# Test Automation: Implementation Micro-Credential

Syllabus


# Table of Contents

## Test Automation - Implementation

# Overview

In this syllabus, we discuss the techniques and methods for implementing test automation.

# LEARNING OBJECTIVES FOR TEST AUTOMATION: TOOLS & SOLUTIONS

### Designing and obtaining test data
Explain the required contents of test data

### Environment requirements
Provide examples of key attributes of a good test automation environment

### Stubbing, Mocks, and Service Virtualization
Explain the purpose of using stubs, mocks, and service virtualization

### Using grids and agents
Explain the purpose of using grids and agents during test automation execution

### Selecting the right thing to automate - ROI, feasibility
Summarize the characteristics of items that should be automated first

### Selecting the Right Time to Automate Tests
Summarize the considerations for the timing of test automation implementation

### Understanding the Risk of Decisions
Summarize the significant risk items to be considered when making test automation decisions

### Configuration Management (CM)
Explain how configuration management of test automation assets can be implemented

### Designing for Maintainability
Summarize techniques that can be used to improve maintainability

### Documentation for Maintenance in the Future
Explain the importance of documentation in test automation

### Architecting Test Automation for Maintainability
Explain the concept of keyword-driven testing

### Good Automated Scripting Practices
Summarize the concepts of good scripting practices

# Designing & Obtaining Test Data

Repeatable and reliable test automation depends on having a source of test data that the test automation tool can read and process. This test data must contain the conditions needed to perform the automated tests and the expected results needed to verify the results. Ideally, this data is controlled in an automated manner, which allows it to be created, modified, and restored as needed.

These tasks can become complex when dealing with time-sensitive or interrelated data, as these data require frequent updates to stay current with processing dates.

Test data can be obtained by the following methods:

- Extraction from external sources, such as databases and existing files
- Created by the use of test data generation tools
- Manipulation of existing data, such as changing dates

Regardless of the method used, it is imperative that internal consistency be maintained.

# Environment Requirements

Before test automation tools can be implemented, a suitable test environment must first be designed and implemented. The test environment will depend on several key factors:

- Technology to be tested, such as mobile, web, or cloud
- Technology to support the planned tool(s)
- The size of the environment in terms of storage capacity to handle test data and test tools with their assets
- The processing speed needed for tests to run efficiently

The following are key attributes of a good test automation environment:

## Stability

The test automation environment must be stable enough to accomplish the execution of many tests in a repeatable and reliable way. The environment should not be subject to frequent crashes and unexpected restarts and error conditions.

An important task in test automation is test environment maintenance. Part of maintenance is periodically restarting the system(s) that host the test automation. This also includes periodic clean-up of unneeded files (such as old screenshots and logs) and restart of the tool(s) to clear memory. These are typically manual tasks.

However, it may be possible to automate restarts, initiate tests, and perform maintenance tasks using CRON jobs (Mac OS and Linux) . (Fig 1) To automate tasks in Windows, there is software available to do this, such as VisualCron.

```
15 17 * * * cd /usr/local/code/qa-lab-test-auto/ && ./cron_env_setup.sh ./regression.py android > /tmp/opentest_android.log
1 15 * * * cd /usr/local/code/qa-lab-test-auto/ && ./cron_env_setup.sh ./regression.py ios > /tmp/opentest_ios.log
2 15 * * * cd /usr/local/code/qa-lab-test-auto/ && ./cron_env_setup.sh ./regression.py weboa > /tmp/opentest_weboa.log
0 * * * * cd /usr/local/code/qa-lab-test-auto/ && /usr/local/bin/git pull && /usr/bin/crontab /usr/local/code/qa-lab-test-auto/crontab_server
0 * * * * cd /usr/local/code/qa-lab-test-auto/ && ./cron_env_setup.sh && ./remove_files.py
45 8,11,14 * * * cd /usr/local/code/qa-lab-test-auto/ && ./cron_env_setup.sh ./get_latest_rc.py
```

Figure 1 – Sample CRON Job in Mac OS

## Integrations

Test tools must be able to integrate with other tools and other components involved in testing. For example, a test automation tool may integrate with a test management tool for test execution and reporting. Another example is the integration needed between test data generation tools and test execution tools. (Fig 2)

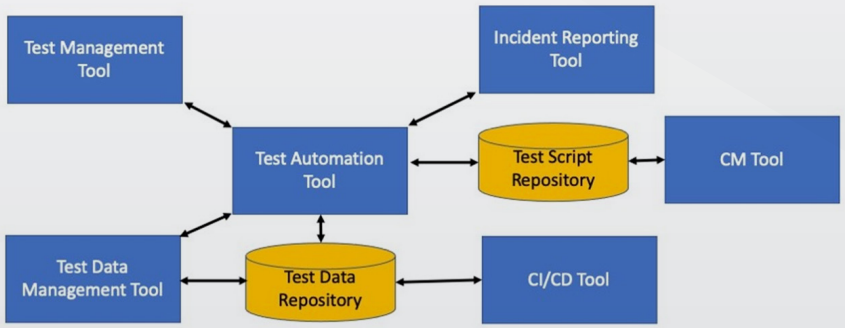


Figure 2 – Integration with Other Tools

## Suitable data available (anonymization)

A major concern in test data is that of maintaining privacy while still achieving realistic test data. There are two common strategies for this.

Create contrived test data that includes all the needed test conditions. The contrived nature of the data eliminates the need to make the data anonymous.

Create a set data derived from actual data supplemented with the needed test conditions. In this case, private data needs to obfuscated in some way. There are a few common methods for this, all facilitated by test data tools or by special scripting that can accomplish this task. These methods include:

- Masking – Simply replacing the actual data with "X" or similar
- Scrambling – Rearranging the values in a data field. For example, a phone number like 405-927-6677 becomes 405-967-2777.
- Replacement - Substituting the values in a data field. For example, a phone number like 405-927-6677 becomes 405-816-5566.

## Support for devices (cloud, simulators, emulators)

The test automation environment must be able to support all devices, including virtual devices such as simulators and emulators. If cloud-based platforms and devices are being tested, the need for support extends to those as well.

Maintaining device support and compatibility can be a significant challenge as the devices under test often require updates, some of which may require major updates to the test environment, such as operating system upgrades.

## Tool support and compatibility

As tools go through upgrades and changes, so must the test environment. This includes both commercial and open source tools. It should also be noted that tool upgrades can break existing test automation assets.

In some cases, depending on the extent of tool changes (such as major version releases), entire test automation libraries may be subject to changes in response to tool updates.

## Representative of production environment (scaled for data, horsepower)

Ideally, the test automation environment should have the same technology and profiles of the actual production environment. The test data may be different to protect privacy and also include needed test cases, but the closer the test environment is to the actual environment, the better.

If performance testing is part of the test automation scope, then careful consideration must be given to the components that impact performance, such as database size, network bandwidth, and hardware processor speed and memory.

## Controllable

The automated test environment must be controllable to ensure reliable and repeatable results and the correct association with releases or versions to be tested. For an environment to be controllable, you must be able to do the following:

- Create
- Configure
- Restore
- Teardown

To be sustainable, these capabilities should be automated.

# Stubbing, Mocks, and Service Virtualization

It is not uncommon for some items to be unavailable for testing. For example, a component might not yet be developed.

In these cases, the unavailable components may need to be in the form of a stub or mock object. These stubs and mocks can provide a placeholder for the future actual components to allow the larger test to be automated. The output from a stub or mock object is contrived so that the test automation run can be completed. At a future date, the actual components can be used (Fig 3).

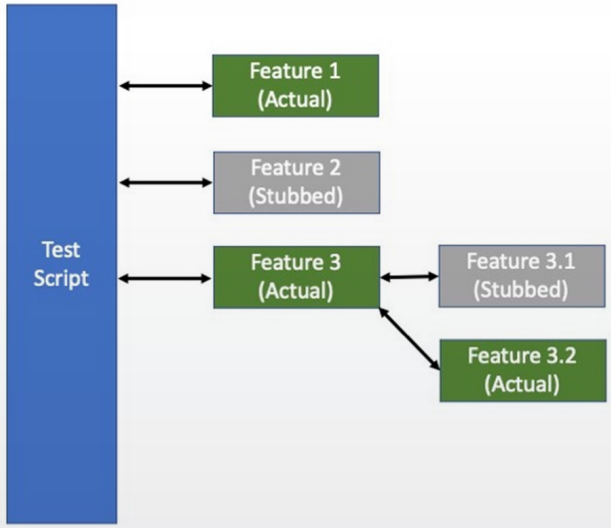


Figure 3 – Stubbed Components

Service virtualization is another way to test services. A service or microservice is simply small functionality that is accessed, typically through the cloud.

# Using grids and agents

Grids are used to distribute processing during test automation run across multiple computers. The processing is often performed by agents residing on each computer in the grid and controlled by a central controller or server. (Fig 4)

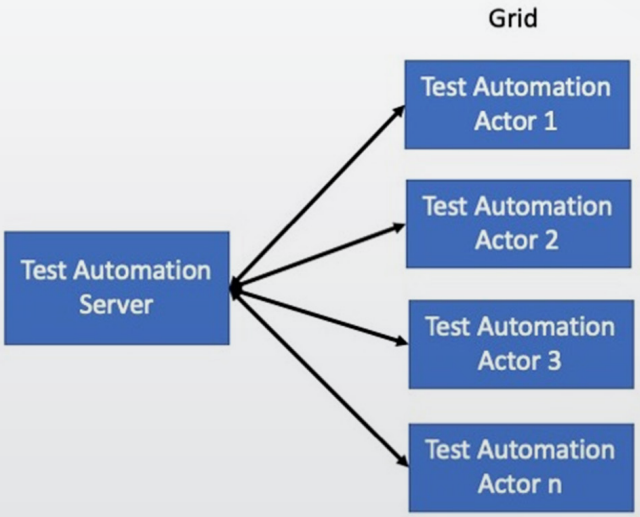


Figure 4 – Test Automation Grid

The benefits of using a grid are to:

- Reduce overall test automation run time by distributing the tests across multiple computers simultaneously
- Have a variety of test environment configurations

# Selecting the right thing to automate - ROI, feasibility

An important question to ask and answer is “What do we automate first?” A good place to start in test automation development is with those tests that are very repetitive and simple. Achieving early success in test automation can show quick return on investment (ROI) and

Starting test automation with large and complex tests can be frustrating and time-consuming. Such tests can give the impression that test automation is too difficult.

It is also important to understand that not every feature can or should be automated. There are some tests that require such a high degree of human interaction and evaluation that it is best to perform those tests manually. Automating simple and repetitive tests can provide the leverage to perform more time-consuming and creative manual testing.

# Selecting the Right Time to Automate Tests

Just as important as choosing the right tests to automate tests is choosing the right time to start automating tests. Much of this has to do with test automation readiness which includes people, processes and technology.

## Building into the CI/CD Pipeline

The entire concept of Continuous Integration (CI) and Continuous Delivery (CD) is based on the ability to automate tests during the build and delivery process. It is certainly possible to automate testing in the absence of CI and CD. However, if CI and CD are part of the development and delivery process, then test automation should support them.

CI and CD tests are not intended to be robust tests. That would be impractical as it takes more time to develop test automation scripts or other assets than is available in a typical development, test and release cycle. Instead, CI and CD tests are intended to test the stability of a build by testing basic functionality at a smoke test or sanity test level.

## Achieving Cost-effectiveness by Stubbing

To achieve reasonable cost effectiveness in test automation, some parts of the test may need to be simulated or “stubbed” to allow a larger feature or workflow to be automated more quickly as described in 2.3. (Fig 3)

## Readiness for End-to-end Test Automation

While test automation scripts can focus on small features, at some point the greater value is seen in automating tests of a full workflow process or task

from start to finish. (Fig. 5) This is known as “end-to-end” testing. End-to-end testing follows a workflow or task sequence and encompasses as many of the functions in the workflow as possible. Functions can be further decomposed into sub-functions (Fig 5) . Each block in the workflow represents a function that can be tested by automation. Some functions in Figure 5, such as “Fulfillment”, require some degree of human performance.

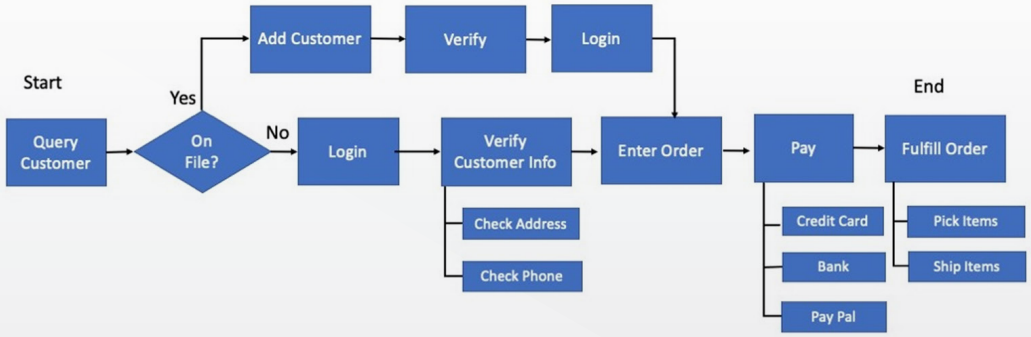


Figure 5 – Example End-to-end Test Workflow

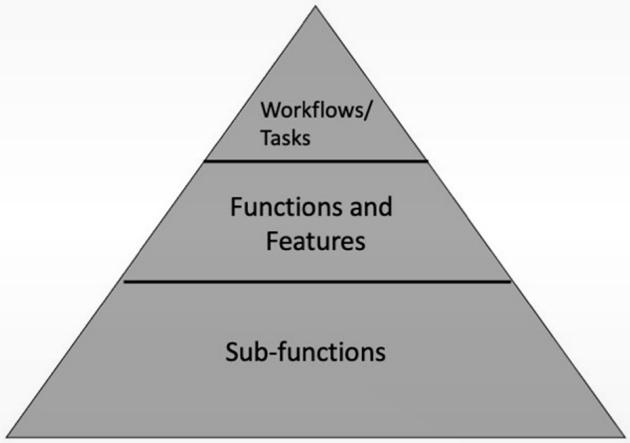


Figure 6 – Decomposition of Workflows

The ability to perform end-to-end test automation often depends on the ability to have test environments that span technologies. For example, a test might start with a new user setting up an account in a mobile app and placing an order. To achieve an end-to-end test, the order might need to be processed and fulfilled by another system in a totally different technological and operational environment.

Modular scripting as described in 2.9.1 can be very helpful in automating end-to-end workflows, as each modular script can be reused and combined in a variety of ways with the need to create new test scripts.

## Cost of Automating Too Early

It tempting to start creating test automation as early as possible. However, one must consider the degree of change likely to occur early in feature development.

Test automation maintenance can be very time-consuming and labor-intensive, and therefore, costly. While maintenance can't be avoided entirely, it does help to wait until the features have stabilized before starting to automate them.

# Understanding the Risk of Decisions

Decisions in test automation carry risk. The key is to get the best information possible and validate feasibility by performing proof-of-concepts (POCs) and pilot projects before committing too much time and money in a particular too or approach.

Common decisions that carry significant risk are:

- Which tool to acquire
- When to start automation efforts
- Where to start automation efforts
- How to deploy test automation to the larger organization

# Configuration Management (CM)

It is extremely important to verify that the correct versions of applications, data, test scripts, and any other version-based test environment items (databases, operating systems, simulators and so forth) are in place to ensure valid results are seen in test execution.

CM can be maintained manually, however, the most practical and reliable means of maintenance is through the use of tools. This is especially true when maintaining test scripts and other assets that change on a frequent basis. These assets should be treated like code and should reside in a central repository where changes can be committed. Branches can be created for making changes and testing them before merging into the main trunk ( Fig 7).


Figure 7 – Code Branch for Test Automation

Tools such as Bitbucket and Git can be very helpful in this process. Backups should be taken on a frequent basis in case the repository gets corrupted in some way.

# Designing for Maintainability

Maintenance of test automation assets can be reduced significantly by how test automation is designed. Approaches such as modular scripting, data-driven testing and keyword-driven testing are effective and proven ways to reduce maintenance.

## Modular Scripting

The problem with linear test scripts that are just a line by line set of execution instructions, is that they tend to become long and redundant. Consider the situation where you may be testing the same function multiple times with varying order of execution and test data.

One example might be in automating a workflow where a customer is added to the database, their information is changed in some way, then eventually the customer is deleted or deactivated. These functions could be expressed in one script (Fig 8).

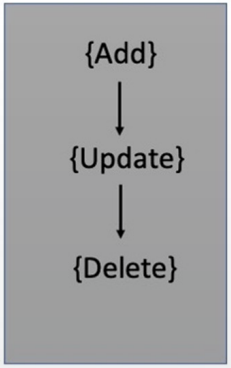


Figure 8 – Single Script with Multiple Functions

However, if we break the long linear script into three smaller scripts (add, change, and delete), we can treat them as building blocks to achieve many tests (Fig 9). The greatest advantage is if we need to change the “Add” script or others (“Change”, “Delete”, and similar.), we only need to change one script as opposed to many scripts which each perform the “Add” function.

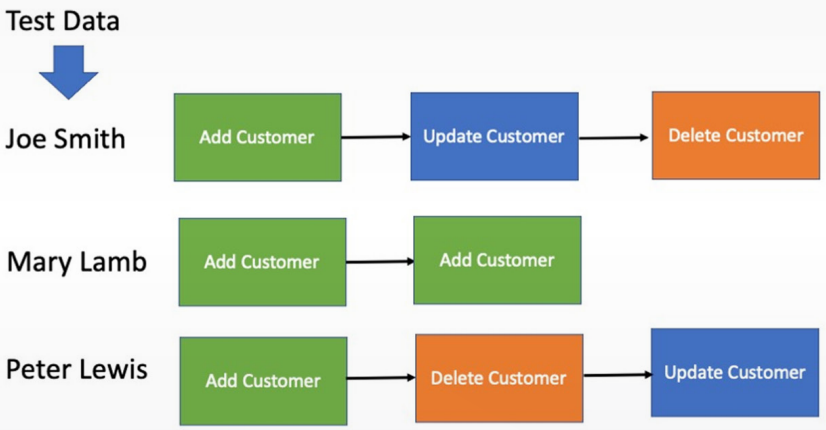


Fig 9 – Modular Scripts Combined in Different Ways

Sometimes, modularity is achieved by using macros called from a script (Fig 10). It is not uncommon for macros (or modules) to call other macros or modules (Fig 11). These examples are in pseudocode.

```
DESCRIPTION: This script reads customer data from a test data file and places an order as an existing customer
START;

LAUNCH APPIUM;
CALL MACRO LOGIN;
# We are reading ten customers because that is how many customer and product variations are possible.
FOR I,1,10
CALL MACRO READ_TEST_DATA_CUST;
CALL MACRO VERIFY_CUSTOMER;
CALL MACRO SELECT_PRODUCT;
CALL MACRO PLACE_ORDER;
CALL MACRO CONFIRM_ORDER;
CALL MACRO LOGOUT;
NEXT;

FINISH;
```

Fig 10 – Sample Script Using Macros

```
DESCRIPTION: This macro performs login and handles exceptions
START;

# We should be at the home page of the app
TAP ON ACCOUNT PROFILE AVATAR;
TAP SIGN-IN { id: .com.sign-in-link}
# We should be at the login entry page
TAP USERNAME { id: .com.username}
SENDKEYS {id: .com.username, keys: cust_name}
TAP PASSWORD { id: .com.password}
SENDKEYS {id: .com.password, keys: cust_pass}
TAP SIGN-IN-BUTTON { id: .com.sign-in-link}
# If there is a login failure, it is logged and we move to the next
test customer
IF ERROR
   CALL LOGIN_ERR
ELSE;

FINISH;
```

```
DESCRIPTION: This macro is called if
there is a login failure

LOG "SIGN IN FAILED"
TAKE SCREENSHOT
```

Fig 11 – A Macro Calling Another Macro

One important thing to remember is that when modular scripting is used, there must be a uniform start and stop point for each modular script. That way, when each modular script is invoked, the correct starting and ending points are consistent. For example, a login script could always end on the first page seen after login.

## Data-Driven Testing

A good practice in both manual and automated testing is to separate data from actions. If data remains in the scripts, then you need a different script for each instance of test data. In many cases, this would lead to hundreds or thousands of scripts, which is infeasible to execute and maintain.

When data is stored outside of the test script, then only one script is needed per function. Each time a test is completed for one instance of test data, the next instance is input to the script and the cycle repeats until all the test data has been processed (Fig 12). This is the essence of data-driven testing.


Fig 12 – Data-Driven Testing

# Documentation for Maintenance in the Future

Documentation is often skipped or neglected because it takes additional time and effort. People sometimes think that documentation is a non-productive activity, but just the opposite is true – it increases productivity by saving time that would be spent in trying to re-create test automation assets and environments.

Documentation is not just for the people who initially create test automation, but for those people in the future who will be maintaining and extending it.

Items to document in test automation include:

- Test environment setup and maintenance
- How to troubleshoot issues
- Test script creation and maintenance

Documentation should not only reside in a central location such as a wiki or tool, but also in the automation itself, such as comments in a test script.

# Architecting Test Automation for Maintainability

Test automation architecture has to do with large concepts which allow test automation to incorporate frameworks such as keyword-driven testing. The right test architecture allows people to more easily interact with the test tools and also facilitates maintenance.

## Keyword-Driven Testing

One of the most popular test automation architectures is that of keyword-driven testing. Keyword-driven testing is based on the concept of building a test framework that allows tests to be invoked when a particular keyword or "action word" (such as login, search, and so forth) are encountered in a main script, sometime known as a "driver script." Keyword-driven testing often uses data-driven testing to feed test data to the driver script. (Fig 13)

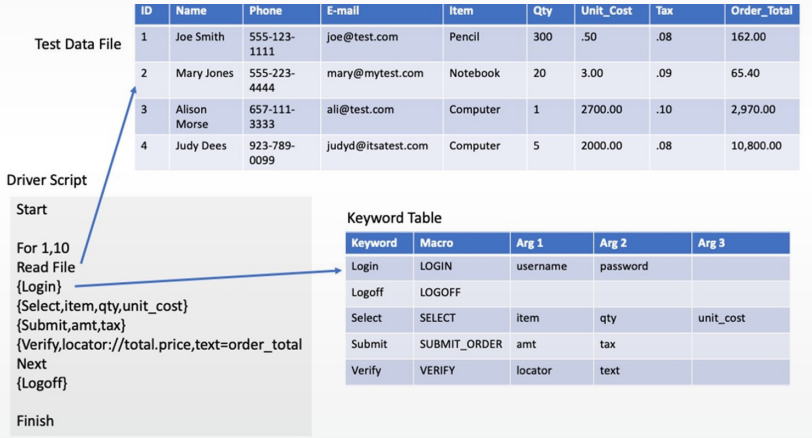


Figure 13 – Keyword-Driven Test Architecture

The benefits of the keyword-driven approach are:

- Easier maintenance because tests are modular and reused
- The ability to scale tests because the modular scripts are data-driven
- Higher engagement and adoption by non-technical testers since a test engineer is typically responsible for the creation and maintenance of the framework. Testers can contribute test conditions and data into the framework through a user interface or other means, such as a spreadsheet file, csv file and so forth.

# Good Automated Scripting Practices

When creating automated test scripts, it is good to follow practices that enhance maintainability and readability.

## Clear Comments

Comments are most helpful when they explain the things that are not obvious. For example, why was a particular approach taken. Another example is to provide guidance:

“If the following command fails, check the location settings on the device. They should be set to ….”

## Simplicity

Simple and shorter scripts are best because they are easier to understand and maintain. In addition, simple scripts often run faster than longer and more complex scripts.

## Modularity

Modularity helps achieve other things such as data-driven testing and keyword-driven testing. It also helps to isolate functionality which reduces maintenance.

## Debugging Features

It is helpful to include screenshots, log comments and similar things to help find problems when a script fails.

## Reliable Locators

A pre-requisite for test automation is to have a way to locate and interact with objects. There are good options, such as object ID, name, and so forth. The problem is that software developers don’t define these attributes in meaningful and stable ways.

Use the most reliable object locators possible. Try to find a locator that is less likely to change, such as name, id, class, and so forth.

XPath and x/y coordinates will often work for a short time can be fragile and lead to more maintenance. Only resort to these kinds of locators when none others are available.

It is good to communicate the need to reliable locators to the developers. They may actually have the same need.

## GOOD PRACTICE IN DEFINING RELIABLE XPATH

With all due warning about the inconsistency of XPath, sometimes it may be the only option. At that point, how the XPath is coded in the test script has a lot to do with the stability of the script.

Things to do if Xpath is used as a locator:

Use Relative XPath: Start from a known, stable ancestor element rather than the absolute root of the document. This makes your tests far less brittle.

- Example: //div[@id='my-section']//button[@type='submit'] (finds a submit button within a div with id "my-section")

Unique Attributes: Leverage attributes like id, name, class (if unique), type, or custom data attributes to pinpoint elements.

- Example: //input[@type='email' and @name='emailAddress']

Text-Based Locators (Use with Caution): If an element's text is reliably static, you can use contains() or text(). However, these can be fragile if the text changes.

- Example: //a[contains(text(),'Product Details')]

Things to Avoid in XPath:

Absolute XPath: These start from the root (/html/...) and are extremely brittle. Any change in the page structure breaks them. Unfortunately, copying and pasting the absolute XPath is easy, which is why so much automation breaks.

Index-Based XPath: Relying on element position (e.g., //div[3]/p[2]) is risky, as the order of elements can easily change.

Overly Complex XPath: Keep your XPath expressions as simple and readable as possible. Complex ones are hard to debug and maintain.

Some other helpful XPath tips:

Test Your XPath: Use your browser's developer tools (usually opened with F12) to test your XPath expressions in the console. This helps ensure they're targeting the correct element.

Add Comments: For complex XPath locators, add comments explaining their logic. This greatly aids maintainability.

Have a Locator Strategy: Have a clear strategy for prioritizing locators such as (ID > Name > CSS > Relative XPath > other).

## PAGE OBJECT MODELS

The Page Object Model (POM) is a design pattern widely used in test automation to enhance test maintainability, readability, and reusability. It promotes the separation of test logic from page-specific elements and interactions. By adhering to the Page Object Model, you create a more robust, maintainable, and scalable test automation framework.

For each web page in your application under test, you create a corresponding "Page Object" class. This class encapsulates:

- Elements (Locators): The locators (e.g., XPath, CSS selectors, IDs) are used to identify web elements on the page (buttons, text fields, links, etc.).
- Actions/Methods: Methods that represent the actions a user can perform on the page (e.g., clicking a button, entering text, selecting an option from a dropdown).

Benefits of POM:

- Improved Maintainability: If the UI changes (e.g., an element's ID is modified), you only need to update the locator in the Page Object class, not in every test case that uses it.
- Increased Reusability: Page objects can be reused across multiple test cases, reducing code duplication.
- Enhanced Readability: Test cases become more concise and easier to understand because they interact with page objects through descriptive methods (e.g., login_page.enter_username()) rather than directly with element locators.
- Better Code Organization: Separating page logic from test logic improves the overall structure of your test framework.
- Abstraction: Hides the underlying implementation details of interacting with web elements from the test cases.

Key Principles:

- One Class per Page: Each web page should have its own dedicated Page Object class.
- Public Methods for Page Actions: The Page Object class should provide public methods that represent the actions a user can perform on the page.
- Private Locators: Element locators should generally be private within the Page Object class to encapsulate the page's structure.
- Return Page Objects (For Navigation): When a page action results in navigation to another page, the method should return an instance of the corresponding Page Object. This enables method chaining. Method chaining, in this context, is a technique that makes your test code more concise and readable by allowing you to call multiple methods on an object in a single line of code. It's also sometimes referred to as a "fluent interface."

## USING CSS

CSS stands for Cascading Style Sheets. It's a stylesheet language used to describe the presentation of a document written in HTML or XML (including various XML dialects like SVG or XHTML). In simpler terms, CSS controls how web pages look.

In addition to formatting, CSS can also describe structure. When other means of locators don't work, CSS can be an option to try. However, CSS may be fragile at times.

Consider this example:

```
<div id="product-list">

  <div class="product">

    <h2 class="product-title">Awesome Widget</h2>

    <button class="add-to-cart" data-product-id="123">Add
to Cart</button>

  </div>

</div>
```

Here are some CSS selector examples:

- #product-list .product .product-title (this selects the product title)
- button.add-to-cart (this selects the "Add to Cart" button)
- [data-product-id="123"] (this selects the element with the specified data attribute)

## OBJECT IDS

Object IDs are the most reliable way to specify a locator IF they are stable (not changed for the convenience of the developer(s). This also means that an object ID can't be dynamic to be used as a locator. The object ID must be unique and consistent, and hopefully, meaningful. For example, a user name field name might be "user_name".

Consider the following example:

```
<form id="loginForm">

  <label for="usernameInput">Username:</label>

  <input type="text" id="usernameInput"
name="username">

  <label for="passwordInput">Password:</label>

  <input type="password" id="passwordInput"
name="password">

  <button type="submit"
id="submitButton">Login</button>

  <div id="errorMessage"></div>

</form>
```

In this example, the IDs are well-chosen due to the following:

- loginForm: Identifies the form itself.
- usernameInput and passwordInput: Clearly indicate the input fields.
- submitButton: Identifies the submit button.
- errorMessage: A container for error messages.


www.atsqa.org