Okapi Search by Anchors HowTo

1. Introduction

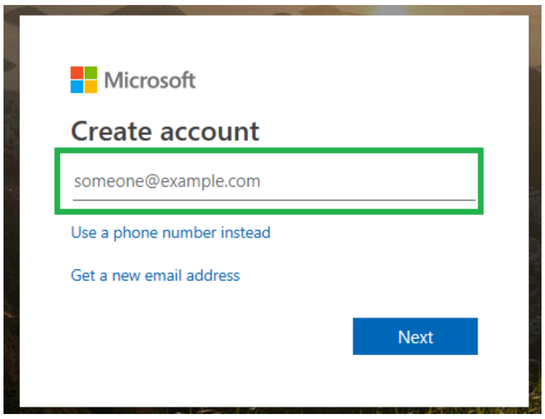
Besides supporting the traditional UI Web element search methods such as by XPath, by CSS Selector, by id, by class name, etc., Okapi introduces two unique and very powerful search methods – by anchor; by two anchors.

These powerful search methods allow automation test script developers to identify UI Web elements in a quick and intuitive way without the need to have deep knowledge on XPath, CSS, DOM (document object model) and Html.

2. How Search By Anchor Works

Search by anchor algorithm finds an UI Web element based on the shortest DOM distance from a known UI Web element, called anchor.

Let’s look into the simple example below.
To automate the tests for Microsoft registration Web page, we need to find locating information for the email text box in the picture below.



The easiest method to do that is to inspect the page’s html with Chrome browser and use copy XPath function of Chrome. It returns this XPath - `//input[@id="MemberName"]`

With Okapi, we write some simple lines of code:

```
public static ITestObject EmailTextBox => TestObject.New(“//input[@id=’MemberName’]”);
EmailTestBox.SendKeys(“john.doe@microsoft.com”);
```

What about in a week from now, a front-end developer at Microsoft changes the id of this Web element from ‘MemberName’ to ‘MemberEmail’? Our test scripts based on this XPath will fail and we have to visit them and change the XPath accordingly.

Okapi search by anchor for this Web element can be written as below:

```
public static ITestObject EmailTextBox => TestObject.New(SearchInfo.OwnText(“Create account”),
SearchInfo.TagName(“input”));
EmailTestBox.SendKeys(“john.doe@microsoft.com”);
```

Why are we able to do that? We know that the email text box on this Web page starts with html tag ‘input’ (We can double check with Chrome’s inspect) and our intuition lets us know that this text box is close to the text 'Create account' (picture above) and this text is quite unique on the Web page.

`SearchInfo.OwnText("Create account")` is the ANCHOR and `SearchInfo.TagName("input")` is the information of the Web element we want to search.

When we apply search by anchor, firstly, we use human intuition more than IT domain knowledge, which makes it easier for us and saves us time and effort as well. Secondly, developers at Microsoft can change the Web page html source a lot later on and it is unlikely that we need to refactor the test scripts to catch up with the changes.

If the developers change the displaying text from 'Create account' to something else then the test scripts will fail. However, these fails are good fails because they offers an opportunity for testers to check if the UI changes are valid or developers accidentally change the UI texts.

Less technical and more intuition, less test script refactoring.

3. **Search By Anchor May Fail?**
While search by anchor works nicely and beautifully for most of the cases, there are cases where the method cannot find the expected Web element or it finds more Web elements than expected.

Unable to find the expected element:
We expect it returns Web element A but it actually returns Web element B.
Why? The algorithm is based on SHORTEST DOM DISTANCE and the usage is based on INTUITION. No intuition-based things give us 100% correction. If we want to move toward intuition-based usage we have to accept this fact. Otherwise, just go with traditional search methods.

Most of the time, shortest DOM distance is also the shortest distance on the Web page that our eyes can see but not necessary 100% of the cases. Where possible, I think developers should make sure that shortest DOM distance is also the shortest distance on the Web pages. Reality is not ideal.

To be able to improve this situation, why not include second, third, etc. shortest distances in the search result outcome? Okapi ITestObject has `SetShortestDomDistanceDepth()` method for users to do just that. However, more than one Web elements may be found.

More than one Web elements found:
There are two cases where more than one Web elements are found.
The first case is there are more than one Web elements having the same DOM distances to the anchor. The second case is where users use `SetShortestDomDistanceDepth()` to tell the algorithm to be less strict on the search conditions.

If you try out the previous mentioned example, it will fail because the Next button also starts with an html 'input' tag, which has the same DOM distance as the email input box. Two Web elements are returned.

What to do when more than one Web elements found:
When more than one Web elements are found, Okapi returns them in a list.
When there are more than one Web elements found, by default Okapi picks the first one (index as 0). Okapi ITestObject has `SetElementIndex()` method for users to set element index. For the example in previous section, the email text box is the second item in the list and is indexed as 1.

```
EmailTestBox.SetElementIndex(1).SendKeys(“john.doe@microsoft.com”);
```

But wait a minute, how do we know which index to use? Trials and errors. But trials and errors takes more time and effort than constructing an XPath ourselves?

Okapi has some facilities to help users to make decisions.

ITestObject has the method `TryGetElementCount()` so we can use it to determine how many Web elements are found. We can loop through the found elements and use `TryHighlight()` to highlight them on the Web page (some Web elements are hidden, not visible so `TryHighlight()` does not highlight them though).

ITestObject also has property `AllLocators` which returns XPATHs of all the found Web elements. You can try these XPATH locators with inspect functionality of Chrome browser to find out which is the expected one.

And finally, Okapi ITestObject has `FilterByDisplayed()`, `FilterByEnabled()` and `FilterByClickable()` which can be applied to narrow down the search outcomes. We can chain them and AND logic is applied.

Example:
```
TestObject
        .New(SearchInfo.OwnText("create account"), SearchInfo.TagName("input"))
        .SetShortestDomDistanceDepth(2)
        .FilterByDisplayed()
        .FilterByClickable()
        .SetElementIndex(1)
        .Highlight()
        .SendKeys(“test”);
```

4. Mini Development

When we encounter such above cases, does it still worth it to stick with search by anchor for those cases? Depends. If we want to take the hard work now and fewer refactors in the future then continue search by anchor otherwise we construct XPATH locators for these cases.

Okapi has a facility to help us to do it with less effort, reducing the hard-works. It is called mini development.

For normal Web UI test automation script development, we have to capture Web element identifiers/locators, write a script, and execute/debug the script to double check whether the locators are correct and the script is good enough. Then we may modify the locators and the script and run it again. We may need to repeat these steps multiple times to finish one test script with quality. This is a long and time wasting process.

Imaging a script has to log in to a Web site, do something on page one, then go to page two to do something else and so on so forth up to page ten then log out. So far we have automated up to page nine and in the process of automating page ten. With the script development cycle as mentioned above, we have to run the test script again and again, stepping through page one to page nine multiple times while we know that the script is working well with those pages and what we want to focus on development now is page ten.

Mini development helps to reduce this effort, focusing on page ten only.

In mini development, we can run or debug a test script and stop running where we want but we don't close the browser. Then we can do something manually on the Web page on the browser and then run another test script to continue doing something else on the previous browser without the need to launch a new browser instance and start all over again.

Mini development can be achieved using Okapi
`DriverPool.Instance.CreateReusableDriverFromLastRun();`

Example:
Step 1: run test script 1 to login to a Web site and do something
Step 2: stop/end test script 1 but not closing the browser
Step 3: (optional) do something manually on the open page on the browser
Step 4: capture Web elements on the browser, automating the page with script 2
Step 5: run newly developed script 2 to confirm if it works correctly on the previous opened browser and on the Web page left opened from step 3

```
[Test]
public void Script1()
{
    DriverPool.Instance.ActiveDriver.LaunchPage("https://signup.live.com");
    //code to do other things here
}

[Test]
public void Script2()
{
    DriverPool.Instance.CreateReusableDriverFromLastRun().SetTimeoutInSeconds(10);

    TestObject
        .New(SearchInfo.OwnText("create account"), "input")
        .SetShortestDomDistanceDepth(2)
        .FilterByDisplayed()
        .FilterByClickable()
        .SetElementIndex(1)
        .SendKeys("hello");
}
```

When using Chrome driver (https://chromedriver.storage.googleapis.com/index.html?path=72.0.3626.7/), to be able to use `DriverPool.Instance.CreateReusableDriverFromLastRun()`, versions before 73 must be used. Later versions set W3C standard implementation as default and `CreateReusableDriverFromLastRun()` does not work with them.

5. **Smart Search**

For search by anchor and search by two anchors, it is important to provide the texts correctly.

In the above example, there is one text for anchor and no text for search element is needed.

The text displayed on the Web page is ‘Create account’, matching with the text in html code

```
<div id="CredentialsPageTitle" class="row text-title" role="heading" aria-level="1" data-bind="text: strings.pageTitle">Create account</div>
```

Okapi algorithm works based on the own text of a tag in html code, not the text displayed on the page. Most of the time, both are matched but not necessary. To be precise, we need to capture tag own texts from html code (using inspect of Chrome browser) but this is not intuitive and requires more effort.

It is always possible that while typing/copying and pasting the texts, we make mistakes. The common human mistakes can be wrong cases, extra spaces, missing parts of the texts, etc.

Okapi smart search will anticipate these human mistakes and try to compensate for them in the search algorithm.

For the above example, the smart search returns the same result if we enter the text as ‘Create account’, ‘create account’, ‘creaTe acccouNt’, ‘CREATE ACCOUNT’, ‘create accoUnt’, ‘create   acc’, etc.

By default, smart search is OFF. Smart search can be applied globally through ITestEnvironment implementation when configuration is class injected

```
internal class TestEnvironmentConfig : ITestEnvironment
{
    public DriverFlavour DriverFlavour => DriverFlavour.Chrome;
    public bool RemoteDriver => false;
    public Uri SeleniumHubUri => new Uri("http://localhost:4444/wd/hub");
    public bool Log => true;
    public bool QuitDriverOnError => true;
    public bool QuitDriverOnFailVerification => true;
    public bool TakeSnapshotOnOK => false;
    public bool TakeSnapshotOnError => true;
    public string SnapshotLocation => "Snapshots";
    public bool SmartSearch => true;
}
```

Or through app.config (overwrites class configuration injection)

```
<EnvironmentSection>
    <Environments>
      <add targetTestEnvironment="Cloud9"
        active="true" driverFlavour="Chrome"
        remoteDriver="false"
        driverTimeoutInSeconds="120"
        quitDriverOnError="true"
        quitDriverOnFailVerification="true"
        log="true"
        takeSnapshotOnOK="false"
        takeSnapshotOnError="true"
        snapshotLocation="Snapshots"
        smartSearch="true" />
    </Environments>
  </EnvironmentSection>
```

Users are allowed to overwrite global smart search setting at test object level via ITestObject method `SmartSearch(bool)`

```
TestObject
    .New(SearchInfo.OwnText("create  AccoUnt"), SearchInfo.New("input"))
    .SetShortestDomDistanceDepth(2)
    .SetElementIndex(1)
    .SmartSearch(true)
    .SendKeys("hello");
```

6. Basic Usages

```
TestObject.New(SearchInfo.New("anchor tag Name", "anchor tag own text"), "search element tag
name")

TestObject.New(SearchInfo.New(HtmlTag.label, "anchor tag own text"), "search element tag
name")

TestObject.New(SearchInfo.New("anchor tag Name", "anchor tag own text"),
SearchInfo.New("search element tag Name", "search element tag own text"))

TestObject.New(SearchInfo.TagName("anchor tag Name"), SearchInfo.New("search element tag
Name", "search element tag own text"))

TestObject.New(SearchInfo.OwnText("anchor tag own text"), SearchInfo.New(("search element tag
Name"))

TestObject.New(SearchInfo.OwnText("anchor tag own text"), SearchInfo.New(HtmlTag.div))

TestObject.New(SearchInfo.Newt("anchor tag Name", "anchor tag own text"))
...
```

Any tag name text used above can be a single tag name, i.e. 'div' or a CSS locator.

It is recommended that where needed, use simple direct child CSS combinators, i.e. `'ul>li>span'` or descendant CSS combinators, i.e. `'div span'` to improve the search accuracy.

Example:

A Cancel button close to a label 'Confirmation'

```
TestObject.New(SearchInfo.New("h2", "Confirmation"), SearchInfo.New("span", "Cancel"));
```

A text box close to label 'First Name'

```
TestObject.New(SearchInfo.New("h2", "First Name"), SearchInfo.TagName("input"));
```

Special cases where Anchor is also Search element, i.e. a button Next can be defined

```
TestObject.New(SearchInfo.New("button", "Next"))
```

7. Dynamic Contents

When multiple Web elements on a Web page are quite similar, dynamic contents can be set for a TestObject instance and it can be used to point to different real Web elements on the Web page by just changing dynamic contents.

Example 1:

On a Web page, there are input box labelled 'First Name' and another one labelled 'Last Name', we can define a TestObject as below:

```
var genericElement = TestObject.New(SearchInfo.New("label", "{0}"),
SearchInfo.TagName("input"));
```

Where {0} is dynamic contents for anchor.

When used for First Name text box

```
genericElement.SetAnchorDynamicContents("First Name").SendKeys("John");
```

When used for Last Name text box

```
genericElement.SetDynamicContents("Last Name").SendKeys("Doe");
```

`SetDynamicContents()` can also be used as the replacement of `SetAnchorDynamicContents()`. `SetDynamicContents()` method were created to use mainly for traditional search methods. However, it can also be used for search by anchor and search by two anchors. It will set dynamic contents for anchor in search by anchor and search by two anchors.

Example 2:

On a Web page, there is a dropdown with multiple dropdown items, we can define a TestObject instance for multiple dropdown items as in the example below:

```
var dropDownItems = TestObject.New(SearchInfo.New("label", "Product Type"), SearchInfo.New("p-
dropdownitem>li>div>div", "{0}"));
```

When used to select Book item

```
dropDownItems.SetSearchElementDynamicContents("Book").Click();
```

When used to select Computer item

```
dropDownItems.SetSearchElementDynamicContents("Computer").Click();
```

`SetSearchElementDynamicContents`() is used to set dynamic contents for search Web element.

8. Enhance Search Accuracy

For situation where anchor tag own text and/or search element tag own text are quite unique on a Web page, tag names can be omitted.

When we want to narrow down the search, expecting higher level of possible accuracy, try to include tag names or even CSS locators.

9. Search By Two Anchors

Everything is similar to search by anchor, except for the fact that there are two anchors. The first anchor is named ‘parent anchor’. The second anchor is called ‘anchor’.

Anchor is supposed to be near parent anchor. And search element is supposed to be near anchor.

```
TestObject.New(SearchInfo.New(“parent anchor tag Name”, "parent anchor tag own text"),
SearchInfo.New(“anchor tag Name”, "anchor tag own text"), “search element tag name”)

TestObject.New(SearchInfo.New(“parent anchor tag Name”, "parent anchor tag own text"),
SearchInfo.New(“anchor tag Name”, "anchor tag own text"), SearchInfo.New(“search element tag
Name”, "search element tag own text"))
...
```

`SetParentAnchorDynamicContents()` is used to set dynamic contents for parent anchor.

One real example of the two anchor search is on a Web page, there is a question (parent anchor), then multiple answers (each answer is an anchor), and next to each answer is a checkbox (search element).