

# E2E Combinatorial Testing Tool

# GWS

## Developer's Guide

### Revision History

Version	Date	Author	Description
Draft	6/01/2017	Masayuki Otoshi	Document Created
1.0.0	12/17/2017	Masayuki Otoshi	Added Combinatorial testing
1.0.1	1/6/2018	Masayuki Otoshi	Changed package name to gws
1.0.2	3/20/2018	Masayuki Otoshi	Added switch expression in label
1.0.3	4/26/2018	Masayuki Otoshi	Removed pageld, error from GwAction

## Table of Contents

1. Introduction .....	3
2. Installation .....	3
Dependencies.....	3
Install GWS.....	4
Check Versions of Jar dependencies .....	4
Use different version.....	4
Add Your Own Jar .....	4
3. Getting Started .....	6
Hello World .....	6
Wizard HTMLs .....	7
Key-value pair (Object literal).....	10

Combinatorial Testing .....	12
Run from TestNG .....	14
4. Value types in Test Script .....	16
@import, @include .....	16
String, Number .....	17
Boolean .....	17
JavaScript function .....	18
JavaScript array .....	18
JavaScript object .....	20
<% JavaScriptlet %> .....	24
5. JavaScript Annotations .....	25
6. Built-in JavaScript libraries .....	26
gws.GwAction .....	26
gws.Assert .....	26
7. System defined variables and functions .....	27
8. Create your own GwAction library .....	28
GwAction.....	28
Custom GwAction.....	29
9. Updates by Ajax.....	30
10. Command Usages.....	35
gws .....	35
TestRunner.....	35
TestGenerator .....	36
TestNG.....	37
11. Troubleshooting .....	38
Bind error (socket) on Windows.....	38
Kill driver process .....	39
Out of disk space .....	40
12. Combinatorial Testing .....	41

## 1. Introduction

Selenium is a tool widely used to code tests in test automation. It is very efficient to make sure all functions work as we expected. In order to make our tests reusable and maintainable, for example, applying Page Object Model (POM), some amount of programming is required. With this approach, you need to create page classes and define properties that represent elements to be displayed on target web page. This concept works fine while your web application works stable. In real word, we need to continuously change the code to expand features and fix issues. They make your tests break and you also need to fix them to pass. With this reason, developers are spending a lot of time to manage tests as well as application code.

To make test automation available with less load, it is important that test tool is highly flexible and describable. GWS is a test tool that reads coma separated values (CSV format) and sends the values to a browser through Selenium APIs. However, it is not an ordinary CSV that accepts a series of text values. You can describe JavaScript function, array and object containing key-value pairs as well as String value. With the help of JavaScript, GWS makes you easy to describe complicated test scenarios.

GWS also supports combinatorial testing with using 2 and 3-wise algorithm to reduce the number of combinations. A test script created for a single test scenario can be easily extended to the script for combinatorial testing by adding possible values to each element.

For more details, please see chapter [Combinatorial Testing](#).

## 2. Installation

### Dependencies

GWS requires the following software:

- Node.js  
<https://nodejs.org/>
- Selenium Browser Drivers: We tested the following drivers only:
  - Google Chrome Driver for Chrome (**RECOMMENDED**)
  - Mozilla Gecko Driver for FireFox
  - Microsoft Internet Explorer Driver for IE (32-bit version)

You will find the links to download drivers in the following site:

<https://www.npmjs.com/package/selenium-webdriver>

The drivers are standalone executables and the directory where you downloaded should be placed on your system **PATH**. For example, if you store the driver files in the 'drivers' folder, you need to add the folder in PATH variable as shown below:

```
SET PATH=%PATH%;drivers
```

## Install GWS

Once you installed Node.js on your machine, run the command below from a command line:

```
C:> npm install gws
```

Now it's ready to run gws. You can specify Java options as well as GWS options as shown below:

```
C:> gws run test.ts -Dfile.encoding=UTF-8 -b chrome
```

## Check Versions of Jar dependencies

gws.jar and all dependencies of jars are installed under `node_modules/gws/jdeploy-bundle` folder. You can check to see which versions are bundled for dependencies of jars in the folder.

```
gws.jar
gwsdeploy.jar
jcommander-1.7.jar
jdeploy.js
selenium-server-standalone-3.8.1.jar
testng-6.13.6.jar
```

## Use different version

You may want to use different version of Selenium, for example, selenium 3.7.1 rather than 3.8.1, depending on browser version installed on your machine. To do so, download the specific version and replace with the jar. Here is the step:

```
C:> npm install selenium-server-standalone-jar@3.7.1
C:> cd node_modules
C:> copy /Y selenium-server-standalone-jar\jar\selenium-server-standalone-3.7.1.jar
gws\jdeploy-bundle\selenium-server-standalone-3.8.1.jar
```

**Note that you must keep the original file name even if you overwrite with different version from the original number.**

## Add Your Own Jar

GWS does read Class-Path in MANIFEST.MF rather than CLASSPATH environment variable, hence your own jar file is not recognized even if you add it in your CLASSPATH. You need to update MANIFEST.MF stored in gwsdeploy.jar.

Here is the step to add your jar in the Class-Path:

```
C:> cd node_modules/gws/jdeploy-bundle  
C:> jar xfm gwsdeploy.jar META-INF/MANIFEST.MF
```

MANIFEST.MF file will be generated in META-INF folder. Open the file with your editor, and you will see the contents below:

```
Manifest-Version: 1.0  
Class-Path: gws.jar selenium-server-standalone-3.8.1.jar jcommander-1.7.jar testng-  
6.13.6.jar  
Created-By: 1.7.0_79 (Oracle Corporation)  
Main-Class: gws.Gws
```

Add a space and your jar file name in the Class-Path: line.

```
Manifest-Version: 1.0  
Class-Path: gws.jar selenium-server-standalone-3.8.1.jar jcommander-1.7.jar testng-  
6.13.6.jar yourOwn.jar  
Created-By: 1.7.0_79 (Oracle Corporation)  
Main-Class: gws.Gws
```

Run the command below:

```
C:> jar cfm gwsdeploy.jar META-INF/MANIFEST.MF
```

And put your jar file in the `node_modules/gws/jdeploy-bundle` folder where is the same folder that the gwsdeploy.jar exists.

### 3. Getting Started

In order to make sure that your environment is ready to use GWS, run some sample applications.

#### Hello World

Let's start with a minimal code that displays a Hello message on a console. Open your favorite text editor and type the following code, and then save as hello.ts.

**Note that extension of test script file must be '.ts'.**

hello.ts

```
@import gws.GwAction,
function() {
    print("Hello, World!");
},
```

Run the script from a command line. Below is an example to run it on Windows. You will see the hello message on the command screen.

```
C:\samples\hello> SET PATH=%PATH%;C:/Windows/system32/drivers/
C:\samples\hello> gws run hello.ts
Starting ChromeDriver 2.xx (abcdefg..) on port 12345
Only local connections are allowed.
org.openqa.selenium.remote.ProtocolHandshake createSession
INFO: Detected dialect: OSS
Hello, World!
Total time: 0.1 sec
C:>
```

As you can see, there are some settings required to run the test.

Add the following paths in PATH environment variable:

- Windows/system32 directory to close browser. (Only for Windows)
- drivers directory where Selenium browser drivers (e.g. chromedriver.exe) are stored.

## Wizard HTMLs

Next, we will see a little more realistic example that accesses a web site and populates values on the pages. You will find the HTML and script files in samples/wizard directory. The site consists of three HTML pages. The first two pages show a form and next button and the last page shows values you entered.

**Page 1**

Name :

Date of Birth :  /  /

**Page 2**

Make :

Type : ☐ Auto ☒ Truck

Agreement : ☒

**Page 3**

**Account**  
Name : John Smith  
Date of Birth : 1911/02/03

**Vehicle**  
Make : Toyota  
Type : Truck

**Others**  
Agreement : true

To fill the values and click Next buttons automatically, create a test script:

test.ts

```
@import gws.GwAction,
@import gws.Assert,
<%
function next() {
    gw.println('next');
    const elements = gw.filterActiveElements(driver.findElements(By.xpath('//button')));
    if (!elements.isEmpty()) gw.clickElement(elements.get(0));
}
%>

function() {
    driver.get(new java.io.File('Page1.html').toURI());
},

// Page 1
waitForPage('Page 1'),
'John Smith',
1911, 2, 3,
next,

// Page 2
waitForPage('Page 2'),
'Toyota', // make
false, // type-auto
true, // type-truck
next,

// Page 3
waitForPage('Page 3'),
function () {
    assertEquals(text('Name').getText(), 'John Smith', 'Name on Page 3');
    assertEquals(text('Make').getText(), 'Toyota', 'Make on Page 3');
    assertEquals(text('Type').getText(), 'Truck', 'Type on Page 3');
    assertTrue(text('Agreement').getText(), 'Agreement on Page 3');
},
```

Open a command window and run the test.js.

```
C:\samples\wizard> gws run test.ts
```

The run command reads values comma separated from the top.

The first two lines are @import values to import and evaluate external JavaScript files. The first 'gws.GwAction' library defines GwAction object that implements GwAction APIs. We will explain about the APIs in later chapter. The second 'gws.Assert' library defines TestNG assertion APIs such as assertEquals, assertTrue, etc.



At the line 11, a JavaScript function is defined. You here need to describe a code to open target web page where you want to start your testing.

After that, you need to call `waitForPage()` function with a String that is shown on the page. It waits until the String appears on the page, which is ready to start testing on the page. It also accepts By object, which is a locator to specify the location of the element, in its parameters.

#### Format #1:

```
waitForPage ( pageId [, timeoutInSeconds ] );
```

Example 1: Wait until a String 'Page Title' appears up to 60 seconds.

```
waitForPage ('Page Title', 60);
```

#### Format #2:

```
waitForPage ( by [, timeoutInSeconds ] );
```

Example 2: Wait until a H2 tag, which body text is 'Page Title', appears (up to 20 seconds).

```
waitForPage (By.xpath("//h2[text()='Page Title']"));
```

And then, describe a series of values to fill in elements. Once you filled all values on the first page, call `next` function that clicks on the Next button. The action causes opening a new page, you need to call `waitForPage()` function again with a String to be displayed on the second page. On the third page, there is no element to fill. Instead, it shows all values you entered so that you can validate with the shown values to see if the web site properly processed your values. To do so, describe an anonymous JavaScript function and call assertion APIs to validate.

On the console, page title and filled-in values are displayed.

```
Page-1:[Page 1]
1:[John Smith]
2:[1911]
3:[2]
4:[3]
next
Page-2:[Page 2]<<[Page 1]
1:[Toyota]
2:[false]
3:[true]
4:[true]
next
Page-3:[Page 3]<<[Page 2]
Total time: 3.1 sec
```

If Auto was unexpectedly selected, you will see the following assertion error:

```
Page-3:[Page 3]<<[Page 2]
java.lang.AssertionError: Type on Page 3 expected [Truck] but found [Auto]
```

### Key-value pair (Object literal)

The wizard example can be also described with key-value pairs instead of a series of values. Key-value allows you to find an element by using a label shown near the target element. See the Page1 again:

**Page 1**  
Name :   
Date of Birth :  /  /

The name input element is displayed with a label saying 'Name', so you can describe the value with the label in JavaScript object literal notation syntax:

```
{Name: 'John Smith'},
```

Likewise, the Date of Birth can be specified as show below:

```
{'Date of Birth': [1911, 2, 3]},
```

Key-value accepts an array value as well as a String value. In case of an array, each element value in the array is filled in from the closed web element.

If you want to specify n-th number from the label to identify the target web element, add the index number in parentheses.

```
{  
  'Date of Birth(1)': 1911,  
  'Date of Birth(2)': 2,  
  'Date of Birth(3)': 3,  
},
```

If exact same label string is displayed more than once on the page, you can add the index number as the second parameter in the parentheses. The example below populates the date values in web elements following the third 'Date of Birth' label. (0-origin)

```
{'Date of Birth(1, 2)': [1911, 2, 3]},
```

Also, if the text is a partial text, specify true at the third parameter.

```
{'Date of Birth'(1, 2, true)': [1911, 2, 3]},
```

Since GWS waits until the key String is shown on the page, you do not have to call `waitForPage()` function at the beginning of each page. With using this notation, the test.js for wizard example can be rewritten as shown below:

test.ts

```
@import gws.GwAction,
@import gws.Assert,
<%
function next() {
  gw.println('next');
  const elements = gw.filterActiveElements(driver.findElements(By.xpath('//button')));
  if (!elements.isEmpty()) gw.clickElement(elements.get(0));
}
%>

function() {
  driver.get(new java.io.File('Page1.html').toURI());
},

// Page 1
{
  Name: 'John Smith',
  'Date of Birth': [1911, 2, 3],
},
next,

// Page 2
{
  Make: 'Toyota',
  Type: [false, true],
}
next,

// Page 3
function () {
  assertEquals(text('Name').getText(), 'John Smith', 'Name on Page 3');
  assertEquals(text('Make').getText(), 'Toyota', 'Make on Page 3');
  assertEquals(text('Type').getText(), 'Truck', 'Type on Page 3');
  assertTrue(text('Agreement').getText(), 'Agreement on Page 3');
},
```

## Combinatorial Testing

All test scripts we saw in previous section represent a single test scenario per script. GWS allows you to easily extend the script to handle combinatorial testing that is a way to test software with more cases at lower cost.

There is a sample test script for combinatorial testing, comb.tsc. You will find the script in samples/wizard directory.

**Note that the extension of combinatorial test script file must be '.tsc'.**

comb.tsc

```
@import gws.GwAction,
@import gws.Assert,
<%
function next() {
  print('next');
  const elements = gw.filterActiveElements(driver.findElements(By.xpath('//button')));
  if (!elements.isEmpty()) gw.clickElement(elements.get(0));
}
%>

function() {
  driver.get(new java.io.File('Page1.html').toURI());
},

// Page 1
{
  Name: 'John Smith'; null,
  'Date of Birth': [1911, 2, 3]; [2001, 12, 31],
},
next,

// Page 2
[ {
  Make: 'Toyota',
  Type: [false, true],
}; {
  Make: 'Honda',
  Type: [true, false],
}],
next,

// Page 3
function () {
  assertEquals(text('Name').getText(), 'John Smith', 'Name on Page 3');
  assertEquals(text('Make').getText(), 'Toyota', 'Make on Page 3');
  assertEquals(text('Type').getText(), 'Truck', 'Type on Page 3');
  assertTrue(text('Agreement').getText(), 'Agreement on Page 3');
},
```

The above script is almost same as previous test.ts. The difference is that two possible values are specified separated by semi-colon in Name and Date of Birth key-value pairs. If you want to specify multiple possible object literals, enclose braces, `[{ obj1 }];[{ obj2 }]`, as you can see the sample code for Page 2 in the above code.

Before execution, you need to convert it into regular test scripts.

```
C:\samples\wizard> gws generate comb.tsc
```

The generate command makes combinations of the possible values by using 2-wise (pairwise) algorithm and generates test scripts that are executable by run command. In this example, four test scripts will be generated.

1. comb-1.ts → John Smith & 1911, 2, 3
2. comb-2.ts → John Smith & 2001, 12, 31
3. comb-3.ts → null & 1911, 2, 3
4. comb-4.ts → null & 2001, 12, 31

Remaining values are same in all the scripts.

The generate command accepts `-t` option to specify t number of t-wise. Below are the numbers available to specify:

- 1 : 1-wise
- 2 : pairwise (default)
- 3 : 3-wise
- 0 : All combinations

It also accepts `-d` option to change working directory where test scripts are generated.

```
C:> gws generate comb.tsc -d work
```

With the `-d` option, the four JavaScript files (comb-1.ts to comb-4.ts) are generated in “work” folder. And comb.xml is also generated in it. If you want to execute all the generated test scripts, run the xml by using run command.

```
C:> gws run work/comb.xml
```

## Run from TestNG

The generate command generates a TestNG XML file as well as test scripts. TestNG allows you to run scripts with multiple threads and generate report of the test results.

testng.xml

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="suite name" parallel="classes" thread-count="5">
  <parameter name="browser" value="chrome"/>
  <parameter name="common" value="-V"/>
  <test name="test name">
    <classes>
      <class name="comb-1.ts"/>
      <class name="comb-2.ts"/>
      <class name="comb-3.ts"/>
      <class name="comb-4.ts"/>
    </classes>
  </test>
</suite>
```

Basically it follows the specifications of TestNG.

<http://testng.org/doc/documentation-main.html#testng-xml>

However, there are some features extended by gws:

- Use classes tag to specify test script. (Do not use package and methods tags)
- Define browser parameter if you want to change from default 'chrome'.
- Define common parameter to apply options or common JavaScript file paths to all classes in the test.

Here is an example to explain difference of effective parameters specified by common parameter and class name.

```
<parameter name="browser" value="chrome"/>
<parameter name="common" value="common.js -V"/>
<test name="test name">
  <classes>
    <class name="comb-1.ts"/>
    <class name="comb-2.ts"/>
    <class name="comb-3.ts -b firefox"/>
    <class name="comb-4.ts"/>
  </classes>
</test>
```

Comb-1, 2, and 4 are executed with 'common.js -V -b chrome' parameters, but comb-3 has additional parameters, '-b firefox'. As a result, only comb-3 is opens FireFox browser and execute the test on it while other tests are running on Chrome.

To execute the XML, type below from a command line:

```
C:\samples\wizard> gws run testng.xml
```

Once it's finished, test-output directory is created. If you open index.html in it, you will see the test result in a report generated.

Test results	
1 suite	
All suites	Methods in chronological order
suite1	gwttest.test.Test1
	test(test.js, chrome) 0 ms
Info	test(comb-1.js, chrome) 0 ms
	gwttest.test.Test2
Results	test(comb-2.js -b firefox, chrome) 8417 ms

If you want to execute testng.xml GWS generated, create another xml and specify the generated xml path in suite-file tag. In this case, all parameters defined in parent xml are effective in the child xml unless parameters with the same name are defined in the child.

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Parent Suite" parallel="classes" thread-count="5" preserve-order="true">
  <parameter name="browser" value="chrome"/>
  <parameter name="common" value="common.js -V"/>
  <suite-files>
    <suite-file path="comb/testng.xml"/></suite-file>
  </suite-files>
</suite>
```

## 4. Value types in Test Script

As you can see in previous chapter, gws accepts comma separated values. This chapter explains about value types available to describe in your test script. Below are valid value types in test script:

- @import, @include,
- String,
- Number,
- Boolean,
- JavaScript function,
- JavaScript array, nested array
- JavaScript object
- <% JavaScriptlet %>

Each value (except JavaScriptlet) must be separated by a comma (CSV format) as shown below:

```
1: @import gws.Assert,  
2: @import './lib/qnb7.js',  
3: <% var By = org.openqa.selenium.By; %>  
4: function() {  
5:   driver.get(new java.io.File('Page1.html').toURI());  
6: },  
7:  
8: // Page 1  
9:   'John Smith',  
10:   1911, 2, 3,  
11:   next,  
12:  
13: // Page 2  
14:   'Toyota', // make  
15:   false, // type-auto  
16:   true, // type-truck  
17:   true, // Agreement  
18:   next,
```

### @import, @include

Line 1: imports an external JavaScript file (Assert.js) from CLASSPATH so that you can call the functions defined in the library in your script.

Line 2: imports an external JavaScript file (qnb7.js) by using relative path from the current directory.



Note that @import accepts two types of formats. If the JavaScript file is in CLASSPATH use package format. If you want to specify absolute path or relative path, use file path format. @include is a similar annotation to the @import. @import evaluates the script read from external JavaScript file, whereas @include simply includes the file contents.

### String, Number

Line 9, 10, 14: String values and numbers are sent to current web element. Current element is determined how many times you sent values after new page started. gws has an elementIndex internally. When a new page is opened, the elementIndex is initialized to 0 which represents the first target element on the page. In this example, 'John Smith' is the first value in the script, so it is filled in the first textbox on the Page 1. And then the elementIndex is automatically incremented and current element is now pointing to year textbox of Date of Birth.

**Page 1**

Name :

Date of Birth :  /  /

Annotations:

- elementIndex = 0 (points to the Name field)
- elementIndex = 1 (points to the first year field of the Date of Birth)

### Boolean

Line 16: If you specify true, it will make a click on the target element.  
Line 15: If it is false, nothing happens.

**Page 2**

Make :

Type : ☐ Auto ☒ Truck

Agreement : ☒

Annotations:

- Click on the current element if **true**. (points to the Truck radio button)
- Do not click if **false**. (points to the Auto radio button)

## JavaScript function

Line 4 is an anonymous JavaScript function. GWS calls the function with a GwObject instance so that you can access the GwObject through 'this' keyword from the function.

Line 11 and 16 call a JavaScript function named 'next' with no parameters. If you want to invoke with some parameters, make an anonymous function and call the 'next' with parameters.

```
function() {  
    next(2);  
},
```

### IMPORTANT:

If you return a value from the function, the return value is sent to the current web element, and `elementIndex` is counted up to point the next element.

```
function() {  
    return 'John Smith';  
},
```

If the above function is invoked, 'John Smith' is filled in current element (e.g. Name field). After that, current element is moved to the next (e.g. Date of Birth field).

However, if it returns undefined (which means that the function does not return a value), a value is not sent to web element, and `elementIndex` is not updated, either.

```
function() {  
    assertEquals(name, 'John Smith');  
},
```

For example, the above function only evaluates an assertion and there is no return sentence. So, it does not change a value of the current web element.

## JavaScript array

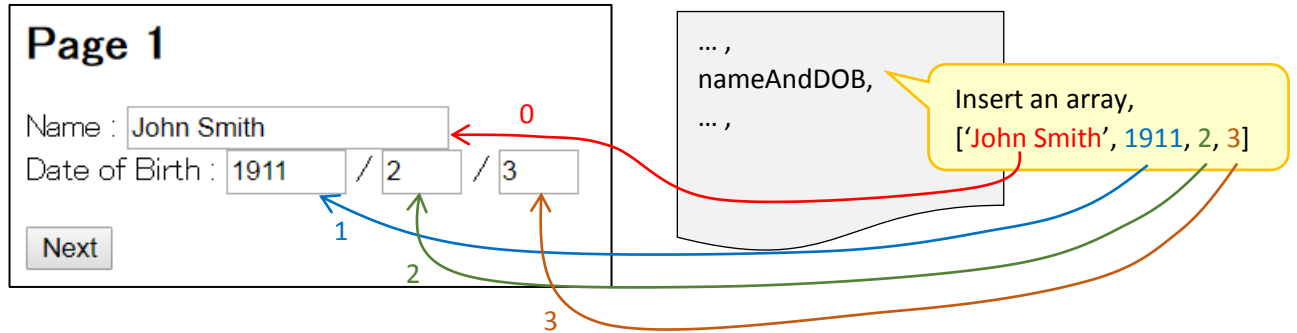
An array groups a series of values. gws executes each value from the left. It is nothing special to test script, but it helps you work on the following situations:

### Return value from a function:

An array enables you to return multiple values from a function.

```
Function nameAndDOB() {  
    return ['John Smith', 1911, 2, 3];  
},
```

The above function populates name and date of birth when it's invoked.



### Group multiple values for Combinatorial Testing:

You can group multiple values as one possible value in combinatorial testing.

When GWS generates combinations, the three values (year, month, day), shown in the example below, represented a date are treated as one value.

Test.js

```
'Date of Birth': [1911, 2, 3]; [2001, 12, 31],
```

test-1.js

```
'Date of Birth': [1911, 2, 3],
```

test-2.js

```
'Date of Birth': [2001, 12, 31],
```

### Multiple selections (nested array):

This is different usage from other arrays. If the target element is a select box with a multiple attribute, you can pass multiple values to be selected by using a nested array, [[ ]].

Note that the parent array of the nested array must be one value that is the child array, e.g. [[ 'value' ]], [[ 1, 2, 3 ]].

```
'Toyota', // make
[['Tokyo','Other']], // area (multiple selection)
false, // type-auto
true, // type-truck,
```

If you run the above code on the page below, the child array, ['Tokyo','Other'], is passed to the second element that is an Area select box, and Tokyo and Other options are chosen:

**Page 2**

Make :

Area :

Type : ☐ Auto ☒ Truck

This rule is also applied on object literal. You can revise the previous code using object literal as shown below:

```
{
  Make: 'Toyota',
  Area: ['Tokyo', 'Other'],
  'Type' : [false, true],
},
```

## JavaScript object

An object literal notation allows you to identify a web element by using a label string.

```
{Name: 'John Smith'},
```

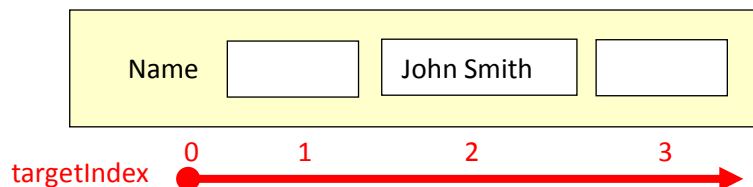
gws fills the value 'John Smith' in an input element next to the label 'Name'.

In order to specify the position of the input element or label, the key accepts three parameters.

```
{'Name( targetIndex, labelIndex, isContains )': 'John Smith'},
```

If the input element locates at the second position from the label, you can specify (2):

```
{'Name(2)': 'John Smith'},
```



If there are multiple Name labels on the same page, specify the position with 0-origin number:

```
{'Name(1, 2)': 'John Smith'},
```

labelIndex	
0	Name <input type="text"/>
1	Name <input type="text"/>
2	Name <input type="text" value="John Smith"/>

If the given text is a partial text, specify true.

```
{'Name(1, 0, true)': 'John Smith'},
```

Your Name	<input type="text" value="John Smith"/>
-----------	---

If the label index is 0, you can omit the value and specify the Boolean value at the second parameter.

```
{'Name(1, true)': 'John Smith'},
```

Likewise, if the target index is 1, you can omit the value and specify the Boolean value at the first parameter.

```
{'Name(true)': 'John Smith'},
```

With this expression, the value is filled in the first input element from the third Name label. Also, JavaScript object accepts multiple key-value pairs in a object literal.

```
{  
  Name: 'John Smith',  
  'Date of Birth': [1911, 2, 3],  
},
```

### Prefix in label

The label also accepts a prefix to use other locating mechanism, e.g. xpath.

```
{'prefix:Name( targetIndex )': 'John Smith'},
```

Note that **labelIndex** and **isContains** parameters are only available for 'label'.

Below are available prefixes:

Prefix	Description
label	Finds element based on the label associated with label and labelFollowing functions defined in GwAction. <b>(default)</b>
className	Finds element based on the value of the "class" attribute.
cssSelector	Finds element via the driver's underlying W3 Selector engine.
id	Finds element based on the value of the "id" attribute.
linkText	Finds element based on the body text of the "a" tag.
name	Finds element based on the value of the "name" attribute.
partialLinkText	Finds element based on the partial body text of the "a" tag.
tagName	Finds element based on the tag name.
xpath	Finds element based on the given xpath.

```
<form>  
  <span id="agreement">Agreement</span>  
  <input type="checkbox">  
</form>
```

Example 1: Find the checkbox element by the label text:

```
{'Agreement': true},
```

Example 2: Find the element by id:

```
{'id:agreement': true},
```

Example 3: Find the element by xpath:

```
{'xpath://*[@id="agreement"]': true},
```

### **Switch expression in label**

Multiple labels can be described in a key to switch processes based on order of label appearance on the page. GWS watches all the labels described in the key and executes only the value for the first label appeared.

```
{'label1|label2|label3|...|labelN': [value1, value2, value3, ..., valueN],}
```

Labels are separated by '| '.

If label1 is the first label appeared on the page (label2 and 3 do not exist yet at the time), value1 is executed. Likewise, if label2 is the first one, execute value2.

Value2 and after the value2 are optional. If there is no value corresponding to the label, skip the process for the label.

**Example 1:** After searching users, there are two possible results. One is that there is no such user and system displays a 'Create New' button to create a new user. The other is that system finds the user and shows the user profile with a label saying 'User Profile'. We don't know which happens when writing the script. In this case, we can describe both cases using switch expression.

In the key section, there are two labels separated by '| '. And two values are given in an array in the value section. If there is no such user and 'Create New' label appears, gw.clickElement is invoked. If system found the user and displayed the user profile with the label 'User Profile', the anonymous function to validate the last name is executed.

```
{'Create New(0)|User Profile(0)': [  
  gw.clickElement(this.element), // Click the Create New button  
  function() {  
    assertEquals(text('Name').getText(), 'Smith', 'Last Name');  
  },  
],}
```

**Example 2:** If user needs to pay, system displays a 'Pay' button. If not, a 'Continue' button is displayed. Only when the 'Pay' button appears, user needs to enter its account info.

The code below waits until 'Pay' or 'Continue' label appears, and execute the function defined in its value if 'Pay' button is displayed first. If 'Continue', do nothing.

```
{'Pay(0)|Continue(0)': function() {  
  gw.clickElement(this.element); // click the appeared element  
  return {  
    'Account Number': '12345',  
    'Bank Code': '100',  
    'Bank Name': 'ABC Bank',  
  };  
},}
```

### <% JavaScriptlet %>

A JavaScriptlet is a piece of JavaScript code that can be described in your test script. It must be enclosed with <% and %> tags. Inside the tags, you can add valid JavaScript code.

Example 1: With the declaration of a By variable in the JavaScriptlet, you can access `org.openqa.selenium.By` class in your function without specifying full package name.

```
<%  
  var By = org.openqa.selenium.By;  
%>  
function () {  
  assertEquals(driver.findElement(By.id('name')).getAttribute('value'),  
    'John Smith',  
    'Failed to fill in name element.');
```

Example 2: Declare a costNew variable to save the value entered.

```
<%  
  var costNew = -1;  
%>  
{'Cost New', '$12345'},  
function () {  
  costNew = element('Cost New').getAttribute('value');
```



## 5. JavaScript Annotations

In JavaScript files including test scripts, you can use the following annotations:

Name	Description
@import	Imports and evaluates a JavaScript file.
@include	Includes the file contents.

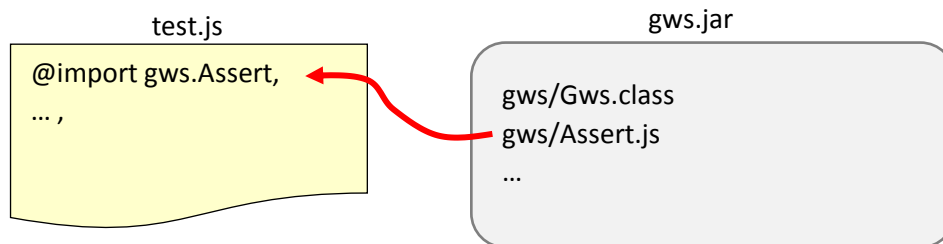
@import and @include accept the following two types of path format:

Format	Example
Package format	@import org.sample.Test Searches org/sample/Test.js file in your <b>CLASSPATH</b> .
File path format	@import 'test.js' Searches test.js file in your <b>current directory</b> .

Example 1:

Import **Assert.js** file in gws directory.

```
@import gws.Assert,
```



Example 2:

Import **qnb7.js** file in lib directory where exists in your current directory.

```
@import './lib/qnb7.js',
```

## 6. Built-in JavaScript libraries

gws provides the following JavaScript libraries. By importing the libraries with `@import` annotation, you can use the functions in your test script.

```
@import gws.Assert,
```

### `gws.GwAction`

Import this library in test script to test web application. This provides the following functions:

Name	Description
<code>element ( label [ , targetIndex  , labelIndex  , isContains ] )</code>	<p>Finds a WebElement by using a label string. This function only finds input type elements such as input, select, textarea, a tags.</p> <p><b>label:</b> Label string. <b>targetIndex:</b> index number of the target element, starting from the label element. 0 represents the label element itself. (Optional, default: 1) <b>labelIndex:</b> If there are multiple elements with the same label, specify the index number starting from 0. (Optional, default: 0) <b>isContains:</b> Allow partial matching with the given text. (Optional, default : false)</p>
<code>text ( label [ , targetIndex  , labelIndex  , isContains ] )</code>	<p>Finds a WebElement by using a label string. This function only finds text elements such as span, div, p tags.</p> <p><b>label:</b> Label string. <b>targetIndex:</b> index number of the target element, starting from the label element. 0 represents the label element itself. (Optional, default: 1) <b>labelIndex:</b> If there are multiple elements with the same label, specify the index number starting from 0. (Optional, default: 0) <b>isContains:</b> Allow partial matching with the given text. (Optional, default : false)</p>

### `gws.Assert`

Import this library in test script to assert values. This provides functions defined in TestNG Assert class. For the API specifications, see <http://testng.org/javadocs/org/testng/Assert.html>

## 7. System defined variables and functions

gws declares some variables and functions for ease of development of your test scripts.

Variables:

Name	Description
driver	Returns a WebDriver object.
gw	Returns a GwObject implemented useful properties and functions below:  <b>pageId</b> : Returns the current pageId <b>element</b> : Returns the current WebElement <b>clickElement</b> (element): Click on the element.  <a href="#">See GwObject reference section for all the properties and functions.</a>

Examples:

Call a get function of the current WebDriver to open a specific web page.

```
() => driver.get(new java.io.File('Page1.html').toURI()),
```

Functions in test script can access GwObject instance through 'this' keyword.

```
() => {  
  if (this.pageId == 'Page 1') {  
    this.element.sendKeys('test value');  
  }  
},
```

Regular functions in additional JavaScript files access GwObject through 'gw' variable.

```
var myFunc = (element) => gw.clickElement(element);
```

Functions:

Name	Description
close()	Stops running the test and close the browser window.
error()	Returns an error message captured by an error function defined in Action object.
exit()	Stops running the test but keep the browser open.
resetPage()	Resets system variables related to the current page. Call if you need to populate values from the first element on a same page.

## 8. Create your own GwAction library

### GwAction

GwAction is a default library that implements GwAction APIs and some other useful functions for sample HTML application. Since it is just a sample implementation, it won't perfectly fit to your web application and you need to create your own library according to the specifications of your application.

Having said that, most of implementations in the GwAction library could be commonly used, so it is good to start with reading the GwAction.js to see what we need to implement.

You will find the GwAction.js file in gws.jar.

GwAction is a mandatory object to execute your tests with gws run command. Here are the functions defined in GwAction:

Name	Description
driver( browser )	Returns a WebDriver object for a browser given.
scanTestElements ( driver, pageId )	Returns a list of WebElement objects that you want to fill a value.
label ( label, labelIndex, isContains )	Returns a XPath string to find an element identified by the given label and index.
labelFollowing ( targetIndex )	Returns a XPath string to find an element following the label.
labelFollowingText ( targetIndex )	Returns a XPath string to find a text element following the label.

It also provides you the following functions:

Name	Description
element(label, targetIndex, labelIndex, isContains)	Returns an element identified by the given parameters.
text(label, targetIndex, labelIndex, isContains)	Returns a text element identified by the given parameters.
submit(xpath, name, param)	Submits a form and move to the next page.

## Custom GwAction

Here shows a custom GwAction that overrode some APIs of original GwAction.

customGwAction.js

```
/*
 * Custom GwAction
 */
@import gws.GwAction;

var By = org.openqa.selenium.By;

GwAction.scanTestElements = function (driver, pageId) {
    const elements = gw.filterActiveElements(driver.findElements(By.xpath("//div[contains(@class, 'wizard-form')]//*[self::input or self::textbox or self::select]")));
    return elements;
};
```

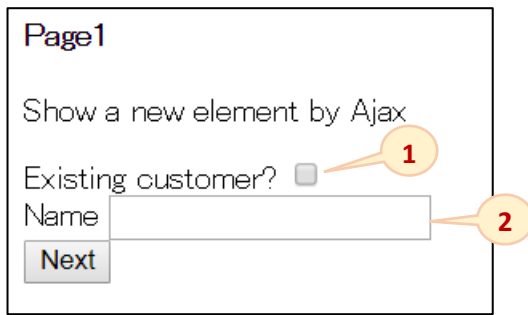
Import the JavaScript from your JavaScript or test script.

```
@import 'customGwAction.js';
```

## 9. Updates by Ajax

By using Ajax, you can dynamically update web contents. Ajax enables you to show and hide elements from the page, without submitting the form. This behavior may make you confuse with getting unexpected results while creating test scripts, especially if you use a series of values. When you click on a link or button to invoke Ajax code, it looks updating contents with no time, but it actually takes time until browser completes the update. Hence, if you make an Ajax action in your script, you must add some code to explicitly wait for the completion of Ajax update.

Here is an example of HTML page using Ajax. The sample code can be found in samples/ajax folder. When you open the first page, Page1.html, there are two elements on it as shown below:



Page1

Show a new element by Ajax

Existing customer? ☐ 1

Name  2

Next

If you are not an existing customer, you can simply describe the values in the order of the shown elements:

```
@import gws.GwAction,  
( ) => driver.get(new java.io.File('Page1.html').toURI()),  
false, 1  
'John Smith',  
next, 2
```

However, if you are an existing customer and click on the checkbox, the HTML dynamically show a new input element for User ID. In this case, index numbers are numbered on the shown elements:

Page1

Show a new element by Ajax

User ID

Existing customer? ☒

Name

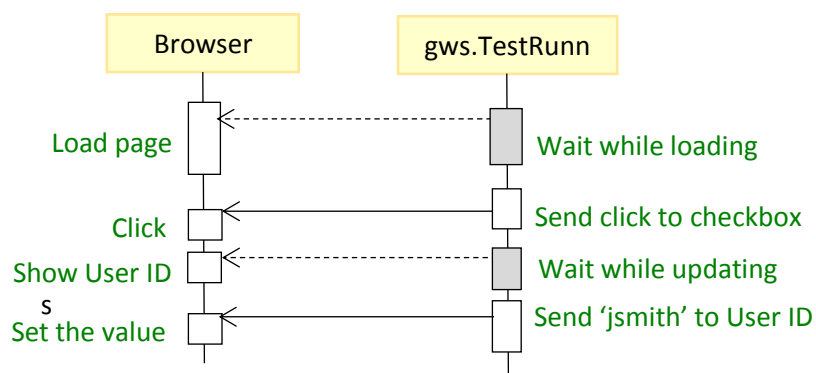
Next

The index number for checkbox, which has been already clicked, stays the same. And index numbers are numbered on the rest of elements, starting at 2 from the top. Thus, test script for existing customer looks like below:

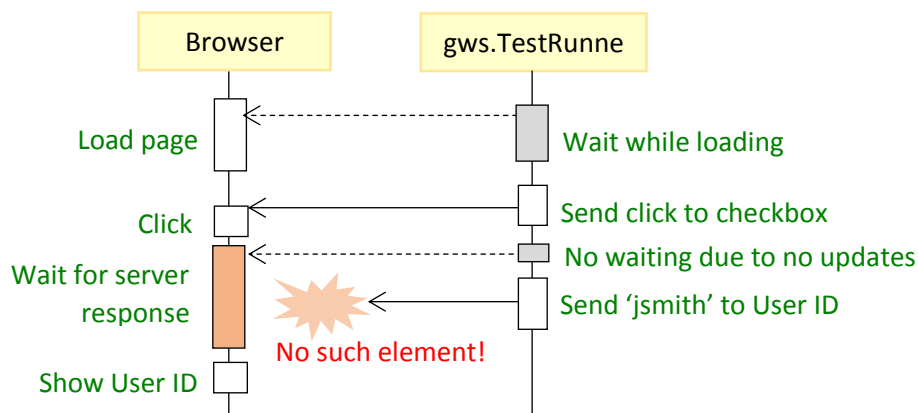
```
@import gws.GwAction,
() => driver.get(new java.io.File('Page1.html').toURI()),
true,
'jsmith',
'John Smith',
next,
```

But if you execute the script, it will fail to set the user ID 'jsmith' when web server is working very slow and takes time to response back to your browser. To simulate the situation, this HTML makes a time delay (1 second delay) to add the User ID element. If you look carefully the page, you will realize that there is a pause between clicking on the checkbox and appearing the User ID element.

GWS checks to see if the page is in updating by Ajax and wait for the completion, before entering the next value. However, in some cases, such that web server is working very slow, GWS cannot recognize the new element. The diagram below shows the case browser displays a User ID element timely, hence GWS is able to detect page updating and wait for the completion before sending the user ID value, 'jsmith'.



If the click fires a function that makes a web service call, which makes a HTTP request to the server and waits for the response, it may take longer time until User ID element appears on the browser depending on the loads on the server side. If it takes longer, the sequence will be changed to below. When GWS checked to see if the browser is in updating, browser is still waiting for a response from the server and rendering for User ID element is not started yet. Thus GWS cannot recognize that the click action makes a page update by Ajax, and sends the next value, 'jsmith', to User ID element, which does not exist yet. As a result, you will get an exception, no such element, with the test script.



In order to avoid the error, you have to explicitly wait until the target element appears. In this case, you can call `gw.waitForText()` to wait until the label text 'User ID' is shown on the page.

```

@import gws.GwAction,
() => driver.get(new java.io.File('Page1.html').toURI()),
true,
() => gw.waitForText('User ID'),
'jsmith',
'John Smith',
next,

```

The `gw.waitForText` waits until the given text appears on the page.

The `gw` object provides you the following functions for waiting the completion of Ajax updates.

- `gw.waitForNotText( text )` : Waits until a tag whose body text is exact same as the given text, disappears from the page.
- `gw.waitForPartialText( text )` : Waits until a tag whose body text contains the given text appears on the page.
- `gw.waitForNotPartialText( text )` : Waits until a tag whose body text contains the given text, disappears from the page.



- `gw.waitFor(function, message)` : Waits until the given function returns true. `waitFor` calls the given function until it returns true. If it does not return true within a certain amount of time, it throws an exception with the given message.

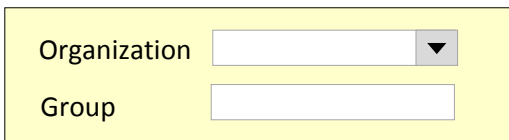
Example: Wait until a Search button appears.

```
gw.waitFor(
  () => !(driver.findElements(By.xpath("//button[text()='Search']")).isEmpty()),
  'search button'
),
```

If you use key-value pair (object literal), you don't have to add such a code to wait in this case. Object literal waits until the label element appears on the page before sending the value to the target element. Hence, you can simply describe each pair of a label text and a value to set.

```
@import gws.GwAction,
() => driver.get(new java.io.File('Page1.html').toURI()),
{
  'Existing customer?': true,
  'User ID': 'jsmith',
  Name: 'John Smith',
},
next,
```

Having said that, key-value pair is also not perfect. In some cases, you need to add code to wait for next element even if you use object literal expression. Here is an example:



Assume that Organization and Group fields are empty when the page is loaded. And when you select a value in Organization, a Group value is automatically filled by Ajax. In order to test the scenario, you may create the script below:

```
{
  'Organization': 'Dev',
  () => assertEquals(element('Group').getAttribute('value'), 'IT'),
},
```

This will work successfully on local machine, but you may get an error that says **Expected [IT] but []**, when you run the same script against remote server or at parallel execution using

multiple threads. It will happen when GWS checks the Group value before browser populates a value to the Group element. In this case, although element() function also waits for the label appearance, the Group label was initially shown at the page loaded. Thus the getAttribute of the element returns an empty String. To avoid this error, you need to check the value of the Group input box and wait until a value is set in it, before you call the assertEquals(). Here is the revised code.

```
{
  'Organization': 'Dev',
  () => {
    gw.waitUntil(() => element('Group').getAttribute('value'), 'Group input value');
    assertEquals(element('Group').getAttribute('value'), 'IT');
  },
},
```

The waitUntill executes the function in the first parameter until it returns true. Since the above code returns Group value, it is evaluated as false while the value is null or " (empty). Once Ajax populate a value in the Group input box, waitUntil gets a String and evaluate as true and then move to the next line to execute.

## 10. Command Usages

### gws

Generates test scripts for combinatorial testing and runs a test script.

```
gws [ generate | run ] parameters...
```

Gws accepts the following options at the first parameter:

Option	Description
generate	Generates test script files and TestNG XML. Internally calls TestGenerator class
run	Runs a test script (ts file) or TestNG XML file Internally calls TestRunner or org.testng.TestNG class

### Examples:

Generate test scripts from a tsc file.

```
gws generate comb.tsc -t 2 -threadCount 3
```

Run a test script with Chrome browser.

```
gws run test.ts -b chrome
```

Run test scripts by using Test NG XML.

```
gws run testng.xml
```

### TestRunner

Executes a test script.

```
java -cp gws.jar;"lib/*" gws.TestRunner [-b <browser>] [-v] [-V] [-s <level>] test.ts [func.js] ...
```

TestRunner accepts the following options:

Option	Description
-b	Browser name  <b>chrome</b> : Chrome ( <b>default</b> ) <b>firefox</b> : FireFox <b>ie</b> : Internet Explorer
-s	Screenshot level  <b>debug</b> : Take screenshots when clicked. <b>error</b> : Take a screenshot when an error occurred. ( <b>default</b> ) <b>off</b> : No screenshot taken.
-v	Generates verbose output to standard output. ( <b>default</b> )

-V	Does not generate verbose output. This option is useful for parallel execution.
----	---

#### Examples:

Execute a test script, test.js with FireFox browser. Before executing, it evaluates common.js so that you can access variables and functions defined in the common.js from your test script.

```
java -cp gws.jar;"lib/*" gws.TestRunner test.ts common.js -b firefox
```

Execute a test script with no messages displayed on the console.

```
java -cp gws.jar;"lib/*" gws.TestRunner test.ts -V
```

### TestGenerator

Generates test scripts.

```
java -cp gws.jar;"lib/*" gws.TestGenerator [-v] [-V] test.ts ...
```

TestGenerator accepts the following options:

Option	Description
-t	t-wise number (a degree of thoroughness)  <b>0</b> : All combinations <b>1</b> : Single <b>2</b> : Pairwise ( <b>default</b> ) <b>3</b> : 3-wise
-d	Working directory to generate test scripts and textng.xml. (default: <b>test-work</b> ) If the directory specified does not exist, TestGenerator generates the directory.
-threadCount	Thread count number to be set in thread-count attribute of testing.xml generated. (default: 5)
-v	Generates verbose output to standard output. ( <b>default</b> )
-V	Does not generate verbose output.

#### Examples:

Parse test1.ts and generate test scripts. Repeat for test2.ts.

```
java -cp gws.jar;"lib/*" gws.TestGenerator test1.ts test2.ts
```

Generate all combinations from test.ts with no messages displayed.

```
java -cp gws.jar;"lib/*" gws.TestGenerator test.ts -t 0 -V
```

## TestNG

Executes test suite.

```
java -cp gws.jar;"lib/*" org.testng.TestNG testng.xml
```

## 11. Troubleshooting

### Bind error (socket) on Windows

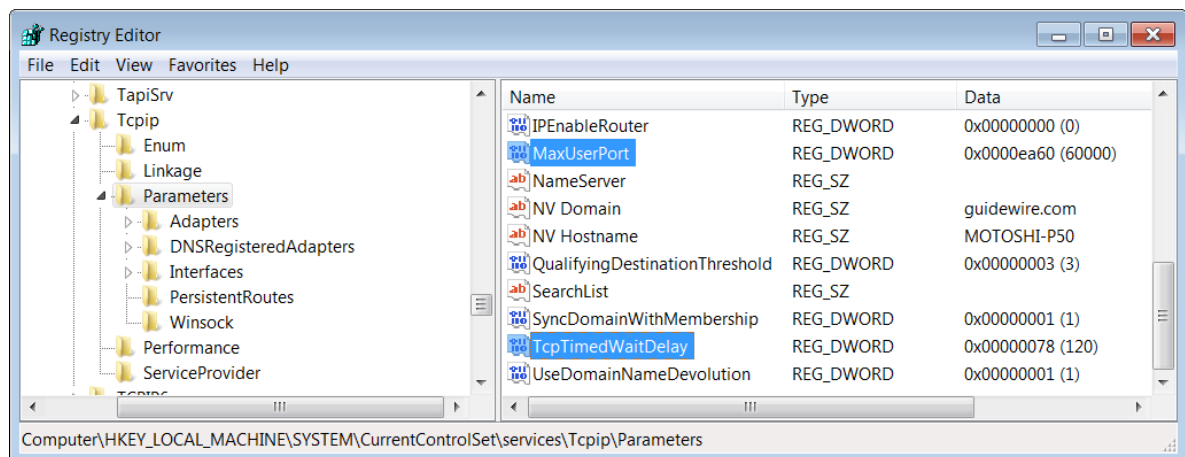
If you run tests with parallel executions on Windows machine, you need to edit the following two Windows Registry values.

Modern browsers make HTTP connections with keep-alive. Even though the communication was done, the connection is still alive. It makes the number of using connections increase and uses up all available ports and you will see socket bind errors in your console. To avoid the error, you need to increase the number of available ports and change to shorter timeout to release used connections timely.

Location:

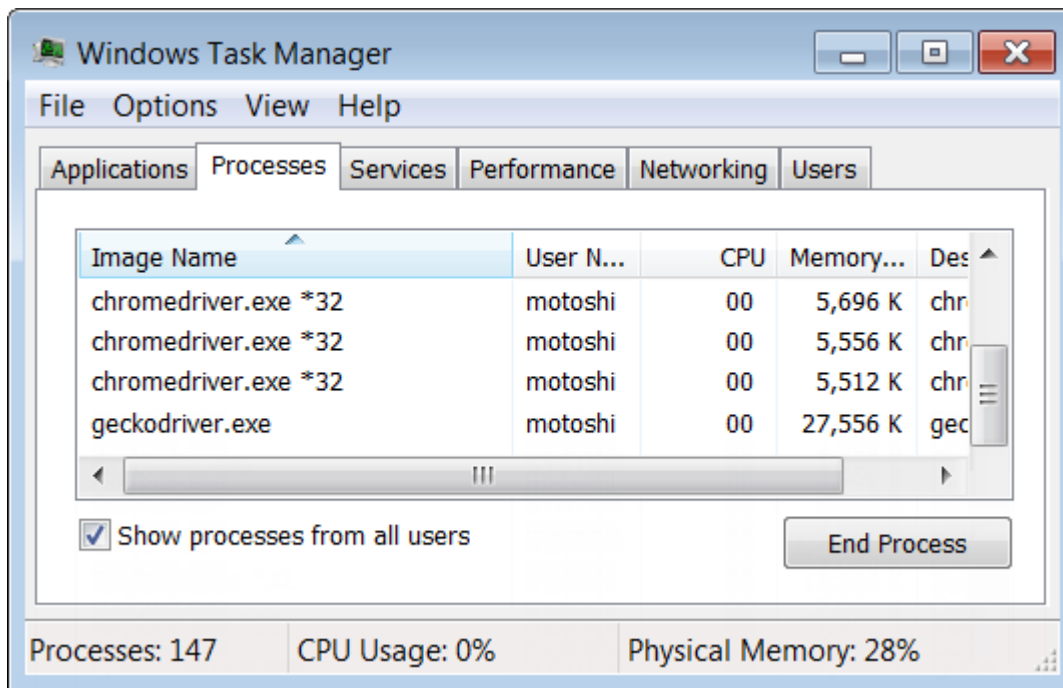
HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

Parameter Name	Data Type	Description	Recommended Value
MaxUserPort	REG_DWORD	Determines the highest port number TCP can assign when an application requests an available user port from the system.	65534
TcpTimedWaitDelay	REG_DWORD	Determines the time that must elapse before TCP can release a closed connection and reuse its resources. This interval between closure and release is known as the TIME_WAIT state or 2MSL state.	60 to 120



### Kill driver process

Even though you closed the browser by using close function, the driver process (e.g. chromedriver.exe for Chrome, geckodriver.exe for FireFox, IEDriverServer.exe for IE) still exists.



To kill the process, you need to run command below from command line:

```
C:> taskkill /im chromedriver.exe /im geckodriver.exe /im IEDriverServer.exe /f
```

## Out of disk space

Selenium drivers create temporary folders and some of folders are not removed after WebDriver finishes the process. Thus, disk space on your machine will be consumed and you will encounter out of disk space error.

Where and what folders are created depends on selenium driver version and your machine. Here is an example of chromedriver.exe on Windows 7.

Location:

C:\Users\<username>\AppData\Local\Temp

or

%AppData%\..\Local\Temp

Folders and files created by Selenium:

scoped\_dir1234\_56789  
seleniumSslSupport12345678901234567890.selenium.doesnotexist  
screenshot12345678901234567890.png

We strongly recommend you watching in your temporary directory and clean up regularly if you found such folders.



## 12. Combinatorial Testing

For those who never heard about Combinatorial Testing, this chapter explains the basic idea of the testing. Combinatorial Testing is a method of software testing to create test cases that covers all combinations for all possible parameter values. There are many terms to express Combinatorial Testing, for example, All Pairs, Pairwise, 2-wise, t-wise, etc. The letter 't' represents the number of parameters that cover all combinations, so it expresses the degree of thoroughness and bigger number generates more test cases. The 2-wise is one of instances of t-wise, in this case, 2-wise represents t-wise testing with 2 degree of toughness. It will come up a question which degree is enough to detect errors. Below shows cumulative percent of faults triggered by t-wise testing:

t	RAX convergence	RAX correctness	RAX interf	RAX engine	POSIX modules	Medical Devices	Browser	Server	NASA GSFC
1	61	72	48	39	82	66	29	42	68
2	97	82	54	47	*	97	76	70	93
3	*	*	*	*	*	99	95	89	98
4	*	*	*	*	*	100	97	96	100
5	*	*	*	*	*		99	96	
6	*	*	*	*	*		100	100	

\*= not reported

Source: IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 6, JUNE 2004

Software Fault Interactions and Implications for Software Testing

<https://pdfs.semanticscholar.org/1ad8/adab7815cf9299b752e00ea860bc28c4c090.pdf>

According to the case study, if your target application is a mission critical system that requires extremely high quality, you may need to test with t=3 to 6. But it requires more test cases to test, and it is unrealistic to apply such a big number to all tests. Hence, in general, we apply t=2 (pairwise) on general test scenarios and apply t=3 (3-wise) on some critical scenarios.

To understand how pairwise covers test patterns, let's see test cases created for all combinations and test cases created by using pairwise. Consider pairs of three parameters: A, B and C. And each parameter could have two values: 0 or 1. If we simply test for all patterns, the combinations will be below (There are totally eight cases):

Test Case #	A	B	C
1	0	0	0
2	0	0	1
3	0	1	0
4	0	1	1

5	1	0	0
6	1	0	1
7	1	1	0
8	1	1	1

On the other hand, if we create test cases using pairwise, we can reduce the number of the test cases. Here is one of examples generated by using pairwise algorithm:

Test Case #	A	B	C
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Pairwise only covers all patterns for between two parameters, that is, the test cases covers 00, 01, 10, 11 patters only for AB, AC and AC parameter pairs. Let's check each test case. First, test case #1 covers AB=00, BC=00, AC=00. Next, test case #2 covers AB=01, BC=11, AC=01. #3 covers AB=10, BC=01, AC=11. #4 covers AB=11, BC=10, AC=10. In the four cases, we see all value patterns (00, 01, 10, 11) for all parameter pairs (AB, BC, AC).

As you saw, we could reduce the number of test cases to 4 from 8 (all combinations) with pairwise. You may feel that it is not a big difference, but in real world, we could have more elements and possible values. Assume that there are six elements on a page and each element could have three values.

First Name  <Empty>, John, LooooongName

Last Name  <Empty>, Smith, LooooongName

Organization  <Empty>, OrgA, OrgB

Element 4

Element 5

Element 6

The total number of all combinations between elements and possible values are  $3^6 = 729$  patterns. But if we apply pairwise (2-wise), it can be decreased to **about 15 to 30** test cases (the number is vary depending on implementation of algorithm) which is reasonable number that we can execute.

For more information about Combinatorial Testing, see the site below:

<https://inductive.no/pairwiser/knowledge-base/introduction-to-pairwise-testing/>