UDEMY: The complete Selenium WebDriver with Java

# Sección 1: Conceptos básicos de Java

## Necesario tener

* Cuenta en Oracle
* JDK 10 o superior
* IDE Eclipse

## Crear proyecto

* New Project
* Java Project
* Nombre de proyecto (UpperCamelCase)

## Normas básicas de estilo

Packages:

* En minúsculas y sin espacios, si hay mas de una palabra utilizar \_
* No puede empezar por número o carácter especial (solo se permite $)

Clases:

* En mayúscula y sin espacios, si hay más de una palabra utilizar \_
* No puede empezar por número o carácter especial (solo se permite $)

Sigue con JAVA.……………………………………………………………………………………

# Sección 15: Introducción Selenium WebDriver

## Arquitectura WebDriver

## Proyecto Maven

Crear proyecto:

* New Project
* Maven 🡪 Maven Project
* Filter 🡪 Maven-archetype-quickstart (org.apache, el ultimo)
  + Artifact Id 🡪 Nombre del proyecto
  + Group Id 🡪 ej. Com

Dependencias:

* Pom 🡪 añadir dependencias de Selenium

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.11</version>

</dependency>

<dependency>

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

<artifactId>selenium-java</artifactId>

<version>3.0.1</version>

</dependency>

<dependency>

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

<artifactId>selenium-support</artifactId>

<version>3.0.1</version>

</dependency>

</dependencies>

Web Driver:

* En el Proyecto, New 🡪 Source folder
* Name 🡪 src/test/resources
  + New 🡪 Other
    - General 🡪 Folder
      * Name: WebDriver
* Añadimos en la carpeta el *chromedriver.exe* (tiene que coincidir con la versión Chrome instalada)

Clase de pruebas:

* En la carpeta test/java/com\*/nombre 🡪 New 🡪 Other
  + Java 🡪 JUnit 🡪 JUnit Test Case
    - Name 🡪 Nombre del test
    - Check 🡪 setUp() y tearDown()
* En la clase creada
  + Dentro de la clase, declarar la variable global del driver

private WebDriver driver;

* + - Como variables globales declararemos los elementos de la página con los que querremos interactuar

//Ejemplo

By titulo = By.*className*(“site-title”);

* En la función setUp()

//localización del driver

System.setProperty(“webdriver.chrome.driver”, “./src/test/resources/WebDriver/chromedriver.exe”);

driver = new ChromeDriver();

//pantalla completa del navegador

driver.manage().window().maximize();

//web con la que trabajamos

driver.get(“https://........”);

* En la función tearDown()

//cerrar el driver

driver.quit();

* Funciones de test:
  + Por defecto se crea una función llamada test()
  + Podemos crear tantos test como sea necesario (ej: testIdioma, testMenu…), de forma que los podremos ejecutar por separado
  + En estas funciones test escribiremos las acciones y comprobaciones que queremos hacer en la página (ej. click, findElement, prints en consola…)

## Automatización de la navegación

Funciones:

//ir a una pagina web

driver.get(“https://........”);

//ir a otra página

driver.navigate().to(“https://........”);

//atrás en el navegador

driver.navigate().back();

//volver en el navegador

driver.navigate().forward();

//refrescar página

driver.navigate().refresh();

## Get de detalles básicos de la página

//get URL

driver.getCurrentUrl();

//get titulo de la págia

driver.getTitle();

//get ID de la Ventana (identificador que le da Selenium)

driver.getWindowHandle();

# Sección 16: localizadores de elementos

Me salto “HTML Basics”……………………………………………………………………….……

## Inspeccionar elementos

Vamos al inspector de la página y nos situamos en el elemento que interesa (botón de inspección  y seleccionamos el elemento o volvemos a inspeccionar encima del mismo).

Desde allí decidimos que es lo que nos interesa para referirnos a él (tipo, id, clase…).

## Declarar elementos y usarlos

Se puede hacer por variables globales y llamar desde la función, o declarar y llamar en la función.

* Variables globales:

//ejemplo declaración

By element = By.*id*(“id-elemento”);

* Llamar en la función

//si se declara como global, se podrá usar en distintos métodos del test

driver.*findElement*(element);

//si no se declara, se deberá declara cara vez que se quiera usar

driver.*findElement*(By.*id*(“id-elemento”));

## Encontrar elementos por ID o enlaces

//encontrar por atributo id

By.*id*(“id-elemento”)

//encontrar por enlace (es de tipo <a> y tiene atributo href)

By.*linkText*(“Texto completo del enlace”)

//si el texto es muy largo (ej. Silla plegable marron y negro)

//podemos utilizar la siguiente función, escribiendo parte del texto

By.*partialLinkText*(“Silla plegable”)

## Encontrar elementos por nombre o clase

//encontrar por nombre (atributo name)

By.*name*(“nombre”)

//encontrar por clase (atributo class)

By.*className*(“Texto del enlace”)

# Sección 17: XPath

## SelectorsHub

Se trata de una extensión de Chrome que permite ver de forma más clara los atributos y distintas propiedades de los elementos.

Se añade una nueva pestaña en el inspector y desde allí puedes seleccionar el elemento deseado y te muestra la información.

## XPath utilizando atributos

XPath es una forma de rederenciar el elemento, y se puede hacer de distintas formas. Utilizando los atributos tendría la siguiente estructura: **//tipo[@nombre-atributo=’valor del atributo’]**

## XPath utilizando el texto

Ejemplo:

By.xpath(“//button[text()=’Sign Up’]”)

## XPath para elementos dinámicos

Si necesitamos más de un atributo para poderlo referenciar, lo podemos hacer separando por *and* (de la misma manera que funcionará el *or* para indicar que tenga uno de los dos atributos).

Ejemplo:

By.xpath(“//button[@class=’b\_searchbox’ and text()=’Search Now’]”)

Si el texto o el valor de un atributo puede cambiar o es muy largo se puede utilizar la función *contains*, que indica que tiene un determinado valor parcial, sin importar lo que vaya delante o detrás (de la misma forma funciona el *starts-with*, pero para indicar que empieza de determinada forma).

Ejemplo:

By.xpath(“//button[contains(@class=’b\_’)]”)

# Sección 18: Ejes en XPath

## XPath utilizando padre-hijo

Hay veces que encontramos elementos que no tienen suficientes atributos para referenciarlos, de manera que tenemos que referenciar alguno que si (el padre, por ejemplo) y encontrarlo a partir de éste.

* Ejemplos para acceder al o los hijos:

//cogería a todos los hijos

By.xpath(“//div[@id=’padre’]/child::\*”)

//cogería al hijo/hijos de tipo span

By.xpath(“//div[@id=’padre’]/child::span”)

//cogería al primer hijo de tipo div

By.xpath(“//div[@id=’padre’]/child::div[0]”)

* Ejemplo para acceder al padre

//utilizamos \* porque solo se puede tener un padre

By.xpath(“//div[@id=’hijo’]/parent::\*”)

## XPath utilizando ancestro-descendiente

Decimos anestro y descendiente a los elementos que no tienen una relación directa de padre (ej. el padre del padre o más) o hijo (hijo del hijo o más).

Utilizando *ancestor* o *descendant* podemos llegar a ellos.

* Ejemplo ancestro

//el div padre del padre

By.xpath(“//div[@id=’hijo-partida’]/*ancestor*::div[1]”)

//el ancestro de tipo label

By.xpath(“//div[@id=’hijo-partida’]/*ancestor*::label”)

* Ejemplo descendiente:

//el tercer descendiente div (el hijo del hijo del primer hijo)

By.xpath(“//div[@id=’padre-partida’]/descendant::div[2]”)

//el tercer descendiente div (el hijo del hijo del primer hijo)

By.xpath(“//div[@id=’padre-partida’]/descendant::div[2]”)

En el caso de descendiente, vas por orden encontrando los que cumplan la condición, de forma que la expresión en general se refiere al tercer div hijo, da igual el nivel.

## XPath utilizando precedente y siguiente

Utilizaremos el *preceding* para referirnos a los elementos por encima hasta llegar al elemento padre, que queda fuera de la selección.

//todos los que están por encima dentro del mismo padre

By.xpath(“//div[@id=’elemento’]/preceding::\*”)

Lo mismo pasa con *following*, pero con todos los elementos que lo siguen dentro del mismo padre.

//todos los que están por debajo dentro del mismo padre

By.xpath(“//div[@id=’elemento’]/following::\*”)

También podemos añadir *-sibling­* para referirnos a los que están al mismo nivel sin contar sus hijos.

//todos los que están por debajo dentro del mismo padre al MISMO nivel (hermano)

By.xpath(“//div[@id=’elemento’]/following-sibling::\*”)

## XPath absoluta y relativa

* **Absoluta:** se refiere hasta la raíz, es decir, des del elemento <html> (empieza con /html/body/…)
* **Relativa:** desde donde indiquemos, desde donde se encuentre el elemento que cumpla las características indicadas (empieza con //…)

# Sección 19: Selectores CSS

Me salto “Que es CSS y CSS Selector”…………………………………………………………….

## CSS Selector utilizando atributos

Funciona casi igual que XPath, pero sin / o // en el inicio de la expresión, y sin el @ para los atributos: **TagName[Attr=’Value’]**

By.cssSelector(“TagName [Attr=’value’]”)

Si se trata del id, igual que en CSS podemos cambiar el nombre por # (**div#Value**) y si es una clase, por un punto (**div.Value**) y se simplifica la expresión.

//name

By.cssSelector(“div#name-value”)

//class

By.cssSelector(“div.class-value”)

## CSS Selector elementos dinámicos

//valor parcial

By.cssSelector(“tagName[Attr\*=’value’]”)

//valor empieza por

By.cssSelector(“tagName[Attr^=’starting-with]”)

//valor acaba por

By.cssSelector(“tagName[Attr$=’ending-with’]”)

## CSS Selector múltiples elementos

Si queremos especificar más de un atributo, lo haremos abriendo y cerrando [ ] por cada uno.

By.cssSelector(“tagName[Attr1=’value’][Attr2=’value’]”)

Y si queremos especificar un atributo u otro, lo separamos con una coma.

By.cssSelector(“tagName[Attr1=’value’],[Attr2=’value’]”)

## CSS Selector opciones avanzadas

Si queremos seleccionar todos los elementos de un tipo, no hace falta añadir nada más que el nombre del tipo. Y si queremos que tengan un atributo (da igual el valor) no hace especificar nada en el valor.

//valor parcial

By.cssSelector(“tagName”)

//valor empieza por

By.cssSelector(“tagName[Attr]”)

Si queremos seleccionar mas de un tipo de elemento, escribimos los dos y separamos por coma. Las expresiones que utilizamos son las mismas utilizadas para seleccionar en CSS.

//que coincidan con los dos

By.cssSelector(“tagName1[Attr],tagName2[Attr]”)

Si queremos seleccionar los descendientes de un elemento, separaremos con un espacio.

//que coincida con el segundo elemento que sea descendiente del primero

By.cssSelector(“tagName1[Attr] tagName2[Attr]”)

Para seleccionar los hijos de un elemento, utilizaremos el símbolo mayor que. Tener en cuenta que solo coge los hijos, no sus descendientes.

//los hijos del elemento 1

By.cssSelector(“tagName1[Attr]> tagName2[Attr]”)

Para seleccionar el siguiente hermano de un elemento utilizamos el símbolo mas.

//El hermano del primer elemento con las siguientes caracteristicas

By.cssSelector(“tagName1[Attr]+tagName2[Attr]”)

Si queremos todos los hermanos que están a continuación de un elemento, utilizamos ∼.

//El hermano del primer elemento con las siguientes caracteristicas

By.cssSelector(“tagName1[Attr]∼tagName2[Attr]”)

## CSS Selector con el índice de elemento

Si tenemos una lista pero queremos un elemento en concreto, podemos hacerlo de distintas formas.

Si queremos el primer elemento hijo, escribiremos :first-child.

//El primer hijo del primer elemento que cumpla la condición

By.cssSelector(“tagName1[Attr]>tagName2:first-child”)

Si queremos un hijo en concreto, podemos poner :nth-child(indice).

//El segundo hijo del primer elemento que cumpla la condición

By.cssSelector(“tagName1[Attr]>tagName2:nth-child(2)”)

También Podemos seleccionar el ultimo hijo con :last-child.

//El ultimo hijo del primer elemento

By.cssSelector(“tagName1[Attr]:last-child”)

# Sección 20: Trabajando con colecciones de elementos

## Como recuperar una colección de WebElements

Si queremos recuperar una colección necesitaremos guardarlos en una lista, de esta forma podremos trabajar con ellos.

//recuperar una lista con todos los links (<a>)

List <WebElement> links = driver.getElements(By.cssSelector(“a”));

Una vez lo temenos, Podemos recorrerlos todos con *for* o *foreach*.

//ejemplo imprimiendo los textos de los links y el atributo class

for (WebElement link : links) {

System.out.println( link.getText() );

System.out.println( link.getAttribute(“class”) );

}

## Elementos hidden y visibles

Los elementos pueden existir y ser invisibles. Para ver si un elemento es visible podemos utilizar la función isDisplayed(), que devolverá true si es visible y false si es invisible.

//imprimir si el titulo es visible

By titulo = By.*className*(“site-title”);

System.out.println( titulo.isDisplayed() );

Nos puede servir para comprobar si esta visible primero, y si lo está, hacer una determinada acción. Por ejemplo, comprobar primero que un botón este visible antes de pulsarlo.

## Elemento dentro de otro elemento

Un objeto, como podría ser un articulo en una tienda, esta formado por distintos elementos. Pongamos la situación que un precio se corresponde con el título del artículo. Si capturamos el precio para ver si está dentro del rango, ¿cómo podemos volver al titulo para ver que corresponda correctamente?

Primero tenemos que capturar los artículos enteros, el elemento que contiene todos los elementos del mismo en una lista. Después recorreremos cada uno de los artículos, y es allí que comprobaremos el precio, accediendo a el, y si tiene el precio que nos interesa ya tenemos el articulo entero, sobre el que podremos acceder al titulo o cualquier otro elemento.

//ejemplo

List<WebElement> items = driver.findElements(By.cssSelector(“div.item-wrapper”));

for (WebElement item : items) {

String precio = item.findElement(By.cssSelector(“span.price:first-child”));

double precioNum = Double.parseDouble(precio);

If (precioNum > 20 || precioNum < 50) {

//ya temenos el titulo y podemos trabajar con el

WebElement titulo = item.findElement(By.cssSelector(“a.title”));

titulo.click();

}

}

# Sección 21: Capturar detalles de WebPage para hacer validaciones

## Capturar los detalles básicos

Podemos guardar los detalles en objetos como Strings, de esa forma podemos hacer comparaciones o lo que necesitemos con ellos de forma mas fácil.

//capturar class

String class = driver.findElement(By.id(“id-elemento”)).getAttribute(“class”);

//o

String class = driver.findElement(By.id(“id-elemento”)).getClass();

//capturar aria-label

String aria = driver.findElement(By.id(“id-elemento”)).getAttribute(“aria-label”);

//capturar tag

String tag = driver.findElement(By.id(“id-elemento”)).getTagName();

## Capturar los detalles del CSS

También podemos capturar las características estéticas del CSS, para poder luego comparar o hacer las acciones que convengan.

//capturar color

String color = driver.findElement(By.id(“id-elemento”)).getCssValue(“color”);

//capturar color de fondo

String bgColor = driver.findElement(By.id(“id-elemento”)).getCssValue(“background-color”);

## Verificar si un elemento esta visible o habilitado

Visible se refiere si se puede ver en la pagina o no, y habilitado (enabled) si se puede interaccionar o no (un botón, checkbox, input…).

Antes de realizar una acción, quizás es mejor comprobar si esta visible y si se puede interactuar.

//ver si esta visible, true si está visible

boolean isDisplayed = driver.findElement(By.id(“id-elemento”)).isDisplayed();

//ver si esta habilitado, true si se puede interactuar

boolean isEnabled = driver.findElement(By.id(“id-elemento”)).isEnabled();

## Verificar si un elemento existe o no

Seria el paso anterior a verificar si esta visible, ya que si no existe no hace falta realizar ninguna comprobación más, además que saltará error. No existe una función como tal, sino que lo que tenemos que hacer es rodear el la línea donde llamamos el isDisplayed() con un try/catch.

//ver si esta visible, true si está visible

try {

boolean isDisplayed = driver.findElement(By.id(“id-elemento”)).isDisplayed();

} catch (Exception e) {

//a aquí Podemos ejecutar lo que queramos si fallan las sentencias dentro del try

}

## Verificar si un elemento está seleccionado

Es importante verificar si, por ejemplo, un checkbox está seleccionado o no antes de realizar la acción, según lo que queramos hacer.

//verificar si esta seleccionado el checkbox

boolean checkboxSelected = driver.findElement(By.id(“id-elemento”)).isSelected();

# Sección 22: Automatizando WebElements especiales

## Select-Boxes

Para capturar el elemento seleccionado del select, tenemos que acceder al atributo value.

//capturer el mes seleccionado en el Select

String monthSelected = driver.findElement(By.id(“select-month”)).getAttribute(“value”);

Y para cambiar el valor, utilizamos capturamos la lista de todas las opciones, recorremos la lista hasta encontrar la que deseamos, entonces le damos click.

//capturamos la lista de opciones (tipo select, con id select-month, hijos opciones)

List<WebElement> monthList = driver.findElement(By.cssSelector(“select#select-month>option”));

//recorremos la lista y seleccionamos Dec

for (WebElement month : monthList) {

if (month.getText().equals(“Dec”)) {

month.click();

break;

}

}

## MultiSelect-Boxes

Hay selects que te permiten seleccionar diferentes opciones, por ejemplo, un filtro de una tienda de ropa, que puedes seleccionar los colores de ropa que quieres que te aparezcan.

Por eso también capturaremos la lista de las opciones, y recorreremos la lista, añadiendo en el if las opciones que queremos seleccionar.

//capturamos la lista de opciones

List<WebElement> colors = driver.findElement(By.cssSelector(“#colors li”));

//recorremos la lista y seleccionamos Verde y Azul

for (WebElement color : colors) {

if (color.getText().equals(“Green”) || color.getText().equals(“Blue”)) {

color.click();

}

}

## Seleccionador de fechas

Para seleccionar la fecha de un calendario, por ejemplo un checkin y un checkout, se debe seleccionar el botón del calendario y

//mes deseado

String checkinMonth = “Aug 2022”;

//click en la fecha

driver.findElement(By.xpath(“//button[@id=’calendar-checkin’]”)).click();

//capturamos el mes actual

String currentMonth = driver.findElement(By.cssSelector(“th.title-month”)).getText();

//añadimos un contador por si no encontramos el mes

Int count = 0;

//vamos pasando de mes hasta llegar al deseado

while(!currentMonth.contains(checkinMonth)) {

//si no es el mes deseado, click a siguiente

driver.findElement(By.cssSelector(“button.next”)).click();

count++;

Thread.sleep(1000);

currentMonth = driver.findElement(By.cssSelector(“th.title-month”)).getText();

if (count = 12) { Break; }

}

//una vez temenos el mes, seleccionamos el día

String checkinDay = “15”;

List<WebElement> dates = driver.fingElements(By.cssSelector(“table.cal-month td”));

For (WebElement date : dates) {

String d = date.getText().trim(); //quitamos espacios porsiacaso

If (d.equals (checkinDay) ) {

date.click()

break;

}

}

## Tablas

Primero lo que tendremos que hacer es coger todas las filas de la tabla y recorrerlas con un bucle, desde allí podremos hacer las acciones deseadas por cada una, o a la deseada.

//todas las filas

List<WebElement> rows = driver.findElements(By.xpath(“//table[contains(@class,’tg’)]/tbody/tr”));

//bucle

for (WebElement row : rows) {

String name = row.findElement(By.xpath(“td[1]”)).getText();

If (name.equals(“Luis”)) {

//accion deseada

}

}

## Objetos en celdas de tablas

Las tablas puede que tengan botones, checkbox o selects a parte de texto.

Lo que tendremos que hacer es encontrar algo característico de la fila con la que se quiere trabajar (nombre por ejemplo) y luego atacar al objeto o acción que se quiere realizar.

//todas las filas

List<WebElement> rows = driver.findElements(By.xpath(“//table[contains(@class,’tg’)]/tbody/tr”));

//bucle

For (WebElement row : rows) {

String name = row.findElement(By.xpath(“td[1]”)).getText();

If (name.equals(“Luis”)) {

//checkbox e la cuarta columna

row.findElement(By.xpath(“td[4]/input”)).click();

//select e la quinta columna

List<WebElement> selects = row.findElement(By.xpath(“td[5]/select/option”));

//recorremos las opciones para seleccionar la deseada

For(WebElement opt : selects) {

If(opt.getText().contains(“Sports”)) { opt.click(); }

}

//input en la sexta columna

row.findElement(By.xpath(“td[6]/input”)).sendKeys(“Helloooo”);

break;

}

}

# Sección 23: Automatizando pestañas en el navegador

## Cambiar de pestaña en el navegador

Hay veces que dando click a un link se abre una nueva pestaña, y tenemos que cambiar para seguir haciendo comprobaciones.

//click al enlace

driver.findElement(By.linkText(“Data Policy”)).click();

//se abre otra Ventana y capturamos en la que estamos

String parentWindow = driver.getWindowHandle();

//capturamos todas las ventanas buscamos

Set<String> windows = driver.getWindowHandles();

//recorremos las ventanas hasta encontrar la que se ha abierto

For(String window : windows) {

If(!window.equals(parentWindow)) {

//si no es igual a la pagina donde estamos, cambiamos la pestaña

driver.switchTo().window(window);

}

}

//click en el botón de la segunda pantalla

driver.findElement(By.linkText(“Accept”)).click();

//para volver otra vez a la pestaña anterior

driver.switchTo().window(parentWindow);

## Trabajando con múltiples pestañas

Ponemos el caso de que acabas teniendo muchas pestañas abiertas, ¿cómo puedes identificarlas?

Primero tendremos que capturarlas en una lista, recorrer las pestañas y pedir a driver la información que nos interesa (url, titulo, id…).

//capturamos todas las ventanas

Set<String> windows = driver.getWindowHandles();

//recorremos las ventanas hasta encontrar la que se ha abierto

For(String window : windows) {

//cambiamos de pestaña

driver.switchTo().window(window);

//imprimimos info de la misma

System.out.println(window); //id

System.out.println(driver.getCurrentUrl()); //url

System.out.println(driver.getTitle()); //titulo

//si queremos trabajar con una url en concreto

If (driver.getCurrentUrl().equals(“https://www.facebook.com”)) {

Break; //salimos y nos quedamos en la pagina deseada

}

}

## Cerrar todas las pestañas

Ponemos el caso de que tenemos muchas pestañas abiertas y las queremos cerrar excepto la principal.

// capturamos en la que estamos

String parentWindow = driver.getWindowHandle();

//capturamos todas las ventanas abiertas

Set<String> windows = driver.getWindowHandles();

//recorremos las ventanas

For(String window : windows) {

//si la Ventana no es la principal, la cerramos

If(!window.equals(parentWindow)) {

// cambiamos la pestaña y la cerramos

driver.switchTo().window(window);

driver.close();

Thread.sleep(1000);

}

}

//cambiar a la página principal porsiacaso

driver.switchTo().window(parentWindow);

## Diferencia entre colse y quit

Con driver.close() se cierra la pestaña principal del driver, o la pestaña en activo (cuando utilizas andtes del close el switchTo().window(window)).

El driver.quit(), cierra todas las pestañas del navegador.

# Sección 24: Automatizando iFrames

## iFrames

Un iframe es un documento html dentro de otro. Lo que hay que hacer para trabajar con los elementos del iframe es cambiar el driver a este html y trabajar en el entonces.

//ponemos el caso que hay un iframe con un botón que queremos pulsar

//en vez de window, utilizamos la función frame y el índice de cual queremos

driver.switchTo().frame(0);

//ahora ya podemos hacer las acciones que queramos como si fuera en una página normal

driver.findElement(By.linkText(“Click Me”)).click();

También podríamos pulsarlo utilizando el id o nombre del frame.

driver.switchTo().frame(“iFrame1”);

driver.findElement(By.linkText(“Click Me”)).click();

Otra forma es utilizando el WebElement.

//primero capturamos el frame que nos interesa

WebElement frame1 = driver.findElement(By.xpath(“iframe[@id=’iFrame1’]”));

//y cambiamos el driver con el objeto capturrado

driver.switchTo().frame(frame1); //a continuacion, como las opciones anteriores

## iFrames anidados

Llamamos iFrame anidado a un iframe que está dentro de otro iframe. En este caso, desde la pagina principal no vemos el iframe de dentro, de forma que tendremos que tratarlo diferente.

Lo que tendremos que hacer es cambiar el driver al iframe padre, y un a vez desde allí, ya podremos ver el hijo y cambiar.

//pimero cambiamos al iframe padre

driver.switchTo().frame(“parentFrame”);

//ahora podemos cambiar al iframe hijo

driver.switchTo().frame(“nestedFrame”);

//ahora, ha Podemos hacer acciones dentro del iframe hijo

driver.findElement(By.linkText(“Click Me”)).click();

//para volver al iframe padre

driver.switchTo().parentFrame();

# Sección 25: Automatización de alertas Javascript

Una alerta JavaScript no se puede ignorar, ya que, si no aceptas o cancelas la notificación, la página se queda congelada, no se puede interactuar con ella.

## Alert Box Javascript

Una Alert Box es una alerta que sala, la cual advierte de algo y solo aparece un botón, el de “Aceptar”.

//dar click al botón que hará saltar la alerta

driver.findElement(By.linkText(“Click Me”)).click();

Thread.sleep(2000);

//cambiamos a la alerta y guardamos el elemento

Alert alert = driver.switchTo().alert();

//ahora ya podemos interactuar con las funciones de la alerta

String textoAlerta = alert.getText(); //para capturar el texto

alert.accept(); //como si diéramos click al botón aceptar de la alerta

## Box de confirmación Javascript

Una Confirmation Box es una alerta que da la posibilidad de aceptar o cancelar una acción, es como una verificación de “¿seguro que quieres continuar?”.

//dar click al botón que hará saltar la alerta

driver.findElement(By.linkText(“Click Me”)).click();

Thread.sleep(2000);

//cambiamos a la alerta y guardamos el elemento

Alert alert = driver.switchTo().alert();

alert.accept(); //para dar a ok

//o

alert.dismiss(); //para cancelar

## Prompt Box Javascript

Una Prompt Box es una alerta que acepta un input del usuario, pudiendo luego aceptar o cancelar a acción.

//dar click al botón que hará saltar la alerta

driver.findElement(By.linkText(“Click Me”)).click();

Thread.sleep(2000);

//cambiamos a la alerta y guardamos el elemento

Alert alert = driver.switchTo().alert();

alert.sendKeys(“Holaaaaaaa”); //rellenar el input

alert.accept(); //para dar a ok

//o

alert.dismiss(); //para cancelar

# Sección 26: Automatización de acciones del ratón

## Hover

Hover es cuando pones el ratón encima de un elemento. Hay elementos que al hacer esta acción cambian de color, o aparecen opciones nuevas (ej. Un menú desplegable).

//creamos una acción

Actions action = new Actions(driver);

//Buscamos el elemento que queremos hacer el hover y lo capturamoss

WebElement menEl = driver.findElement(By.linkText(“Clients”));

//ponemos el raton encima de este

action.moveToElement(menEl).perform();

Thread.sleep(2000);

//ahora, que ya ha aparecido el menú desplegable, ya podemos hacer click al elemento

driver.findElement(By.linkText(“Client info”)).click();

## Click botón derecho

//creamos una acción

Actions action = new Actions(driver);

//Buscamos el elemento que queremos hacer el click derecho y lo capturamos

WebElement btn = driver.findElement(By.linkText(“Right Click”));

//hacemos el click derecho

action.contextClick(btn).perform();

//ahora, que ya ha aparecido la ventana, ya podemos hacer acciones como bajar con el teclado hasta la opción deseada

action.sendKeys(Keys.*ARROW\_DOWN*).perform(); //↓

Thread.sleep(1000);

action.sendKeys(Keys.*ARROW\_DOWN*).perform(); //↓

Thread.sleep(1000);

action.sendKeys(Keys.*RETURN*).perform(); //enter

## Drag and Drop

Los elementos que pueden ser agarrados y soltados suelen estar dentro de in iFrame (comprobar primero para ver si es el caso), de esta forma lo primero que hay que hacer es entrar en el iframe y luego desde allí realizar la prueba.

driver.switchTo().frame(“iFrameDragDrop”);

Pongamos el caso que tenemos un elemento que se puede coger y otro donde lo podemos dejar. Primero tenemos que capturar ambos elementos utilizando algún atributo que los identifique, como el id.

//capturamos ambos elementos

WebElement agarrar = driver.findElement(By.id(“draggable”));

WebElement soltar = driver.findElement(By.id(“droppable”));

//creamos una action para poder hacer el drag and drop

Actions action = new Actions(driver);

action.clickAndHold(agarrar).moveToElement(soltar).release(agarrar).build().perform();

## Redimensionar elementos UI

Los elementos que pueden ser agarrados y soltados suelen estar dentro de in iFrame (comprobar primero para ver si es el caso), de esta forma lo primero que hay que hacer es entrar en el iframe y luego desde allí realizar la prueba.

//cambiar a iframe

driver.switchTo().frame(“iFrameRedimension”);

//capturamos el elemento del extremo, desde donde podemos/queremos redimensionar

WebElement redimensionable = driver.findElement(By.id(“resize”));

//creamos una action para poder redimensionar

Actions action = new Actions(driver);

//primero dejamos seleccionado y lo movemos los pixeles que indicamos desde el origen

action.clickAndHold(redimensionable).moveByOffset(30,20).build().perform();

## Slider

Un slider es un elemento de las interfaces gráficas que permiten seleccionar un valor moviendo un indicador o, en algunos casos, el usuario puede hacer clic sobre algún punto del slider para cambiar hacia ese valor (ejemplo, la barra para seleccionar el límite de precio en una tienda online).

Lo primero que tenemos que hacer es localizar y capturar el elemento indicador (comprobar si está dentro de un iframe, si no está, cambiar a él), que es el que moveremos para hacer cambiar el valor.

//capturamos el indicador

WebElement indicador = driver.findElement(By.id(“ind”));

//creamos una action y movemos el indicador

Actions action = new Actions(driver);

//igual que cuando redimensionamos, pero no nos movemos en el eje de la Y

action.clickAndHold(indicador).moveByOffset(-20,0).build().perform();

## Operaciones Multi-Key

Hay algunas acciones que requieren pulsar un botón del teclado para seleccionar diferentes elementos (como cuando seleccionamos archivos en una carpeta con ctrl+click).

//capturamos los elementos para seleccionar (mirar si está en un iframe, si es así, entrar en él)

WebElement elemento1 = driver.findElement(By.id(“sel1”));

WebElement elemento4 = driver.findElement(By.id(“sel4”));

WebElement elemento5 = driver.findElement(By.id(“sel5”));

//creamos una action y movemos el indicador

Actions action = new Actions(driver);

//click a ctrl

action.keyDown(Keys.*CONTROL*).perform();

//realizamos los clicks

action.click (elemento1);

action.click (elemento4);

action.click (elemento5);

//soltamos ctrl

action.keyUp(Keys.*CONTROL*).perform();

# Sección 27: Sincronización del Test

## Que es la sincronización de test

Es la habilidad de hacer coincidir la velocidad de ejecución de la aplicación con la velocidad de ejecución de nuestro test.

A veces hay acciones que no se ejecutan al instante, sino que necesitan un tiempo de carga para procesar la información. En estos casos tendremos que hacer esperar al test, si no, dará error anuqué el programa esté correcto.

//ejemplo de formulario, lo rellenamos

driver.findElement(By.id(“first-name”)).sendKeys(“John”);

driver.findElement(By.id(“second-name”)).sendKeys(“Doe”);

//click en submit

driver.findElement(By.linkText(“Click Me”)).click();

//tarda un rato al dar la respuesta, así que esperamos unos segundos

Thread.sleep(8000); //8 segundos (no se recomiendan más de 10 segundos)

//check del element que aparece al procesar la información

If(driver.findElement(By.id(“ok-message”)).isDisplayed()) {

System.out.println(“BIEN! Aparece el mensaje”);

}

## Implicit Wait

El ejemplo propuesto anterior no está mal, pero si el proceso dura menos, es inútil esperar 8 segundos para seguir ejecutando.

Existe la sentencia “implicit wait”, que se aplica al driver para que se espere el tiempo indicado antes de lanzar la excepción de que no se ha encontrado, comprobando en cada momento si **existe** el elemento.

//esta función se tiene que llamar antes de entrar a la URL

driver.manage().implicitlyWait(10, TimeUnit.*SECONDS*); //tiempo, unidad de tiempo

## Explicit Wait

Si el elemento que se quiere interactuar al terminar una acción si existe pero no está activo (por ejemplo, un botón inactivo), el “implicit wait” no nos servirá (se realizará el click pero no pasará nada).

//se tiene que crear después que el driver (se puede crear como global e inicializar a aquí)

WebDriverWait wait = new WebDriverWait(driver, 10); //driver y segundos

Una vez inicializado, ya se puede utilizar allí donde sea necesario.

//ejemplo de formulario, lo rellenamos

driver.findElement(By.id(“first-name”)).sendKeys(“John”);

driver.findElement(By.id(“second-name”)).sendKeys(“Doe”);

//click en submit

driver.findElement(By.linkText(“Click Me”)).click();

//esperamos (existen muchas acciones para ExpectedConditions)

wait.until(ExpectedConditions.elementToBeClickable(driver.findElement(By.id(“ok-link”)));

//click

driver.findElement(By.id(“ok-link”)).click();

# Sección 28: Selenium 4, nuevas funcionalidades

## Configuración de Selenium v.4

Se tiene que instalar el JDK 11 cambiar la dependencia Selenium por la nueva en el archivo dom.xml.

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

<dependency>

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

<artifactId>selenium-java</artifactId>

<version>4.0.0-alpha-2</version>

</dependency>

Después habrá que actualizar el proyecto (encima de la carpeta del proyecto, botón derecho, Maven, Update Project…).

## Implicit Wait

Funciona casi igual, pero en esta versión la otra función está obsoleta y se tiene que llamar de la siguiente forma.

//esta función se tiene que llamar después de entrar a la URL

driver.manage().timesouts().implicitlyWait(Duration.ofSeconds(10)); //hay distintas funciones de .of que acepta otras unidades de tiempo

## Abriendo pestañas y ventanas de navegador

Es una nueva función de esta versión. Permite abrir pestañas y nuevas ventanas del navegador.

//abrimos una pagina

driver.get(“https://www.amazon.com”);

//abrir una nueva pestaña

driver.switchTo().newWindow(WindowType.*TAB*); //cambia automaticamente a la nueva

driver.get(“https://www.facebook.com”);

//abrir una nueva ventana

driver.switchTo().newWindow(WindowType.*WINDOW*); //automaticamente va a la nueva

driver.get(“https://www.google.com”);

Si queremos visitar ahora la pagina de Amazon por ejemplo, tenemos que guardar las id y hacer el cambio con el switchTo().

//recuperamos los IDs de las ventanas y pestañas

Set<String> windowsID = driver.getWindowHandles();

//creamos las variables para poder cambiar mas fácilmente en el futuro

String amazonId = “”;

String facebookId = “”;

String googleId = “”;

//guardamos la id en la variable

For ( String id : windowsID ) {

driver.switchTo().window(id);

String url = driver.getCurrentUrl();

If (url.contains(“facebook”)) {

facebookId = id;

} else If (url.contains(“amazon”)) {

amazonId = id;

} else {

googleId = id;

}

}

//cambiamos a Amazon

driver.switchTo().window(amazonId);

## Localizadores relativos

Te permiten localizar elementos en referencia a otro elemento. Es útil cuando hay elementos duplicados en la página, por ejemplo botones, o para comprobar que los elementos están en la posición deseada.

//por encima de

WebElement button1 = driver.findElement(By.textLink(“boton-abajo”));

Try { //encontrar elemento “boton-arriba” esta por encima de boton1(“boton-abajo”)

driver.findElement(By.textLink(“boton-arriba”)).above(button1);

} catch (Exception e) {

//no lo ha encontrado

}

//por debajo de

WebElement button1 = driver.findElement(By.textLink(“boton-arriba”));

Try { //encontrar elemento “boton-abajo” está por debajo de boton1(“boton-arriba”)

driver.findElement(By.textLink(“boton-abajo”)).above(button1);

} catch (Exception e) {

//no lo ha encontrado

}

Existen más funciones como

//cerca de

driver.findElement(By.textLink(“enemento1”)).near(elemento2);

//a la izquierda de

driver.findElement(By.textLink(“enemento1”)).toLeftOf(elemento2);

//a la derecha de

driver.findElement(By.textLink(“enemento1”)).toRightOf(elemento2);

## Minimizar el navegador

Añade la posibilidad de minimizar la ventana del navegador. Si minimizamos la ventana, aunque esté escondida la página, el test sigue hasta terminar.

driver.manage().window().minimize();

## Actualización de la clase Actions

//click

~~action.moveToElement(By.textLink(“boton”)).click();~~

action.click(By.textLink(“boton”));

//clickAndHold

~~action.moveToElement(By.textLink(“boton”)).clickAndHold();~~

action.clickAndHold(By.textLink(“boton”));

//contextClick

~~action.moveToElement(By.textLink(“boton”)).contextClick();~~

action.contextClick(By.textLink(“boton”));

//dobleClick

~~action.moveToElement(By.textLink(“boton”)).dobleClick();~~

action.dobleClick(By.textLink(“boton”));

Entre otros.

# Sección 29: Operaciones varias

## Screenshot de la página (WebDriver)

Pantallazo de la pantalla completa.

//se crea el archivo con el pantallazo

File file = ((TakesScreenshot)driver).getScreenshotAs(OutputType.*FILE*);

//guarda el pantallazo a la dirección que se desee

FileUtils.copyFile(file, new File(“./screenshots/file1.png”));

//pedirá un throws IOexception, se aplica

## Screenshot de una sección especifica de la página

Pantallazo de un elemento en concreto.

//capturas el objeto que desear hacer el pantallazo

WebElement elemento = drive.getElement(By.id(“element”));

//se crea el archivo con el pantallazo

File file = elemento.getScreenshotAs(OutputType.*FILE*);

//guarda el pantallazo a la dirección que se desee

FileUtils.copyFile(file, new File(“./screenshots/file1.png”));

## Obtener la altura y ancho de un elemento

Primero lo que se tiene que hacer es crear el WebElement que quieras conocer altura y ancho, creas un objeto de la clase Dimension o Rectangle, que captará sus dimensiones y desde este se pueden consultar

//capturas el elemento

WebElement elemento = drive.getElement(By.id(“element”));

//creas el objeto Dimension

Dimension d = elemento.getSize();

//ahora ya podemos acceder a la altura y ancho

System.out.ptintln(d.getHeight());

System.out.ptintln(d.getWidth());

//con Rectangle funciona igual

Rectangle r = elemento.getRect();

System.out.ptintln(r.getHeight());

System.out.ptintln(r.getWidth());

## Obtener las coordenadas X e Y de un elemento

Con la clase Point podemos conocer la posición X e Y del extremo superior izquierdo de un elemento.

//capturas el elemento

WebElement elemento = drive.getElement(By.id(“elemento”));

//guardas el punto (importar la clase de Selenium)

Point p = elemento.getLocation();

//ahora ya podemos acceder a la altura y ancho

System.out.ptintln(p.getX());

System.out.ptintln(d.getY());

//con Rectangle también podemos acceder a XY

Rectangle r = elemento.getRect();

System.out.ptintln(r.getX());

System.out.ptintln(r.getY());

# Sección 30: Testing (Parte 1)

## Instalación de TestNG (Extensión)

Al ejecutar un test con esta extensión, se abre una pestaña por el área del terminal donde aparece más información sobre la ejecución.

Dentro de Eclipse:

* Help 🡪 Eclipse Marketplace…
* Buscar: TestNg for Eclipse
* Install
* Reiniciar IDE

Una vez reiniciado Eclipse:

* Buscar en línea la dependencia Maven de TestNG
* Añadirla al pom.xml

Estos pasos servirán para cualquier extensión.

Me salto “Primer test automatizado”………………………………………………………………

## Assertions

Assert es una clase que tiene distintas funciones de testing, de esta forma se pueden hacer comprobaciones sin que salte error en el programa. El resultado obtenido será positivo si pasa el test y negativo si no pasa. Si no pasa el test, el código deja de ejecutarse en este punto.

//assertEquals: compara si dos valores son iguales o no

Assert.assertEquals(valorActual, valorEsperado);

//assertEquals 3 parametros: identificador del test (si falla, más fácil de saber que test es)

Assert.assertEquals(valorActual, valorEsperado, “Comprobar título correcto”);

//assertNotEquals: compara si dos valores NO son iguales o no (true si son distintos)

Assert.assertNotEquals(valorActual, valorNoEsperado);

//assertTrue: comprobar si una condición devuelve true

Assert.assertTrue(elemento.isEnabled(), “El botón esta activado”);

//assertFalse: comprobar si una condición devuelve false

Assert.assertFalse(elemento.isEnabled(), “El botón esta inactivo”);

//assertNull: comprobar si un objeto es null

Assert.assertNull(elemento, “El elemento NO existe”);

//assertNotNull: comprobar si un objeto NO es null

Assert.assertNotNull(elemento, “El elemento existe”);

## Hard vs Soft Assertions

Los asserts vistos el en apartado anterior son los llamados Hard Assertions. Este tipo de asserts lo que hacen es, como ya se ha dicho, dejar de ejecutar en cuanto un test no dé el resultado esperado.

Hay veces pero, que queremos que si una condición no pasa, el test siga ejecutándose, ya que quizás no sea una comprobación que comprometa el resto del código. En estas ocasiones se tendrán que utilizar Soft Assertions.

Lo que tendremos que hacer es crear un objeto SoftAssert y cuando queramos hacer el assert de este tipo lo llamaremos desde este nuevo objeto, no desde la clase, como hacemos con los hard. De esta forma podremos utilizar las funciones que ya hemos visto sin que el test se pare al fallar.

//creamos el objeto

SoftAssert sa = new SoftAssert();

//utilizamos la clase para que sea soft

sa.assertEquals(valorActual, valorEsperado);

## Anotaciones BeforeMethod y AfterMethod

Hay partes del código que se repiten en distintos test, sobre todo pasa con el inicio (abrir navegador, cargar página…) y final (cerrar navegador…). Hay una forma para optimizar el código: utilizando las anotaciones BeforeMethod (haciendo que la función se ejecute antes de cada @Test) y la AfterMethod (que se ejecuta al terminar).

Así pues, se crea una función void nombrada setUp() con la anotación @BeforeMethod, y dentro las sentencias de inicio.

@BeforeMethod

public void setUp() {

//codigo: abrir navegador, cargar página…

System.setProperty(“webdriver.chrome.driver”, “./src/test/resources/WebDriver/chromedriver.exe”);

driver = new ChromeDriver();

driver.manage().window().maximize();

driver.get(“https://........”);

}

Y lo mismo con @AfterMethod, con una función void llamada tearDown().

@AfterMethod

public void tearDown() {

//codigo: cerrar navegador…

driver.close();

}

## Anotaciones BeforeClass y AfterClass

Las anotaciones @BeforeClass y @AfetrClass funcionan de forma similar que las anotaciones anteriores, per en lugar de ejecutarse antes de los métodos, se ejecutan antes que la clase en sí (primero BeforeClass – BeforeMethod – Test1 – AfterMethod – BeforeMethod – Test2 – AfterMethod – AfterClass). Se tienen que importar de test.Annotations, no de junit.

@BeforeClass

public void beforeClassSetUp() {

//codigo

}

@AfterClass

public void afterClassSetUp() {

//codigo

}

## Gestión de test utilizando archivos XML

Utiliza la extensión TestNG para ver la estructura (supongo que sin extensión funciona igual).

Para ejecutar todos los test, se puede crear un archivo XML y desde allí indicar cuales son los que queremos ejecutar (se ejecutaran consecutivamente). Para hacerlo, seguimos los pasos:

* Botón derecho a la carpeta del proyecto
* New 🡪 File
* Introducimos el nombre: driver1.xml (por ejemplo)
* Finish

Una vez se nos crea el archivo, empezamos a escribir la estructura. La estructura será la misma que aparece en la pestaña de TestNG (“Results of running class…”), pero con etiquetas xml.

<suite name=”Suite test name”>

<test name=”Test name”>

<classes>

<class name=”ruta de la clase”/> //ej.: com.p1.tests.ClassName

<class name=”ruta de la 2ª clase”/>

//…

</classes>

</test>

//podriamos repetir otra ver el tag test con más clases para añadirlas

</suite>

## Añadir y excluir métodos a ejecutar en XML

Pongamos el caso que de una clase no queremos incluir todos los test para que se ejecuten en el XML, ya sea escogiendo los que queremos o excluyendo los que no.

<suite name=”Suite test name”>

<test name=”Test name”>

<classes>

<class name=”ruta de la clase” >

<methods> //incluir

<include name=”nombre\_metodo”></include>

//…

</methods>

</class>

<class name=”ruta de la 2ª clase”/>

<methods> //excluir

<exclude name=”nombre\_metodo”></ exclude>

//…

</methods>

</class>

</classes>

</test>

//podriamos repetir otra ver el tag test con más clases para añadirlas

</suite>

## Anotaciones BeforeTest y AfterTest

Si nos fijamos en la estructura XML, el método @BeforeTest se ejecutaría antes del <tag> test, y el @AfterTest después del cierre. Se ejecutará tantas veces como <test>.

@BeforeTest

public void beforeTestSetUp() {

//codigo

}

@AfterTest

public void afterTestSetUp() {

//codigo

}

## Anotaciones BeforeSuite y AfterSuite

Si nos fijamos en la estructura XML, el método @BeforeSuite se ejecutaría antes del <tag> test, y el @AfterSuite después del cierre. De esta forma, solo se ejecutará una vez cada test: uno al inicio y el otro al final.

@BeforeSuite

public void beforeSuiteSetUp() {

//codigo

}

@AfterSuite

public void afterSuiteSetUp() {

//codigo

}

# Sección 31: Page Object y Page Factory

Para un mejor mantenimiento y optimización, conviene modular el contenido del programa, así como el de los test. Crear clases con los elementos de las paginas facilitará el trabajo.

Mejor ir paso por paso para entender qué de está haciendo. Hay cosas que irán cambiando de localización a medida que vayamos modularizando el proyecto.

## Page Factory

Page Factory es una clase creada en base a una página. Por ejemplo, a la página de login tendríamos los elementos de input de usuario, contraseña y el botón de login. Estas serien las variables de la clase LoginPage, con el constructor (lo veremos en el punto [Inicializando los elementos en Page Factory](#_Inicializando_los_elementos)).

Estas clases las crearemos en el package src/main/java y dentro en com.proyecto.**elements** (no a la test).

public class LoginPage {

//variables 🡪 elementos de la pagina

//se puede escribir todo en línea o separado

@FindBy(id=”name”) //puedes buscar con lo que necesites: xpath, css…

public WebElement inputName;

@FindBy(id=”pass”)

public WebElement inputPassword;

@FindBy(css=”button.login”)

public WebElement btnLogin;

//…

}

Tener en cuenta que tenemos que hacer tantas clases como páginas para hacer los test correctamente.

## Elements Collection en Page Factory

Es tan fácil como en vez de WebElements, que sea una lista de WebElements, declarando de la misma forma que hemos visto en el punto anterior.

//un ejemplo de un select con sus opciones

@FindBy(css=”select#e1-1>option”) public List <WebElement> slctOptions;

## Inicializando los elementos en Page Factory

Para inicializar y utilizar esta clase en nuestros test, tendremos que:

* Primero de todo, es muy importante hacer los elementos public, de esta forma podremos acceder y trabajar con ellos
* Declarar un WebDriver para poder trabajar
* Constructor para inicializar los elementos

public class LoginPageElements {

//driver

WebDriver driver;

//elementos

@FindBy(id=”name”)

public WebElement inptName;

@FindBy(id=”pass”)

public WebElement inptPassword;

@FindBy(css=”button.login”)

public WebElement btnLogin;

//constructor

public LoginPageElements(WebDriver driver) { // le pasaremos el driver (chrome)

this.driver = driver;

PageFactory.initElements(driver, this); //le pasamos el driver y la pagina en si (this), de esta forma capta todos los elementos y los inicializa

}

}

Una vez tenemos esto acabado, en nuestra clase para hacer los test podemos declarar objetos de las paginas, inicializarlos y de esta forma podremos acceder a los elementos.

Hay que tener en cuenta que **NO** lo utilizaremos de esta forma, simplemente es para entender cómo funcionan las Page Factory.

public class TestClass {

WebDriver driver;

LoginPageElements loginPage;

//si hay más clases, se declaran

@BeforeMethod

public void setUp() {

System.setProperty(“webdriver.chrome.driver”, “./src/test/resources/WebDriver/chromedriver.exe”);

driver = new ChromeDriver();

driver.manage().window().maximize();

driver.get(“https://........”);

//inicializar las paginas

loginPage = new LoginPageElements(driver);

//si hay más se inicializan de la misma forma

}

//…

}

Ahora que ya están inicializados, ya podemos utilizar sus atributos (los elementos de las paginas) para trabajar con ellos. Esto lo podemos hacer de la siguiente manera.

@Test

public void testLogin() {

~~WebElement inputName = driver.findElement(By.id(“name”));~~

WebElement inputName = loginPage.inptName;

//y si se trata de listas de elementos, igual

~~List<WebElement> selectOptions = driver.findElements(By.cssSelector(”select#e1-1>option”));~~

List<WebElement> selectOptions = loginPage.slctOptions;

//…

}

De esta forma ya funcionaría, pero la forma correcta es utilizando una clase intermedia, que tendrá los métodos que utilizamos. Esta será la llamada Page Object Model.

## Page Object Model

Estas clases las crearemos en el package src/main/java y dentro en com.proyecto.**pages** (no a la test). Se creará una por cada página que utilicemos, igual que las Factory Page.

Esta nueva clase servirá para hacer el mapping, es decir, para hacer de intermediario entre clases y acciones. Esta página tendrá los métodos para realizar las acciones (clicks, rellenar inputs…) y esta será la que se relacionará directamente con las Page Factory.

Creamos una clase normal en el package indicado por cada página que utilizamos (en nuestro caso de ejemplo, para Login Page).

Así pues, si lo primero que tenemos que hacer es declarar el driver y la clase de la página, y el constructor. Una vez lo tenemos, nos fijamos en las acciones que tendremos que hacer en el test: rellenar el input de usuario y el de password, así que creamos dichos métodos, y hacemos lo mismo con el de click en el botón de login.

public class LoginPage {

//driver y clase de la página

WebDriver driver;

LoginPageElements loginPageElements;

//constructor

public void LoginPage(WebDriver driver) {

this.driver = driver;

PageFactory.initElements(driver, this);

}

//métodos

//input de nombre

public void introducirUsuario(String username) {

loginPageElements.inptName.sendKeys(username);

}

//input de contraseña

public void introducirPassword(String password) {

loginPageElements.inptPassword.sendKeys(password);

}

//botón login activo

public boolean loginButtonEnabled() {

return loginPageElements.btnLogin.isEnabled();

}

//click botón login

public void clickLoginButton() {

loginPageElements.btnLogin.click();

}

}

Así pues, en la página de test sólo tendremos que inicializar esta nueva clase, y llamar a los métodos de esta clase.

public class TestClass {

WebDriver driver;

LoginPage loginPage; //NO la de los elements, la Object Model

//si hay más clases, se declaran

@BeforeMethod

public void setUp() {

System.setProperty(“webdriver.chrome.driver”, “./src/test/resources/WebDriver/chromedriver.exe”);

driver = new ChromeDriver();

driver.manage().window().maximize();

driver.get(“https://........”);

//inicializar las paginas

loginPage = new LoginPage (driver);

//si hay más se inicializan de la misma forma

}

//…

@Test

public void testLogin() {

driver.get(“https://........”);

String name = “Admin”;

String pass = “1234”; //hay otra manera más correcta para los valores

//rellenar inputs

loginPage.introducirUsuario(name);

loginPage.introducirPassword(pass);

//comprobar si el botón está activo

Assert.assertTrue(loginPage.loginButtonEnabled(), “Verify login enabled”);

//click botón login

loginPage.clickLoginButton();

//a aquí podríamos llamar un método de otra página (ej. homePage.comprobarHomePage()) y comprobar que se haya navegado correctamente con un assertTrue.

}

//…

}

Si trabajamos con listas, recorrerla y encontrar la opción deseada se haría desde la Object Model.

## Creando la “Base Class”

Esta llamada Base Class, servirá para apartar los métodos @Before… y @After… (Class, Test, Suite, Method…), para que quede más organizado y para poder ser reutilizado en otros test sin tener que repetir código.

Así pues, en la misma carpeta donde tenemos los test (src/test/java/com/proyecto/tests) creamos una nueva clase, AbstractBaseTest. En esta añadimos los métodos before y after, inicializando el driver y las clases intermediarias (Page Object).

Todo lo que añadimos a aquí, se borra de las clases de test (quedan solo con los métodos @Test).

Si lo queremos reutilizar en distintos test, tenemos que tener en cuenta que no en todos utilizaremos la misma URL para trabajar, así que esa parte no la pondremos en esta clase.

public class AbstractBaseTest {

WebDriver driver;

LoginPage loginPage;

//si hay más clases, se declaran

@BeforeMethod //si tienes el XML para ejecutar varios, mejor @BeforeTest

public void setUp() {

System.setProperty(“webdriver.chrome.driver”, “./src/test/resources/WebDriver/chromedriver.exe”);

driver = new ChromeDriver();

driver.manage().window().maximize();

//inicializamos las clases Page Object

loginPage = new LoginPage(driver);

}

@AfterMethod //si tienes el XML para ejecutar varios, mejor @AfterTest

public void tearDown() {

driver.close();

}

}

Para tenerlo en todas las clases test, lo que haremos es que hagan *extends* de esta, así tendrán inicializado todo y podrán acceder de la misma forma a las clases intermediarias.

Ahora, lo que nos faltará es el driver.get(“https…”) para cargar la página, que por ahora lo dejaremos al principio de cada @Test.

public class TestClass extends AbstractBaseTest {

@Test

public void testLogin() {

driver.get(“https://........”);

//…

}

//…

}

## Creando la Page Library

Con la Page Library nos ayudará a resolver un par de problemas que persisten por el momento, para facilitar el trabajo como programadores:

* Para los test de una página, cada vez tenemos que ir llamando la clase intermediaria sabiendo cual es. Es mejor hacer como un “caminito de migas” para facilitar si alguien no conoce el código.
* Cuando inicializamos las clases en la Base Class, la lista va incrementando. Es mejor hacer un documento a parte para hacer el mantenimiento (tenerlas todas organizadas y controladas) e inicializarlas.

Es por esta razón que crearemos un package nuevo en scr/mani/java llamado com.proyecto.**lib**, y crearemos dos clases nuevas: una llamada AppLib y la otra PageLib.

En la PageLib, añadiremos todas las clases intermediarias inicializadas en la Base Class, así como las declaraciones de estas. También declararemos el driver, ya que lo pasaremos por constructor para inicializar.

Lo hacemos todo privado, de forma que necesitaremos también los getters para acceder a ellos.

Quedaría de la siguiente forma.

public class PageLib {

private WebDriver driver;

private LoginPage loginPage;

//si hay más clases, se declaran

public void PageLib(WebDriver driver) {

this.driver = driver;

loginPage = new LoginPage(this.driver);

//si hay más clases, se inicializan

}

//getters

public LoginPage getLoginPage() {

return this.loginPage;

}

//hacer lo mismo si hay más clases

}

En la AppLib, vamos a declarar el PageLib y un WebDriver, que lo recibirá por el constructor. Como van a ser variables privadas, añadiremos los getters.

public class AppLib {

private WebDriver driver;

private PageLib pageLib;

public void AppLib (WebDriver driver) {

this.driver = driver;

pageLib = new PageLib(this.driver);

}

//getter

public PageLib getPage() {

return this.pageLib;

}

}

Así pues, podemos borrar las declaraciones y inicializaciones de las Object Model de la Base Class y añadir nuestra nueva clase AppLib (con getter, ya que será privada), así que quedaría de la siguiente manera.

public class AbstractBaseTest {

WebDriver driver;

private AppLib app;

@BeforeTest

public void setUp() {

System.setProperty(“webdriver.chrome.driver”, “./src/test/resources/WebDriver/chromedriver.exe”);

driver = new ChromeDriver();

driver.manage().window().maximize();

app = new AppLib(driver);

}

@AfterTest

public void tearDown() {

driver.close();

}

//getter

public AppLib getApp() {

return this.app;

}

}

Así pues, inicializamos la AppLib, y esta inicializa la PageLib con todas las clases Object Model.

De esta forma, desde la clase de pruebas, tendremos que llamar a los métodos de la siguiente forma, que nos irá dando pistas a medida que vayamos escribiendo. Queda más largo per es más claro.

//…

@Test

public void testLogin() {

driver.get(“https://........”);

String name = “Admin”;

String pass = “1234”;

getApp().getPageLib().getLoginPage().introducirUsuario(name);

getApp().getPageLib().getLoginPage().introducirPassword(pass);

//comprobar si el botón está activo

Assert.assertTrue(getApp().getPageLib().getLoginPage().loginButtonEnabled(), “Verify login enabled”);

//click botón login

getApp().getPageLib().getLoginPage().clickLoginButton();

}

//…

}

## Flow Library

Se trata de una clase que contiene acciones que vamos a pedir al driver, como la URL, título de la página…

La vamos a crear en scr/mani/java/com/proyecto/**lib**, igual que AppLib y PageLib. Le vamos a añadir un WebDriver, que se lo vamos a pasar mediante el contructor. A continuación, añadiremos los métodos.

public class FlowLib { //el nombre puede cambiar, es un standard

WebDriver driver;

public FlowLib(WebDriver driver) {

this.driver = driver;

}

//ir a la pagina

public void navigateToUrl(String url) {

this.driver.get(url);

}

//obtener la pagina actual

public String getCurrentPageUrl() {

return this.driver.getCurrentUrl();

}

//tutulo de la pagina

public String getCurrentPageTitle() {

return this.driver.getTitle();

}

//…

}

También tendremos que añadir esta nueva clase a la clase AppLib, así lo inicializaremos todo junto.

public class AppLib {

private WebDriver driver;

private PageLib pageLib;

private FlowLib flowLib;

public void AppLib (WebDriver driver) {

this.driver = driver;

pageLib = new PageLib(this.driver);

flowLib = new FlowLib(this.driver);

}

//getter

public PageLib getPage() {

return this.pageLib;

}

public FlowLib getFlow() {

return this.flowLib;

}

}

Así pues, en la clase de los test tendremos que llamarlo de la misma forma que en el punto anterior.

//…

@Test

public void testLogin() {

getApp().getFlow().navigateToUrl (“https://........”);

//…

}

//…

}

Ultima sección es una ampliación de acciones con el TestNG y el XML………………………….

# Sección final: ampliación

## Resumen packages

Estructura actual de packages y clases:

* **Page Factory**: clase con la declaración de los elementos de una página.
* **Page Object Model**: clase que contiene las acciones (métodos) que se pueden realizar a partir de los elementos de Page Factory (click, isEnabled…), con la que interactúa directamente.
* **Base class**: clase que contiene las partes comunes de distintos test (métodos @Before… y @After…, con la inicialización del driver y AppLib).
* **Page Library**: clase que inicializa todos los Page Object Model.
* **App Library**: clase que inicializa la Page Library y Flow Library.
* **Flow Library**: clase que contiene los métodos que interactúan con el driver (get(“url”), getTitle…).
* **Clase Test**: clase donde están los métodos @Test, extends de Base class.

## Otra organización de packages y clases nuevas (Yuzhi)

Organización distinta, se podría intentar complementar entre las dos formas. Subrayado encontramos elementos nuevos.

* **Main**: clase main desde donde se llaman todas las funciones para ejecutar el test (inicializando y utilizando DriverSingletonx, WebDriver, clases de las páginas…).
* **Archivos en package ‘driver’**: clases que implementan DriverStrategy y inician un determinado driver (clase por driver).
* **DriverSingleton**: contiene las los métodos compartidos de los distintos drivers, como crear instancia, get y close.
* **DriverStrategy**: es una interfaz, declara el método setStrategy().
* **DriverStrategyImplement**: contiene un método para escoger qué driver usar y lo inicializa.
* **Archivos en package ‘page’**: clase con la declaración de los elementos de una página, también contienen los métodos (= Page Factory + Model Object Factory).
* **Constants**: clase con las constantes de los test.
* **FrameworkPoperties**: clase que se utiliza para acceder a las constantes en Constants.
* **Utils**: contiene métodos varios que se pueden utilizar durante el test (screenshot, encriptar…).
* **Framework.properties**: archivo de clave-valor que Constants utiliza.
* **Test**: clase de test.