhaustive list of considerations and industry expectations from test automation solutions.

- The spice of Randomness: The TAS is expected to be able to randomly select test data. This can be done by leveraging on test adapters for databases and API. The idea is to be able to vary the nature of test data (length, type or use of special characters) each time test automation is executed.
- Soft failure of tests: The failure of one test case in the automated test script should not be impacting the other independent tests in the same test suite. The ripple effect of test failure should thus be restricted to the failed tests.
- Recovery strategy of tests: The TAS should be able to recover from failed test. If the test execution stops because a web element was not found, then it defeats the purpose of having test automation.
- Looping of tests cases: The TAS is expected to run the same test repeatedly and preferably with different test data for each repetition. This is useful in situation requiring large data manipulation.
- High configurations of test cases: The flexibility to configure test cases for execution on different test environment and test conditions without requiring changing the source code is important.
- High coverage quicker: Test scripting for automation is expected to be easy to learn and carry out. Having such TAS design helps achieving higher coverage quicker.
- Extended verification checks: In addition to simple UI checks by the TAS, it should be possible to do verification checks deeper in nature. For example, checks in databases and webservices.
- Unattended test execution: The essence of test automation is to be able to execute tests without the need for active monitoring and intervention from humans. This way tests can be executed after working hours.
- Reliable results: The value of test automation is the test result. The reliability of the result is therefore of utmost importance. Failure must be reported when actual and expected result differs.
- Easy access to real-time results: It is always helpful to have results as fast as possible. As such, as soon as a failure happen in test automation, manual checks can be triggered. Hence a real time result monitoring is of great value in test automation.
- Quicker tests executions: It is a growing expectation that as soon as the SUT is deployed, the test results should follow. Leveraging on the CI/CD pipeline and parallel test execution are some strategies to make test execution faster.
- Reliable metrics for ROI: TAEs should have means to calculate the time saved with test automation to report to management. This need is to be considered in the TAS design phase.

1.2 Manual testing and test automation

Traditionally, in most organizations, manual test cases are implemented first. When converting the existing (manual) tests to automated ones, the state of the manual tests must be evaluated to determine the most effective approach to adopt. The existing structure of manual tests may not be suited for test automation, and in that case, a re-write of the tests may be required by making a maximum reuse of part of existing manual tests (e.g., input values, expected outcomes, navigational paths).

Not all tests can or should be automated. Prior to investing automation effort, one needs to consider the suitability and long-term maintenance effort for the automation tests.
The suitability criteria may include, but are not limited to:

- Frequency of execution of the test
- Complexity to automate the tests
- Compatibility of tool support
- Maturity of test process
- Suitability of test automation for the stage of the software product lifecycle
- Reliability of the automated test interfaces
- Sustainability of the automated test on the long term
- Return on investment for the automated tests

Also note that certain test types can only be executed (effectively) in an automated way, e.g., reliability tests, stress tests, or performance tests.

(Source: ISTQB® Test Automation Engineer Syllabus V1.0)

1.2.1 Benefits & limitations of manual testing

The basis of software testing is manual testing. Of all the testing types, it is the most important stage. Software test analysts write test cases to later execute them manually. In most organizations, test automation is explored after manual testing reaches a mature state. However, manual test analysis activity is at the heart of static & dynamic testing. Test automation cannot fully guarantee error free UI testing, reliability testing and usability testing. Although artificial intelligence is helping on this front, manual testing remains key for these tests. For example, exploratory testing is still an important part of the test activity and is often conducted manually to get the human feel of quality of the SUT.

There are situations when manual testing may be more beneficial:

- Human observation during manual testing can be more effective for identifying potential flaws.
- Manual testing offers a precise simulation of the user experience.
- Manual testing concentrates on complex features and functions.
- It helps to identify defects that are not code related.
- No prior programming knowledge is required for manual tests.
- It is highly recommended for dynamic GUI designs.

- It is preferred for usability and exploratory testing due to the human factors.
- It is more suited to applications that are not mature yet and that have ever changing requirements.

Nevertheless, manual testing has the following limitations:

- It sometimes requires excessive time or resources, sometimes both.
- Human errors are always possible.
- Lower consistency and repeatability of tests when done manually.
- The effectiveness of the manual testing is heavily dependent on the knowledge and knowhow of the test analyst.
- Some tests are impractical to be done manually. For example, performance testing or tests that require high speed interaction with the SUT.
- It is impractical to manually compare large datasets such as whole databases.
- Repeatedly running the same tests is a time-consuming and monotonous activity.

However, small projects with few test cases, a strong focus on UX/UI, and complex user scenarios are better suited for manual testing.

1.2.2 Benefits & limitations of test automation

Determining which of the two test methods, manual and automated, is more effective is challenging, and has long been open for debate. The benefit of manual testing is that a human's ability to interpret test findings is superior to that of a TAS. Test automation, however, is far superior and beneficial in terms of accuracy, test consistency, execution time, verifications, and long-term cost.
The benefits of test automation are:

- Tests can be executed on demand and as often needed.
- Running repetitive tests automatically is often faster and more effective than doing so manually.
- Performing tests that are often difficult to carry out manually (such as reliability or performance tests).
- Reduction in test execution time allows more test execution per build or release.
- Easier to achieve higher coverage of test basis (for example cross- browser testing)
- Ability to scale up test execution such as concurrency in test execution.
- Decreased corporate expenses in the long term and efficient use of resources.
- Enabling manual testers to conduct more engaging and challenging manual tests (e.g., exploratory tests).
- Minimizing errors made by manual testers due to lack of focus, particularly for repeated tests like regression tests.
- Regular and timely feedback on software quality allowing early defect removal.
- Test execution outside regular business hours is possible.

- Increasing self-assurance and confidence in builds or releases
- If automated tests are well designed, they can be highly reusable.

Test automation ensures long term return on investment (ROI) on software quality assurance. However, it is limited in what it can accomplish, and automated testing has some drawbacks to manual testing. Some test automation restrictions are inherent in the system and must be balanced against manual testing, whilst others are the result of imprecise pre-programming, such as the failure to create efficient automation test hooks. Some of these limitations and constraints can be reduced by good programming principles and guidelines, while others cannot.

The limitations include:

- Not every test can be or should be automated as manual testing will nevertheless be necessary (exploratory testing, certain fault attacks, etc.)
- Skillsets for automation coding, scripting, and testing are not readily available.
- Automated test oracles could differ from manual test oracles, necessitating identification.
- The likelihood of manual test analysis, planning, and execution is still high.
- False sense of confidence brought on by several automated tests running with few flaws discovered.
- Using cutting edge technologies or methodologies can cause technical issues for the project.
- Collaboration and cooperation with the development team is a must.
- Verification needs to be implemented correctly in test automation to reduce the effects of false negative failures.
- It is only by thinking long-term that automation projects will succeed; short term thinking can completely derail the automation effort.
- Success requires organizational maturity; automation based on poorly maintained testing procedures merely produces mediocre quality testing.

The success of a test automation project is possible and frequent. However, that accomplishment is solely the consequence of close attention to detail, sound engineering guidelines, and prolonged, arduous labor Success on an automation project does not happen by accident.

1.2.3 Balancing manual and automated testing

It is uncommon to come across an organization in the IT industry that does not use a combination of automated and manual testing frameworks. When the two work well together, manual testing and automated testing do not compete with one another; rather, they strengthen and complement one another and produce a higher test quality.

Automated testing typically improves testing consistency and speed, but it can only be as effective as the test scripts that it executes. In order to find problems from the viewpoint of the user or unanticipated defects from unplanned scenarios, etc., manual testing is complementary to automated testing. Along with effective test automation, human testing heuristics are greatly needed.

Some instances of a good automated and manual testing combination include using a variety of these tests to cover various elements of the same feature, having manual tests pick up where automation left off, and having tests that are only "semi-automatic" and require manual intervention in between tests in order to move on to the next automation set of tests.

Traditionally, the majority of testing teams use a combination of manual and automated testing. Manual testing is also a safety net for automation tests. A TAS is software after all. There may be numerous problems causing a TAS not to work. For example, there may be a browser update and the Selenium based automation test does not work anymore and requires several hours of work to get it fixed. In the meantime, the tests can be executed manually and support early testing principles.

However, the reality of the software quality industry encourages a new strategy, where test leaders must coordinate testing efforts across all testing tools and methodologies. This usually starts with manual testing (scripted and exploratory) and is subsequently replaced by the automation of repetitive tests managed by the CI/CD pipeline. This is most suited for projects with frequent builds requiring regular quick validation, especially those having continuous integration and delivery processes in place.

The major benefit of the reconciliation of manual and automated testing approaches is that it combines:

- The speed and maximized coverage of automated tests.
- The adaptable and unrestricted nature of manual testing.

Effective reconciliation of manual and automated testing is comparable to leading an orchestra, using various tools, playing various tunes, occasionally at various speeds, and working towards a common objective that is larger than the sum of its parts. Whilst careful planning and execution are necessary, the outcomes are frequently remarkable.

1.3 Test Automation Solutions Design

The interaction between a TAS and the system under test (SUT) is complex in nature and varies with technologies and framework architecture. When speaking of test automation, the most dominant idea that comes to mind is a software which is interacting with a system under test (SUT) at the UI level. Test automation is however not limited to only the UI level interactions. Automation tests can be carried out in the backend layer as well. For example, running an API call automatically or running a database stored procedure automatically. Such automation tests are implemented through test adapters which interface between the system under test and the automation test scripts. These tests fall under test automation as long as the human intervention in the process is non-existent or limited. There are also interactions that happen in memory instead of the UI level. This is referred to as headless test automation (see section 3.3.3).

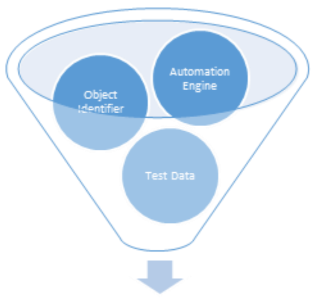
Nevertheless, in any type of automation test solution, there are always some common components.

The different arrangements of these components make up different architecture designs.
Any TAS requires of 3 main components to sustain a stable operation:

1) Object Identifier / locator
2) Test Data
3) Automation Engine

The arrangement and grouping of these three components make up the test automation solution which subsequently allows automation test control.

Each component of the framework has its unique purpose as each answer one of the basic questions of where, what and how.



Automation Test Control

The object identifier answers the ‘where’ question. The object identifier explains where the interaction should take place. This can be a user interface component or object (like a textbox) or hosting service for an API call. For selenium, object identifiers are implemented as locators like XPath, CSS selector etc.

The test data component answers the ‘what’ question. Test data describes what piece of information to use for testing. It can be information to enter on a textbox or simply parameters to use for an API call. Some simple automation interactions may not necessarily need test data. For example, a click interaction requires only an object identifier component.

The automation engine answers the ‘how’ question. The engine describes how the interaction required in the test script would be carried out on the SUT. For example, a click interaction on an object would be treated as an API call from the Selenium engine towards the browser (WebDriver) to perform the click on the component specified in the object identifier.

Consider the below two cases of tests.

Automation Component	Scenario 1	Scenario 2
	Enter “Username” in a textbox on webpage	Doing a REST API call to do a payment
Object Identifier	XPath for the textbox	The host and service to handle the payment request
Test Data	Username to be entered	The user account details
Automation Engine	Selenium Automation Engine	Service protocols to execute the request. E.g., SOAP

In scenario 1, the user is required to enter a specified text, in this case, the username, into a textbox found on a web page, described by the XPath. The interaction will be handled by the Selenium Automation Engine.

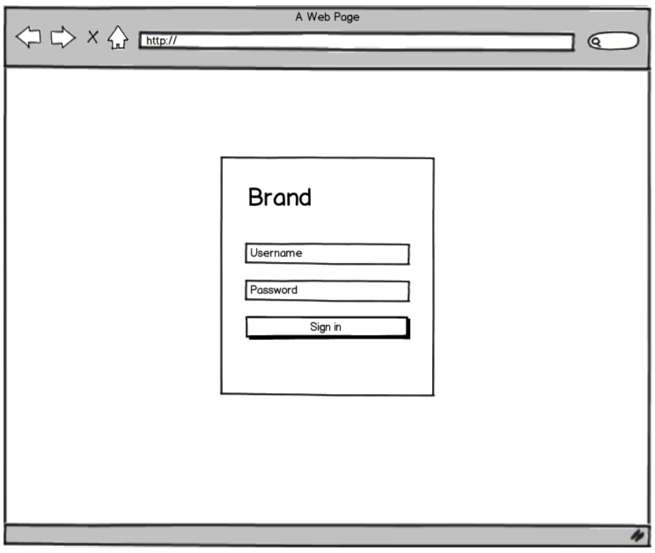
In scenario 2, the user is doing an API call towards a payment service. The object identifier in this case will be the host name where the service is deployed, and the name of the service being called. For the API call to be authorized, some parameters such as the user authorization token or the user credentials will be required. This constitutes the test data aspect of the automation test. The API call will rely on a protocol that defines how the call will be done for example, the SOAP. The protocol can thus be termed as the automation engine.

1.3.1 Page pattern automation solution

The arrangement of the object identifier, test data and the automation engine define the design of the automation solution. In a page pattern automation solution, the components are bundled together in the automation script.

To explain this, consider the following example:

Below is a classic login webpage where the user is required to enter a username and a password to login.



When automating the login action of the user, we can create a class in which we will enter the username, the password and then click on Sign In button.
The automation test script will look something like this:

```
public class LoginPage
{
        public void Login()
        {
                driver.findElement(By.name("Username")).sendKeys("Lovelesh");
                driver.findElement(By.name("Password")).sendKeys("secret");
                driver.findElement(By.name("Sign-In")).click();
        }
}
```

The page (e.g., LoginPage) on which the automation interaction happens is the class name and the individual set of interactions (e.g. Login) on the pages becomes a method. If we have the feature of forget password under the login page, then that set of interaction would be coded as a method, say ForgetPassword under the class LoginPage.

Given that the codes follow the pattern of the webpage, this type of automation framework is termed as page pattern automation framework.

The main advantage of opting for a page pattern automation framework is the speed with which the test can be scripted. Given that every component of the automation framework is set together, it is faster and easier to get a working automation test.

Nevertheless, this framework comes with some limitations. The main one being that this framework is highly dependent on the developer when it comes to test maintenance. If ever the test data need to be maintained (e.g., change in the username or password) or the object identifier changes due to changes or evolution of the website, then the developer will need to dive into the codes to locate where and what need to be changed. The maintenance therefore requires some technical and coding skills. An added inconvenience is that test data updates and object identifier updates are regarded as a code change and therefore each time maintenance is done, the automation scripts need to be re-compiled or re-interpreted for execution.

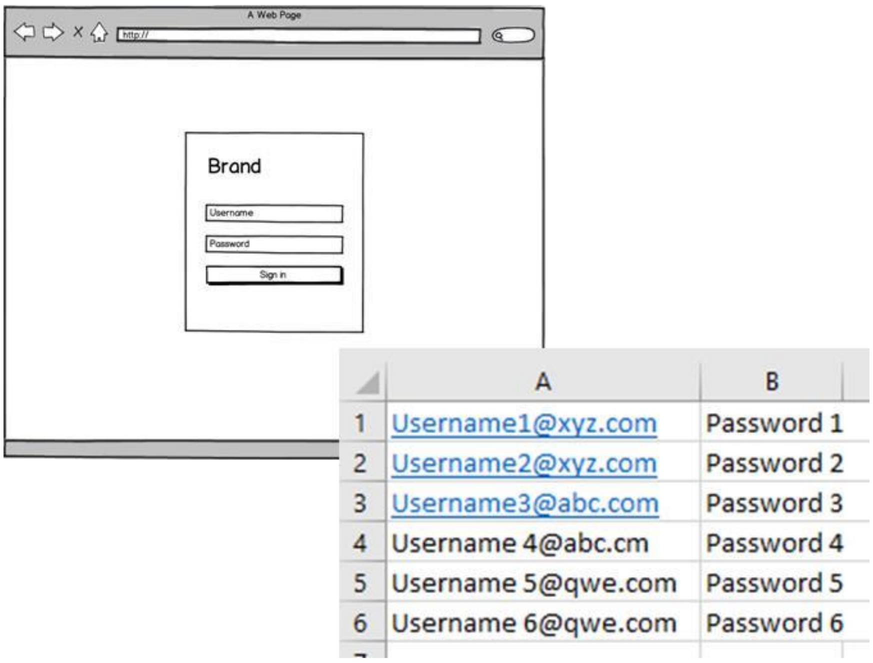
Also, it is to be noted that each page is a class as such when the system under test grows, the automation scripts grow accordingly which subsequently adds additional load on the test automation engineer together with the maintenance.

1.3.2 Data driven automation solution

A data driven automation solution is an extension of a page pattern automation solution. The concept is to externalize or remove the test data from the code of the automation scripts. This way, the test data maintenance load is removed from the automation test engineer as the test data can then be maintained by someone else who does not necessarily need test automation skills.

The test data can be externalized to flat files or into a spreadsheet in various formats (CSV, XML etc.).

To explain data driven automation solution, consider the following example:



	A	B
1	Username1@xyz.com	Password 1
2	Username2@xyz.com	Password 2
3	Username3@abc.com	Password 3
4	Username 4@abc.cm	Password 4
5	Username 5@qwe.com	Password 5
6	Username 6@qwe.com	Password 6

```
public class LoginPage
{
        public void Login()
        {
                UN = Read UserNameFromExcel();
                PW = Read Password from Excel();
                driver.findElement(By.name("Username")).sendKeys(UN);
                driver.findElement(By.name("Password")).sendKeys(PW);
                driver.findElement(By.name("Sign-In")).click();
        }
}
```

In the above example, the test data (the usernames and the passwords) is read from the external spreadsheet file. Each time there is a change in the credentials of the users, the data in the file can be amended without the need for a code change and therefore the code does not need to be recompiled.

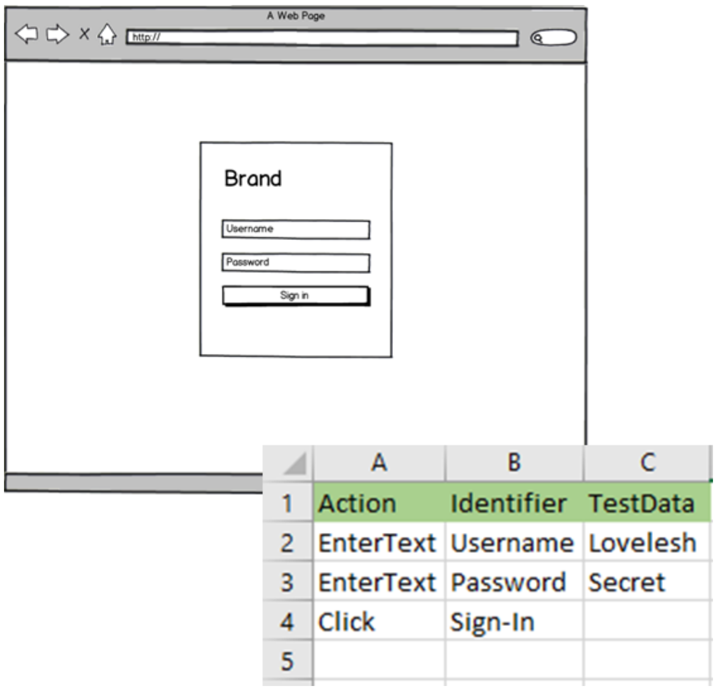
Nevertheless, when the object identifier needs to be updated, the code does need to be changed. Therefore, not all maintenance loads are removed from the automation engineer. Given that a data driven automation solution is thus an extension of a page pattern automation solution, the latter suffers from similar drawbacks. When the SUT grows the framework scripts and the test data stored grow as well.

In practice, when this solution design is used, the automation test engineer will be responsible for the scripting of automation scripts while the manual test engineer will support the automation effort by owning the test data maintenance.

1.3.3 Keyword driven automation solution

A keyword driven automation solution is another TAS design pattern that focusses on code re-use and ease of test maintenance. Like data driven automation design, the test data is kept external to the TAS, but the design goes one step further by also keeping the test cases separate from the TAS source code. The test case may include the object identifiers, the actions or interaction that needs to be carried out and the test data needed to carry out the test.

To explain keyword driven automation design, consider the following example:



	A	B	C
1	Action	Identifier	TestData
2	EnterText	Username	Lovelesh
3	EnterText	Password	Secret
4	Click	Sign-In	
5			

```
public class ExecuteTest
{
        public void PerformAction( identifier,action,testdata)
        {
                switch (action)
                {
                case (EnterText)
                {
                driver.findElement(By.name(identifier)).sendKeys(testdata);
                }
                case (Click)
                {
                driver.findElement(By.name(identifier)).click();
                }
        }
}
```

Considering this example, the external file used is a spreadsheet storing the test case and the test date. To login, we need to enter the username, enter the password, and click on login. Therefore, the action to enter a text in a textbox is to be repeated twice. In a page pattern automation design, the action to enter text is coded only once. There is a selection construct to invoke the needed interaction. In this example, it is structured as a case construct based on the action that is scripted in the externalized file. In this case, the actions used are called unit keywords because they cannot be broken down to other simpler actions. For example, we cannot break the action click to simpler interactions with the system.

Alternatively, we can combine multiple unit actions or keywords to form compound keywords. Compound keywords represent high level business interactions with a system that are meaningful for the test analyst. In this example, the unit actions of 'EnterText' and 'Click' can be combined to create the compound keyword 'Login'. The 'Login' action will encompass the action of entering the username, the password and clicking on the login button. This has the advantage of avoiding repetitive test steps and thus making the test case more compact, but it also takes away some flexibility from the test analyst in case the login process changes. If there is a change in the login process, then the compound action 'Login' will need to be maintained in the source code.
Hence, the key in this framework design is finding the right balance between the needed abstraction to make test scripting easy and keeping the maintenance flexibility. This TAS design derives its name from the fact that the "action" keyword drives the test execution.

The obvious and most compelling advantage of this framework is that the whole test scripting can be done by anyone without technical expertise after the TAE has coded the keywords. Also, the test case is coherent and easy to follow as all components are linked to each other in one external file. The tedious task of designing the test cases can thus be taken off the shoulders of the TAE. This is a huge advantage as an increase in automation test coverage can be achieved relatively quickly as the manual test team can support this and everything is not centralized on the TAE. Also, with this framework, only the size of the test scripts (externalized files) increases when the number of test cases increases, the TAS size is unaffected. Maintenance of the tests does not require code re-compilation and therefore can be done independently.

This framework does nevertheless have some drawbacks. Given that the underlying architecture is more complex than a page pattern framework, the coding of this framework is more complex and therefore requires more time. The externalized files designs and streams need to be coded correctly so that it is maintainable on the long term. If these designs are not flexible and not extensible, then the framework may need to be coded repeatedly for each small change in the design. For this reason, the framework design requires careful design upfront adapted to the business need. This pre-coding task defines the success of the framework in the organization. It is also undeniable that those doing the test scripting do need some basic training on how to extract the object identifiers from the SUT in a standard way.

Version control is of the utmost importance in both data driven and page pattern automation frameworks, as both the externalized file and the framework need to work hand in hand. If wrongly version controlled, test cases, test data and the test automation solution source code changes can be lost or overwritten.

A keyword driven automation test framework does require foresight from the test automation engineers as its success is heavily dependent on the flexibility and extensibility of the design. Therefore, when implemented correctly, this framework design brings a quick return on investment.

2 Automation Web Technologies – 250 minutes

Keywords

Locators, document object model, XPath, CSS selector, HTML, XML, rendering, tree structure

Learning Objectives

STF2-1 (K2) Understand how different web technologies co-exist together

STF2-2 (K1) Remember the different locators used by Selenium

STF2-3 (K3) Use different selenium locators to find GUI elements

STF2-4 (K2) Understand the structure of a DOM tree

STF2-5 (K3) Apply XPath expression to locate elements

STF2-6 (K4) Analyze a DOM tree to distinguish the most appropriate locator to use

STF2-7 (K3) Apply locator semantics to locate malformed locator expressions

STF2-8 (K3) Apply best practices to convert increase reliability of locator expression

STF2-9 (K2) Understand how to use a CSS selector to search for a node

STF2-10 (K3) Execute a CSS selector expression to search for an HTML node(s)

2.1Webpage Architecture

Selenium tools work only on webapps and webpages. Understanding the underlying technology of webpages and web apps is very important for a TAE to efficiently automate tests.

2.1.1 HTML

HyperText Markup Language (HTML)

HTML is the language most widely used to develop web pages. Formerly, HTML was created with the intention of defining the structure of documents such as paragraphs, headings, lists, etc. to ease the sharing of scientific information amongst researchers.

An HTML document is a plain text file which contains elements that specify certain contextual meanings when the document is parsed. The elements work together to identify how a browser should render those parts of the document.

HTML Tags and Attributes

Tags and attributes are the basis of HTML. They work together but perform different tasks.

Tags are used to mark up the start of an element and are usually enclosed in angled brackets. For example, `<h1>.`

Most tags must be opened (`<h1>`) and closed (`</h1>`) to function.

Attributes contain the additional pieces of information about the element. Attributes are placed inside an opening tag.

Example:

`<img src="myCar.jpg" alt="A pic of my car.">`

In this example, the image source (src) and alt text(alt) are the attributes of the `<img>` tag.

Note:

1. Most tags must be opened (`<tag>`) and closed (`</tag>`) with the element information such as the title or text resting between the tags.
2. When using multiple tags, they must be closed in the order that they were opened. For example:
   `<strong><em>This is really important!</em></strong>`

HTML is somewhat flexible, allowing some variation in the way tags are used (e.g., some tags may not have a closing tag.) This has caused some browsers to render pages erratically. XML, which will be discussed below, is a language which is more restrictive than HTML, requiring that each page be "well formed" with every opening tag balanced with a closing tag. When written in a well-formed way (i.e., all open tags are matched with close tags), **HTML is a subset of XML**.

Basic HTML Tags:

Tag	Used for
<html> … </html>	Signifies root of HTML document
<!DOCTYPE>	Defines document type (not needed in HTML 5)
<head> … </head>	Definition and meta data for document
<body> … </body>	Defines main content for the document
<p> … </p>	Defines a paragraph
<br />	Inserts a single line break
<div> … </div>	Defines a section in the document
<!-- … -->	Defines a comment (may be multi-line)

Heading tags (h1,h2,….,h6) define different levels of headers. The actual format of the text (size, boldness, font) can be specified in CSS stylesheets.

Links

`<a href=”URL”>Link Text</a>`

The anchor tag(<a …> … </a>) defines a hyperlink which can be clicked on. The **href**=”URL” is an attribute which signifies the destination of the link. The link text between the tags represents the text that will appear in the link to be clicked on to take the user to the URL target.

Images

`<img src="car.jpg" alt="The car" />`

The basic img tag defines an image that will be placed at this point in the document. The attribute src=”car.jpg” is the link address of the actual image that will be shown. The other attribute, alt=“The car”, represents the text that will be shown if the image cannot be found or displayed.

List and tables

Tag	Used for
<ul> … </ul>	Defines an unordered (bulleted) list
<ol> … </ol>	Defines an ordered (numbered) list
<li> … </li>	Defines a list item (for <ul> or <ol>)
<table> … </table>	Defines an HTML table
<tr> … </tr>	Defines a table row
<th> … </th>	Defines the column header for a table
<td> … </td>	Defines a table data cell
<tbody> … </tbody>	Groups body content in an HTML table
<thead> … </thead>	Defines an HTML table header
<tfoot> … </tfoot>	Defines an HTML table footer
<colgroup> … </colgroup>	Groups table columns for formatting

HTML Forms

These are used to gather input from users. Below are tags needed to render the forms and the controls on them.

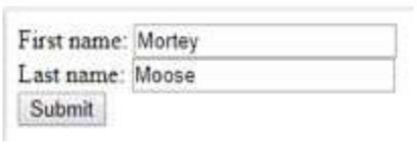
`<form> … </form>`:

Defines an HTML form for user input

`<input>`:

Defines an input control. The type of control is defined by the attribute type. Possible types include text, radio button, checkbox, submit, file, etc.

```
<form action="/action_page.php">
First name: <input type="text" name="FirstName" value="Mortey"><br>
Last name: <input type="text" name="LastName" value="Moose"><br>
<input type="submit" value="Submit">
</form>
```

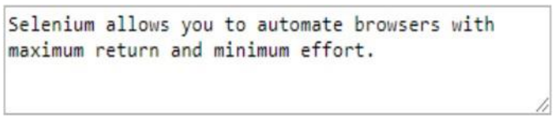


First name:	Mortey
Last name:	Moose

`<textarea> … </textarea>`:

Defines a multiline input control. The text area can hold an unlimited number of characters.

```
<textarea rows="4" cols="50">
Selenium allows you to automate browsers with
maximum return and minimum effort.
</textarea>
```


Selenium allows you to automate browsers with maximum return and minimum effort.

`<button> … </button>`:

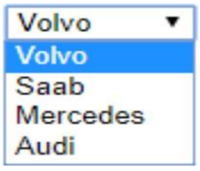
Defines a clickable button.

<button type="submit" class="button-native" part="native</button>

`<select> … </select>`:

Defines a drop-down list. When used with the `<option> … </option>` tag, the author can define a drop-down list and populate it as follows:

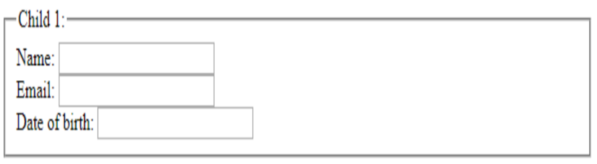
```
<select>
  <option value="volvo">Volvo</option>
  <option value="saab" >Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```



`<fieldset> … </fieldset>`:

These tags allow the author to group related elements into a form. When used with the tag,`<legend>`, it draws a named box around controls which are deemed to be related as shown:

```
<fieldset>
   <legend>Child 1:</legend>
     Name: <input type="text"><br>
     Email: <input type="text"><br>
     Date of birth: <input type="text">
 </fieldset>
```



2.1.2 XML

Extensible Markup Language (XML)

XML is a simple text-based format for representing structured information. It is a markup language that is used to define rules for formatting documents in a way that is machine readable but also highly readable by humans.

XML is very similar to HTML; in fact, we can say that HTML is a subset of XML. However, the syntax rules of XML are strict: XML tools will not process files that contain errors. HTML was designed to display data with a specific focus on how the data looks. XML was designed to be a software and hardware-independent tool which can be used to transport and store data in a legible format.

Unlike HTML tags, XML Tags are not predefined. Instead, tags are specified by the author of the XML document. The format of the tags is much like HTML. For example, here is a set of fields in XML:

```
<?xml version="1.0" ?>
<note>
      <date>2018-06-12</date>
      <hour>10:30</hour>
       <to>Francis</to>
       <from>Morrow</from>
       <body>Please pick me up this weekend!</body>
</note>
```

Note that each opening tag, <date>, has a matching closing tag, </date>. The total construct is called an **element**.

Sections can be embedded in other sections as shown in the example above. An XML document always forms a tree structure. The first element in the above figure shows that this is an XML document. In addition to tags, XML supports attributes which supply extra information about the element they are associated with. An attribute consists of a pair of terms separated by an equal sign.
For example:

`<person gender=”Male”>`

The attribute gender is inside the opening tag. Instead of using an attribute, the same information can be stored as an element. For example, the 2 XML contains the exact same information:

```
<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

Attributes are not as flexible as elements. For example, the following points are stressed by W3C, the group that controls the XML standard:

- Attributes cannot contain multiple values (elements can)
- Attributes cannot contain tree structures (elements can)
- Attributes are not easily expandable (for future changes)

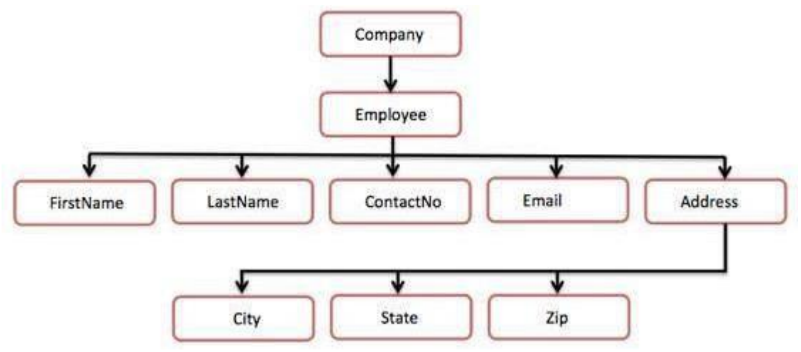
2.1.3 Tree structure

Both HTML and XML use the **tree structure,** and this plays an important role to describe any XML document easily. The use of this structure allows automation engineers to know all the succeeding branches and sub-branches starting from the root. The navigation starts at the root, then moves down the first branch to an element, takes the first branch from there, and so on to the leaf nodes.

Consider the following XML:

```xml
<?xml version = "1.0"?>
<Company>
   <Employee>
      <FirstName>James</FirstName>
      <LastName>Bond</LastName>
      <ContactNo>1234567890</ContactNo>
      <Email>Homesh@xyz.com</Email>
      <Address>
         <City>Qwerty</City>
         <State>Mauritius</State>
         <Zip>560212</Zip>
      </Address>
   </Employee>
</Company>
```

The following diagram represents the above XML Document:



Relationship between nodes

Since XML always describes a tree format, we can define specific relationships between elements. These relationships can be defined as follows:

- The **current node** is any arbitrary node that we choose. All other relationships derive from the current node.
- Each node can identify itself as **self**.
- A **parent node** is always one level higher in the hierarchy than the node selected as current. Each element node and attribute has exactly one parent (except for the root element.)
- A **child node** is always one level below its parent in the hierarchy.
- A **sibling node** is on the same level as the current node, under the same parent.
- An **ancestor node** is one in a direct path from the current node up through its parent, grandparent, great-grandparent, etc.
- A **descendent node** is one in a direct path down from the current node in the hierarchy (i.e., a child, a child of a child, etc.)

2.1.4 Rendering

When we think of rendering, we think of beautification, and this is what is truly done in this stage of processing of a webpage. While HTML defines the skeleton of the webpage, the structure is crude and does not provide a good user experience (UX). In order to make the use of the page appealing to the users, the rendering technology applies a layer that takes the crude structure in the HTML and applies colors, fonts and animations to the page objects. This layer of technology thus beautifies the HTML by adding flesh to the skeleton of the page.

CSS is the most famous rendering layer for webpages. CSS can target rendering processing based on the object type or other attributes which makes it very efficient for this purpose. However, care should be taken that the CSS rendering does not add too much latency to the load time of the webpage.

2.1.5 Functional logic

Having a beautified webpage without any functional logic does not provide much value for the end user unless it is only conveying information. Nevertheless, these static webpages are quite rare on the internet nowadays. Most pages on the web are now expected to have some basic functionalities and most of these functionalities require some processing to be done on the browser level on the user side, which is termed as client-side processing. In other cases, some transactions may require some processing to be done on the server side, which is termed as server-side processing.

For example, let's consider a user trying to login on a webpage. If no user credentials are entered and the user hits on login, an error message is to be displayed stating that a username and password is required to be able to login. This type of processing will be done on the user browser level and does not require any server intervention for validation. This is an example of client-side processing.

However, if user credentials are entered and the user hits on login, the credentials will need to be validated against the database hosted on the server. This is therefore a server-side processing transaction.

JavaScript and Angular are popular scripting languages that implement these functional logics.

As a whole, if we combine the structure, rendering and functionalities together in an analogy, we can think of the structure as the bricks of a house. The rendering will be the paint on the walls and the functional logic will be lights and fittings, etc. All the three components make it comfortable to be in the house. Similarly, all the three components are needed to make a webpage appealing and of some value to the end user.

2.2 Document Object Model

The Document Object Model (DOM) is a **programming interface** for **XML** and **HTML** documents. It states the **logical structure** of the document and the way it is accessed and manipulated.

A DOM is a means to represent the webpage in a structured hierarchical way in order to make it easier for users (programmers, etc.) to navigate through the document. With a DOM, we can easily access and manipulate IDs, tags, attributes, classes, or elements of HTML using commands or methods provided by the document object. Using a DOM, the JavaScript gets access to HTML as well as CSS of the web page and can also add behavior to the HTML elements.

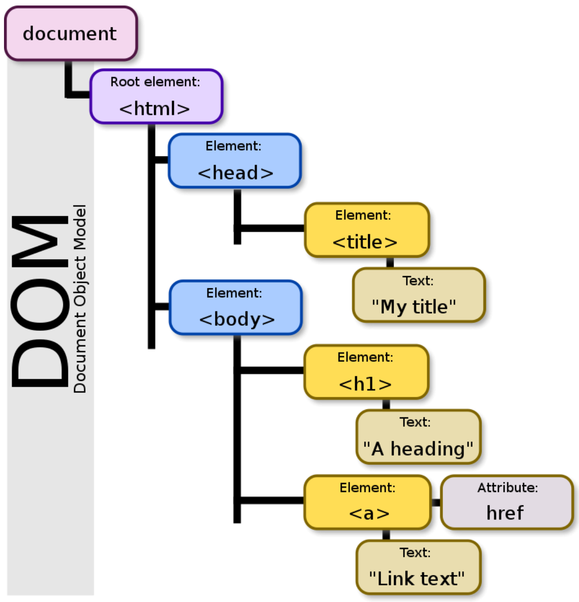
Why is a DOM required?

HTML structures a webpage and JavaScript adds behavior to it. When an HTML document is loaded into a browser, the JavaScript cannot understand the HTML directly. Therefore, a corresponding document is created (a DOM). So basically, a DOM represents the same HTML document in a different format using objects.

2.2.1 Structure of a DOM

Structure of a DOM

A DOM can be imagined as a tree or forest (more than one tree). A **Structural model** is more often used to describe the tree-like representation of a document. Each branch of the tree ends in a node and each node contains objects. Event listeners can be added to nodes that are triggered on the occurrence of a given event.



DOM Nodes

Everything in an HTML document is a node:

- The entire document is a document node
- Every HTML element is an element node
- The text inside HTML elements are text nodes
- Every HTML attribute is an attribute node (deprecated)
- All comments are comment nodes

Node Relationships

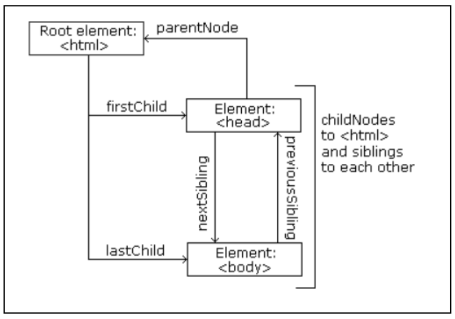
All the nodes in the tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships.

- In a node tree, the top node is called the root (or root node)
- Every node has exactly one parent, except the root (which has no parent)
- A node can have a number of children
- Siblings (brothers or sisters) are nodes with the same parent

Consider the following example:

```
<html>
  <head>
    <title>A4Q DOM Example</title>
  </head>
  <body>
    <h1>DOM Lesson one</h1>
    <p>Hello world!</p>
  </body>
</html>
```



From the HTML above, it can be concluded that:

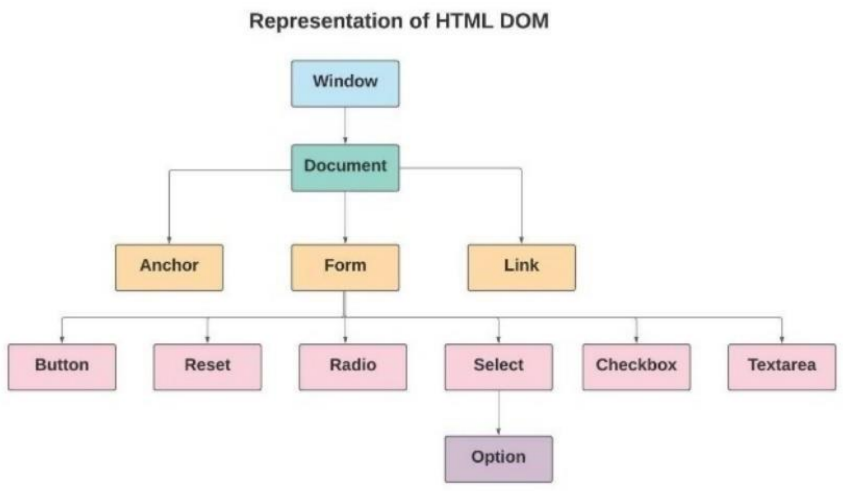
- `<html>` is the root node
- `<html>` has no parents
- `<html>` is the parent of `<head>` and `<body>`
- `<head>` is the first child of `<html>`
- `<body>` is the last child of `<html>`

and:

- `<head>` has one child: `<title>`
- `<title>` has one child (a text node): " `A4Q DOM Example` "
- `<body>` has two children: `<h1>` and `<p>`
- `<h1>` has one child: "DOM Lesson one"
- `<p>` has one child: "Hello world!"
- `<h1>` and `<p>` are siblings

2.2.2 Properties of DOM

The properties of the document object that can be accessed and modified by the document object are as below.



- **Window Object:** The “Window Object” at the top of the DOM hierarchy.
- **Document Object:** When the HTML document is loaded in a browser, it is converted into a document object.
- **Anchor Object:** href tags are utilized for representing the Anchor Objects.
- **Form Object:** form tags are utilized for representing the Form Objects.
- **Link Object:** link tags are utilized for representing the Link Objects.
- **Form Control Elements**: Forms can also have other control elements such as buttons, reset, radio buttons, textarea, checkboxes, etc.

Another important property of a DOM is “**structural isomorphism**”. It states that if any two DOM implementations are used to create a representation of the same document, they will create the same structure model, with precisely the same objects and relationships.

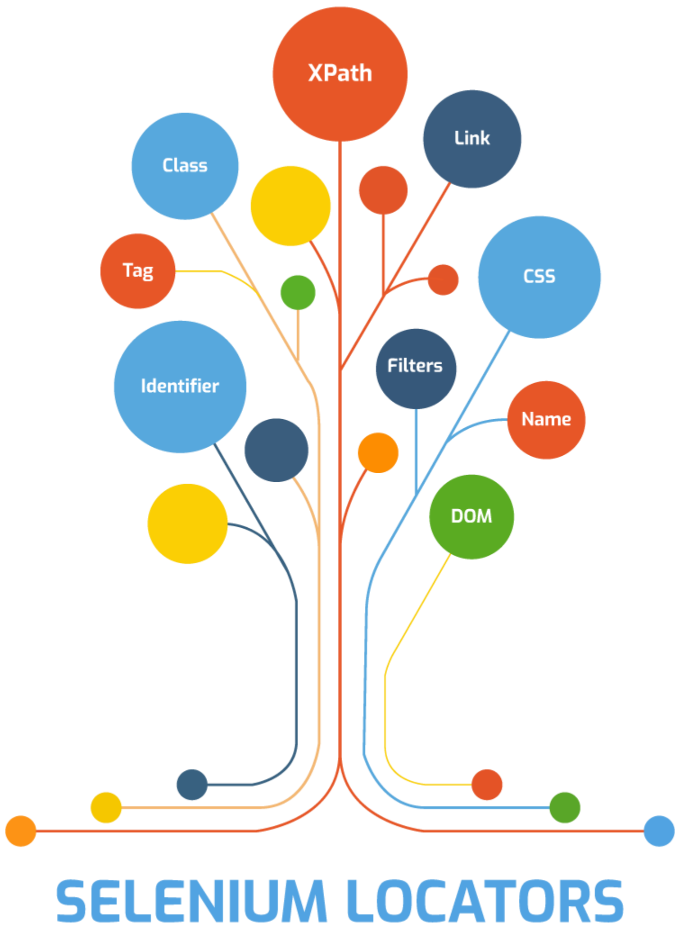
2.3 Automation locators

Locators are used to identify an *HTML* element on a web page, and most UI automation tools provide the capability to use locators for the identification of *HTML* elements on a web page. Similarly, Selenium also possesses the ability to use "***Locators***" for the identification of HTML elements and are popularly known as "***Selenium Locators***".

Identification of the correct GUI element on a web page is pre-requisite for creating any successful automation script. It is where the locators come into the picture. ***Locators*** are one of the essential components of Selenium infrastructure, which help Selenium scripts in ***uniquely identifying the web elements*** (*such as text box, button, etc*.) present of the web page.

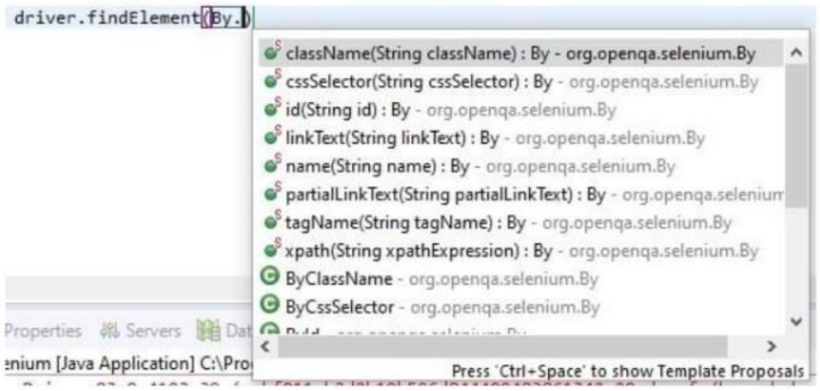
2.3.1 Types of locators

There are various types of locators which we can use to identify a web element uniquely on the webpage. The following figure shows a good depiction of several types of locators that Selenium supports.



To use these locators, Selenium provides the **By** class, which locates elements within the DOM. It offers several different methods (some of which are in the image below) like **className, cssSelector, id, linkText, name, partialLinkText, tagName, and XPath, etc**., which can identify the web elements based on their corresponding locator strategies.

The possible locators supported by Selenium is shown in the visible methods under the **By** class below:



```
driver.findElement(By.)
```

```
className(String className) : By - org.openqa.selenium.By
cssSelector(String cssSelector) : By - org.openqa.selenium.By
id(String id) : By - org.openqa.selenium.By
linkText(String linkText) : By - org.openqa.selenium.By
name(String name) : By - org.openqa.selenium.By
partialLinkText(String partialLinkText) : By - org.openqa.seleniur
tagName(String tagName) : By - org.openqa.selenium.By
xpath(String xpathExpression) : By - org.openqa.selenium.By
ByClassName - org.openqa.selenium.By
ByCssSelector - org.openqa.selenium.By
Press 'Ctrl+Space' to show Template Proposals
```

Selenium supports the following locators:

- **ClassName** – A class name operator uses a class attribute to identify an object.
- **cssSelector** – CSS is used to create style rules for webpages and can be used to identify any web element.
- **id** – Similar to class, the ‘id’ attribute can be used to identify elements
- **linkText** – Text used in hyperlinks can also locate element
- **name** – Name attribute can also identify an element
- **partialLinkText** – Part of the text in the link can also identify an element
- **tagName** – We can also use a tag to locate elements
- **XPath** – XPath is the language used to query the XML document. The same can uniquely identify the web element on any page

2.4 XPath

For automation to be possible, the automator must be able to locate objects or elements in an XML document. One of the most powerful ways to do so is to use **XPath**.

XPath stands for XML Path Language. It uses a non-XML syntax to provide a flexible way to navigate and identify nodes in an XML document. Since HTML is a subset of XML, XPath can also be used to search HTML documents. XPath uses a path notation (as in URLs) for navigating through the hierarchical structure of an XML document.

The following expressions will select nodes:

Expression	Description
TheNode	Selects all elements with name “TheNode”
/	Selects from the root element
//	Returns descendants of the current element
.	Return the current element
..	Selects the parent of the current element
@	Selects an attribute of the current element

Below is a sample XML document and some examples of XPath usage.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Movies>
    <Movie>
        <title lang="en">The Avengers</title>
        <Date>
            <Day>20</Day>
            <Month>10</Month>
            <Year>2012</Year>
        </Date>
        <genre>Adventure/Action</genre>
        <rating>9.8</rating>

    </Movie>

    <Movie>
        <title lang="fr">Justice League</title>
        <Date>
            <Day>17</Day>
            <Month>8</Month>
            <Year>2017</Year>
        </Date>
        <genre>Adventure/Thriller</genre>
        <rating>9.3</rating>
    </Movie>
</Movies>
```

Below are some path expressions and the results from them.

Path Expression	Result
Movies	Selects all nodes with the name “Movies”
/Movies	Selects the root element “Movies”
/Movies/Movie	Selects all Movie elements of Movies
//Movies	Selects all Movie elements in the document
Movies//Movie	Selects all Movie elements that are descendants of Movies
@lang	Selects all attributes named lang

Predicates are expressions used to filter nodes selected by a particular path. Predicates are always surrounded by square brackets and give specific information(s) about the element.

Path Expression	Result
/Movies/Movie[1]	Selects the first Movie element
/Movies/Movie[last()]	Selects the last child Movie element
/Movies/Movie[last()-1]	Selects the second to last Movie element
//title[@lang]	Selects all title elements with attribute “lang”
//title[@lang=’en’]	Selects all title elements with attribute lang=en

XPATH allows the use of **wildcards** to write more robust path expressions where the use of specific path expressions is either impossible or undesirable

Wildcard	Description
*****	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

2.4.1 XPath operators

XPath defines operators and functions on nodes. An XPath expression returns either a node-set, a string, a Boolean, or a number.

XPath operators can in different categories according to their property. Following are the different types of XPath operators:

1. Comparison Operators

Comparison operators are used to compare values.

Operator	Description
=	It specifies equals to
!=	It specifies not equals to
<	It specifies less than
>	It specifies greater than
<=	It specifies less than or equals to
>=	It specifies greater than or equals to

2. Boolean Operators

Boolean operators are used to check 'and', 'or' & 'not' functionalities.

Operator	Description
and	It specifies that both conditions must be satisfied.
or	It specifies that any one of the conditions must be satisfied.
not()	It specifies function to check condition not to be satisfied.

3. Numeric Functions/Operators

A list of **number operators** that are used with XPath expressions:

Operator	Description
+	It is used for addition operation.
-	It is used for subtraction operation.
*****	It is used for multiplication operation.
div	It is used for division operation.
mod	It is used for modulo operation.

A list of **functions on numbers** that are used with XPath expressions:

Functions	Description
ceiling()	It is used to return the smallest integer larger than the value provided.
floor()	It is used to return the largest integer smaller than the value provided.
round()	It is used to return the rounded value to nearest integer.
sum()	It is used to return the sum of two numbers.

4. String Functions

A list of XPath string functions:

Functions	Description
starts-with(str1, str2)	It returns true when str1 starts with the str2.
contains(str1, str2)	It returns true when the str1 contains the str2.
substring(str, offset, length?)	It returns a section of the string. The section starts at offset up to the length provided.
substring-before(str1, str2)	It returns the part of str1 up before the first occurrence of str2.
substring-after(str1, str2)	It returns the part of str1 after the first occurrence of str2.
string-length(string)	It returns the length of string in terms of characters.
normalize-space(string)	It trims the leading and trailing space from string.
translate(str1, str2, str3)	It returns str1 after any matching characters in str2 have been replaced by the characters in str3.
concat(str1, str2, ...)	It is used to concatenate all strings.
format-number(number1, str1, str2)	It returns a formatted version of number1 after applying str1 as a format string. Str2 is an optional locale string.

5. Node Functions

A list of functions on nodes to be used with the XPath expression:

Functions	Description
node()	It is used to select all kinds of nodes.
processing-instruction()	It is used to select nodes which are processing instruction.
text()	It is used to select a text node.
name()	It is used to provide the name of the node.
position()	It is used to provide the position of the node.
last()	It is used to select the last node relative to current node;
comment()	It is used to select nodes which are comments.

2.4.2 XPath axes

As location path defines the location of a node using either a relative or absolute path, **axes** are used to identify elements by their relationship like **parent, child, sibling**, etc. Axes are named as such because they refer to axis on which elements are lying relative to an element.

Below is a list of the different axes:

Axis Name	Result
ancestor	These axes indicate all the ancestors relative to the context node, also reaching up to the root node.
ancestor-or-self	This one indicates the context node and all the ancestors relative to the context node and includes the root node.
attribute	This indicates the attributes of the context node. It can be represented with the “@” symbol.
child	This indicates the children of the context node.
descendant	This indicates the children, grandchildren, and their children (if any) of the context node. This does NOT indicate the Attribute and Namespace.
descendant-or-self	This indicates the context node and the children, and grandchildren and their children (if any) of the context node. This does NOT indicate the attribute and namespace.
following	This indicates all the nodes that appear **after** the context node in the HTML DOM structure. This does NOT indicate descendent, attribute, and namespace.
following-sibling	This one indicates all the sibling nodes (same parent as the context node) that **appear** after the context node in the HTML DOM structure. This does NOT indicate descendent, attribute, and namespace.
namespace	This indicates all the namespace nodes of the context node.
parent	This indicates the parent of the context node.
preceding	This indicates all the nodes that appear **before** the context node in the HTML DOM structure. This does NOT indicate descendent, attribute, and namespace.
preceding-sibling	his one indicates all the sibling nodes (same parent as context node) that appear **before** the context node in the HTML DOM structure. This does NOT indicate descendent, attribute, and namespace.
self	This one indicates the context node.

XPath traversal

XPath traversal refers to the navigation between the different nodes in a DOM. Navigation is possible downward (parent to child node), upward (child to parent) or even among siblings with the appropriate use and understanding of the relationship between them.

2.4.3 Absolute and relative XPath

There are two types of XPaths:

1. Absolute XPath
2. Relative XPath

Absolute XPath

The absolute XPath has the complete path starting from the root to the element which we want to identify. The key characteristic of XPath is that it begins with the single forward slash (/).

The major drawback of an absolute XPath is that if there is a change in any node from the root to the expected element, the XPATH will no longer be valid.

```
XPathExpression: /Movies/Movie[1]/Date/Day[1]/text()
```

```
Result:
           20
```

Relative XPath

A relative XPath can start anywhere in the HTML DOM structure or even refer directly to the element that we want to identify.

A relative XPath starts with the // symbol. It is mainly used for automation since even if an element is removed or added in the DOM, the relative XPath is not impacted.

An absolute XPath is lengthy and difficult to maintain (html/body/tagname/…). While a relative XPath is short (//*[@attribute='value']).

```
XPathExpression: //*[text()='20']
```

```
Result:
              20
```

2.5 CSS Selector

Cascading Style Sheets(CSS)

CSS is a language used to describe the look and feel (presentation) of a document (HTML/XML). CSS specifies how the elements will be rendered on screen (or other media). That is, HTML is a markup language and CSS is a style sheet language. While HTML and CSS give an author very powerful tools for displaying materials, most experts do not consider them as real programming languages.

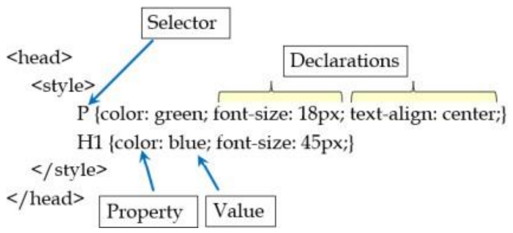
As far as Selenium testing is concerned, CSS is very useful when finding elements for test automation. CSS can be used in HTML Documents in 3 different ways:

1. An external style sheet: each HTML page must include a reference to the external style sheet file inside the element which goes inside the section.
2. An internal style sheet: when a single HTML page is to have a unique style; the styles are defined within the section of the document.
3. An inline style: applies to a specific element and is added directly to the element as an attribute.

When the same element's style is defined by multiple CSS Style, the value from the last read stylesheet will be used. Therefore, the order a style will be used will be defined (starting at the top) as follows:

1. Inline style (as an attribute inside an HTML element)
2. External and internal style sheets defined in the head section
3. The browser default value

A CSS ruleset consists of selector(s) and declaration block(s) as follows:



```
<head>
  <style>
     P {color: green; font-size: 18px; text-align: center;}
     H1 {color: blue; font-size: 45px;}
  </style>
</head>
```

Each selector refers to an HTML element to style. The declaration blocks consist of one or more declarations separated by semicolons (;). Each declaration includes a CSS property name and a value separated by a colon (:). Declaration blocks are surrounded by curly braces.

CSS selectors are used to find elements in the HTML document based on element name, ID, class, attribute, or other specifiers. Below is a table showing the differences between XPath and CSS selector:

CSS Selector	XPath	Result
div.even	**//div[@class=”even”]**	Elements div with an attribute class=”even”
#login	**//*[@id=”login”]**	An element with id=”login”
*****	**//***	All elements
Input	**//input**	All input elements
p,div	**//div//input**	All p and all div elements
div input	**//div/input**	All input elements inside all div elements
div>input		All input elements that have the div element as the parent
br + p		Selects all p elements that are placed immediately after element br
p ~ br		Selects all p elements that are placed immediately before element br

CSS selectors also work with attributes as follows:

CSS	Result
[lang]	All elements with the lang attribute
[lang=en]	All elements with the lang attribute of exactly en
[lang^=en]	All elements with the lang attribute starting with the string en
[lang | =en]	All elements that have the lang attribute equal to en or starting with the string en followed by a hyphen
[lang$=en]	All elements that have the lang attribute ending with the string en
[lang~=en]	All elements that have the lang attribute whose value is a whitespace-separated list of words, one of which is exactly the string en
[lang*=en]	All elements that have lang attribute containing string en

When dealing with form elements, there are a variety of CSS selectors available:

CSS	Result
:checked	Selects all checked elements (for check boxes, radio buttons, and options that are toggled to an on state
:default	Selects any form element that is the default among a group of related elements
:defined	Selects all elements that have been defined
:disabled	Selects all elements that have been disabled
:enabled	Selects all elements that have been enabled
:focus	Selects the element currently with focus
:invalid	Selects any form elements that fail to validate
:optional	Selects form elements which do not have required attribute set
:out-of-range	Selects any input elements whose current value is outside the min and max attributes
:read-only	Selects elements which are not editable by user
:read-write	Selects elements which are editable by user
:required	Selects form elements with required attribute set
:valid	Selects form elements that do validate successfully
:visited	Selects the links that a user has already visited

3 Selenium Automation Tools – 155 minutes

Keywords

Selenium WebDriver, Selenium IDE, Selenium Grid, architecture, browser controller, headless test automation, client, server, JSON Wire protocol

Learning Objectives

STF3-1 (K1) Remember the different Selenium frameworks and the supported languages

STF3-2 (K2) Understand the function of Selenium IDE, Selenium WebDriver, and Selenium Grid

STF3-3 (K2) Understand the architecture on which Selenium WebDriver 4 is built

STF3-4 (K2) Understand the concept and uses of headless automation

STF3-5 (K4) Distinguish the optimum parameters for Selenium automation tools given a scenario

STF3-6 (K3) Use appropriate browser controller commands in correct sequence given a scenario

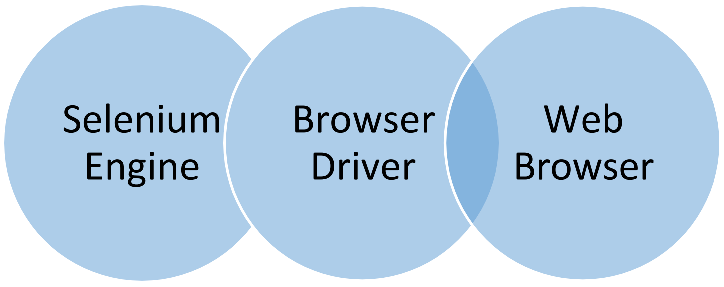
STF3-7 (K2) Understand the new features of Selenium 4

STF3-8 (K3) Apply the correct feature of Selenium 4 given a scenario

3.1 Overview of Selenium test automation

Selenium has evolved over the years to be one of the most preferred open-source automation tools on the market. Selenium is a suite of tools used to automate web applications or webpages. They have a very versatile library to allow versatile interactions with web objects. Selenium has solutions that allow record and playback of tests, building of automation frameworks, and also scaling of the tests to multiple nodes for execution.
These solutions have been instrumental in promoting Selenium to its current position on the market as these solutions development were driven by demands from the software quality industry.

Selenium interacts with the web browsers through APIs. This gives Selenium the cross-browser test execution ability. In other words, scripted test could be executed on multiple web browsers. Selenium interactions are directed to browser drivers which then interact with the real browser to carry out the action on the web browser.



Given that Selenium project is open source, the list of browser drivers can be developed and maintained by a community of passionate developers and testers. Therefore, it is highly probable that more drivers for new browsers will be developed in the near future.

3.1.1 Contemporary Selenium automation

The automation engine support both 32-bits and 64-bits machine instruction set. The Selenium engine execution is supported on Microsoft Windows, macOS and Linux operating systems. Selenium source code is still under an open-source project and the code is hosted on a GitHub repository.

The Selenium project has language bindings to sustain scripting on more programing languages than ever.
The core language bindings that allow scripting on the Selenium engine include the following programing languages:

1) Java
2) Python
3) C#
4) Ruby
5) JavaScript

Other bindings also exist to support Selenium non-core programming languages such as Go, Haskell, Perl, PHP, R, Dart and Pharo.

Selenium supports test execution on the most popular browsers such as Mozilla Firefox, Microsoft Edge, Google Chrome, Opera and Safari.
Execution on these browsers requires browser drivers as listed below for the Selenium project:

1) Mozilla GeckoDriver
2) Microsoft EdgeDriver

3) Google ChromeDriver
4) Opera ChromiumDriver
5) Apple SafariDriver

Selenium, when used for testing web applications, needs some testing frameworks. The Selenium engine, as a standalone tool, only exposes its native APIs as methods or functions to the TAE. The TAE must then program a testing framework to use these APIs to carry out the tests required. A testing framework can be conceived from scratch or existing frameworks could be used. The Selenium project supports multiple testing frameworks in diverse languages.
Some are listed below:

Testing Framework	Supported Language
Capybara	Ruby
CodeceptJS	JavaScript
FluentLenium	Java
Helium	Python
Nerodia	Python
QAF	Java
Selenide	Java
SeleniumBase	Python
Watir	Ruby
WebDriverIO	JavaScript
Nightwatch.js	JavaScript

The language bindings, browser drivers and frameworks can be downloaded from www.selenium.dev.

3.1.2 Selenium: powers and limitations

The versatility of the language bindings, browser drivers, existing frameworks and the support from companies and community, makes Selenium a very powerful automation technology. Selenium automation comes in different flavors. For those who need a quick and easy way to script short lived tests, there is the record and playback tool and for those who need long term maintainable and flexible tests, there is the Selenium WebDriver. For those who need to scale up testing, there is the Selenium Grid.

The power of Selenium’s automation technology also resides in the large community of developers and automation test engineers contributing to the project. For most issues encountered, defects or simply help on the technology there are blogs and forums to seek support and the community is very active in helping to solve and resolve issues and requests. The existing testing frameworks also make it easy for TAEs to onboard automation using Selenium.

Nevertheless, Selenium remains a tool that cannot fulfill all the specific needs for all organizations. Therefore, for some organizations, there may be a need to develop a custom test automation framework. Development of a new test automation framework may require some in-depth programming knowledge and can be highly technical. The ROI for these projects is also questionable as it is highly dependent on the design for the test framework.

Also, Selenium satisfies a specific need in the automation world. It is meant only for the browsers. Often, the management in an IT organization tend to see Selenium as the ultimate tool for all automation needs and often instruct to conceive automation scripts that go beyond the browsers. For example, Selenium cannot natively manipulate local resources (moving files around or renaming files) on the machine. There are better tools, such as RPA for automation of such operations.

TAEs also need to be careful with updates on browsers. It may happen that browser updates require updated browser drivers as well. It is therefore important to consider the availability of these drivers before browsers are updated. This compatibility may sometimes be tricky to handle.

3.2 Flavors of the Selenium automation tool

As discussed in section 3.1, the Selenium automation solution is not a standalone tool, it is a suite of tools. It comes in three distinct forms, and each serves different purposes.

3.2.1 Selenium IDE

Selenium IDE is a browser extension or plugin that integrates with Google Chrome or Mozilla Firefox to allow the user to easily record manual steps done on the browser and play the recorded steps multiple times over and over again.

Selenium IDE is therefore suitable for short lived tests or automation needs. The simplicity of the tool compensates for its lack of extended flexibility. However, the latest version of Selenium IDE is more resilient for tests. This is because when an interaction with a web object occurs, multiple locators for the object is recorded and during playback each locator is attempted. There is therefore better resiliency in the scripted tests. There is also better re-use of existing test cases in the new Selenium IDE. This means that existing test cases could be called in other test cases without duplication. The re-use functionality therefore reduces the test maintenance effort as the called test cases are shared and not duplicated.

The new Selenium IDE has constructs that allow better control and execution flow. Controls like the if statement and looping structure come in very handy when we need to implement real life tests that are based on some logical flow. The extensibility of the new Selenium IDE is appealing as it allows custom commands to be written and third-party plugins to be integrated as well.

Nevertheless, if higher control flow, flexibility and extensibility is needed, the Selenium IDE may not be the best tool. Selenium WebDriver is better when it comes to these needs.

3.2.2 Selenium WebDriver

Selenium WebDriver, as the name suggests, interacts with the browser in the same way that a real user would. WebDriver is an end-to-end test compilation tool for web applications that is also frequently referred to as "Selenium WebDriver". Both the language bindings and the actual implementations of the unique browser controlling code are referred to as Selenium WebDriver. Before WebDriver, Selenium automation used to depend on Selenium Remote Control technology, most concisely referred to as Selenium RC. The Selenium RC uses an intermediate RC server to interact with the browser. On the other hand, WebDriver provides simpler and direct programming interfaces, which is built in a way that makes it easier to use than the Selenium RC 1.0 API. This program performs exactly what a user would anticipate from a browser: it automates browser control so that a user can repeat the automated actions. Although it appears to be a straightforward challenge to overcome, there are several steps that must be taken for it to function. The Selenium WebDriver is thus the interface in the TAS that allows interaction with the browser.
The Selenium WebDriver is a W3C recommendation which implies that it now has simple and concise programming interfaces based on compact object-oriented APIs to work as effectively as possible with web browsers.

The Selenium WebDriver is a major building block for a TAF. It runs the browser significantly more efficiently whether it is operated by a local user or a remote user. As a result, the functional test coverage constraints of the previous version of Selenium, such as those related to file uploads, downloads, pop-ups, and dialogue barriers, are removed. The language bindings and implementations of the separate browser controlling code have been merged into Selenium WebDriver. The Selenium WebDriver is thus entirely an object-oriented API. The implementation of WebDriver normally differs for each browser.

Numerous language bindings are supported by Selenium WebDriver and works with modern web browsers (see section 3.1.1).

From WebDriver architecture, it is known that WebDriver directly calls the browser when tests are performed in a native machine by using the browsers built-in JavaScript support for the automation unlike Selenium RC. A point to be noted is that in Selenium RC, the architecture works in a different way where it passes the HTTP requests to the server and the server passes it to the Selenium Core. Conversely, WebDriver does not have any proxy server in between the client and the browser.

In contrast to Selenium RC, The WebDriver API is a more potent tool for accelerating test execution. Prior to utilizing WebDriver, Selenium RC was unable to communicate with mobile applications or the browser's rich content API, making mobile browser-based tests completely impractical. Almost all the most recent browser versions on the market are supported by WebDriver. Officially, WebDriver is the sole platform that will be used for any future improvements.
Using Selenium WebDriver has benefits and limitations.
The advantages from which an organization may gain are as below:

- Selenium WebDriver is a collection of the crude API interfaces and can hence be adapted to the organization's need.
- The required adapters like database connections, webservices connect, etc. can be added to make the tests effective.
- The TAS based on Selenium WebDriver, if well designed, can be highly flexible on the long term.
- The availability of off-the-shelf TAS based on Selenium WebDriver gives a quicker implementation in organizations.

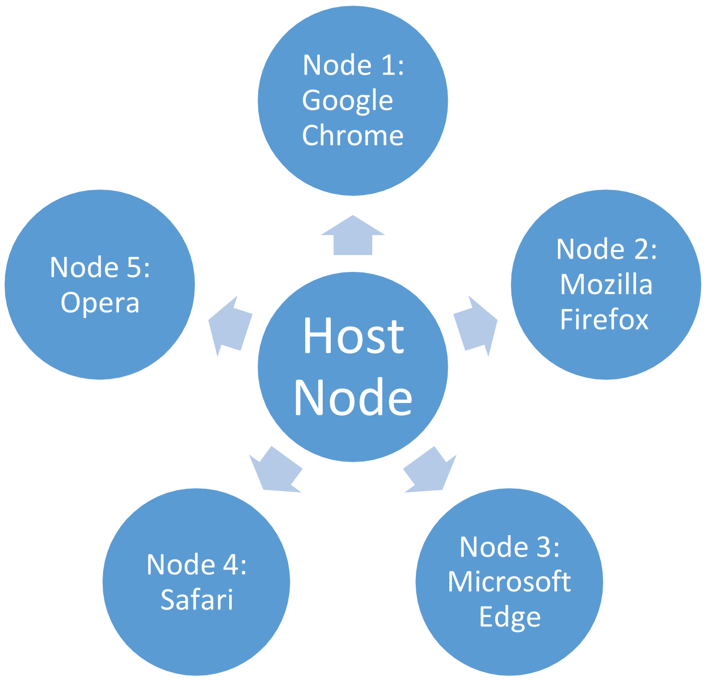
Drawback associated with using Selenium WebDriver include the following:

- Technical skills are required to make effective use of the Selenium WebDriver for a custom-built TAS.

3.2.3 Selenium Grid

Selenium Grid empowers the Selenium WebDriver to run tests on multiple machines concurrently. This has numerous advantages. The first one being that the time of test execution is greatly reduced. Given that a long running test or a set of test suites can be executed concurrently on different machines, it means that the time taken for all the test to complete will be significantly reduced. Nevertheless, the tests need to be carefully scripted to ensure that the tests suites are independent of each other. The next major advantage is that the automated test coverage levels with respect to the test basis (devices, browsers & versions, and the OS) can be easily increased. Given that Selenium WebDriver supports tests execution on a wide range of browsers, versions and operating systems, Selenium Grid exploits this capability to allow concurrency in test execution.

The reduced test execution time and the maximized test coverage levels gives an instant high ROI for an organization investing in Selenium Grid. One possible implementation of Selenium Grid is shown below. The Host Node will be the one instructing the execution nodes (node 1 to node 5) on which test to execute. Each node will have their own browser and OS. Even the host node can be configured to be an executing node.



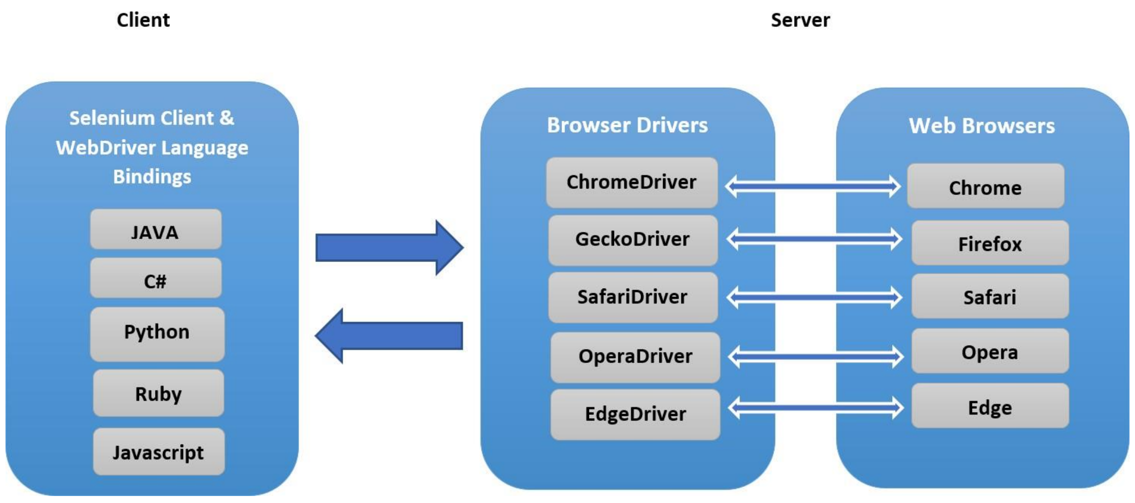
3.3 Selenium WebDriver ecosystem

Selenium 4 has brought quite some changes in the WebDriver ecosystem. These changes have brought more stability and optimization in the tool operations.

3.3.1 Architecture of the Selenium WebDriver

Cross-browser testing and concurrent testing are both supported by the Selenium WebDriver architecture. For code compilation, Selenium WebDriver offers integration with several frameworks, including Maven and ANT. To enhance test automation and reporting, it also allows interfacing with testing frameworks such as TestNG.

The architectural representation of Selenium 4 WebDriver is shown below:



The Selenium WebDriver architecture consists of the following three elements:

- Selenium Client Library / Language Bindings
- Browser Drivers
- Browsers

Selenium Client Library / Language Bindings

Multiple programming languages, including Java, C# and Python are supported by Selenium 4. Language bindings were created by Selenium developers to accommodate a variety of languages. Once the preferred language is chosen, the language binding library can be added it to the automation development project. If Python programming language is used, the language binding can be added using the pip command. If Java is used, the language binding can be imported as an external JAR (Java Archive) or, it can be added as dependencies if a Maven or a Gradle project is used for development.

Browser Drivers

Each of the Selenium-supported browsers has its own implementation of the W3C standard because Selenium supports a wide range of browsers. As a result, browser-specific binaries are readily available; these binaries are particular to the browser and shield the user from the implementation logic. The client libraries and the browser binaries are now directly interacting with each other as opposed to the previous versions of Selenium where the JSON Wire Protocol was in between to make the interface between the two.

Browsers

Only browsers that are locally installed, either on the local workstation or the server machines, will be allowed to conduct tests on Selenium. Installing a browser is therefore mandatory.

3.3.2 Browser Controllers

Browser controllers provides various methods for the Selenium client to interact with the browsers. Some of the most popular Browser controllers for Selenium WebDriver are listed below:

1) Get Command

Method: **`get(String arg0) : void`**

This method loads a fresh tab in the browser and load a new webpage. Takes a String as an input and gives no output.

The respective command to load a fresh web page is:

```
driver.get(URL);
```

2) Get Title Command

Method: **`getTitle(): String`**

The title of the current page is retrieved using this method. Returns a String value and takes nothing as a parameter.
The respective command to retrieve the title of the currently displayed page:

```
driver.getTitle();
```

3) Get Current URL Command

Method: **`getCurrentUrl(): String`**

This method retrieves the string that corresponds to the currently open URL in the browser. Returns a String value and takes nothing as a parameter.

The respective command to retrieve the string that represents the current URL is:

```
driver.getCurrentUrl();
```

4) Get Page Source Command

Method: **`getPageSource(): String`**

The page's source code is returned by this method. Returns a String value and takes nothing as a parameter.

The respective command to obtain the current web page's source code is:

```
driver.getPageSource();
```

5) Close Command

Method: `close(): void`

This method closes the window that WebDriver is currently in control of.
Returns nothing and accepts nothing as an argument.

The respective command to close the browser window is:

```
driver.close();
```

Quit Command

Method: `quit(): void`

All windows that the WebDriver has opened are closed by this method. Returns nothing and accepts nothing as an argument.

The respective command to shut down all windows is:

```
driver.quit();
```

3.3.3 Headless test automation

Selenium WebDriver's UI automation relies heavily on web browsers. An essential component of web test automation is opening a browser and running the test cases on it. However, we frequently come into problems when running Selenium tests on any of the browsers, such as poor browser rendering, conflicts with other software programs, etc. Additionally, many CI/CD systems are non-UI (such as Unix based systems). This may hinder running automation tests on UI browsers in the CI/CD pipeline.

As a result, to run the test cases on those systems, there needs to be a method to run the tests without the UI. This is where headless browser testing becomes essential. It facilitates the execution of Selenium Headless Browser tests without displaying any user interface. The headless browser can examine and comprehend webpages directly in the memory. As a result, a headless browser can give the real browser perspective without heavily consuming the system's memory. The time it takes a typical browser to load CSS, JavaScript, and render HTML because the browser GUI is not opened. Ten times quicker performance scaling can be seen with headless testing. The headless browser might be the best option if performance and system resource consumption are the crucial factors.

Selenium uses HtmlUnitDriver, a different WebDriver implementation that is comparable to FirefoxDriver, ChromeDriver, and other WebDriver implementations, to provide headless testing. The library must be manually added in order to use HTMLUnitDriver, which is available as an external dependency. HTML Unit is meant to be used within another testing framework, such as Junit or testing, and is not a general-purpose unit testing framework. It is a method for testing purposes to simulate a genuine browser like Chrome, Safari, and Firefox. Most headless testing is now being done with headless Chrome or Firefox.

The benefits of running a Selenium test case in headless mode include:

- Faster execution of automation tests
- Multi-Tasking
- Useful for web scraping
- Multiple browsers are supported.
- Tests can be executed on non-UI operating systems.
- Easier integration in the CI (continuous integration) workflow

Although headless browsers have many benefits, there are a few considerations to keep in mind when using them:

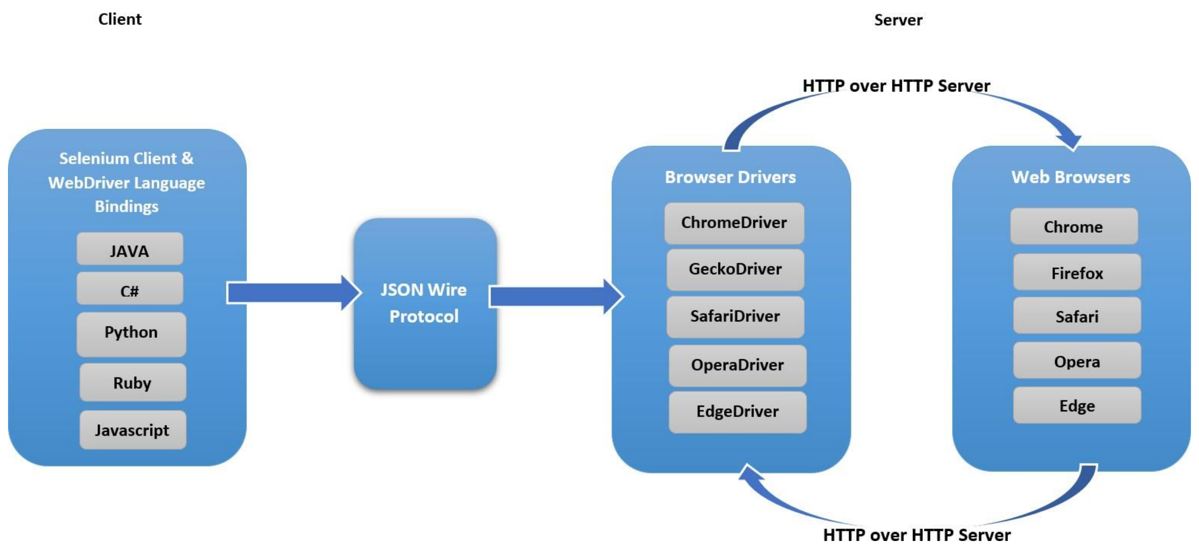
- Running tests in headless mode may make the debugging process more challenging as there is no GUI that can be used to identify failure points.
- Due to the pace at which tests are run, headless browsers don't accurately simulate user activity, and some tests may fail.
- Exploratory testing, visual regression testing, and other forms are testing are restricted due to the absence of UI.

3.4 Selenium 4

Selenium has undergone significant evolution over nearly two decades. Selenium 4 is the latest generation of the automation tool, and it has undergone a number of major architectural changes.

3.4.1 Architectural change in the Selenium WebDriver

The main Architectural difference is that previous generations of Selenium had JSON Wire Protocol. JSON Wire Protocol transferred the information from the client to the server over HTTP (Hypertext Transfer Protocol), in this, a selenium request is sent from a selenium client, the request is received by the JSON Wire Protocol over HTTP, and secured by the browser Driver, after that a response returned by the server and received by the client.



In Selenium 4 the communication between the client and server is direct. The client-server architecture is maintained. The client has 2 parts, the Selenium client & the WebDriver language bindings. The server is the browser drivers. Selenium client sends out a request to perform a command. The WebDriver language bindings is a code library that is mainly designed to drive actions. Browser drivers receive the request sent by the client and then return a response after an automation test step is executed on the web browser.

The Selenium client and WebDriver language bindings are an important part of the architecture where each language has its own unique bindings. Bindings mean that the same commands can be used by different languages. For example, a command written in java language has the same implementation in other languages like C#, Python, Ruby, etc.

WebDriver drives each browser using the browser's built-in automation support. A browser driver such as Chrome Driver controls the Chrome browser.

3.4.2 New Features

Although Selenium 4 comes with a lot of new features and changes, note that the change in protocol does not impact existing users as the drivers have completely adopted W3C protocols. In a nutshell, JSON wire protocol is deprecated in Selenium 4 and that does not impact users still using Selenium 3.

1. Enhanced Selenium Grid

Earlier Versions of Selenium Grid were complex to implement and rigid in terms of scaling. However, this new version comes with Docker support. This will allow TAEs (or developers) to spin up the containers rather than setting up heavy machines.
Managing Selenium Grid is now smoother and easier to use as there is no longer any need to set up and start hubs and nodes separately.

The Grid can now be deployed in 3 modes:

- Standalone mode
- Hub and Node
- Fully distributed

Unlike in earlier versions, the Grid will now support IPv6 addresses, and one can communicate with the Grid using the HTTPS protocol. In Grid 4, the configuration files used for spinning up the grid instances can be written in TOML (Tom's Obvious, Minimal Language) which will make it easier for humans to understand.
The Grid in Selenium 4 also comes with an enhanced user-friendly GUI. Overall, the revamped Selenium Grid will enhance the DevOps process as it provides compatibility with tools like Azure, AWS, and more.

2. Upgraded Selenium IDE

Most automation engineers are familiar with the record and playback tool, Selenium IDE. This IDE was once only available as a Firefox add-on which was then deprecated with the latest Firefox versions. This is because the add-ons in the latest Firefox (ver. 55) were standardized under the **web extension mechanism.**

With Selenium 4, the IDE is revived and now its add-on is available for major web-browsers like Firefox and Chrome.
The new Selenium 4 IDE provides some notable features including:

1. Improved GUI for intuitive user experience.
2. The new IDE also comes bundled with a SIDE tool (Selenium IDE) runner. It allows software testers to run '.side' projects on a node.js platform. This SIDE runner also enables individual QAs to run cross browser tests on local or Cloud Selenium Grid.
3. Improved control flow mechanism that enables testers to write better "while" and "if" conditions.
4. The new IDE comes with an enhanced element locator strategy (like a backup strategy) which helps locate an element in case the web element couldn't be located. It will result in the creation of stable test cases.
5. The code for test cases recorded using Selenium IDE can be exported in the desired language binding like Java, C#, Python, .NET, and JavaScript.

3. Relative or friendly locators in Selenium 4

Selenium 4 brings an easy way of locating elements with the inclusion of relative locators. This means testers can now locate specific web elements using intuitive terms that are often used by users like:

- **toLeftOf():** Find the element to the left of a specified element.
- **toRightOf():** Find the element to the right of the specified element.
- **above():** Find the element above with respect to the specified element.
- **below():** Find the element below with respect to the specified element.
- **near():** Find the element is at most 50 pixels far away from the specified element. The pixel value can be modified.

The introduction of this new method in Selenium 4 helps locate web elements based on the visual location relative to other DOM elements.

4. Better Window/Tab Management in Selenium 4

There are several instances in test automation where in one might need to open a particular link in a new tab or window to perform certain actions. To achieve this in previous generations of Selenium, TAEs had to create a new driver object and then perform the switch operation using the Window Handle method to perform subsequent steps.
This has changed in Selenium 4 as it comes with a new API interface: newWindow() that allows users to create and switch to a new window/tab without creating a new WebDriver object.

Sample code snippet to open a new window

```
driver.get("https://www.google.com/");

// Opens a new window and switches to new window

driver.switchTo().newWindow(WindowType.WINDOW);

// Opens BrowserStack homepage in the newly opened window

driver.navigate().to("https://allianceforqualification.com/");
```

Sample code snippet to open a new tab within the same window

```
driver.get("https://www.google.com/");

// Opens a new tab in existing window

driver.switchTo().newWindow(WindowType.TAB);

// Opens Browserstack homepage in the newly opened tab

driver.navigate().to("https://allianceforqualification.com/");
```

5. Deprecation of Desired Capabilities

Desired Capabilities were primarily used in the test scripts to define the test environment (browser name, version, operating system) for execution on the Selenium Grid.

In Selenium 4, capabilities objects are replaced with Options. This means testers now need to create an Options object, set test requirements, and pass the object to the Driver constructor.

Listed below are the Options objects to be used going forward for defining browser-specific capabilities:

- Firefox – FirefoxOptions
- Chrome – ChromeOptions
- Internet Explorer(IE) – InternetExplorerOptions
- Microsoft Edge – EdgeOptions
- Safari – SafariOptions

6. Capture screenshots of a specific web element

In previous Selenium versions, it was possible to capture screenshots of the entire page only but with the new methods on Selenium 4, on top of full page capture, it is now possible to capture screenshots of a specific web element or even section of the page.

The getScreenshotAs() method is used to get screenshot specified by the locator.

Sample code snippet for element level or section level capture in Java

```
public class SeleniumelementTakeScreenshot
{
  public static void main(String args[]) throws IOException
  {
    WebDriver driver = new ChromeDriver();
    driver.get("https://www.example.com");
    WebElement element = driver.findElement(By.cssSelector("h1"));
    File scrFile = element.getScreenshotAs(OutputType.FILE);
    FileUtils.copyFile(scrFile, new File("./image.png "));
    driver.quit();
  }
}
```

Sample code snippet for element level or section level capture in Python

```
from selenium import webdriver
from selenium.webdriver.common.by import By
driver = webdriver.Chrome()
driver.get("http://www.example.com")
ele = driver.find_element(By.CSS_SELECTOR, 'h1')
ele.screenshot('./image.png')
driver.quit()
```

4 Using Selenium WebDriver – 150 minutes

Keywords

Initialization, assertions, reporting, GUI interface, element states, exceptions, machine learning, self-healing tests, selectors, parallel test execution

Learning Objectives

STF4-1 (K2) Understand the different libraries available for Selenium in Python and / or Java

STF4-2 (K2) Know how Selenium WebDriver is initialized, executed, assertions made and terminated

STF4-3 (K1) Remember the important information needed on a test automation report

STF4-4 (K2) Understand the different common interactions possible with Selenium automation tool

STF4-5 (K2) Understand the concept of parallelism of tests and how it can be used for performance testing

STF4-6 (K2) Know how machine learning can help in reducing false positives and maintenance effort

STF4-7 (K3) Use the best practices in test automation given a scenario

STF4-8 (K3) Use appropriate strategy to handle different Selenium exceptions

STF4-9 (K3) Apply appropriate strategy to achieve test parallelism on Selenium tools

STF4-10 (K3) Apply appropriate test assertions given a test scenario

STF4-11 (K4) Analyze a test scenario to distinguish the best sequence of Selenium interactions

4.1 Managing Selenium libraries

Selenium supports multiple languages. In this section we will explore how Selenium libraries is handled in Python and Java.

Python provides an installation of library packages though the 'pip' command. To use Selenium 4, minimum version of Python 3.7 will be required.

For example, after Python is installed and the necessary environment variables are set, the command below will install Selenium 4 on the machine.

```
pip install selenium==4.5.0
```

The below command will install ChromeDriver on the machine:

```
pip install ChromeDriver
```

Alternatively, the binary of the library can be referred through its path. The variable 'binary_path' stores the path to the chrome driver executable.

```
from ChromeDriver_py import binary_path
```

In Java, there are dependency management and build solution tools. Maven and Gradle are the most common for Selenium.

The Maven solution introduces a project object model file referred to POM file which is an XML structure file. Every time a change is made to the project code, it updates the build status and continuously maintains and monitors the framework components and build. If there are no compilation problems with the framework, it provides a "build success" message; otherwise, it provides a "build failure" message.

A minimum version of Java 8 is recommended to install Maven dependency management. Environment variables need to be set. Once done, dependencies can be added to the ‘pom.xml’ file.

After adding the dependency for Selenium, the Pom.xml file will look like this:

```
<dependencies>

   <!-- more dependencies ... -->

   <dependency>

      <groupId>org.seleniumhq.selenium</groupId>

      <artifactId>selenium-java</artifactId>

      <version>4.5.0</version>

   </dependency>

   <!-- more dependencies ... -->

</dependencies>
```

Similar to how it is done with Maven, adding dependencies to a Gradle project is simple as shown below:

```
dependencies {

implementation group: 'org.seleniumhq.selenium', name: 'selenium-java', version: '4.5.0'}
```

Gradle automatically executes all tests that it finds, which it does by looking at the test classes that have been compiled. Gradle searches for and executes all methods marked with the @Test annotation when useTestNG() is specified.

TestNG is a Java testing framework that supports the use of assertions. Assertion is a way to confirm if the expected result and the actual result matches or not.

4.2 Selenium WebDriver in action

Once the libraries are all set, the WebDriver can be instantiated to run tests. The initialization is done based on the browser controller we want to invoke.

4.2.1 WebDriver initialization

To initialize the WebDriver in Python, after the necessary libraries are installed, the WebDriver class need to be imported from Selenium into the solution.

The below code will invoke a Google Chrome browser controller.

```
from selenium import WebDriver

driver = WebDriver.Chrome()
```

Similarly for Firefox browser, the below code can be used.

```
from selenium import WebDriver

driver = WebDriver.Firefox()
```

If specific browser configurations are needed, then object 'options' can be used as shown below:

```
from selenium.WebDriver.firefox.options import Options as FirefoxOptions

options = FirefoxOptions()

options.browser_version = '92'

options.add_argument("--headless")

options.platform_name = 'Windows 10'

driver = WebDriver.Firefox(options=options)
```

In the above Python code, we are declaring 'options' as configuration for a Firefox browser controller, setting the version of the browser to load as 92, in headless mode for Microsoft Windows 10.

For Java, the following libraries need to be invoked for the WebDriver initialization are:

1. **org.openqa.selenium.*–** contains the WebDriver class needed to instantiate a new browser loaded with a specific driver

2. **org.openqa.selenium.firefox.FirefoxDriver** – contains the FirefoxDriver class needed to instantiate a Firefox-specific driver onto the browser instantiated by the WebDriver class.

   OR

   org.openqa.selenium.chrome.ChromeDriver – contains the ChromeDriver class needed to instantiate a Chrome-specific driver onto the browser instantiated by the WebDriver class.

   OR

   org.openqa.selenium.edge.EdgeDriver – contains the EdgeDriver class needed to instantiate an Edge-specific driver onto the browser instantiated by the WebDriver class.

   OR

   org.openqa.selenium.safari.SafariDriver – contains the SafariDriver class needed to instantiate a Safari-specific driver onto the browser instantiated by the WebDriver class.

Note that org.openga.selenium.* will import all packages in the library which is not always the best way to move forward. Therefore, if more specific packages are needed, it may be better to just import the required ones.

For example:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
ChromeOptions options = new ChromeOptions();
options.addArguments("--headless");
options.setPlatformName("Windows 10");
options.setBrowserVersion("92");
driver = new ChromeDriver(options);
```

In the above Java code, we are declaring ‘options’ as configuration for a Chrome browser controller, setting the version of the browser to load as 92, in headless mode for Microsoft Windows 10.

4.2.2 Test Start

After the initialization of the WebDriver, the test scenario needs to be scripted. This step involves opening the browser and navigating to the web site under test or the SUT. We will initialize a variable to store the WebDriver object. The variable ‘driver’ will be the generic variable that will store our WebDriver object (in this case ChromeDriver) and will allow Selenium interactions (Navigate, click, sendkeys, etc.)

The Python code below opens a Firefox browser, navigates to the website, and maximizes the window.

```
driver = webdriver.Firefox()
driver.get(‘https://www.saucedemo.com’)
driver.maximize_window()
```

The Java code below opens a Chrome browser, navigates to the website and maximizes the browser windows.

```
driver = new ChromeDriver();
driver.navigate().to("https://www.saucedemo.com");
driver.manage().window.maximize();
```

4.2.3 Test sssertions

Given that the browser is now open and responding to the actions requested, outcomes of the carried actions need to be cross-checked. The automation browser is opened and the request to carry out the action to load the website was made. The next step is then to ensure that the website is truly loaded.

This is done through test assertions. For instance, to verify that the correct website was truly loaded, the website page title could be used. The more assertions inserted in the test, the better are the chances to spot deviations from the expected results. This is a real value addition when using automation technology for software testing.

There are two kinds of assertions:

1) Hard assertions: The test execution is aborted if the test does not mean the assertion condition. The test case is then marked as failed.
2) Soft assertions: The test execution continues till the end of the test case even if the assert condition is not met.

In Python, the assertion will be done as below on the page title:

```
assert " Page Title Example " in driver.title
```

In Java, an assertion on the page title can be done using TestNG library as below:

```
Assert.assertEquals(“Page Title Example”, driver.getTitle(););
```

The assert command throws an error if those two objects are not equal.

4.2.4 Test termination

At the end of any test script, the WebDriver is to be closed to conclude the test scenario.
The code is simple and short for both Python and Java as shown below:

```
For Python: driver.quit()
For Java:      driver.quit();
```

This procedure is also known as driver tear down in Selenium.

4.2.5 Test Reporting

Once the test is executed, the test results need to be reported. There are several test reporting tools that help outline the failure and passed steps.

Furthermore, a test automation report is expected to contain some important information such as:

- The test environment on which the test was execute.
- The date and time of the test start, test end and the duration of the test
- The device identifier / name on which the test was executed.
- Parameters of the test (browser, OS, etc.)
- The sequence of test steps executed.
- Status of the test (Pass or Fail)
- Screen capture for test step that failed.

The above information will allow TAEs or test analysts to further investigate the root cause of discrepancies between expected and actual results. Not all discrepancies are defects. There are also possibilities of false positive cases. The elements in the report should be complete enough to allow this investigation.

The reporting module in a TAS can be either an off-the-shelf reporting library or a custom-made reporting module. Most reporting modules will produce the results in an HTML format to ease common understanding of the results. Nevertheless, a detailed log for technical errors and exceptions is also expected. This will allow TAEs to investigate technical issues with the TAS itself.
Depending on the choice of programming language, there are popular off-the-shelf reporting library reporting libraries that just plug and play in the Selenium TAS.

Some of these reporting libraries are:

- TestNG Reporter
- JUnit Reporter
- Extent Report
- Allure Report

Each reporting library requires some configuration to import it in the working environment for test report generation.

4.3 Selenium GUI interface

Selenium interacts with browsers and GUI objects and as such it has a lot of interfaces to support those interactions.

The interfaces fall in 3 categories:

1) Browser control
   This allows Selenium to directly manipulate the browser.

2) Web elements interface
   This allows Selenium to read the DOM of the webpage, search for GUI objects and interact with them.

3) Debugging aid
   The debugging aid allows for investigation of failures on Selenium side. It is of great help to TAEs to adjust the test, the execution flow or update the TAS.

Debugging aid includes common exceptions as described below:

Exception	Description
NoSuchWindowException	Attempt to switch to a browser window when the window is no longer existing
NoSuchFrameException	Attempt to switch to a frame when it is no longer possible
NoSuchElementException	Attempt to find or access elements on the web page or application
NoAlertPresentException	Attempt to interact with a browser alert when the alert is not to be found
InvalidSelectorException	Attempt to interpret a selector having a syntax problem. For example, a malformed XPath
TimeoutException	Command did not execute or complete within the wait time
ElementNotVisibleException	Attempting an interaction with an element which is hidden or invisible
ElementNotSelectableException	Attempt to select an element which is not selectable
NoSuchSessionException	Attempt to interact with a browser session which closed or crashed
StaleElementReferenceException	Attempt to interact with an element which is no longer part of a webpage.

4.3.1 Interactions

Selenium Browser interactions includes the following common methods:

Method	Description
manage()	Gets the Option interface to manage the browser
navigate()	An abstraction allowing the WebDriver to access the browser's history and to navigate to a given URL
findElement(by selector)	Find the first element using the given selector
findElements(by selector)	Find all elements within the current page using the given locator
getTitle()	Get the title of the current page
getPageSource()	Get the source of the last loaded page
SwitchTo()	Move the focus to a frame / iframe element on page
close()	Close the current window and if last windows, closes the browser
quit()	Terminates the driver closing all associated window

The below methods are the basic interactions that can be done on a web element.
When a click() operation is executed, the automation solution needs one parameter which defines where to click.

When a sendKeys() operation is executed, the automation solution needs two parameters which define where to enter the text and what text to enter.
For web element defines the 'where' parameter and the parameter specified in the method defines the 'what' parameter.

The table below summarizes the most common methods available for a web element.

Python Method	Java Method	Description
click()	click()	Clicks on the element
send_keys(string)	sendKeys(string)	Enter the string or characters that is specified in the element
clear()	clear()	Clears the text present in the element
text	getText()	Retrieves the text present in the element
get_attribute(string)	getAttribute(string)	Retrieves the value contained in the attribute name specified from the DOM for the element

Consider the following DOM segment for an element:

```
<a href="https://www.alliance4qualification.info/" id="link" />
```

In Python, the getAttribute() method will be used as below:

```
element = driver.find_element(By.ID, "link")
element.get_attribute('href')
```

In Java, the getAttribute() method will be used as below:

```
element = driver.findElement(By.id("link"));
element.getAttribute ('href')
```

The output of the above instructions would then be:
https://www.alliance4qualification.info/

4.3.2 GUI elements states

The interaction with the web element depends on the state of the GUI element. An element can be visible, interactable, or even focused on the page.

These three states are described below:

Python Method	Java Method	Description
is_displayed()	isDisplayed()	Returns true / false value if the element is displayed
is_enabled()	isEnabled()	Returns true / false value if the element is enabled
is_selected()	isSelected()	Returns true / false value if the element is selected

Checking the states of the GUI element before attempting an interaction is recommended to avoid false positives in the results. For example, before attempting a click, it makes senses to ensure that the element on which the click is to be attempted is enabled and is displayed.

4.4 Parallelism of tests

The ability to run tests in parallel comes with a lot of advantages. The main one being that the test's execution duration is reduced significantly, and the test results are available even earlier. Given that tests are executed in parallel, it also implies that more tests can be conducted in the same time lapse.

There are two concepts when doing parallel tests. Separate test suites can be executed on different browsers concurrently on the same machines or separate tests suites can be executed on different machines concurrently.

4.4.1 Selenium automation and performance testing

It is advantageous to use Selenium automation tools for performance testing. The tests are indeed scalable, but care should be taken to limit the probe effect.

The probe effect is defined as the effect on the component or system by the measurement instrument when the component or system is being measured. Performance testing is about measurement of some metrics such as response time, usage of processing capacity and memory, etc. These measurements can be tampered with if ever the TAS is not properly designed and built.

The probe effect is most prevalent when a single machine is used to simulate multiple users. This will be implemented as multiple execution threads on a single machine and may entail that threads are blocked or enter a deadlock state. Each thread will be instantiating a browser controller and the browser to allow

execution. Therefore, some lags are expected in response time due to the overload in processing and memory usage.
For example, if the TAS is used to simulate 20 users on a single machine using Google Chrome browser, it will mean that 20 Chrome browsers will initialize and the actions will be done on each concurrently. The system will start lagging and the response time for the actions and tests will be falsified. As the device lag will introduce a probe effect in the results.

Therefore, this should be considered when using the TAS for performance testing. Ideally, the tests architecture should enable execution of tests on multiple devices concurrently and in headless mode to limit the probe effect to its minimum. Usage of Selenium Grid is very convenient then for these purposes.

4.5 Machine learning and test automation

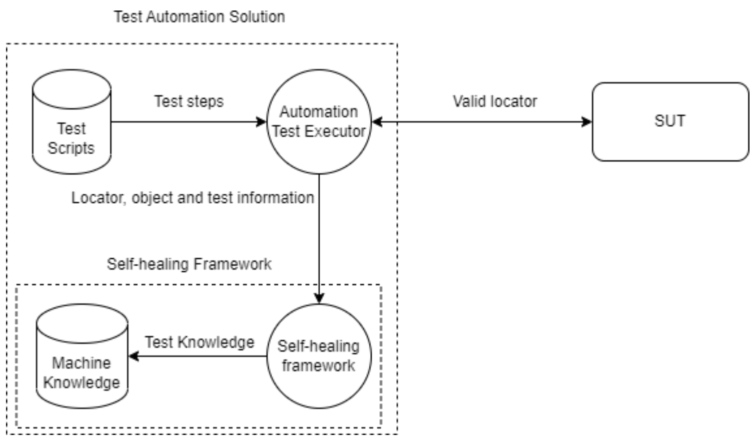
Using artificial intelligence in the field of automation is relatively new. The idea is to have the capability to rely on machine ability to learn on how to recover from failure or false positive to keep the test going.
Machine learning is a branch of artificial intelligence which focuses on the use of data and algorithms to imitate the way that humans learn, gradually improving its accuracy.

4.5.1 Self-Healing tests

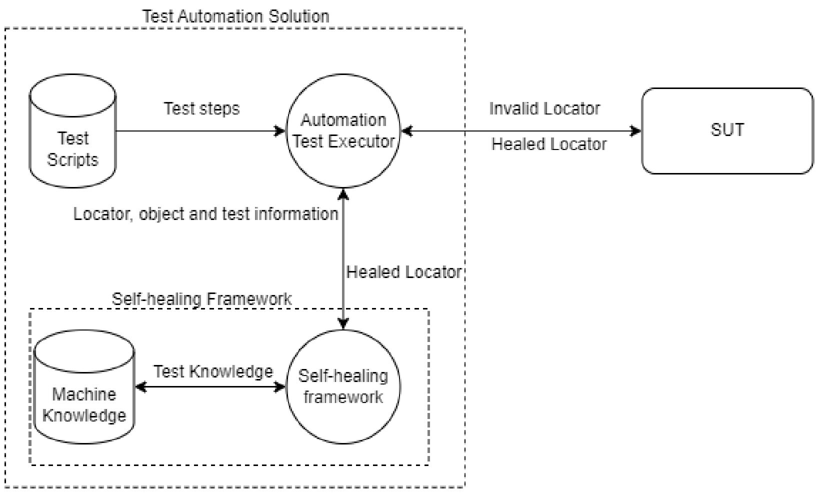
According to a study made by Telerik, 73.5% of tests are intermittently failing due to brittle locators. In other words, the locators used in the tests are not good or strong enough to sustain the tests over long period of test execution. This may happen due to changing the SUT, the technology used and the way the tests are written. In any case, this implies that tests are required to be continuously maintained and this does hinder the increase in test coverage and reduce the return of investment of test automation initiatives.

Self-healing testing is an approach of using machine learning to allow broken locators to be repaired in order to keep the tests operational and help in test automation maintenance. The way the mechanism works is that the self-healing framework plugs into the TAS and monitors on-going tests. The self-healing framework learns about the tests, locators and objects being interacted with when the tests are running.
In another test run, assuming that there was a slight change in the SUT, and some locators are no longer valid, when the invalid locators are attempted, the self-healing framework take over and suggest alternative locators (healed locators) that may work for the test. This way, the tests continue and still work without immediate test maintenance. Maintenance will still be necessary, but it can be done at a later stage. The operation of the self-healing framework is summarized in the diagram below.

Learning phase:



Healing phase:



There are both commercial and open-source self-healing frameworks. One self-healing framework that is made for Selenium based project is Healenium. Healenium provides the TAE with a detailed report at the end of the tests to describe which locator was healed. Healenium integrates well with java-based Selenium project and supports concurrent test execution.

4.6 Best practices

To maintain success of a test automation project, it is important to have some best practices on the underlying process of test scripting and TAS coding. The test automation project is a development project which implies that the best practices for coding are expected. It is expected that the TAS code is well annotated and indented, the source code is modularized to facilitate technical maintenance and the coding standard to denote the nomenclature for variable names, database field, etc. is maintained.

There are also other best practices to be considered for test automation projects.

4.6.1 Static and dynamic wait mechanism

Static wait, also referred to as implicit wait, is the concept of causing a break in execution for a specified time. It is often used in automation projects to allow page refresh or load before the next automated test step can be executed. It is not recommended to use static wait in test scripts for active test execution. This is because this can either create false positives or simply unnecessary waiting time during execution.

For example, consider an automated test executing on a browser. The TAE scripted 5s to allow the page to load before attempting to login. The test works well in the local environment without failure over multiple runs. The same test is deployed to be executed on the test environment but there it fails as the page takes 5.5s to load. The TAE then decides to increase the waiting time to 10s to be on the safe side that the test will run both on his local environment and the test environment.

This example shows how a static wait can cause false positives when the test environment is changed and also how around 4.5s is wasted waiting when executed on the test environment. 4.5s can be a short time for one test step but when added over long running tests, this can cumulate to hours.

It is therefore recommended not to use static wait for active test execution. It can surely be used when scripting or testing the scripts to investigate failure but when deployed for active test execution, it must be changed to a dynamic wait mechanism.

A dynamic wait mechanism in test automation, helps to negate the disadvantages of the static wait. This mechanism implements a condition to determine if wait should continue or not till a timeout limit is reached.

This is explained in the pseudo code below.

```
timeout = false
count = 1;
While (webelement not interactable and timeout = false)
{
        wait for 0.5s
        count = count +1
        If (count =60)
        {
            timeout = true
        }
}
```

The pseudo code above allows a flexible waiting time. It implicitly waits for half a second and checks if a certain web element is interactable or not. This web element can be used to determine if the next test step can be executed or not. If not, it will wait for another half second and at most, the loop will be executed 60 times before if exists. Using a dynamic waiting mechanism will therefore be beneficial as it pauses the test automation to a more reasonable time and is also more immune to false positives results.

4.6.2 Test Scripts Acceptance Criteria

When an automation test case is scripted, it is recommended that there is a certain quality standard to benchmark the test case against. This is also termed as a quality gate or acceptance criteria for the test case. This will allow test scripts to be checked for completeness and strength of locators used.
For example, for a login test case, it may be worthwhile to add as many as verification points as possible in the test case to detect possible defects. Some verification points may include the below:

1) Verify that the username is displayed on the page.
2) Verify that the password is input masked and is displayed as asterisk.
3) Verify that the login button is enabled when username and password are typed in.
4) Verify that the URL changes (or not) after login is attempted.
5) Verify that an entry / update is made in the database if the login is successful and if the TAS has interface with the database.

The above verification points will make the test more complete. These steps may be time consuming to do manually each time the login test is executed but for test automation it does take some time to script but is very quick to execute.
It is also very important to ensure that the locators used are strong as well. One way to ensure this is manual verification of the locator. For example, ensuring that the locator is not using any alterable attribute from the DOM or in case of an XPath selector, then relative XPaths are used.

Another way is to run the test multiple times over different test environments to ensure that the locators and the test case logic is not environment dependent.
For example, one test case can be executed 3 times on a test environment and 3 times on a pre-production environment. The acceptance criteria for the test case can then be that all 6 test runs are successfully passing. This verification is quite useful to detect brittle locators and therefore avoid costly false positives.

4.6.3 Choice of selectors and locators

The choice of the selector is important in the test automation.

The criteria for a good selector would be as below:

1) A selector and locator that stays constant.
   This will avoid a lot of test maintenance. There are some development technologies that changes the DOM elements at each compile or build time. It means that IDs of elements may not be constant as such using such selector and locator will be useless.

2) A selector and locator that can be used to uniquely identify web element.
   Given that most common actions on a Selenium automation tool requires a web element to be uniquely identified, it makes sense to use attributes that makes the element easy to uniquely locate. At times it may be required to use conjunction statements such as 'and' keyword to be able to uniquely identify a web element.

3) A selector and locator that has meaning.
   Having a meaningful value in the selector and locator is always helpful when following the test steps whether for review or maintenance. This will cut down the need to open the SUT and manually check which element is referred to in the test.

   For example, the below locator (.//*[contains(@type,"email")]) is more intuitive than (.//*[contains(@id,"text1091")]).

It is always a good idea to work closely with the development team when building the TAS and writing automated test scripts. It is at times possible for the development team of the SUT to inject a specific attribute in the DOM element that could be eventually used for test automation.

5 Implementation of Test Automation in an organization – 80 minutes

Keywords

Efficiency, effectiveness, phased implementation, ROI, CI/CD pipeline

Learning Objectives

STF5-1 (K1) Remember the factors to consider for implementing test automation in an organization

STF5-2 (K2) Understand how the evaluation of a TAS can be done

5.1 Factors to consider

With the growing dependency on technology, companies are required to deliver more innovative products and services at a fast pace to keep thriving in any market. Many businesses are turning to test automation technology to save money and improve software quality. Test automation undoubtedly can improve testing capabilities, replacing some of the resource-intensive manual efforts, and can be executed independently or in conjunction with manual testing. But there are several factors which need to be considered when adopting test automation in an organization.

The software application(s) should be analyzed to identify which parts can be automated. The aspects below are considered good candidates for test automation.

1. Features that require many manual interventions. For example, testers need to fill in multiple forms and perform several click actions to test a simple validation message.

2. Tests that are time-consuming and repetitive.

3. Tests that are complicated or challenging to carry out. For example, test cases that require processing of large amount of data within a pre-defined timeframe.

4. Test cases that are prone to human error when put through manual testing.

5. Regression testing to guarantee good functioning of existing software functionalities.

The above analysis helps to determine whether the software application(s) has enough scope for automation. If it is the case, the next steps would be to consider the below factors:

- **Business costs & benefits**

  Businesses should perform a cost analysis to determine the cost and the benefits of implementing test automation. The ROI profitability metric can be used to evaluate and compare alternative solutions before proceeding with the implementation. The time the test team is expected to save during test execution is a good measure of the ROI.

- **Leadership approval**

  Management approval and support is crucial for the successful implementation of the test automation. The TAM needs to build a strong business case outlining the cost, needs, benefits, timeline, and implementation plan.

- **TAS tools selection**

  Selecting the correct tool for test automation is important. The organization is recommended to do appropriate research to understand all available options and weigh the advantages and disadvantages of each before finalizing the tools to be used. Characteristics that can be important for comparison of tools would be technology used, skillset available internally or to be recruited, technical difficulty to implement the tool, pricing of the tool, organization / community supporting the tool, limitations of the tool, adaptation of the TAS tool towards the process. For example, Selenium WebDriver is a technical tool that supports major common programming languages and therefore the skillset can be easily trained / oriented towards the tool. The Selenium project is open sourced and supported by many large organizations and a wide community of passionate developers and software testers. Selenium automation tools are nevertheless limited to webpages and webapps.

- **Test environment**

  The organization should identify a team who will be responsible for setting up the test automation environment based on the hardware and software specifications of the test automation. It is recommended to have a dedicated environment for automation testing so as to avoid creating bottlenecks for the manual test teams.

- **Test automation team**

  Organizations should setup a team consisting of people with the appropriate skillset for test automation. For a Selenium automation team, the team can include developers and TAEs.

- **Prototype**
  The phased approached towards test automation allows time to adapt and fine tune the tool design and implementation. The test automation project normally starts as a pilot or proof of concept project that allows management and TAEs to judge reliability and realign expectations.

- **Documentation**
  Having the right processes in place contributes to the success of the project. Having these processes documented as a project management document is important to align the team and help in onboarding new members. Technical documentation is equally important. Technical documentation includes TAS designs, how to document and troubleshooting steps. These documents serve as a knowledge repository and give a certain degree of freedom to the team scripting and executing the tests. For example, steps describing how to integrate the latest browser controller on a Selenium based TAS to get the solution to work again after the executing browser has been updated, would be expected in the troubleshooting documents.

- **Other considerations**
  An automation project is subjected to changes in technology, browser, programming languages, innovations, etc. It is therefore very important for TAEs to stay connected to the technologies used through research and continuous learning. Regular workshops or training are required to share the gathered knowledge across the team.

A test automation project is a long-term project. It requires investment upfront, but it can provide a good ROI on the long term if it is implemented correctly. Test automation initiatives with short term goals and vision has greater chances of failing.

Evaluation of tools
The key to any test automation effort's success is choosing the appropriate automation tool. Prior to making a choice, each tool must be thoroughly analyzed. This calls for extensive planning and work. For any tool to be shortlisted to be used on the SUT, the tool must support both observability and controllability over the SUT (see section 1.1).
Organizations frequently purchase tool licenses because record and playback functionalities are satisfactory at the start. When the scalability and maintainability of tests then become important, then the record and playback tool does not appear to be the right choice. It is therefore important to maintain a long-term vision when evaluating tools.

There are three main stages in the tool evaluation process:

- Requirements gathering
- Tool selection
- POC (Proof of Concept) using the selected tool.

1. Requirements gathering

The requirements for the automation tool must be laid out during the requirements gathering stage of tool evaluation. Several crucial questions need to be answered, including the following:

- Which issues will the tool solve?
- What technological requirements must the tool meet to work in the environment?
- What are the expected gains and benefits from the TAS?

The following points can help to create a requirements list:

- **Management goals and objectives:** Usually, this will be determined by the product roadmap and the management's desired scope for test automation. The TAM might consider revising the timing based on the product roadmap which the utility is now available. Management will define a budget and coordinate actions based on it.

- **Tool's target audience:** Another crucial factor in the evaluation of test tools is the expertise of those who will be involved in test automation as well as that of those who will script and maintain the test automation scripts.

- **Compatibility issues:** The testing tool must be compatible with the operating systems that the product supports, the development environments that were used to construct the product, as well as the third-party software with which the product integrates.

- **Testing requirement:** This includes issues with manual testing, time restrictions for making minor system changes, shorter regression testing timeframes, setting up test data and tracking defects, higher test coverage and improved testing process efficiency.

- **Technology:** Technologies that the tool must support and features that the tool must have.

2. Tools selection

Selection is the second phase in the evaluation of test automation tools. It's crucial to keep in mind that not a single tool will meet all needs when choosing tools. After consultation with stakeholders, the tool that satisfies most of the evaluation criteria should be picked. The automation operations must be organized in accordance with the tool's limits in light of the requirements. Any tool that satisfies most of the assessment requirements may be evaluated. When many tools are shortlisted based on the evaluation requirements, further tool analysis should be conducted.

List each tool's characteristics in accordance with the following categories to do a feature categorization:

- **Mandatory features**: Features imperative to have in the TAS. Such as ability to schedule tests or integrate with the CI/CD pipelines.
- **Desirable features**: Features that are nice to have in the TAS. For example, ability to make API calls or queries to the database.
- **Irrelevant features**: Features that will add no value to the automation project in the organization. For example, the ability for the TAS to integrate with tools not used in an organization.

3. POC using selected tool

The proof of concept is the last step in the tool evaluation process. Although conceptually the tool may seem to meet the evaluation criteria, it must be tested in a few test scenarios or situations before being used in the final product. Most tool providers offer an evaluation period or trial period on their product for this use. For the POC, using the evaluation version is adequate. The scenarios selected for POC are crucial. They should be chosen so that the scenarios cover most of the controls and a few key features that are common to the entire product. It should be a sample that, when automated, should inspire confidence in the tool's ability to successfully automate the product.

Regardless of who performs the review, tool evaluation is in fact a process in and of itself that necessitates extensive research. The process outlined above enables one to choose the right tool to support a software test automation approach in an informed manner.

5.2 Evaluation of efficiency and effectiveness of test automation

Efficiency is about how well something is achieved with the least amount of wasted time, money, and effort while effectiveness is the measure of success of something. Efficiency of a test automation project is how well the project was completed with the least wastage of time, money, and effort.

Some metrics to measure efficiency of a test automation project are:

1) Estimated and actual cost of the test automation project.
2) Estimated and actual effort invested on the test automation project.
3) Estimated and actual test automation project timeline.

These estimated values of these metrics help in deciding whether the project should be done or not. The actual values of these metrics define how efficiently the project carried out.

The effectiveness of a test automation project is the measure of how successful the automation project is with respect to the set targets. The effectiveness is benchmarked against a set of targets that are defined at the start of the project.
Example of such metrics includes:

1) Percentage reduction in manual test effort such as measurement of EMTE (Equivalent Manual Test Effort)
2) Percentage reduction in test execution time
3) Ratio of false positives to defects
4) Rate of defect detection and removal
5) Automated test coverage level

A test automation project is considered successful when they are both efficient and effective. In other words, the investment made in the project is compensated for by the value it creates for the stakeholders. Effectiveness and efficiency metrics are not one-off measurements. Measurement and monitoring of these metrics is a continuous activity due to fact that the test automation project is an on-going project. These metrics will need to be recorded and analyzed by the Test Automation Manager (TAM) and be reported to stakeholders. Appropriate control over the test automation project can then to be applied by the TAM in case the metrics shows the project going off-track.

5.3 Success Factors

As a TAE in a test automation project based on Selenium, some factors need to be considered and focused on to make the project a success. Some of them are discussed below:

Organizational factors

1) **Always report the progress of the project**

   Implementing a Selenium automation project can be time consuming and getting a ROI from the project is a long-term journey as such visibility on the project status needs to be given to stakeholders and management to keep the momentum of the project.

2) **Phased approach**

   Implementation of an automation project needs to be done in phases. A big bang approach does impact the success of the project. If a company has multiple projects which are automatable with Selenium, then it makes sense to select a project which is not too critical and not too trivial to start automating the tests. This will ensure that in case of failure of the automation initiative, the impact is limited and in case of success, the ROI is encouraging. Once successful in one project the same can be extended to others in phases.

3) **Single TAS for multiple projects**

   Implementation of a Selenium project may be taking quite significant resources and as such, for a company with multiple projects that can be automated through Selenium tool, it is important to have a long-term plan for the TAS. The ROI of a TAS is amplified if there is a single generic TAS for multiple projects. Only the test scripts change and the TAS stays the same. This is also where keyword driven test automation framework design is very helpful.

4) **SDLC adaptation**

   To incorporate the test automation project in an organization in an efficient way, there may be some fine tuning of the processes required. This may be in terms of SDLC and the test process. Discussions about where the automated regression tests will fit in the process, what type of tests will be needed when and how often, etc. need to take place as early as possible.

People related factors

1) **Cooperation with the development team**

   Given that the development of an automation framework is a development project, having a good collaboration with the development team is beneficial. Software engineers can help technically in the TAS development and share important updates on the SUT for the test case scripting. It is recommended that software engineers also participate in reviews for the coding of the TAS and the test case scripting.

2) **Co-existence with the manual test team**

   There is always the fear that automation tests will replace manual tests. Managing this fear in the manual test team is very important because the two teams need to co-exist. Having regular meetings with the manual test team to explain the scope of the test automation project and how the two teams can support each other will help to alleviate concern.

3) **Upskilling the test team**

   The test automation project requires a specific skillset. The manual test team can greatly help in the success of the test automation project. Some upskilling is required but the team can assist in test scripting. There is indeed a higher reach for test automation coverage with the help of the manual test team.

Technical factors

1) **Use the latest stable Selenium language binding and version:**
   As a Selenium TAE, it is necessary to regularly check the updates on the Selenium project. This includes updates in Selenium WebDriver, Grid, IDE, and the associated language bindings. For proof-of-concept projects, the alpha or beta versions of the project can be used but for execution purposes, always ensure that the stable version is used.
   The official website for Selenium is https://www.selenium.dev/

2) **Ensure that the Selenium TAS is compatible with the latest browsers.**
   Unless there is a business need to run on outdated browsers, for example for legacy products, it is always recommended to maintain a Selenium TAS running on the latest browsers. Given that most browsers are usually updated automatically, it can be largely assumed that the end user of the product is also using the latest browsers and therefore it makes sense to have test automation running on the same parameters as the end users.

3) **Keep in touch with the Selenium community**
   The Selenium project is a community driven project and as such, it is expected that TAEs keep in touch with the community to stay up to date on the defects, resolution, and features. TAEs also have the possibility to report defects and discuss features. The Selenium community is very active on GitHub threads, Slack channels, IRC and even emails.

4) **Scheduling and the CI/CD pipeline**
   Scheduling of tests overnight or during the weekend is very helpful for regular quality checks. This helps to find defects early on. The inclusion of test automation in the CI/CD pipeline is also very valuable for implementing early testing. This way once the deployment is complete by the CI/CD tool, the automated test can be executed.

5) **Dedicated test environment for test automation**
   A dedicated test environment for test automation gives the TAE the flexibility to have specific versions of a SUT deployed on the environment to test automated scripts and do root cause analysis for failure. This helps a lot and if the environment is dedicated for test automation, it implies that the test environment is not impacted and therefore there is no impact for the manual test team.

6 Adapting a Selenium TAS – 45 minutes

Keywords

Database, API, variable, runtime

Topics & Learning Objectives for this Chapter:

Learning Objectives

STF6-1 (K2) Understand how dynamic variables can help on test automation

STF6-2 (K2) Understand why custom actions may be needed on test automation

STF6-3 (K2) Understand what additional verification checks can be undertaken by a Selenium based TAS

6.1 Dynamic Variables

Test automation frameworks do need to be fed with the appropriate test data to operate. At times, the data to be used needs to be unique. For example, if in a system we are trying to create a user and it is required to input the username and the email address. It is expected that the information is unique in the database. Therefore, when this test is automated, the values used need to be always unique also. It is not always possible to hard delete the records created by test automation or deleting the record in the database directly may compromise the integrity of the database itself. In such cases, there is a need to keep the test data coherent in the test itself rather than attempting to delete them after the tests.

Given that Selenium is supported by a lot of mature programming languages there is the possibility to create test data on demand. For example, concatenate timestamp numbers to test data to make it unique. This does give a certain flexibility in running the tests. The test data can be generated at runtime and test can be executed repeatedly without worrying about its uniqueness.

The quality of the test data is also very important. Using the same test data again and again will not help in finding data related issues. When testing software in other localization settings such as languages, the use of characters from that language is expected. For example, if an application is being tested in French, the use of characters such as 'ç', 'é' and 'è' are expected in the test data for names and addresses. The classic use of test data like 'John Doe' or 'Test Tester' will not be high quality test data as we may be missing our character encoding issues in the system. Given the versatility of Selenium supported programming languages, localized test data can be generated at run time as well using specific libraries and web services.

6.2 Custom Automation Actions

Some interactions in automation testing may be more complex than click or enter a specific text in a text box. There may be logic that requires complex interactions such as clicking on a web element if a certain event happens (conditional click action) or repeatedly clicking on elements until the element is no longer to be found on the page. Repeated clicks need to be implemented using a loop structure as Selenium can interact with one web element at a time. As such, these interactions require custom actions to be implemented.

For example, the below pseudo codes show implementations of the conditional click action and repeated click actions.

```
Function ConditionalClick(locator1, locator 2)
{
        if(web element described by locator1 is visible on page)
        {
            click on web element described by the locator2
        }
}
```

```
Function RepeatedClick(locator)
{
        while(count of web element described by the locator > 0)
        {
            click on web element described by the locator
        }
}
```

The automated tests are required to be as close as possible to the real-life use of the system. The end user will interact with the system based on the system's response. Having custom actions / interactions on a TAS makes the test more flexible and simulation of the end user actions even better.
For example, the conditional click action can be used to check if when leaving a page there is an error message. If there is, then the button to close the error message is clicked. If not, there is no click action performed. This also allows a certain recovery mechanism for the TAS not to carry failure from one test case to affect the next test cases.

6.3 Extended verifications

The more verification points we have in a test, the better it is to capture deviations from actual results. In this way, all deviations appear in the test results and are investigated to ensure that none go unattended.

6.3.1 Database checks

One of the checks that can be done is on the database level, if there is a possibility to connect the database through the TAS. When an action is done on the GUI, it is always good to confirm that the action was successfully carried out on the GUI level but also confirm the relevant changes in the database are carried out as well.

For example, consider the case where the TAS is creating a new user on a system and accessing the system as the newly created user. Once the user is created on the UI level, the next test can confirm that the user is truly created on the system by searching it on the UI level. Then, the same could be confirmed on the database level through a database query to verify that the record is in the correct format, correct fields, and correct table. Once the newly created user is logged in on the software, another query could be made by the TAS to the database to check if the corresponding record for the user accessing the software is present in the audit tables.

Connecting the TAS to the database can be done using different methods such as using dedicated integrations or connection strings such as JDBC which provides an API based interface to database systems. The versatility of the different programming languages supported by Selenium again helps in the integration of different database packages.

6.3.2 API calls

One of the most common interfaces that most systems implement is an API. Having the capability of making API calls and interpreting the results in a TAS is therefore very useful. This greatly helps to make test cases unattended.
For example, consider an e-commerce website which is integrated with an online wallet-based payment system. For testing purpose, the SUT of the e-commerce website is connected to the test system of the payment system. An API call can be made to clear a payment or decline a payment on the test system of the payment system. Automated system tests / automated end to end tests, requires the whole purchase process to be tested. In case, the TAS does not have the ability to make API calls, the test will have a manual intervention where the software tester will need to manually make the API call to allow the test to continue. In the other case, if the TAS can make the API call and interpret its response, then the test becomes completely unattended, and this gives a lot of advantages. As an example, the test can be scheduled to run automatically after work or right after deployment in the CI/CD pipeline.

7 Bibliography

ID	Author	Resource / Title	Edition / Version
01	ISTQB®	Test Automation Engineer Syllabus	2016
02	ISTQB®	Certified Tester Foundation Level Syllabus	2018
03	ISTQB®	Standard Glossary of Terms used in Software Testing Version 3.1	2018
04	Telerik Test Studio	10 tips on how to dramatically reduce test maintenance	2017
05	Selenium Project	https://www.selenium.dev/	2022
06	Larry Yang	Factors to Consider When Implementing Automated Software Testing	2016
07	Software Testing Magazine	Self-Healing Automated Tests	2022
08	Pratham Software (PSI)	Common Selenium Exceptions	2021
09	Aditya Garg and Pallavi Sharma	Selenium 4 - A quick and practical guide	2021
10	Baiju Muthukadan	Selenium Python Bindings	2022
11	T Tutorials Point	XPATH query language for XML	2018
12	Tutorials Point	Cascading Style Sheets	2017

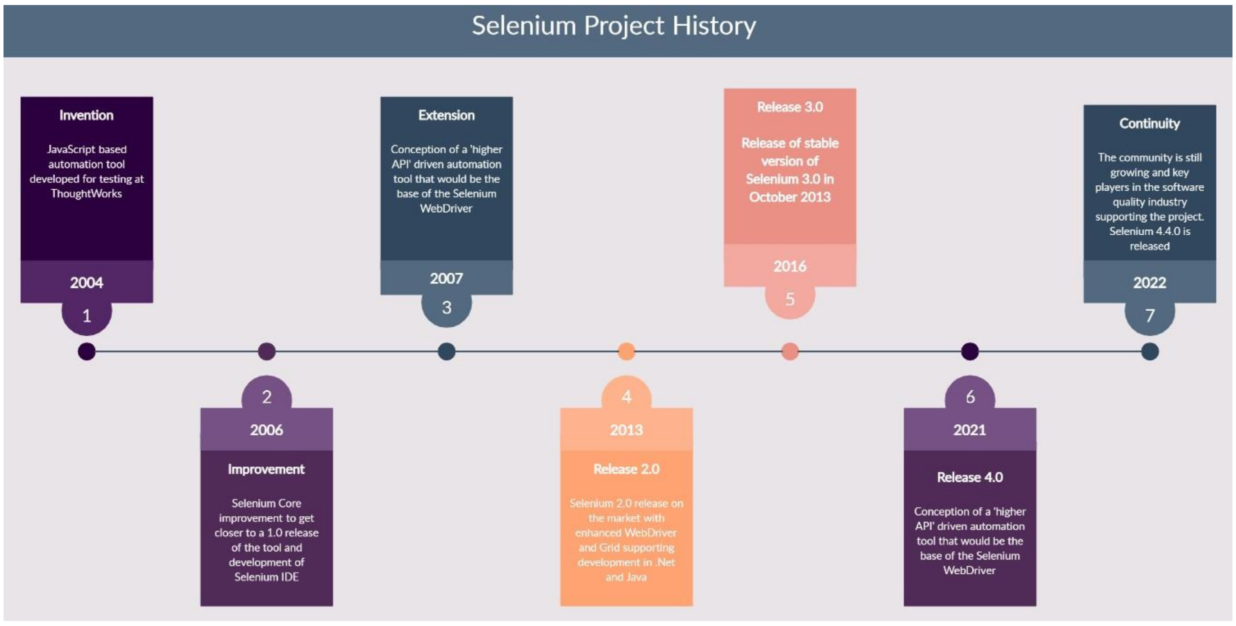
8 Appendix A

8.1 History and evolution of the Selenium automation tool suite

Selenium test automation tools have been around in the software quality area since 2004 and since then the tools have gained increasing traction and support from the community. The Selenium project started in Chicago at Thought Works (by Jason Huggins) with the intention to test an internal application in the company.

The project became popular over the years and even today the Selenium community and contributors continues to growing. The project is backed up by major organizations in the IT industry.

The diagram below summarizes key events in the Selenium project.


Selenium Project History
Invention
JavaScript based automation tool developed for testing at ThoughtWorks
2004
1
2
2006
Improvement
Selenium Core improvement to get closer to a 1.0 release of the tool and development of Selenium IDE
Extension
Conception of a 'higher API' driven automation tool that would be the base of the Selenium WebDriver
2007
3
4
2013
Release 2.0
Selenium 2.0 release on the market with enhanced WebDriver and Grid supporting development in .Net and Java
Release 3.0
Release of stable version of Selenium 3.0 in October 2013
2016
5
6
2021
Release 4.0
Conception of a 'higher API' driven automation tool that would be the base of the Selenium WebDriver
Continuity
The community is still growing and key players in the software quality industry supporting the project. Selenium 4.4.0 is released
2022
7

8.2 Learning objectives/cognitive level of knowledge

The following learning objectives are defined as applying to this syllabus. Each topic in the syllabus will be examined according to the learning objective for it. The level of Knowledge is the same as per the International Software Testing Qualifications Board (ISTQB®).

Level 1: Remember (K1)

The candidate will recognize, remember, and recall a term or concept.

Keywords: Identify, Remember, retrieve, recall, recognize, know

Level 2: Understand (K2)

The candidate can select the reasons or explanations for statements related to the topic, and can summarize, compare, classify, categorize, and give examples for the testing concept.

Keywords: Summarize, generalize, abstract, classify, compare, map, contrast, exemplify, interpret, translate, represent, infer, conclude, categorize, construct models

Level 3: Apply (K3)

The candidate can select the correct application of a concept or technique and apply it to a given context.

Keywords: Implement, execute, use, follow a procedure, apply a procedure

Level 4: Analyze (K4)

The candidate can separate information related to a procedure or technique into its constituent parts for better understanding and can distinguish between facts and inferences. Typical application is to analyze a document, software or project situation and propose appropriate actions to solve a problem or task.

Keywords: Analyze, organize, find coherence, integrate, outline, parse, structure, attribute, deconstruct, differentiate, discriminate, distinguish, focus, select