Validación y Pruebas

(Selenium)

UNIVERSIDAD DE BURGOS

Autores:

Mario Ubierna San Mamés

Jorge Navarro González

Tabla de contenido

Introducción

En este caso a nosotros nos ha tocado la siguiente combinación:

Ruby / Test::Unit / Remote Control

En esta práctica vamos a tener que acceder a una página web en la que exista alguna clase de formulario que tengamos que rellenar, dicho formulario deberá de contener ciertos campos que sean obligatorios y otros campos que sean opcionales.

Además tendremos que rellenar este cuestionario de manera correcta, y también con algún error, es decir, dos veces.

En nuestro caso como hemos dicho instalaremos el Selenium y grabaremos los dos formularios que vamos a rellenar. A continuación, exportaremos todos los ficheros y analizaremos el código que ha generado Selenium.

Descripción de la Web y los Casos de Prueba

En nuestro caso elegimos la siguiente web:

https://puntosdevistadiferentes.wordpress.com/contacto/

Ya que resultaba un cuestionario sencillo en el que tenemos tres campos obligatorios a rellenar y uno de ellos opcional (el sitio web) cumpliendo así con lo requerido en el ejercicio.

En cuanto a los casos de prueba vamos a tener dos tipos de casos de prueba, en ninguno de ellos vamos a rellenar el campo opcional y en uno de ellos vamos a dejar sin rellenar el comentario obligatorio que debemos hacer, con lo cual va a fallar.

1- En el primer caso de prueba, que va a ser el exitoso, rellenaremos todos los campos excepto el opcional, con lo cual no nos dio ningún problema ya que ese campo no era necesario de rellenar.
2- En el segundo caso de prueba, el erróneo, rellenamos los campos excepto el opcional y también el último que es un comentario que hay que hacer de forma obligatoria, con lo cual nos dios fallo, no dejando crear el formulario.

Análisis del Script

Lo primero que nos vamos a encontrar en el Script es una cosa común en ambos:

```
require "test/unit"
require "rubygems"
gem "selenium-client"
require "selenium/client"
```

En estas líneas es donde se cogerán las librerías que vamos a necesitar.

Después ya empezaríamos con el código, en el cual tendríamos alguna información sobre nuestro ordenador:

```
def setup
  @verification_errors = []
  @selenium = Selenium::Client::Driver.new \
    :host => "localhost",
    :port => 4444,
    :browser => "*chrome",
    :url => "https://puntosdevistadiferentes.wordpress.com",
    :timeout_in_second => 60

  @selenium.start_new_browser_session
end
```

```
def setup
  @verification_errors = []
  @selenium = Selenium::Client::Driver.new \
    :host => "localhost",
    :port => 4444,
    :browser => "*chrome",
    :url => "https://puntosdevistadiferentes.wordpress.com/contacto/",
    :timeout_in_second => 60

  @selenium.start_new_browser_session
end
```

Como podemos ver tenemos alguna información como el host que estamos usando, en la línea :host => “localhost”, y así otras informaciones como el buscador que tenemos que en este caso es Chrome, la URL donde hemos

iniciado nuestro Selenium y también el tiempo que tendremos para conectarnos con el driver que serían 60 segundos.

Aunque ponga que nuestro buscador es Chrome, nosotros lo hicimos con el Firefox Portable que se encuentra en la plataforma de UbuVirtual.

Se puede observar que la URL que tenemos en el caso de fallo, es distinta a la URL que tenemos en el éxito, esto es debido al momento en el que empezamos a grabar, nos encontrábamos en un punto de la web distinto.

```
def teardown
  @selenium.close_current_browser_session
  assert_equal [], @verification_errors
end
```

En esta parte del código es donde cerraremos la sesión.

Caso de Éxito

```
def test_blog_exito_exportado
  @selenium.open "/contacto/"
  @selenium.type "id=g3-nombre", "PruebaValidacion"
  @selenium.type "id=g3-correoelectrnico", "pruebavalidacion1997@gmail.com"
  @selenium.type "id=contact-form-comment-g3-comentario", "Hola, esto es un prueba para la Universidad de Burgos, no te asustes =)"
  @selenium.click "css=input.pushbutton-wide"
  @selenium.wait_for_page_to_load "30000"
end
```

Este sería la función en la que se rellenan todos los campos:

- @selenium.open “/contacto/” podemos ver que se abre la pestaña de contacto en la que está el formulario.
- @selenium.type tenemos tres, que se corresponden con cada uno de los campos que vamos a rellenar, tendríamos como podemos ver el nombre, el correo electrónico y el comentario. No está el apartado de sitio web ya que es un campo opcional no rellenado.
- @selenium.click nos dice que hemos pulsado en el botón de “Enviar” una vez rellenado nuestro cuestionario.
- @selenium.wait_for_page_to_load “3000” son los milisegundos que vamos a esperar para que la página cargue, como dice la propia etiqueta traducida.

Caso de Error

```
def test_blog_fallo_exportado
  @selenium.open "/contacto/"
  @selenium.type "id=g3-nombre", "PruebaValidacionFallo"
  @selenium.type "id=g3-correoelectrnico", "pruebavalidacion1997@gmail.com"
  @selenium.click "css=input.pushbutton-wide"
end
```

En este caso, es igual que el anterior, pero vemos que es más corto ya que da el fallo y rellenamos menos campos. Al igual que antes las etiquetas son:

- @selenium.open "/contacto/" para abrir la pestaña de contacto donde se encuentra el formulario.
- @selenium.type al igual que antes son los campos que estamos rellenando, en nuestro caso solo el nombre y el correo electrónico.
- @selenium.click sería el click en "Enviar" que hacemos una vez el cuestionario está realizado.

Suponemos que la espera para que la web cargue, no se ha dado debido a que al dar error, no tiene que esperar a que cargue nada.

Explicación de los tres componentes asignados

Ruby / Test::Unit / Remote Control

Lenguaje Ruby

Este lenguaje es orientado a objetos y fue creado por un programador japonés llamado Yukihiro "Matz" Matsumoto, que lo pondría público en 1995. Combina varios lenguajes de programación como son Python, Perl, Lisp, Lua, Dylan y CLU, es una licencia de software libre y tiene algunas características como son:

- En ruby es todo un objeto.
- Combina lenguajes funcionales con imperativos orientados a objetos.
- Los elementos tienen un comportamiento bastante parecido, pero se puede personalizar para conveniencias de cada usuario.
- Tiene una funcionalidad de bloques que viene dada por los lenguajes funcionales en el que podemos hacer bloques de código a los que llamaremos cláusulas a cualquier método, de esta forma podemos decir cómo queremos que actúe.

- La manera para llamar a las variables es la siguiente: var (variable local), @var (variable de instancia) y $var (para una variable global), de esta forma permite que podamos identificarlos con facilidad.
- También tiene manejo de excepciones.
- Corre tanto en sistemos Windows, como Linux, Unix, dos, OS/2…
- Tiene un manejo de hilos independientemente del sistema operativo.

Framework Test::Unit

En Ruby este framework fue obligatorio hasta la 1.8, a partir de la versión 2.2 se quedó obsoleto y solo era un conjunto de minitest.

Este framework es un framework de test unitarios de ruby que está basado en los principios de xUnit. Permite escribir tests, verificar los resultados y tests automáticos en Ruby.

Los tests unitarios de Ruby son muy útiles ya que te permiten no solo escribir códigos de test sino tener esos tests, es decir poder tenerlos guardados para ejecutarlos cuando quieras de tal forma que podamos cambiar algo de nuestra aplicación y aun así podamos seguir ejecutando ese test y no usarlo una vez y deshacernos de él. Nos es muy útil ya que nos ayuda a hacer debug y evaluar nuestro código.

La principal idea que tienen estos test unitarios son que tú creas un método test con algunas aserciones que tú quieras en tu código, estos métodos se van metiendo en una “suite” de test de tal manera que podemos ejecutarlos cuando queramos.

En Test::Unit tenemos diversas partes, primero tendremos las aserciones las cuales nosotros ponemos y si se cumple la aserción todo irá correctamente y si no se cumple se propagará un error. Después tendríamos el “Test Method” que es una clase de test fuera del propio código. También tendremos un conjunto de datos común para los tests que estarán dentro de otro elemento del framework llamado “Test Fixture”. También tenemos los llamados “Test Runners” que son gracias a los cuales podemos ejecutar dichos tests y por último tendremos el “Test Suite” que es donde podremos agrupar a un grupo de test, y también podremos meter en un “Test Suite” otro “Test Suite” distinto.

La API Remote Control

Primero lo que se hace es que el router elegirá el controlador más apropiado para ser usado, y entonces lo que hace será enviarle una petición a dicho controlador, a continuación lo que se hará es que el controlador cogerá esa petición haciéndola útil para nosotros y dándole un poco de orden y sentido, ya que hacen de intermediario entre los modelos y las vistas, tienen que coger los datos y ponerlos de manera que puedan ser vistos por el usuario de una forma entendible, y por último nos dará la salida óptima. Este controlador hará la mayor parte del trabajo por nosotros.

La petición que recibe el controlador es invisible a nosotros y lo que hace es generar un salida HTML del modelo y los datos que tiene. No es problema que el controlador tenga que hacer algunas cosas algo distintas a lo normal, ya que esta es la forma en la que trabajan los controladores.

Bibliografía

Información sobre Ruby:

https://es.wikipedia.org/wiki/Ruby

https://es.wikibooks.org/wiki/Programaci%C3%B3n_en_Ruby/Caracter%C3%ADsticas_especiales_del_lenguaje

Framework Test::Unit:

https://ruby-doc.org/stdlib-1.8.7/libdoc/test/unit/rdoc/Test/Unit.html

API Remote Control:

https://guides.rubyonrails.org/action_controller_overview.html#http-authentications

Otros Links:

http://rubytutorial.wikidot.com/incluir-ficheros