Day 3: Interaction with Data and Locators

# Interacting with dropdown menus

Most dropdown widgets on the web are built using the *select* element. The *select* element defines a list of selectable options to pick from. Rather than executing a findElement or findElements call against the containing *select* and its internal *option* elements, Selenium provides a simple abstraction located in the *support.ui* package.

Its usage is simple: locate the *select* element on the page, and then wrap the returned WebElement in a call to the Select class. If all goes well, then you can use the Select class’s methods to perform common operations on a dropdown list, such as selecting/deselecting by index, or by value, or checking the currently selected option. A code example follows:

Select selectElement = **new** Select(driver.findElement(By.*tagName*("select")));

selectElement.selectByIndex(0);

Note that some web pages do not use the *select* element to model their dropdown lists and instead rely on dynamically generated HTML to build their dropdowns. In these cases, this helper class cannot be used. The reader is encouraged to verify that an element on screen is actually a *select* element before proceeding with using this class in their code.

# Interacting with tables

HTML tables are a common way for displaying tabular data in a web application. Almost every test automation engineer will encounter a situation in which they have to work with them to some degree. Common scenarios include verifying that a table is sorted, fetching rows from a table, fetching columns from a table, and iterating over the rows of a table.

A table is represented through a combination of html tags. The <table> tag encapsulates the entire table. Optionally, <th> tags can be used for table headers, but aren’t always leveraged. Each <tr> tag will represent a table row and each <td> tag will represent an individual cell of data. Td, in this instance, stands for table data.

The first step to working with any table is reliably identifying the table itself, as all elements within will be children of this element and, consequently, rely on this identifier. For the purpose of this example, we will talk in terms of xpaths.

<html>

<body>

<table>

<tr>

<th>**A**</th>

<th>**B**</th>

<th>**C**</th>

<th>**D**</th>

</tr>

<tr>

<td>**1**</td>

<td>**2**</td>

<td>**3**</td>

<td>**4**</td>

</tr>

<tr>

<td>**a**</td>

<td>**b**</td>

<td>**c**</td>

<td>**d**</td>

</tr>

<tr>

<td>**!**</td>

<td>**@**</td>

<td>**#**</td>

<td>**$**</td>

</tr>

<tr>

<td>**)**</td>

<td>**(**</td>

<td>**\***</td>

<td>**^**</td>

</tr>

</table>

</body>

</html>

Take the above document for example. It contains a table with a single row of table headers and several rows of data. As stated above, the starting point for any work with tables will be reliably identifying the table itself. In this particular document, we can do this with a simple XPath (//table).

If we wanted to fetch a list of all of the table header elements (and had a WebDriver variable called driver), we could do that quite simply:

List<WebElement> headers = driver.findElements(By.xpath(“//table/tr[1]/th”));

This can be useful when attempting to report data of a cell to the test report or log. Identifying the column header and row number along with any problematic data will be useful for the engineer analyzing the results to accurately identify problem areas in the application while filing bug reports.

If we want a collection of all of the rows in the table, this can be done quite easily as well:

List<WebElement> rows = driver.findElements(By.xpath(“//table/tr”));

In this particular case you’ll want to be careful to handle the row of table headers separately from the rows of data. Selenium will ensure, however, that the elements in your list are in the same order that they appear in the markup. This means that, for any table which is guaranteed to have header elements, headers will always be in the first position in the list returned by the above code.

If you want to fetch all of the data elements of a particular column, we have to get slightly more creative with our identifier:

List<WebElement> columnData = driver.findElements(By.xpath(“//table/tr/td[3]”));

The above xpath describes any table data element which has an index of 3 and is a child of a table row element. Therefore, we will grab the data belonging to the entire third column. It is important to note that this only works because we did not specify an index along with ‘tr’. This means that we will grab the third td of every row rather than just a particular row.

Combining this information with our knowledge of Java, it becomes very easy to iterate over any such collection. For example, if we wanted to iterate over all of the rows of the table, we could do so as follows:

List**<**WebElement**>** rows **=** driver**.**findElements**(**By**.**xpath**(**"//table/tr"**));**

**for(**WebElement row **:** rows**)** **{**

//do something with each row element

**}**

Similarly, if we had a desire to iterate over all of the data elements of a particular column:

List**<**WebElement**>** data **=** driver**.**findElements**(**By**.**xpath**(**"//table/tr/td[3]"**));**

**for(**WebElement datum **:** data**)** **{**

//do something with the third td of each row

**}**

Since tables are very concretely fixed in two dimensions, they become easy to navigate and, in fact, quite friendly to automation. If you discover a need to work with tabular data in a web application, first take stock of whether or not the information is actually organized into a table element. If it is, you’re quire in luck as this will generally be easier to work with than the alternative.

# More on Locators

Locator expressions offer a wide range of capabilities to identify elements on the page. As discussed in earlier lessons, the most common locator expressions are by **id**, **CSS Selector**, and **XPath**. In this section, more detail will be provided on utilizing the expressive power of XPath in Selenium.

Our focus will be on using some common XPath functions to more flexibly define elements on the page. This section will cover: *concat(), substring(), contains(), normalize-space(), position()*.

*concat()*

The concat() function joins two or more strings, returning the resulting string.

*substring()*

The substring() function returns *length* characters starting at a *position x*. The method signature is substring(stringToTest, startIndex, length) where the length parameter defaults to the length of stringToTest if not provided.

*contains()*

The contains() function checks whether a certain input string contains a certain substring.

*normalize-space()*

The normalize-space() function strips leading and trailing whitespace, and turns multiple adjacent spaces into a single space. It takes as input a string.

*position()*

The position() function returns the number equal to the context position from the expression evaluation context. For example, the XPath *//a[position() = 2]* returns the second *a* element anywhere in the document.

# Exercise 2

Suppose we have the following HTML:

<ul id="menu">

<li>This is Item 1</li>

<li>This is Record 2</li>

<li>This is Item 3</li>

<li> This is Item 4</li>

<li>This is Record 5</li>

</ul>

Our task is to select all <li> elements with the word ‘Item’ exactly one space after the ‘This is’ portion of the text. We want to ignore extraneous whitespace and treat each string as if each word was separated by a single space. The following XPath selects the 1st, 3rd, and 4th <li> elements:

*//\*[@id=’menu’]/li[substring(normalize-space(), 9, 13)]*

This XPath selects our containing element, then selects all its <li> children that satisfy the above condition. *normalize-space()* without an argument implicitly receives the current element node’s text content. In this case, the current node examined is any of the <li> nodes.