

Guía de uso Selenium

Br. Rafael Rodríguez Guzmán

22/Septiembre/2017

Introducción

¿Qué es Selenium?

Selenium es un entorno de pruebas de software para aplicaciones basadas en web. Selenium nos da la oportunidad de automatizar pruebas a aplicaciones web. Permite el uso de una variedad de lenguajes de programación tales como Java, C#, Ruby, Python o JavaScript(Node).

¿Qué necesito para usar Selenium?

Para poder usar Selenium es necesario descargar la biblioteca basada en el lenguaje que sea desea usar. Esta se puede descargar del siguiente link:

<http://www.seleniumhq.org/download/>

Además, es necesario descargar también el controlador del navegador web bajo el cual se realizarán las pruebas automatizadas. El controlador permite relacionar las funcionalidades del navegador web con el código de la biblioteca de Selenium. Un ejemplo sería poder abrir el navegador web, introducir una url o dar click en algún botón. Los links a los distintos controladores de navegadores también se encuentran en el link anterior.

Para crear nuestros códigos usaremos el IDE IntelliJ Idea de JetBrains. Podemos descargarlos del siguiente link:

<https://www.jetbrains.com/idea/>

Y trabajaremos usando el navegador Google Chrome, que puede ser descargado aquí:

<https://www.google.com.mx/chrome/browser/desktop/index.html>

Además usaremos el lenguaje de programación Java. Para ello debemos descargar el Java SE Development Kit (JDK). En el siguiente link podemos descargar la versión adecuada para nuestro sistema operativo:

<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>

En sistemas operativos Linux basados en el manejador de paquetes apt (Debian/Ubuntu) podemos descargar el JDK con el siguiente comando en consola:

```
sudo apt-get install default-jdk
```

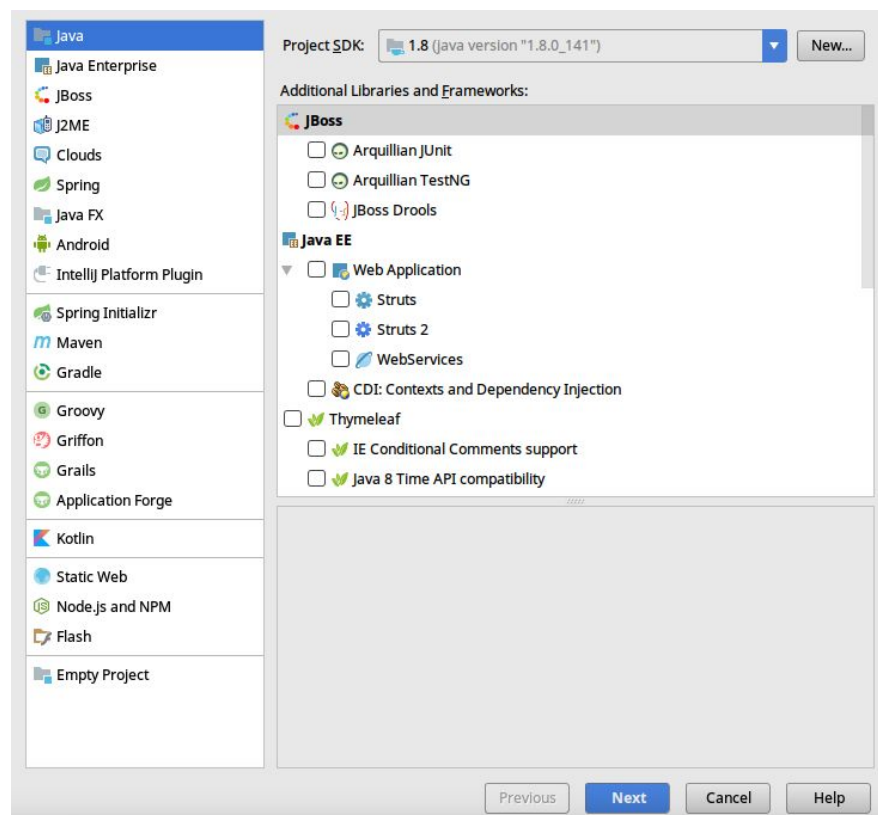
Nos preguntará si deseamos descargar e instalar los paquetes. Escribimos “y” para aceptar y presionamos la tecla “Enter”.

¿Cómo puedo usarlo? Un ejemplo práctico

Crearemos un proyecto de ejemplo en el que accederemos a la página de Walook (<http://www.walook.com.mx/>). Primero crearemos un nuevo proyecto:

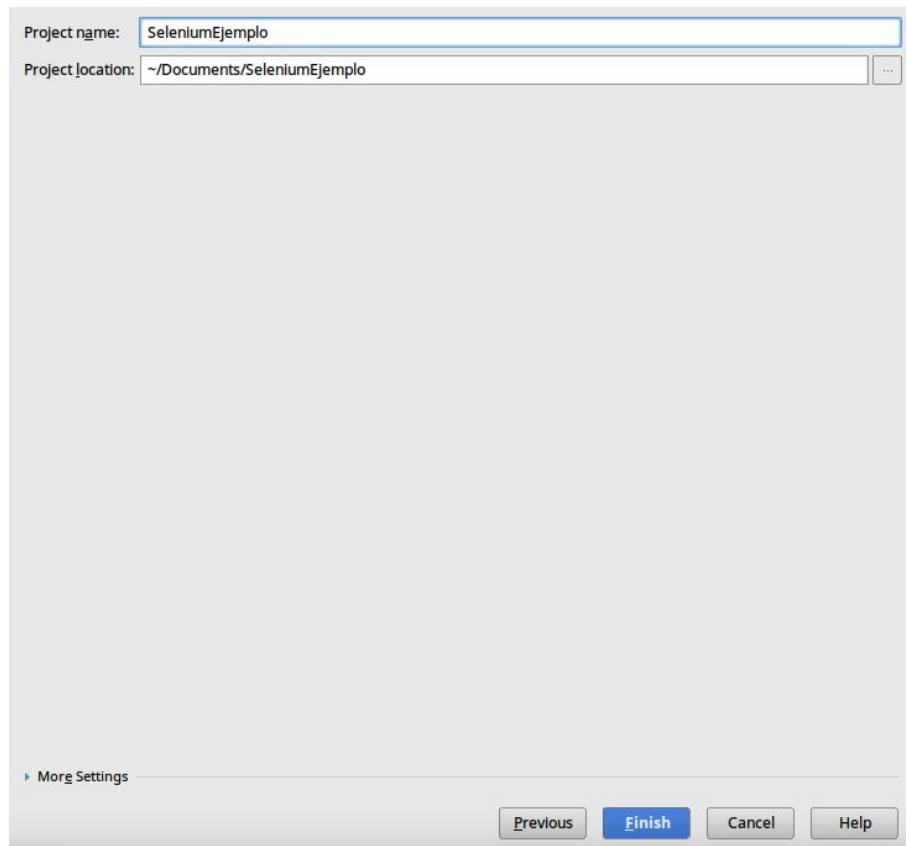


Crearemos usando un nuevo proyecto Java:

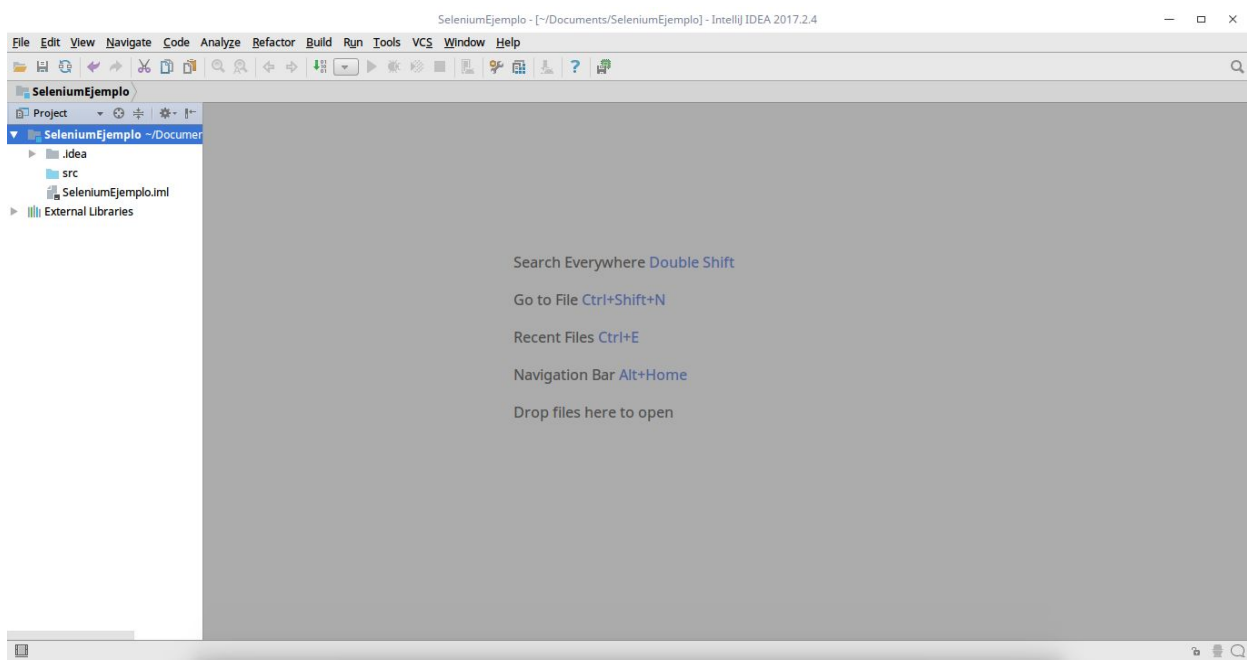


Daremos click en “Next”. Ahora se nos preguntará si queremos usar una plantilla para el proyecto. En este caso no usaremos ninguna y daremos click en “Next”.

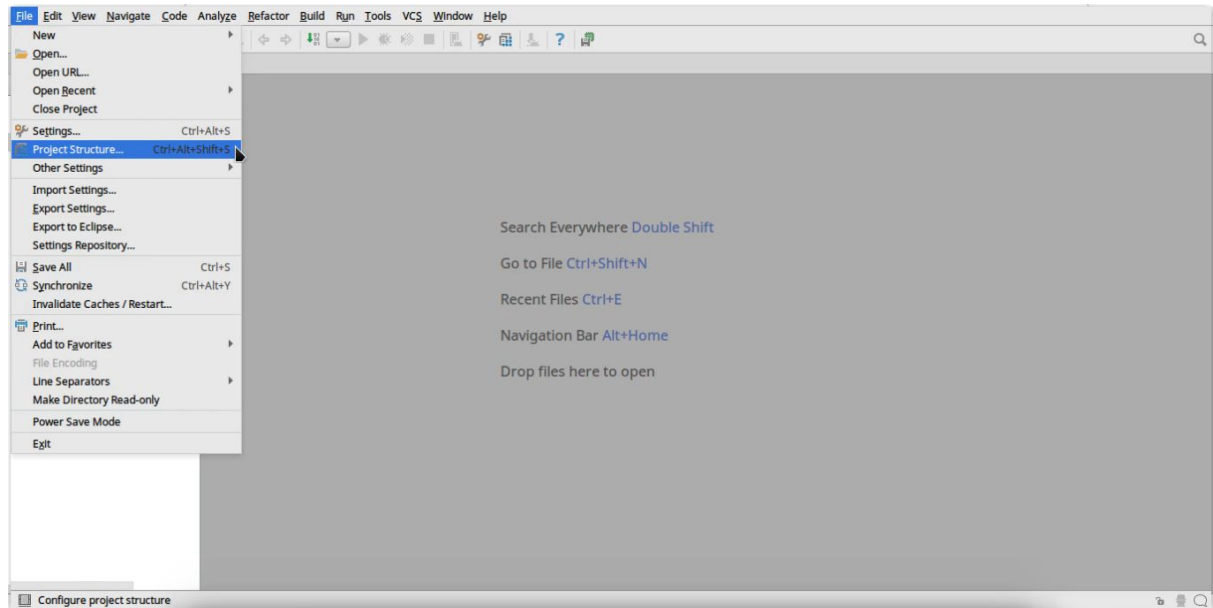
En la siguiente ventana se nos preguntará qué nombre deseamos usar para el proyecto y dónde deseamos que se guarde en nuestro disco duro. Estos datos se pueden colocar a gusto personal. Al finalizar estos cambios daremos click en “Finish”.



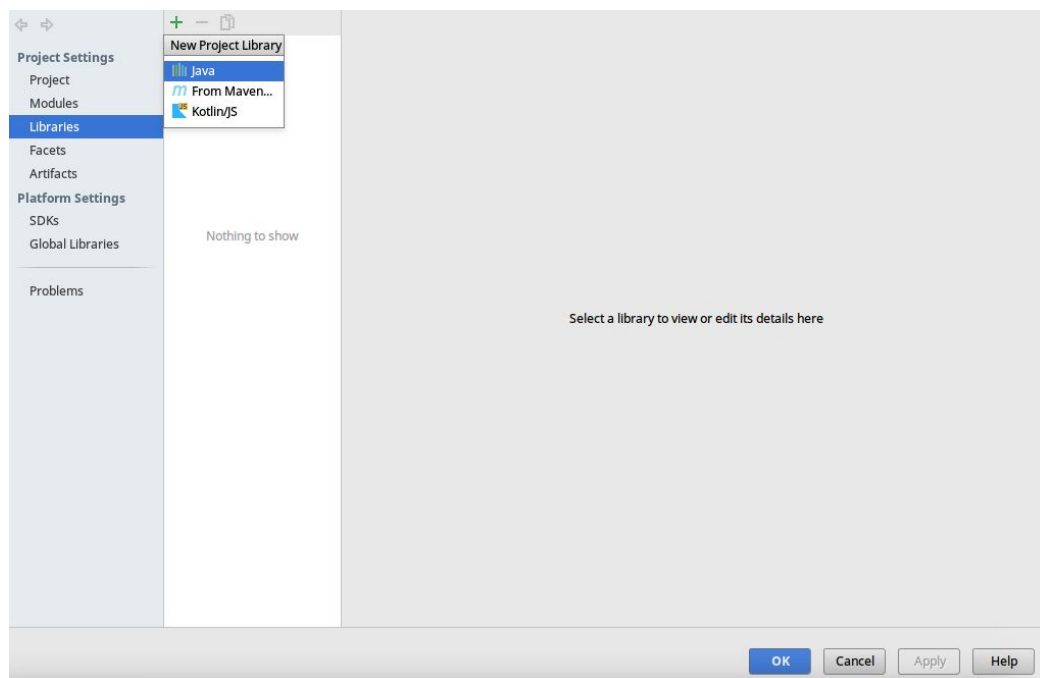
La siguiente ventana que se nos muestra es nuestro espacio de trabajo. Del lado izquierdo podemos ver todos los archivos de clases, texto, etc. que tenemos en nuestro proyecto. Importamos ahora los archivos necesarios para trabajar con Selenium.



Damos click en “File”, “Project Structure”.

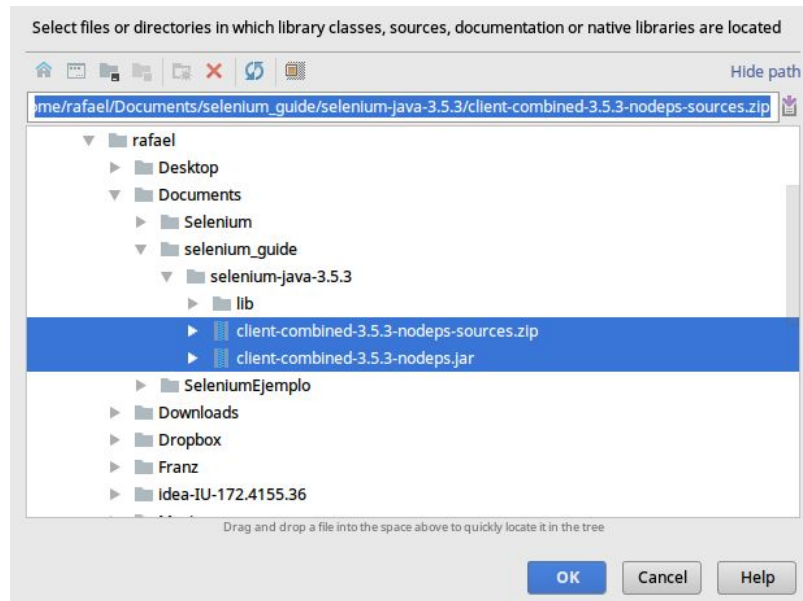


Nos aparece la siguiente ventana:



En la opción de “Libraries” damos click en el símbolo “+”, “Java”. La siguiente ventana nos permite ubicar la carpeta de la biblioteca de Selenium Java que descargamos previamente. Ubicamos esa carpeta y al abrirla podemos ver dos archivos que debemos seleccionar:

```
selenium-java-3.5.3/client-combined-3.5.3-nodeps.jar  
selenium-java-3.5.3/client-combined-3.5.3-nodeps-sources.zip
```

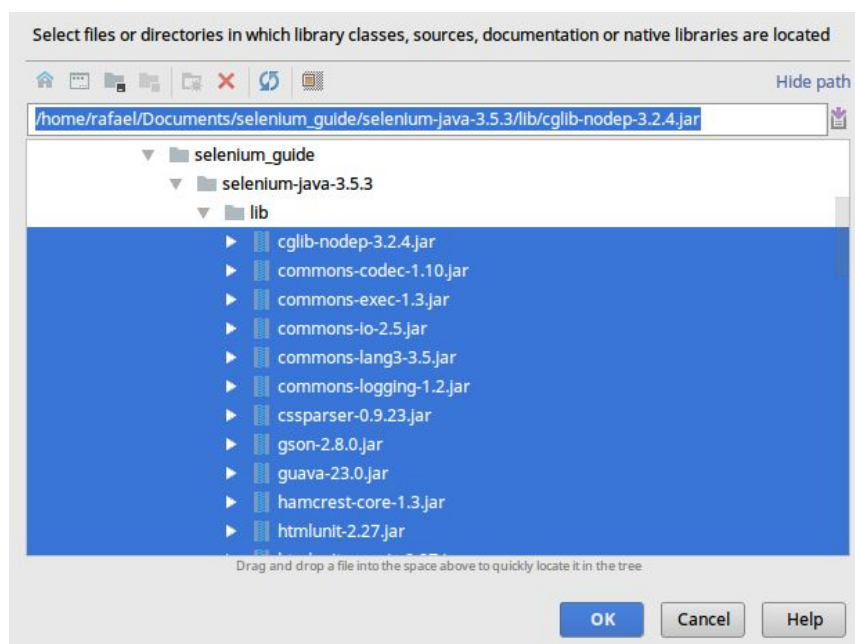


Una vez ubicados y seleccionados dichos archivos daremos click en el botón “OK”. Vemos ahora que los archivos se han agregado a nuestro proyecto. Daremos click nuevamente en el símbolo “+” para agregar más archivos necesarios para nuestro proyecto.

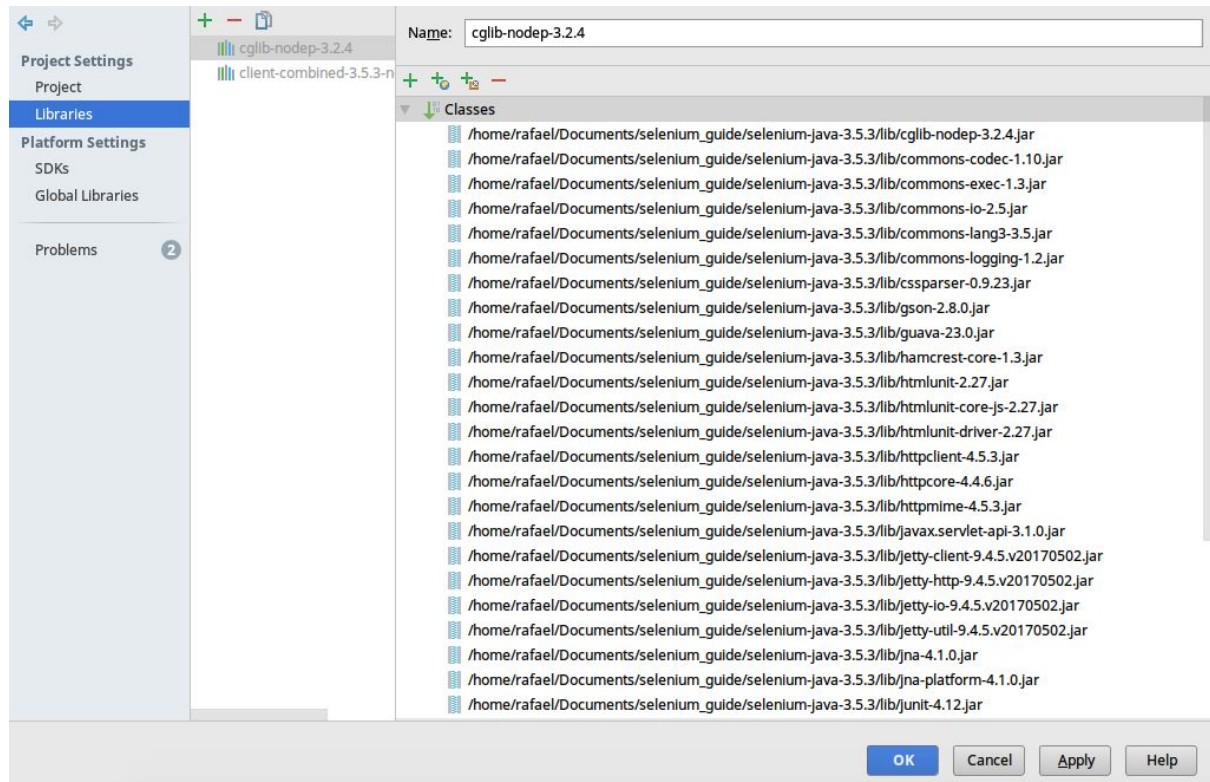
Seleccionaremos todos los archivos ubicados en la ruta:

`selenium-java-3.5.3/lib/`

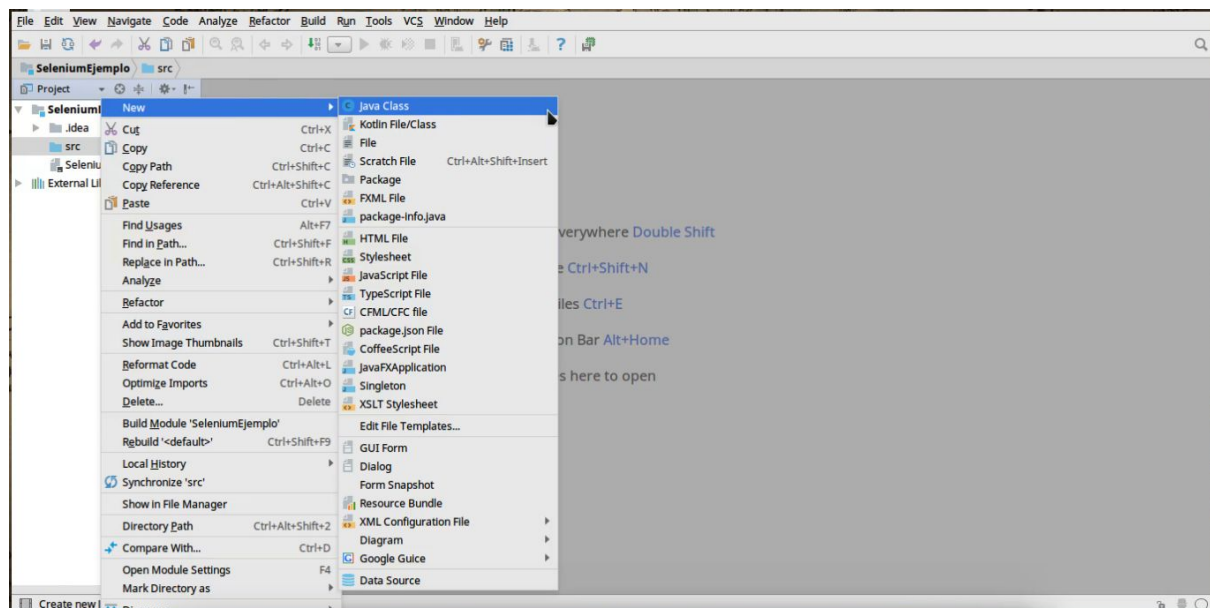
Damos click en “OK” para agregar los archivos seleccionados.

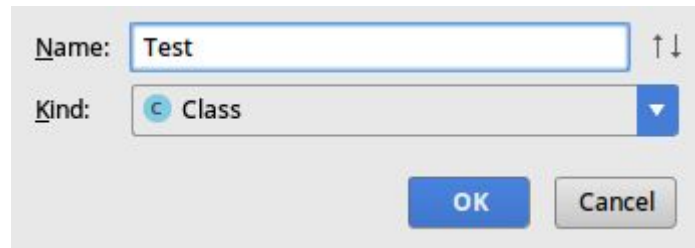


Podemos ver que ya se han agregado todos los archivos necesarios para poder trabajar en nuestro proyecto. Finalizamos dando click en “OK”.



Crearemos una nueva clase donde trabajaremos nuestro ejemplo.





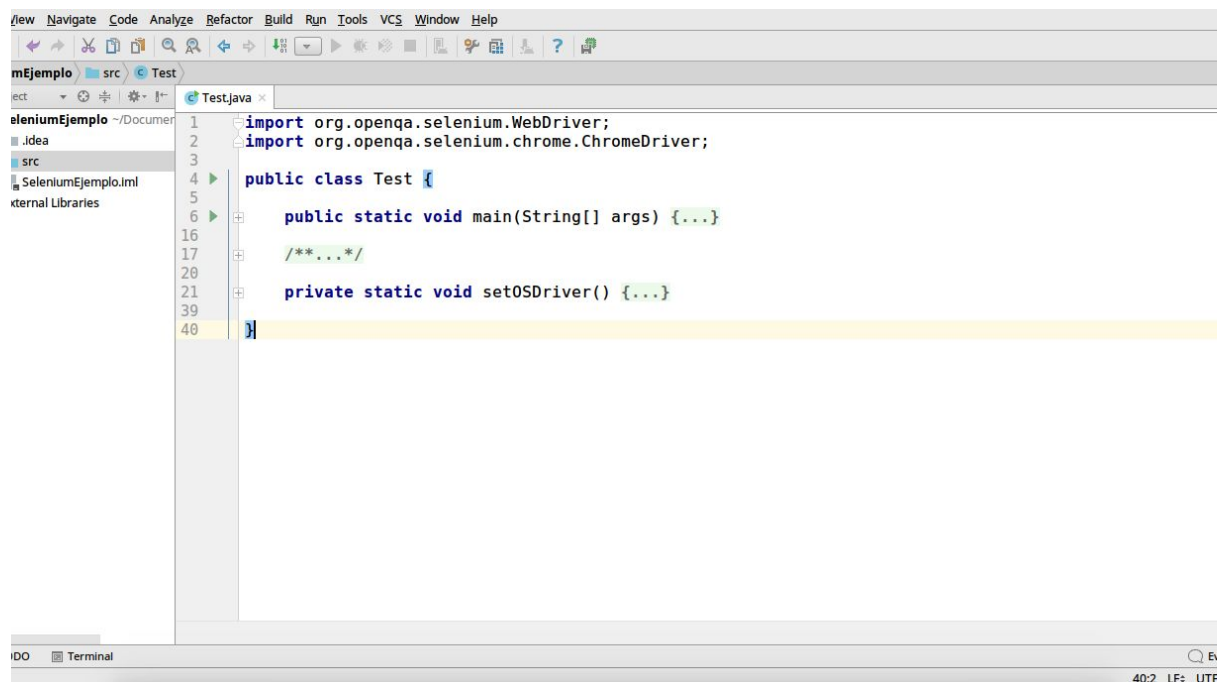
Name: ↑↓

Kind: Class ▼

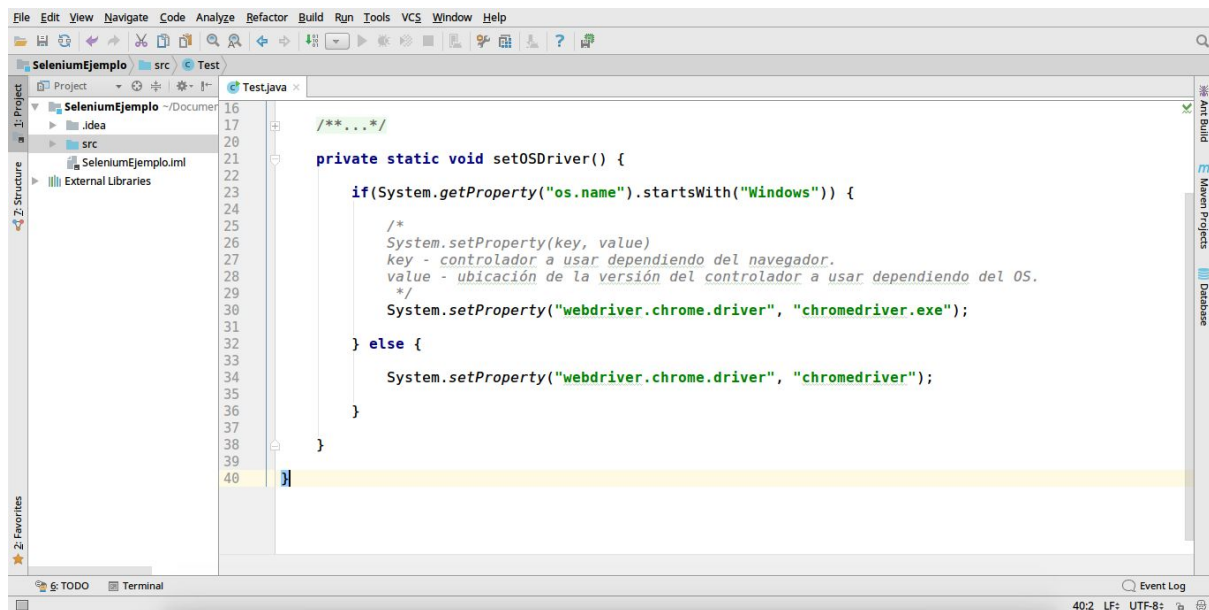
Una vez que creamos nuestra clase sobre la cual trabajar debemos realizar las importaciones de las bibliotecas a usar:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
```

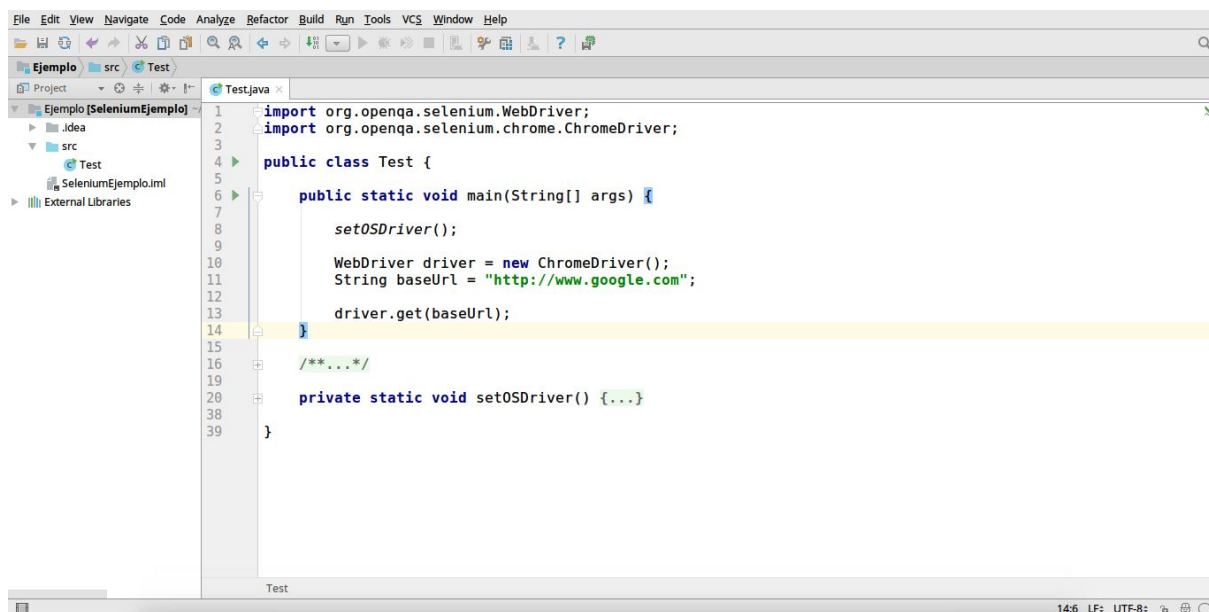
“WebDriver” es una clase que nos permite usar el controlador web para realizar las distintas funciones en él, mientras que “ChromeDriver” es la clase específica para controlar Google Chrome.



“setOSDriver()” es una función que nos permite establecer el controlador adecuado para el navegador según el sistema operativo (OS). En Windows el controlador es un archivo ejecutable (“chromedriver.exe”), pero para MacOS o Linux el archivo tiene otra extensión. Es necesario entonces establecer qué archivo se debe usar según el OS. La función nos permite seleccionar automáticamente el controlador adecuado para Windows o Linux.



Lo primero que haremos es abrir Google Chrome y acceder a <https://www.google.com.mx/>. Esto se muestra en el siguiente código:



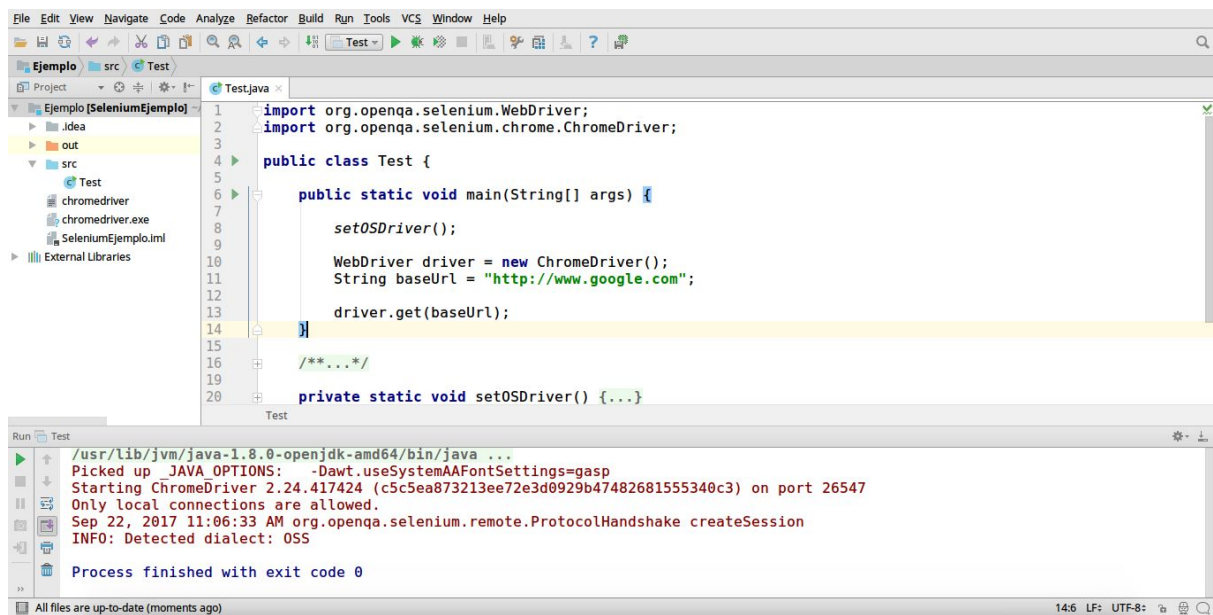
Las funcionalidades de este código son las siguientes:

- “SetOSDriver();” llama a la función que selecciona el controlador adecuado para usar Google Chrome según el sistema operativo.
- “WebDriver driver = new ChromeDriver();” crea un nuevo controlador de navegador web y lo inicializa como un controlador de Google Chrome. “driver” será el objeto a través del cual podremos acceder a los métodos que contiene “WebDriver”, y así interactuar con los elementos web de las aplicaciones.

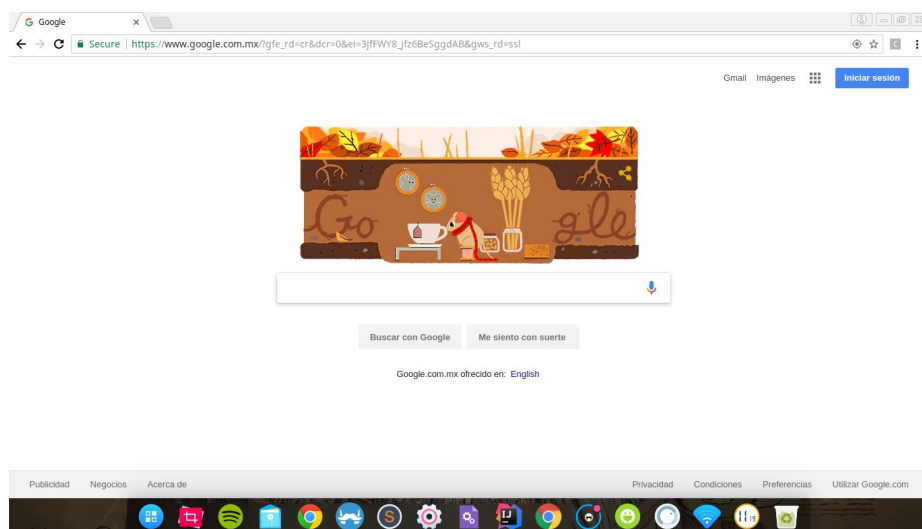
- “String baseUrl;” es una cadena que contiene la url de la página del buscador web a usar para encontrar “walook”. usaremos esta cadena para almacenar una url a la que queremos acceder, y será parámetro del método “driver.get()”.
- “driver.get(baseUrl);” abre una nueva instancia de Google Chrome y accede a la url que se le pasa como parámetro.

Podemos ejecutar ahora nuestro código al hacer click derecho sobre el código, y seleccionamos “Run”. En la parte inferior veremos algunos mensajes de información, y si todo salió correctamente el último mensaje debe ser el siguiente:

Process finished with exit code 0

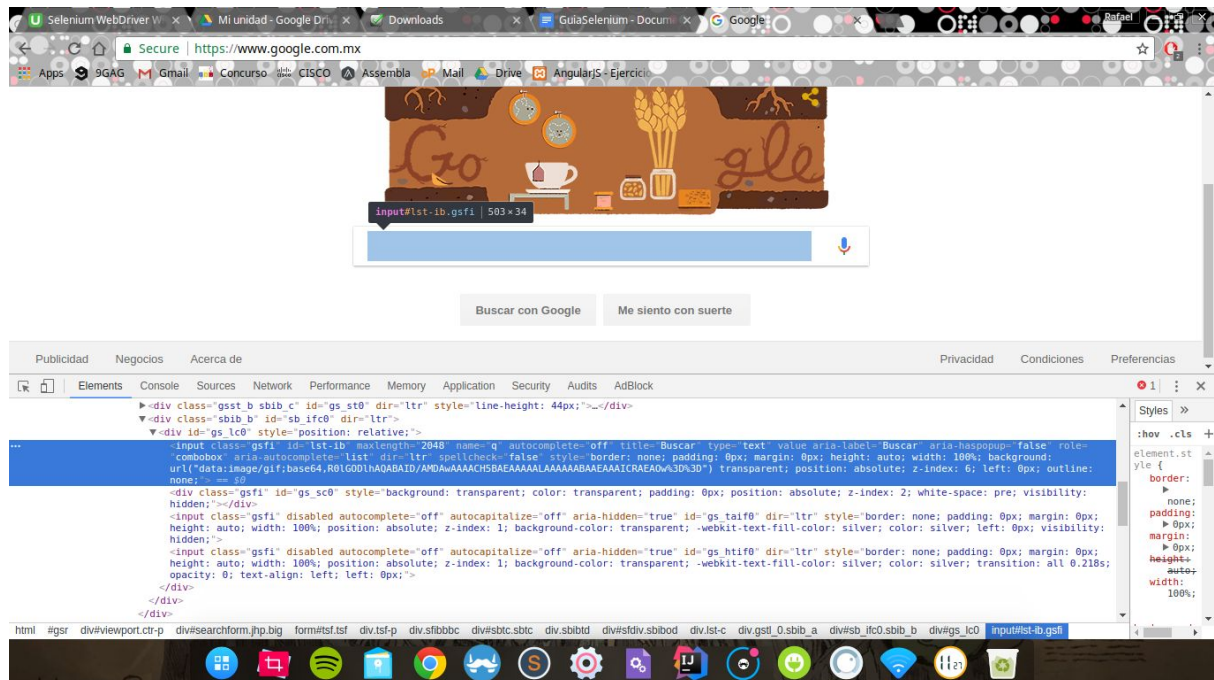


Además, si todo es correcto, se debe ejecutar una instancia de Google Chrome y abrir el url <https://www.google.com.mx/>.



Accediendo a los elementos web de las aplicaciones

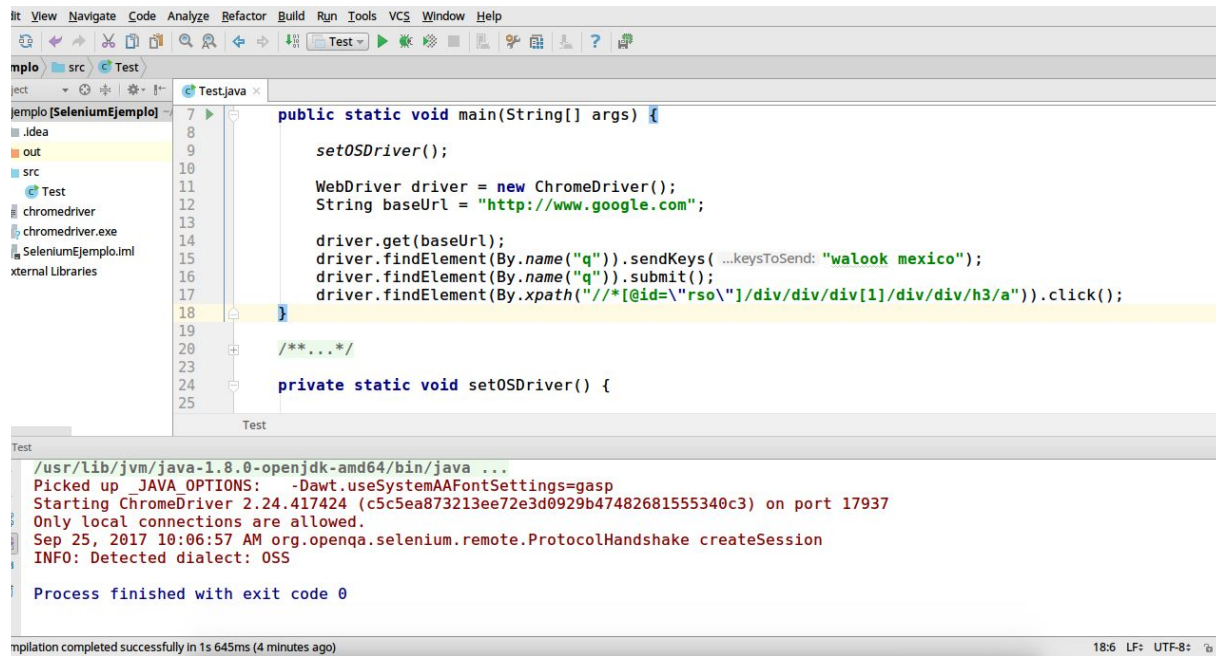
Para poder controlar los elementos de una aplicación web es necesario poder hacer referencia a ellos. En el caso de elementos HTML podemos hacer referencia usando los atributos de las etiquetas. Por lo general usamos el atributo “id” o “class” para ello, pero también podemos buscar elementos por el texto entre etiquetas o elementos CSS. Usaremos de ejemplo la página principal de Google (<https://www.google.com.mx/>).



Si oprimimos la tecla “F12” o damos click derecho sobre el cuadro de búsqueda y seleccionamos “Inspect”, Google Chrome abrirá un cuadro inferior donde nos muestra los elementos HTML de la página. y además señalará el bloque correspondiente al cuadro de búsqueda. Podemos apreciar que el cuadro es una etiqueta “input”, y que tiene como atributo “id=“lst-ib””. Podemos usar este atributo en nuestro código para acceder a ese cuadro de búsqueda e introducir el término a buscar. También, si hacemos click derecho sobre el código resaltado, seleccionamos “Copy”, “Copy XPath”, obtendremos el XPath del cuadro de búsqueda. Este XPath luce así: `//*[@id="lst-ib"]`. Podemos entonces acceder a elementos web a través del XPath, pues representa la ruta en la que se encuentran esos elementos dentro del Document Object Model (DOM).

Podemos ahora encontrar el XPath para la etiqueta <a> que tiene el link a la página de Walook México, y hacer click para abrir la página:

```
driver.findElement(By.xpath("//*[@id=\"rso\"]/div/div/div[1]/div/div/h3/a")).click();
```



The screenshot shows an IDE window with a project named 'jemplo [SeleniumEjemplo]'. The 'Test.java' file is open, displaying the following code:

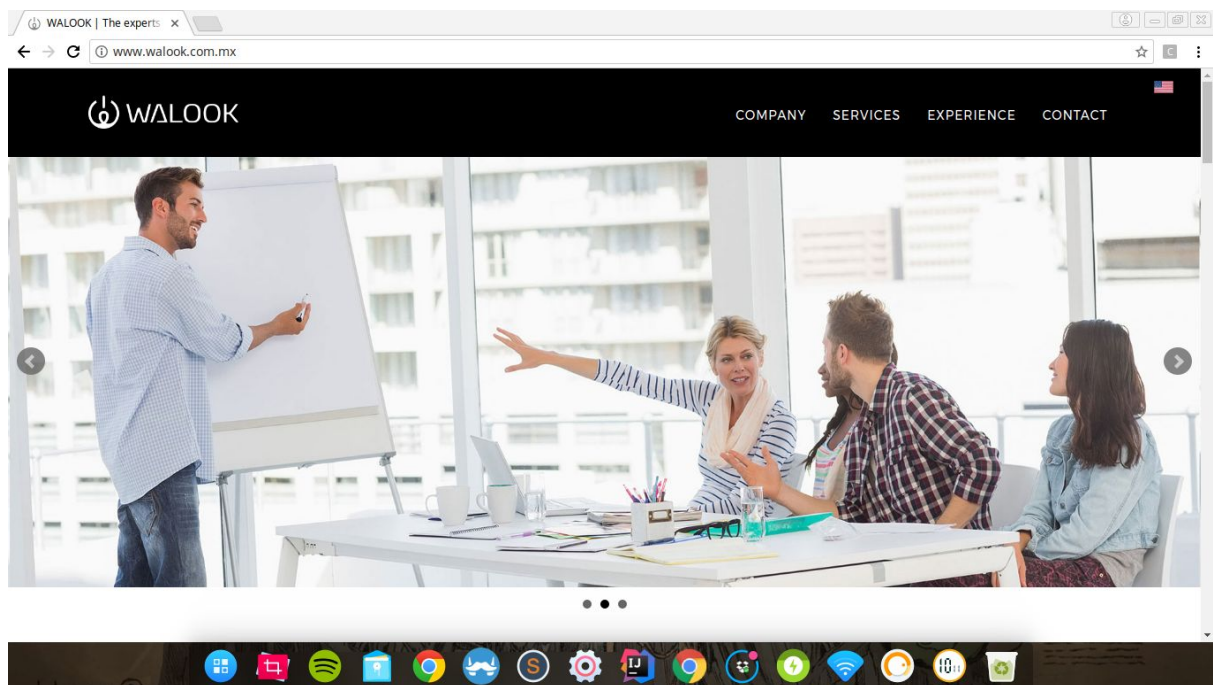
```
public static void main(String[] args) {  
    set0SDriver();  
  
    WebDriver driver = new ChromeDriver();  
    String baseUrl = "http://www.google.com";  
  
    driver.get(baseUrl);  
    driver.findElement(By.name("q")).sendKeys("walook mexico");  
    driver.findElement(By.name("q")).submit();  
    driver.findElement(By.xpath("//*[id=\"rso\"]/div/div/div[1]/div/div/h3/a")).click();  
}  
  
/** ... */  
  
private static void set0SDriver() {
```

The 'Test' tab at the bottom shows the execution output:

```
/usr/lib/jvm/java-1.8.0-openjdk-amd64/bin/java ...  
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=gasp  
Starting ChromeDriver 2.24.417424 (c5c5ea873213ee72e3d0929b47482681555340c3) on port 17937  
Only local connections are allowed.  
Sep 25, 2017 10:06:57 AM org.openqa.selenium.remote.ProtocolHandshake createSession  
INFO: Detected dialect: OSS  
  
Process finished with exit code 0
```

At the bottom, a status bar indicates 'Compilation completed successfully in 1s 645ms (4 minutes ago)' and the system clock shows '18:6 LF: UTF-8'.

Vemos ahora que efectivamente hemos accedido a la página de Walook México de manera automática.



Generando pruebas

Estructura del código de pruebas

Para realizar las pruebas se debe generar una clase que contenga las acciones que formarán parte de la prueba. Para ello haremos uso de la estructura de una clase JUnit. Analizaremos el siguiente código como ejemplo:

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import java.util.concurrent.TimeUnit;

public class BasicActions02 {

    private WebDriver driver;
    private String baseUrl;

    /*
     Acciones a realizar antes de ejecutar la prueba.
    */
    @Before
    public void setUp() throws Exception {
        //Creamos una instancia del driver para Google Chrome
        driver = new ChromeDriver();
        //Usaremos esta url como base para trabajar.
        baseUrl = "http://letscodeit.teachable.com/";
        //Se esperan 10 segundos a que carguen los elementos web antes de lanzar una
        excepción.
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        //Maximiza la ventana del navegador.
        driver.manage().window().maximize();
        //Abre una nueva instancia del navegador con la url solicitada.
        driver.get(baseUrl);
    }

    /*
     Acciones a realizar como prueba.
    */
    @Test
    public void test() {
        //Encontramos el botón "Login" y damos click en él.
        driver.findElement(By.xpath("//div[@id='navbar']/a[@href='/sign_in']")).click();

        //Limpiamos el campo de correo del usuario.
        driver.findElement(By.id("user_email")).clear();
    }
}
```



```

//Enviamos un correo al campo de correo de usuario.
driver.findElement(By.id("user_email")).sendKeys("part1@email.com");

//Limpiamos el campo de contraseña del usuario.
driver.findElement(By.id("user_password")).clear();

//Enviamos la contraseña al campo de contraseña de usuario.
driver.findElement(By.id("user_password")).sendKeys("password");
}

/*
 Acciones a realizar después de la prueba.
 */
@After
public void tearDown() throws Exception {
    //Esperamos 2 segundos antes de realizar la siguiente acción.
    Thread.sleep(2000);
    //Cerramos la instancia del navegador.
    driver.quit();
}
}

```

Podemos observar que la clase de pruebas tiene tres métodos: `setUp()`, `test()` y `tearDown()`, y cada método tiene las anotaciones `@Before`, `@Test` y `@After` respectivamente.

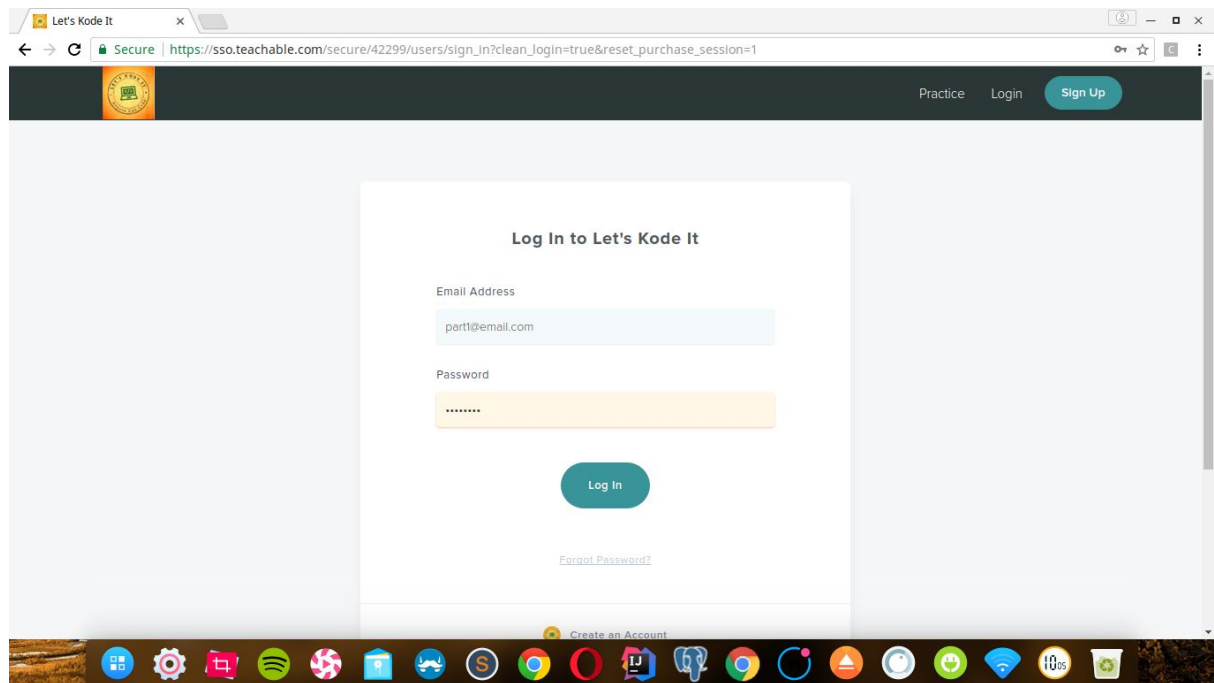
`@Before` indica que el método señalado se ejecutará antes de realizar las pruebas. Por lo tanto podemos ver que en `setUp()` tenemos la instanciación del web driver a usar, la url con la que vamos a trabajar y la apertura de una nueva instancia del navegador. `implicitWait()` nos permite establecer un tiempo de espera para que la página web cargue los elementos que necesitamos. En muchas ocasiones los elementos web no estarán disponibles para manipular hasta que la página termine de cargarse correctamente, así que es adecuado establecer un tiempo de espera antes de que Selenium trate de manipularlos y, al no encontrarlos disponibles, lance una excepción. El método con la etiqueta `@Before` (Etiqueta de JUnit) siempre se usa para realizar las acciones que se deben ejecutar antes de que las pruebas puedan ser ejecutadas. Por ejemplo: realizar una conexión a la base de datos, o en nuestro caso, establecer qué web driver vamos a usar y con qué url vamos a trabajar.

`@After` indica que el método señalado se ejecutará después de realizar las pruebas. Esta es una etiqueta JUnit. Tenemos entonces que el método `tearDown()` espera dos segundos antes después de que finalizan las pruebas antes de cerrar la ventana del navegador.

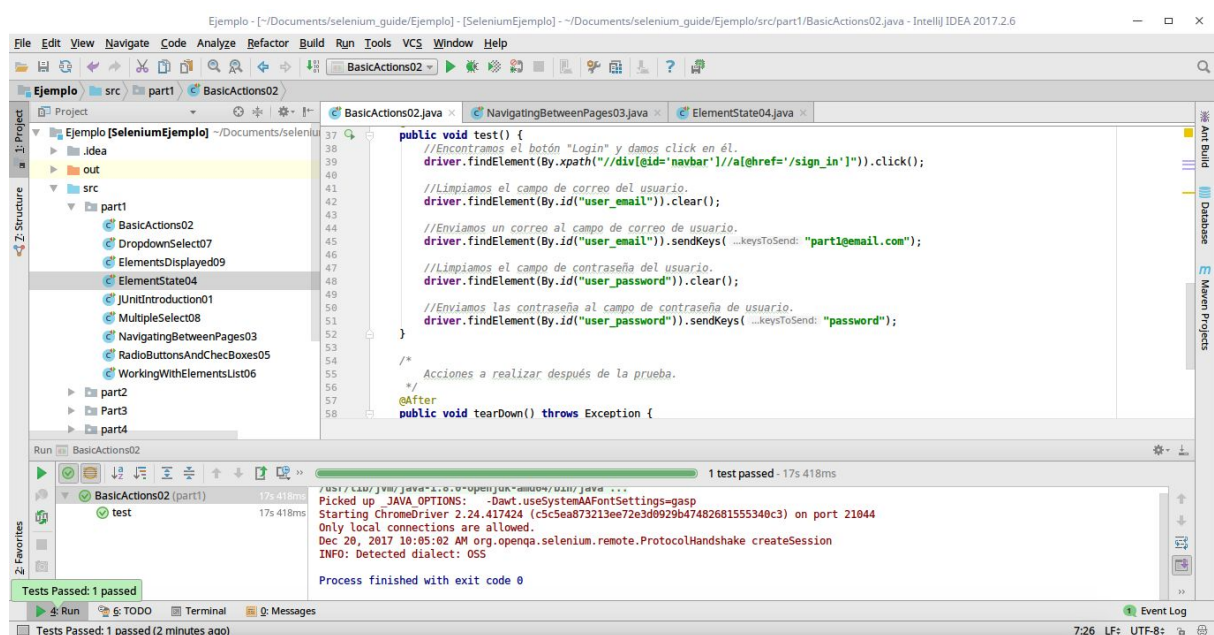
`@Test` indica que el método señalado será una prueba a ejecutar. El método `test()` contiene acciones que forman parte de la prueba que queremos realizar. En este caso accedemos a la página de "Let's Kode It", encontramos el botón de "Login" y damos click en él. Esto nos redirecciona a la página de login. Ya en esta página introducimos cadenas en los campos de correo y contraseña. Hay que notar que para acceder a los objetos web usamos el web

driver, y a través del método `findElement(By)` podemos encontrar los elementos web que deseemos. Existen distintas formas de encontrar elementos web en el DOM haciendo uso de Selenium. Por ejemplo: encontrar elementos por su id, por nombre de sus clases, por el texto que se encuentra en su etiqueta, por una ruta absoluta en el DOM (XPath), etc. Hablaremos más a detalle de cómo encontrar elementos en el DOM más adelante.

Al ejecutar la clase, obtenemos lo siguiente:



Y la ventana del navegador se cierra después de unos segundos. Si todo sale bien, podremos ver que IntelliJ nos muestra los resultados de la prueba como positivos.



Navegando entre páginas

Entre las funciones que provee Selenium podemos encontrar métodos para desplazarnos entre distintas páginas web.

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import java.util.concurrent.TimeUnit;

public class NavigatingBetweenPages03 {

    private WebDriver driver;
    private String baseUrl;

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
        baseUrl = "http://letscodeit.teachable.com/";
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        driver.manage().window().maximize();
        driver.get(baseUrl);
    }

    @Test
    public void test() throws InterruptedException {
        //Obtenemos el título de la página
        String title = driver.getTitle();
        System.out.println("Title of the page: " + title);

        //Obtenemos la url donde nos encontramos actualmente.
        String currentUrl = driver.getCurrentUrl();
        System.out.println("Current URL: " + currentUrl);

        //Url hacia donde queremos navegar.
        String urlToNavigate =
"https://sso.teachable.com/secure/42299/users/sign_in?clean_login=true&reset_purchase_session=1";
        //Navegamos a la url especificada.
        driver.navigate().to(urlToNavigate);

        currentUrl = driver.getCurrentUrl();
        System.out.println("Current URL: " + currentUrl);

        //Tiempo de espera entre navegación. Lanza InterruptedException.
        Thread.sleep(2000);

        //Navegamos a la página anterior.
```



```

    driver.navigate().back();
    System.out.println("Navigate back");
    currentUrl = driver.getCurrentUrl();
    System.out.println("Current URL: " + currentUrl);

    Thread.sleep(2000);

    //Navegamos a la página siguiente, de la que provenimos al navegar atrás.
    driver.navigate().forward();
    System.out.println("Navigate forward");
    currentUrl = driver.getCurrentUrl();
    System.out.println("Current URL: " + currentUrl);

    Thread.sleep(2000);

    //Volvemos a la página anterior
    driver.navigate().back();
    System.out.println("Navigate back");
    currentUrl = driver.getCurrentUrl();
    System.out.println("Current URL: " + currentUrl);

    //Refrescamos/recargamos la página.
    driver.navigate().refresh();
    System.out.println("Navigate refresh");

    //Podemos refrescar usando get, pues nos lleva al url que estamos refrescando.
    driver.get(currentUrl);
    System.out.println("Navigate refresh");

    //Obtenemos el código fuente HTML de la página.
    String pageSource = driver.getPageSource();
    System.out.println(pageSource);
}

@After
public void tearDown() throws Exception {
    Thread.sleep(2000);
    driver.quit();
}
}

```

navigate.to() nos lleva a la url especificada, mientras que navigate.forward() y navigate.back() avanzan o retroceden desde la página actual. navigate.refresh() recarga la página actual, y driver.get() también puede ser usado para refrescar, pues carga la url indicada.

Obteniendo el estado de elementos web

@Test

```
public void test() {  
    WebElement element = driver.findElement(By.id("lst-ib"));  
    System.out.println("element is enabled? " + element.isEnabled());  
    element.sendKeys("letskodeit");  
}
```

Podemos obtener el estado de un objeto web, es decir, saber si el objeto está habilitado y deshabilitado. Para ello debemos obtener el elemento web que deseamos y usar la función `isEnabled()`.

Obtener una lista de elementos web

@Test

```
public void test() throws InterruptedException {  
    String xPathRadio = "//input[contains(@type, 'radio') and contains(@name, 'cars')]";  
    List<WebElement> radioButtons = driver.findElements(By.xpath(xPathRadio));  
  
    for (int i = 0; i < radioButtons.size(); i++) {  
        boolean isChecked = false;  
        isChecked = radioButtons.get(i).isSelected();  
  
        if (!isChecked) {  
            radioButtons.get(i).click();  
            Thread.sleep(2000);  
        }  
    }  
}
```

Podemos obtener una lista de elementos web usando `driver.findElements(By)`. Las opciones para encontrar los elementos siguen siendo las mismas que para un solo elemento. Si no se encuentran elementos del tipo especificado, la lista quedará vacía. `isSelected()` nos permite saber si un radio button está seleccionado o no.

Seleccionar elementos de un dropdown select

@Test

```
public void test() throws InterruptedException {  
    WebElement element = driver.findElement(By.id("carselect"));  
    //Conjunto de elementos web, correspondientes a las opciones de select con id  
    "carselect".  
    Select select = new Select(element);  
  
    Thread.sleep(2000);  
    System.out.println("Select Benz by value");  
    //Seleccionar elemento por el atributo "value".  
    select.selectByValue("benz");  
  
    Thread.sleep(2000);  
}
```

```

System.out.println("Select Honda by index");
//Seleccionar elemento por posición en la lista.
select.selectByIndex(2);

Thread.sleep(2000);
System.out.println("Select BMW by visible text");
//Seleccionar elemento por el texto visible en la página.
select.selectByVisibleText("BMW");

Thread.sleep(2000);
System.out.println("Print the list of all options");
List<WebElement> options = select.getOptions();

for (int i = 0; i < options.size(); i++) {
    String optionName = options.get(i).getText();
    System.out.println(optionName);
}
}

```

Para visualizar elementos de listas dropdown haremos uso de la clase Select. La etiqueta select tiene un id='carselect', con el cuál podemos encontrarla dentro del DOM. La clase Select debe tener asignada dicha etiqueta del mismo nombre, que nos permite acceder a las distintas opciones de la misma. La selección que hagamos se verá reflejada en la página.

```

@Test
public void test() throws InterruptedException {
    WebElement element = driver.findElement(By.id("multiple-select-example"));
    Select select = new Select(element);

    Thread.sleep(2000);
    System.out.println("Select orange by value");
    select.selectByValue("orange");

    Thread.sleep(2000);
    System.out.println("De-select orange by value");
    select.deselectByValue("orange");

    Thread.sleep(2000);
    System.out.println("Select peach by index");
    select.selectByIndex(2);
    //select.deselectByIndex(2);

    Thread.sleep(2000);
    System.out.println("Select apple by visible text");
    select.selectByVisibleText("Apple");
    //select.deselectByVisibleText("Apple");

    Thread.sleep(2000);
    System.out.println("Print all selected options");
}

```

```

List<WebElement> selectedOptions = select.getAllSelectedOptions();

for (WebElement option : selectedOptions) {
    System.out.println(option.getText());
}

Thread.sleep(2000);
System.out.println("De-select all selected options");
select.deselectAll();
}

```

Métodos útiles

Se tienen distintas funciones para cada elemento web. Al crear un nuevo objeto y asignarle un valor a través del controlador (WebElement element = driver.findElement(By.id(""))) se nos proporciona una lista de métodos útiles para el objeto web. Sin embargo la lista de métodos es la misma para todo WebElement, así que hay que tener cuidado para usar el método correcto para el objeto web adecuado. Por ejemplo: si tenemos WebElement element = driver.findElement(By.id("opentab"));, que corresponde a un botón que abre una nueva ventana, podemos usar la siguiente función: element.sendKeys("sending a string to the input textbox");. Sin embargo, esta función sólo es aceptada por objetos web de tipo campo de texto, así que al intentar ejecutarla Selenium lanzará una excepción.

```

WebElement element = driver.findElement(By.id("opentab"));
int time = 3;
int x = 0;
int y = 0;

```

//Valida si el elemento está seleccionado
element.isSelected();

//Valida si el elemento se encuentra en a la vista o está oculto
element.isDisplayed();

//Limpia los caracteres de un cuadro de entrada de texto
element.clear();

//Obtiene el nombre de etiqueta del elemento
element.getTagName();

//Ejecuta un click sobre el elemento web.
element.click();

//Obtiene el texto del botón "open tab"
String elementText = element.getText();

//Obtiene el nombre del atributo dado como parámetro. En este caso el placeholder
String attributeValue = element.getAttribute("placeholder");

//Enviar caracteres a un campo de texto

```
element = driver.findElement(By.id("name"));
element.sendKeys("sending a string to the input textbox");

//Encuentra un elemento web, usando el id, css, texto visible, etc.
driver.findElement(By);

//Cierra la ventana actual del navegador
driver.close();

//Cierra todas las ventanas del navegador
driver.quit();

//Carga la url introducida como parámetro en el navegador
driver.get(baseUrl);

//Encuentra una lista de elementos web. Si no encuentra ningún elemento la lista estará vacía
driver.findElements(By);

//Obtiene el título de la página actual
driver.getTitle();

//Si se tienen múltiples ventanas abiertas, cambia el foco a la ventana deseada.
driver.switchTo().window("handleName");

//Obtiene la url de la ventana que está enfocada
driver.getCurrentUrl();

//Obtiene el "handle" de la ventana que está enfocada
driver.getWindowHandle();

//Obtiene todos los handles de todas las ventanas abiertas
driver.getWindowHandles();

//Obtiene el código fuente de la página que está enfocada
driver.getPageSource();

//Permite navegar entre páginas
driver.navigate().to(baseUrl);
driver.navigate().back();
driver.navigate().forward();

//Recarga la página
driver.navigate().refresh();

//Manejo de tiempos de espera. Se le brinda cantidad de tiempo y unidad de tiempo como parámetros
driver.manage().timeouts().implicitlyWait(time, TimeUnit.SECONDS);
driver.manage().timeouts().pageLoadTimeout(time, TimeUnit.SECONDS);
driver.manage().timeouts().setScriptTimeout(time, TimeUnit.SECONDS);
```

//Manipulación de las ventanas del navegador

```
driver.manage().window().maximize();  
driver.manage().window().fullscreen();  
driver.manage().window().getPosition();  
driver.manage().window().setPosition(new Point(x, y));  
driver.manage().window().getSize();  
driver.manage().window().setSize(new Dimension(x, y));
```

Tiempos de espera implícitos y explícitos

Los tiempo de espera implícitos son periodos que se asignan a todos los elementos web para que la página termine de cargarlos. En muchas ocasiones, al tratar de realizar acciones automatizadas sobre ellos, los elementos en las páginas aún no se encuentran disponibles. Es conveniente por lo tanto asignar un par de segundos de espera. Para hacer esto usamos **driver**.manage().timeouts().implicitlyWait(3, TimeUnit.**SECONDS**);. La función recibe dos argumentos: el primero es el tiempo que deseamos darle a cada elementos para esperar, y el segundo es el unidad con la que mediremos la espera. Al usar esta función, Selenium realiza una espera de tres segundos antes de manipular cada objeto, con el objetivo de que el elemento web se encuentre disponible. Si al tratar de acceder al objeto este aún no se encuentra disponible, se lanzará una excepción. El tiempo por defecto es cero, así que si el elemento no se encuentra disponible al momento es adecuado establecer un tiempo de espera. Hay que tener en cuenta que el tiempo designado se aplicará a todos los elementos web, así que un periodo corto es más adecuado.

Los tiempo de espera explícitos son periodos de espera que se pueden asignar bajo ciertas condiciones, es decir, una acción no será ejecutada hasta que un objeto se encuentre disponible. Por ejemplo:

```
WebDriverWait wait = new WebDriverWait(driver, 3);  
WebElement emailField = wait.until(  
    ExpectedConditions.visibilityOfElementLocated(By.id("user_email")));  
  
emailField.sendKeys("test_email");
```

En esta prueba, se espera a que el textbox de correo del usuario se encuentre disponible antes de enviar una cadena. La clase WebDriverWait establece el tiempo de espera máximo para un elemento, y la función wait.until() puede validar distintas condiciones para realizar la espera. Por ejemplo: ExpectedConditions.elementToBeClickable, ExpectedConditions.presenceOfElementLocated o ExpectedConditions.visibilityOfElementLocated.

Las esperas explícitas se aplican a elementos individuales, a diferencia de las implícitas. No es conveniente combinar ambos tipos de espera porque pueden causar resultados inesperados. Por ejemplo: una espera implícita de 10 segundos y una espera explícita de 15 segundos en la misma prueba puede causar que se aborte la ejecución de la prueba a los 20 segundos.

Ejemplo de espera implícita:

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import java.util.concurrent.TimeUnit;

public class ImplicitWaitDemo {

    private WebDriver driver;
    private String baseUrl;

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
        baseUrl = "https://letscodeit.teachable.com/p/practice";

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

        //Espera 3 segundos a la carga de los elementos.
        driver.manage().timeouts().implicitlyWait(3, TimeUnit.SECONDS);
    }

    @Test
    public void test() throws Exception {
        driver.get(baseUrl);
        driver.findElement(By.linkText("Login")).click();
        driver.findElement(By.id("user_email")).sendKeys("test");
    }

    @After
    public void tearDown() throws Exception {
        Thread.sleep(3000);
        driver.quit();
    }
}
```

Ejemplo de espera explícita:

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
```

```

import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;

public class ExplicitWaitDemo {

    private WebDriver driver;
    private String baseUrl;

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
        baseUrl = "https://letscodeit.teachable.com/p/practice";

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

        driver.get(baseUrl);
    }

    @Test
    public void test() throws Exception {
        WebElement loginLink = driver.findElement(By.linkText("Login"));

        loginLink.click();

        WebDriverWait wait = new WebDriverWait(driver, 3);
        WebElement emailField = wait.until(
            ExpectedConditions.visibilityOfElementLocated(By.id("user_email")));

        emailField.sendKeys("test_email");
    }

    @After
    public void tearDown() throws Exception {
        Thread.sleep(2000);
        driver.quit();
    }
}

```

Tomar capturas de pantalla

Algo que nos puede ser útil durante la prueba capturar la pantalla del navegador en determinado tiempo. Para ello podemos usar el siguiente ejemplo:

```

import org.apache.commons.io.FileUtils;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

```



```

import java.io.File;
import java.util.concurrent.TimeUnit;

public class Screenshots {

    private WebDriver driver;
    private String baseUrl;

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
        baseUrl = "https://letscodeit.teachable.com/p/practice";

        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    }

    @Test
    public void testJavaScriptExecution() throws Exception {
        driver.get(baseUrl);

        String fileName = "screenshotSelenium.png";

        File sourceFile = ((TakesScreenshot) driver).getScreenshotAs(OutputType.FILE);
        FileUtils.copyFile(sourceFile, new File(fileName));
    }

    @After
    public void tearDown() throws Exception {
        Thread.sleep(3000);
        driver.quit();
    }
}

```

Algo que tomar en cuenta es que podemos establecer la ubicación de dónde deseamos que se guarde la captura de pantalla, concatenando el String de la ruta con el String del nombre del archivo al pasarlo como parámetro en `FileUtils.copyFile(sourceFile, new File(fileName))`.

Ejecutar código JavaScript con Selenium

Podemos ejecutar código JS usando la clase `JavaScriptExecutor` y usando la función `executeScript()` como se muestra en el siguiente ejemplo:

```

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.JavascriptExecutor;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;

```

```

import org.openqa.selenium.chrome.ChromeDriver;

import java.util.concurrent.TimeUnit;

public class JavaScriptExecution {

    private WebDriver driver;
    private String baseUrl;
    private JavascriptExecutor js;

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
        baseUrl = "https://letscodeit.teachable.com/p/practice";
        js = (JavascriptExecutor) driver;

        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    }

    @Test
    public void testJavaScriptExecution() throws Exception {
        //Navigation
        //driver.get(baseUrl);
        js.executeScript("window.location = 'https://letscodeit.teachable.com/p/practice'");

        //Finding elements
        //WebElement textBox = driver.findElement(By.id("name"));
        WebElement textBox = (WebElement) js.executeScript("return
document.getElementById('name');");

        textBox.sendKeys("test");
    }

    @After
    public void tearDown() throws Exception {
        Thread.sleep(3000);
        driver.quit();
    }
}

```

Usando WebDriver accedemos a una url con el método get(), y obtenemos un elemento web con findElement(), sin embargo esas operaciones se pueden reemplazar con código JavaScript. El objeto js.executeScript("") permite que la cadena que se le pase como parámetro sea ejecutada como código JS. En este caso usamos js.executeScript("window.location = 'https://letscodeit.teachable.com/p/practice'") en vez de driver.get(baseUrl) para acceder a la página, y usamos WebElement textBox =

(WebElement) `js.executeScript("return document.getElementById('name');")` en vez de `driver.findElement(By.id("name"))` para recuperar un objeto web.

Ejemplo: Obteniendo el alto y ancho de una página:

```
@Test
public void testJavaScriptExecution() throws Exception {
    //Navigation
    js.executeScript("window.location = 'https://letscodeit.teachable.com/p/practice'");

    //Size of the window
    long height = (Long) js.executeScript("return window.innerHeight;");
    long width = (Long) js.executeScript("return window.innerWidth;");

    System.out.println("Height is: " + height);
    System.out.println("Width is: " + width);
}
```

Ejemplo: Enfocando un objeto de la página y haciendo scroll

```
@Test
public void testJavaScriptExecution() throws Exception {
    //Navigation
    //driver.get(baseUrl);
    js.executeScript("window.location = 'https://letscodeit.teachable.com/p/practice'");

    Thread.sleep(2000);
    //Scroll hacia abajo con 1900px
    js.executeScript("window.scrollTo(0, 1900);");
    Thread.sleep(2000);
    //Scroll hacia arriba con -1900px
    js.executeScript("window.scrollTo(0, -1900);");
    Thread.sleep(2000);

    WebElement element = driver.findElement(By.id("mouseover"));

    //Se hace scroll hacia donde se encuentra el objeto dado como parámetro
    js.executeScript("arguments[0].scrollIntoView(true);", element);
    Thread.sleep(2000);
    js.executeScript("window.scrollTo(0, -190);");
}
```

Cambiar entre ventanas abiertas

Si al realizar nuestras pruebas es necesario abrir más de una ventana, Selenium siempre tendrá a la ventana original enfocada, así que las acciones siempre se realizarán sobre ella. Por ejemplo: tenemos una ventana con un botón para realizar un login, pero al presionar ese botón se abre una ventana emergente donde debemos introducir el nombre de usuario y contraseña, y tratamos de enviar caracteres a los campos, estos se intentarán enviar a la

ventana original y no a la emergente. Para corregir esto es necesario cambiar la ventana en la que se está enfocando.

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

import java.util.Set;
import java.util.concurrent.TimeUnit;

public class SwitchWindow {

    private WebDriver driver;
    private String baseUrl;

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
        baseUrl = "https://letscodeit.teachable.com/p/practice";

        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
        driver.get(baseUrl);
    }

    @Test
    public void test() throws InterruptedException {
        //get the handle
        String parentHandle = driver.getWindowHandle();
        System.out.println("Parent handle: " + parentHandle);

        //find open window button
        WebElement openWindow = driver.findElement(By.id("openwindow"));
        openWindow.click();

        //get all handles
        Set<String> handles = driver.getWindowHandles();

        //switching between handles
        for (String handle : handles) {
            System.out.println(handle);

            if (!handle.equals(parentHandle)) {
                driver.switchTo().window(handle);

                Thread.sleep(2000);
            }
        }
    }
}
```

```

WebElement searchBox = driver.findElement(By.id("search-courses"));
searchBox.sendKeys("python");

Thread.sleep(2000);

//Solo cierra la ventana actual, pero no cierra el navegador
driver.close();

break;
}
}

//switch back to the parent window
driver.switchTo().window(parentHandle);
driver.findElement(By.id("name")).sendKeys("test successful");
}

@After
public void tearDown() throws Exception {
    Thread.sleep(3000);
    driver.quit();
}
}

```

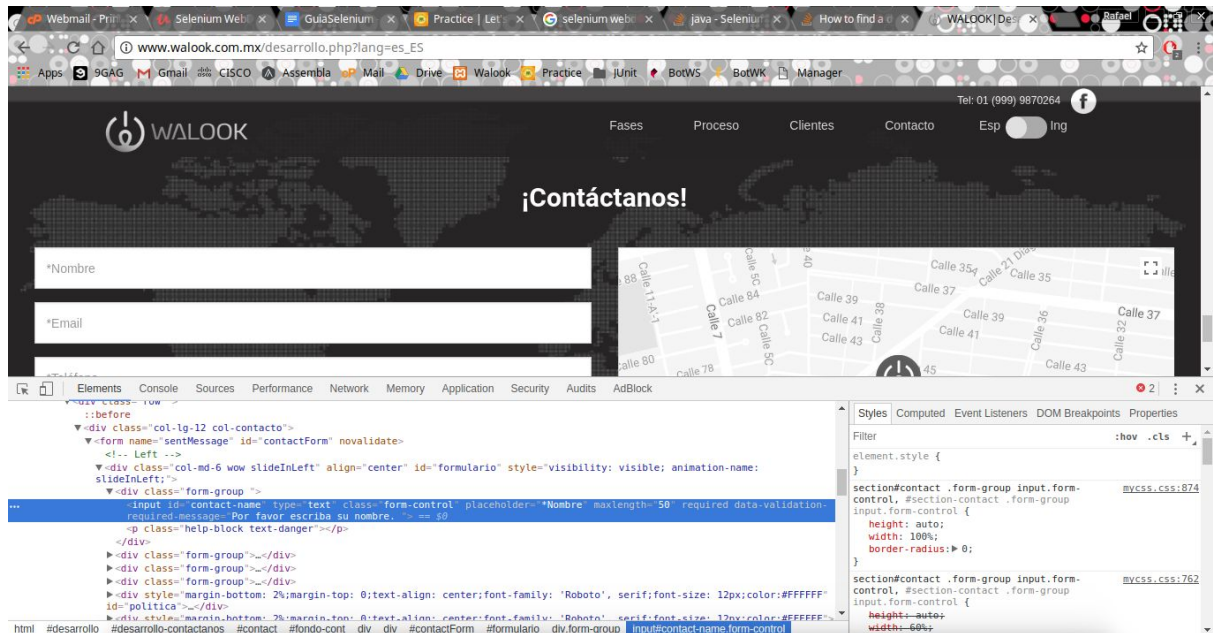
Un "handle" nos permite saber qué ventana estamos utilizando, así podemos usar `driver.switchTo().window(handle)` para cambiar el enfoque entre las distintas ventanas que el navegador tenga abiertas.

Formas de encontrar elementos web

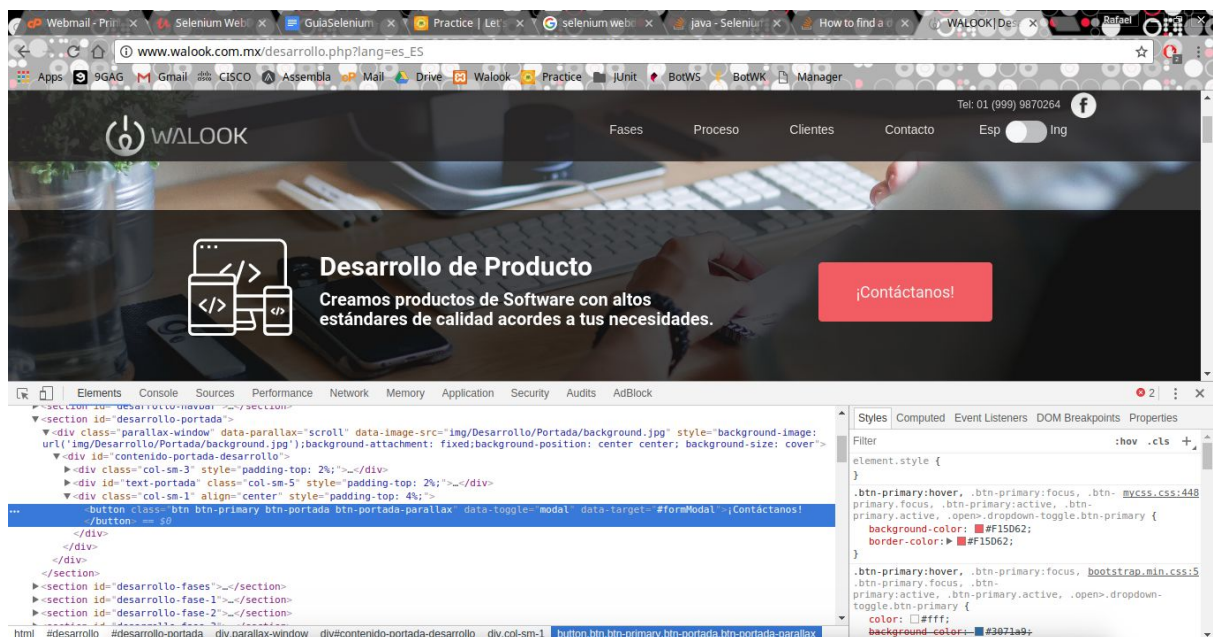
Cada elemento en una página web tiene asignada una etiqueta. Así, cada etiqueta de HTML contiene una serie de atributos que podemos usar para identificar elementos únicos. Por ejemplo, podemos usar el id, el nombre de la clase css que tiene asignado, el atributo name, etc. Además, podemos identificar objetos por el texto visible que tienen, como por ejemplo, identificar un botón o una etiqueta anchor por el texto visible entre la apertura y cierre de dichas etiquetas.

```
<input id="contact-name" type="text" class="form-control" placeholder="*Nombre"
maxlength="50" required="" data-validation-required-message="Por favor escriba su
nombre. ">
```

La etiqueta anterior corresponde a un cuadro de texto para introducir el nombre del contacto. Podemos ver que tiene atributos id, class o placeholder que podemos usar para identificarlo. Como regla general, los id para cada etiqueta deben ser únicos, así que son una buena forma de localizar elementos. Sin embargo existen casos donde los elementos no contienen un id, así que podemos usar otros atributos para encontrar el elemento.



Por ejemplo, el siguiente botón de contacto no cuenta con un id, por debemos encontrar otra forma de ubicarlo.



Otra forma de encontrar elementos es ubicando su ruta dentro del DOM. El DOM representa una estructura jerárquica en la que se encuentran todos los elementos. Es una estructura arborescente y a través de ella podemos acceder a elementos usando la ruta o XPath.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```

<title>Página de prueba</title>
</head>
<body>
  <div>
    <h1>Mi página</h1>
  </div>
  <div>
    <section>
      <button>Haz click aquí</button>
      <br />
      <a href="www.facebook.com">Página de Facebook</a>
    </section>
    <section>
      <button>Contacto</button>
    </section>
  </div>
</body>
</html>

```

En el código HTML anterior podemos ver que existe un botón de contacto que no cuenta con ningún atributo, así que una forma de obtenerlo en Selenium es hacer uso de su XPath: /html/body/div[2]/section[2]/button.

Métodos útiles para encontrar elementos web:

```

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;

```

```

import java.util.List;
import java.util.concurrent.TimeUnit;

```

```

public class FindWebElements {

    private WebDriver driver;
    private String baseUrl;
    private String url2;

    @Before
    public void setUp() throws Exception {
        driver = new ChromeDriver();
        baseUrl = "https://letscodeit.teachable.com/p/practice";
        url2 = "https://letscodeit.teachable.com/";

        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
    }
}

```


@Test

```
public void findWebElements() {
```

```
    driver.get(baseUrl);
```

```
    WebElement element;
```

```
    //Buscar elemento usando id
```

```
    element = driver.findElement(By.id("openwindow"));
```

```
    //Buscar elemento usando XPath
```

```
    element = driver.findElement(By.xpath("//*[@id='opentab']"));
```

```
    //Buscar un conjunto de radio buttons usando name
```

```
    element = driver.findElement(By.name("cars"));
```

```
    //Busca el primer elemento con la clase "btn-style"
```

```
    element = driver.findElement(By.className("btn-style"));
```

```
    driver.navigate().to(url2);
```

```
    //Busca un elemento cuyo texto entre la etiqueta sea "Login". Funciona para etiquetas anchor
```

```
    element = driver.findElement(By.linkText("Login"));
```

```
    //Busca un elemento cuyo texto entre la etiqueta contenga la subcadena "Prac".
```

```
    Funciona para etiquetas anchor
```

```
    element = driver.findElement(By.partialLinkText("Prac"));
```

```
    List<WebElement> elements;
```

```
    //Busca todos los elementos con la etiqueta anchor
```

```
    elements = driver.findElements(By.tagName("a"));
```

```
}
```

@After

```
public void tearDown() throws Exception {
```

```
    Thread.sleep(3000);
```

```
    driver.quit();
```

```
}
```

```
}
```

Recuperar elementos con atributos dinámicos o sin atributos: XPath y CSS Selector

Por regla general las etiquetas en HTML deben contener id únicos, sin embargo puede darse el caso en que las etiquetas no contengan el atributo id. También se puede dar el caso donde el atributo id no sea fijo, sino que se genera de manera automática conforme el usuario interactúa con la página. En estos casos es complicado poder obtener un elemento

web usando un atributo fijo como el id. Para solucionar esto podemos generar un XPath o un CSS Selector para los elementos que necesitemos.

CSS Selector

Sintaxis: `tag[attribute='value']`

Para generar un CSS Selector podemos usar la sintaxis anterior. `tag` es el nombre de la etiqueta HTML que queremos encontrar, `attribute` es un atributo de dicha etiqueta con el que podamos identificar el elemento y `value` es el valor de dicho atributo. Podemos usar también caracteres especiales para identificar el id o la clase: `"#"` representa id mientras que `"."` representa una clase. Por ejemplo: `#display-text` representa `id='display-text'` mientras que `.display-class` representa `class='display-class'`.

Ejemplo:

```
<input id="name" name="enter-name" class="inputs" placeholder="Enter Your Name" type="text">
```

`input#name` es equivalente a `input[id='name']`

`input.inputs` es equivalente a `input[class='inputs']`

En el ejemplo anterior usar `input[class='inputs']` funciona por que la etiqueta `input` sólo tiene una clase. Sin embargo, si la etiqueta tuviera más de una clase asignada y quisiéramos usar `input[class='class1']`, no recuperaremos nada. Por ejemplo:

```
<input id="displayed-text" name="show-hide" class="inputs displayed-class" placeholder="Hide/Show Example" type="text">
```

La etiqueta cuenta con varias clases. Al usar `input.inputs` obtendremos todas las etiquetas `input` que contengan la clase `inputs`, sin embargo, al usar `input[class='inputs']` no se recuperará nada. Para obtener esta etiqueta de la segunda forma es necesario usar todas las clases que contiene: `input[class='inputs displayed-class']`.

Usando el caracter `"."` para seleccionar elementos web por sus clases nos permite cierta versatilidad, es decir, podemos usar las clases que necesitemos para recuperar elementos. Y si deseamos encontrar un elemento único a través de sus clases, podemos adjuntar clases hasta encontrar el elemento. Por ejemplo, para la etiqueta anterior, `input.inputs` encontrará al menos dos elementos, pero `input.inputs.displayed-class` encontrará un solo elemento entre las dos etiquetas anteriores.

Escribir antes del atributo el nombre de la etiqueta sirve como filtro, pues nos recuperará todas las etiquetas de tipo `input` que tengan un `id='display-text'`. Si no fijamos una etiqueta antes, obtendremos todos los elementos en el DOM cuyo `id='display-text'`.

Existen caracteres especiales que funcionan como comodines al escribir los nombres de los atributos:

`"^"` representa el inicio del texto

“\$” representa el final del texto

“*” representa el texto contenido

Sintaxis: tag[attribute<caracter especial>='value']

Si en nuestro DOM se tienen las siguientes etiquetas:

Etiqueta 1:

```
<input id="name" name="enter-name" class="inputs" placeholder="Enter Your Name" type="text">
```

Etiqueta 2:

```
<input id="displayed-text" name="show-hide" class="inputs displayed-class" placeholder="Hide/Show Example" type="text">
```

Entonces:

input[class='inputs'] recupera la etiqueta 1.

input[class^='inputs'] recupera las dos etiquetas, pues ambas tienen una subcadena 'inputs' al principio del atributo class.

input[class='displayed-class'] no recupera ninguna etiqueta, pues no hay ninguna etiqueta cuyo atributo class sea exactamente igual a la cadena 'displayed-class'.

input[class\$='class'] recupera la etiqueta 2 pues la etiqueta 2 tiene un atributo class que termina con la subcadena 'class'.

input[class*='displayed-class'] recupera la etiqueta 2, pues la etiqueta 2 tiene un atributo class que contiene la subcadena 'displayed-class'.

Las reglas anteriores funcionan para cualquier atributo que las etiquetas contengan (placeholder, name, etc.), incluyendo los comodines, y no solo en el atributo class.

Una forma útil de filtrar elementos es seleccionar sólo los elementos que son hijos de una etiqueta. Por ejemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Página de prueba</title>
</head>
<body>
  <div>
    <h1>Mi página</h1>
  </div>
  <div>
    <section id="redes-sociales">
```

```

    <button>Haz click aquí</button>
    <br />
    <a href="www.facebook.com">Página de Facebook</a>
  </section>
  <section id="contacto">
    <button>Contacto</button>
  </section>
</div>
</body>
</html>

```

Si queremos recuperar el botón de contacto, podemos filtrar a través de la etiqueta `section`, usando el caracter `">"`. Así `section[id='contacto']>button` nos recuperará el botón de contacto, pues es el único botón de los hijos de `section[id='contacto']`. Podemos usar el caracter `">"` para movernos dentro de las ramas del DOM, además de que también podemos usar las reglas anteriores para encontrar elementos únicos.

XPath

No cada elemento contiene un id, un id estático o un link text único. Para esos elementos necesitamos construir un XPath para encontrarlos y realizar acciones sobre ellos.

Hay que recordar que sin importar qué usemos, id, name, XPath, etc., siempre debe ser único para identificar un solo elemento, a menos que lo que queramos sea encontrar un conjunto de elementos.

Para movernos entre las ramas del DOM, y encontrar elementos más profundos o hijos inmediatos de las ramas usaremos el caracter `"/"`. Un solo `"/"` quiere decir que hay que buscar el elemento inmediato que tenga como hijo el elemento padre. Por ejemplo

```

<div>
  <section id="redes-sociales">
    <button>Haz click aquí</button>
    <br />
    <a href="www.facebook.com">Página de Facebook</a>
  </section>
  <section id="contacto">
    <button>Contacto</button>
  </section>
</div>

```

La etiqueta `div` tiene dos elementos hijo, que son dos etiquetas `section`. Si usamos el XPath `div/section`, accederemos a la etiqueta `section` con el `id="redes-sociales"`, pues el uso de un solo `"/"` nos devuelve el elemento hijo inmediato.

Por otro lado, si usamos doble `"/"`, podemos encontrar cualquier elemento hijo que se encuentre disponible. El uso de doble `"/"` indica que se puede buscar en cualquier lado de las ramas de DOM a partir del elemento padre. Por ejemplo, usando el código anterior, si

tenemos el siguiente XPath `div//section[id="contacto"]`, podemos acceder a la etiqueta `section` con `id="contacto"` sin problemas. Si intentamos usar un solo `/` para encontrar dicha etiqueta, no encontraríamos ningún elemento pues dicha etiqueta no está inmediatamente adyacente a la etiqueta padre.

Sintaxis: `//tag[@attribute='value']`

Usamos `//` para indicar que se buscará en cualquier parte del DOM. `tag` es el nombre de la etiqueta que deseamos buscar. `@attribute` es el nombre del atributo de la etiqueta no nos ayudará a identificar al elemento y `value` es el valor de dicho atributo.

Podemos generar dos tipos de rutas XPath: absolutas y relativas. Las rutas absolutas son rutas que explícitamente muestran toda la jerarquía del elemento dentro del DOM, comenzando desde el HTML. Por ejemplo, usando el siguiente código, si queremos obtener la ruta absoluta para la etiqueta `section` con el `id="contacto"` esta quedaría así:

`html/body/div[2]/section[2]`. Algo que es importante destacar es que usar las rutas absolutas no suele ser lo más adecuado, pues si en algún momento la página recibe alguna modificación que involucre a nuestra ruta, la ruta que generamos no funcionará más. Para ello es mejor usar rutas relativas.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Página de prueba</title>
</head>
<body>
  <div>
    <h1>Mi página</h1>
  </div>
  <div>
    <section id="redes-sociales">
      <button>Haz click aquí</button>
      <br />
      <a href="www.facebook.com">Página de Facebook</a>
    </section>
    <section id="contacto">
      <button>Contacto</button>
    </section>
  </div>
</body>
</html>
```

Las rutas relativas permiten generar rutas más pequeñas basadas en elementos web fijos. En vez de buscar un elemento web desde la raíz del DOM, podemos ubicar un elemento hijo del DOM que sea único y a partir de ahí buscar más abajo en la jerarquía. Siguiendo el ejemplo anterior podemos ubicar la etiqueta `section` con `id="contacto"` de la siguiente forma: `//section[@id='contacto']`. Como podemos ver nuestro XPath resulta mucho más

corto comparado con la ruta absoluta. Además, si el DOM sufre cambios pero la etiqueta section no, aún podemos ubicarla fácilmente.

Ahora, si la etiqueta section que deseamos encontrar no contiene un id y no contiene atributos únicos para encontrarla con facilidad, podemos ubicarla a partir de las etiquetas padre, ubicando una que sea única y bajando a partir de ahí en la jerarquía. Por ejemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Página de prueba</title>
</head>
<body>
  <div>
    <h1>Mi página</h1>
  </div>
  <div id="botones">
    <section>
      <button>Haz click aquí</button>
      <br />
      <a href="www.facebook.com">Página de Facebook</a>
    </section>
    <section>
      <button>Contacto</button>
    </section>
  </div>
</body>
</html>
```

En el código anterior vemos que no tenemos manera inmediata de ubicar la etiqueta section, así que usando una ruta relativa podemos acceder a ella usando `//div[@id="botones"]//section[2]`. Usando `//` podemos ubicar una etiqueta div única con un id="botones", y a partir de ella podemos ubicar a section[2] como uno de sus hijos, y que corresponde a la etiqueta que buscábamos.

Problemos otro ejemplo usando el código anterior. Supongamos que ahora queremos encontrar el botón con el texto "Contacto". Nuevamente tenemos un problema pues el botón no tiene un id o atributo con cuál identificarlo. Para ello podemos usar la función `text()` que nos recuperará el elemento con cuyo texto estemos buscando. En la etiqueta div con id="botones" podemos ver que tenemos dos botones. Si sólo queremos identificar uno podemos usar `//div[@id="botones"]//button[text()='Contacto']`. Al usar `text()='Contacto'` estamos ubicando un botón cuyo texto sea igual a "Contacto" dentro de la etiqueta padre designada. Esto funciona porque sólo existe un botón con el texto "Contacto". Si tuviéramos más botones con dicho texto sería necesario refinar más la ruta para llegar a un elemento único.

Buscar elementos por su texto puede ser útil cuando conoces el texto exacto, sin embargo, ubicar el texto usando `text()='Contacto'` puede traer problemas si el texto contiene espacios en blanco o no conocemos el texto exacto. Para ello podemos usar `//div[@id="botones"]//button[contains(text(), 'Contac')]`. Con `contains()` podemos ubicar subcadenas fácilmente. También podemos ubicar clases, especialmente por que el

XPath sigue las mismas reglas al que un CSS Selector al escribir las clases contenidas en una etiqueta, es decir, debemos escribir exactamente todas las clases y en el orden que aparecen para ubicar el elemento. Si usamos `contains()` esta restricción desaparece. Además, podemos agregar más de una restricción para encontrar un elemento. Por ejemplo: `//div[@id='navbar']/a[contains(@class, 'navbar-link') and contains(@href, 'sign_in')]`. También podemos usar `or` para encontrar la ruta.

Revisando el siguiente código, si queremos encontrar un botón usando las clases como identificador único y usando `contains()` podemos encontrar problemas, pues `contains()` busca la subcadena en cualquier parte del atributo. Así si usamos `//button[contains(@class, 'class1')]` encontraremos que los dos botones usan esa clase.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Página de prueba</title>
</head>
<body>
  <div>
    <h1>Mi página</h1>
  </div>
  <div id="botones">
    <section id="redes-sociales">
      <button class="class1">Haz click aquí</button>
      <br />
      <a href="www.facebook.com">Página de Facebook</a>
    </section>
    <section id="contacto">
      <button class="class1 class2">Contacto</button>
    </section>
  </div>
</body>
</html>
```

Podemos usar entonces `//button[starts-with(@class, 'class1')]` para ubicar solo el primer botón. Con esto lo que estamos diciendo es que buscamos un botón que en el valor del atributo `class`, la cadena que representa su valor comience con `class1`.

Algo que nos puede ser útil para ubicar elementos es encontrar los elementos hermanos de una etiqueta o la etiqueta padre. Para ello podemos usar lo siguiente:

```
xpath-to-some-element//parent::<tag>
```

```
xpath-to-some-element//preceding-sibling::<tag>
```

```
xpath-to-some-element//following-sibling::<tag>
```

Una vez que encontremos un elemento único, podemos indicarle cuál es su elemento padre, hermano anterior o hermano siguiente usando las sintaxis anteriores. Es importante no olvidar añadir el nombre de la etiqueta de dicho elemento a buscar.

```

<div class='collapse navbar-collapse navbar-header-collapse'>
  <ul class='nav navbar-nav navbar-right'>
    <li>
      <a class='fedora-navbar-link navbar-link' href='/pages/practice'
target=''>
        Practice
      </a>
    </li>
    <li>
      <a class='navbar-link fedora-navbar-link' href='/sign_in'>
        Login
      </a>
    </li>
    <li>
      <a class='btn btn-primary pull-right btn-lg' href='/sign_up'>
        Sign Up
      </a>
    </li>
  </ul>
</div>

```

Para el código anterior tenemos el siguiente XPath:

```
//a[@href='/sign_in']//parent::li//preceding-sibling::li//following-sibling::li[2]
```

Primero encontramos un elemento único, después accedemos a su etiqueta padre, luego accedemos a la etiqueta hermana que le antecede y por último accedemos a las etiquetas que le preceden, filtrando a una sola etiqueta del listado usando li[2].

Algo que debemos saber es que todas las reglas y consejos de uso de XPath se pueden combinar para encontrar los elementos web que deseemos.

Un ejemplo práctico

Digamos que el atributo “id” de un campo de nombre de usuario es 'username_123' y el XPath será // * [@ id = 'username_123'], pero cuando se vuelve a abrir la página, el atributo 'id' del 'nombre de usuario' puede haber cambiado y el nuevo valor puede ser 'username_234'.

En este caso, la prueba fallará porque el Selenium no pudo encontrar el XPath que ha pasado anteriormente ya que la identificación del campo ha cambiado a algún otro valor.

Podemos ver que “Username_123” ha cambiado a “username_234”, pero “username” siempre ha permanecido constante. Podemos construir un XPath de la siguiente manera:

```
driver.findElement(By.xpath ("//input[contains(@id, 'username')]")). sendKeys
("username");
```

o

```
driver.findElement(By.xpath ("//input[starts-with(@id, 'user')]")). sendKeys
("username");
```