Test Automation(Selenium)

Lecture 4 –

Controlling the Test Flow & More on Selenium API

IT Learning &
Outsourcing Center

Lector: Georgi Tsvetanov
Skype: georgi.pragmatic
E-mail: george.tsvetanov@hotmail.com
Facebook: https://www.facebook.com/georgi.tsvetanov.18

www.pragmatic.bg

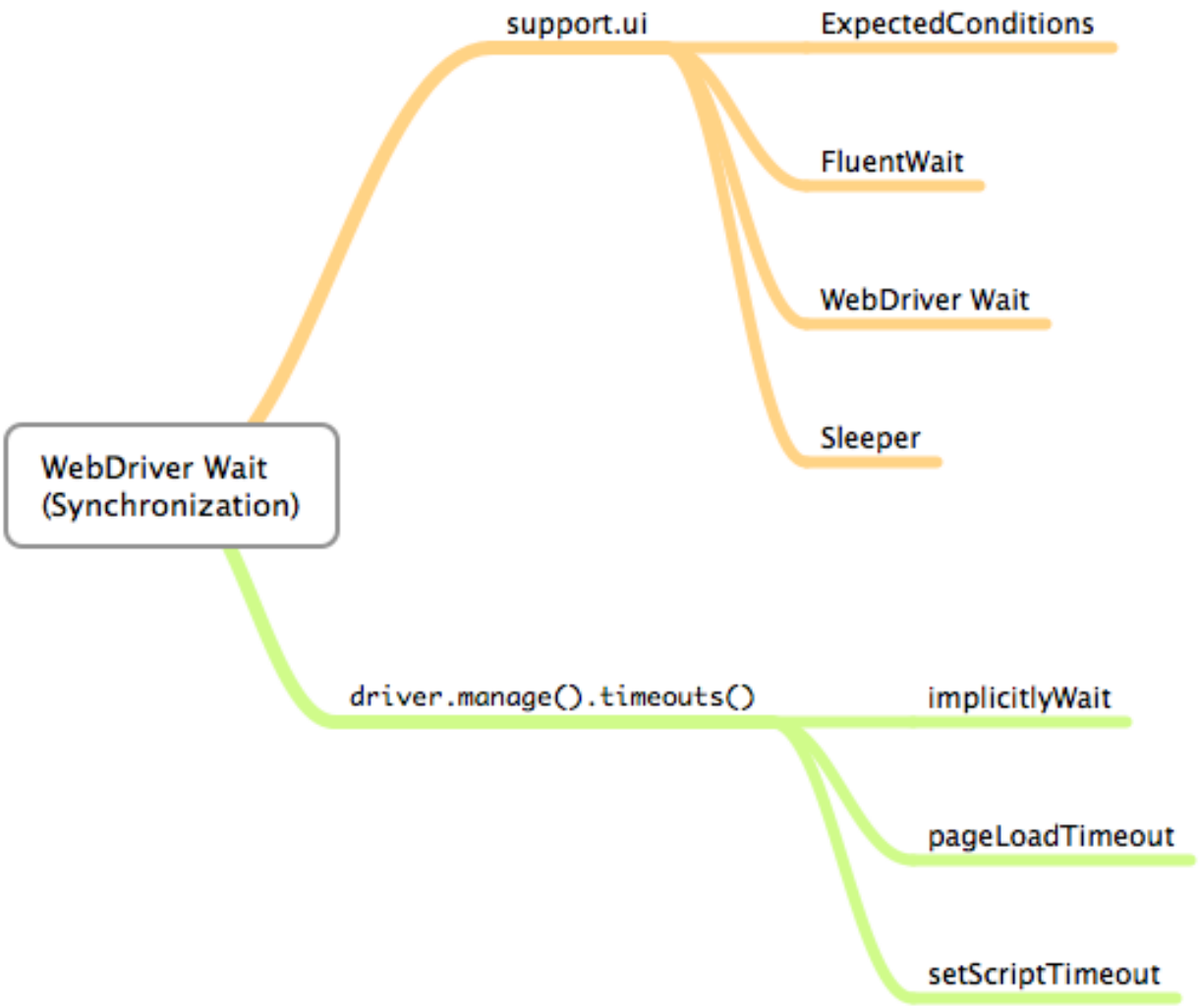
Summary - overall

- Synchronizing a test with an implicit wait
- Synchronizing a test with an explicit wait
- Synchronizing a test with custom-expected conditions
- Checking an element's presence
- Checking an element's status
- Identifying and handling a pop-up window by its name
- Identifying and handling a pop-up window by its title
- Identifying and handling a pop-up window by its content
- Handling a simple JavaScript alert
- Handling a confirm box alert
- Handling a prompt box alert
- Identifying and handling frames
- Identifying and handling frames by their content
- Working with IFRAME

Introduction

- While building test automation for a complex web application using Selenium WebDriver, we need to ensure that the test flow is maintained for reliable test automation.
- When tests are run, the application may not always respond with the same speed. For example, it might take a few seconds for a progress bar to reach 100 percent, a status message to appear, a button to become enabled, and a window or pop-up message to open.
- You can handle these anticipated timing problems by synchronizing your test to ensure that Selenium WebDriver waits until your application is ready before performing a certain step. There are several options that you can use to synchronize your test. Selenium RC has various waitFor methods; however, being a pure web automation API, Selenium WebDriver provides very limited methods for synchronization. In this lecture, you will see how to use the WebDriverWait class to implement synchronization in tests.

Wait - Mindmap



Synchronizing a test with an implicit wait (part 1)

- The Selenium WebDriver provides an implicit wait for synchronizing tests. When an implicit wait is implemented in tests, if WebDriver cannot find an element in the Document Object Model (DOM), it will wait for a defined amount of time for the element to appear in the DOM.
- In other terms, an implicit wait polls the DOM for a certain amount of time when trying to find an element or elements if they are not immediately available. The default setting is 0.
- Once set, the implicit wait is set for the life of the WebDriver object's instance. However, an implicit wait may slow down your tests when an application responds normally, as it will wait for each element appearing in the DOM and increase the overall execution time.
- We will briefly explore the use of an implicit wait, however, it is recommended to avoid or minimize the use of an implicit wait.

Synchronizing a test with an implicit wait (part 2)

- Lets explore the ImplicitWaitTest.java class in the code examples

Synchronizing a test with an explicit wait (part 1)

- The Selenium WebDriver also provides an explicit wait for synchronizing tests, which provides a better control when compared with an implicit wait. Unlike an implicit wait, you can write custom code or conditions for wait before proceeding further in the code.
- An explicit wait can only be implemented in cases where synchronization is needed and the rest of the script is working fine.
- The Selenium WebDriver provides WebDriverWait class and ExpectedCondition interface for implementing an explicit wait.
- The ExpectedConditions class provides a set of predefined conditions to wait before proceeding further in the code. The following table shows some common conditions that we frequently come across when automating web browsers supported by the ExpectedConditions class:

Synchronizing a test with an explicit wait (part 2)

Predefined condition	Selenium method
An element is visible and enabled	`elementToBeClickable(By locator)`
An element is selected	`elementToBeSelected(WebElement element)`
Presence of an element	`presenceOfElementLocated(By locator)`
Specific text present in an element	`textToBePresentInElement(By locator, java.lang.String text)`
Element value	`textToBePresentInElementValue(By locator, java.lang.String text)`
Title	`titleContains(java.lang.String title)`

- For more conditions, visit: https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/ui/ExpectedConditions.html

Synchronizing a test with an explicit wait (part 3)

- Lets explore the ExplicitWaitTest.java class in the code examples and more specific the test: testExplcitWaitTitleContains()

Synchronizing a test with custom-expected conditions (part 1)

- The Selenium WebDriver also provides a way to build custom-expected conditions along with common conditions using the ExpectedCondition class. This comes in handy when a wait can't be handled with a common condition supported by the ExpectedConditions class.

- In this recipe, we will explore how to create a custom condition.

Synchronizing a test with custom-expected conditions (part 2)

- Lets explore the ExplicitWaitTest.java class in the code examples and more specific the test: testExplicitWait()

Synchronizing a test with custom-expected conditions (part 3)

- Waiting for element's attribute value update
  - Based on the events and action performed, the value of an element's attribute might change at runtime. For example, a disabled textbox gets enabled based on the user's rights. A custom wait can be created on the attribute value of the element. In the following example, the ExpectedCondition class waits for a Boolean return value, based on the attribute value of an element:

```
(new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>()
{
  public Boolean apply(WebDriver d) {
    return d.findElement(By.id("userName")).
    getAttribute("readonly").contains("true");
   }
});
```

Synchronizing a test with custom-expected conditions (part 4)

- Waiting for an element's visibility
  - Developers hide or display elements based on the sequence of actions, user rights, and so on. The specific element might exist in the DOM, but are hidden from the user, and when the user performs a certain action it appears on the page. A custom-wait condition can be created based on the element's visibility as follows:

```
(new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>()
{
   public Boolean apply(WebDriver d) {
     return d.findElement(By.id("page4")).isDisplayed();
   }
});
```

Synchronizing a test with custom-expected conditions (part 5)

- Waiting for DOM events
  - The web application may be using a JavaScript framework such as jQuery for AJAX and content manipulation. For example, jQuery is used to load a big JSON file from the server asynchronously on the page. While jQuery is reading and processing this file, a test can check its status using the active attribute. A custom wait can be implemented by executing the JavaScript code and checking the return value as follows:

```
(new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>()
{
   public Boolean apply(WebDriver d) {
      JavascriptExecutor js = (JavascriptExecutor) d;
      return (Boolean)js.executeScript("return jQuery.active == 0");
   }
});
```

Checking an element's presence (part 1)

- The Selenium WebDriver doesn't implement Selenium RC's isElementPresent() method for checking if an element is present on a page. This method is useful for building a reliable test where you can check an element's presence before performing any action on it.

- In this recipe, we will write a method similar to the isElementPresent() method.

Checking an element's presence (part 2)

- Lets explore the ElementStateTests.java class in the code examples and more specific the test:

testIsElementPresent() with its' helper private method isElementPresent(By by)

Checking an element's status (part 1)

- Many a time a test fails to click on an element or enter text in a field as the element is disabled or exists in the DOM, but is not displayed on the page. This will result in an error being thrown and the test resulting in failures. For building reliable tests that can run unattended, a robust exception and error handling is needed in the test flow.
- We can handle these problems by checking the state of elements. The WebElement class provides the following methods to check the state of an element:

Method	Purpose
`isEnabled()`	This method checks if an element is enabled. Returns `true` if enabled, else `false` for disabled.
`isSelected()`	This method checks if element is selected (radio button, checkbox, and so on). It returns `true` if selected, else `false` for deselected
`isDisplayed()`	This method checks if element is displayed.

Checking an element's status (part 2)

- Lets explore the ElementStateTests.java class in the code examples and more specific the test: testElementIsEnabled()

Identifying and handling a pop-up window by its *name* (part 1)

- In Selenium WebDriver, testing pop-up windows involves identifying a pop-up window, switching the driver context to the pop-up window, then executing steps on the pop-up window, and finally switching back to the parent window.

- The Selenium WebDriver allows us to identify a pop-up window by its name attribute or window handle and switching between the pop-up window and the browser window is done using the driver.switchTo().window() method.

Identifying and handling a pop-up window by its *name* (part 2)

- In this recipe, we will identify and handle a pop-up window by using its name attribute. Developers provide the name attribute for a pop-up window that is different from its title. In the following example, a user can open a pop-up window by clicking on the Help button.

- In this case, the developer has provided HelpWindow as its name:

```
<button id="helpbutton"
onClick='window.open("help.html","HelpWindow",
"width=500,height=500");'>Help</button>
```

Identifying and handling a pop-up window by its *name* (part 3)

- Lets explore the WindowPopupTest.java class in the code examples and more specific the test: testWindowPopup()

Identifying and handling a pop-up window by its *title* (part 1)

- Many times developers don't assign the name attribute to pop-up windows. In such cases, we can use its window handle attribute. However, the handle attributes keep changing and it becomes difficult to identify the pop-up window, especially when there is more than one pop-up window open. Using the handle and title attributes of the page displayed in a pop-up window, we can build a more reliable way to identify the pop-up windows.

- In this recipe, we will use the title attribute to identify the pop-up window and then perform operations on it.

Identifying and handling a pop-up window by its *title* (part 2)

- Lets explore the WindowPopupTest.java class in the code examples and more specific the test:
  testWindowPopupUsingTitle()

Identifying and handling a pop-up window by its *content* (part 1)

- In certain situations, developers neither assign the name attribute nor provide a title to the page displayed in a pop-up window. This becomes more complex when a test needs to deal with multiple pop-up windows open at the same time—to identify the desired pop-up window.

- Ideally, you should reach out to developers and recommend that they add either the name attribute or title for better testability and accessibility.

- As a workaround to this problem, we can check the contents of each window returned by the driver.getWindowHandles() method to identify the desired pop-up window.

Identifying and handling a pop-up window by its *content* (part 2)

- Lets explore the WindowPopupTest.java class in the code examples and more specific the test:
  testWindowPopupUsingContents()

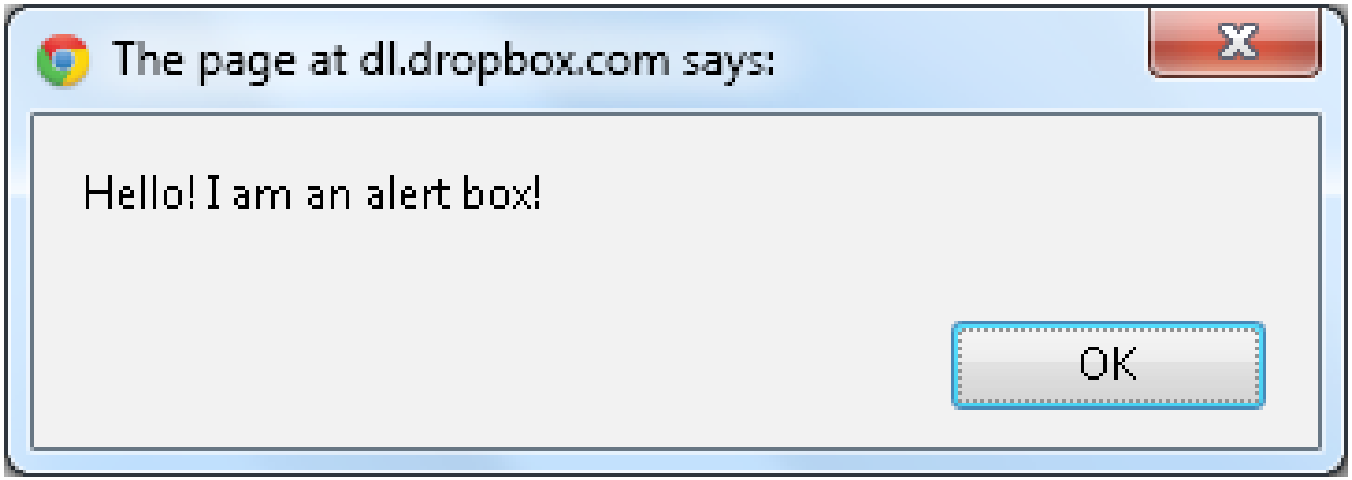
Handling a simple JavaScript alert (part 1)

- Web developers use JavaScript alerts for informing users about validation errors, warnings, getting a response for an action, accepting an input value, and so on.

- Tests will need to verify that the user is shown correct alerts while testing. It would also be required to handle alerts while performing an end-to-end workflow. The Selenium WebDriver provides an Alert class for working with JavaScript alerts.

Handling a simple JavaScript alert (part 2)

- In this recipe, we will handle a simple alert box using Selenium WebDriver's Alert class. A simple alert box is often used to notify the user with information such as errors, warnings, and success. When an alert box pops up, the user will have to click on the OK button to proceed, as shown in the following screenshot:



Handling a simple JavaScript alert (part 3)

- Lets explore the AlertsTest.java class in the code examples and more specific the test:
  testSimpleAlert()

Handling a confirm box alert (part 1)

- A confirm box is often used to verify or accept something from the user. When a confirm box pops up, the user will have to click either on the OK or the Cancel button to proceed, as shown in the following screenshot:

- If the user clicks on the OK button, the box returns true. If the user clicks on the Cancel button, the box returns false.
- In this recipe, we will handle a confirm box using the Selenium WebDriver's Alert class.

Handling a confirm box alert (part 2)

- Lets explore the AlertsTest.java class in the code examples and more specific the test: testConfirmAccept() and testConfirmDismiss()

Handling a prompt box alert (part 1)

- A prompt box is often used to input a value by the user before entering a page. When a prompt box pops up, the user will have to click either on the OK or the Cancel button to proceed after entering an input value, as shown in the following screenshot:

- If the user clicks on the **OK** button, the box returns the input value. If the user clicks on the **Cancel** button, the box returns null.
- In this recipe, we will handle a prompt box using the Selenium WebDriver's Alert class.

Handling a prompt box alert (part 2)

- Lets explore the AlertsTest.java class in the code examples and more specific the test:
  testPrompt()

Identifying and handling frames (part 1)

- HTML frames allow developers to present documents in multiple views, which may be independent windows or sub-windows. Multiple views offer developers a way to keep certain information visible, while other views are scrolled or replaced. For example, within the same window, one frame might display a static banner, the second a navigation menu, and the third the main document that can be scrolled through or replaced by navigating in the second frame.

Identifying and handling frames (part 2)

- A page with frames is created by using the <frameset> tag or the <iframe> tag. All frame tags are nested with a <frameset> tag. In the following example, a page will display three frames, each loading different HTML pages:

```
<html>
  <frameset cols="25%,*,25%" FRAMEBORDER="NO" FRAMESPACING="0" BORDER="0">
     <frame id="left" src="frame_a.htm" />
     <frame src="frame_b.htm" />
     <frame name="right" src="frame_c.htm" />
  </frameset>
</html>
```

- Frames can be identified by an ID or through the name attribute. In this recipe, we will identify and work with frames by using the driver.switchTo().frame() method, using the *id*, *name*, instance of WebElement, and the index of a frame.

Identifying and handling frames (part 3)

- !!!IMPORTANT WARNING!!!:
  - Warning: While working with multiple frames, when an operation is completed on a frame and a test flow needs to move to another frame, calling the driver.switchTo().frame() method will not move the context to the desired frame. The test will first need to activate the main document by calling driver.switchTo().defaultContent() and then activating the desired frame.

Identifying and handling frames (part 4)

- Lets explore the FramesTest.java class in the code examples and more specific the tests:
  - By *name* testFrameWithIdOrName()
  - By *index* testFrameByIndex()

Identifying and handling frames by their content (part 1)

- While working with frames, you will find that the id or name attributes are not defined. Still frames can be identified by using their index. This may not be a reliable way when applications are dynamic and there is a need to ensure that the correct frame is activated.

- In this recipe, we will identify frames by the content of the document loaded in these frames to make tests more reliable.

Identifying and handling frames by their content (part 2)

- Lets explore the FramesTest.java class in the code examples and more specific the test:
  testFrameByContents()

Working with IFRAME (part 1)

- Developers can also embed external documents or documents from another domain using the <iframe> tag. Various social media websites provide buttons which can be embedded in your web applications to link these websites. For example, you can add Twitter-follow button in your application as follows:

```
<iframe allowtransparency="true" frameborder="0" scrolling="no"
src="http://platform.twitter.com/widgets/follow_button.html?screen_
name=strahinski" style="width:300px; height:20px;"></iframe>
```

- Identifying and working with the <iframe> tag is similar to a frame created with the <frameset> tag. In this recipe, we will identify the <iframe> tag, which is nested in another frame.

Working with IFRAME (part 2)

- Lets explore the FramesTest.java class in the code examples and more specific the test:
  testIFrame()

Create a TestSuite and Run it in JUnit4

- You would normally want to run all your tests together and not run them separately one from another. In our case we're using JUnit4, so let's explore AllTests.java in the code examples in order to see how this is done.

Questions?