ACAMICA

Globant
we are ready

Level 4

Selenium Webdriver

4.1 - Introduction and Locators

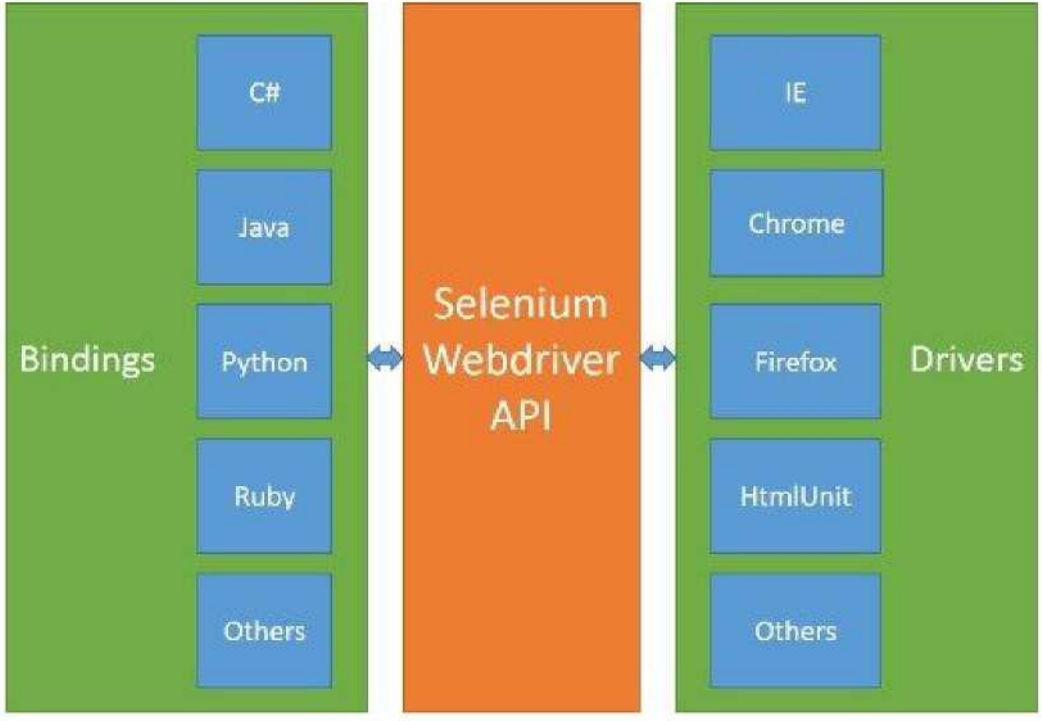
Selenium is an automation framework for web applications.

Selenium was developed in 2004, then, in 2007, Webdriver was developed, a tool that joined the Selenium project in 2009, creating what we know as Selenium Webdriver or Selenium 2.0.

Today Selenium Webdriver is in version 3. With Selenium we can perform actions on a web browser, just as a regular user would do. This works thanks to a specific driver implemented in each of the browsers, which sends the commands to the browser and returns a result.

There is a driver for each of the browsers, whether it is Firefox, Chrome, IE, Safari, etc.

The way in which the communication between Selenium Webdriver and the browsers is carried out is the following:


Bindings
C#
Java
Python
Ruby
Others
Selenium Webdriver API
IE
Chrome
Firefox
HtmlUnit
Others
Drivers

4.2 - Getting Started with WebDriver

Locators

Imagine a scenario where we have to automate a web application with Selenium Webdriver, where we have to simulate a user using the application to be able to validate certain result, whether it is data entered, the existence of an element, a message on the screen, etc.

As users, we browse the applications by clicking on links, we also fill out forms by selecting data from combo boxes or text fields. Selenium Webdriver allows us to identify each of these elements in different ways, and then do something with them.

Let’s see then what the element locators supported by Selenium Webdriver are:

To identify elements in a website, we will use a tool that makes it easy to inspect them, in this case, firebug, a Firefox add-on that helps with the task. It should be noted that it is not necessary to use it, and that we can do the same by looking at the source code of the page, but it helps a lot.

For this example, we will use the following link:

https://www.wikipedia.org/.

ID: The ID is a type of locator that should be unique per element. In case of having elements with an ID, this is the recommended way to locate them.

By analyzing the HTML code of a website, we can find the locators. For example, let’s see Wikipedia’s home page:

<input id="searchlnput" name-'search" size="20" autofocus-'autofocus" accesskey="F" dir="auto" results="10" autocomplete-'off' list="suggestions" style-'padding-right: 64px;" type="search">

Name: This identifier should be the second in preference, since it tends to be unique, although it is not always so, and allows us to clearly identify elements with a descriptive name.

Going back to our previous example, we will see that we can identify the field using the name search.

Link: This identifier allows us to identify elements of link type (a), using its text.

In Wikipedia’s home, we have a list of links with country names, each can be identified by the text. Let’s see this example:

Español
<small>1 306 000+ artículos</small>

XPath: Xpath is a type of locator that allows us to navigate the html element tree translated into xml, in this way we can identify any element in the page that we are using. This method is effective when we do not we have IDs or names to identify an element, but it is the least recommended given its slowness and high maintenance.

Let’s see how we can locate the search field we obtained by name and ID using Xpath:

Xpath= //*[@id-search Input']

In addition to these locators, there are others, such as:

- DOM.
- CSS.
- Ill-element.

Recommended Locators

As we saw earlier, we can identify elements in different ways; unfortunately, not all of them are as effective or stable, for example, if an element changes place, its xpath would change, or if a link changes its text, we should also redefine it in our test cases.

Therefore, it is good practice to use more stable locators, which last over time, as a good automation strategy.

The recommended order of preference is then, the following:

1. ID
2. Name
3. CSS
4. XPATH

The ID is unique, and even if an element changes, if we have it identified it its ID, it will keep working.

The name may not be unique, but it usually is, therefore it is our second option.

CSS and xpath are useful when we do not have the previous options, but they are less stable and slower.

When to Use Xpath?

As we mentioned earlier, we can use xpath when we do not have other elements, such as ID or name, but it is not the only occasion where we can use it.

As an example, suppose we have an HTML table, if you are not familiar with the structure of a table, we can think of it as a list of rows containing cells.

Let’s see the table we have as example in:
http://www.w3schools.com/html/html tables.asp.

Company	Contact	Country
Alfreds Futterkiste	Maria Anders	Germany
Centro comercial Moctezuma	Francisco Chang	Mexico
Ernst Handel	Roland Mendel	Austria
Island Trading	Helen Bennett	UK
Laughing Bacchus Winecellars	Yoshi Tannamuri	Canada
Magazzini Alimentari Riuniti	Giovanni Rovelli	Italy

If we wanted to identify cell 1:1 through xpath, we will see that it is:

.//*[@id-customers']/tbody/tr[2]/td[1]

This is, second row (tr[2]) column 1 (td[1]).

Let’s see what the xpaths for the 3 cells in the first **row** are:

- //*[@id-customers']/tbody/tr[2]/td[1]
- //*[@id='customers']/tbody/tr[2]/td[2]
- //*[@id='customers']/tbody/tr[2]/td[3]

And now the xpaths for the third **column**, with each of the cells:

- //*[@id-customers']/tbody/tr[2]/td[3]
- //*[@id='customers']/tbody/tr[3]/td[3]
- //*[@id='customers']/tbody/tr[4]/td[3]
- //*[@id='customers']/tbody/tr[5]/td[3]
- //*[@id='customers']/tbody/tr[6]/td[3]
- //*[@id='customers']/tbody/tr[7]/td[3]

As we can see, we can leave a row fixed and iterate the columns increasing its index in 1, or leave a column fixed and change the row increasing its index in 1.

If we wanted to have a function that adds values to a table, we could use this to iterate cells in a column in a table, get the values, add them and return the result.

Advanced Xpath

Xpath allows us to use different functions in order to identify elements in different ways.

Let’s see some of them:

Contains:

We usually use this to find an item containing certain text, for example, in the example mentioned above, if we wanted to get the cell with the text “Roland” we could do it as follows:

```
//*[@id='customers']/tbody/tr/td[contains(text(),"'Roland")]
```

Basically, this function returns the cell (TD) containing “Roland”, and inside an element with ID = customers.

It is also possible to concatenate conditions together, for example, that a cell contains Roland but in turn contains Mendel:

//*[@id='customers']/tbody/tr/td[contains(text(),"Roland") and contains(text(), "Mendel")]

Last: “Last” helps us locate the last item in a list. By way of example, suppose we want to get the last cell of the last row and the last column, we could do it using the following xpath:
//*[@id-customers']/tbody/tr[last()]/td[last()]

Or the next to last, with:

//*[@id='customers']/tbody/tr[last()-1]/td[last()]

Starts-With: This command allows us to locate an element with a text, ID, name, etc. that starts with certain text, for example, to select Roland Mendel’s column, using starts-with, we could say:

//*[@id='customers']/tbody/tr[4]/td[starts-with(text(),"Roland")]

4.3 - The Page Object Model (POM) Pattern

The PageObject pattern allows us to organize the elements that we are going to use in our tests. If we define and declare the elements within our test cases, we will have a high cost of maintenance, as well as little code reusability.

For this, there is this pattern based on declaring our pages, whether they are complete or only fragments, in a code class, which we can then use in our tests.

We will then define, using the Wikipedia example, a page. For that, we will use wikipedia.org's home page:

The first thing we will do is create a new package to keep our code organized, we will call it: com.automation.training.pages.

Within it, we will create a class called BasePage, and a class called WikiHomePage, and we will define the elements that we are going to use.

The BasePage class will contain every method common to all pages, as well as who will access the selenium driver. Let’s see its content:

```
public abstract class BasePage {

    private WebDriver driver;

    public BasePage(WebDriver pDriver) {
        PageFactory.initElements(pDriver, this);
        driver = pDriver;
    }

    protected WebDriver getDriver() {
        return driver;
    }

    public void dispose() {
        if (driver != null) {
            driver.quit();
        }
    }
```

It receives in its constructor an object of the Webdriver type, which has already been created, but we will see in more detail later. Basically, that object is an instance of the driver we are using.

Then, in its constructor, it calls PageFactory, which is the method responsible for initializing the elements defined in the page.

This class also has a dispose method, which is responsible for closing the driver; this closes the open instance of the browser as well as all the open sessions.

Returning to our WikiHomePage, we will implement the elements that we are going to use, in this case, the search field and the search button:

```
public class WikiHomePage extends BasePage{

        public WikiHomePage(WebDriver driver) {
            super(driver);
            driver.get("http://wikipedia.org");
        }

        @FindBy(id="searchInput")
        private WebElement searchInput;
        @FindBy(xpath="//*[@id='search-form']/fieldset/button")
        private WebElement searchButton;
```

As we can see, Selenium allows us to use a very simple syntax for declaring the elements of a page, in this case we use the annotation **@FindBy** and declare them by ID and xpath.

We declare them always as private, since the page is responsible for using them and exposing only the functionalities that we are interested in. For this example, we are going to implement a search, which should be defined in the page. Since the search returns a new page, in this case, one with an article, we will create a class, for now empty, called **ArticlePage**. It is good practice to have each action of a page returning another one return the other page ready to be used in the tests.

```
public ArticlePage buscar(String busqueda) {
    searchInput.sendKeys(busqueda);
    searchButton.click();
    return new ArticlePage(getDriver());
```

As we can see, this function receives a search string and then calls **searchInput -> SendKeys** (in charge of writing in the field) with the string. Then, it clicks the search button, returning a new page called **ArticlePage**.

Let’s now create the **ArticlePage**, which is the page that will contain the elements of the article. Suppose we want to do a couple of validations in that page, for example, that the title appears according to the search, and that it has a link to Wikipedia’s privacy policies at the bottom.

For this, we need to define those two elements, the title and the link, to be able to do a validation in our tests. We should also create a method to return the content of the title, to validate that it is according to the Search string:

We already have two pages, and we can create our first test, as we will see later.

```
public class ArticlePage extends BasePage{

    public ArticlePage(WebDriver driver) {
        super(driver);
    }

    @FindBy(id="firstHeading")
    private WebElement pageTitle;

    @FindBy(linkText="Privacy policy")
    private WebElement privacyLink;

    public String getPageTitle() {
        return pageTitle.getText();
    }
}
```

4.3.1 - Legacy between Pages

It is worth mentioning that one page can inherit from another one, to make our code more reusable and understandable.

In the Wikipedia example, we will see that many pages maintain the left column and the header; therefore, it is a good practice when designing our pages to create a page with that content and make the others, such as **ArticlePage**, inherit from that, in this way, we can access all the menu items from any page, without re-declaring the element in each of them.

4.4 - Our First Test Case

A test case is a set of actions with a validation, which serves to verify that a functionality does what it is expected to do.

The actions are those that we have previously defined in the created pages, that is, in the example we have so far, we could create a test that does a search on Wikipedia, and then validate the result page.

Before writing the test case, let’s see how we can structure our classes to have a secure framework.

We will organize our code in a new package, in this case:
com.automation.training.tests

BaseTests:

This class will contain an instance of **MyDriver**, which is the class that initializes our driver, according to the parameter that we pass through the **suite.xml** of TestNg, as follows:

```
public class MyDriver {

    private YlfebDriver driver;

    public MyDriver(String browser) {
        switch (browser) { case
        "firefox":
            driver - new FirefoxDriverQ;
            break;
        case "chrome":
            System.setPropertyC'webdriver.chrome.driver", "/Users/santiago.hernandez/Documents/chromedriver");
        driver - new ChromeDriver(); break; default: break;
        }
    >

    public WebDriver getDriver() {
        return this.driver;
    >
```

As we can see, the constructor receives a **Browser** parameter and based on it, it initializes one or the other. It is a very simple class that also adds a getDriver method that allows us to access it.

Going back to our **BaseTests** class:

```
public class BaseTests {

    MyDriver myDriver;

    private WikiHomePage wikiHome;

    @BeforeSuite(alwaysRun=true)
    @Parameters({"browser"})
    public void beforeSuite(String browser) {
        myDriver = new MyDriver(browser);
        wikiHome = new WikiHomePage(myDriver.getDriver())
    }

    @AfterSuite(alwaysRun=true)
    public void afterSuite() {
        wikiHome.dispose();
    }

    public WikiHomePage getWikiHomePage() {
        return wikiHome;
    }
```

This class handles three important things:

@BeforeSuite:
It is responsible for initializing the myDriver variable with the corresponding browser; in turn, it initializes the **wikiHomePage** page, which in its constructor, as we saw earlier, initializes the browser and goes to wikipedia.org.

@AfterSuite
This method is responsible for calling the Dispose method, which as we saw previously, it is responsible for closing the browser.

getWikiHomePage
This method returns the instance of the homePage that we are going to use.

Having then our **BaseTests** class, which includes all the methods that we will use in our tests, we can create our specific class for test cases, which we will call **WikiTests**, and which will extend from **BasePage**.

It will contain a method called **testWikiSearch**, which will be our test case; for this reason we will write it with the annotation @**Test**.

```
public class WikiTests extends BaseTests{


    @Test
    public void testWikiSearch() {
        WikiHomePage home = getWikiHomePage();
        ArticlePage articlePage = home.buscar("Java");
        Assert.assertEquals(articlePage.getPageTitle(), "Java");
    }

}
```

As we see, it is very simple to understand:

1 - It initializes the homePage

2 - It searches for “Java” and then assigns the result page to the articlePage variable

3 - It makes an assertion to validate that the title of the destination page is “Java”

4.5 – Waits

Waits in Selenium WebDriver are how we are going to wait for a certain element before performing an action, for example, before clicking a button, or before completing a field.
There are 3 main waits:

- **pageLoadTimeout**: It is the maximum time to wait for a page to load before showing an error
- **setImplicitTimeout**: It is the maximum time to wait for an element if it is not present
- **setScriptTimeout**: Time to wait for a Javascript function to end

Recommendations to use waits:

- Never use a fixed time to wait for an item
- Never use **Thread.sleep()**

Explicit waits:
An explicit wait is defined as the act of actively waiting for something to happen before executing an action, in WebDriver we refer to this as **Explicit Wait:**

```
WebDriverWait wait = new WebDriverWait(aDriver, 10);
wait.until(ExpectedConditions.elementToBeClickable(anElement));
```

In this example, we see how we ask WebDriver to wait 10 seconds for the element to be clickable: if it is clickable before, the action will be executed before, if the 10 seconds pass, an error will be returned.

Example

We will create an explicit wait in our **BasePage**, for that we will add the following lines to it:

```
public ArticlePage buscar(String busqueda) {
    searchInput.sendKeys(busqueda);
    getWait().until(ExpectedConditions.elementToBeClickable(searchButton));
    searchButton.click();
    return new ArticlePage(getDriver());
}
```

Example

Then, from our pages, we can call the **getWait()** function to wait for an item, for example, the search button:

```
public ArticlePage buscar(String busqueda) {
    searchInput.sendKeys(busqueda);
    getWait().until(ExpectedConditions.elementToBeClickable(searchButton));
    searchButton.click();
    return new ArticlePage(getDriver());
```

Which Wait Strategy should we Use?

Explicit wait

✓ Documented and well defined
✓ Runs in the local part of selenium
✓ Works under any condition you can think of
✓ Returns error or pass
✓ You can define absence of elements as a condition of success
✓ You can configure delays between retries

Implicit wait

✓ Undocumented, unpredictable result
✓ Runs in the remote part of selenium
✓ Only works in findelements
✓ Returns only element not found or element found
✓ To check the absence of elements, we must always wait for the timeout
✓ You cannot customize beyond the global option

We strongly suggest to always use explicit waits, we even suggest that the implicit should always equal **0**.

4.6 - iFrames - Alerts - Popups

When we have iFrames or Alerts that we want to manage, we need to change the context in which Selenium WebDriver works.

For selenium, an iFrame is another page, which is managed on a different context, therefore, if we look for an element of iFrame that is in the parent page context, we will not find it without first passing the driver to the iFrame, as follows:

driver.switchTo().frame("id-del-iframe");

Once this is done, we are inside the iFrame and we can continue working as we have been doing.

An alert is a browser popup with certain message, it is usually a message with options like accept or cancel. In order for selenium to be able to handle them, we have to somehow tell it to do something with it, for that, we can use the following commands:

driver.switchTo().alert();

driver.switchTo().alert().accept();

When we have a popup that is a new page opening, we need to know the window handles. A handle is a unique identifier of each window, and we can obtain the list of them through:

driver.getWindowHandles()

This, if for example we have the original window and a popup, will return two handles, on which we can iterate to make the switch:

Set<String> handles = driver.getWindowHandles(); *//* get all window handles Iterator<String> iterator = handles.iteratorQ; while (iterator.hasNext()){ subWindowHandler = iterator.next(); } driver.switchTo().window(subWindowHandler)

Summary of the Level

At this level, we have learned the basic concepts required to begin to understand and use Selenium / WebDriver. We saw how to locate and interact with the elements of the application under test, as well as how to select the appropriate synchronization strategy and how to implement the PageObjects pattern.

By setting the general principles, we have laid the foundations to start developing an automation project using Maven, Java and Selenium/WebDriver.