**CLASE 4**

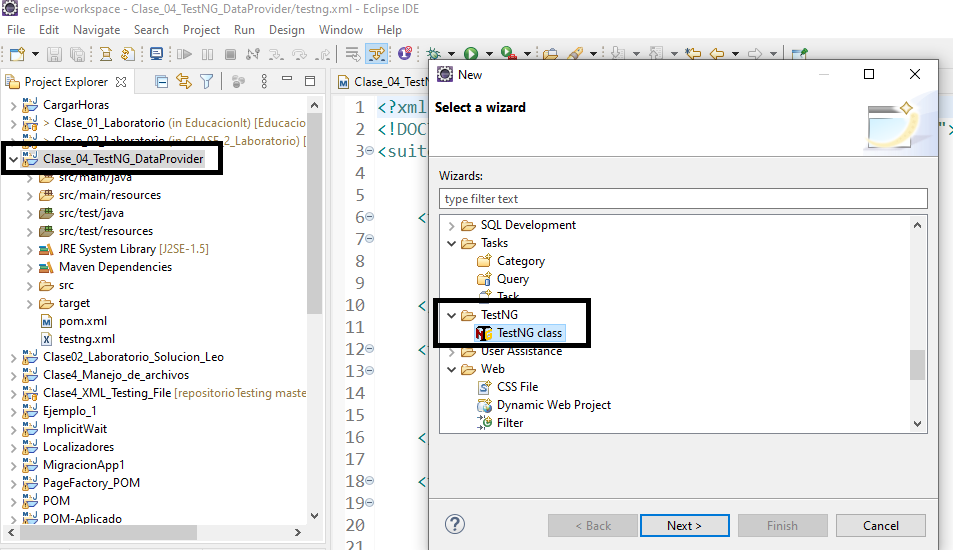
**TestNG: @DataProvider** (Pagina 2 a 5)

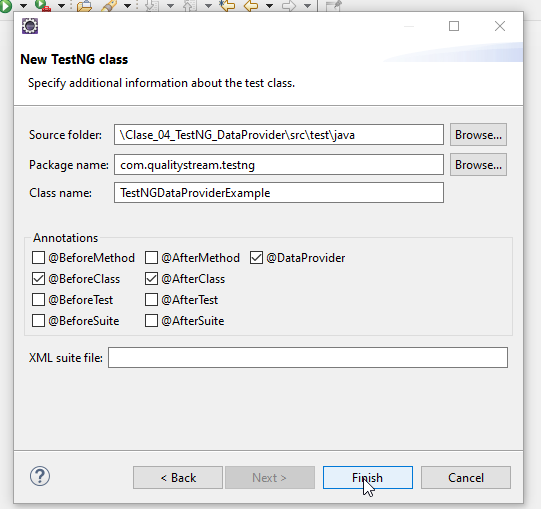
Codigo: **CLASE\_04\_DATA\_PROVIDER**



Ahora lo que vamos a hacer es en base al registro que hicimos en el código **TESTNG – Clase TestB** nos loguearemos con 2 usuarios diferentes.

1. Creamos una TestNG Class:





Utilizamos el siguiente código:

**Run as/Debug as > TestNG Test**

**Manejo de Archivos**

(Paginas 6 a 11):

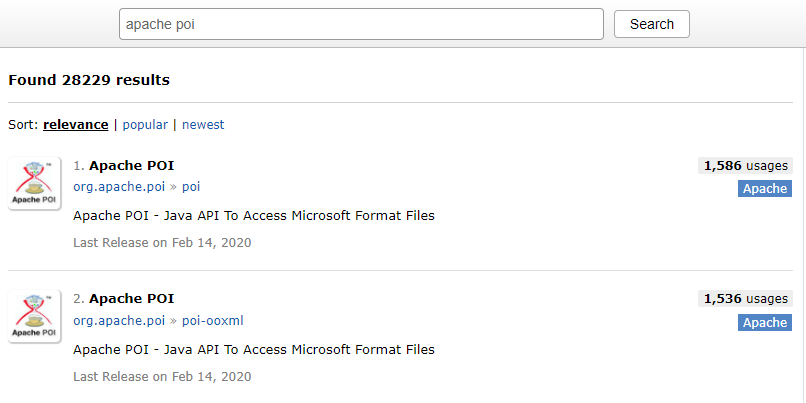
**Configuraciones Previas:**

1. **File.**

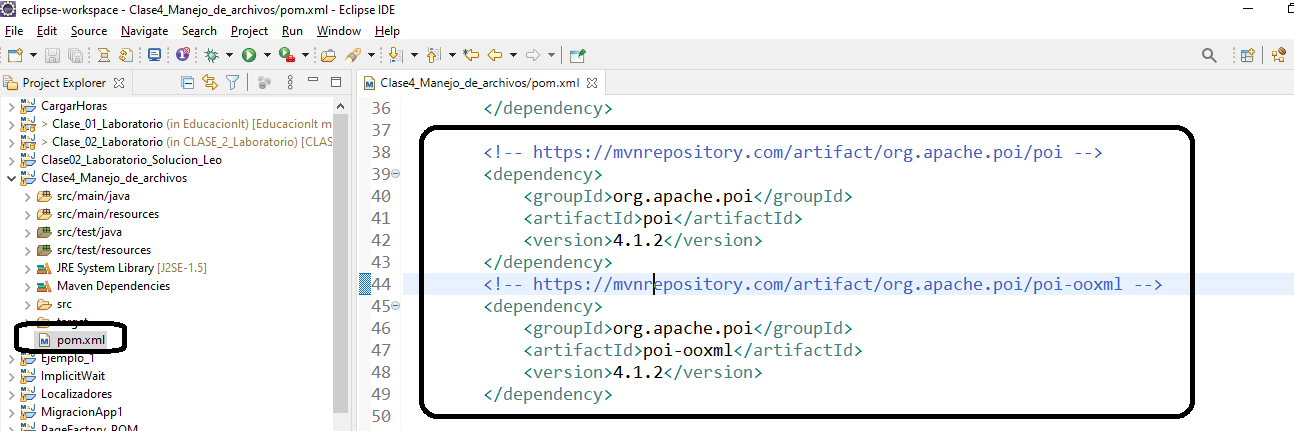
File file = new File(“rutaarchivo”);

InputStream inputStream = new FileInputStream(file);

1. Incluimos la librería de ApachePOI en el pom.xml.



La primera trabaja con las versiones de Excel 97 a 2003. La segunda con versiones actuales de Excel (XLSX). El pom nos debe quedar de esta manera:



**Nota**: Recordar también incluir las dependencias de Maven-surfire-report, Maven-plugin, selenium-java, Maven-compiler-plugin. Y también subir el plugin de Junit en caso de correr los test con Junit. Como hacemos en todos los proyectos.

1. **XLSX.**

XSSFWorkbook

XSSFSheet

XSSFRow

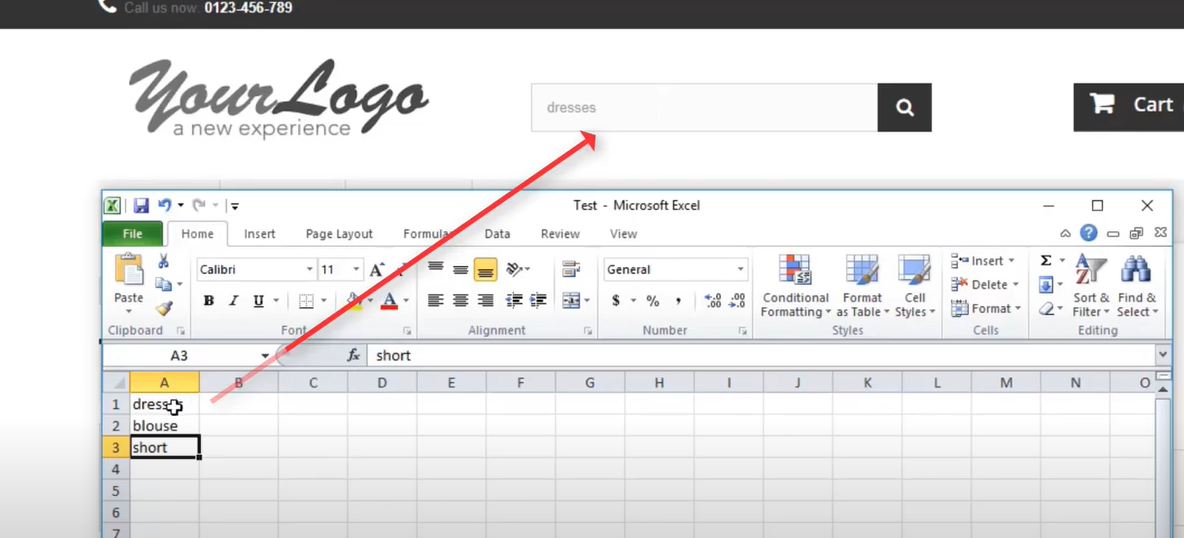
XSSFCell

Codigo: **CLASE\_04\_MANEJO\_DE\_ARCHIVOS (Junit)**

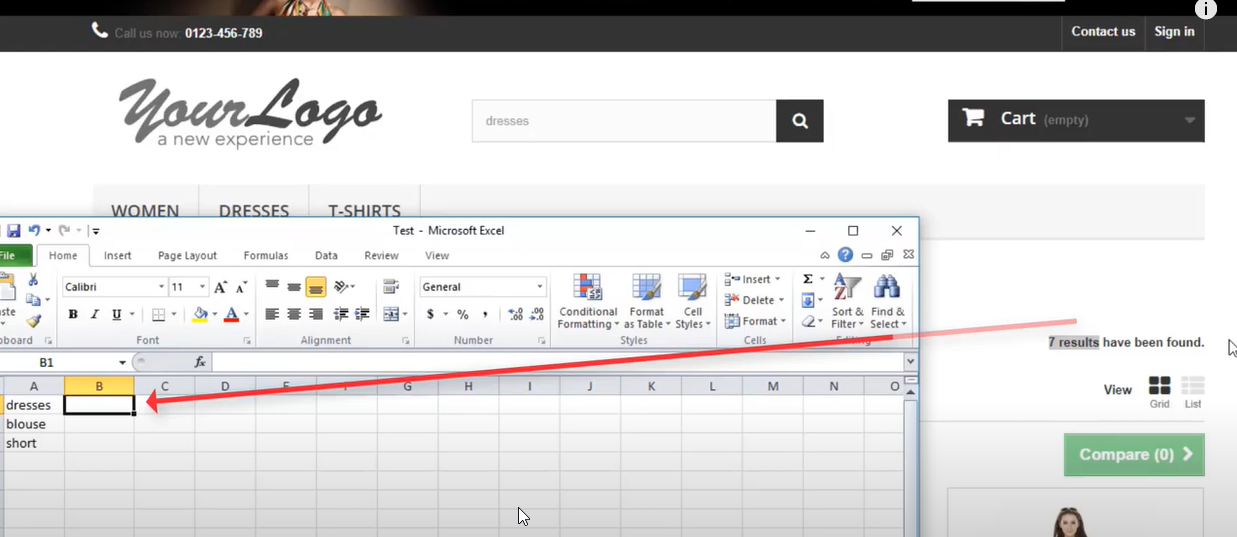


**Prueba:** Consiste en ir a <http://automationpractice.com/index.php> ir a la caja de búsqueda, buscar por la categoría de dresses, damos buscar (nos va a devolver todos los vestidos) y vamos a chequear que aparezca el mensaje “7 results have been found”.

Entonces vamos a crear un Excel que tenga lo siguiente:



Estos términos son los que vamos a ir colocando en la caja de búsqueda. Y luego vamos a ir escribiendo en la columna B el texto que vamos encontrando “7 results have been found”



Creamos un Junit Test Case llamado *DataDriverTesting\_SWD\_Test.*

Lo corremos como Junit Test y vemos como escribe en la columna B el resultado de lo encontrado (en el archivo Excel ubicado en C:\Users\lrlopez\Desktop\TESTING AUTOMATION CON SELENIUM\CLASE 4\Test.xls)

**Debug as/Run As => JUNIT Test**

1. **XML:** Probamos sin realizar test la funcionalidad de lectura de XML

Codigo: **CLASE\_04\_XML\_TESTING\_FILE,** Package**: “a.Pruebas”**



DocumentBuilderFactory

DocumentBuilder

Document

NodeList

Node

Creamos un archivo xml. Imprimimos por consola el contenido del tag APP\_URL que se encuentra en el archivo data.xml del proyecto. En este caso lo ejecutamos como una aplicación Java normal porque no necesitamos entrar a alguna pagina.

Debug As /Run As => Java aplication

1. Partiendo del codigo anterior agregamos algunas mejoras al mismo (paquetes b, c, d) con ejemplos sobre el tratamiento de xml.

Creamos un nuevo xml **data1.xml** y lo completamos con los datos del POM.

<project xmlns=*"http://maven.apache.org/POM/4.0.0"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>Clase4\_XML\_Testing\_File</groupId>

<artifactId>Clase4\_XML\_Testing\_File</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>Clase4\_XML\_Testing\_File</name>

<description>Clase4\_XML\_Testing\_File</description>

<dependencies>

<dependency>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.seleniumhq.selenium/selenium-java -->

<dependency>

<groupId>org.seleniumhq.selenium</groupId>

<artifactId>selenium-java</artifactId>

<version>3.141.59</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.maven/maven-plugin-api -->

<dependency>

<groupId>org.apache.maven</groupId>

<artifactId>maven-plugin-api</artifactId>

<version>3.6.3</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.maven.plugins/maven-surefire-report-plugin -->

<dependency>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-report-plugin</artifactId>

<version>2.18.1</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.poi/poi -->

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi</artifactId>

<version>4.1.2</version>

</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.poi/poi-ooxml -->

<dependency>

<groupId>org.apache.poi</groupId>

<artifactId>poi-ooxml</artifactId>

<version>4.1.2</version>

</dependency>

</dependencies>

</project>

Entonces si queremos obtener el texto asociado al tag **description** hacemos (Ver Package: **“b.LecturaXml2”**):

//Obtenemos el texto asociado al tag description

System.out.println("URL: "+ element.getElementsByTagName("description").item(0).getTextContent());

Y una forma de obtener todos los tags con su respectivo texto hacemos (Ver package “**c.Alternativa\_lectura\_XML”**). Tambien lo que permite hacer este codigo es si un tag tiene otros tags muestra el nodo en general y despues muestra cada uno de sus nodos nuevamente:

package Pruebas;

import java.io.File;

import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;

import javax.xml.parsers.DocumentBuilderFactory;

import javax.xml.parsers.ParserConfigurationException;

import org.w3c.dom.Document;

import org.w3c.dom.Element;

import org.w3c.dom.Node;

import org.w3c.dom.NodeList;

import org.xml.sax.SAXException;

public class ReadXMLData1 {

public static void main(String[] args) throws ParserConfigurationException, SAXException, IOException {

File xmlFile = new File("..\\Clase4\_XML\_Testing\_File\\src\\test\\java\\Pruebas\\data1.xml");

DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();

DocumentBuilder dbuilder = dbFactory.newDocumentBuilder();

Document doc = dbuilder.parse(xmlFile);

NodeList n1 = doc.getDocumentElement().getChildNodes();

int cantNodos = n1.getLength();

for (int i = 0; i < cantNodos; i++) {

printTags((Node) n1.item(i));

}

}

private static void printTags(Node node) {

if(node.hasChildNodes() || node.getNodeType()!=3 ) {

System.out.println(node.getNodeName() + " : " + node.getTextContent());

NodeList n1 = node.getChildNodes();

for (int i = 0; i < n1.getLength(); i++) {

printTags(n1.item(i));

}

}

}

}

Y si queremos por ejemplo que nos muestre el nombre del tag y su contenido para un nodo en particular, ej para el nodo “dependencies” del POM (ver package **d.Tag\_Particular**).

**GIT y GITHUB:**

**Repositorio de proyecto**

Paginas (12 a 24)

1. **Git, Que es?** Pagina 12.

Es un Sistema de control de versiones. Controla y administra las distintas versiones de un programa a medida que un proyecto va creciendo en complejidad.

Fue creado por Linux Torvalds para administrar la complejidad del Kernel de Linux.

Con esta metodología todos los desarrolladores tendran una copia del código con las ultimas modificaciones. Teniendo la posibilidad de volver a versiones anteriores para revertir los cambios.

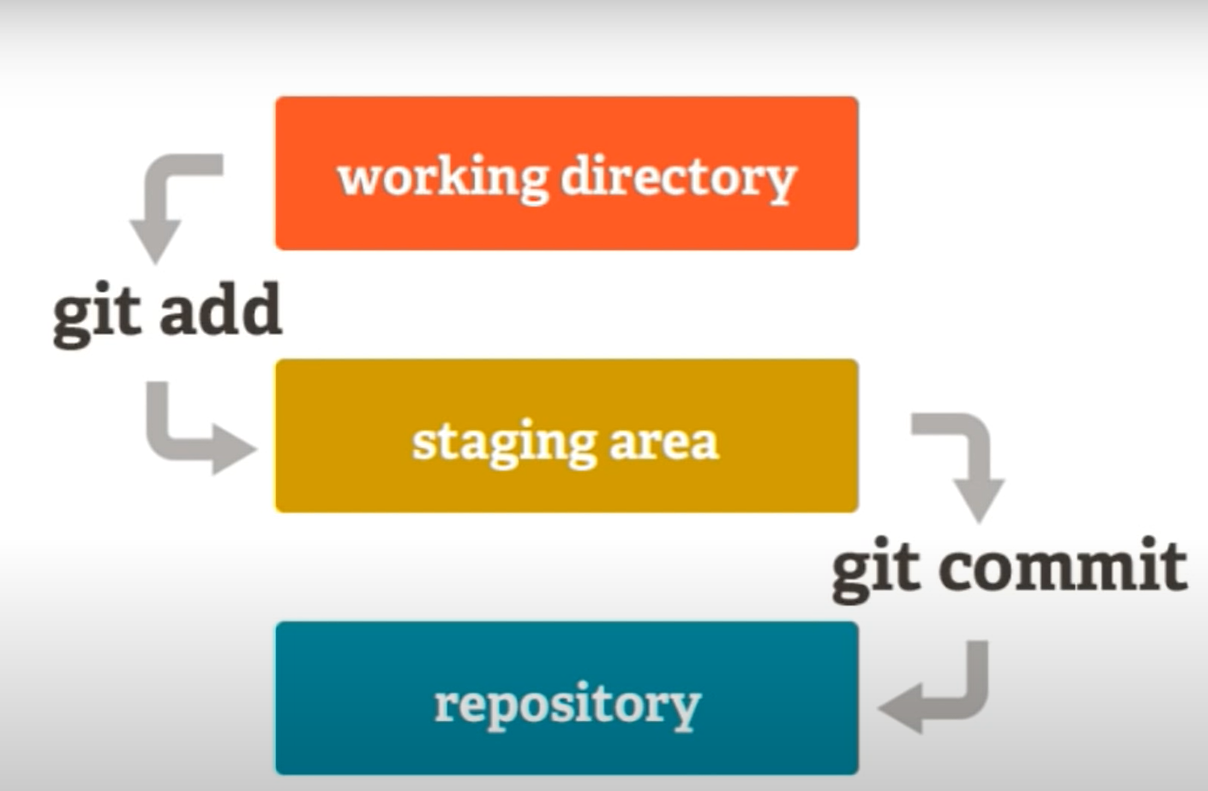
Otras características:

Se puede trabajar con repositorios locales y remotos.

*Snapshots:* Git toma fotos de nuestro código.

*Git* interactua con nuestro Sistema Operativo. Se maneja con el cmd y se instala una consola git.

**Estados** de Git:



*Working directory:* es donde estaremos trabajando con nuestros archivos.

*Staging Area:* es donde se agregaran los archivos que se prepararan para el guardado, cuando estemos a punto de crear una nueva versión.

*Repository:* Cuando se guarda la versión o el cambio realizado.

*Comandos:*

**git init**: Le decimos al SO o a git que crearemos un proyecto nuevo.

**git add<file>**: Pasar los archivos de working directory a staging área.

**git status**: Para saber en que estado están nuestros archivos (working directory , staging área)

**git commit**: Para pasar del staging área al repositorio, creando un primer snapshot de la versión del código.

**git push**: para subirlo a un repositorio remoto. Asi puede ser accedido por otros programadores.

**git pull**: Traer los cambios que han hecho los otros desarrolladores.

**git clone**: Hace una copia desde el servidor central (donde se encuentra el código fuente) a nuestro ambiente para poder trabajar.

Instalamos Git.

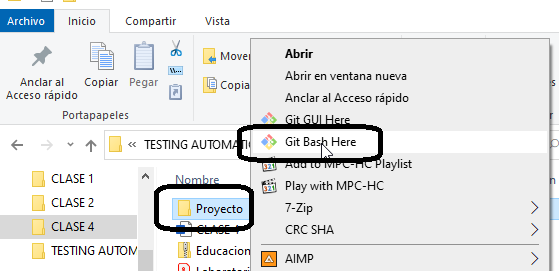


**Comenzamos con Git:**

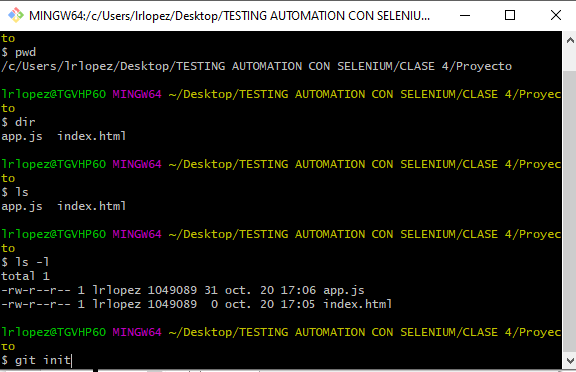
1. Crear una carpeta llamada “proyecto”

(la mia se encuentra en: C:\Users\lrlopez\Desktop\TESTING AUTOMATION CON SELENIUM\CLASE 4\Proyecto).

1. Arrastramos la carpeta a Sublime Text o Visual Studio Code y creamos un index.html y un app.js
2. Para continuar tenemos que **inicializar** nuestro **repositorio** de **código**.



git init desde la consola, para comenzar a trabajar en el working directory.



1. **GitHub, Que es?** Pagina 12.

Pagina oficial: https://github.com/



**Requisitos Git.** Pagina 13.

1. **Crear un Repositorio.** Pagina 14. Creamos un nuevo repositorio.

El repositorio de código es una plataforma web donde vas a poder subir todos los códigos de tus proyectos de manera privada o publica. En este caso como GitHub es openSource, solo se permite subir código de tipo OpenSource. O pagando se puede acceder y subir proyectos reales.

**Se sugiere crearse una cuenta y completar el registro.**

**COMO SUBIR NUESTRO CODIGO (generico):**

****

**COMO SUBIR NUESTRO CODIGO eclipse de PRUEBAS a GitHUB:**

****

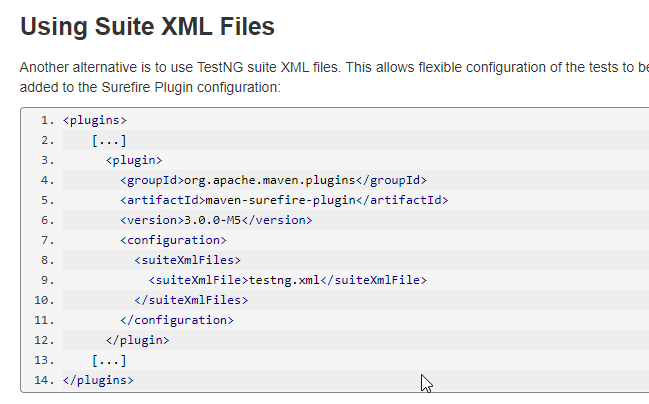
**Ejecutar proyectos via CMD:**

**TESTNG**

1. Abrimos un cmd y luego **java -version** para ver la versión de Java instalada.
2. **mvn -version** devuelve la versión de Maven instalada.
3. Como requisito es necesario tener el plugin de surfire instalado. El objetivo de este plugin es permitir la ejecución de un test con testNG, Junit, etc.

Para eso vamos a la siguiente pagina (para TestNg):

https://maven.apache.org/surefire/maven-surefire-plugin/examples/testng.html



Y lo colocamos en el pom.xml después de dependencias:



En el tag **suiteXmlFile** debemos indicar la ruta de nuestro testng.XML. En este caso como se desprende directamente de nuestro proyecto lo dejamos asi.

1. Abrimos el CMD y vamos a la locación de nuestro proyecto

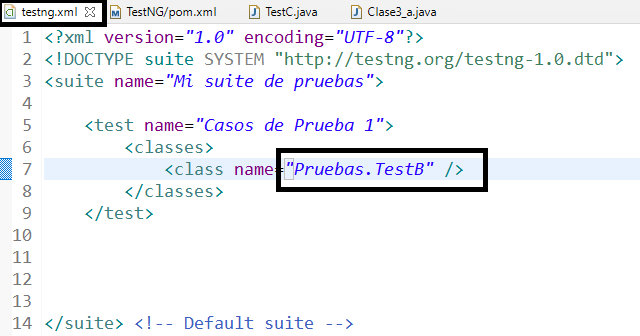
**cd C:\Users\lrlopez\eclipse-workspace\TestNG**.

1. Con **mvn package -Dtestng.dtd.http=true** creamos un JAR en nuestro repositorio local (TARGET/).

Lo que hace es buscar el xml que tiene seteado el POM, y ejecutarlo. En este caso el POM tiene seteado el suite testng.XML

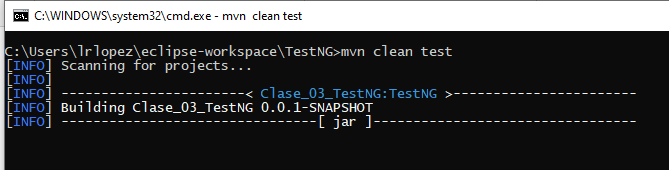


Y si vamos a ese xml vemos que se ejecutaran los casos de prueba del **TestB.**

****

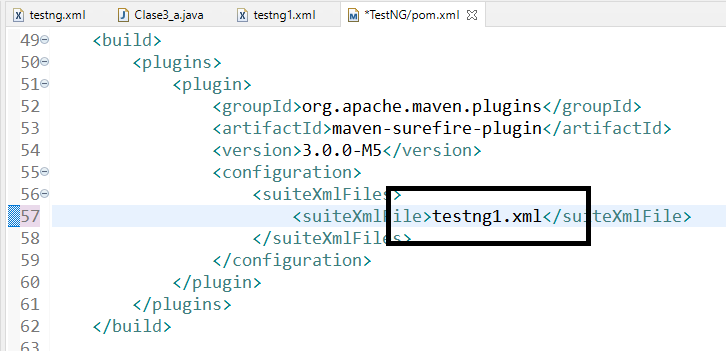
Podemos cambiarlo con las otras pruebas.

1. **mvn clean test -Dtestng.dtd.http=true** para hacer un clean del proyecto.



El agregado **-Dtestng.dtd.http=true** soluciona un error puntual.

1. Si queremos agregar mas de 1 Suite de pruebas lo podemos hacer. Entonces creamos un testng1.xml con otro set de pruebas. Y si queremos ejecutar el mismo podemos cambiarlo en el pom.xml



Y luego corremos normalmente con **mvn clean test -Dtestng.dtd.http=true** (que es la otra forma de correrlos)

1. Ahora si queremos ejecutar directamente un testng suite desde el cmd debemos hacer lo siguiente:
2. Colocar la siguiente referencia en el pom.xml => ${fileName} y desde la línea de comandos debemos agregar el siguiente parámetro -DfileName=testng1.xml.

**mvn clean test -DfileName=testng1.xml** (ejectuarlo en el paso b)

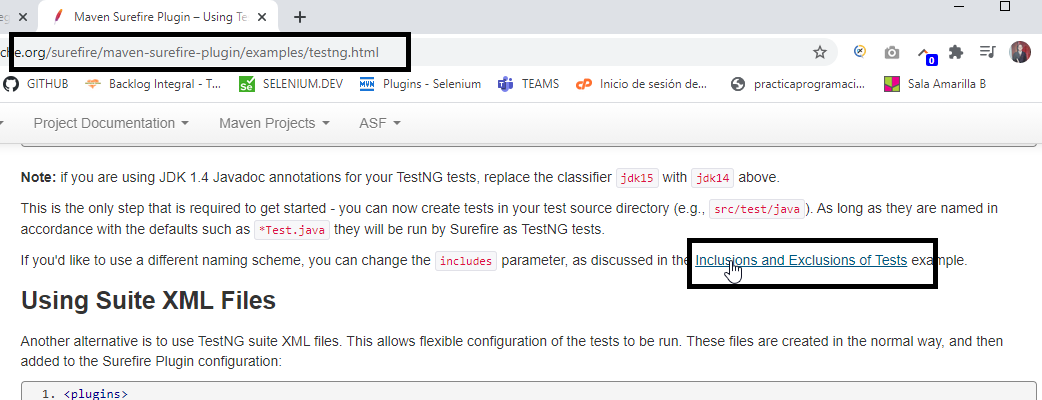
1. Entonces nos quedaría: **mvn clean test -DtestngFile=testng1.xml -Dtestng.dtd.http=true**

Y también podemos probar con el suite testng.xml

**mvn clean test -DtestngFile=testng.xml -Dtestng.dtd.http=true**

1. INCLUIR O EXCLUIR test específicos:

Vamos a la pagina de surfire <https://maven.apache.org/surefire/maven-surefire-plugin/examples/testng.html> y vemos el siguiente link

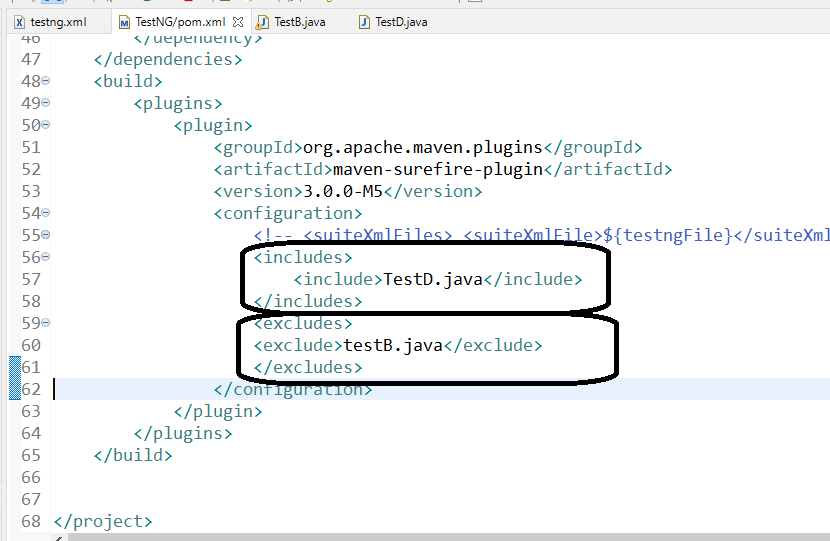


Como menciona tenemos los **tag <include>** si queremos incluir algún test en particular o **<exclude>** si queremos excluir un test.

Por ejemplo en nuestro caso vamos a incluir el testD el cual resulta fallido y excluiremos el testB.

En el pom.xml en este caso no usamos el tag **<suiteXmlFiles>**. Por eso lo comentamos.

El pom.xml nos queda de la siguiente manera:



Lo corremos: **mvn clean test -Dtestng.dtd.http=true**

Entonces excluyo el test B e incluyo el test D. Este ultimo resulto fallido.