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 herramienta Selenium WebDriver, desde el navegador Firefox.

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, proporcionaremos los casos de prueba obtenidos a partir de un requerimiento o escenario de la aplicación a probar. Recuerda que nuestro objetivo concreto es validar la funcionalidad del sistema.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P08-WebDriver** dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2020-Gx-apellido1-apellido2.

Recuerda que si trabajas desde los ordenadores del laboratorio primero deberás configurar git y clonar tu repositorio de Bitbucket.

Ejercicios

Vamos a implementar nuestros tests de pruebas de aceptación para una aplicación Web denominada Guru Bank, a la que accederemos con Firefox usando la url http://demo.guru99.com/V5.

Una vez en la página principal, debéis **crearos una cuenta** de administrador para usar la aplicación (siguiendo las instrucciones que se muestran en la parte inferior de dicha página). Recuerda que, tal y como se indica, el usuario y password asignados tienen una validez de 20 días, por lo que si necesitas usar la aplicación pasado ese tiempo deberás crear otra cuenta (y actualizar dicha información en el código de tus tests).

Para los ejercicios de esta sesión crea, en la carpeta *ppss-2020-Gx-apellido1-apellido2/P08-WebDriver/* un proyecto maven, con groupId = **ppss**, y artifactID= **guru99Bank**.

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, pero realmente son tests de aceptación, tal y como ya hemos explicado en clase.

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. Independientemente de que usemos o no el patrón page object. Este código webdriver lo incluiremos en nuestro test (aunque usemos el patrón Page Obect) 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(10, TimeUnit.SECONDS);
```

Ejemplo de wait explícito para una ventana de alerta:

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

Puedes consultar aquí un ejemplo de uso de *Alert Box* con WebDriver:

https://www.guru99.com/implicit-explicit-waits-selenium.html

➪ ➪ Ejercicio 1:Tests Login

Vamos a crear el paquete *ejercicio1.sinPageObject*, en el que implementaremos dos casos de prueba sin usar el patrón Page Object.

Los casos de prueba a implementar son los siguientes (en una clase **TestLogin**):

- Un primer caso de prueba (con el nombre ***test_Login_Correct()***) en el que accederemos a la aplicación bancaria con un nombre de usuario y contraseña correctos y en pantalla aparecerá el correspondiente mensaje de bienvenida (podéis usar el mensaje completo o parte de él para verificar el resultado).
- Un segundo caso de prueba, al que llamaremos ***test_Login_Incorrect()*** en el que accederemos a la aplicación bancaria con un nombre de usuario y contraseña incorrectos, y debemos verificar que se muestra el mensaje "User is not valid" en un *alert Box.*

  Para implementar el driver necesitas usar la clase Alert, que representa un elemento *Alert Box*..

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:

También puedes consultar aquí un ejemplo de uso de *Alert Box* con WebDriver:

```
//Operaciones sobre ventanas de alerta
//cambiamos el foco a la ventana de alerta
Alert alert = driver.switchTo().alert();
//podemos mostrar 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");
```

https://www.guru99.com/alert-popup-handling-selenium.html

Puedes crear un elemento de configuración Maven desde IntelliJ con el comando:

```
mvn test –Dtest="ejercicio1.sinPageObject.TestLogin
```

➪ ➪ Ejercicio 2:Tests Login 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 haremos uso de 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 TestLogin, y los nombres de cada caso de prueba serán los mismos que en el ejercicio anterior (test_Login_Correct()y test_Login_Incorrect().

En este caso necesitaréis implementar dos *page objects*: *LoginPage* y *ManagerPage.*

Puedes crear un elemento de configuración Maven desde IntelliJ con el comando:

```
mvn test –Dtest="ejercicio2.conPO.TestLogin
```

➪ ➪ Ejercicio 3: Tests NewClient

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

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

Los casos de prueba que vamos a implementar son los siguientes (en la clase TestNewClient):

- ***testTestNewClientOk()***: entramos en el banco con nuestros credenciales y damos de alta a un nuevo cliente.

IMPORTANTE!!

Cuando creamos un nuevo cliente, la aplicación le asigna de forma automática un identificador único. Dicho valor será necesario para realizar cualquier operación con el cliente (por ejemplo editar sus datos, borrarlo...). Por lo tanto, deberás almacenar dicho valor (como un atributo en la clase que contiene los tests) para poder consultarlo desde todos los tests que necesiten trabajar con dicho cliente. También deberías imprimir dicho valor por pantalla por si el código de test contiene errores y necesitas deshacer manualmente la operación de creación del nuevo cliente y poder repetir la ejecución del test.

Los datos concretos para el nuevo cliente, pueden ser éstos:

- Nombre del cliente: tu login del correo electrónico de la ua
- Género: “m” (o “f”)
- Fecha de nacimiento: tu fecha de nacimiento en formato “aaaa-mm-dd”
- Dirección: "Calle x"
- Ciudad: ”Alicante”, (para el campo State puedes poner el mismo)
- Pin: “123456" (puedes poner cualquier otra secuencia de 6 dígitos)
- Número de móvil: “99999999"
- e-mail: tu email institucional
- password: “123456”, (o cualquier otra secuencia de dígitos)

Una vez introducidos los datos pulsaremos sobre el botón “Submit” (asegúrate de que haces scroll de la pantalla para que el botón submit esté visible).

El resultado del test será “pass” si hemos conseguido crear el nuevo cliente. Para ello podemos comprobar que en la nueva página aparece el texto (o una parte de él) “Customer Registered Successfully!!!”.

Cómo realizar “scroll” de la ventana del navegador

Cuando vamos a introducir los datos del nuevo cliente que queremos crear, observa que no “caben” en la pantalla todos los cuadros de texto, y que para introducir todos los datos, necesitaremos hacer “scroll” en la ventana del navegador (de no hacerlo así obtendremos un error, puesto que no podemos interaccionar con elementos de la página que no están visibles).

Para ello necesitaremos usar el método javascript ***scrollIntoView()***. A continuación mostramos un ejemplo en el que hacemos scroll del contenido de la ventana hasta que sea visible el webelement al que hemos denominado “submit” y que representa el botón para enviar el formulario con los datos del cliente.

```
...
//Si no hacemos esto, el botón submit no está a la vista y
//no podremos enviar los datos del formulario
JavascriptExecutor js = (JavascriptExecutor) driver;
//This will scroll the page till the element is found
js.executeScript("arguments[0].scrollIntoView();", submit);
//ahora ya está visible el botón en la página y ya podemos hacer click sobre él
submit.click();
...
```

Puedes consultar aquí un ejemplo de uso de *Scroll* con WebDriver:
https://www.guru99.com/scroll-up-down-selenium-webdriver.html

Otras observaciones a tener en cuenta:

- Recuerda que si usamos el patrón Page Object nuestros tests NO deben contener código Webdriver.
- Recuerda que debes verificar, cada vez que cambiemos de página, que estamos en la página correcta, para ello puedes utilizar el título de la página o alguna información de la misma, por ejemplo el login del administrador (en el caso de la página resultante de hacer login).
- El campo e-mail del cliente es único para cualquier cliente. Si intentamos dar de alta un cliente con un e-mail ya registrado, la aplicación nos mostrará un mensaje indicando que no se puede repetir el valor de dicho campo.

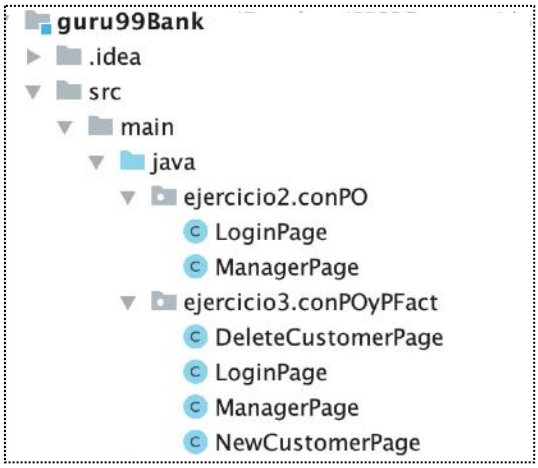
Adicionalmente, fíjate que para poder repetir la ejecución del test debemos "eliminar" el cliente que acabamos de crear después de ejecutar con éxito el test. Para ello tendrás que crear una nueva *page object*, que puedes llamar "DeleteCustomerPage" y que representa la página de nuestra aplicación para borrar un cliente. Una vez implementada la nueva page object, modifica el test para borrar el nuevo cliente creado (tendrás que utilizar el identificador del cliente para poder borrarlo), y además aceptar los mensajes de dos alertas que nos aparecerán para confirmar que queremos borrar dicha información.

- ***testTestNewClientDuplicate()***: en este segundo test, se trata de crear un nuevo cliente usando un e-mail que ya existe. En este caso, después de introducir los datos del cliente y pulsar el botón "Submit" nos debe aparecer un mensaje de alerta con el mensaje "Email Address Already Exist !!". Para garantizar que el cliente que introducimos como entrada ya existe, debes añadirlo previamente, y a continuación, repetir la operación para asegurarnos de que efectivamente se trata de un cliente repetido. Debemos comprobar que la aplicación nos muestra el mensaje anterior y que no nos deja insertar dos veces el mismo cliente. Al igual que antes, debes borrar los datos del cliente creado después de ejecutar el test para poder repetir su ejecución con las mismas condiciones.

Puedes crear un elemento de configuración Maven desde IntelliJ con el comando:

```
mvn test –Dtest="TestNewClient
```

Finalmente, mostramos una captura de pantalla del proyecto maven para esta práctica:



Resumen

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

DISEÑO DE PRUEBAS DE SISTEMA

- La selección de comportamientos a probar se realiza desde el punto de vista del desarrollador, y el objetivo es encontrar defectos (verificación)
- Usaremos técnicas de caja negra, por ejemplo: método basado en casos de uso, método de transición de estados

AUTOMATIZACIÓN DE PRUEBAS DE SISTEMA

- Podemos usar webdriver para implementar los tests, exactamente igual que para pruebas de validación (si nuestro SUT es es una aplicación web

DISEÑO DE PRUEBAS DE ACEPTACIÓN DE PROPIEDADES EMERGENTES FUNCIONALES

- La selección de comportamientos a probar se realiza desde el punto de vista del usuario, y el objetivo es valorar en qué grado se satisfacen las expectativas del cliente (validación).
- Usaremos técnicas de caja negra, por ejemplo: método basado en requisitos, método basado en escenarios.

AUTOMATIZACIÓN DE PRUEBAS DE ACEPTACIÓN

- Consideraremos el caso de que nuestro 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 Firefox)
- Si usamos webdriver directamente en nuestros tests, estos dependienterán 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 nuestro 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.