P08- Pruebas de aceptación: Selenium WebDriver

Pruebas de aceptación de aplicaciones Web

El objetivo de esta práctica es automatizar **pruebas de aceptación de pruebas emergentes funcionales** sobre una aplicación Web. Utilizaremos la librería Selenium WebDriver, desde el navegador Chrome.

Tal y como hemos explicado en clase, usaremos un proyecto Maven que contendrá únicamente nuestros drivers, puesto que no disponemos del código fuente de la aplicación sobre las que haremos las pruebas de aceptación. Al igual que en la sesión anterior, proporcionamos los casos de prueba obtenidos a partir de un escenario de la aplicación a probar. Recuerda que nuestro objetivo concreto es validar la funcionalidad del sistema.

GitHub

El trabajo de esta sesión también debes subirlo a *GitHub*. Todo el trabajo de esta práctica deberá estar en el directorio **P08-WebDriver** dentro de tu espacio de trabajo.

ChromeDriver

La librería webdriver interactúa con el navegador a través un driver. Como vamos a ejecutar nuestros tests a través de Chrome necesitamos usar el driver ***chromedriver***.

En la máquina virtual este driver se encuentra en la carpeta /home/ppss/chromedriver (también podríamos descargarlo desde: https://chromedriver.storage.googleapis.com/index.html)

Es importante que la versión del driver coincida con la versión del navegador. Por ejemplo, la versión de Chrome que tenemos instalada en la máquina virtual es la 108.0.5359.124. Puedes comprobarlo usando el siguiente comando desde el terminal:

```
> google-chrome --version
```

Hemos descargado el fichero *chromedriver_linux64.zip* de la carpeta cuya versión coincide o es lo más cercana a la versión que tenemos instalada (en nuestro caso, la 108.0.5359.71) y lo hemos descomprimido en la carpeta */home/ppss/*) **IMPORTANTE!!**: como en el fichero oculto `.profile` hemos añadido la ruta `/home/ppss/chromedriver` al `PATH`, si no quieres editar dicho fichero, tendrás que crear la carpeta chromedriver y mover el fichero a dicha carpeta.

También hemos añadido la ruta del driver en nuestro PATH, para que esté accesible desde cualquier directorio. Para ello, hemos editado el fichero oculto ***.profile*** que se encuentra en nuestro $HOME, y hemos añadido la línea (al final del fichero):

```
export PATH=$PATH:/home/ppss/chromedriver
```

El driver necesita usar la librería libconf 2.4. La hemos instalado con el comando:

```
> sudo apt install libgconf-2-4
```

Podemos probar el driver desde un terminal tecleando el nombre del ejecutable (no importa desde que carpeta lo ejecutemos, ya que hemos configurado la variable PATH en el fichero .profile). Para detener la ejecución del driver usaremos Ctrl-C):

```
> chromedriver
```

Podemos usar el driver desde nuestro código, a partir de una instancia de tipo ChromeDriver:

```
WebDriver driver = new ChromeDriver();                    (opción 1)
```

De forma alternativa, podríamos copiar el driver (fichero chromedriver), por ejemplo en la carpeta src/test/resources/drivers de nuestro proyecto maven. En ese caso, en nuestro código, usaríamos las sentencias:

```
System.setProperty("webdriver.chrome.driver",
                   "./src/test/resources/drivers/chromedriver");      (opción 2)
WebDriver driver = new ChromeDriver();
```

Ambas opciones son válidas. Con la opción 2 no tendremos que preocuparnos de si en la máquina hemos descargado o no chromeDriver, pero usaremos la opción 1 y actualizaremos el PATH del sistema a través del fichero *.profile*.

Cookies

Las *cookies* son datos almacenados en ficheros de texto en el ordenador. Cuando un servidor web envía una página a un navegador, la "conexión" termina, y el servidor "olvida" cualquier cosa sobre el usuario.

Las *cookies* se inventaron para resolver el problema de "cómo recordar información sobre el usuario", de forma que cuando un usuario visita una página web, se almacena cierta información en una cookie, que se enviará al servidor, de forma que dicho servidor sabrá que la petición proviene del mismo usuario.

Por lo tanto, las *cookies* constituyen un mecanismo para mantener el estado en una aplicación Web. Es decir, la idea es permitir que una aplicación web tenga la capacidad de interaccionar con un determinado usuario, diferenciándolo del resto. Si no mantenemos el estado, cada vez que un usuario accede a una página web, el servidor no sabrá si se trata del mismo usuario o si cada petición es de un usuario diferente. Como ejemplo: es como cuando dejamos la ropa en la tintorería, y el dependiente nos da un ticket, de forma que cuando vayamos a recogerla, sabrán que hemos ido antes a dejarla, y qué ropa venimos a recoger. Pues bien, ese ticket es similar a una cookie.

Cada cookie se asocia con una serie de propiedades: nombre, valor, dominio, ruta (path), fecha de expiración, y si segura o no.

Cuando validamos por ejemplo una aplicación de venta *on-line*, necesitaremos automatizar escenarios de prueba como hacer un pedido, ver el carrito de compra, proporcionar los datos de pago, confirmar el pedido, etc. Si no almacenamos las cookies, necesitaremos *loguearnos* en el sistema cada vez que ejecutemos cualquiera de los escenarios anteriores, lo cual incrementará el tiempo de ejecución de nuestros tests.

Una posible solución es almacenar las cookies en un fichero, y posteriormente recuperarlas y guardarlas en el navegador. De esta forma podremos "saltarnos" el proceso de login en nuestro test puesto que el navegador ya tendrá esta información.

En la carpeta **Plantillas-P08**, hemos proporcionado una clase ***Cookies***, con 3 métodos, para poder:

- almacenar en un fichero de texto (en el directorio *target*) las cookies generadas cuando nos *logueamos* en el sistema,
- leer la información sobre las cookies almacenada en el fichero y guardarla en el navegador
- imprimir por pantalla las cookies almacenadas en nuestro navegador.

Usaremos la clase *Cookies* en el tercer ejercicio de esta práctica.

Ejercicios

Vamos a implementar nuestros tests de pruebas de aceptación para una aplicación Web denominada Madison Island, a la que accederemos desde Chrome usando la url http://demo-store.seleniumacademy.com/.

Una vez en la página principal, podéis "navegar" por la aplicación para familiarizaros con ella. Opcionalmente podéis **crearos una cuenta** (anota el email y password de la nueva cuenta para recordarla y así no crear cuentas adicionales innecesarias). Debes tener en cuenta que vamos a implementar un test precisamente para crear una nueva cuenta de usuario, por lo que si os creáis una cuenta para familiarizaros con la aplicación, tendréis que crear otra diferente para poder ejecutar el test.

Para los ejercicios de esta sesión crea un proyecto maven, en la carpeta *P08-WebDriver* de tu directorio de trabajo, con groupId = **ppss**, y artifactID= **madisonIsland**.

Lo PRIMERO que tienes que hacer es configurar correctamente el pom. Debes incluir las propiedades, dependencias y plugins necesarios para implementar tests unitarios con webdriver. Recuerda que para Maven serán tests unitarios (y se ejecutarán a través del plugin surefire), pero realmente son tests de aceptación, tal y como ya hemos explicado en clase.

Si estuviésemos haciendo las pruebas de aceptación sobre código desarrollado por nosotros, en src/main tendríamos el código fuente a probar, y en src/test tendríamos tanto los tests unitarios (ejecutados con *surefire*), como el resto de tests, incluidos los de aceptación (ejecutados con *failsafe*).

Aunque como también hemos indicado en clase, no es inusual implementar los tests de aceptación en un proyecto maven propio, que sólo contenga este tipo de pruebas.

Observaciones sobre esperas implícitas y/o explícitas

Para sincronizar la carga de las páginas con la ejecución de nuestros drivers, usaremos un **wait implícito**. Recuerda que en este caso establecemos el mismo temporizador para todos los *webelements* de las páginas. Sólo se usa una vez, antes de ejecutar cada test. Este código webdriver lo incluiremos en nuestro test (aunque usemos el patrón Page Object) ya que no se verá afectado por posibles cambios de las páginas html a las que accede dicho test. El valor del temporizador será en segundos, y dependerá de la máquina en la que estéis ejecutando los tests. Podéis probar inicialmente con 5 segundos y ajustarlo a un valor mayor o menor si es necesario.

También puedes usar un **wait explícito**, en cuyo caso sólo afectará al elemento al que asociemos el temporizador.

Ejemplo de *wait* **implícito**:

```
driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
```

Ejemplo de *wait* **explícito** para una ventana de alerta:

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));
wait.until(ExpectedConditions.alertIsPresent());
```

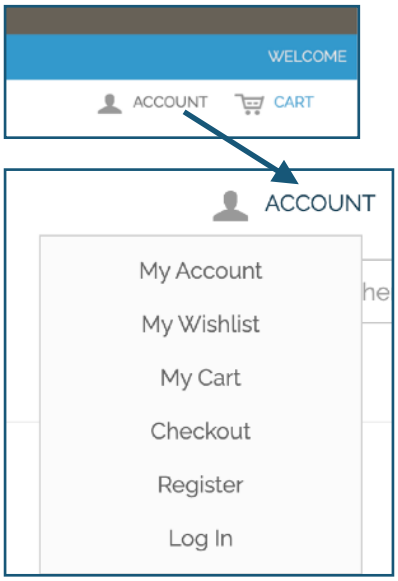
Nota adicional:

Para realizar este tipo de pruebas intenta tener el menor número de aplicaciones abiertas de forma innecesaria, ya que esto puede "ralentizar" la carga de las páginas en el navegador, dificultando así el proceso de testing (generando errores debido a que no se cargan las páginas con la suficiente rapidez).

Observaciones sobre la selección de *locators* para los *WebElements*

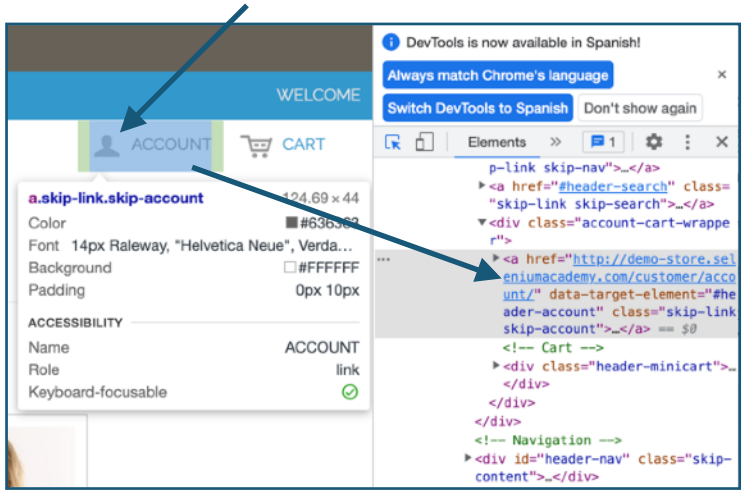
Para "localizar" cada uno de los elementos html de la página necesitamos "ver" cuál es su código usando la utilidad "inspeccionar elemento" de Chrome. Debes tener en cuenta que si vas a interaccionar con dicho elemento, debes seleccionar el código html de forma correcta.

Por ejemplo: en la página principal vemos el elemento con el texto "ACCOUNT".



Cuando hacemos "hacemos click" con botón izquierdo sobre este elemento, se muestran una serie de hiperenlaces ("My Account", "My Wishlist", "My Cart" ...).

Si inspeccionamos el elemento resaltado en azul:

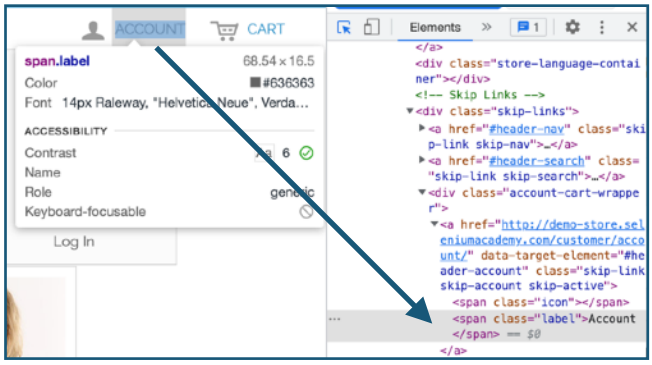


Fíjate que se trata de un hiperenlace. Pero en este caso el texto asociado son tres puntos "...", por lo que debemos elegir otro "locator". Podemos obtener, por ejemplo, el css selector seleccionando desde el menú contextual del código html "*Copy →Copy Selector*". Para averiguar el *xpath* seleccionamos "*Copy →Copy XPath*". Estas acciones copian el valor de css selector (o de *xpath)* en el portapapeles, y ya lo podemos pegar en nuestro código. Puedes comprobar que el valor de *css selector* para este elemento es:

```
#header > div > div.skip-links > div > a (1)
```

Si seleccionamos únicamente el texto "ACCOUNT" obtenemos el código de ese texto, y podemos copiar su css selector, que en este caso es:

```
#header > div > div.skip-links > div > a > span.label (2)
```



Pero sobre un texto NO podemos interaccionar!!

Por lo que si lo que queremos es hacer "click" sobre "Account" y que nos muestre la lista de hiperenlaces tendremos que usar (1) en lugar de (2).

Es muy importante que tengas claro el proceso para elegir el "locator" adecuado, lo cual nos permitirá interaccionar con el elemento html correspondiente de forma correcta.

Observaciones sobre la obtención del título de la página

La forma de obtener el título de la página es usando el método ***getTitle()*** :

`driver.getTitle().getText()` → devuelve la cadena de caracteres con el título de la página

Si intentáis buscarlo usando el método *findElement()* veréis que siempre devuelve una cadena de caracteres VACIA.

Esto ocurre porque el texto asociado a la etiqueta <title> NO es visible en la página.

Si ejecutáis: `driver.findElement(By.tagName("title")).isDisplayed()` veréis que devuelve *false*

➪ ➪ Ejercicio 1:Tests sin usar *Page Objects*

Vamos a crear el paquete ***Ejercicio1.sinPageObject***, en el que implementaremos varios casos de prueba sin usar el patrón Page Object.

Los casos de prueba a implementar son los siguientes:

- En la clase *TestCreateAccount*, crearemos el driver ***createAccount()***. Este driver sólo lo vamos a ejecutar una vez, y lo anotaremos con el *Tag "OnlyOnce"* (usaremos este *Tag* más adelante, cuando tengamos que ejecutar todos los tests de esta práctica).

  Crea una *Configuration* con el nombre "***Ejercicio1.CreateAccount***" para ejecutar este driver (usa la opción *-Dtest*). Recuerda que todas las *Configurations* debes guardarlas en la carpeta intellij-configurations, igual que hemos hecho en prácticas anteriores.
- En la clase *TestLogin*, crearemos los drivers ***loginOK()*** y ***loginFailed()***.

  Debes crear una "*Configuration*" con el nombre "***Ejercicio1.sinPO.TestLogin***" que ejecute los drivers de esta clase (usa la opción *-Dtest*)

A continuación describimos cada uno de los tests:

createAccount()

1. Verificamos que el título de la página de inicio es el correcto ("Madison Island")
2. Seleccionamos Account, y a continuación seleccionamos el hiperenlace Login
3. Verificamos que el titulo de la página es el correcto ("Customer Login")
4. Seleccionamos el botón "Create Account"
5. Verificamos que estamos en la página correcta usando el título de la misma ("Create new Customer Account")
6. Rellenamos los campos con los datos de la cuenta excepto el campo "Confirmation" (cada uno de vosotros elegirá unos valores diferentes), y enviamos los datos del formulario. Nota: el valor del password debe tener 6 o más caracteres.
7. Verificamos que nos aparece el mensaje "This is a required field." debajo del campo que hemos dejado vacío
8. Rellenamos el campo que nos falta y volvemos a enviar los datos del formulario.
9. Verificamos que estamos en la página correcta usando su título ("My Account").

loginFailed()

1. Verificamos que el título de la página de inicio es el correcto ("Madison Island")
2. Seleccionamos Account, y a continuación seleccionamos el hiperenlace Login
3. Verificamos que el titulo de la página es el correcto ("Customer Login")
4. Rellenamos los campos con el email de la cuenta que hemos creado en el driver createAccount() , y con un password incorrecto. Enviamos el formulario
5. Verificamos que nos aparece el mensaje "Invalid login or password

loginOK()

1. Verificamos que el título de la página de inicio es el correcto ("Madison Island")
2. Seleccionamos Account, y a continuación seleccionamos el hiperenlace Login
3. Verificamos que el titulo de la página es el correcto ("Customer Login")
4. Rellenamos el campo email con el email de la cuenta que hemos creado en el driver createAccount() y enviamos el formulario
5. Verificamos que nos aparece el mensaje "This is a required field" debajo del campo que hemos dejado vacío
6. Rellenamos el campo con la contraseña y volvemos a enviar los datos del formulario.
7. Verificamos que estamos en la página correcta usando su título ("My Account").

OBSERVACIONES:

- Debes usar las anotaciones @BeforeEach y @AfterEach (siempre debes cerrar el navegador después de cada test).
- Recuerda que para no "perder" las "configurations" al subir tu proyecto a GitHub, tendrás que marcar *Store as Project File* (en la parte superior derecha de la ventana), y guardarlo en la carpeta

  madisonIsland/intellij-configurations/

  (esto tendrás que hacerlo para CADA "configuration")
- Con respecto a los títulos de las páginas, podéis verificarlos usando el texto "literal" o también podéis hacer la verificación comprobando que el título contiene parte del texto.

➪ ➪ Ejercicio 2:Tests usando *Page Objects*

Vamos a crear el paquete ***ejercicio2.conPO***, en el que implementaremos los dos casos de prueba del ejercicio anterior usando el patrón Page Object. No usaremos la clase *PageFactory*. Recuerda que, además de los drivers, tienes que implementar la/s Page Object de las que dependen los tests, y que éstas estarán en src/main, tal y como hemos explicado en clase.

Los casos de prueba se implementarán en una clase ***TestLogin2***, y los nombres de cada driver serán: ***test_Login_Correct()*** y ***test_Login_Incorrect()*** que se corresponden con los tests *loginOK()* y *loginFailed()* anteriores, respectivamente.

Nota: en el test ***test_Login_Correct()*** introducimos las credenciales correctas sin dejar el campo de la contraseña vacío. Es decir, en el paso 6 rellenamos el email (sin enviar los datos del formulario, y omitiremos el paso 7.

Necesitarás implementar tres *page objects*: *HomePage, CustomerLoginPage y* y *MyAccountPage.*

- La clase ***HomePage*** interactúa con dos *WebElement*. Debes tener en cuenta que NO podrás inicializar los dos en el constructor de dicha clase, ya que el hiperenlace para realizar el "login" no estará visible hasta después de seleccionar la opción "Account".
- La clase ***CustomerLoginPage*** interactúa con cuatro WebElement (los dos cuadros de texto donde introduciremos las credenciales del usuario, el botón para enviar el formulario, y el mensaje de error). Observa que el mensaje de error NO podremos "localizarlo" en el constructor de la clase por la misma razón de antes.

  Puedes usar dos métodos: uno para introducir los datos correctos, y que nos llevará (devolverá) a la siguiente página (MyAccountPage); y un segundo método para introducir los datos incorrectos, en cuyo caso devolveremos el mensaje de error.
- La clase ***MyAccountPage*** no interactúa con ningún WebElement. Unicamente tendrá el constructor y un método para devolver el título de la página.

Para ejecutar los tests crea un elemento "configuration" con el nombre "***Ejercicio2.TestLogin2***" que ejecute los drivers de esta clase (usa la opción *-Dtest*).

Recuerda marcar "*Store as Project File*" para guardar la configuración en la *carpeta src/test/resources/ configurations*.

➪ ➪ Ejercicio 3: Test usando *Page Objects*, *PageFactory* y *Cookies*

Vamos a crear el paquete ***ejercicio3.conPOyPFact***, en el que implementaremos drivers para automatizar casos de prueba usando el patrón *Page Object*, junto con la clase *PageFactory*.

Necesitarás implementar las Page Objects necesarias en src/main (en el mismo paquete que nuestros drivers).

Usa el nombre ***TestShoes*** para la clase que contiene los tests. Dado que vamos a probar acciones asumiendo que ya estamos logueados en la tienda, tendremos que acceder al sistema y guardar las Cookies en un fichero. Esta acción la realizaremos UNA SOLA vez y ANTES de cualquier test. Para ello debes usar el método correspondiente de la clase Cookies que se proporciona como plantilla.

Por otro lado, ANTES de ejecutar CADA test tendremos que:

- Cargar las *cookies* en el navegador desde fichero que hemos creado y que habremos guardado en la carpeta *target.*
- Para cada uno de los tests, por lo tanto, accederemos directamente a la página de la cuenta de usuario: *http://demo-store.seleniumacademy.com/customer/account/.*
- Configuraremos las opciones del navegador para poder ejecutar los tests sin abrir *Chrome*. Para ello usaremos la *variable* "***chromeHeadless***", en la configuración del plugin *webdriver*, a la que podremos asignarle el valor "true" a través de la *property* ***headless.value,*** que definiremos en el *pom* (lo tenéis explicado al final del ejercicio).

Vamos a implementar un único driver con nombre ***compareShoes()***:

MyAccountPage

compareShoes()

1. Verificamos que el título de la página es el correcto ("My Account")
2. Seleccionamos Accessories → Shoes (ver observaciones sobre este paso)

ShoesPage

3. Verificamos que el titulo de la página es el correcto ("Shoes - Accessories")
4. Seleccionamos dos zapatos para compararlos (pulsando sobre "Add to Compare". En concreto queremos seleccionar los dos últimos (ver imagen y las observaciones sobre este paso)
5. Seleccionamos el botón "COMPARE" (ver observaciones sobre este paso)

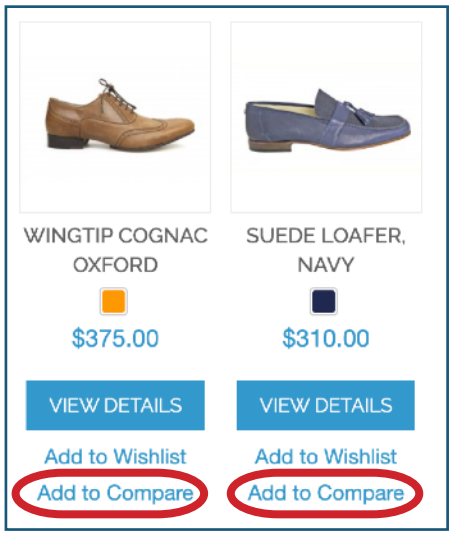
Products

6. Verificamos que estamos en la página correcta usando el título de la misma ("Products Comparison List ...")
7. Cerramos la ventana con la comparativa de productos (ver las observaciones sobre este paso)

ShoesPage

8. Verificamos que estamos de nuevo en la ventana ("Shoes - Accessories")
9. Borramos la la comparativa (hiperenlace "Clear All"), y verificamos que nos aparece una ventana de alerta con el mensaje "Do you like to remove all products from your comparison?" (ver las observaciones sobre este paso)
10. Verificamos que en la página aparece el mensaje: "the comparison list was cleared"

zapatos a comparar:



Los valores de css de los dos hiperenlaces "Add to Compare" de los zapatos a seleccionar se diferencian por la posición que ocupan en una lista de elementos (etiquetas <li>), en concreto son las posiciones 5 (para Wingtip Cognac Oxford) y 6 (para Suede Loafer Navy).

Recuerda que **TODO** el código webdriver de nuestro test que depende del código html estará en las Page Object correspondientes.

Observaciones a tener en cuenta sobre el código webdriver (estará en las **Page Objects!!**):

PASO 2: El elemento Accessories muestra el menú con los hiperenlaces correspondientes, uno de los cuales es "Shoes". Observa que para que aparezca el menú NO hay que "clickar" con botón izquierdo sino que hay que MOVER el ratón hasta situarlo sobre el elemento (sin necesidad de hacer click). Si movemos el ratón y lo alejamos del elemento, vemos que inmediatamente "desaparece" dicho menú.

Por lo tanto, para mostrar el menú y hacer "click" sobre el hiperenlace "Shoes" tendremos primero que "mover" el puntero del ratón a dicho elemento usando la clase Actions:

```
//movemos el ratón sobre el elemento "accessories"
Actions builder = new Actions(driver);
builder.moveToElement(accessoriesMenu);
builder.perform();
//ahora serán visibles los hiperenlaces, y podremos clickar sobre "shoes"
```

En el código anterior, la variable ***accesoriesMenu*** es uno de los atributos WebElement de la Page Object asociada a la página con el título "My Account".

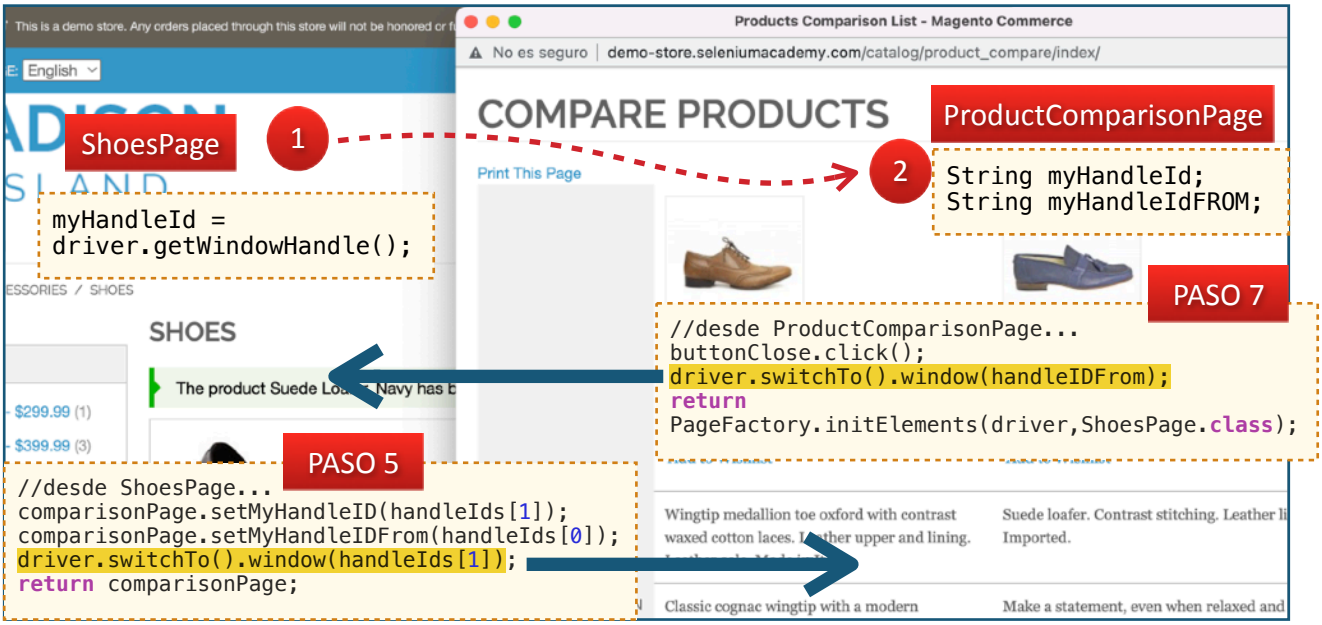
PASO 4: Para seleccionar los hiperenlaces "Add to Compare" posiblemente necesitarás hacer "scroll" de la página para que estén visibles. Para evitar redundancias de código podemos crear un método cuyo parámetro sea la posición del elemento <li> correspondiente y que nos permitirá discriminar el hiperenlace elegido (zapato a comparar).

A continuación mostramos un ejemplo de posible código:

```
public void selectShoeToCompare(int number) {
  JavascriptExecutor jse = (JavascriptExecutor) driver;
  switch(number) {
     case 5: jse.executeScript("arguments[0].scrollIntoView();", wingtipShoe);
             wingtipShoe.click();
             break;
     case 6: jse.executeScript("arguments[0].scrollIntoView();", suedeShoe);
             suedeShoe.click();
             break;
  }
}
```

PASO 5: Desde la página con título "Shoes - Accesssories", pulsamos sobre el botón "COMPARE", y esto provoca que se abra una nueva ventana (con título "Products Comparison List") en la que se nos muestra la información de los dos productos para que podamos compararlos y ver las diferencias entre ellos.

Necesitaremos conocer el "manejador" de ambas ventanas para poder "movernos" entre ellas, por lo que deberás incluir un atributo de tipo String en cada una de las Page Object para "guardar" dicho valor. Además, en la ventana "Products Comparison List" necesitas almacenar también el manejador de la ventana "Shoes - Accessories".



Cuando hacemos click sobre el botón "COMPARE", el "foco" permanece en la ventana ShoesPage, y se crea un nuevo "manejador" asociado a la nueva ventana ProductComparisonPage.

Puedes comprobarlo con el siguiente código (desde ShoesPage):

```
//obtenemos el manejador de la ventana ShoesPage
String myHandleId = driver.getWindowHandle();
//pulsamos sobre el botón para hacer la comparación
buttonCompare.click(); //se abre una nueva ventana

//el handleID de la nueva ventana se añade al conjunto de manejadores del navegador
Set<String> setIds =  driver.getWindowHandles();
String[] handleIds = setIds.toArray(new String[setIds.size()]);
System.out.println("ID 0: "+handleIds[0]); //manejador de la ventana ShoesPage
System.out.println("ID 1: "+handleIds[1]); //manejador de la venana ProductComparisonPage
```

PASO 7: Desde la página con título "Products Comparison List", pulsamos sobre el botón "CLOSE WINDOW" para cerrar la ventana y debemos volver a la ventana anterior (título "Shoes - Accesssories") usando el manejador de dicha ventana (ver el código con la etiqueta "Paso 7" de la imagen que acabamos de comentar).

PASO 9: Después de hacer "click" en el hiperenlace "Clear All" nos aparecerá una ventana de alerta que deberemos cerrar para poder continuar con la ejecución. Además, deberemos recuperar el mensaje de texto asociado a dicha alerta para poder verificar que dicha alerta ha "aparecido".

Para gestionar las "alertas" necesitas usar la clase Alert, que representa un elemento Alert Box..

Un Alert Box no es más que un pequeño “recuadro” que aparece en la pantalla que proporciona algún tipo de información o aviso sobre alguna operación que intentamos realizar, pudiendo solicitarnos algún tipo de permiso para realizar dicha operación

A continuación mostramos un ejemplo de algunos métodos que podemos usar:

```
//Operaciones sobre ventanas de alerta
//cambiamos el foco a la ventana de alerta
Alert alert = driver.switchTo().alert();
//podemos obtener el mensaje de la ventana
String mensaje = alert.getText();
//podemos pulsar sobre el botón OK (si lo hubiese)
alert.accept();
//podemos pulsar sobre el botón Cancel (si lo hubiese)
alert.dismiss();
//podemos teclear algún texto (si procede)
alert.sendKeys("user");
```

Para ejecutar el test crea una *Configuration* con el nombre "***Ejercicio3***" que ejecute los drivers de esta clase (usa la opción *-Dtest*). En este caso sólo hemos creado un test, pero la idea es que la clase TestShoes contenga más drivers usando escenarios a partir de la página Shoes.

Recuerda marcar "*Store as Project File*" para guardar la configuración en la *carpeta src/test/resources/configurations*.

Modo "*headless*"

Podemos ejecutar los tests sin necesidad de mostrar el navegador, con lo cual ahorraremos tiempo de ejecución. Para ello tenemos que configurar el driver en modo *headless*. La idea es cambiar el modo de ejecución de los tests desde el proceso de construcción. Para ello tendremos que:

- Modificar la configuración del plugin ***surefire***. Vamos a añadir una "propiedad" en la etiqueta <systemPropertyVariables>. Llamaremos a esta propiedad "***chromeHeadless***" (es una elección personal, podemos elegir cualquier nombre que consideremos oportuno). El valor de la propiedad "*chromeHeadless*" tendrá como valor el de la *<property>* ***headless.value*** (de nuevo este nombre depende de nuestra elección personal).

```
<properties>
    ...
    <headless.value>false</headless.value>
</properties>

<plugin>
    ...
    <artifactId>maven-surefire-plugin</artifactId>
    ...
    <configuration>
        <systemPropertyVariables>
            <chromeHeadless>${headless.value}</chromeHeadless>
        </systemPropertyVariables>
    </configuration>
</plugin>
```

- En el código del driver (en el método anotado con *@BeforeEach*) creamos la instancia de *ChromeDriver* activando el modo *headless.* Para ello usaremos el valor de la propiedad *chromeHeadless* que hemos definido en la configuración del plugin *surefire*.

```
ChromeOptions chromeOptions = new ChromeOptions();
//recuperamos el valor de la propiedad chromeHeadless definida en surefire
boolean headless = Boolean.parseBoolean(System.getProperty("chromeHeadless"));
//el método setHeadless() cambia la configuración de Chrome a modo headless
chromeOptions.setHeadless(headless);
//ahora creamos una instancia de CromeDriver a partir de chromeOptions
driver = new ChromeDriver(chromeOptions);
```

- Ahora podemos activar el modo headless usando el valor "true" para la *property headless.value* de nuestro *pom*.

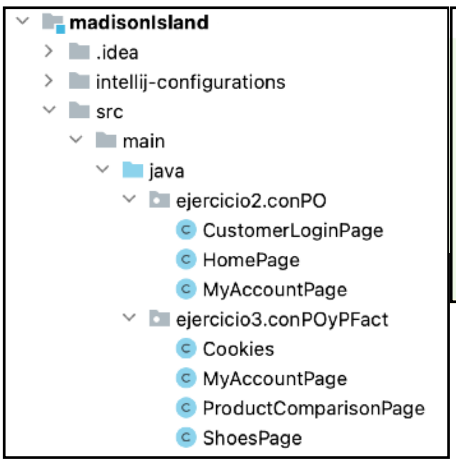
```
mvn test ... -Dheadless.value=true
```

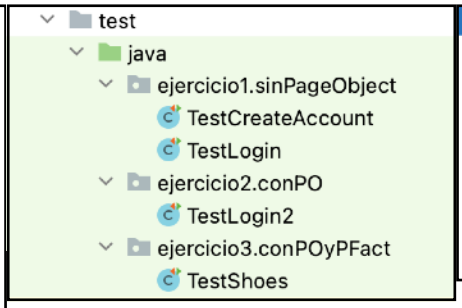
Finalmente crea tres *Configurations* más:

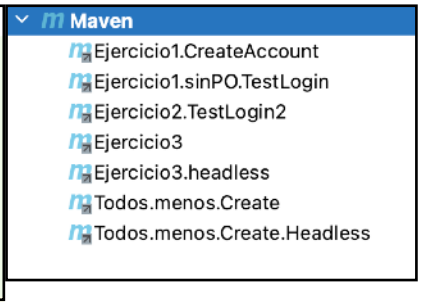
- Nombre: "***Ejercicio3.headless***", en el que ejecutarás los drivers de la clase TestShoes pero en modo "headless".

- Nombre: "***Todos.menos.Create***", para ejecutar todos los tests de validación menos los tests con la etiqueta "OnlyOnce".
- Nombre: "***Todos.menos.Create.Headless***", para ejecutar todos los tests de validación menos los tests con la etiqueta "OnlyOnce" pero en modo *Headless*.

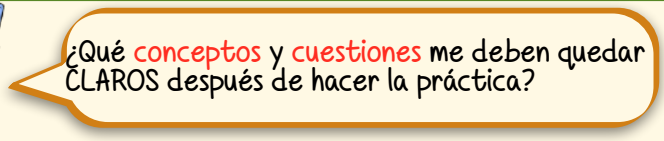
Mostramos capturas de pantalla del proyecto maven y *configurations* para esta práctica:







Resumen



AUTOMATIZACIÓN DE PRUEBAS DE ACEPTACIÓN

- Consideraremos el caso de que nuestra SUT sea una aplicación web, por lo que usaremos webdriver para implementar y ejecutar los casos de prueba. La aplicación estará desplegada en un servidor web o un servidor de aplicaciones, y nuestros tests accederán a nuestro SUT a través de webdriver, que será nuestro intermediario con el navegador (en este caso usaremos Chrome, junto con el driver correspondiente)
- Si usamos webdriver directamente en nuestros tests, éstos dependerán del código html de las páginas web de la aplicación a probar, y por lo tanto serán muy "sensibles" a cualquier cambio en el código html. Una forma de independizarlos de la interfaz web es usar el patrón PAGE OBJECT,, de forma que nuestros tests NO contendrán código webdriver, independizándolos del código html. El código webdriver estará en las page objects que son las clases que dependen directamente del código html, a su vez nuestros tests dependerán de las page objects.
- Junto con el patrón Page Object, usaremos la clase PageFactory para crear e inicializar los atributos de una Page Object. Los valores de atributos se inyectan en el test mediante la anotación @Findby, y a través del localizador correspondiente. Esta inyección se realiza de forma "lazy", es decir, los valores se inyectan justo antes de ser usados.
- Con webdriver podemos manejar las alertas generadas por la aplicación a probar, introducir esperas (implícitas y explícitas), realizar scroll en la pantalla del navegador, agrupar acciones sobre los elementos, movernos entre ventanas, y manejar cookies, entre otras cosas.
- El manejo de las cookies del navegador nos será útil para acortar la duración de los tests, ya que podremos evitar loguearnos en la aplicación para probar determinados escenarios.
- También podremos acortar los tiempos de ejecución de nuestros tests si los ejecutamos en modo headless. Podemos decidir si vamos a ejecutar o no nuestros tests en modo headless aprovechando la capacidad del plugin surefire y/o failsafe, de hacer llegar a nuestros tests ciertas propiedades definidas por el usuario. De esta forma podremos, cuando lancemos el proceso de construcción, ejecutar los tests en ambos modos sin modificar el código.