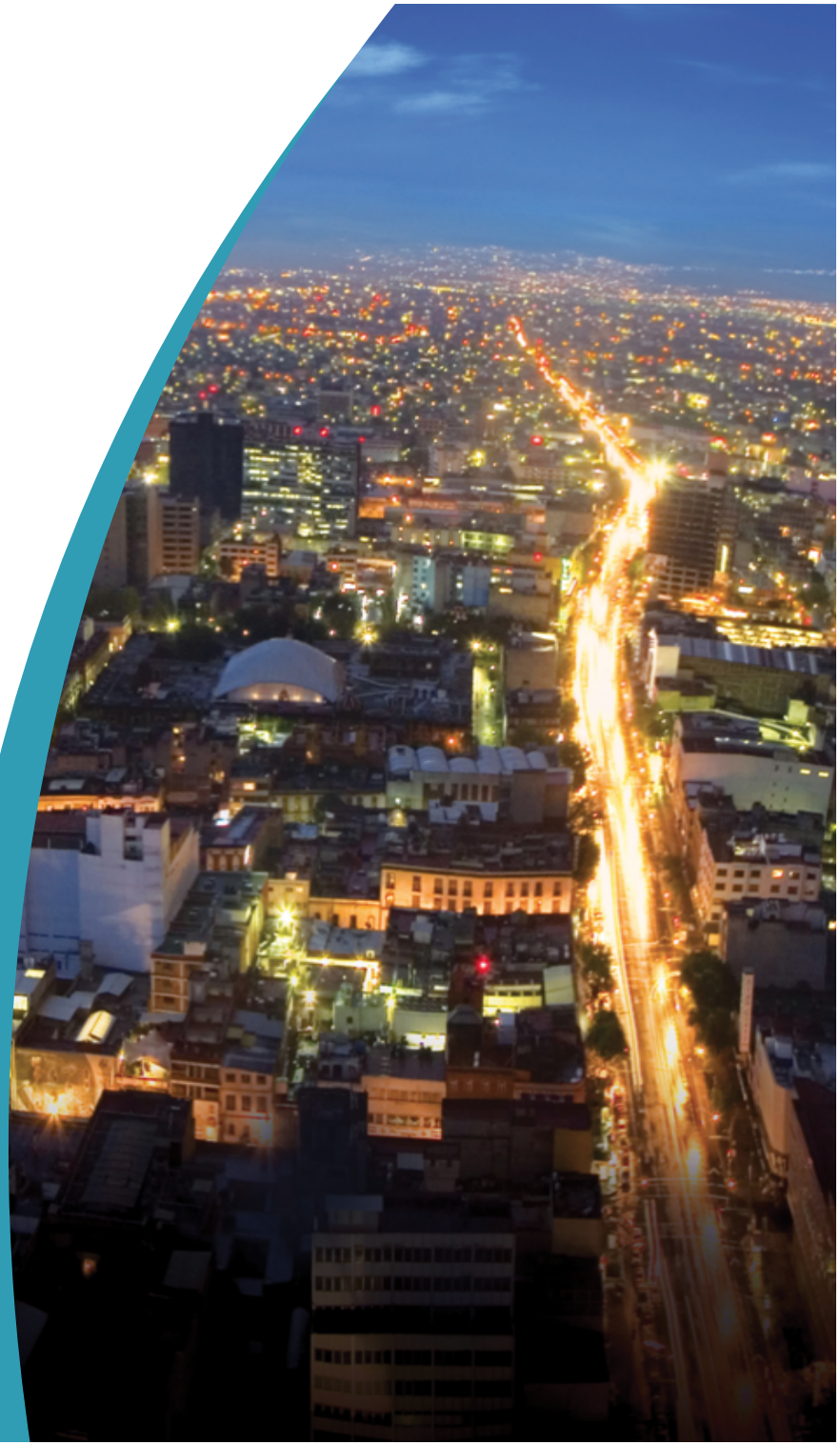
THALES

Selenium

Nicolas LETHELLIER



Why should I choose a web automation tool?

To reduce test costs

- If everything is automatic, it can be launched at any time, without any manual action

To remove human made errors

- Test campaigns can take time to run, with many repetitive tasks
- No added value

Need to focus on test results instead of test run action

(Quasi) immediate feedback for NR tests

- Tests results are pushed to Jenkins, or sthg else

Tests over several web browsers

- With different datasets

THALES

Selenium is ...

A web browser automation tool

- What the user can perform using its favourite web browser, Selenium does it quicker
- Supports a large bunch of web browsers

Can be ran standalone, or in client-server mode

Here is an example of Selenium running for an Orchestra tool

TCM-Selenium.mkv

Which tool should I choose ?

2 Selenium components are available

- Selenium IDE
- Selenium WebDriver

Depending on your testing purposes, you will have to choose which one is appropriate for you

Selenium IDE

What is it ?

- Firefox add-on : simple record-and-playback of interactions with the browser

Choose this component if you want to

- Create quick bug reproduction scripts
- Create scripts to aid in aitomation-aided exploratory testing
- Using self-recorded macros

- https://addons.mozilla.org/fr/firefox/addon/selenium-ide/

Selenium WebDriver

What is it ?

- Successor of Selenium RemoteControl, officially deprecated
- Robust engine suitable for automatic testing purposes

Choose this component if you want to

- Create robust, browser-based regression automation suites and tests
- Scale and distribute scripts across many environments

This tutorial will not deal about the Selenium IDE but the WebDriver

Source code

GitHub

Find all source code for this tutorial at this URL :

- https://github.com/Loustik/selenium-tuto

- https://www.alexecollins.com/

Summary

- Step01 : Create a sample web application (15 mn)
- Step02 : Add test dependencies (5 mn)
- Step03: Add a unit test sample (5 mn)
- Step 04 : A bit of integration tests with Selenium (5 mn)
- Step 05 : Inject a context with Spring (10 mn)
- Step 06 : Rearrange source code (10 mn)
- Step 07 : Page Object Pattern (10 mn)
- Step 08 : Make test fail (5 mn)
- Step 09 : Create users using POP (15 mn)
- Step 10 : Another direct method to interact (5 mn)
- Step 11 : Read results after form submission (10 mn)
- Step 12 : Let's Google that ☺ (20 mn)

- Going further… (5 mn)

Step 01

Create a sample web application

Step 01 – Create sample web application

Goal :

- Create a very simple web application
- View it in web browser
- Set a basic web app configuration
- Create 3 pages using JSP :
  - Index (default) -> some text
  - Error -> an error message
  - Create-user -> simple form to create a user
- Create a servlet
  - That wil forward all requests to « index »

Step 01 – Create sample web application

Install Maven on your machine

- apt-get install maven

Create a web application

- Use the maven-archetype-plugin
  - **Mvn archetype:generate**
    - –DgroupId=com.thalesgroup.ts.dojo
    - -DartifactId=selenium-sample-dojo
    - –DarchetypeType=maven-archetype-webapp
  - Choose filter n°38 (maven archetype-webapp) then version 1.0 (n°5)
- It will create a default pom.xml file
- It will create a Cargo container using Jetty for your web app
  - The cargo container is more suitable for this purpose that Tomcat, because it is possible tu launch it before running tests and to stop it afterward

Step 01 – Create sample web application

Run your Webapp

- Launch the Webapp executing the following command :
  - mvn cargo:run
- If there you have not defined plugin groups in your Maven config file, then you should execute this command instead :
  - mvn org.codehaus.cargo:cargo-maven2-plugin:run

Then open your browser and navigate to your web page

- http://<*IP_ADDRESS*>:8080/selenium-sample-tuto/
- You should see the default Maven Hello World page.

Step 01 – Create sample web application

Configure Cargo

- Configure it to launch a Jetty container before launching tests, and then to stop it afterwards. As a result, the tests suites could be executed as many times as needed

- Into the plugins section of the build part :

```
<plugin>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <version>1.6.4</version>
    <executions>
        <execution>
            <id>start</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>start</goal>
            </goals>
        </execution>
        <execution>
            <id>stop</id>
            <phase>post-integration-test</phase>
            <goals>
                <goal>stop</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <type>standalone</type>
        <configuration>
            <properties>
                <cargo.servlet.port>8081</cargo.servlet.por
            </properties>
        </configuration>
    </configuration>
</plugin>
```

Step 01 – Create sample web application

Create a Java Server Page

- Create a JSP file (index.jsp) to render web pages

```
<html>
<head>
  <meta charset="utf-8" />
  <title>Selenium sample tuto</title>
</head>
<body>
<h2>Hey, this is the default page !</h2>
  <div>
      <header>
          <nav class="menu">
              <a href="index.jsp" title="Home">Home</a> |
              <a href="create-user.jsp">Create user</a>
          </nav>
      </header>
      <p id='lorem'>Lorem Ipsum bla bla bla...</p>
      <p id='new_paragraph'>This is a brand new paragraph</p>
      <footer><small>&copy;Selenium makes life easier :)</small></footer>
  </div>
</body>
</html>
```

Step 01 – Create sample web application

Create a Java Server Page

- Create another JSP page (send-data.jsp) to use a basic HTML form

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
  <title>Send data using a HTML form</title>
</head>
<body>
  For instance, let's have a form to create dummy users :<br />
  <br />
  <form method="post" name="create-user-form">
    Firstname: <input name="firstname"/> <br/>
    Lastname: <input name="lastname"/><br/>
    Age: <input name="age"/><br/>
    <br />
    <input type="submit" value='Send all data'/>
  </form>
</body>
</html>
```

Step 01 – Create sample web application

Create a Servlet

- Create a Servlet that will handle all requests and redirect them to the JSP file

```java
public class WelcomeServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    getServletContext().getRequestDispatcher("/index.jsp").forward(request, response);
  }
}
```

Step 01 – Create sample web application

Add dependencies for servlet

- Pom.xml

```
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>servlet-api</artifactId>
        <version>2.5</version>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Step 01 – Create sample web application

Create page to handle errors

- error.jsp

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">


<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        <!-- SERVER_INTERNAL_ERROR -->
        <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
        <title>An error has occured on the server</title>
</head>
<body>
        Sample error text
</body>
</html>
```

Step 01 – Create sample web application

Configure WEB-INF/web.xml to redirect requests

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd" >
<web-app>
        <servlet>
                <servlet-name>WelcomeServlet</servlet-name>
                <servlet-class>com.nlethellier.sample.selenium.WelcomeServlet</servlet-class>
        </servlet>

        <servlet-mapping>
                <servlet-name>WelcomeServlet</servlet-name>
                <url-pattern>/</url-pattern>
        </servlet-mapping>

        <error-page>
                <exception-type>java.lang.Exception</exception-type>
                <location>/error.jsp</location>
        </error-page>
</web-app>
```

Step 01 – Create sample web application

Ensure it works

- Run the following command :
  - mvn verify

Step 01 – Create sample web application

Run your application

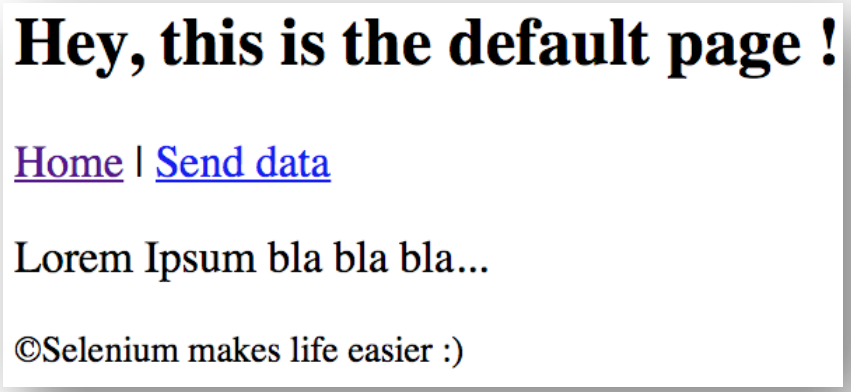
- Execute the following command
  - Mvn cargo:run

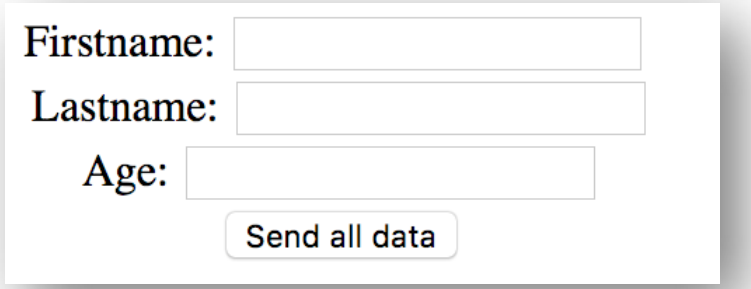
```
parallels@ubuntu:/media/psf/Home/GitHub/selenium-tuto/step01/selenium-sample-tuto$ mvn cargo:run
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building selenium-sample-tuto 1.0.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] --- cargo-maven2-plugin:1.2.0:run (default-cli) @ selenium-sample-tuto ---
[INFO] No container defined, using a default [jetty6x, embedded] container
[INFO] [en2.ContainerRunMojo] Resolved container artifact org.codehaus.cargo:cargo-core-container-jett
[INFO] [beddedLocalContainer] Jetty 6.x Embedded starting...
2017-02-09 07:08:53.321:INFO::Logging to STDERR via org.mortbay.log.StdErrLog
2017-02-09 07:08:53.373:INFO::jetty-6.1.26
2017-02-09 07:08:53.488:INFO::Extract /media/psf/Home/GitHub/selenium-tuto/step01/selenium-sample-tuto
_0_0_8081_selenium.sample.tuto.war__selenium.sample.tuto__t66vhw/webapp
2017-02-09 07:08:53.692:INFO::Extract /media/psf/Home/GitHub/selenium-tuto/step01/selenium-sample-tuto
to /tmp/Jetty_0_0_0_0_8081_cargocpc.war__cargocpc__.x4ckg/webapp
2017-02-09 07:08:53.754:INFO::Started SelectChannelConnector@0.0.0.0:8081
[INFO] [beddedLocalContainer] Jetty 6.x Embedded started on port [8081]
[INFO] Press Ctrl-C to stop the container...
^[
```

Step 01 – Create sample web application

Then open your browser and navigate to your web page

- http://<*IP_ADDRESS*>:8080/selenium-sample-tuto/





Step 02

Add test dependencies

Step 02 – Add test dependencies

Add test dependencies in the maven pom.xml file

- We will use junit for that tutorial

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
</dependency>
```

Unit tests versus Integration tests

Step 02 – Add test dependencies

Add the maven failsafe plugin

- Better than the Surfire maven plugin. Because at the first failure, Surfire halts the build but the container will keep running. Using failsafe, tests run until the end.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <version>2.12</version>
    <executions>
        <execution>
            <id>default</id>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Step 02 – Add test dependencies

Ensure the test is configured properly

- Run mvn test or mvn verify

Step 03

Sample unit test

Step 03 – Add a unit test sample

Create a basic class that « does something »

- **src/main/java**/com/nlethellier/sample/selenium……..

```
package com.nlethellier.sample.selenium;

public class MyMathApplication {

    public int sum(int numberOne, int numberTwo) {
        return numberOne + numberTwo;
    }
}
```

Step 03 – Add a unit test sample

Create a basic unit test

> **src/test/java**/com/nlethellier/sample/selenium

```
package com.nlethellier.sample.selenium;

import org.junit.Assert;
import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class DummyUTest {


    public DummyUTest() {
    }

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void testSumMethod() {
        MyMathApplication math = new MyMathApplication();
        Assert.assertEquals(7, math.sum(5, 2));
    }
}
```

Step 03 – Add a unit test sample

Run the basic unit test

- mvn test or mvn verify
- Ensure it is running (and I hope it won't fail, depending on your test result)

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.nlethellier.sample.selenium.DummyUnitTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.15 sec

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

Step 04

Integration test with Selenium

Step 04 – Integration tests : A bit of Selenium

Selenium

- It is an engine able to drive an application, and act as an end-user

Introducing GeckoDriver

- Starting version 47.0, Firefox's API has evolved. > Selenium driver 3.0.0 ++ compliance
- A wrapper have been published by Mozilla fundation to handle data exchange between Selenium and Firefox > GeckoDriver
- Copy the « binary » into an executable directory (available in the $Path)

GeckoDriver is a proxy for using W3C WebDriver-compatible clients to interact with Gecko-based web browsers (eg. FF)

Step 04 – Integration tests : A bit of Selenium

Here we are ! First of all, add Selenium dependencies

- It will be limited to the test scope

```
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-firefox-driver</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-support</artifactId>
    <version>${selenium.version}</version>
    <scope>test</scope>
</dependency>
```

- Then mvn verify → failure

Step 04 – Integration tests : A bit of Selenium

Selenium properties

- Did you notice that the selenium version is a variable? Well, this is Maven mechanisms that won't be explained here. Add the following property in the pom.xml file just the <build> tag

```
<properties>
    <selenium.version>3.6.0</selenium.version>
</properties>
```

- Yep. We will be able to use this mechanism for other purposes…
  - Browser type ? Browser version?
  - URL to be tested ?
  - And so many more.

Step 04 – Integration tests : A bit of Selenium

Add a simple selenium test

- We'll test it using the Firefox web browser

- Create test environment : the driver

```
public class FirstSeleniumIT extends TestCase{
    private URI website;
    private WebDriver driver;

    @Override
    protected void setUp() throws Exception {
        super.setUp();
        website = new URI("http://localhost:8081/selenium-sample-tuto-04");
        driver = new FirefoxDriver();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
        driver.close();
    }
```

Step 04 – Integration tests : A bit of Selenium

Create a very simple test

```
@Test
public void testWeSeeLoremIpsum() {
    driver.get(website.toString());
    Assert.assertTrue(driver.getPageSource().contains("Lorem Ipsum"));
}
```

Then run it

> mvn verify

Step 05

Inject a context using Spring

Step 05 – Spring context

Goal

- To avoid hard-coded values, we will inject values and dependencies using Spring.

Add Spring dependencies

- Add these dependencies in the pom.xml file

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
```

```
<properties>
    <selenium.version>3.6.0</selenium.version>
    <spring.version>4.1.7.RELEASE</spring.version>
</properties>
```

Step 05 – Spring context

Create dependency descriptor

- Create a file src/test/resources/applicationContext-test.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="siteBase" class="java.net.URI">
    <constructor-arg value="http://localhost:8081/${project.artifactId}/" />
  </bean>

  <bean id="drv" class="org.openqa.selenium.firefox.FirefoxDriver" destroy-method="quit"/>
</beans>
```

- The destroy method will ensure that every browser launched during tests will be shutdown after test complete.
  - Thus it is possible to remove the shutdown hook from the testcase ☺

Step 05 – Spring context

Now use the dependencies

- Update your files to use injected dependencies

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/applicationContext-test.xml" })
public class FirstSeleniumIT extends TestCase{

    @Autowired
    private URI website;

    @Autowired
    private WebDriver driver;

    @Before
    @Override
    public void setUp() throws Exception {
        super.setUp();
        getDriver().manage().deleteAllCookies();
        getDriver().get(getWebsite().toString());
    }
```

- Then generate getters and setters
- You can still view your app using mvn clean install cargo:run

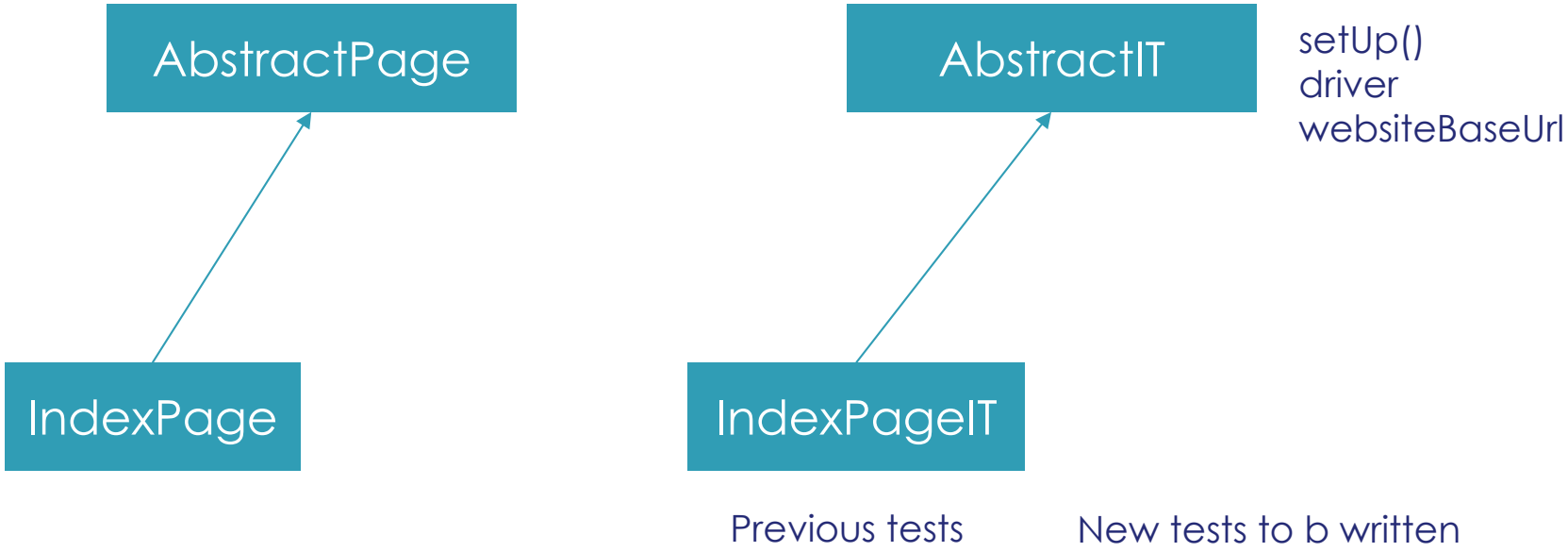
Step 06

Rearrange source code

Step 06 – Organize source code

Factor

- Add some abstract classes not to have common code duplicated in all java files you will create.



Step 06 – Organize source code

```
public abstract class AbstractPage {

    public static boolean isErrorPage(WebDriver drv) {
        return drv.getPageSource().contains("<!-- SERVER_INTERNAL_ERROR -->");
    }

}

public class IndexPage extends AbstractPage {

    /**
     * @param drv A web driver.
     * @param siteBase The root URI of a the expected site.
     * @return Whether or not the driver is at the index page of the site.
     */
    public static boolean isAtIndexPage(WebDriver drv, URI siteBase) {
        return drv.getCurrentUrl().equals(siteBase.toString());
    }

    private final WebDriver drv;
    private final URI siteBase;

    public IndexPage(WebDriver drv, URI siteBase) {
        if (!isAtIndexPage(drv, siteBase)) { throw new IllegalStateException(); }
        this.drv = drv;
        this.siteBase = siteBase;
    }
}
```

Step 07

Page Object Pattern

Step 07 – POP (Page Object Pattern)

Page Object Pattern

- Abstraction layer between the test object and the test scripts
  - Easy to maintain
  - Reusability of code
  - Better readability
  - Reduce code redundancy

PageFactory

- Java POP extension
- Used to initialize the elements of the Page Object design pattern a Page class without having to use 'FindElements' or instantiate the Page Objects itself
- Used to initialize the elements of a Page class without having to use 'FindElements'
- Possible to use @Annotations

Step 07 – POP (Page Object Pattern)

Enhance the code

- Add this element in your class' constructor (which has its own dependencies injected)

```
public IndexPage(WebDriver drv, URI siteBase) {
    if (!isAtIndexPage(drv, siteBase)) {
        throw new IllegalStateException();
    }

    this.drv = drv;
    this.siteBase = siteBase;

    PageFactory.initElements(drv, this);
}
```

- This call to PageFactory.initElements will populate fields annotated with the @FindBy annotation with the object matching the element on the web page

- It is now easier to find an element in the HTML code ☺

```
@FindBy(css = "a[href='create-user.jsp']")
private WebElement createUserLinkWE;
```

Step 07 – POP (Page Object Pattern)

Test using a PageObject

- IndexPageIT

```
private IndexPage indexPage;

@Before
public void init() {
    indexPage = new IndexPage(getDriver(), getWebsite());
}
```

```
@Test
public void testCreateUserLinkPresence() {
    Assert.assertNotNull(indexPage.getCreateUserLinkWE());
    Assert.assertEquals("a", indexPage.getCreateUserLinkWE().getTagName());
    Assert.assertEquals("Create user", indexPage.getCreateUserLinkWE().getText());
}
```

Step 08

Make test fail

Step 08 – Make test fail

Make test fail

- IndexPageIT

```
@Test
public void testThatShouldFail() {
    Assert.assertNotNull(indexPage.getSendDataLinkWE());
    Assert.assertEquals("This is not the link title", indexPage.getSendDataLinkWE().getText());
}
```

- See results into target/failsafe-reports

```
Tests run: 5, Failures: 1, Errors: 0, Skipped: 0
```

```
(default) on project selenium-sample-tuto-08: There are test failures.
```

```
[INFO] ------------------------------------------------------------------------
[INFO] BUILD FAILURE
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 35.526 s
[INFO] Finished at: 2017-10-18T13:42:08+02:00
[INFO] Final Memory: 20M/54M
[INFO] ------------------------------------------------------------------------
```

Step 09

Create users with POP

Step 09 – Use PageFactory to create users

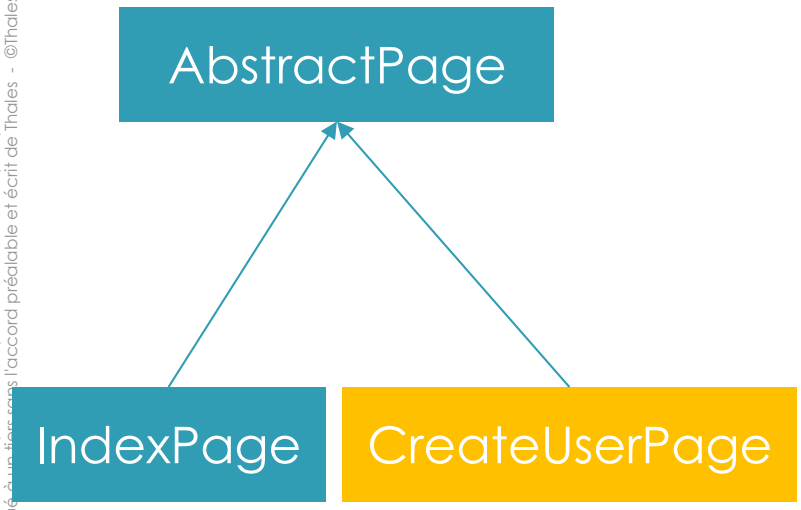
Java API for Selenium

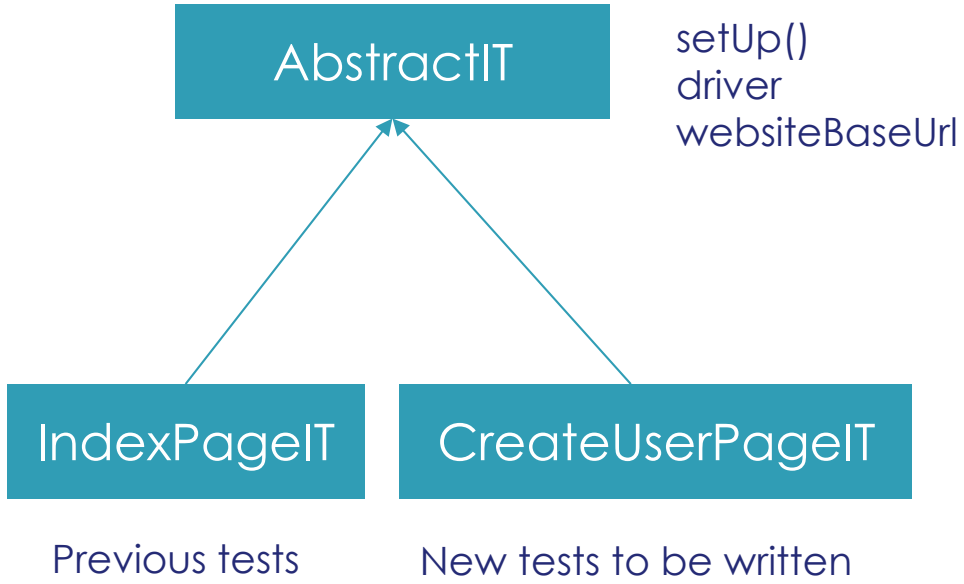
- Browse it at http://seleniumhq.github.io/selenium/docs/api/java/index.html

Web Driver documentation

- http://www.seleniumhq.org/docs/03_webdriver.jsp#locating-ui-elements-webelements

Write the CreateUserPage class





Step 09 – Use PageFactory to create users

```
public class CreateUserPage extends AbstractPage {

    /**
     * @param drv A web driver.
     * @param siteBase The root URI of a the expected site.
     * @return Whether or not the driver is at the page holding the form.
     */
    public static boolean isAtCreateUserPage(WebDriver drv, URI siteBase) {
        return drv.getCurrentUrl().equals(siteBase.toString() + "create-user.jsp");
    }

    private final WebDriver drv;
    private final URI siteBase;

    @FindBy(css = "form[name='create-user-form'] input[name='firstname']")
    private WebElement firstnameWE;

    @FindBy(css = "form[name='create-user-form'] input[name='lastname']")
    private WebElement lastnameWE;

    @FindBy(css = "form[name='create-user-form'] input[name='age']")
    private WebElement ageWE;

    @FindBy(css = "form[name='create-user-form'] input[type='submit']")
    private WebElement submitButtonWE;
```

Step 09 – Use PageFactory to create users

```
public WebElement getFirstnameWE() {
    return firstnameWE;
}

public WebElement getLastnameWE() {
    return lastnameWE;
}

public WebElement getAgeWE() {
    return ageWE;
}

public WebElement getSubmitButtonWE() {
    return submitButtonWE;
}

public CreateUserPage(WebDriver drv, URI siteBase) {
    if (!isAtCreateUserPage(drv, siteBase)) {
        throw new IllegalStateException();
    }

    this.drv = drv;
    this.siteBase = siteBase;

    PageFactory.initElements(drv, this);
}
```

Step 09 – Use PageFactory to create users

Navigate to the CreateUserPage

- Create a method on IndexPage which navigates to CreateUserPage

```
public CreateUserPage navigateToCreateUserPage() {
    getCreateUserLinkWE().click();
    return new CreateUserPage(drv, siteBase);
}
```

- Finally test the navigation feature
  - Write a test into IndexPageIT

```
@Test
public void testNavigateToSendDataPage() {
    indexPage.navigateToCreateUserPage();
    assertTrue(CreateUserPage.isAtCreateUserPage(getDriver(), getWebsiteBaseUrl()));
}
```

Step 09 – Use PageFactory to create users

Then create a test class to test for CreateUserPage

- CreateUserPageIT
- How to access this page ?
- Use the sendKeys() method to fill-in input values

Step 09 – Use PageFactory to create users

Then create a test class to test for CreateUserPage

```
public class CreateUserPageIT extends AbstractIT {

    private CreateUserPage createUserPage;

    @Before
    public void init() {
        createUserPage = new IndexPage(getDriver(), getWebsiteBaseUrl()).navigateToCreateUserPage();
    }

    @Test
    public void testFillInFirstName() {
        createUserPage.getFirstnameWE().sendKeys("penelope");
        Assert.assertEquals("penelope", createUserPage.getFirstnameWE().getAttribute("value"));
    }

    @Test
    public void testFillInLastName() {
        createUserPage.getLastnameWE().sendKeys("fillon");
        Assert.assertEquals("fillon", createUserPage.getLastnameWE().getAttribute("value"));
    }

    @Test
    public void testFillInAge() {
        createUserPage.getAgeWE().sendKeys("46");
        Assert.assertEquals("46", createUserPage.getAgeWE().getAttribute("value"));
    }

}
```

Step 10

Another method to interact with the web page

Step 10 – Another method to send items to the web interface

Then create a test class to test for CreateUserPage

- Use PageObject to create a bunch of actions to be performed

```
public CreateUserPage submit(String firstname, String lastname, String age) {
    firstnameWE.sendKeys(firstname);
    lastnameWE.sendKeys(lastname);
    ageWE.sendKeys(age);
    submitButtonWE.click();
    return new CreateUserPage(drv, siteBase);
}
```

- Then execute these actions from a test case

```
@Test
public void testSubmit() {
    createUserPage.submit("jean-pierre", "FOUCAULT", "12 ans");
}
```

Step 11

Read form results after form submission

Step 11 – Retrieving and tesing results of a form submission

Make the create-user-form send results to another page

- Give a target to the form

```
<form method="post" action="create-user-results" name="create-user-form.jsp">
```

- Set the data type to « text » for the 3 text fields

```
Firstname: <input type="text" name="firstname"/> <br/>
Lastname: <input type="text" name="lastname"/><br/>
Age: <input type="text" name="age"/><br/>
```

Step 11 – Retrieving and tesing results of a form submission

Make the create-user-form send results to another page

- Create the page that will receive data from the post method

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1" />
  <title>Receive data using a HTML form with POST method</title>
</head>
<body>
    <h1>A new user has been created !</h1>
    <br />
    <p name="result">Welcome <%= request.getParameter("firstname") %> <%=
    request.getParameter("lastname") %>. I assume you are <%=
    request.getParameter("age") %> years old.</p>
</body>
</html>
```

Step 11 – Retrieving and tesing results of a form submission

Create a dedicated PageObject class

```
public class CreateUserResultPage extends AbstractPage {

    /**
     * @param drv A web driver.
     * @param siteBase The root URI of a the expected site.
     * @return Whether or not the driver is at the page holding the result.
     */
    public static boolean isAtCreateUserResultPage(WebDriver drv, URI siteBase) {
        return drv.getCurrentUrl().equals(siteBase.toString() + "create-user-results.jsp");
    }

    private final WebDriver drv;
    private final URI siteBase;

    @FindBy(css = "p[name='result']")
    private WebElement resultSentenceWE;

    public WebElement getResultSentenceWE() {
        return resultSentenceWE;
    }

    public CreateUserResultPage(WebDriver drv, URI siteBase) {
        if (!isAtCreateUserResultPage(drv, siteBase)) {
            throw new IllegalStateException();
        }
        this.drv = drv;
        this.siteBase = siteBase;
        PageFactory.initElements(drv, this);
    }
```

Step 11 – Retrieving and tesing results of a form submission

Let us now create a test

- Test we are redirected to the relevant URL (result)

```
public class CreateUserResultPageIT extends AbstractIT {

    private CreateUserResultPage createUserResultPage;

    @Before
    public void init() {
        IndexPage indexPage = new IndexPage(getDriver(), getWebsiteBaseUrl());
        CreateUserPage createUserPage = indexPage.navigateToCreateUserPage();
        createUserResultPage = createUserPage.submit("Patrice", "CAINE", "41");
    }


    @Test
    public void testRedirectionUrl() {
        assertTrue(CreateUserResultPage.isAtCreateUserResultPage(getDriver(), getWebsiteBaseUrl()));
    }
```

Step 11 – Retrieving and tesing results of a form submission

Let us now create a test

- Test values after redirection

```
@Test
public void testValuesAfterRedirection() {
    String result = createUserResultPage.getResultSentenceWE().getText();
    Assert.assertThat(result, CoreMatchers.containsString("Patrice CAINE"));
    Assert.assertThat(result, CoreMatchers.containsString("41 years old"));
}
```

Step 12

Let’s Google that !

Step 12 – Let's Google that

Create a selenium procedure :

- Query Google with any word(s) you want
- Run search
- Browse the third link from the search results
- Ensure that the target page contains at least 10 times the keyword(s) you are looking for

Going further

Going further

- Another way to do that is to use parametrized variables
- Could be done via a script or via a Jenkins job for instance
- To do that, first create a config file

```java
/**
 * Main configuration for unit tests
 */
@SuppressWarnings({ "nls", "javadoc" })
public class Config {

  /** Whether to enable connection via SSL or not */
  public static final boolean __ENABLE_SSL = false;

  /**
   * Whether to enable the proxy settings to allow Fiddler to catch every HTTP
   * WARNING : if true, the tests can be very slow
   */
  public static final boolean __ENABLE_FIDDLER_DEBUG = false;

  /** Name of the project used to run the tests */
  public static final String __PROJECT_KEY = "COYOTE";

  /** Name of the basic project used to run the tests */
  public static final String __DEMO_PROJECT_KEY = "DEMO";

  /** Base type of the targeted base */
  public static BaseType __BASE_TYPE = BaseType.JIRA_6_PLUS;

  public static boolean __IS_JIRA_FROM_SCRATCH = false;

  /** Address of the URL base (used if {@link #__ENABLE_SSL} is false) */
  public static String __URL_BASE = "http://tptj61.thales:8080";
```

Going further

Then make your tests act as different services

```
public class ServiceClient {

  private LoginService _loginServices;
  private IssueService _issueServices;
  private FieldService _fieldServices;
  private ApplicationInfoService _appInfoServices;
  private DashboardService _dashboardServices;
  private FilterService _filterServices;
  private DashboardChartService _dashboardChartServices;
  private JiraAgileChartService _jiraAgileChartServices;
  private ElementVersionService _elementVersionServices;
  private CategoryVersionService _categoryVersionServices;
  private ElementItemService _elementItemServices;
  private AdminService _adminServices;


  public CategoryVersionService getCategoryVersionServices() {
    if (null == _categoryVersionServices) {
      _categoryVersionServices = new CategoryVersionService(getLoginServices());
    }
    return _categoryVersionServices;
  }
```

Going further

Create test suites that will launch test cases

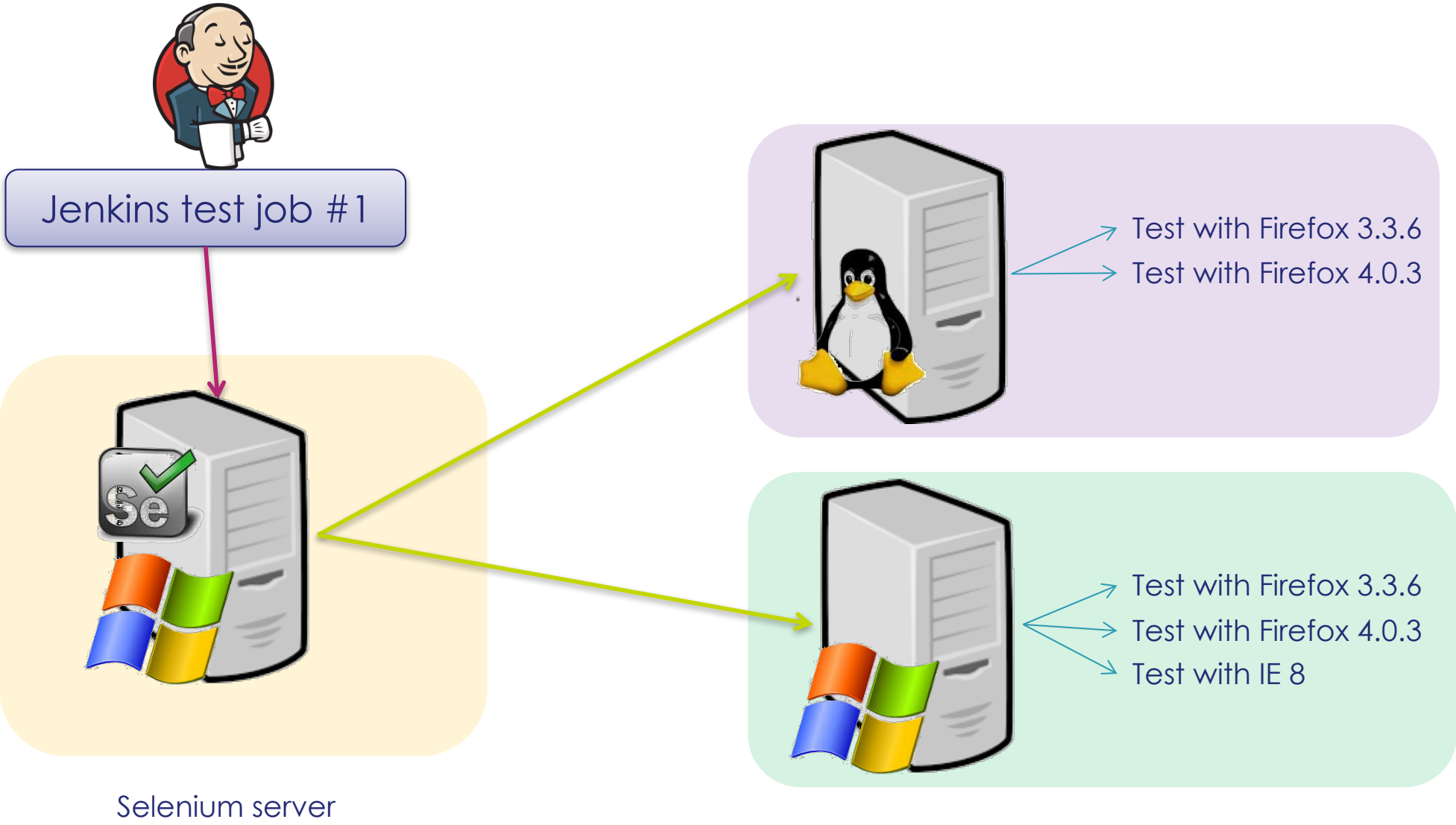
```java
/**
 * Runs all administrations tests
 */
@RunWith(Suite.class)
@SuiteClasses(value = { ClientLoginTest.class, AttachmentTest.class,
                AdminServicesClientTest.class })

@SuppressWarnings("javadoc")
public class MyTestSuite {

  @BeforeClass
  public static void setUp() {
    Config.__BASE_TYPE = BaseType.URL_server_1;
    Config.__JIRA_BASE = "http://tptj51.thales:8080";
  }
}
```

Going further

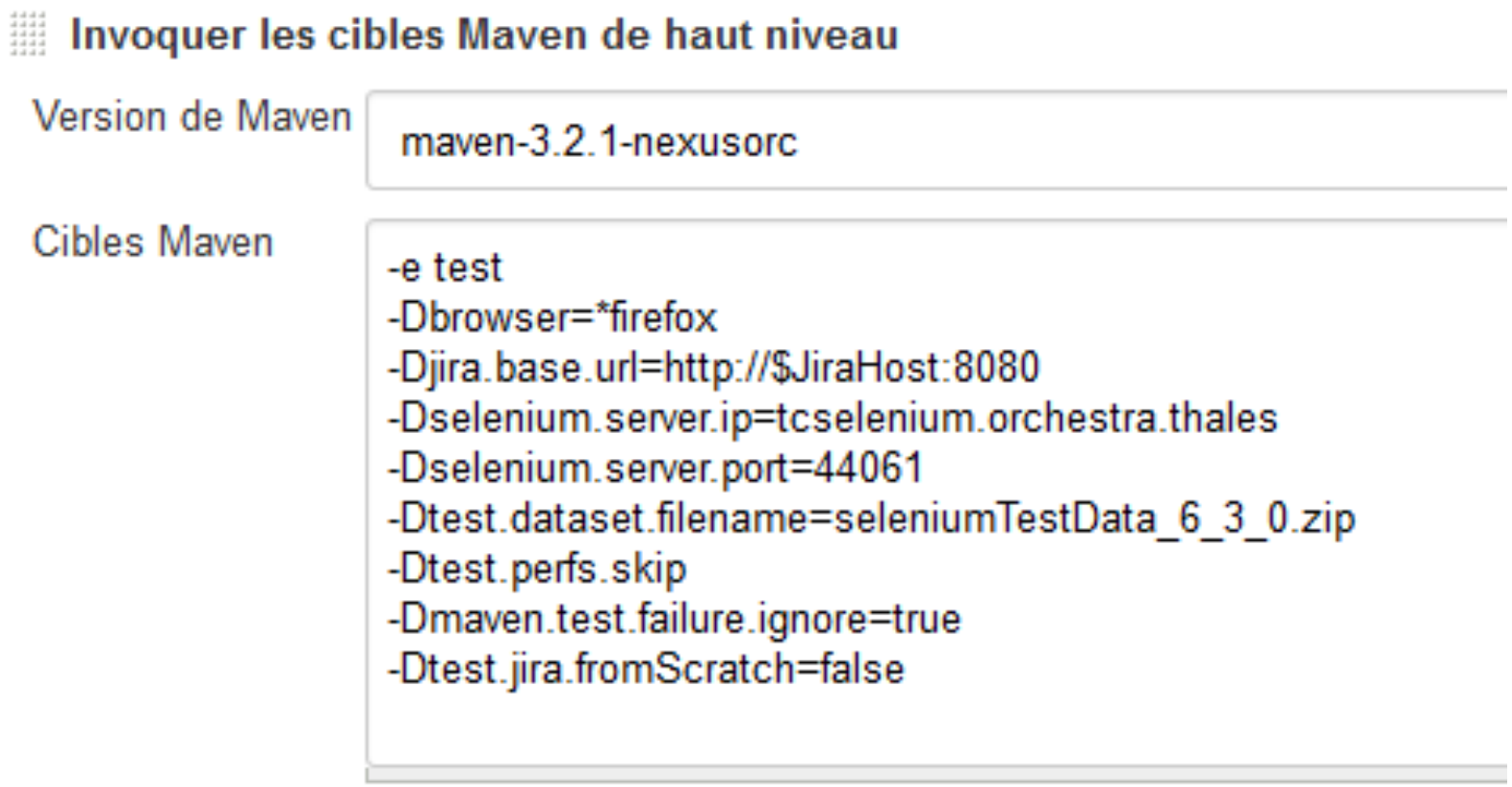
Client-server mode



Going further

Here is an example of a Jenkins configuration

- Ability to pass parameters via Java VM



Going further

Fiddler

- It is a free web debugging proxy
- Activate it to Selenium to make your code easier to debug
- http://www.telerik.com/fiddler