Test automation with QWords

Tips & Best Practices

Table of Contents

This document is meant for test automation engineers just getting started with **Copado Robotic Testing** and especially **QWords** style of test automation.

Copado Robotic Testing - QWord Libraries

QWords are simple, maintainable cross-platform keywords which aim to make test automation easy and maintainable. Compared to for example **selenium**, QWords make creating test automation scripts faster and maintenance a lot easier.

Here are different QWord libraries which are included in Copado Robotic Testing:

QWord libraries

- QWeb - Generic Web automation library
  - QWeb provides ways of automating any web application.
- QMobile
  - Automating mobile applications
- QVision
  - Computer Vision library. Works with any UI application.
- QForce (Salesforce)
  - QWeb "extension" that provides salesforce specific funtionality
- QS4Hana (SAP)
  - QWeb "extension" that provides (web) SAP specific functionality

Importing libraries

```
Library   QWeb

# or

Library   QMobile

# or

Library   QVision

# or

Library   QForce

# or

Library   QS4Hana
```

You can also import more than one of our libraries to same test suite / test case. For example:

```
Library   QMobile
Library   QVision
```

When importing multiple libraries implementing keywords with exact same name are used, you need to set the search order:

```
Set Library Search Order     QMobile    QWeb
```

...or call QWords with fully qualified name:

```
QMobile.ClickText     Home
QVision.ClickText     File
```

QMobile, QForce and QS4Hana include QVision features. You can use computer vision to find / click texts that can't be found with normal means (for example shadow DOM elements)

Please note that QWords are continuously developed and maintained and hence refer to the latest version of QWords documentation.

Structure of a test project

The recommended directory structure to any Copado Robotic Testing test automation project:

```
project_name (root folder)
 ├──── tests ①
     ├──── my_suite.robot
 ├──── resources ②
     ├──── locators.robot ③
     └──── keywords.robot ④
     ├──── app ⑧
         ├──── android_app.apk
         ├──── iosd_app.ipa
     ├──── config ⑨
         ├──── mobile_config.yml
 ├──── libraries ⑤
     ├──── custom_library.py
 ├──── images ⑥
     ├──── web_logo.png
 ├──── files ⑦
     ├──── test_data.csv
```

① Test scripts/suites should be stored under **tests** folder

② Additional resources should be stored under **resources** folder

③ locators.robot file should store variables for element locators (i.e xpaths etc.)

④ keywords.robot should store any additional robot FW keywords

⑤ Any custom Python robot libraries should be stored under **libraries** folder

⑥ Reference images used in image comparison should be stored in **images** folder

⑦ Any additional files, for example csv files that contain test data, should be stored in **files** folder

⑧ QMobile specific: test applications which should be installed should be stored in **resources/app** folder

⑨ QMobile specific: .yaml files for mobile device configuration should be stored in **resources/config** folder

Locator strategies

One of the greatest conceptual difference of **QWords** compared to other test automation tools is the use of UI texts as a locator when ever it's possible. This is usually a bit hard to gasp for people with selenium background; they have learnt to use xpaths or css selectors everywhere. If you only take one advice from this guide, this would be it: Use textual locators **when ever possible**. Textual locators are easier for non-technical people to understand. There is no need to inspect html source, you can just use what you see on the screen. On the other hand, textual locators are usually easier to maintain.

There are situations where more complex locators (xpaths etc.) might be needed, but it's usually a small portion of test cases / steps. If you notice yourself using xpaths in most steps, please consider re-factoring your test script implementation!

Here are four ways of finding elements. These are presented in order of preference of using them:

Text

The most preferred way is using UI texts as locators. Examples:

```
*** Settings ***
Library         QWeb

*** Test Cases ***
Example test case using textual locators
    OpenBrowser         https://qentinelqi.github.io/shop       chrome
    VerifyText          The animal friendly clothing company
    ClickText           Scar the Lion
    ...
    TypeText            Username        myusername@test.com     # finds input field
based on label
```

Keywords expecting UI text as a locator usually end with "Text" (ClickText, VerifyText, IsText etc.)

Text keywords

Attribute values

If there is no UI text available, it's possible to find the element using any attribute value the element has.

```
*** Test Cases ***
Example test case using attribute values as locators
    OpenBrowser         https://www.google.com      chrome
    # finds input field based on attribute title value
    TypeText            Search                      Copado Robotic Testing
    ClickItem           Clear
```

Keywords expecting attribute value as a locator usually end with "Item" (ClickItem, VerifyItem, IsItemt etc.)

Item keywords

Xpaths

Xpaths are also supported as locators. These are mostly used by QWords ending with ***Element**.

```
*** Test Cases ***
Example test case using xpaths as locators
    OpenBrowser         https://qentinelqi.github.io/shop    chrome
    VerifyElement       //ul[@class\="product-list"]
    ClickText           Our Story
    VerifyNoElement     xpath\=//ul[@class\="product-list"]
```

Icons / images

Image comparison can also be used. In that case, you need to have a reference image and it will be found (or clicked) on a screen. Typically image comparison tends to be a bit harder to maintain.

Keywords for using icons/images have a naming convention ***Icon**. (ClickIcon, VerifyIcon etc.)

Icon keywords

Appstates

Appstate is usually the first keyword in every test case. It should handle all preconditions for a test case, i.e. make sure that when the actual testing starts, we know exactly what is the state of the application.

As an example, if you are testing *shopping cart* functionality in a web application and need to validate that the amount of items in a cart match to expected value, then you should make sure in your appstate that shopping cart is in know state (usually no items in shopping cart) prior to executing rest of the test case.

Due to nature of differences in applications Appstates are usually keywords that test automator has to define by himself/herself. Custom keywords are defined in **resource** file. Here is one example of the Appstate structure:

In resource file (for example *./resources/keywords.robot*) create a custom keyword for Appstate implementation:

```
Qentinel_FI
    [Documentation]     Navigates to Qentinel Finnish homepage
    GoTo                https://qentinel.com/fi/
    VerifyText          Finland. Change location

Qentinel_DE
    [Documentation]     Navigates to Qentinel German homepage
    GoTo                https://qentinel.com/de/
    VerifyText          Germany. Change location
```

Then, in your test cases you could make sure correct language specific pages are open by calling the Appstate:

```
Test the global home page
    [Documentation]     Accepts the cookie policy, changes the
    ...                 location to 'Global' and verifies home page text.
    [Tags]              Home
    Appstate            Qentinel_FI
    ClickText           Accept
    ClickText           Change location
    ClickText           Global
    VerifyText          Robotic software testing. Qentinel Pace
```

```
Test the global home page
    [Documentation]     Accepts the cookie policy, changes the
    ...                 location to 'Global' and verifies home page text.
    [Tags]              Home
    Appstate            Qentinel_DE
    ClickText           Accept
    ClickText           Change location
    ClickText           Global
    VerifyText          Robotic software testing. Qentinel Pace
```

Usually Appstates are used for example to login to application on web or making sure certain dialog or view is open on native/mobile applications.

Assertions

You can assert that an element exists/is visible using keywords starting with **Verify**. In addition, you can just check if element exists and return the result as boolean using keywords starting with **Is**.

```
Assertions example
    [Documentation]     Example of different assertions
    [Tags]              Assert
    VerifyText          Robotic software testing
    VerifyNoText        This should not exist

    VerifyItem          Remove item
    VerifyNoItem        emove item      30

    VerifyElement       //button[@title="Send"]
    VerifyNoElement     //button[@title="Copy"]
```

Assertion keywords

Anchors

Intro to Anchors

Sometimes webpage contains multiple, duplicated UI texts. While QWord libraries try their best to click only elements that are actually clickable, you may face situations where there are duplicated elements and clicks go to the "wrong" one. This is where concept of **anchors** steps in.

In a nutshell, anchors are little hints to underlying test execution framework on which of the duplicate elements to select. They work with basically all interaction keywords which uses UI text as a locator (for example ClickText, TypeText etc.)

Anchors can be either text or numbers.

Textual anchors bind text to be found to another text close by. So using for example the following script:

```
ClickText    Login      anchor=Cancel
```

...QWeb tries to find and click text **Login** which is closest to text **Cancel**.

Numeric anchors are positional; QWeb finds nth instance of the element on screen. For example, using the following script:

```
ClickText    Login      anchor=3
```

...QWeb would click third instance of element containing text **Login**.

SearchDirection

You can also specify the relative direction **from** anchor, which contains the correct element. This can be done by using keyword **SetConfig SearchDirection**:

```
# finds input "My Locator" on the right of text "Robot"
SetConfig    SearchDirection       right
TypeText     MyLocator   Typing this    Robot

# finds input "My Locator" above of text "Robot"
SetConfig    SearchDirection       up
TypeText     MyLocator             Typing this    Robot

# setting back the default value
SetConfig    SearchDirection       closest
```

Sleeps / waits / timeouts

There should not be reason to use **Sleep** keywords or such except in very exceptional cases. Almost all QWords have an inbuilt timeout, i.e they keep re-trying to find elements and do their action until action succeeds OR until timeout limit is reached. By default this timeout limit is 10 seconds.

You can change this timeout temporarily by giving an argument timeout:

```
VerifyText      Username    timeout=3       # only wait 3 seconds for element to
appear
ClickText       Login       timeout=30      # wait max 30 seconds for element to
appear
```

You can also change timeouts for every keyword (globally) with SetConfig:

```
SetConfig     DefaultTimeout    60   # sets default timeout to 60 seconds, affecting
all keywords
```

After setting defaultTimeout with Setconfig, you can use keywords without any arguments:

```
SetConfig       DefaultTimeout    60
VerifyText      Username    # would wait 60 seconds for text to appear before failing
ClickText       Login       # would wait max 60 seconds for element to appear
```

This is especially useful in case you run your tests against different environments that may have different resources; you can dynamically set the timeout based on environment under test.

If you must use explicit delay in certain step, you can give argument delay to a :keywords:

```
VerifyText      Username    delay=3       # always wait 3 seconds before checking if
element exist
```

It's better to give timeout argument than to give delay argument. Timeout will ensure that test execution continues as soon as element appears, but with delay argument you would always wait for the given time, regardless if text appears or not.

QVision "backup"

QMobile, QForce and QS4Hana include some QVision features. You can use computer vision to find / click texts that can't be found with normal means (for example shadow DOM elements). This can be done by using additional argument `recognition_mode=vision` to **ClickText** and **VerifyText** QWords. There is no need to specifically import QVision library.

```
*** Settings ***
Library         QForce

*** Test Cases ***
Example test case using QVision recognition
    OpenBrowser         https://qentinelqi.github.io/shop       chrome
    VerifyText          Products                recognition_mode=vision
    ClickText           Scar the Lion           recognition_mode=vision
```

Tags and test case documentation

Get familiar with `[Documentation]` and `[Tags]` notation for test cases. Every test case should be documented, so that other persons understand why they were made and/or what they aim to test.

Tags are useful in categorizing and filtering test cases. You can run just the test cases containing certain tag(s) from a larger test suite.

```
*** Settings ***
Library         QWeb

*** Test Cases ***
Example test case with documentation and tag
    [Documentation]     Your test case should be documented HERE
    [Tags]              smoke   example
    OpenBrowser         https://qentinelqi.github.io/shop       chrome
    VerifyText          The animal friendly clothing company
    ClickText           Scar the Lion
```

Settings / Configuration

QWord libraries have a lot of settings that can be changed to modify the default behaviour of our libraries. These settings can be changed using **SetConfig** keyword. Changes done with SetConfig will apply globally, meaning once setting is changed it's in use for all test cases / QWords until changed again.

Here are few most common settings that are often changed based on application behaviour:

```
*** Settings ***
Library         QWeb

*** Test Cases ***
Examples of configs
    # Do nothing after typing to a field (default: use tab to go to next field)
    SetConfig   LineBreak    ${EMPTY}

    # do not clear text field prior to writing
    SetConfig    ClearKey    {NULL}

    # set default timeout for all QWords 60 seconds (default: 10 seconds)
    SetConfig   DefaultTimeout    60
```

SetConfig documentation

QEditor

If you use Visual Studio Code as an editor, then QEditor extension extremely useful.

3rd party libraries in Copado Robotic Testing

You can use any 3rd party library in Copado Robotic Testing. Some libraries may need to be installed before they can be used.

Here are instructions how to install libraries / dependencies.

These libraries should be installed by default:

```
robotframework                          4.1.1
robotframework-appiumlibrary            1.6.3
robotframework-databaselibrary          1.2.4
robotframework-datadriver               1.5.0
robotframework-debuglibrary             2.2.1
robotframework-extendedrequestslibrary 0.5.5
robotframework-faker                    5.0.0
robotframework-oxygen                   0.2
robotframework-pabot                    2.1.0
robotframework-requests                 0.6.4
robotframework-stacktrace               0.4.1
```

Robot Framework specific

The tips in this chapter are not restricted to Copado Robotic Testing or QWord libraries, but more about Robot Framework's syntax.

IF's

Robot Framework supports branching using IF/ELSE clauses. The basic syntax is given in the example below.

```
*** Settings ***
Library         QWeb

*** Test Cases ***
If/Else example
    OpenBrowser             https://www.google.com              chrome
    ${count}=       GetElementCount     //a
    IF    ${count} == 0
        Log    There were no links on the page
    ELSE IF    ${count} < 5
        Log    There were less than five links on the page
    ELSE
        Log   There were 5+ links on the page
    END
```

Loops

Robot Framework supports FOR loops. The basic syntax is given in the example below.

```
*** Settings ***
Library         QWeb

*** Test Cases ***
Looping throught url's
    OpenBrowser             about:blank             chrome
    @{urls}=                Create List             https://www.google.com
https://www.copado.com
    FOR                     ${url}                  IN                  @{urls}
        GoTo                ${url}
        ${title}=           GetTitle
        Log                 ${title}
    END
```

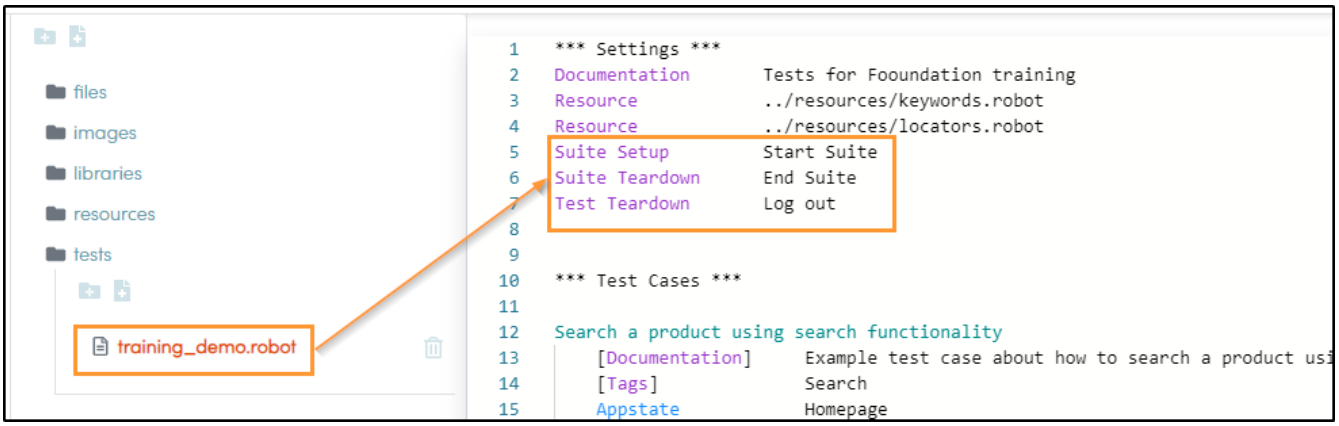
Setups and Teardowns

Robot Framework has support for two types of setups and teardowns: Suite level and Test case level setups/teardowns.

Use **Suite Setup/Teardown** to handle conditions that will be done once per test suite. For example:

- Opening/closing browser
- Loading test data etc.
- Specific configurations used in suite

Use **Test Setup/Teardown** to handle conditions that will be done once per test case, for example logout or loading test data specifically for one test case.



Test Setup is a bit similar conceptually than Appstate, but Appstate should be more about navigating to current view etc. Test Setup is optional, but every test case should have an Appstate.

More about setups/teardowns

Appendix

More information

There is free QWeb workshop material available at GitHub. This is using only QWeb library, but it will introduce these and even more complex concepts quite well.