Testing Web Applications with WebDriver

And Stuff
Version 0.1

Reviews History

Date	Version	Description	Author
08/06/2016	0.1	Initial draft of the document	Juan Krzemien

Table of Contents

About this document **4**
- Prerequisites for this document 4
- Companion source code 4
- General notes on automated testing 5

Picking the right test runner **6**
- TestNG 6
  - Execution diagram 9
- JUnit 11
  - Execution diagram 11
- Data Driven Tests 13
  - TestNG 14
  - JUnit 16
  - Things to avoid while doing DDT 17

Web Front End Testing with Selenium/WebDriver **18**
- What is WebDriver? 18
- WebDriver Architecture Overview 19
- Recommended way of using WebDriver 22
- Locators 24
  - Why did the locator fail? 26
- Model your pages as objects - Page Object strategy 28
  - Structure 30
  - How do I define a locator in a POM? 33
  - How do I initialize all elements defined in my POM? 35
  - Some words on element staleness 36
  - Documentation 36
- Interactions between non public types 37
- Handling synchronization 38
- Multithreading and parallelism 42
- Logging 44
- Configuration 45
  - Externalize configuration 45
  - Auto detect configuration when possible 46
  - Java Proxy Settings 47
- Handling Authentication 48
  - About usage of AutoIt to bypass browser modal dialogs 49

Bypassing Authentication 49
Basic HTTP Authentication via URL 49
Cookies Injection 50
Authentication Proxy 51
Test Scripts 52
Documentation 53
Things to avoid when testing front end 54
On prioritizing tests candidates for automation 55
Issues related to parallel test executions 56
Flaky Tests with “Random” Failures 56
Variable Test Performance and Performance Related Failures 57

Automated Web Tests in Continuous Integration Servers 58

Appium 64
Installing Appium 67
Differences with WebDriver 68
Appium in Selenium Grid 69
Appium Node configuration 69
Known Issues 70
Tips & Tricks 71
How to enable Developer Mode in Android devices 71
Using Chrome Remote Debugging to inspect web pages on devices 72
Listing and downloading APK files from devices 73
Listing activities and other useful information from APK files 73

Additional resources worth reading 75

About this document

Given the increase in Quality Assurance projects Globant is experiencing, there is an unquestionable need to standardize the way Quality Engineers develop the kind of automated solutions that clients are expecting from them as professionals.

This document will focus on automated testing with WebDriver in Java; but the root concepts detailed here can be extrapolated to different languages.

Main goal of this document is to define certain “rules” that make up, what I consider, good practices. This should enable Quality Engineers to:

- Adhere to DRY (Don’t Repeat Yourself) principle.
- Increase test script development speed as the project advances.
- Make test code more robust, readable and reusable.
- Decrease maintenance costs of test code.
- Increase the quality of test design and its code.
- Increase the Return on Investment (ROI) of tests.

Prerequisites for this document

Knowledge and experience in software testing is a mandatory requirement, but also you will benefit from having some degree of expertise in:

- Object Oriented Programming paradigm
- Java programming language

Companion source code

To ease reading and diminish the length of this document, code snippets and larger examples for some topics mentioned in this document have been published in a Git repository. You can find it here.

General notes on automated testing

- 🚫 First and foremost, **do not** take Selenium's HQ (official) documentation at heart. I mean it. Use it as an API reference to know what WebDriver can and can't do, but that's it. The examples provided there are generally either obsolete or too simple. Most of the time they are written in a procedural programming style and are far from being OOP. Either way, most of the "cool stuff" to know in their documentation has had a "*TODO*" written in it for the past 5 years.

- 🚫 **Do not just copy and paste code without fully understanding what it is doing and why**. This applies to everything you do, not just "automated testing". When "Googling" for an answer, do not take the first result that actually solves your current issue. Compare some of the solutions and figure out which one is smarter, more flexible and/or "cleaner".

- 🚫 **Do not** *just* work on "creating test scripts"; create or employ whatever tool you can get your hands on to make testing easier for the whole team:

    - 💡 Set up Continuous Integration (CI) jobs

    - 💡 Write tasks/targets for the build system (Maven, Gradle, Grunt, etc) that your team is using and instruct teammates on how to use them.

    - 💡 Propose ideas to your team like adding pre-commit hooks to your VCS to run tests before pushing new code on certain branches.

  It is pointless and very frustrating when you are the only one that knows about "testing stuff" in your team. Leaving feelings aside, it *is* your job to do that.

- ✓ Test early and often.

- ✓ Include the possibility for developers to run automated test in their boxes before pushing new changes to version control system.

- ✓ Be fluent with Java generics, lambdas, varargs, and `this` reference. For test framework development, you will probably need to consider knowing about annotations and reflection library.

Picking the right test runner

One very important thing to take into account at the beginning of an automation project initiative is the test runner itself. Pick one that supports parallel test executions.

In Java, the standard test runners are:

- TestNG, which has embedded support for parallel test runs.
- JUnit, which is awesome, specially in its latest version. But I'd advise you to stick to version 4.x until version 5 gets full IDE support.

Obviously, there are many more out there, but these two are the most well known. If you are reading this document with the intention of designing your framework, POMs and tests in a parallel way for a language **other than Java**, you will have to invest some time researching what's in the market for your platform.

It is also important to know **how** these tests runners run your tests. The behaviour between them is quite different, despite the end result for both will be "*oh good, yeah, my tests are been ran*".

TestNG

I think of it as the wizard's apprentice test runner. I mean, everybody loves (and advices to use) TestNG when first writing their tests. Maybe it is due to the fact that it looks simpler, or maybe due to the fact that it came out after JUnit, learning from its initial failures and, taking advantage of that, brought to the table annotations, data providers, factories and much more that made everything looks nicer and cleaner. At that time, JUnit had some nasty ways of declaring test methods and lacked many useful features such as parameterization. So I think its popularity is due to the bad fame that JUnit had back then. Now, things have changed.

TestNG makes running tests in parallel a breeze, just change a word in here and a number over there in an XML config file (an XML...really?), and *BAM!* your tests run in parallel. But, ease of use comes with a hidden price: it makes it difficult to write tests and frameworks that work correctly (as expected), due to the way TestNG runs the tests (as in, test *methods*) in parallel.

Specifically, you can set TestNG XML suite `parallel` attribute to:

Parallel	Documented behaviour	Personal observations
methods	TestNG will run all your test methods in separate threads. Dependent methods will also run in separate threads but they will respect the order that you specified.	● This case is the only **real** way of achieving **fully** parallel test runs in TestNG. ● It **does not** spawns a **different** instance of the test class for each thread. ● You better have a thread safe class when you run your tests or everything will fail.
classes	TestNG will run all the methods in the same class in the same thread, but each class will be run in a separate thread.	● Will use **same** thread for the **whole** class. ● Parallelism happens at "suite level", not at "test level". If you have a large amount of tests per suite (class), it won't help to speed up test execution that much. ● No thread safety issues *at class level* are to be expected here.
tests	TestNG will run all the methods in the same <test> tag in the same thread, but each <test> tag will be in a separate thread. Will use as many threads as possible to run the tests.	● Good luck maintaining your XML ● You actually loose running tests (methods) in parallel ● Non thread safe suite classes will run fine
instances	TestNG will run all the methods in the same instance in the same thread, but two methods on two different instances will be running in different threads.	● No one has ever seen more than one test instance ● You actually loose running tests (methods) in parallel ● Non thread safe suite classes will run fine

A *tests* value in `parallel` attribute refers to `<test>` tags in its XML file, *not to actual test methods* annotated with `@Test` (that would be *methods* value). I know...confusing.

Similarly, `thread-count` attribute allows you to specify a positive integer to define how many threads you want to spawn.

Following there are examples of a typical TestNG XML file, in both single and multi threaded versions:

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Sample" verbose="0">
   <test name="Regression">
       <classes>
           <class name="com.automation.tests"/>
       </classes>
   </test>
</suite>
```

Regular single threaded XML file definition

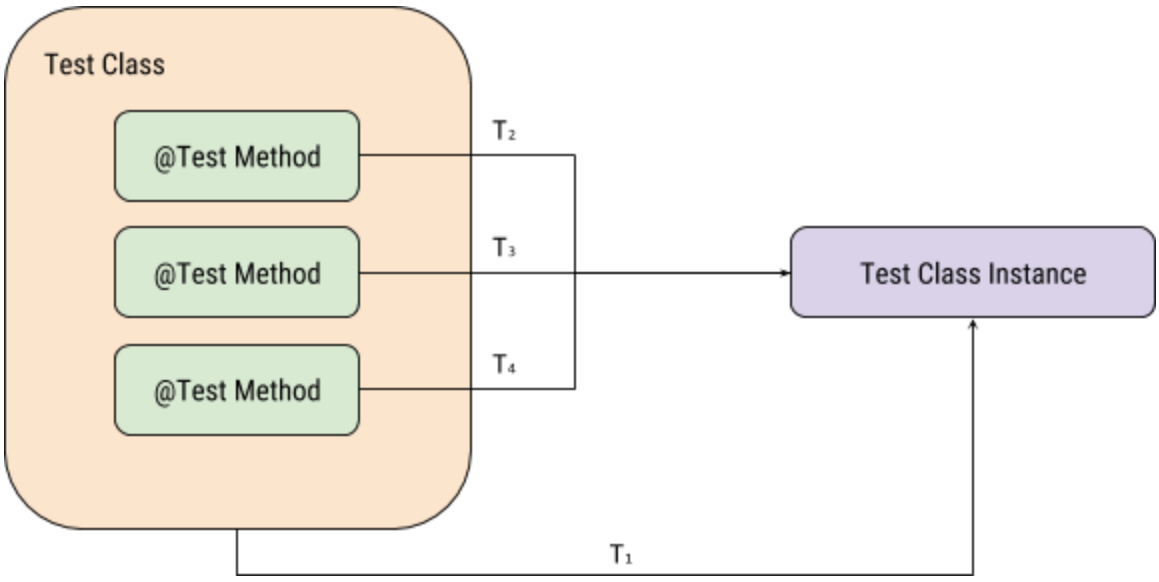
```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >
<suite name="Sample" verbose="0" parallel="methods" thread-count="10">
   <test name="Regression">
       <classes>
           <class name="com.automation.tests"/>
       </classes>
   </test>
</suite>
```

Same XML file definition with multithreaded (per test method) support

This is what really happens when you run your *tests methods* in parallel with TestNG:

1. It creates a ***single*** instance of the test class in one thread (let's call it T_1).
2. According to a numeric value in its XML config, it creates a thread pool of *X* threads: T_2 to T_n.
3. It runs each method marked with a `@Test` within the test class using the defined threads against the test class created in step 1.

Execution diagram



This means that *any* variable defined *outside* methods annotated with `@Test` (this is, any non local variables) **must** comply with one of the following things:

- To be Thread-Safe
- Make it Thread-Safe. For example, hold it inside a ThreadLocal.
- Not to exist. Find a way to not have to define the variable there, and achieve the same expected behaviour by doing things in a different way.

If you fail to meet this requirements, then you will run into weird and unpredictable issues, typical problems in the realm of multi-threading.

In short, the problem with TestNG is not the **number of threads** it can create or **how** it creates them, the problem resides in the **number of class instances** that created threads run against: It is always many threads against a single class instance.

You can see some **official** mentions (as Cédric Beust, author of TestNG, provides the answers) to this issue in the following pages:

- TestNG Users Google Group
- Cédric Beust blog
- TestNG Users Nabble Group

It is worth mentioning also that TestNG, does not always run its `@AfterXXXX` and `@BeforeXXXX` annotations when those are defined higher in the test class hierarchy. It is not wise to assume that the code that you put in there will be executed. You might want to consider declaring attribute `alwaysRun = true` in your annotations if your setup/teardown methods are not being called.

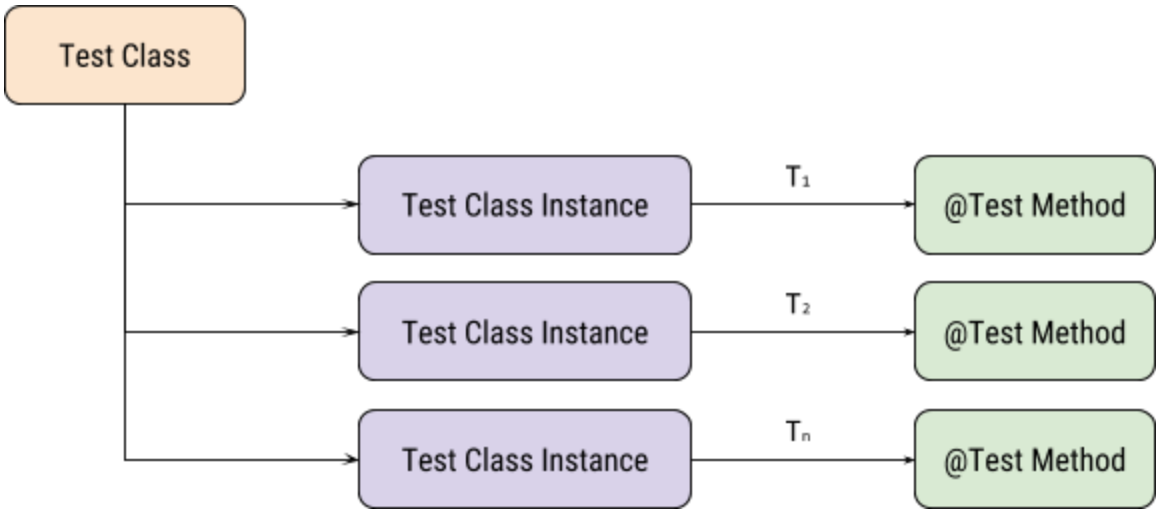
JUnit

My personal favourite. Associated mostly with unit testing in Java, it is a great tool to write your everyday integration tests with it.

The way JUnit runs the tests in parallel:

1. It scans for test methods in test classes.
2. It creates an instance of the test class for every test method .
3. It creates a thread pool of *X* threads: T_1 to T_n.
4. It runs each test method using the available threads against its corresponding test class instance created in step 2.

Execution diagram



JUnit favors running tests in *isolation*. Making it *a little bit* more error resilient when dealing with multiple threads. This does *not* mean that I'm telling you to write crappy code that does not care about thread safety.

In its later versions, JUnit has included Watcher, Categories, Rules, Theories and many other really powerful notions for testing that will be the pillars of next generation testing tools.

🔗 **Link:** Take a look at a [this simple class implementation] of a test runner in JUnit that provides parallel test execution as well as parameterized tests.

With both test runners, you should assume that there is a “*main*” thread which sets up/tears down tests, launches listeners and *then* spawns *other* threads that will be the ones that actually execute the test code...or not , it actually depends on which *other* frameworks/extensions/listeners you may be using together with your test runner. But it’s good to be always suspicious.

Data Driven Tests

Referred to as DDT, is a term that describes testing done using different sets of inputs along with their verifiable output for a particular test logic. This way tests can be *parameterized* with one or more arguments, avoiding having to write the same test for different values and also to hardcode values inside it.

Tests runners are the ones responsible for running your tests in this fashion, when specified to do so. Both Junit and TestNG support this feature, although they differ in naming and usage.

There are several sources of test data but, at the end, they will depend on the size and amount of data your tests need. Ideal sources are:

- ✓ Use an in-memory database such as hsqldb so that you can pre-populate it with a "production-style" set of data from a `.sql` file
- ✓ Generate complex Java objects to be used as test parameters from:
  - 👌 Factory methods
  - 👌 Unmarshall JSON/YAML/XML (in that order or preference) files previously stored in disk representing your object states
- ✓ JPA can be used to access real databases to retrieve values in real time. Try to rely on well known frameworks to ease this operations. For example, Spring.

TestNG

Implements DDT with DataProviders, which allows to supply the values you need to test. A Data Provider is simply a method annotated with `@DataProvider`. This method can reside in the same class or in a different one than the tests that use it reside.

The Data Provider method can return one of the following two types:

- An array of array of objects (`Object[][]`) where the first dimension's size is the number of times the test method will be invoked and the second dimension size contains an array of objects that must be compatible with the parameter types of the test method.

- An `Iterator<Object[]>`. The only difference with `Object[][]` is that an Iterator lets you create your test data lazily. TestNG will invoke the iterator while `hasNext()` is true and then the test method with the parameters returned by `next()` one by one. This is particularly useful if parameter initialization is costly, such as consuming from a database. This is the preferred way to define Data Providers.

```
@DataProvider(name = "UsernameProvider")
public Object[][] createUsernames() {
   return new Object[][]{
           {"Ruso"},
           {"I'm gonna fail with this one"},
   };
}

@Test(dataProvider = "UsernameProvider")
public void parameterizedTest(String username) {
   assertEquals(username, "Ruso");
}
```

TestNG data provider usage example

```
@DataProvider(name = "UsernameProvider")
public Iterator<Object[]> createData() {
   return getDataFromSomewhere()
           .stream()
           .map(s -> new Object[]{s})
           .collect(toList())
           .iterator();
}
```

TestNG data provider Iterator version example

As you can see from the example, a test that wants to make use of a data provider must define so with the `dataProvider` argument in the `@Test` annotation. It is also possible to specify which is the class that contains such data provider and to refer to the dataprovider method name, instead of defining and referencing a name in the dataprovider annotation.

JUnit

Here is where a clever and modularized design in the test runner shines bright. At first look JUnit does not seem to have such fancy DDT feature. Truth is that not only it does but it also allows for people to create and use different implementations of them.

Before moving forward, I must clarify that JUnit calls “test runners” to sub-modules that extends from a `Runner` abstract class provided by the framework. You can think of JUnit as an engine that runs runners, which in turn runs tests. Having this in mind, the following paragraphs will mention different *test runners* that you can use in JUnit by marking your base test class with the `@RunWith(TestRunnerOfChoice.class)` annotation.

You can use the Parameterized test runner to create data driven tests. It's not terribly well documented, but the basic idea is to create a *public static* method (annotated with `@Parameters`) that returns a *Collection* of Object arrays. Each of these arrays are used as the arguments for the *test class constructor* (not the tests), and then the usual test methods can be run using fields set in the constructor. It is also possible to inject data values directly into *public* fields without needing a constructor in the test class by using the `@Parameter` annotation.

```
public ParallelTests(String browserName, String browserVersion) {
   this.browser = browserName;
   this.version = browserVersion;
}

@Parameters(name = "Capability {0} {1}")
public static LinkedList<String[]> getEnvironments() throws Exception {
   return new LinkedList<String[]>() {
       {
           add(new String[]{"chrome", "50"});
           add(new String[]{"firefox", "latest"});
       }
   };
}

@Test
public void testSimple() throws Exception {
   out.println(format("Running test using %s / %s", browser, version));
}
```

JUnit parameterized test example

For completeness sake, you can see the complete test sample here.

Just in case you do not like this particular test runner, you can try out any of the following ones:

- JUnit-Dataprovider
- JUnitParams
- Zohhak
- Keep Googling...
- Build your own 😀

Things to avoid while doing DDT

🚫 If you opt for a database storage, **do not** perform SQL queries manually. Use ORM libraries like Hibernate, Sormula or pBeans[2] to delegate the dirty work.

🚫 **Do not** use TestNG's feature for providing arguments from its *testng.xml* config file. You will end up having several places from where data comes from and, either way, that XML cannot represent complex structures or objects that your tests may need.

Web Front End Testing with Selenium/WebDriver

What is WebDriver?

WebDriver is a remote control interface that enables introspection and control of browsers.
It provides is a set of interfaces to discover and manipulate DOM elements in web documents and to control the behaviour of a user agent. It is primarily intended to allow us to write tests that automate a user agent (browser) from a separate controlling process (tests).

It is platform and language neutral, meaning that you can find implementations for all major platforms (Windows, Linux, OS X, etc) and programming languages (Java, C#, Ruby, JS, etc).

Selenium, is just one implementation of WebDriver.

In practical simpler terms, WebDriver is a library for remotely controlling browser applications such as Chrome, Firefox and Internet Explorer, among others.

Along this document, you will see terms that map to a correlative term in WebDriver API.
Some of them are:

Term		Java WebDriver API analogue
Driver		`WebDriver`
Web Element, element, DOM element/node	Singular	`WebElement`
	Plural	`List<WebElement>`
Locator		`By (and its subclasses)`

WebDriver Architecture Overview

WebDriver has a server/client architecture, the client is what we call the driver instance (just a `RemoteWebDriver` subclass / `WebDriver` implementation) in our Java code, and the server is an *external* binary process that controls the browser.

Here you have the relations:

Client (WebDriver implementation)	**Server**
GeckoDriver	GeckoDriver server
FirefoxDriver	Embedded into selenium-server.jar until recently (versions 2.x). Firefox 47+ and Selenium Server 3.x must use GeckoDriver.
SafariDriver	SafariDriver now requires manual installation of the extension prior to automation. Latest release 2.48.0
ChromeDriver	ChromeDriver server
InternetExplorerDriver	IEDriver server
EdgeDriver	MicrosoftWebDriver.exe
AndroidDriver	Embedded into Appium server
IOSDriver	

Communication between clients and servers is done in a stateful session using HTTP protocol carrying JSON payloads. A session begins when a client is instantiated (and connects to the server) and is closed by the server upon one of the following conditions:

- A `.quit()` method of WebDriver API is invoked
- When an inactive session times out and expires

Not all server implementations support every WebDriver feature. Therefore, the client and server use a particular JSON object with properties describing which features a user requests from a session to support. This JSON object is known as Capabilities, although people also calls it DesiredCapabilities, but that is actually the name of the implementation of this object in Java. It simply describes a series of key/value pairs.

A list of the *commonly* used capabilities that a session *may* support can be found here.

Take into account that every server implementation can support capabilities that are not in the WebDriver standard. Here you have some more capabilities for Chrome driver, as an example.

If a session cannot support a capability that is requested in the desired capabilities, no error is thrown; a read-only capabilities object is returned that indicates the capabilities the session actually supports.

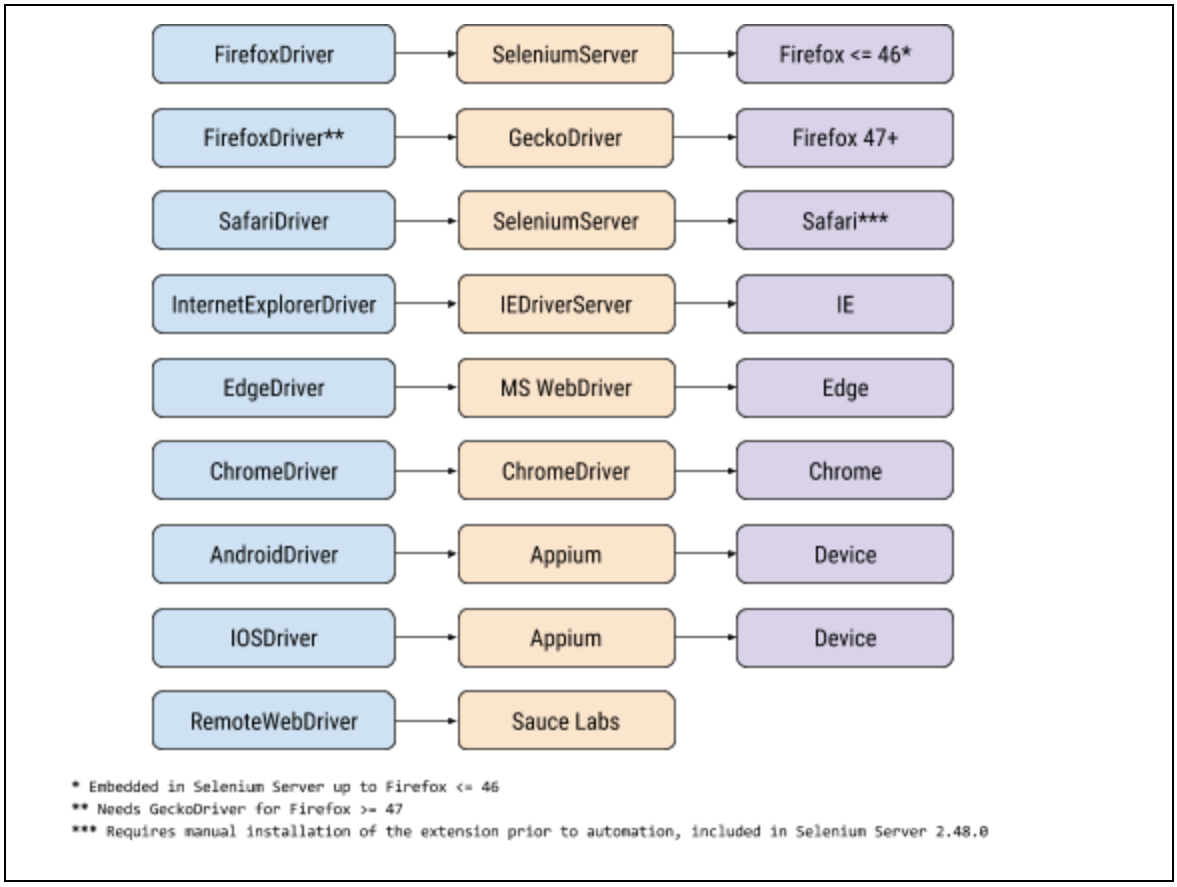
As you can see, there are multiple “servers”. So, how does Selenium knows which one to use when you have tests running against multiple browsers? It takes a look at the values present in the Capabilities object (specifically `browserName`, `platform` and `version`) being sent as part of the JSON payload in the communication. This way, a server is aware if it should answer it.

Despite there are many servers, one of them (specially) is not like the others. Most of these server are, as I call it, “*leaf servers*” because once they start running they:

- Open a TCP port to listen for incoming connections
- Accept new connections
- Spawn their respective browser process to control it.

That’s it. If you request a capability that they are not supposed to handle they just report an error, they are always the “leaf node” in the tree call.

Let me try to come up with a diagram that may show this a little bit better:

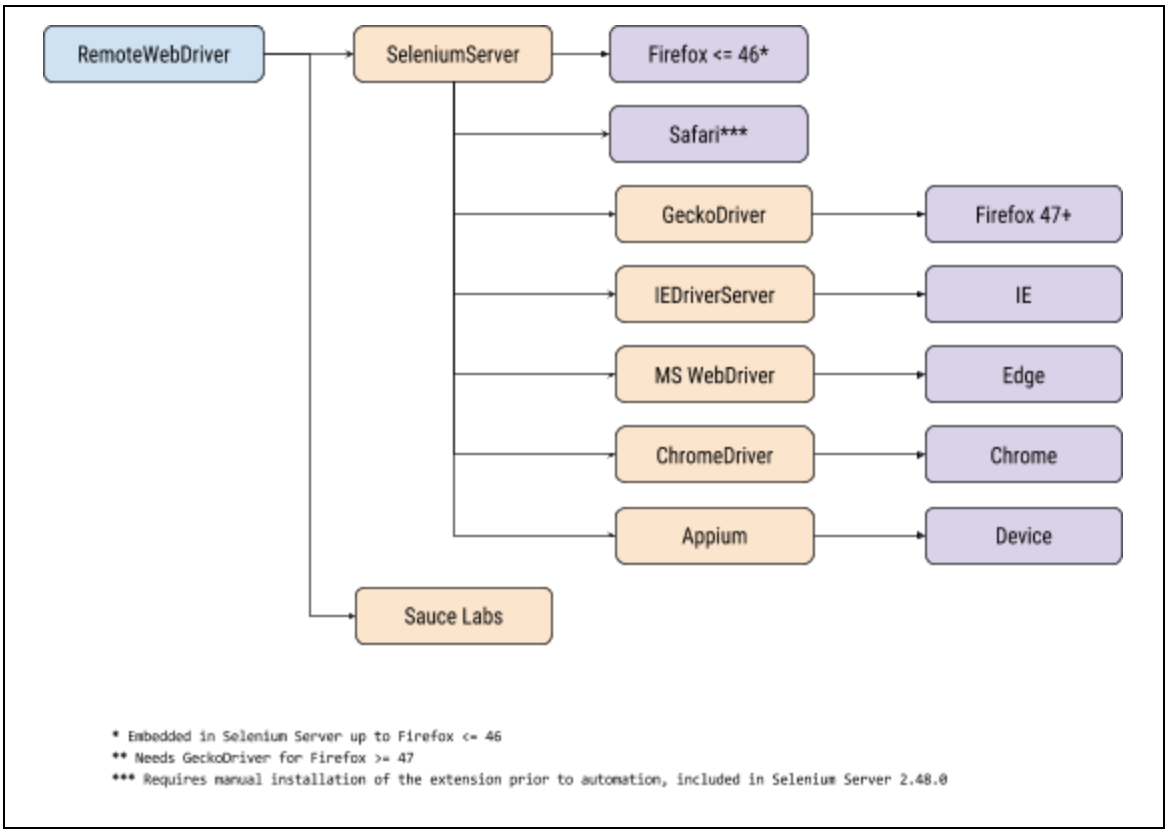


Trivial usage of WebDriver implementations

On the other hand, selenium-server not only does the same for Firefox and Safari browsers, but it also acts as a “router”. Meaning that, if a desired capability is not within itself to handle, it will try to launch the process of the server that can handle such request and, if successful, it will route the connections to it.

Recommended way of using WebDriver

Understanding WebDriver architecture allows you to greatly simplify the way your framework could create instances of WebDriver. I propose to implement things in a way similar to this one:



Suggested WebDriver usage

According to the diagram above, the only client that we need to take care of is RemoteWebDriver!. Now we can focus only on the Capabilities we want it to have, but not particular implementations.

🔗 **Link:** An example implementation of this can be found here.

Also, there are some JVM properties (since Selenium Server is written in Java) that allows it to know where in the file system each of the WebDriver servers can be found:

WebDriver Server	JVM Property
GeckoDriver	webdriver.gecko.driver
FirefoxDriver	Not needed for Firefox <= 46
SafariDriver	Not needed
ChromeDriver	webdriver.chrome.driver
InternetExplorerDriver	webdriver.ie.driver
EdgeDriver	webdriver.edge.driver
OperaDriver	webdriver.opera.driver
PhantomJsDriver	phantomjs.binary.path

This means that you can set these values via JVM arguments to the Selenium Server at launch time:

```
> java -jar selenium-server.jar
-Dwebdriver.chrome.driver=/absolute/path/to/binary/chromedriver
```

Or programmatically, in Java, as:

```
System.setProperty("webdriver.chrome.driver", "/absolute/path/to/chromedriver");
System.setProperty("webdriver.opera.driver", "/absolute/path/to/operadriver");
System.setProperty("webdriver.ie.driver", "C:/absolute/path/to/IEDriverServer.exe");
System.setProperty("webdriver.edge.driver", "C:/absolute/path/to/MicrosoftWebDriver.exe");
System.setProperty("phantomjs.binary.path", "/absolute/path/to/phantomjs");
System.setProperty("webdriver.gecko.driver", "/absolute/path/to/geckodriver");
```

Locators

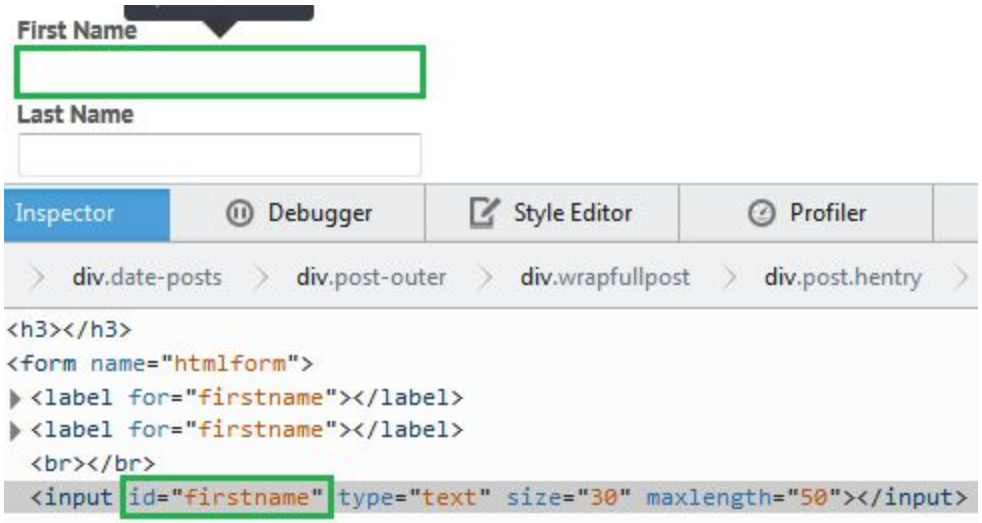
A locator is an *statement*/*definition* that allows to identify/locate a single (or many) web element(s) in a web page, hence the name "*locator*".

In WebDriver terms, it holds two things:

- *How* (strategy) to find the element:
    - Id
    - Name
    - CSS
    - XPath
    - ClassName
    - TagName
    - LinkText
    - PartialLinkText
- An *expression* (a String) that the above strategy supports to find it.

Each strategy represents a way of finding elements in a web page. Their names derive from the actual pattern they look for in a web element to consider it found.

`Id` strategy will try to find the element by looking at the *id* attribute of a node in the DOM tree of the tested web page:



`Name`, `ClassName`, `TagName`, `LinkText` and `PartialLinkText` strategies behave the same way as `Id` does but attempt to match elements by their name, class, tag and text() attributes respectively.

`CSS` and `XPath` strategies are the only ones that find elements by matching an *expression* (not just a literal string that you can find in the actual DOM) against nodes of the DOM tree.


The preferred selector order should be: id > name > css > xpath

There are a few things you should consider when deciding the locator to use:

- 🚫 **Do not** use brittle locators. It's easy and tempting to use complex XPath expressions like `//body/div/div/*[@class="someClass"]` or CSS selectors like `#content .wrapper .main` but while these might work when you are developing your tests, they will almost certainly break when changes to the web application's HTML output are made.

- ⚠ `name` attributes are *not guaranteed* to be *unique*, but *usually* they tend to be.

- ⚠ Be smart with XPath usage. Avoid using *full paths*. This type of locator should be as short, simple and straightforward as possible.

- ⚠ Keep track of flaky locators and, whenever possible, talk to the development team and have them include a unique `Id` attribute for the troublesome WebElements.

Why did the locator fail?

Before you define a better locator for the failed web element, it is important to first understand *why* the locator failed. In some cases, the problem might not even be with the locator, so changing the locator would not help.

Here are some of the common causes:

Problem	Possible Cause
Element attributes changed in a way that invalidated the locator strategy	Synchronization issue
Element does not exist (yet)	
Element not visible/clickable	
Test proceeding too fast	
Element is in an IFrame/different window	Wrong WebDriver scope

Sometimes, although we are using a really specific and good locator to find our web element, test keeps failing because the “element is not visible” or “another element would receive the click”. This is due to the way HTML is structured, causing to be “invisible layers” on top of our target element.

Consider this real life example of a simple checkbox for a “Remember Me” functionality in a login page:

```
<li id="remember-li">
   <div class="checker" id="uniform-remember1">
       <span class="">
           <input id="remember1" name="remember" type="checkbox" value="true">
       </span>
   </div>
</li>
```

The “logical” assumption would be to choose the input’s id attribute as selector for our checkbox. The thing is, there is a div wrapping it with a kind of suspicious class that seems to give the checkbox some “format” (“checker” class) therefore, something is rendering *on top* of the checkbox to make it more “visually appealing”, but rendering the *original* input invisible. Now you click on a div, which, in turn, will forward the event to the underneath input. This screws up our direct interaction with the input tag from WebDriver tests with exceptions and error

messages similar to “Element is not visible” or “Another element would receive the click”.

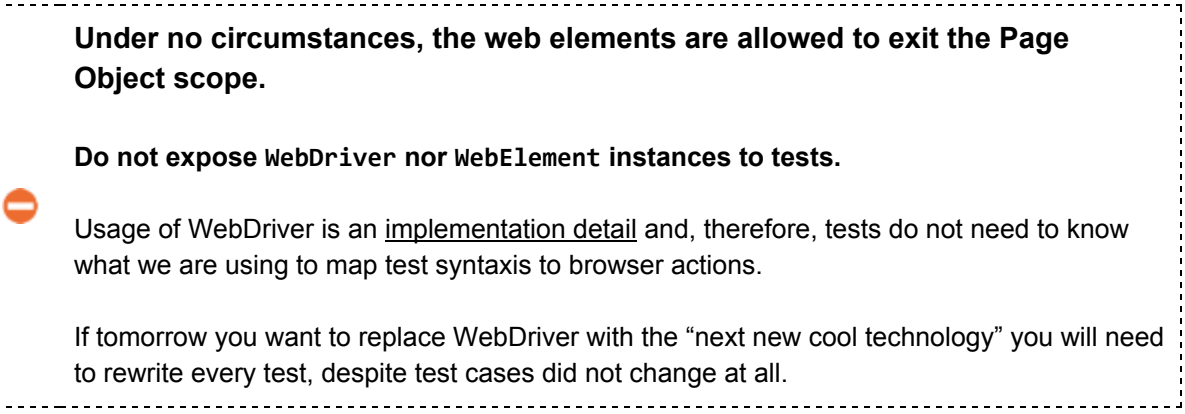
In short, sometimes things are not what they look like and, as in this case, you need to fallback to using another locator rather than the most obvious one: *id=”uniform-remember1”*.

Model your pages as objects - Page Object strategy

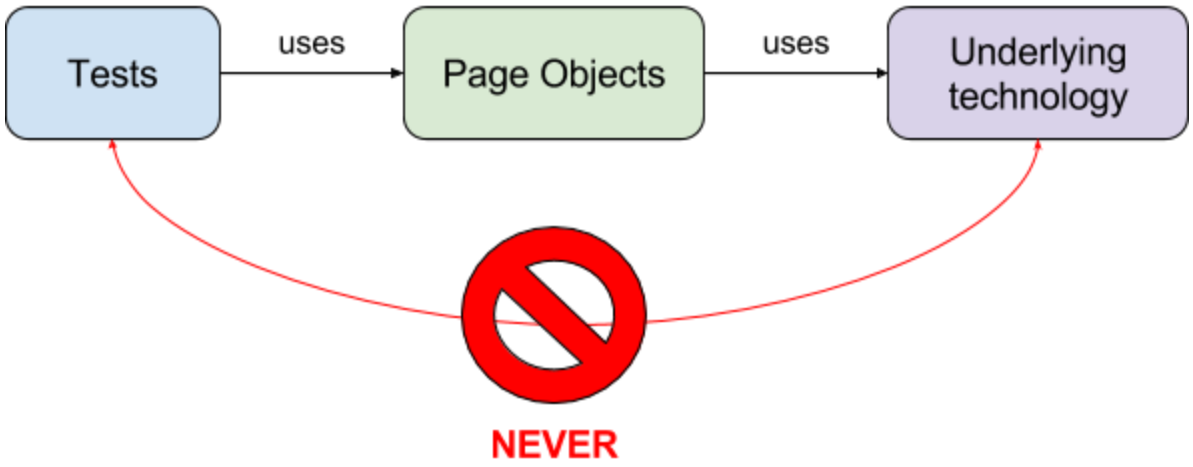
Page Object is the name given to a pattern typically used in web front end automated testing. Despite the “cool” name, it is nothing more than saying “let’s write all our web automated test in an OOP way, not procedurally”. This means modelling the web pages that we are automating as objects. Therefore, Page Object Model - POM, for short.

In simpler terms, this means that we create classes that:

- ✓ Will contain **private** references to interesting/interactive web elements existing on a web page.
- ✓ Those private references *may have* accessor methods of **protected** scope to let subclasses of the POM access the references.
- ✓ Will expose **public methods** for tests to deal with those private web elements reference



The end goal situation should be:



For the purposes of this document, the underlying technology is WebDriver, but that's something that the tests should never be aware of.

The reasons for doing all this should be obvious, but just in case, I'll state them:

- ✓ Reduces the amount of duplicated code (100 tests going through the same page would not define 100 times the same elements/actions)
- ✓ Centralizes related code; whatever needs fixing, the fixing will (and should) take place at a single place
- ✓ Enhances test maintenance
- ✓ Encapsulates the mechanics required to find and manipulate the elements/data in the UI
- ✓ Provides an interface that's easy to program with and hides the underlying widgetry in the window

Structure

- ✓ Page Object models System Under Test (SUT) pages and its actions and behaviours. **That's it**. Anything else, should be done by a test framework (preferably) and/or tests.

- ✓ Prefer composition over inheritance. Java does not support multiple inheritance either way. This way you can model a module (section of a page) of the SUT in a separated class, and if such module appears in different pages you can still compose it in all your POMs.

- ✓ Move functionality (methods) and state (class fields) common to **all** POMs to an abstract superclass. Make fields `private` and create accessors for them with `protected` visibility.

- ✓ Each POM should be responsible of initializing its own internals, even WebElements. It is also acceptable to delegate initialization to an abstract superclass.

- 🚫 **Do not** have constructor arguments for Page Object classes. At least, not constructor arguments of *WebDriver related types*. I know that you will probably argue something like *"If I don't have a constructor that accepts, at least, one argument for my POMs how I am supposed to provide the WebDriver instance to it?"*. My answer would be: *"You don't...or you shouldn't"*.

  Why? Because doing that would *imply* that **tests** (which *usually* are the ones that create POM instances) would *also* have to create an instance of WebDriver to pass it to the POM. As stated before, this contradicts the *holy commandment* of "**Do not expose `WebDriver` nor `WebElement` instances to tests."**.

  Take a look at this section of the document to know more about possible workarounds.

- ✓ Page navigation operations such as: `back()`, `forward()`, `refresh()`, `stop()` and maybe even `deleteCookies()` should be present in the abstract superclass common to all Page Object classes. Again, composed via appropriate holders like: Navigation and Cookies.

- ✓ Always try to model your objects using **interfaces**. Not only because they allow you to **really** design your framework in an OOP way, but also because interfaces in Java provide a **great deal of flexibility**. You can easily create *decorators* and *dynamic proxies* for their instances. You could do things like "*whenever accessing a web element, do this before or do that after*". You can think of it as Aspect Oriented Programming for free!

✓ Expose operations (methods) over web elements, **not** the web elements themselves. Tests should **never** handle web elements by themselves. Tests should not be aware of existence of WebDriver behind the scene. Public getter methods return primitives and objects but **not** WebElements.

✓ If an operation/action method in POM causes the transition from the current web page to another, it *must* return an instance of the destination's page POM. If the operation does not cause a page transition, return the instance (*this*) of the same POM. This will allow you to chain calls to the POM. Despite you can do this, do *not* invoke **long** chained operations just because you can. Keep the test readable!

✓ WebElements should be in a correct state before POM interacts with it. Wait *explicitly* for conditions to be fulfilled before operating on a web element. Example: POM exposes method *bookRoom()* that (internally) clicks on a button named "Book". Then, that method is also *responsible* for making sure that such button exists and that it is in a "clickable" state.

✓ Since tests must not deal with WebElements or any other WebDriver specific class, make sure your framework/POM required classes have the appropriate *scope*. This means, take some time to *guarantee that methods that deal with WebDriver related stuff have a package level, protected or private scopes*. They should not be required from tests, only from POMs. The lesser the scope, the better. You can always increase a scope without issues but, going from a larger scope into a smaller one **breaks client code**.

✓ Use Domain Specific Language (DSL) when possible, if *really* needed, pass around locators as By objects instead of Strings.

🚫 Avoid defining *many* static methods (so called *helpers*), try to *see* and *design* your tests, POMs and frameworks with your OOP hat on.

✓ Name public methods after *meaningful business oriented* actions or events. Meaning that, instead of having public methods like:

```
🙈 typeUsername(String username)
🙈 typePassword(String password)
🙈 clickLoginButton()
```

Instead, it would be better to have a public method such as `doLoginWithCredentials(String username, String password)`. Tests are supposed to be *functional*, it should not care if the login submit form action is made through a button or a link nor if we type or swipe letters into a text field element. Remember, public methods represent the services that the page offers.

🚫 **Do not** perform assertions in POMs. Tests should decide what, when and where something should fail, not the POMs.

✓ A POM does not *necessarily* needs to represent an *entire* page. This means that you *can* create a POM for a *portion* of a page (a reusable module, like a Navigation bar or a menu) and then *compose it* inside other POMs.

✓ It is always better not to have *huge* Page Object classes. Again, you should refactor them into several smaller POMs and then use *composition* inside other POMs.

How do I define a locator in a POM?

I said that locators are an *statement*/*definition* because, in Java, locators are *mostly* defined using annotations, which are a form of metadata (complementary information, mark or decoration for a type, field or method definition). Specifically, the annotation name is `@FindBy`:

```
class APom extends BasePage {

   @FindBy(how = How.ID, using = "elementId")
   private WebElement anElement;

   @FindBy(how = How.CSS, using = ".selected-item")
   private WebElement selectedElement;

   @FindBy(how = How.XPATH, using = "//div[@someAttr='value']")
   private List<WebElement> severalElements;

}
```

Examples of WebElement(s) definitions in a POM

You can also use a *shorter version* of `@FindBy` that skips the definition of the `how` and `using` arguments like this: `@FindBy(id = "anId")`.

In case of need, know that you can group several `@FindBy` annotations inside a single `@FindBys` or `@FindAll` annotation. If possible **avoid using this feature**, as it will make the POM brittle, less readable and slower.

- `@FindBys` will find all DOM elements that matches **each** of the locators, in sequence.
- `@FindAll` will search for all elements that match **any** of the `@FindBy` criteria. Note that elements are *not* guaranteed to be in document order.

```
@FindBys({@FindBy(name = "name"), @FindBy(css = ".another")})
private List<WebElement> elements;

@FindAll({@FindBy(id = "anId"), @FindBy(css = ".another")})
private WebElement manyElements;
```

Examples of @FindBys and @FindAll definitions in a POM

As an important side note, know that you can *cache* individual WebElements by using the `@CacheLookup` annotation. If you use this the WebElement will be *lazily* evaluated the *first time* that you utilise it, but from that point onwards it will not be looked up again (which means you could start getting *StaleElementExceptions* if the element changes). This is useful to speedup element interactions with known to always-be-there elements.

🚫 **Do not** use `@CacheLookup` on dynamic elements!

✓ Use `@CacheLookup` on elements that you *know for sure* will always be there.

Sometimes, when you have locators that change depending on some *known* circumstance it may be helpful to define a class like:

```
public class DynamicLocators {

   public static class Home {

       // Item not present in DOM until you click on a button
       public static final By DynamicItem = new By.ByCssSelector(".selected-item");
   }

   public static class Login {

       // Message that varies the text depending on who is logging in.
       public static By WelcomeMessage(String user) {
           return new By.ByLinkText(format("Welcome back %s!", user));
       }
   }
}
```

And use it from POMs like:

```
class POM_X {
   public POM_X doStuff() {
       FindElement(DynamicLocators.Login.WelcomeMessage("Ruso"));
       ...
   }
}
```

Again, use it **only** on required situations. **Do not store every locator this way**!

As a final word for this section, you should know that you can *create your own location strategy* if you **really** have a need to do so.

How do I initialize all elements defined in my POM?

Once you have a POM class defined, you can initialize all `WebElement` and `List<WebElement>` declarations by "*processing*" your POM instance or class as:

```
PageFactory.initElements(webDriverInstance, new MyPOM());
PageFactory.initElements(webDriverInstance, MyPOM.class);
```

Different ways of initializing POM elements

When creating a web automation framework you are most likely willing to define a "base" PageObject superclass that will contain common reusable functionality and every POM will be extending from it. Its constructor is, therefore, a good location to place the POM initialization code.

Consider doing something similar to:

```
public abstract class PageObject {
   public PageObject() {
       PageFactory.initElements(webDriverInstance, this);
   }
}
```

Example for initializing POM elements from superclass constructor

This way, every time you create a POM instance, your elements will be initialized from the beginning.

> ⚠ **Warning**
>
> `PageFactory` *always* initializes fields as long as its type are `WebElement` or `List<WebElement>`.
>
> *Do not assume* that because you have not annotated them with `@FindBy`, `@FindBys`, or `@FindAll` it won't process them.
>
> In case there are no such annotations present in the fields `PageFactory` will set a Java proxy with a locator of type "*ID or NAME*" equal to the field name. The field *will never be null*.

Some words on element staleness

A WebElement is a reference to an element in the DOM.

A *StaleElementException* is thrown when the element you were interacting with gets destroyed and then recreated. Most complex web pages these days will move things about on the fly as the user interacts with it and this requires elements in the DOM to be destroyed and recreated.

When this happens, the reference to the element in the DOM that you previously had becomes stale and *you are no longer able to use this reference to interact with the element* in the DOM. You will need to *refresh* your reference or, in real world terms, find the element again.

WebElement instances initialized by `PageFactory.initElements` static method are actually a Java dynamic proxy representing a DOM element. When the DOM element is destroyed by the browser, WebElement is marked "stale" and *can't be used anymore*.

If you have a *similar* DOM element on the page, you can obtain it with *findElement*, and Selenium will create a *new* WebElement object (another Java dynamic proxy) for this *new* DOM element.

Your old WebElement object *will remain stale* because underlying DOM object is destroyed and *can never be restored*. Similarity is not equality.

Documentation

- ✓ Documentation is tough but necessary. At least, document every public method and, if possible, provide a really brief example of how to use it.

Interactions between non public types

Several times I've mentioned that the scope (visibility) of a type *should be as reduced as possible*. Specially when dealing with WebDriver instances. The whole idea is for your POMs and test superclasses to have access to the WebDriver factory, but *not* the client (or any other) code.

The goal here is to design packages and types distribution within your framework in a way that results impossible to gain access to critical types anyhow other than from where intended.

For this goal, you can consider not only the design of the package structure but also some of the following ideas:

- 💡 Use an in-memory event/bus system
- 💡 Use an Inversion of Control (IoC) / Dependency Injection (DI) container, such as:
    - PicoContainer
    - Petite Jodd
    - Dagger
    - As a last resort, Spring (would be an overkill to use it *just* for this purpose)

Handling synchronization

WebDriver performs some pauses after receiving instructions like *"navigate to this site"* [`.get()`], *"click this element"* [`.click()`] and *"find this element in the page"* [`.findElement()`].

Those waits are known as *implicit waiting*, it is something that happens behind the scene and we do not invoke ourselves, although we *can* (and should) control their timeouts.

There are 3 main timeouts that WebDriver's API defines:

```
driver.manage().timeouts().implicitlyWait(10, SECONDS);
driver.manage().timeouts().setScriptTimeout(5, SECONDS);
driver.manage().timeouts().pageLoadTimeout(30, SECONDS);
```

WebDriver timeout definitions examples

Some may look similar but they are not the same!. Be sure to set them up in your test framework:

- **`pageLoadTimeout`**: Time to wait for a page load to complete before throwing an error
- **`setImplicitTimeout`**: Time to wait when searching for an element if it is not immediately present
- **`setScriptTimeout`**:Time to wait for a JavaScript evaluation to return

When these waiting times are not set, are not enough or need to vary from one POM method to another because an action takes longer to fulfill than another, I've seen people do a lot of mediocre stuff to cause the program to "wait" or "be delayed" using statements like `Thread.sleep()` or loops with conditions and counters that ultimately relies on it.

This is wrong!. **Do not** use `Thread.sleep()`, ever….just...**don't**.

Never wait for a fixed amount of time.

We refer to *synchronization* as the act of **actively** waiting for something to happen before or after performing an action. It is not limited to only WebElement interactions. It is also called *explicit wait*.

What's the difference you say? Well, here is a quick rundown on the differences between

explicit and implicit waits:

Explicit wait

- ✓ Documented and defined behaviour
- ✓ Runs in the local part of selenium (your coding language)
- ✓ Works on any condition you can think of
- ✓ Returns either success or timeout error
- ✓ Can define absence of element as success condition
- ✓ Can customize delay between retries and exceptions to ignore

Implicit wait

- 😩 Undocumented and practically undefined behaviour
- 😩 Runs in the remote part of selenium (the part controlling the browser)
- 😩 Only works on find element(s) methods
- 😩 Returns either element found or (after timeout) not found
- 😩 If checking for absence of element must always wait until timeout
- 😩 Cannot be customized other than global timeout

So, basically, **implicit waits sucks**, and I would even suggest you to simply **stop using them**.

How?

In your framework (during driver initialization) set the `setImplicitTimeout` to **1 seconds**.

Why?

Because it will **force you** to define and use **explicit** waits *before* and/or *after* operating on elements. **Your tests will fail so fast** that you will notice that you forgot to add synchronization in some of your POMs before you have time to say *"what the hell?"*.

For quick reference, here is how you define a basic instance of WebDriver explicit wait:

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

It should go without saying it that:

🚫 You **do not** need to create an instance of WebDriverWait *each time* you want to wait. Create it *once* in your base POM and *reuse it*.

🚫 I've hardcoded the timeout value for this example. You are **not allowed** to do that 😏

✓ ExpectedConditions is a "conditions" provider class, you will find pretty much whatever condition you need in it, operating over driver, element(s) and/or By locators. The complete list of conditions, at the time of writing this document, is:

```
titleIs                                 elementToBeSelected
titleContains                           elementSelectionStateToBe
urlToBe                                 numberOfElementsToBeMoreThan
urlContains                             numberOfElementsToBeLessThan
urlMatches                              numberOfElementsToBe
presenceOfElementLocated                attributeToBe
visibilityOfElementLocated              attributeContains
visibilityOfAllElementsLocatedBy        attributeContains
visibilityOfAllElements                 attributeToBeNotEmpty
visibilityOf                            visibilityOfNestedElementsLocatedBy
elementIfVisible                        visibilityOfNestedElementsLocatedBy
presenceOfAllElementsLocatedBy          presenceOfNestedElementLocatedBy
textToBePresentInElement                presenceOfNestedElementLocatedBy
textToBePresentInElement                presenceOfNestedElementsLocatedBy
textToBePresentInElementLocated         invisibilityOfAllElements
textToBePresentInElementValue           or
textToBePresentInElementValue           and
frameToBeAvailableAndSwitchToIt         javaScriptThrowsNoExceptions
frameToBeAvailableAndSwitchToIt         jsReturnsValue
frameToBeAvailableAndSwitchToIt         alertIsPresent
frameToBeAvailableAndSwitchToIt         numberOfwindowsToBe
invisibilityOfElementLocated            numberOfWindowsToBe
invisibilityOfElementWithText           not
elementToBeClickable                    findElement
elementToBeClickable                    findElements
stalenessOf                             attributeToBe
refreshed                               textToBe
elementToBeSelected                     textMatches
elementSelectionStateToBe
```

WebDriver list of available conditions for WebDriverWait

Also, know that you can also use the underlying generic class `FluentWait<T>` (included in Selenium library) to create wait instances more flexible than WebDriverWait. You can then create conditions as needed implementing `Predicate<T>` and `Function<T,K>`.

- 🔗 **Link:** Here you can check out some customized conditions.
- 🔗 **Link:** Some implementations of FluentWait based waiting can be found here, make sure to check its usage by looking at the tests that use them.

> ⛔ **Do not** mix WebDriverWait with another wait implementation *Pick one and stick to it*.
>
> If you choose to implement your own custom wait implementation, **do not** mix Selenium's ExpectedConditions methods with the ones you are going to write for your own wait. Just use the ones you define so you have control over them.
>
> Take advantage of Java 8 syntax with lambdas and streams to define them *almost* in line.

Multithreading and parallelism

Before getting our hands dirty with code, designs and ideas, you need to know that there is a difference between "*running tests against multiple browsers*" and "*running tests **in parallel** against multiple browsers*".

Sounds silly, but know that because you've just finished deploying a Selenium Grid with 50 top-notch nodes you won't get your tests to run faster nor in a shorter period of time than before.

Selenium Grid takes care of distributing (and balancing) test executions to different nodes that may be running different browsers/versions but, if test executions are send one after another (sequentially) then there is absolutely no gain in terms of speed.

The reasons that cause this behaviour usually are:

- Test runner does not support parallel test runs or it is not configured to do so.
- Test runner supports parallel test runs but framework/POMs/test code did not contemplate the idea of multiple threads accessing it so it randomly fails with weird exceptions followed by the classic "*It works on my machine*" phrase.

When building a test framework from scratch, *always* design it considering the ability to run tests in *parallel*. Nowadays it is a **must**. Fast feedback is the cornerstone of automated testing, and parallel test execution supposes reduced tests run times.

Of course, the more shared state you include in your framework classes and POMs the more you will have to look out for thread safety.

Unfortunately, **WebDriver is not thread-safe**. Therefore, in my humble opinion, the best practice is to run each test using an individual WebDriver instance in a separate thread.

In practice, this means that you **never** have to define a simple variable or a singleton to contain the WebDriver instance. As I said, you should consider having a WebDriver instance *per thread*.

How many threads you may ask? The optimal number of threads lurks around

```
int threadNum = Runtime.getRuntime().availableProcessors() * 2;
```

For this purpose, the *ThreadLocal* pattern comes in handy. As it's name implies it is a wrapper

class that provides thread-local variables. These variables differ from their normal counterparts in that each thread that accesses one (via its `.get()` or `.set()` method) has its own, independently initialized copy of the variable. ThreadLocal instances are typically *private static* fields in classes that wish to associate state with a thread (e.g., your WebDriver instance).

🔗 **Link:** An *example* of ThreadLocal usage can be found here.

⛔ One of the problems that arise when using ThreadLocal pattern is that you cannot view nor access (not even to dispose of) the total of objects that the ThreadLocal instance may be holding. There is no way to cleanup ThreadLocal values except from within *the thread that put them in there in the first place* (or when the thread is garbage collected - not the case with worker threads). This means **you must take care to clean up your ThreadLocal when a test is finished**, because after that point you may *never* get a chance to enter that specific worker thread, and hence, will *leak memory*.

So in practical terms, to *set* and *remove* instances in a ThreadLocal you will, usually, have to take advantage of the test runner you are using. Specifically, of its *before* and *after* **test/suite** annotations.

Logging

✓ Prefer the Simple Logging Facade for Java (SLF4J) which serves as a simple facade or abstraction for various logging frameworks (e.g. logback, log4j) allowing you to plug in the desired logging framework at *deployment time*.

✓ Use a private static Logger class defined in base POM class and in the testing framework itself to log to a file, console, etc.

✓ Create an implementation of the `WebDriverEventListener` interface and register it in a `EventFiringWebDriver` instance in your testing framework. Listeners are event-driven and provide methods that can be implemented to handle most of the interaction between WebDriver and the System Under Test (SUT). This allows you to perform operations before and after WebDriver steps, without polluting the tests.

> 🔗 **Link:** See an basic implementation of a WebDriver listener here

✓ Another way to extend your logging is to add a listener to the test harness framework that you may be already using (TestNG/jUnit). You'll gain access to event from test suites & methods. At those times, it is usually a good idea to attach a listener to get screenshots and/or HTML source dumps from driver upon test failures.

✓ When you define assertions to check for expected conditions, be very descriptive on the error message you will log. This will save you a lot of time during the analysis of the failures.

🚫 **Do not** log actions manually during test scripts, this render tests longer and unreadable.

🚫 **Do not** use *System.out*.

🚫 Mind the log type. If it is not really an error **do not** log it as such.

Configuration

Externalize configuration

🚫 **Do not** hardcode values into your framework.

🚫 **Do not** have a public class with public static fields manually mapped to properties. This implies that the more the properties you have, the more this class needs to be manually changed.

✓ Use external files to define properties that can change the behaviour of your code. The format can be, in order of preference:

- YAML
- JSON
- Simple Java properties file
- XML

✓ Model your configuration as a class, and deserialize the file into it. Use *Jackson* library preferably, which is clean, well known and supports both JSON and YAML marshalling, or any other library for the same purpose that you like.

🔗 **Link:** An example of externalized configuration can be found here.

Auto detect configuration when possible

🚫 For settings that can be autodetected within Java code, **do not** request them as properties. If you need your code to behave differently depending if you are on Windows or Linux, have a helper class to figure it out. Do not request users to put *another* property in your configuration file. No one likes having to deal with a config file with 1000 parameters.

```java
public class Environment {

   private static final String OS = getProperty("os.name").toLowerCase();
   private static final String ARCH = getProperty("os.arch", "");

   private Environment() {}

   public static boolean isWindows() {
       return OS.contains("win");
   }

   public static boolean isMac() {
       return OS.contains("mac");
   }

   public static boolean isUnix() {
       return OS.contains("nix") || OS.contains("nux") || OS.contains("aix");
   }

   public static boolean is64Bits() {
       return ARCH.contains("64");
   }
}
```

Example for OS and architecture verification

Java Proxy Settings

Many times, a Java app needs to connect to the Internet. If you are having connectivity troubles because you are behind a corporate proxy, set the following JVM flags accordingly when starting your JVM on the command line. This is usually done in a shell script (in Unix) or bat file (in Windows), but you can also define them in your IDE settings for each test run configuration.

JVM variable	Example value	Type
`http.proxyHost`	proxy.corp.globant.com	String
`http.proxyPort`	3128	Integer
`http.nonProxyHosts`	localhost|127.0.0.1|10.*.*.*|*.foo.com	String

Or, if you have your system settings already configured, you can try:

JVM variable	Example value	Type
`java.net.useSystemProxies`	true	Boolean

JVM variables are defined prepending a **`-D`** when calling the java command from your system's terminal.

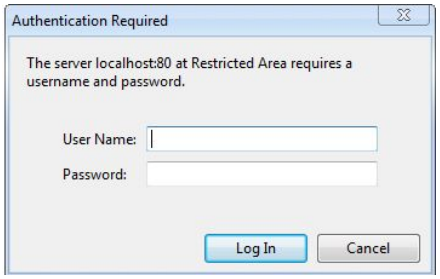
For example: `java -Djava.net.useSystemProxies=true some_file.jar`

You can also set them programmatically, like:

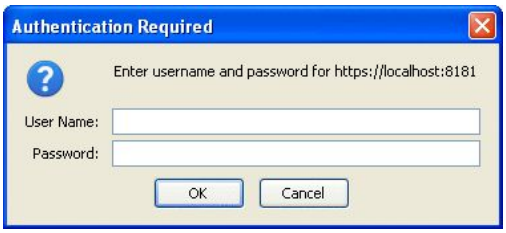
```
System.setProperty("java.net.useSystemProxies", "true");
```

Handling Authentication

If you haven't encounter this issue before, then have not been doing automated testing long enough.



If you've never seen these dialogs before, then you have not been using a computer long enough...or you still live in the 80's.



Dealing with authentication or security dialogs when testing a site or application can be tricky business.

These topics present some options you can use to deal with various kinds of authentication challenges.

There are several types of authentication mechanisms to secure web sites (Basic, Digest, NTLM and other types of hash based challenges). They all usually require the user to type in their credentials but the way in which those credentials are sent to the server differs. Then the browser takes care of re-sending them every time you request something from the same site, until you close the session or it expires. Credentials are not sent in plain text. Generally, they are sent as part of the HTTP headers. Since WebDriver does not know about HTTP headers, mangling them is impossible from WebDriver itself. Also, authentication dialogs are not the same as JS alerts, so Selenium can't handle/dismiss them...and even if you dismiss them, you won't gain access to the site you are supposed to test.

About usage of AutoIt to bypass browser modal dialogs

You will *never* hear good things from me regarding the usage of AutoIt during professional automation practices. That's it. I *do not* consider this a solution, and I don't care if it saved you once already in a previous automation project. Also, it's usage is restricted to Windows only.

> 🚫 **Do not** use AutoIt.

Bypassing Authentication

Basic HTTP Authentication via URL

You can *try* to send the credentials in the URL to provide a standard HTTP "Authorization" header. You supply a username and password in the URL, such as http://test:test@browserspy.dk/password-ok.php. And pray for your browser to support this feature. :)

Browser	HTTP Authentication Support
Chrome	Supported
Firefox	Supported, but Firefox will display a prompt asking you to confirm
Safari	Unsupported
Internet Explorer	Unsupported - https://support.microsoft.com/en-us/kb/834489

Because browser support for basic HTTP authentication is *limited* to *less* than 50% of major browsers, maybe you get lucky but chances are that this method is not going to help you bypass the Basic authentication of your test site.

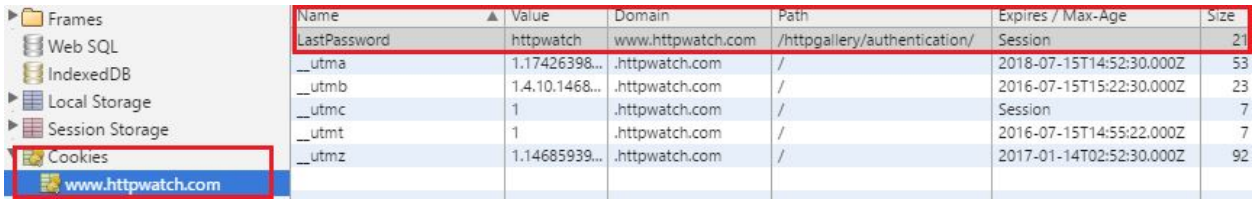
There are workarounds for *some* of the unsupported browsers, but they are so convoluted and flaky that it is kind of pointless to even suggest them as a real solution.

Cookies Injection

Another solution is to have WebDriver inject cookies that will let you bypass authentication by setting an authenticated state beforehand for the application or site you're testing.

1. Open your browser
2. Clear any pre-existing cookies that may be there (at least for the test site)
3. Navigate to the test site and open the dev tool (usually F12 key)
4. Locate the Resources/Cookies section.
5. Take a look at the cookies
6. Log into the test site
7. Take a look at the cookies

If the site accepts authentication via cookies you will see at least one new cookie in the list. Its name is something that the developers *usually* assign so there is no single name for a cookie that I can tell you about.



Name	Value	Domain	Path	Expires / Max-Age	Size
LastPassword	httpwatch	www.httpwatch.com	/httpgallery/authentication/	Session	21
__utma	1.17426398...	.httpwatch.com	/	2018-07-15T14:52:30.000Z	53
__utmb	1.4.10.1468...	.httpwatch.com	/	2016-07-15T15:22:30.000Z	23
__utmc	1	.httpwatch.com	/	Session	7
__utmt	1	.httpwatch.com	/	2016-07-15T14:55:22.000Z	7
__utmz	1.14685939...	.httpwatch.com	/	2017-01-14T02:52:30.000Z	92

Having this info, now you can try to inject the *same* cookie using WebDriver:

```
driver.manage().addCookie(new Cookie("name", "value", "path", new Date()));
```

This can also be tricky, since you *may* need to make a change in the source code of the application so that the cookie is acknowledged, but your tests will be able to run without the need for user authentication credentials.


You must be on the same domain that the cookie is valid for in order for this to work


You can try accessing a smaller page, like the 404 page, for the purpose of injecting the cookie before you access the home page

Authentication Proxy

So, this would the the preferred way. When implemented correctly, this solution works in 100% of cases for every platform. The main idea here is to set up an authentication proxy that will perform the credentials exchange on behalf of WebDriver, which has no way of doing so by itself, given that the HTTP client is embedded inside the Selenium bindings and we operate on them at a much higher level.

As stated before, there are different types of authentication mechanisms (Basic, Digest, NTLM, etc) so you will have to look for a proxy capable of handling your authentication scheme.

Here is a list of some proxies, mostly NTLM based:

- 🔑 NTLMAps
- 🔑 Cntlm
- 🔑 Java NTLM Proxy
- 🔑 Building you own ;)

Once you have decided on your proxy solution, the following steps are quite straightforward:

1. Run your proxy on the machine that will run the tests
2. Specify the WebDriver capabilities for proxy usage, and point it to the proxy you deployed
3. Run your tests and change/tune proxy settings till everything works as expected

After you have all this working, you can make as fancy as you want:

- 💡 If the proxy is written in the same programming language than your testing solution, you may want to embed it in your framework (if proxy code is Open Source, of course) and launch it in a separate thread

- 💡 You can have it in binary form as part of your project structure and execute as an external process from your test runner (before any test)

- 💡 Simply have it always deployed and running in all host machines where WebDriver test run

Test Scripts

- 🚫 **Do not** have a unique web browser instance to run *all* your test suites as it may *run out of memory* due to factors external to your design and implementation.

- 🚫 **Do not** perform many validations per tests. It is preferable to have 100 tests with 1 validation each than 1 test with 100 validations in it. Keep it simple.

- 🚫 **Do not** hardcode strings, numbers or any other kind of literals (known as “magic values”) inside the tests. Use constants and/or external files/sources to hold them.

- 🚫 **Do not** hard code dependencies on external accounts or data. Development and testing environments can change *significantly* in the time between the writing of your test scripts and when they run, especially if you have a standard set of tests that you run as part of your overall testing cycle. Instead, use API requests to dynamically provide the external inputs you need for your tests.

- 🚫 **Avoid** dependencies between tests. Dependencies between tests prevent tests from being able to run in parallel. Running tests in parallel is by far the best way to speed up the execution of your entire test suite. It's much easier to add a virtual machine than to try to figure out how to squeeze out another second of performance from a single test.

- ✓ Use the latest version of Selenium client bindings. The Selenium Project is always working to improve the functionality and performance of its client drivers for supported languages like Java, C#, Ruby, Python, and JavaScript, so you should always be using the latest version of the driver for your particular language.

- ✓ For web based authentication, set your credentials as environment variables that can be referenced from within your tests.

- ✓ Use Setup and Teardown. If there are are "prerequisite" tasks that need to be taken care of before your test runs, you should include a setup section in your script that executes them before the actual testing begins. For example, you may need to to log in to the application, or dismiss an introductory dialog that pops up before getting into the application functionality that you want to test. Similarly, if there are "post requisite" tasks that need to occur, like logging out, or terminating the remote session, you should have a teardown section that takes care of them for you.

- ✓ Create your web browser instance *once* per test class (suite), and destroy it after the suite is finished.

- ✓ Opening and closing a browser instance *per test* **is fine** too. *Slower*, but fine.

- ✓ Have the ability to tag the tests and/or suites to be able to run a sub set of the tests you have. Most current test runners supports this feature. Nobody likes to spend an hour waiting for a whole suite of unrelated tests to finish running just because they changed a link text.

- ✓ There is no rule saying that there should be a 1:1 relation between the manual test case steps and its automated version, if it keeps the tests cleaner / more readable, split a single manual test case into many automated tests.

- ✓ Tests should be synchronous and deterministic. This means, that:

    - ✓ Test should have a well defined beginning and end

    - ✓ Each and every step performed in the test script should translate to a reaction on the SUT

    - ✓ Test should be able to validate such reaction

  If this cannot be done, you will have to find a way to make it happen. This may be achievable by adjusting SUT configuration on demand and/or via mock objects. You can't wait for the SUT to actually deliver that email to your test account till midnight because there were network congestions or the scheduled job wasn't triggered till later on.

- ✓ Organize/structure your tests in a coherent manner. For example, group same web page related tests into the same class, so each class represents a suite for a particular web page.

- ⚠ You can use retry rules in your tests. If waiting patiently doesn't resolve your test flakiness, you can use JUnit `TestRule` class or TestNG `IRetryAnalyzer` interface. These will rerun tests that have failed without interrupting your test flow. However, you should **use these only as a last resort**, and very carefully, as rerunning failed tests can mask both flaky tests and flaky product features. Examples can be found here and here, along with their respective tests: here and here.

- ⚠ If you are developing a new test script and you start mumbling about *"how difficult is to do this!"* or *"Every time I create a new test script I have to do this, or declare that, or copy and paste this block of code"*. Stop right there, because you are probably right and there is something more to be done for the sake of simplifying the script development. Ask your TL or a co-worker for advice and/or insights.

Documentation

- ✓ Include a brief documentation for every test script. Mention its purpose and reference the corresponding manual test case.

Things to avoid when testing front end

- 🚫 **Do not** test for images, videos or any other *'multimedia'* content. You can guarantee that such element's *container* is there, has the correct size, tooltip, and maybe even that the src/href/url is the correct one. That's it. If you *really* need to check for something else (that would mean, if you have a *requirement* to do so), talk about it with your designated TL & client.

- 🚫 **Do not** test for CAPTCHAs. Third party components like CAPTCHA or TinyMCE can be extraordinarily difficult to automate. There's no sense in writing tests around third party components or services that have already been tested by that tool's creators.

- 🚫 **Do not** perform environment setup and teardown from the *UI*. If your tests need a certain user or permission to exist before or after running, then set those conditions directly against the backend. I don't care if your system has a nice UI console for user management. Create a backing infrastructure of helper classes and methods that call into appropriate web services, internal APIs or even directly to the database. Do not have slow and error prone UI based fixtures. If something goes wrong there, all dependant tests will fail/be skipped, it'll just take more time for someone to notice.

- 🚫 I'm within "*common sense*" ground here but, **do not** take the manual Test Case specification as if it were the Holy Bible. People can mess up, including QC Analysts. If the Test Case does not make sense and/or you figure there is a better way to test the same thing, let your team know about that. Have the QC update it.

- 🚫 Avoid creating automated tests around low-value features or features that are relatively stable. To have 1000 automated tests is not something good just 'because'. We'd rather have 100 **good** automated tests than 1000 tests that serves little purpose and take 90% of your time due to maintenance.

On prioritizing tests candidates for automation

- ✓ Focus your UI automation efforts on high-value features. Talk with your stakeholders and product owners. Find out what keeps them awake at night. I'll bet it's not whether or not the right shade of grey is applied to your contact form. I'll bet it's whether customer's orders are being billed correctly. **Automate tests around critical business value cases, not around shiny look-and-feel aspects of your application.**

- ✓ Decide what test cases to automate based on Return Of Investment (ROI). Formally, it is often described as $ROI = \frac{Gains - Investment\ Cost}{Investment\ Cost}$, but extrapolating this into a more practical meaning, you can read it as something between one or all of the following sentences:

  - 💡 "If the test case just needs to be run *once*"

  - 💡 "If you have to spend a lot of time and effort to develop a non-flaky test script when running it manually takes almost no time (example: validate that an image is correctly displayed)".

  - 💡 "The testing framework need major changes to support what the test would need to do and making those changes is out the scope right now"

  - 💡 "If the automation can be done quickly but its maintenance will have to keep going on due to 'X' circumstance"

  If any of these reasons applies to your current test case, just **don't automate it**, as its ROI will not be *convenient*.

- ✓ Manual tests that are *mandatory to exercise* and are *run often* (known as smoke tests) are the *best candidates* during the first stages of every project. Next in our priority list would be the sanity tests suite.

Issues related to parallel test executions

Flaky Tests with “Random” Failures

Symptoms

- 🚑 Tests “randomly” pass and fail each run, or throw transient exceptions
- 🚑 The number of unexpected failures is related to the number of concurrent threads running
- 🚑 When run in a single thread (not parallel), the tests pass or show expected behavior

Possible Root Causes

- √ Use of static shared members in test classes
- √ Use of static members in PageObject classes
- √ General resource contention on remote site due to issues with local test classes

Possible Fixes

- 🔧 Run your tests in a serial fashion and get a baseline
- 🔧 Repeat the process with increasing parallelism (increasing the number of concurrent threads) until you see consistent failures
- 🔧 Do a thorough review of static components in your classes and avoid using the static keyword where possible, and use ThreadLocal class members when needed
- 🔧 Look for resource contention and race conditions by printing resource references to standard output along with thread IDs: `Thread.currentThread().getId();`
- 🔧 Separate test framework from test content by reducing tests to simple actions with randomized data, and check for test thread isolation. For example, run a Selenium test where you enter randomized text input into a field, delay, and read back to confirm. If there’s interference from other tests the test will fail.
- 🔧 Add exception handling code around critical sections that present the errors. Going through your stack trace will greatly help with this.

Variable Test Performance and Performance Related Failures

Symptoms

- 🚨 Test execution times vary drastically from test run to test run
- 🚨 Unexpected failures are observed, followed by poor test performance
- 🚨 Performance improves with reduced thread counts
- 🚨 Tests fail due to timeouts
- 🚨 Test runner freezes and becomes unresponsive

Possible Root Causes

- √ Resource issues on test host
- √ Resource/capacity issues on test target
- √ Network infrastructure issues limiting bandwidth needed

Possible Fixes

- 🔧 Run tests with increasing parallelism (increasing the number of concurrent threads) and monitor resources like memory, CPU utilization, network utilization, and disk space (if applicable) on the test host. If any of these resources reach critical levels, adjust test volume accordingly or add resources.

    - ✓ For your thread count, you can start with “1.5 x # number of CPU cores” as a starting point, and experiment to find the number that would work for your setup.

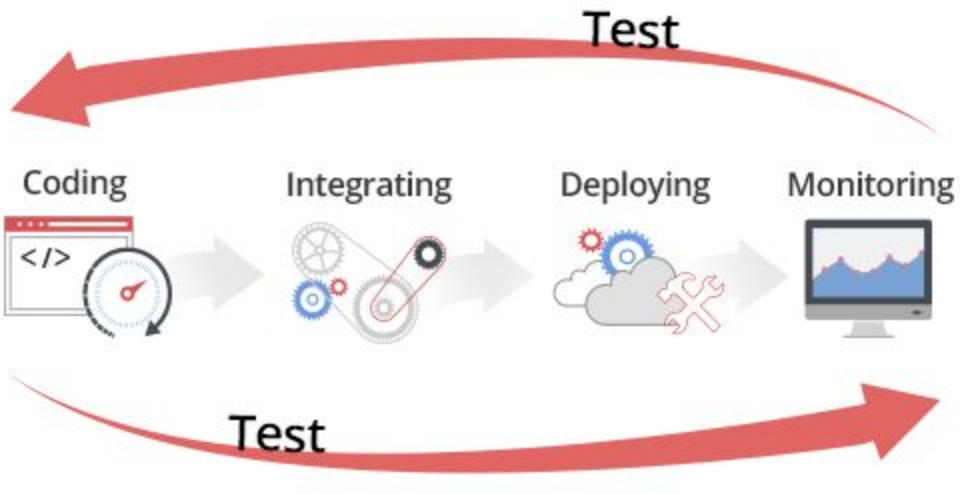
    - ✓ Augment memory allocation for your build process, depending on your needs. For example: `mvn -Xms256m -Xmx1024m <your build command>`

- 🔧 Monitor resources like memory, CPU utilization, network utilization, and disk space (if applicable) on the test target to make sure that the test target can support the traffic generated, and scale as needed.

- 🔧 Do a thorough review of static components in your classes and avoid using the static keyword where possible. These are not garbage collected, and they consume significant amounts of memory, even when not in use.

Automated Web Tests in Continuous Integration Servers

Continuous Integration servers are a specific kind of software that orchestrates the different steps (check out, compile, test, deploy, monitor, etc) during the lifecycle of software applications. It ensures that the steps required for software delivery are being executed in the correct order, metrics are gathered, quality checks are performed and reports on tests ran are archived and shown.



There are several CI servers in use today, most automation projects I've dealt with used tools like Jenkins, Teamcity and/or TFS.

Usually, development teams have no issues setting up a CI server, defining a job/task in it to check out and compile code and run the "unit" tests that they may have written. When done right, the process involving these steps just ranges from some seconds up to a a few minutes, depending on the size of the component(s) to build.

But this scenario changes radically as soons as their management/boss decides that they have to write/include *integration tests* as part of their build pipeline. As you may already know, integration tests are **way slower** that unit tests. A build that used to take a minute to complete now can potentially take hours, depending on the amount of tests and the conditions on which those tests are run. In agile environments, every piece of code a developer commits triggers a build cycle.

Personally, I can clearly see why this is bothersome for the development team, since fast feedback is a main pillar of agile methodologies. On the other hand, I think it is pointless, depressing and demoralizing when the work done by QA part of the team is disregarded and not included as part of the "official" build pipeline.

This means that there needs to be a line drawn somewhere.

There are a couple of “rules” that need to apply when dealing with automated tests (other than unit tests) on a CI server:

- ✓ **Create and maintain** a CI job *exclusively* for integration tests.

  The idea here is to run **all** automated tests. I know that it may take a long time. That is why the "exclusively" word refers to having an extra job/build *separated* from the main one (which is limited to running unit and component tests) that just runs the long life cycle tests.

  The fact that it gets triggered when a *master/trunk* branch is pushed *implies* that not anyone in the team should be able to push to it and, therefore, only after a merge that gets the code "ready for deployment" the tests are triggered.

  Some items that *must* be covered by this job are:

  - 🔀 Triggered upon one of the following situations:
    - 🚀 Master/trunk (ideally) branch commits
    - 🚀 Daily builds at fixed times (let’s say, for example, at 22:00 hs every day)
    - 🚀 You can *always* trigger it manually
  - 🔀 Email notifications are send to *every* team member upon failures. Optionally, message notifications can be send also to team’s chat system (ex: Slack)
  - 🔀 Reports that are easy to read/follow **must** be presented. CI servers tend to have several plugins to cover whatever file format your test runner of choice generates
  - 🔀 Archives of log and screenshots files **must** be kept, at least, for a week.
  - 🔀 Ideally, run the integration UI tests parallelly **and** against a Selenium Grid

⚠ Obviously, if your team does not use *master* branch as “ready for deployment” branch, run your tests upon commit on the one that they *do* use.

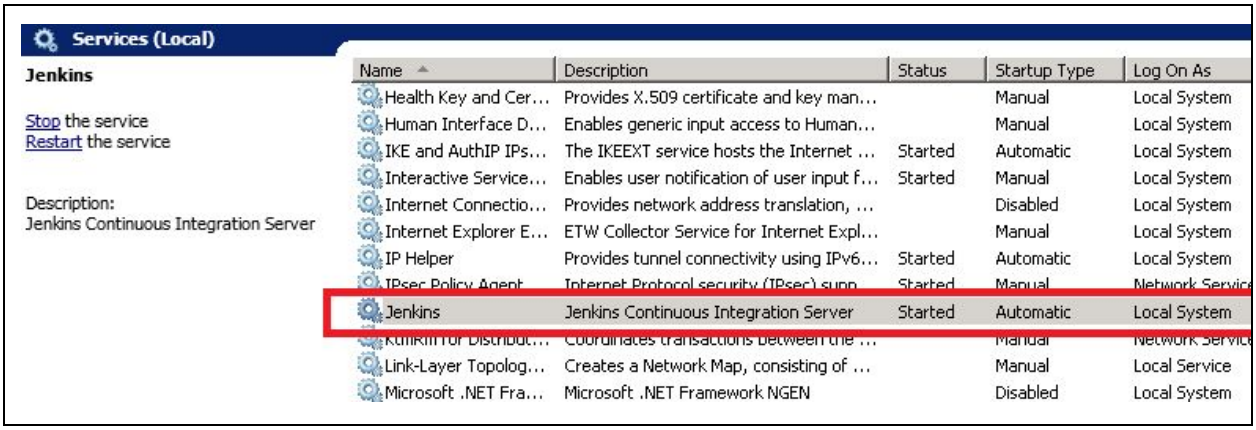
🚫 **Do not** let the team get used to CI server failures due to integration test runs. *Set a high priority on fixing failing tests instead of adding new ones.*

🚫 **Do not** commit tests that you know are flaky or unfinished into the build pipeline. This will only increase the distrust of people in the team about automation testing strategy.

Setting your CI server as a Windows service

If your CI server of choice does not work as a service by default, you can still convert it into a service manually.

You can change/install/remove Windows services safely using the Non Sucking Service Manager (NSSM).



Jenkins running as a Local System service

Problems running UI based tests from a CI server running as a service

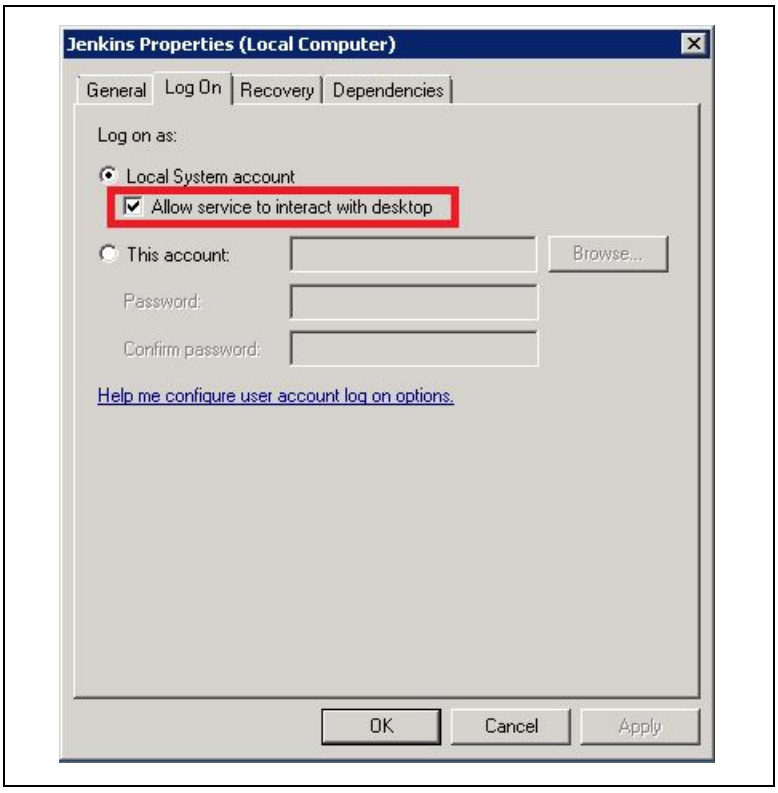
Non-console based windows applications (such as internet browsers) require access to the GUI to be able to perform their UI paints, open/close/handle windows, etc.

In Linux systems it is trivial to have a CI server running as a daemon/service and allocate a virtual framebuffer for it to use.

By default, Windows does not allow a service to interact against the UI. So I've seen people normally solve this issue by running Jenkins from command line under a specific user account on the CI machine or by attempting to install a VNC server. There should be no need for all that.

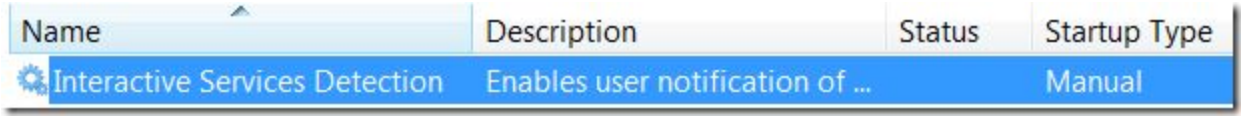
Microsoft provides a way for services to interact against a virtual GUI dedicated just for services.

1. Go to Start Menu → Run (or hit Win key + R) and type: services.msc <Enter>
2. Locate the CI server service and double click on it
3. On Log On tab, check the "Allow service to interact with desktop" box and click on Apply
4. On General tab, restart the service

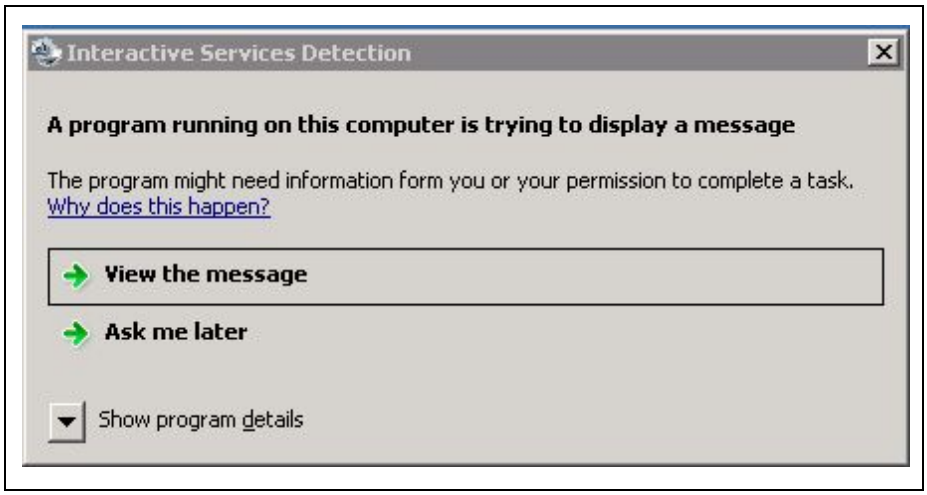


Allowing service to interact with a virtual desktop for UI testing

Also make sure the **"Interactive Services Detection"** service is STARTED. By default the startup type is set to Manual, you may like to set it to Automatic as well.



Every time there is “activity” in this virtual screen (as in a service is rendering something) , you should see a dialog informing that a program is running behind scenes rendering into the virtual screen. This is normal and expected.



Services interacting with the virtual desktop will raise a dialog like this one

Appium

Appium enables iOS and Android automation using Selenium WebDriver. The same WebDriver bindings can be used across web and mobile.

- Open Source: Apache 2.0 License
- Cross Platform
- Test Android on OS X, Windows, Linux
- Test iOS on OS X
- Native, Mobile Web or Hybrid (combination of both)
- Any language, any framework, any test runner.
- No app changes or source code access required.

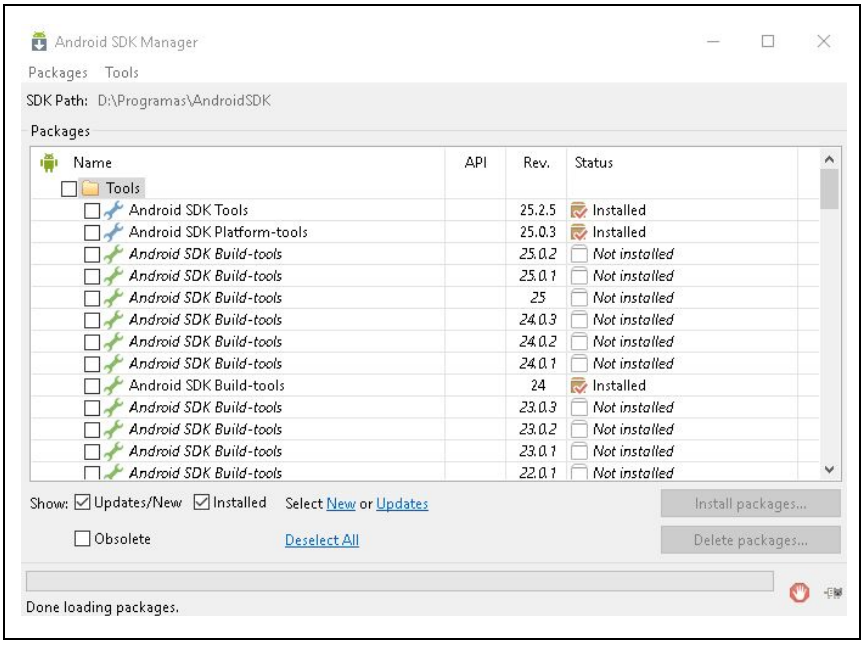
> ⚠ **Warning**
>
> At the moment, Appium does not support/handle multiple sessions with a *single server*. If you open multiple sessions against a single Appium server you will get an error message similar to this one:

```
info: [debug] Responding to client with error: {"status":33,"value":{"message":"A new session could not be created. (Original error: Requested a new session but one was in progress)","origValue":"Requested a new session but one was in progress"},"sessionId":"904d4e59-4c1e-4519-ad09-7a3d2b1edfaa"}
```

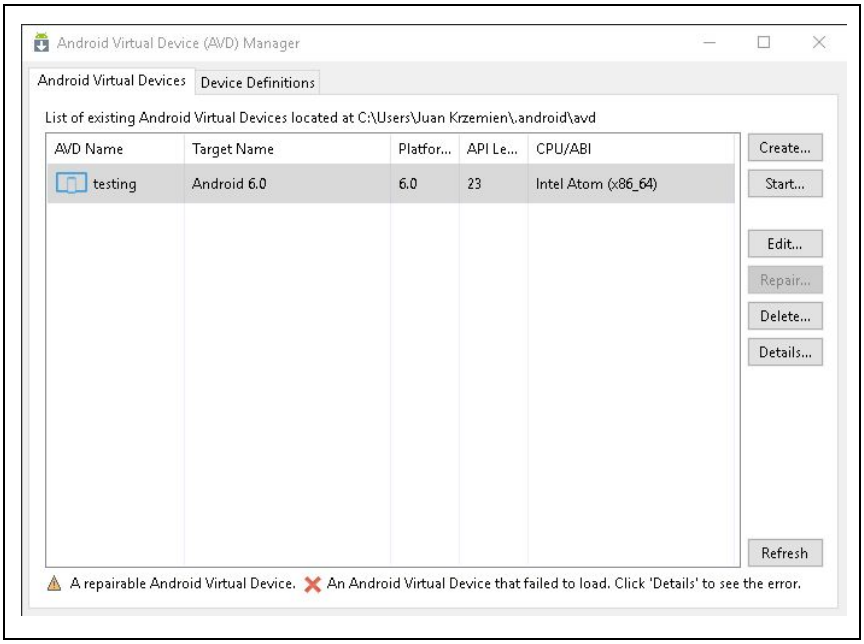
> Take this into consideration especially if you are planning on using Appium as part of a **Selenium Grid**. If you want to run two mobile tests *concurrently*, then you will need two Appium nodes.

From now on, you will need an Android or iOS device to play around with. In case you do not have a physical device, you can use an emulator. There is only one, as far as I know, for iOS. It is included with XCode IDE in Macintosh and since I do not have (nor want) a Mac, that's the end of the explanation, sorry.

You can use the **Android Virtual Device (AVD) Manager** (included in Android SDK Tools) to create as many different emulators for as many platform versions as needed.



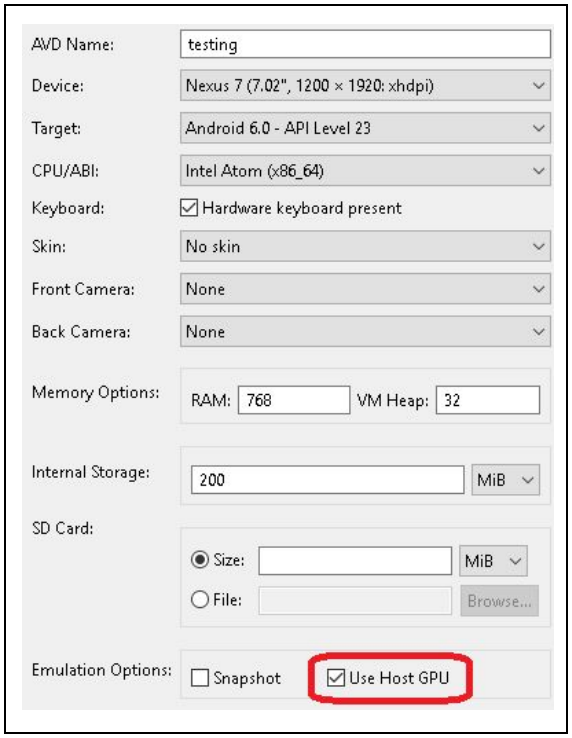
Android SDK Manager



Android Virtual Device (AVD) Manager

To achieve the best emulator performance you may want to consider:

- Installing Intel® Hardware Accelerated Execution Manager (Intel® HAXM)
- Enabling **Use Host GPU** option when creating/editing your emulator



- Keeping your **Memory Options** for **RAM** *under 2048 MB*

Another excellent alternative to AVD Manager is Genymotion. Register an account on their site and download the “free for personal use” version. It is easy to use and *really* fast. No extra magic needed.

Installing Appium

- Download Appium for your platform. And run the installer. Or, if you have npm already installed in your machine, you can just do:
  ```
  $ npm install -g appium
  $ appium
  ```

- Download the Android SDK command line tools (not Android Studio) for your platform.

    - After installing the Android SDK package you will have access to a tool named Android SDK Manager.

    - Use it to install Android build tools and any specific API that you may require. If you do not have a specific requirement, just leave it the default selections and click “Install X packages” button.

    - Set or define environment variable **ANDROID_HOME** to the folder where you installed the Android SDK.

    - Include in your **PATH** variable the route to *<Android SDK folder>/platform-tools*.

- You may need to close all consoles/terminals and start them again for changes to environment variables to take effect. Alternatively, you can also restart your computer.

Differences with WebDriver

Appium implementation for Java behaves similarly to Selenium one but it also brings to the table several types to ease the pain of dealing with mobile based elements. But there are some things that need to be taken into account exclusively while creating Appium based tests:

- Common `@FindBy` annotations present in Selenium implementation are effective against browser apps and web views. They *can* be used against native content, but it is useful to know that `By.css`, `By.link` and `By.partialLinkText` are *invalid* in that context.

- `@AndroidFindBy` annotation is designed to be used for Android native content description.

- `@iOSFindBy` annotation is used in iOS native content

- If it is necessary to use the same Page Object in desktop browser as well as cross platform mobile apps then it is possible to combine different annotations:

```
@FindBy(css = "someBrowserCss")
@iOSFindBy(uiAutomator = ".elements()[0]")
@AndroidFindBy(className = "android.widget.TextView")
public List<WebElement> androidOrIosTextViews;
```

- New interface appear to represent elements: `MobileElement`.

- In *"How do I initialize all elements defined in my POM?"* section we saw how to initialize WebElements inside Page Objects instances with Selenium. Appium provides a new `AppiumFieldDecorator` class that takes care of initializing its own MobileElements and other fields decorated by its own annotations. In this new context, POMs should be initialized now with a statement like:

```
PageFactory.initElements(new AppiumFieldDecorator(webDriverInstance), this);
```

Initialization of POMs - Appium style

Optionally, you can provide time out related arguments to its constructor.

Appium in Selenium Grid

Appium Node configuration

Appium has its own way of contacting the Selenium Grid's Hub node, so you don't have to start *another* selenium-server.jar process as a client node.

You can launch Appium server from command line in different ways:

```
appium --nodeconfig /path/to/nodeconfig.json
```

Or, if running from source:

```
node . --nodeconfig /path/to/nodeconfig.json
```

An example of a valid *nodeconfig.json* would be:

```
{
  "capabilities":[
    {
      "browserName":"Safari",
      "version":"7.1",
      "maxInstances":1,
      "platform":"MAC"
    },
    {
      "browserName":"Browser",
      "version":"4.4",
      "maxInstances":1,
      "platform":"ANDROID"
    }
  ],
  "configuration":{
    "cleanUpCycle":2000,
    "timeout":30000,
    "proxy":"org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
    "url":"http://<ip_or_hostname_of_machine_running_appium_server>:4723/wd/hub",
    "host":"ip_or_hostname_of_machine_running_appium_server",
    "port":4723,
    "maxSession":2,
    "register":true,
    "registerCycle":5000,
    "hubPort":4444,
    "hubHost":"10.136.104.99"
  }
}
```

If you are launching Appium from GUI, you can define the *nodeconfig.json* usage and location

from the "General Settings" button (a Gear icon).

- Check the box for "Selenium Grid configuration file"
- Define the full path to the *nodeconfig.json* file in the text box right below the "Selenium Grid configuration file" checkbox.
- Restart/Launch the Appium server.

More information on Appium and Selenium Grid can be found here.

Known Issues

⚠ Warning

Appium is an open-source tool and, as such, its has limitations and risks. For example, upon a failure during an iOS test run, Appium *can* screw up the MobileSafari application present in the iOS Simulator (comes with XCode), **literally deleting it**.
Posterior test runs against Mobile iOS will throw an error saying:

```
Cannot find MobileSafari under /User/Applications/XCode/..../MobileSafari.app
```

This is because of the way Appium instruments the application, moving it to a temp folder and restoring it after test runs, **if** there was no fatal error.

- 🔧 Solution 1: Have a backup of MobileSafari.app from where to restore it back to the location specified by Appium's error.
- 🔧 Solution 2: reinstall XCode application.

Tips & Tricks

How to enable Developer Mode in Android devices

1. Go to Settings in your device
2. Scroll down to '*About phone*' and tap on it.
3. Scroll down to the bottom again, where you see '*Build number*'.
4. Tap on '*Build number*' seven (7) times, and ***poof***, you've got the developer settings options.
5. Inside it, enable **USB Debugging** option. In newer versions of Android the option is called **Android debugging**.

If everything went well, you should get your device ID by typing the command '**`adb devices`**' in your console/shell:

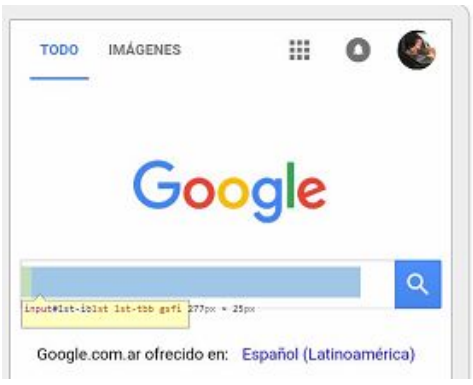
```
C:\Users\Juan Krzemien>adb devices
List of devices attached
e3619733        device
```

Using Chrome Remote Debugging to inspect web pages on devices

Once your device is recognized by ADB, in your *desktop Chrome browser* navigate to `chrome://inspect`, you should see it under the `Devices` section.

Now open the *default browser* in your device. You should see Chrome's Inspector update with the device's current tab info. Click on **Inspect** to see/debug it in real-time.

You can gather locators this way now.



Listing and downloading APK files from devices

1. Use `adb shell pm list packages` to determine the package name of the app, e.g. "com.example.someapp". Skip this step if you already know the package name. Look through the list of package names and try to find a match between the app in question and the package name. This is usually easy, but note that the package name can be completely unrelated to the app name. If you can't recognize the app from the list of package names, try finding the app in Google Play using a browser. The URL for an app in Google Play contains the package name.
2. Get the full path name of the APK file for the desired package with `adb shell pm path com.example.someapp`. The output will look something like: *package:/data/app/com.example.someapp.apk*.
3. Know that there is also a “shortcut” to list packages installed in the device along with its app name and path. There is even a filter to locate 3rd party (non Android OS related) packages also: `adb shell pm list packages -f -3`.
4. Pull the APK file from the Android device to your machine with `adb pull /data/app/com.example.someapp.apk`.

Listing activities and other useful information from APK files

Use `aapt dump badging <file.apk>` to determine the activity name (and many other properties) of the application to test. `aapt` command can be found inside `<ANDROID_HOME>\build-tools\<api version>\` folder.

```
package: name='com.android.browser' versionCode='23'
versionName='6.0-3079352' platformBuildVersionName='6.0-3079352'
sdkVersion:'23'
targetSdkVersion:'23'
original-package:'com.android.browser'
uses-permission: name='android.permission.ACCESS_COARSE_LOCATION'
...
application-label-en-GB:'Browser'
...
application-label-es-ES:'Navegador'
...
application-icon-160:'res/mipmap-mdpi-v4/ic_launcher_browser.png'
application-icon-240:'res/mipmap-hdpi-v4/ic_launcher_browser.png'
application-icon-320:'res/mipmap-xhdpi-v4/ic_launcher_browser.png'
application-icon-480:'res/mipmap-xxhdpi-v4/ic_launcher_browser.png'
application-icon-65535:'res/mipmap-xxhdpi-v4/ic_launcher_browser.png'
application: label='╪°┘ä┘à╪¬╪┘ü╪¡'
icon='res/mipmap-mdpi-v4/ic_launcher_browser.png'
launchable-activity: name='com.android.browser.BrowserActivity'
label='Browser' icon=''
uses-permission: name='android.permission.READ_EXTERNAL_STORAGE'
```

```
uses-implied-permission: name='android.permission.READ_EXTERNAL_STORAGE'
reason='requested WRITE_EXTERNAL_STORAGE'
feature-group: label=''
  uses-feature: name='android.hardware.camera'
  uses-implied-feature: name='android.hardware.camera' reason='requested
android.permission.CAMERA permission'
  ...
  uses-feature: name='android.hardware.wifi'
  uses-implied-feature: name='android.hardware.wifi' reason='requested
android.permission.ACCESS_WIFI_STATE permission'
provides-component:'app-widget'
provides-component:'search'
main
other-activities
other-receivers
other-services
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: 'ar' ... 'zh-TW'
densities: '160' '240' '320' '480' '65535'
```

aapt dump badging Browser.apk output example

Additional resources worth reading

- It is always useful to proof read the official Selenium HQ Wiki Pages. The real ones.