



© JMA 2019. All rights reserved

INTRODUCCIÓN

© JMA 2019. All rights reserved

Introducción

- El Selenium es un conjunto de herramientas para automatizar los navegadores web, robot que simula la interacción del usuario con el navegador, originalmente pensado como entorno de pruebas de software para aplicaciones basadas en la web.
- Como principales herramientas Selenium cuenta con:
 - Selenium IDE:
 - una herramienta para grabar y reproducir secuencias de acciones con el navegador que permite crear pruebas sin usar un lenguaje de scripting para pruebas.
 - Selenium Core:
 - API para escribir pruebas automatizadas y de regresión en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, Php y Python.
 - WebDriver:
 - interfaces que permite ejecutar las pruebas de forma nativa usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX.
 - Selenium Grid:
 - Permite ejecutar muchas pruebas de un mismo grupo en paralelo o pruebas en múltiples entornos. Tiene la ventaja que un conjunto de pruebas muy grande puede dividirse en varias máquinas remotas para una ejecución más rápida o si se necesitan repetir las mismas pruebas en múltiples entornos.

© JMA 2019. All rights reserved

Historia

- Cuando Selenium 1 fue lanzado en el año 2004, surgió por la necesidad de reducir el tiempo dedicado a verificar manualmente el comportamiento consistente en el front-end de una aplicación web. Hizo uso de las herramientas disponibles en ese momento y confió en gran medida en la inyección de JavaScript en la página web bajo prueba para emular la interacción de un usuario.
- Si bien JavaScript es una buena herramienta para permitirte la introspección de las propiedades del DOM y para hacer ciertas observaciones del lado del cliente que de otro modo no se podría hacer, se queda corto en la capacidad de replicar naturalmente las interacciones de un usuario como usar el mouse y el teclado.
- Desde entonces, Selenium ha crecido y ha madurado bastante, convirtiéndose en una herramienta ampliamente utilizada. Selenium ha pasado de ser de un kit de herramientas de automatización de pruebas de fabricación casera a la librería de facto de automatización de navegadores del mundo.
- Así como Selenium RC hizo uso de las herramientas de oficio disponibles en ese momento, Selenium WebDriver impulsa esta tradición al llevar la parte de la interacción del navegador al territorio del proveedor del mismo y pedirles que se responsabilicen de las implementaciones de back-end orientadas al navegador. Recientemente este esfuerzo se ha convertido en un proceso de estandarización del W3C donde el objetivo es convertir el componente WebDriver en Selenium en la librería de control remoto de tu agente de usuario.

© JMA 2019. All rights reserved

Selenium controla los navegadores web

- Selenium significa muchas cosas pero en su núcleo, es un conjunto de herramientas para la automatización de navegadores web que utiliza las mejores técnicas disponibles para controlar remotamente las instancias de los navegadores y emular la interacción del usuario con el navegador.
- Permite al código simular interacciones básicas realizadas por los usuarios finales; insertando texto en los campos, seleccionando valores de menús desplegables y casillas de verificación, y haciendo clics en los enlaces de los documentos. También provee muchos otros controles tales como el movimiento del mouse, la ejecución arbitraria de JavaScript y mucho más.
- A pesar de que es usado principalmente para pruebas de front-end de sitios webs, Selenium es en esencia una librería de agente de usuario para el navegador. Las interfaces son ubicuas a su aplicación, lo que fomenta la composición con otras librerías para adaptarse a su propósito.

© JMA 2019. All rights reserved

Un API para gobernarlos a todos

- Uno de los principios fundamentales del proyecto es permitir una interfaz común para todas las tecnologías de los (principales) navegadores.
- Los navegadores web son aplicaciones increíblemente complejas y de mucha ingeniería, realizando operaciones completamente diferentes pero que usualmente se ven iguales al hacerlo. Aunque el texto se presente con las mismas fuentes, las imágenes se muestren en el mismo lugar y los enlaces te lleven al mismo destino.
- Lo que sucede por debajo es tan diferente como la noche y el día. Selenium abstraer estas diferencias, ocultando sus detalles y complejidades a la persona que escribe el código. Esto le permite escribir varias líneas de código para realizar un flujo de trabajo complicado, pero estas mismas líneas se ejecutarán en Firefox, Internet Explorer, Chrome y los demás navegadores compatibles.

© JMA 2019. All rights reserved

Herramientas y soporte

- El diseño minimalista de Selenium le da la versatilidad para que se pueda incluir como un componente en otras aplicaciones. La infraestructura proporcionada debajo del catálogo de Selenium te da las herramientas para que puedas ensamblar tu grid de navegadores de modo que tus pruebas se puedan ejecutar en diferentes navegadores a través de diferentes sistemas operativos y plataformas.
- Imagina un granja de servidores en tu sala de servidores o en un centro de datos, todos ejecutando navegadores al mismo tiempo e interactuando con los enlaces en tu sitio web, formularios y tablas—probando tu aplicación 24 horas al día. A través de la sencilla interfaz de programación proporcionada para los lenguajes más comunes, estas pruebas se ejecutarán incansablemente en paralelo, reportando cuando ocurran errores.
- Es un objetivo ayudar a que esto sea una realidad para ti, proporcionando a los usuarios herramientas y documentación para controlar no solo los navegadores pero también para facilitar la escalabilidad e implementación de tales grids.

© JMA 2019. All rights reserved

Automatización de pruebas

- Las pruebas funcionales de usuario final, como las pruebas de Selenium son caras de ejecutar, requieren abrir un navegador e interactuar con él. Además, normalmente requieren que una infraestructura considerable este disponible para estas ejecutarse de manera efectiva. Es una buena regla preguntarse siempre si lo que se quiere probar se puede hacer usando enfoques de prueba más livianos como las pruebas unitarias o con un enfoque de bajo nivel.
- Las pruebas manuales son muy costosas y difícilmente repetibles, por lo que se impone una estrategia de automatización. Selenium permite ejecutar y repetir las mismas pruebas en múltiples navegadores de diferentes sistemas operativos.
- Una vez tomada la decisión de hacer pruebas en el navegador, y que tengas tu ambiente de Selenium listo para empezar a escribir pruebas, generalmente realizaras alguna combinación de estos tres pasos:
 - Preparar los datos
 - Realizar un conjunto discreto de acciones
 - Evaluar los resultados
- Querrás mantener estos pasos tan cortos como sea posible; una o dos operaciones deberían ser suficientes la mayor parte del tiempo. La automatización del navegador tiene la reputación de ser “frágil”, pero en realidad, esto se debe a que los usuarios suelen exigirle demasiado.
- Manteniendo las pruebas cortas y usando el navegador web solo cuando no tienes absolutamente ninguna alternativa, puedes tener muchas pruebas con fragilidad mínima.
- Una clara ventaja de las pruebas de Selenium es su capacidad inherente para probar todos los componentes de la aplicación, desde el backend hasta el frontend, desde la perspectiva del usuario. En otras palabras, aunque las pruebas funcionales pueden ser caras de ejecutar, también abarcan a la vez grandes porciones críticas para el negocio.

© JMA 2019. All rights reserved

Tipos de pruebas

- **Pruebas funcionales:** Este tipo de prueba se realiza para determinar si una funcionalidad o sistema funciona correctamente y sin problemas. Se comprueba el sistema en diferentes niveles para garantizar que todos los escenarios están cubiertos y que el sistema hace lo que se supone que debe de hacer. Es una actividad de verificación que responde la pregunta: ¿Estamos construyendo el producto correctamente?
 - Para aplicaciones web, la automatización de esta prueba puede ser hecha directamente con Selenium simulando los retornos esperados. Esta simulación podría hacerse mediante grabación/reproducción o mediante los diferentes lenguajes soportados.
- **Pruebas de aceptación:** Este tipo de prueba se realiza para determinar si una funcionalidad o un sistema cumple con las expectativas y requerimientos del cliente. Este tipo de pruebas implican la cooperación o retroalimentación del cliente, siendo una actividad de validación que responde la pregunta: ¿Me están construyendo el producto correcto?
 - Las pruebas de aceptación son un subtipo de pruebas funcionales, por lo que la automatización se puede hacer directamente con Selenium simulando el comportamiento esperado del usuario mediante grabación/ reproducción.

© JMA 2019. All rights reserved

Tipos de pruebas

- **Pruebas de regresión:** Este tipo de pruebas generalmente se realiza después de un cambio, corrección o adición de funcionalidad.
 - Para garantizar que el cambio no ha roto ninguna de las funcionalidades existentes, algunas pruebas ya ejecutadas se ejecutan nuevamente. El conjunto de pruebas ejecutadas nuevamente puede ser total o parcial, y puede incluir varios tipos diferentes, dependiendo del equipo de desarrollo y la aplicación.
- Las **pruebas no funcionales** tales como rendimiento, de carga, de estrés, ... aunque se pueden realizar con Selenium hay otras opciones, como Jmeter, que las realizan mas eficientemente y además suministran métricas que incluyen rendimiento, latencia, pérdida de datos, tiempos de carga de componentes individuales.
- De igual forma, las **pruebas unitarias** se pueden realizar con Selenium pero, como se deben ejecutar continuamente, suele ser demasiado costoso.

© JMA 2019. All rights reserved

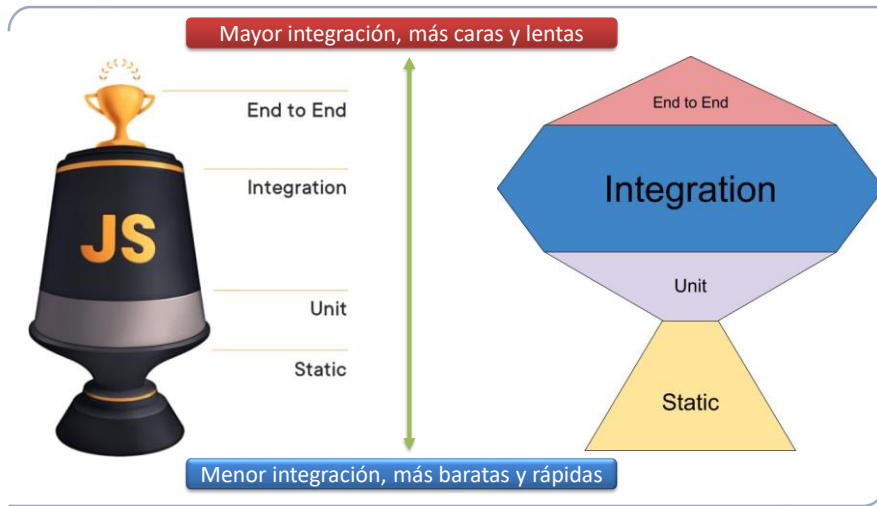
Pirámide de pruebas



El Trofeo de Pruebas

- Testing Trophy es un método de pruebas propuesto por Kent C. Dodds *para aplicaciones web*. Se trata de escribir suficientes pruebas, no muchas, pero si las pruebas correctas: proporciona la mejor combinación de velocidad, costo y confiabilidad.
- Se superpondrán estas técnicas:
 - Usar un sistema de captura de errores de tipo, estilo y de formato utilizando linters, formateadores de errores y verificadores de tipo (ESLint, SonarQube, ...).
 - Escribir pruebas unitarias efectivas que apunten solo al comportamiento crítico y la funcionalidad de la aplicación.
 - Desarrollar pruebas de integración para auditar la aplicación de manera integral y asegurarse de que todo funcione correctamente en armonía.
 - Crear pruebas funcionales de extremo a extremo (e2e) para pruebas de interacción automatizadas de las rutas críticas y los flujos de trabajo más utilizados por los usuarios.

Testing Trophy



© JMA 2019. All rights reserved

Principios F.I.R.S.T.

- El principio FIRST fue definido por Robert Cecil Martin en su libro Clean Code. Este autor, entre otras muchas cosas, es conocido por ser uno de los escritores del Agile Manifesto, escrito hace más de 15 años y que a día de hoy se sigue teniendo muy en cuenta a la hora de desarrollar software.
 - Fast: Los tests deben ser rápidos, del orden de milisegundos, hay cientos de tests en un proyecto.
 - Isolated/Independent (Aislado/Independiente): Los tests no deben depender del entorno ni de ejecuciones de tests anteriores.
 - Repeatable: Los tests deben ser repetibles y ante la misma entrada de datos, producen los mismos resultados.
 - Self-Validating: Los tests tienen que ser autovalidados, es decir, NO debe de existir la intervención humana en la validación.
 - Thorough and Timely (Completo y oportuno): Los tests deben de cubrir el escenario propuesto, no el 100% del código, y se han de realizar en el momento oportuno.

© JMA 2019. All rights reserved

Proceso de Prueba

- Descomponer la aplicación web existente para identificar qué probar
- Identificar con qué navegadores probar
- Elige el mejor lenguaje para ti y tu equipo.
- Configura Selenium para que funcione con cada navegador que te interese.
- Escriba pruebas de Selenium mantenibles y reutilizables que serán compatibles y ejecutables en todos los navegadores.
- Cree un circuito de retroalimentación integrado para automatizar las ejecuciones de prueba y encontrar problemas rápidamente.
- Configura tu propia infraestructura o conéctate a un proveedor en la nube.
- Mejora drásticamente los tiempos de prueba con la paralelización
- Mantente actualizado en el mundo Selenium.

© JMA 2019. All rights reserved

Que probar

- La navegación
- La interacciones con la página y sus elementos
- El rellenado de formularios y sus validaciones
- El arrastrar y soltar, si procede
- La estética, presencia y visualización de elementos esenciales, con especial atención al responsive design.
- Moverse entre ventanas y marcos (aunque están prohibidos por la WAI)
- Uso de cookies, Local Storage, Session Storage, Service Worker, ...

© JMA 2019. All rights reserved

INSTALACIÓN

© JMA 2019. All rights reserved

Node.js: Entorno en tiempo de ejecución

- Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google. Fue creado con el enfoque de ser útil en la creación de programas de red altamente escalables, como por ejemplo, servidores web.
- Al contrario que la mayoría del código JavaScript, no se ejecuta en un navegador, sino en el servidor.
- Descargar e instalar: <https://nodejs.org>
- Verificar desde consola de comandos:
 - node --version

© JMA 2019. All rights reserved

npm: Node Package Manager

- Npm es el administrador de paquetes estándar para Node.js. Aunque se instala con el Node es conveniente actualizarlo:
 - `npm update -g npm`
- Verificar desde consola de comandos:
 - `npm --version`
- Configuración:
 - `npm config edit`
 - `proxy=http://usr:pwd@proxy.dominion.com:8080` ← Símbolos: %HEX ASCII
- Generar fichero de dependencias `package.json`:
 - `npm init`
- Instalación de paquetes:
 - `npm install -g grunt-cli grunt-init karma karma-cli` ← Global (CLI)
 - `npm install jasmine-core tslint --save --save-dev`
 - `npm install` ← Dependencias en `package.json`
- Arranque del servidor:
 - `npm start`

© JMA 2019. All rights reserved

npx

- npx es un ejecutor de paquetes binarios de npm.
- npx es una herramienta destinada a ayudar a completar la experiencia de usar paquetes del registro de npm: de la misma manera que npm hace que sea muy fácil instalar y administrar dependencias alojadas en el registro, npx facilita el uso de herramientas CLI y otros ejecutables alojados en el registro.
- Para evitar tener que instalar globalmente herramientas de uso infrecuente pero que cambian continuamente, npx descarga, ejecuta y descarta las herramientas sin requerir instalación.
- Disponible desde la versión `npm@5.2.0`, se puede instalar manualmente con:
 - `npm i -g npx`
- Para ejecutar un comando al vuelo:
 - `npx create-react-app my-react-app`

© JMA 2019. All rights reserved

Analizadores de código

- Los analizadores de código son herramientas que realizan la lectura del código fuente y devuelve observaciones o puntos en los que tu código puede mejorarse desde la percepción de buenas prácticas de programación y código limpio.
- JSHint es un analizador online de código JavaScript (basado en el JSLint creado por Douglas Crockford) que nos permitirá mostrar puntos en los que tu código no cumpla unas determinadas reglas establecidas de “código limpio”.
- El funcionamiento de JSHint es el siguiente: toma nuestro código, lo escanea y, si encuentra un problema, devuelve un mensaje describiéndolo y mostrando su ubicación aproximada.
- Para descargar e instalar:
 - `npm install -g jshint`
- Existen “plug-in” para la mayoría de los entornos de desarrollo (<http://jshint.com>). Se puede automatizar con GRUNT o GULP.

© JMA 2019. All rights reserved

ESLint

- ESLint (<https://eslint.org/>) es una herramienta para identificar e informar sobre patrones encontrados en código ECMAScript/JavaScript, con el objetivo de hacer que el código sea más consistente y evitar errores.
- Se puede instalar ESLint usando npm:
 - `npm install eslint --save-dev`
- Luego se debe crear un archivo de configuración `.eslintrc.json` en el directorio, se puede crear con `--init`:
 - `npx eslint --init`
- Se puede ejecutar ESLint con cualquier archivo o directorio:
 - `npx eslint **/*.js`

© JMA 2019. All rights reserved

Drivers

- Instalar los WebDriver:
 - Crear una carpeta y referenciarla en el PATH del sistema.
 - Windows: `setx /m path "%path%;C:\Selenium\WebDriver\"`
 - macOS y Linux: `export PATH=$PATH:/opt/Selenium/WebDriver >> ~/.profile`
 - Descargar los drivers (<https://www.selenium.dev/downloads/>)
 - Copiarlos descomprimidos en la carpeta creada.
- Plataformas compatibles con Selenium:
 - Firefox: GeckoDriver está implementado y soportado por Mozilla.
 - Internet Explorer: Solo se admite la versión 11 y requiere una configuración adicional.
 - Safari: SafariDriver es compatible directamente con Apple.
 - Ópera: OperaDriver es compatible con Opera Software.
 - Chrome: ChromeDriver es compatible con el proyecto Chromium.
 - Edge: EdgeDriver está implementado y soportado por Microsoft.

© JMA 2019. All rights reserved

Local

- Instalación de Eclipse.
- Descargar y descomprimir API:
<https://www.selenium.dev/downloads/>
- Creación del proyecto:
 - Create a Java Project
 - Add Library JUnit 5
 - Add External JARs: API descomprimida
 - Crear paquete de tests
 - Añadir JUnit Test Case
 - Ejecutar test
 - Run As → JUnit Test Case.

© JMA 2019. All rights reserved

Maven

- Instalación de Eclipse.
- Creación del proyecto:
 - Create a New Maven Project
 - Create a simple project (skip archetype selection)
 - Editar pom.xml
 - Añadir dependencias
 - Seleccionar src/test/java
 - Añadir JUnit Test Case
 - Ejecutar test
 - Run As → JUnit Test Case.

© JMA 2019. All rights reserved

pom.xml

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>${junit.jupiter.version}</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>${selenium.version}</version>
  <scope>test</scope>
</dependency>
```

© JMA 2019. All rights reserved

Dependencias únicas

- La dependencia selenium-java permite la ejecución del proyecto de automatización en todos los navegadores compatibles con Selenium.
- Si solo es necesario ejecutar las pruebas en un navegador en específico, se puede sustituir dependencia selenium-java por la dependencia para ese navegador en el archivo pom.xml. Por ejemplo, se debe agregar la siguiente dependencia en el archivo pom.xml para ejecutar las pruebas solamente en Chrome:

```
<dependency>  
  <groupId>org.seleniumhq.selenium</groupId>  
  <artifactId>selenium-chrome-driver</artifactId>  
  <version>3.X</version>  
</dependency>
```

© JMA 2019. All rights reserved

SELENIUM IDE

© JMA 2019. All rights reserved

Introducción

- Es el entorno de desarrollo integrado para pruebas con Selenium que permite grabar, editar y depurar fácilmente las pruebas. Es una solución simple y llave en mano para crear rápidamente pruebas de extremo a extremo confiables.
- Selenium IDE no requiere una configuración adicional aparte de instalar una extensión en el navegador pero solo está disponible para Firefox y Chrome.
- Selenium IDE es muy flexible, registra múltiples localizadores para cada elemento con el que interactúa, si un localizador falla durante la reproducción, los demás se probarán hasta que uno tenga éxito.
- Mediante el uso del comando de ejecución se puede reutilizar un caso de prueba dentro de otro.
- Dispone una estructura de flujo de control extensa, con comandos condicionales, bucles, ... que permite modelizar escenarios complejos.

© JMA 2019. All rights reserved

Características

- Dispone de una selección inteligente de campos usando ID, nombre, Xpath o DOM según se necesite.
- Para la depuración permite la configuración de los puntos de interrupción, iniciar y detener la ejecución de un caso de prueba desde cualquier punto dentro del caso de prueba e inspeccionar la forma en el caso de prueba se comporta en ese punto.
- Permite exportar los casos de prueba a Java, C# y Ruby, actuando como embriones en la creación de los casos de prueba para WebDriver.
- Selenium IDE dispone de un amplio conjunto de extensiones adicionales que ayudan o simplifican la elaboración de los casos de pruebas.

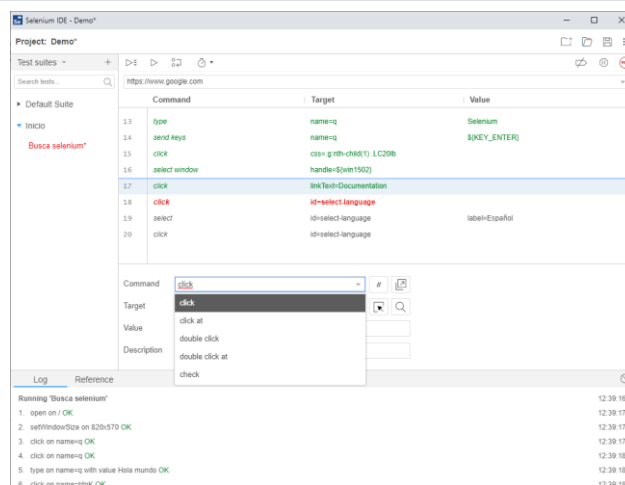
© JMA 2019. All rights reserved

Empezando

- Instale Selenium IDE desde la tienda web Chrome o Firefox .
- Una vez instalado, inícielo haciendo clic en su icono desde la barra de menú de su navegador.
- Aparecerá un cuadro de diálogo de bienvenida con las siguientes opciones:
 - Grabar una nueva prueba en un nuevo proyecto
 - Abrir un proyecto existente
 - Crea un nuevo proyecto
 - Cierra el IDE
- Si se da grabar, pedirá el nombre del proyecto y la URL base de la aplicación que se está probando (se utiliza en todas las pruebas del proyecto).
- Después de completar esta configuración, se abrirá la ventana IDE y una nueva ventana del navegador, cargará la URL base y comenzará a grabar. Interactúe con la página y cada una de las acciones se registrará en el IDE. Para detener la grabación, cambie a la ventana IDE y haga clic en el icono de grabación.

© JMA 2019. All rights reserved

Empezando



© JMA 2019. All rights reserved

Empezando

- La grabación se puede editar, modificar, reproducir, depurar, guardar y exportar.
- Cada grabación es una prueba. El proyecto puede contener múltiples pruebas que se organizan agrupándolas en suites. Por defecto se crea la “Default Suite” a la que se le asigna la prueba de la grabación inicial. Las pruebas y las suites se pueden renombrar, ejecutar o eliminar.
- El proyecto se puede descartar o guardar un único archivo JSON con una extensión .side para recuperarlo posteriormente o ejecutarlo en la línea de comandos.
- La reproducción de la prueba en el IDE permite establecer puntos de ruptura e inspeccionar valores.
- Se puede reanudar la grabación en cualquier punto.

© JMA 2019. All rights reserved

Scripts

- Se pueden desarrollar automáticamente scripts al crear una grabación y de esa manera se puede editar manualmente con sentencias y comandos para que la reproducción de nuestra grabación sea correcta
- Los scripts se generan en un lenguaje de scripting especial para Selenium a menudo denominado Selanese.
- Selanese provee comandos que dicen al Selenium que hacer y pueden ser:
 - **Acciones:** son comandos que generalmente manipulan el estado de la aplicación, ejecutan acciones sobre objetos del navegador, como hacer click en un enlace, escribir en cajas de texto o seleccionar de una lista de opciones. Muchas acciones pueden ser llamadas con el sufijo "AndWait" que indica la acción hará que el navegador realice una llamada al servidor y que se debe esperar a una nueva página se cargue.
 - **Descriptores de acceso:** examinan el estado de la página y almacenan los resultados en variables.
 - **Aserciones:** son como descriptores de acceso, pero las muestras confirman que el estado de la solicitud se ajusta a lo que se esperaba, verifican la presencia de un texto en particular o la existencia de elementos.

© JMA 2019. All rights reserved

Localizadores

- Localizar por Id:
 - id=loginForm
- Localizar por Name
 - name=username
- Localizar por XPath
 - xpath=//form[@id='loginForm']
- Localizar por el texto en los hipervínculos
 - link=Continue
- Localizar por CSS
 - css=input[name="username"]

© JMA 2019. All rights reserved

Acciones

- Sobre elementos de los formularios:
 - click, click at, double click, double click at, check, uncheck, edit content, send keys, type, select, add selection, remove selection, submit
- Del ratón:
 - mouse move at, mouse over, mouse out, mouse down, mouse down at, mouse up, mouse up at, drag and drop to object
- Sobre cuadros de dialogo alert, prompt y confirm:
 - choose ok on next confirmation, choose cancel on next confirmation, choose cancel on next prompt, webdriver choose ok on visible confirmation, webdriver choose cancel on visible confirmation, webdriver choose cancel on visible prompt
- Sobre la ejecución de la prueba:
 - open, execute script, run, echo, debugger, pause, close

© JMA 2019. All rights reserved

Variables

- Se puede usar variable en Selenium para almacenar constantes al principio de un script. Además, cuando se combina con un diseño de prueba controlado por datos, las variables de Selenium se pueden usar para almacenar valores pasados a la prueba desde la línea de comandos, desde otro programa o desde un archivo.
 - `store target:valor value:varName`
- Para acceder al valor de una variable:
 - `${userName}`
- Hay métodos disponibles para recuperar información de la página y almacenarla en variables:
 - `storeAttribute`, `storeText`, `storeValue`, `storeTitle`, `storeXPathCount`

© JMA 2019. All rights reserved

Flujo de control

- Selenium IDE viene con comandos que permiten agregar lógica condicional y bucles a las pruebas, para ejecutar comandos (o un conjunto de comandos) solo cuando se cumplen ciertas condiciones o repetidamente en función de criterios predefinidos.
- Las condiciones en su aplicación se verifican mediante el uso de expresiones de JavaScript.
- Los comandos de flujo de control Control Flow funcionan con comandos de apertura y cierre para denotar un conjunto (o bloque) de comandos.
- Aquí están cada uno de los comandos de flujo de control disponibles acompañados de sus comandos complementarios de cierre.
 - `if ... else if ... else ... end` `// else if y else son opcionales`
 - `times ... end` `// Bucle for`
 - `forEach ... end`
 - `while ... end`
 - `do ... repeat if` `// la condicion en repeat if`
- Se pueden anidar los comandos de control de flujo según sea necesario.

© JMA 2019. All rights reserved

Afirmar y Verificar

- Una "afirmación" hará fallar la prueba y abortará el caso de prueba actual, mientras que una "verificación" hará fallar la prueba pero continuará ejecutando el caso de prueba.
 - Tiene muy poco sentido para comprobar que el primer párrafo de la página sea el correcto si la prueba ya falló al comprobar que el navegador muestra la página esperada. Por otro lado, es posible que desee comprobar muchos atributos de una página sin abortar el caso de prueba al primer fallo, ya que esto permitirá revisar todos los fallos en la página y tomar la acción apropiada.
- Selenese permite múltiples formas de comprobar los elementos de la interfaz de usuario pero hay que decidir el método más apropiado:
 - ¿Un elemento está presente en algún lugar de la página?
 - ¿El texto especificado está en algún lugar de la página?
 - ¿El texto especificado está en una ubicación específica en la página?
- Métodos:
 - `assert`, `assertAlert`, `assertChecked`, `assertNotChecked`, `assertConfirmation`, `assertEditable`, `assertNotEditable`, `assertElementPresent`, `assertElementNotPresent`, `assertPrompt`, `assertSelectedValue`, `assertNotSelectedValue`, `assertSelectedLabel`, `assertText`, `assertNotText`, `assertTitle`, `assertValue`
 - `verify`, `verifyChecked`, `verifyNotChecked`, `verifyEditable`, `verifyNotEditable`, `verifyElementPresent`, `verifyElementNotPresent`, `verifySelectedValue`, `verifyNotSelectedValue`, `verifyText`, `verifyNotText`, `verifyTitle`, `verifyValue`, `verifySelectedLabel`

© JMA 2019. All rights reserved

Ejecutar en línea de comandos

<https://www.selenium.dev/selenium-ide/docs/en/introduction/command-line-runner>

- Requiere tener instalado NodeJS (<https://nodejs.org>)
- Instalar CLI
 - `npm install -g selenium-side-runner`
- Instalar los WebDriver:
 - Crear una carpeta y referenciarla en el PATH del sistema.
 - Descargar los drivers (<https://www.selenium.dev/downloads/>)
 - Copiarlos a la carpeta creada.
- Para ejecutar las suites de pruebas:
 - `selenium-side-runner project.side project2.side *.side`
- Para ejecutar en diferentes navegadores:
 - `selenium-side-runner *.side -c "browserName=Chrome"`
 - `selenium-side-runner *.side -c "browserName=firefox"`

© JMA 2019. All rights reserved

Instalar los WebDriver (npm)

- Chrome
 - npm install -g chromedriver
- Edge
 - npm install -g edgedriver
 - npm install -g @sitespeed.io/edgedriver
- Firefox
 - npm install -g geckodriver
- Internet Explorer
 - npm install -g iedriver

© JMA 2019. All rights reserved

Exportación

- Se puede exportar una prueba o un conjunto de pruebas a código de WebDriver haciendo clic con el botón derecho en una prueba o un conjunto, seleccionando Export, eligiendo el idioma de destino y haciendo clic Export.
- Actualmente, se admite la exportación a los siguientes idiomas y marcos de prueba.
 - C# NUnit
 - C# xUnit
 - Java JUnit
 - JavaScript Mocha
 - Python pytest
 - Ruby RSpec
- Esta previsto admitir todos los lenguaje de programación soportados oficialmente para Selenium en al menos un marco de prueba para cada lenguaje.

© JMA 2019. All rights reserved

WebDriver

```
@BeforeClass
public static void setUpClass() throws Exception {
    System.setProperty("webdriver.chrome.driver", "C:/Archivos/.../chromedriver.exe");
}

@Before
public void setUp() throws Exception {
    driver = new ChromeDriver();
    baseUrl = "http://localhost/";
    driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}

@Test
public void testLoginOK() throws Exception {
    driver.get(baseUrl + "/login.php");
    driver.findElement(By.id("login")).sendKeys("admin");
    driver.findElement(By.id("password")).sendKeys("admin");
    driver.findElement(By.cssSelector("input[type='submit']")).click();
    try {
        assertEquals("", driver.findElement(By.cssSelector("img[title='Main Menu']")).getText());
    } catch (Error e) {
        verificationErrors.append(e.toString());
    }
}
```

Maven

```
<dependency>
<groupId>org.seleniumhq.selenium</groupId>
<artifactId>selenium-java</artifactId>
<version>3.13.0</version>
</dependency>
```

© JMA 2019. All rights reserved

WEBDRIVER

© JMA 2019. All rights reserved

WebDriver

- Selenium permite la automatización de todos los principales navegadores del mercado mediante el uso de WebDriver.
- WebDriver es una API y un protocolo que define una interfaz de idioma neutral para controlar el comportamiento de los navegadores web. Cada navegador está respaldado por un Driver, una implementación específica de WebDriver, llamada controlador. El controlador es el componente responsable de delegar en el navegador, y maneja la comunicación entre Selenium y el navegador.
- Esta separación es parte de un esfuerzo consciente para hacer que los proveedores de navegadores asuman la responsabilidad de la implementación para sus navegadores. Selenium utiliza estos controladores de terceros cuando es posible, pero también proporciona sus propios controladores mantenidos por el proyecto para los casos en que esto no es una realidad.
- El framework de Selenium unifica todas estas piezas a través de una interfaz orientada al usuario que habilita que los diferentes backends de los navegadores sean utilizados de forma transparente, permitiendo la automatización cruzada entre navegadores y plataformas diferentes.

© JMA 2019. All rights reserved

WebDriver

- WebDriver controla un navegador de forma nativa, como lo haría un usuario, ya sea localmente o en una máquina remota utilizando el servidor Selenium, marca un salto adelante en términos de automatización de navegadores.
- Selenium WebDriver se refiere tanto a los enlaces de lenguajes como también a las implementaciones individuales del código controlador del navegador. Esto se conoce comúnmente solo como WebDriver.
- Selenium WebDriver es una Recomendación W3C (<https://www.w3.org/TR/webdriver1/>)
 - WebDriver está diseñado como una interfaz de programación simple y más concisa.
 - WebDriver es una API compacta orientada a objetos.
 - Controla el navegador de manera efectiva.

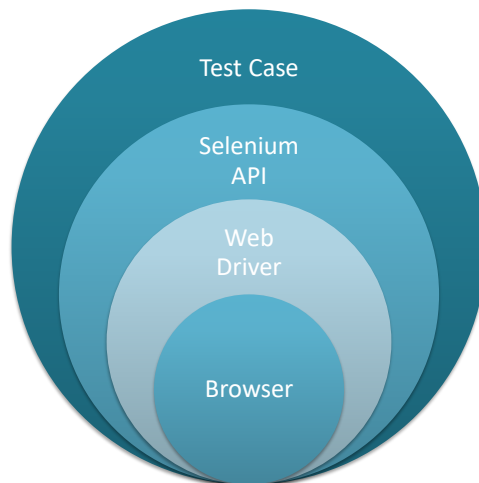
© JMA 2019. All rights reserved

Componentes

- API: Interfaz de Programación de Aplicaciones. Es un conjunto de “comandos” que se utilizan para manipular el WebDriver.
- Library: Un módulo de código que contiene las APIs y el código necesario para implementarlos. Las librerías son específicas para cada lenguaje, por ejemplo ficheros .jar en Java, ficheros .dll para .NET, etc.
- Driver: El responsable de controlar un navegador concreto (proxies). La mayoría de los drivers son creados por los fabricantes del navegador. Los Drivers son generalmente módulos ejecutables que corren en el sistema con el propio navegador, no en el sistema ejecutando la suite de test.
- Remote WebDriver: Permite controlar instancias remotas de navegadores web.
- Framework: Una librería adicional usada como un soporte para la suites de WebDriver. Estos frameworks pueden ser test frameworks como JUnit o NUnit. También pueden ser frameworks soportando lenguaje natural como Cucumber o Robotium. Los frameworks también pueden ser escritos y usados para cosas como la manipulación o configuración del sistema bajo la prueba, creación de datos, test oracles, etc

© JMA 2019. All rights reserved

Componentes



© JMA 2019. All rights reserved

Navegadores

El framework de Selenium soporta oficialmente los siguientes navegadores:

Navegador	Proveedor	Versiones Soportadas
Chrome	Chromium	Todas las Versiones
Firefox	Mozilla	54 y más recientes
Internet Explorer	Selenium	6 y más recientes
Opera	Opera Chromium / Presto	10.5 y más recientes
Safari	Apple	10 y más recientes

También hay un conjunto de navegadores especializados utilizados típicamente en entornos de desarrollo.

Controlador	Propósito	Mantenedor
HtmlUnitDriver	Emulador de navegador headless respaldado por Rhino	Proyecto Selenium

© JMA 2019. All rights reserved

Navegadores simulados

- **HtmlUnit**
 - HtmlUnit es un navegador sin interfaz grafica para programas basados en Java. Modela documentos HTML y proporciona un API que permite invocar las paginas, rellenar formularios, hacer clic en enlaces, etc. Soporta JavaScript y es capaz de funcionar con librerías AJAX, simulando Chrome, Firefox o Internet Explorer dependiendo de la configuración usada.
- **PhantomJS**
 - PhantomJS es un navegador sin interfaz grafica basado en Webkit, aunque es una versión mucho mas antigua que las usadas por Chrome o Safari. El proyecto esta sin soporte desde que se anunció que Chrome y Firefox tendrían la capacidad de ser navegadores sin interfaz grafica.

© JMA 2019. All rights reserved

Instalación

- La gran mayoría de controladores necesitan de un ejecutable extra para que Selenium pueda comunicarse con el navegador. Para ejecutar tu proyecto y controlar el navegador, debes tener instalados los binarios de WebDriver específicos para el navegador.
- Crea un directorio para almacenar los ejecutables en el, como C:\WebDriver\bin o /opt/WebDriver/bin. A través de los diferentes proveedores, descargar y descomprimir en la carpeta recién creada.
- Se puede especificar manualmente donde está ubicado el ejecutable antes lanzar el WebDriver, pero esto hará que los tests sean menos portables, ya que los ejecutables necesitan estar en el mismo lugar en todas las maquinas. Si la ruta de la carpeta está incluida en el PATH no es necesario especificarla en cada test:
 - Windows: setx /m path "%path%;C:\WebDriver\bin\"
 - macOS y Linux: export PATH=\$PATH:/opt/WebDriver/bin >> ~/.profile
- Para probar los cambios, en una terminal de comando escribe el nombre de uno de los binarios que has añadido en la carpeta en el paso previo:
 - chromedriver

© JMA 2019. All rights reserved

Instalación

- Dependencias del proyecto:
 - npm install --save-dev selenium-webdriver mocha assert chromedriver geckodriver @sitespeed.io/edgedriver
- Configurar el comando test en package.json:

```
"scripts": {  
  "test": "npx mocha e2e/**/*.spec.js --timeout 60000"  
},
```
- Ejecutar desde línea de comandos:
 - SET SELENIUM_BROWSER=chrome,firefox && npx mocha --recursive e2e

© JMA 2019. All rights reserved

Programación Asíncrona

- Todas las operaciones asíncronas de la biblioteca WebDriverJS devuelven promesas para aliviar el dolor de trabajar con una API puramente asíncrona.
- Con la aparición de `async/await` en el estándar de JavaScript se puede escribir código como si WebDriverJS tuviera una API de bloqueo síncrona.

```
let driver;
beforeEach(async function() {
  driver = await new Builder().forBrowser('firefox').build();
});
afterEach(async function() {
  await driver.quit();
});
it('example', async function() {
  await driver.get('http://example.com');
  assert.equal(await driver.getTitle(), 'Example Domain')
});
```

© JMA 2019. All rights reserved

Cargar el driver

- Importar el espacio de nombres de WebDriver.
`const {Builder, Browser} = require('selenium-webdriver');`
- Para crear el driver que controlara al navegador:
`let driver = await new Builder().forBrowser(navegador).build();`
- Los posibles valores tipados para navegador son: 'chrome', 'MicrosoftEdge', 'firefox', 'internet explorer', 'safari'.
- Para evitar errores sintácticos se deben usar las propiedades de Browser.
- Se pueden establecer las opciones específicas de cada navegador y el servicio que lo controla, pero al ser específicas requieren objetos y métodos de configuración particulares para cada uno.

```
const edgeDriver = require('@sitespeed.io/edgedriver');
const edge = require('selenium-webdriver/edge');
let options = new edge.Options();
let service = new edge.ServiceBuilder(edgeDriver.binPath());
driver = await new Builder().setEdgeOptions(options).forBrowser(Browser.EDGE)
  .setEdgeService(service).build();
```

© JMA 2019. All rights reserved

suite

- Define un conjunto de pruebas llamando a la función proporcionada una vez para cada uno de los navegadores de destino. Si una suite no está limitada a un conjunto específico de navegadores en las opciones, la suite se configurará para ejecutarse en cada uno de los navegadores del PATH con ejecutables de WebDriver (chromedriver, firefoxdriver, etc.) instalados en el sistema.

```
const { suite } = require('selenium-webdriver/testing');

suite(function(env) {
  describe('Prueba en ${env.browser.name}', function() {
    let driver;
    before(async function() { driver = await env.builder().build(); });
    after(() => driver.quit());

    it('Pagina', async function() { await driver.get('http://example.com'); });
  });
});
```

© JMA 2019. All rights reserved

suite

- Se pueden restringir los navegadores de la suite con la variable de entorno `SELENIUM_BROWSER`.
 - `SELENIUM_BROWSER=chrome,firefox`
- Se pueden excluir determinadas pruebas (xdescribe y xit) para determinados navegadores:

```
const {suite, ignore} = require('selenium-webdriver/testing');
suite(function(env) {
  // Skip tests
  ignore(env.browsers(Browser.CHROME)).
  describe('No en Chrome', async function() {
    ignore(env.browsers(Browser.FIREFOX)).
    it('Esta tampoco en Firefox', async function() {
```

© JMA 2019. All rights reserved

Estrategia de carga de página

- Permite los siguientes valores:
 - NORMAL (por defecto) : Esto hará que WebDriver espere a que se cargue toda la página. Espera a que se descarguen todos los recursos, no las peticiones AJAX, y se dispare el evento load y concluya.
 - EAGER (ansioso): Esto hará que WebDriver espere hasta que el documento HTML inicial se haya cargado y analizado por completo, el acceso DOM está listo, pero es posible que aún se estén cargando otros recursos, como imágenes. WebDriver espera hasta que se dispare el evento DOMContentLoaded y concluya.
 - NONE: Esto hará que WebDriver solo espera hasta que se descargue la página inicial.

```
const { Capabilities } = require('selenium-webdriver');  
let caps = new Capabilities();  
caps.setPageLoadStrategy("eager");  
let driver = await env.builder().withCapabilities(caps).build();
```

© JMA 2019. All rights reserved

Aislar las pruebas

- Hay muchos recursos externos que se cargan en una página que no son directamente relevantes para la funcionalidad que está probando (por ejemplo, widgets de Facebook, Analytics, fragmentos de JavaScript, CDN, etc.). Estos recursos externos tienen el potencial de impactar negativamente las ejecuciones de prueba debido a tiempos de carga lentos.
- Entonces, ¿cómo proteger nuestras pruebas de estas cosas que están fuera de su control?
- Una posible solución es utilizar un servidor proxy en nuestras pruebas, que actúe como un doble de pruebas, donde podremos bloquear los recursos externos que no queremos cargar agregándolos a una lista negra o suministrando una versión ligera.

© JMA 2019. All rights reserved

Proxy

- Un servidor proxy actúa como intermediario para solicitudes entre un cliente y un servidor. Un servidor proxy para scripts de automatización con Selenium podría ser útil para:
 - Capturar el tráfico de la red
 - Simular llamadas de backend realizadas por el sitio web
 - Acceder al sitio web requerido bajo topologías de red complejas o restricciones/políticas corporativas estrictas.
- Si te encuentras en un entorno corporativo, y un navegador no puede conectarse a una URL, esto es muy probablemente porque el ambiente necesita un proxy para acceder.
- Selenium WebDriver proporciona una vía para configurar el proxy:

```
const chrome = require('selenium-webdriver/chrome');
const proxy = require('selenium-webdriver/proxy')
let options = new chrome.Options();
options.setProxy(proxy.manual({http: '<HOST:PORT>'}));
let driver = await new Builder().setChromeOptions(options)
    .forBrowser(Browser.CHROME).build();
```

© JMA 2019. All rights reserved

Perfiles Firefox

- Firefox guarda tu información personal (marcadores, contraseñas y preferencias de usuario) en un conjunto de archivos llamado perfil, que se almacena en una ubicación de disco diferente a la que se utiliza para almacenar los archivos de ejecución de Firefox. Puedes crear distintos perfiles, cada uno de ellos con un conjunto de datos de usuario diferente. Solo puede estar activo un perfil en un momento determinado. Mediante el Administrador de perfiles de Firefox puedes crear, eliminar y renombrar los perfiles. Cuando se desee ejecutar una automatización confiable en un navegador Firefox, se recomienda crear un perfil separado.
- Para iniciar el Administrador de perfiles se introduce en la barra de `about:profiles`. Para abrir Firefox con un determinado perfil: `firefox.exe -P webdrive`. Para establecer el perfil en las opciones:

```
const firefox = require('selenium-webdriver/firefox');
const options = new firefox.Options();
options.setProfile("testProfile");
const driver = new Builder().forBrowser('firefox')
    .setFirefoxOptions(options).build();
```

© JMA 2019. All rights reserved

Visión de conjunto

- Una vez referenciado el controlador ya se puede interactuar con el navegador, todas las acciones se realizarán a través de dicha referencia.
- Con el controlador se navega a una nueva página.
- En la página, se buscarán los diferentes elementos (tag) con el método `findElement` y un localizador, que devolverá un `WebElement`. Este método recupera un solo elemento, si se necesita recuperar varios elementos, el método `findElements` obtiene la colección de elementos localizados.
- Un localizador es un objeto que define el selector de los elementos web basándose en diferentes estrategias como ID, Nombre, Clase, XPath, Selectores CSS, Texto de enlace, etc. Los `WebElements` se pueden encontrar buscando desde la raíz del documento o buscando en otra `WebElement`.
- Un `WebElement` representa un elemento del DOM. El `WebElement` tiene un conjunto de métodos de acción, tales como `click()`, `getText()` y `sendKeys()`.
- Esta es la forma principal para interactuar con un elemento (etiqueta) de la página y obtener información de respuesta de él.

© JMA 2019. All rights reserved

Navegación

- Navegar hacia
`await driver.get("https://selenium.dev"); // Recomendada`
`await driver.navigate().to("https://selenium.dev");`
- Retroceder
`await driver.navigate().back();`
- Avanzar
`await driver.navigate().forward();`
- Actualizar
`await driver.navigate().refresh();`
- Obtener la URL actual
`url = await driver.getCurrentUrl();`
- Obtener el título
`await driver.getTitle();`

© JMA 2019. All rights reserved

Navegación

- Salir y cerrar el navegador al final de una sesión
`await driver.quit();`
 - Muy importante porque:
 - Cerrará todas las ventanas y pestañas asociadas a esa sesión del WebDriver.
 - Cerrará el proceso de navegador.
 - Cerrará el proceso en segundo plano del driver.
 - Notificará al Grid de Selenium que el navegador ya no está en uso y que puede ser usado por otra sesión del Grid de Selenium.
 - Un fallo en la llamada del método salir dejará procesos corriendo en segundo plano y puertos abiertos en tu máquina lo que podría llevar a problemas en un futuro.

© JMA 2019. All rights reserved

Estrategia de carga de página

- Por defecto, cuando WebDriver carga una página, sigue la estrategia de carga NORMAL. Siempre se recomienda detener la descarga de más recursos adicionales (como imágenes, css, js) cuando la carga de la página lleva mucho tiempo.
- La propiedad `document.readyState` de un documento describe el estado de carga del documento actual. Por defecto, WebDriver esperará responder a una llamada `driver.get()` o `driver.navigate().to()` hasta que el estado de documento listo esté completo
- En aplicaciones SPA (como Angular, React, Ember) una vez que el contenido dinámico ya está cargado (es decir, una vez que el estado de `readyState` es COMPLETE), hacer clic en un enlace o realizar alguna acción dentro de la página no disparará una nueva solicitud al servidor ya que el contenido se carga dinámicamente en el lado del cliente sin una actualización de la página. Las aplicaciones de SPA pueden cargar muchas vistas dinámicamente sin ninguna solicitud al servidor, por lo que `readyState` siempre mostrará el estado 'COMPLETE' hasta que se haga un nuevo `driver.get()` y `driver.navigate().to()`.

© JMA 2019. All rights reserved

Cookies

- Una cookie es una pequeña pieza de datos que es enviada desde el sitio web y es almacenada en el ordenador. Las cookies son usadas principalmente para reconocer al usuario y cargar la información personalizada.
- Se gestionan a través de `driver.manage()`, una vez descargada la página:

```
await driver.manage().addCookie({ name: 'key', value: 'value' });
let cookie = await driver.manage().getCookie('key');
let cookies = await driver.manage().getCookies();
await driver.manage().deleteCookie('key');
await driver.manage().deleteAllCookies();
```
- Las propiedades de la cookie son: `name`, `value`, `path`, `domain`, `httpOnly`, `secure`, `sameSite`, `expiry`
- Por defecto, el atributo `sameSite` está fijado como `'Strict'`, la cookie no será enviada junto a las peticiones iniciadas por páginas web externas. Cuando se fija como `'Lax'`, será enviada junto con la petición GET iniciada por páginas web externas.

```
await driver.manage().addCookie({ name: 'key', value: 'value', sameSite: 'Strict' });
```

© JMA 2019. All rights reserved

Ventanas, solapas, Frames e Iframes

- El uso de múltiples ventanas, o solapas, Frames e Iframes está estrictamente prohibido por las reglas de WAI, dado que construyen sitios desde múltiples documentos en el mismo dominio.
- Para trabajar con ventanas o solapas:
 - `const windowHandle = await driver.getWindowHandle();`
 - `await driver.switchTo().window(windowHandle);`
 - `await driver.close();` // Close the tab or window
- Para trabajar con Frames e Iframes
 - `await driver.switchTo().frame("myframe");`
 - `await driver.switchTo().defaultContent();`

© JMA 2019. All rights reserved

Tamaño del navegador

- La resolución de las pantallas puede impactar en como la aplicación se renderiza, sobre todo en contextos de Responsive Design.
- El W3C impone restricciones sobre los elementos presentes pero no visibles.
- WebDriver provee de mecanismos para mover y cambiar el tamaño de la ventana del navegador:

```
await driver.manage().window().setSize({ width: 1024, height: 768 });
await driver.manage().window().maximize();
await driver.manage().window().minimize();
await driver.manage().window().fullscreen();
```

© JMA 2019. All rights reserved

Buscar elementos

- Los WebElements se pueden encontrar buscando desde la raíz del documento utilizando una instancia de WebDriver o buscando en otra WebElement. Las búsquedas son costosas por lo que conviene guardar el resultado para no repetirlas.
- Localizar un elemento o el primer elemento:

```
let loginForm = await driver.findElement(By.id('loginForm'));
let userName = await loginForm.findElement(By.name('Usuario'));
let usr = await driver.findElement(
  By.css('#loginForm [name=Usuario]');
```
- Localizar múltiples elementos:

```
let elements = await driver.findElements(By.tagName("li"));
```
- Localizar el elemento activo (con el foco).

```
let active = await driver.switchTo().activeElement();
```

© JMA 2019. All rights reserved

Localización de elementos

- La interfaz By encapsula las diferentes estrategias de localización de elementos. Hay ocho estrategias diferentes de ubicación de elementos integradas en WebDriver.
- By.id: Localiza elementos cuyo atributo ID coincide con el valor de la búsqueda
`let tag = await driver.findElement(By.id("myForm"));`
- By.name: Localiza elementos cuyo atributo NAME coincide con el valor de la búsqueda
`let tag = await driver.findElement(By.name("username"));`
- By.className: Localiza elementos en el que el atributo class contiene el valor de la búsqueda (no se permiten nombres de clase compuestos)
`let tag = await driver.findElement(By.className("container-fluid"));`

© JMA 2019. All rights reserved

Localización de elementos

- By.cssSelector: Localiza elementos que coinciden con un selector CSS
`let tag = await driver.findElement(By.cssSelector(".header img"));`
- By.linkText: Localiza hipervínculos cuyo texto visible coincide con el valor de búsqueda
`let tag = await driver.findElement(By.linkText("Close"));`
- By.partialLinkText: Localiza hipervínculos cuyo texto visible contiene el valor de búsqueda
`let tag = await driver.findElement(By.partialLinkText("Next"));`
- By.tagName: Localiza elementos cuyo nombre de etiqueta (tagName) coincide con el valor de búsqueda
`let elements = await driver.findElements(By.tagName("img"));`
- By.xpath : Localiza elementos utilizando una expresión Xpath
`let tag = await driver.findElement(By.xpath("//div[2]/input"));`

© JMA 2019. All rights reserved

Localización de elementos

- En general, si los ID o Name del HTML están disponibles, son únicos y predecibles, son el método preferido para ubicar un elemento en una página. Tienden a trabajar muy rápido y evitan costosos recorridos DOM.
- Si las ID únicas no están disponibles, un selector CSS bien escrito es el método preferido para localizar un elemento. XPath funciona tan bien como los selectores CSS, pero la sintaxis es complicada y con frecuencia difícil de depurar. Aunque los selectores XPath son muy flexibles, generalmente su desempeño no es probado por los proveedores de navegadores y tienden a ser bastante lentos.
- Las estrategias de selección basadas en enlaces de texto y enlaces de texto parciales tienen el inconveniente en que solo funcionan en elementos de enlace. Además, internamente llaman a los selectores XPath.
- El nombre de la etiqueta (obsoleto) puede ser una forma peligrosa de localizar elementos. Existen frecuentemente múltiples elementos con la misma etiqueta presentes en la página. Es útil sobre todo cuando se llama al método `findElements(By)` que devuelve una colección de elementos.
- La recomendación es mantener los localizadores tan compactos y legibles como sea posible. Pedirle a WebDriver que atraviese la estructura del DOM es una operación costosa y, cuanto más se pueda reducir el alcance de tu búsqueda, mejor.

© JMA 2019. All rights reserved

Localizadores relativos

- Los `RelativeLocator` son una alternativa al `By` y son útiles cuando no es fácil encontrar un localizador para el elemento deseado, pero es fácil describir su posición relativa a otro elemento fácilmente localizable.
- Que está encima, debajo, a la izquierda o a la derecha de:

```
let emailLocator = locateWith(By.tagName('input')).above(By.id('password'));  
let passwordLocator = locateWith(By.tagName('input')).below(By.id('email'));  
let cancelLocator = locateWith(By.tagName('button')).toLeftOf(By.id('submit'));  
let submitLocator = locateWith(By.tagName('button')).toRightOf(By.id('cancel'));
```
- Si el posicionamiento relativo no es obvio o varía según el tamaño de la ventana, se puede usar el método `cercano` para identificar un elemento que esté alejado como máximo 50px del localizador proporcionado.

```
let emailLocator = locateWith(By.tagName('input')).near(By.id('lbl-email'));
```
- También puede encadenar localizadores si se encuentra arriba/abajo de un elemento y a la derecha/izquierda de otro.

```
let submitLocator = locateWith(By.tagName('button')).below(By.id('email'))  
    .toRightOf(By.id('cancel'));
```

© JMA 2019. All rights reserved

Operar con elementos

- Recuperar el estado del elemento

- `.getText()`: Obtiene el texto visible (es decir, no oculto por CSS) del elemento, incluidos los subelementos.
- `.isDisplayed()`: Determina si el elemento es visible. Este método evita el problema de tener que analizar el atributo "estilo" de un elemento.
- `.isEnabled()`: Determina si el elemento está habilitado actualmente.
- `.isSelected()`: Determina si este elemento está seleccionado, tiene el foco.
- `.getAttribute()`: Obtiene el valor del atributo dado del elemento.
- `.getCssValue()`: Obtiene el valor de una propiedad CSS dada.
- `.getRect()`: Obtiene un objeto con su ubicación (x, y) en píxeles, en relación con el documento, y el tamaño del elemento en píxeles.
- `.getSize()`: Obtiene el ancho y el alto del elemento renderizado
- `.getTagName()`: Obtiene el nombre de la etiqueta de este elemento.

```
assert.equal(await tag.getText(), 'Example Domain');
```

© JMA 2019. All rights reserved

Operar con elementos

- Buscar subelementos:

- `.findElement()`: Encuentra el primer `WebElement` usando el localizador dado.
- `.findElements()`: Encuentra todos los elementos dentro del elemento usando el localizador dado.

- Acciones

- `.click()`: Haz clic en este elemento.
- `.sendKeys()`: Simula escribir en un elemento.
- `.clear()`: Si es un elemento de entrada de texto, borra el valor.
- `.submit()`: Si el elemento es un formulario o un elemento dentro de un formulario, se enviará el formulario al servidor remoto.

```
await driver.findElement(By.name("password")).clear();
await driver.findElement(By.name("password")).sendKeys("god");
await driver.findElement(By.id("myForm")).submit();
```

© JMA 2019. All rights reserved

Esperas

- Generalmente se puede decir que WebDriver posee una API de bloqueo. Porque es una biblioteca fuera-de-proceso que le dice al navegador qué hacer, y debido a que la plataforma web tiene una naturaleza intrínsecamente asíncrona, WebDriver no rastrea el estado activo y en tiempo real del DOM.
- Afortunadamente, el conjunto normal de acciones disponibles en la interfaz WebElement tales como click o sendKeys esta garantizadas para ser síncronas, es decir, las llamadas a métodos no vuelven hasta que el comando se haya completado en el navegador. Las API avanzadas de interacción del usuario, Keyboard y Mouse, son excepciones ya que están explícitamente pensadas como comandos asíncronos “Haz lo que te digo”.
- Esperar es hacer que transcurra una cierta cantidad de tiempo antes de continuar con el siguiente paso en la ejecución automatizada de la tarea.

© JMA 2019. All rights reserved

Esperas explícitas

- Permiten que el código detenga la ejecución del programa, o congelar el hilo, hasta que la condición pasada sea resuelta. La condición se llama con cierta frecuencia, hasta que transcurra el tiempo de espera. Esto significa que mientras la condición devuelva un valor falso, seguirá intentando y esperando.
- Dado que las esperas explícitas permiten esperar a que ocurra una condición, hacen una buena combinación para sincronizar el estado entre el navegador, y su DOM, con el código de WebDriver.
- Para esperar a que la llamada findElement espere hasta que el elemento agregado dinámicamente desde el script se haya agregado al DOM:

```
let element = await driver.wait(until.elementLocated(By.css('p')),10000);
assert.strictEqual(await element.getText(), 'Hello word!');
await driver.findElement(By.id("myForm")).submit();
let rslt = await driver.wait(until.elementIsVisible(By.id('result')),10000);
```

© JMA 2019. All rights reserved

Condiciones esperadas

- Es una necesidad bastante común tener que sincronizar el DOM con el código de prueba, hay condiciones predefinidas para operaciones frecuentes de espera.
- Las condiciones disponibles varían en las diferentes librerías de los lenguajes, pero esta es una lista no exhaustiva de algunos:
 - alert is present (la alerta esta presente)
 - element exists (el elemento existe)
 - element is visible (el elemento es visible)
 - title contains (el titulo contiene)
 - title is (el titulo es)
 - visible text (el texto es visible)
 - text to be (el texto es)

© JMA 2019. All rights reserved

Espera predefinida

- Una instancia de FluentWait define la cantidad máxima de tiempo para esperar por una condición, así como la frecuencia con la que verificar dicha condición.
- Se puede configurar la espera para ignorar tipos específicos de excepciones mientras esperan, como NoSuchElementException cuando buscan un elemento en la página.
- La instancia de FluentWait se puede reutilizar en tantas esperas, con la misma latencia, como sea necesario.

```
let foo = await driver.wait(until.elementLocated(By.id('foo')),  
    30000, 'Timed out after 30 seconds', 5000);
```

© JMA 2019. All rights reserved

Espera implícita

- Una espera implícita es decirle a WebDriver que sondee el DOM durante un cierto período de tiempo al intentar encontrar un elemento o elementos si no están disponibles de inmediato. Esto puede ser útil cuando ciertos elementos en la página web no están disponibles de inmediato y necesitan algo de tiempo para cargarse. La espera implícita está deshabilitada de forma predeterminada y deberá habilitarse manualmente por sesión.
`await driver.manage().setTimeouts({ implicit: 10000 });`
- Si se habilita la espera implícita se pierde la posibilidad de fijar esperas explícitas. No se debe mezclar esperas implícitas y explícitas, causará consecuencias no deseadas, es decir, esperara el máximo de tiempo incluso si el elemento está disponible o la condición es verdadera.

© JMA 2019. All rights reserved

Mensajes emergentes

- WebDriver proporciona una API para trabajar con los tres tipos nativos de mensajes emergentes ofrecidos por JavaScript: alertas, prompts y confirmaciones. Estas ventanas emergentes están diseñadas por el navegador y ofrecen personalización limitada.
- **Alertas:** muestra un mensaje personalizado y un solo botón que cierra la alerta, etiquetado en la mayoría de los navegadores como OK.
`await driver.findElement(By.linkText('See alert')).click();
await wait.until(until.alertIsPresent());
let alert = await driver.switchTo().alert();
assert.strictEqual(await alert.getText(), msgEsperado);
await alert.accept(); // Cerrar`

© JMA 2019. All rights reserved

Mensajes emergentes

- **Confirm:** muestra un mensaje de confirmación y dos botones: uno para confirmar y otro para descartar.

```
let alert = await driver.switchTo().alert();
assert.strictEqual(await alert.getText(), msgEsperado);
await alert.accept(); // confirmar
await alert.dismiss(); // descartar
```

- **Prompt:** solicita un texto al usuario, muestra un mensaje de petición, una caja de texto y dos botones: uno para aceptar y otro para cancelar.

```
let alert = await driver.switchTo().alert();
assert.strictEqual(await alert.getText(), msgEsperado);
await alert.sendKeys("Selenium");
await alert.accept(); // confirmar
await alert.dismiss(); // descartar
```

© JMA 2019. All rights reserved

Interacciones de usuario avanzadas

- WebDriver suministra los métodos de alto nivel `sendKeys` y `click` para simular las entradas habituales de teclado y ratón, pero dispone de la posibilidad de crear acciones compuestas con interacciones de menor nivel.
- La API de interacciones de usuario avanzadas es una API más completa para describir acciones que un usuario puede realizar en una página web. Esto incluye acciones como arrastrar y soltar o hacer clic en varios elementos mientras mantiene presionada la tecla `Ctrl`.
- La clase `Actions` implementa los patrones `Builder` y `Composite` para crear una acción compuesta que contiene todas las acciones especificadas por las llamadas al método. Para generar una secuencia de acciones, se usa el generador de acciones para construirla:

```
const clickable = await driver.findElement(By.id('clickable'))
await driver.actions()
    .move({ origin: clickable }).pause(1000)
    .press().pause(1000)
    .sendKeys('abc').perform()
```

© JMA 2019. All rights reserved

Interacciones de usuario avanzadas

- El objetos Action solo permite la ejecución de las secuencia:
`action.perform();`
- Si la secuencia es de un solo uso, se puede construir y ejecutar en un solo paso:
`await driver.actions().keyDown(Keys.CONTROL)
 .click(someElement).click(someOtherElement)
 .keyUp(Keys.CONTROL)
 .perform();`
- El controlador recuerda el estado de todos los elementos, para liberar todas las teclas y botones del ratón actualmente presionados.
`await driver.actions().clear()`
- La secuencia de acciones debe ser corta: es mejor realizar una secuencia corta de acciones y verificar que la página esté en el estado correcto antes de que tenga lugar el resto de la secuencia.

© JMA 2019. All rights reserved

Teclado

- Hasta ahora, la interacción del teclado se realizaba a través de un elemento específico y WebDriver se aseguraba de que el elemento estuviera en el estado adecuado para esta interacción. Esto consistía principalmente en desplazar el elemento a la ventana gráfica y centrarse en el elemento.
- Dado que el API de interacciones adopta un enfoque orientado al usuario, es más lógico interactuar explícitamente con el elemento antes de enviarle texto, como lo haría un usuario. Esto significa hacer clic en un elemento o enviar un mensaje `Keys.TAB` cuando se enfoca en un elemento adyacente.
- En el API de interacciones se define la secuencia de acciones de teclado sin proporcionar un elemento, se aplican al elemento actual que tiene el foco. Posteriormente se puede cambiar el foco a un elemento antes de enviarle eventos de teclado.
`await driver.actions().keyDown(Keys.SHIFT)
 .sendKeys(search, "Hola ").keyUp(Keys.SHIFT)
 .sendKeys("Mundo").sendKeys(Keys.TAB)
 .perform();`

© JMA 2019. All rights reserved

Ratón

- Las acciones del mouse tienen un contexto: la ubicación actual del mouse. Entonces, al establecer un contexto para varias acciones del mouse (usando `onElement`), la primera acción será relativa a la ubicación del elemento utilizado como contexto, la siguiente acción será relativa a la ubicación del mouse al final de la última acción, etc.
- Mientras que el método de alto nivel `sendKeys` cubre la mayoría de los escenarios de teclado y no suele requerir acciones de bajo nivel, el método `click` cubre el escenario mas común para el ratos pero deja fuera operaciones habituales como el doble click, los desplazamientos, el menú contextual, arrastrar y soltar, ... Para todas estas acciones es necesario el API de interacciones.

```
await driver.actions().move({origin:sourceEle}).press().move({x:8, y:0}).release().contextClick(sourceEle).perform();
await actions.dragAndDrop(draggable, {x: finish.x - start.x, y: finish.y - start.y}).perform();
```

© JMA 2019. All rights reserved

Excepciones

- `NoSuchElementException`: Se produce cuando se intenta interactuar con un elemento que no satisface la estrategia de selector.

```
await driver.findElement(By.linkText("Este no existe")).click();
```
- `StaleElementReferenceError`: Se produce cuando un elemento que se identificó y almaceno previamente ha sido modificado en el DOM e se intentó interactuar con él después de la mutación.

```
let element = await driver.wait(until.elementLocated(By.css('p')),10000);
await driver.findElement(By.linkText("Next")).click();
await element.sendKeys("oHola");
```
- `ElementNotInteractableError`: Lanzada para indicar que aunque un elemento está presente en el DOM, no está en un estado con el que se pueda interactuar: fuera de la vista (scroll, solapado, oculto ...).

```
await driver.actions().keyDown(Keys.CONTROL).sendKeys(Keys.END)
    .keyUp(Keys.CONTROL).perform();
```

© JMA 2019. All rights reserved

Captura de pantalla

- En algunos casos es interesante realizar una captura de pantalla del contexto de navegación actual para su inspección y documentación. La captura de pantalla del endpoint WebDriver devuelve una captura de pantalla que está codificada en formato Base64.

```
const {Builder, Browser} = require('selenium-webdriver');
const fs = require('fs');
let driver = await new Builder().forBrowser('chrome').build();
await driver.get('http://example.com');
let encodedString = await driver.takeScreenshot();
await fs.writeFile('./image.png', encodedString, 'base64');
```

© JMA 2019. All rights reserved

Captura de elemento

- En algunos casos lo interesante es realizar una captura de pantalla de solo un elemento de la página para su inspección y documentación.

```
const {Builder, Browser} = require('selenium-webdriver');
const fs = require('fs');
let driver = await new Builder().forBrowser('chrome').build();
await driver.get('http://example.com');
let ele = await driver.findElement(By.css("h1"));
let encodedString = await ele.takeScreenshot(true);
await fs.writeFile('./element.png', encodedString, 'base64');
```

© JMA 2019. All rights reserved

Imprimir página

- Para imprimir la página actual en un PDF (los navegadores Chromium deben estar en modo headless):

```
const {Builder, Browser} = require('selenium-webdriver');
const fs = require('fs');
const chrome = require('selenium-webdriver/chrome');
let options = new chrome.Options();
let driver = await new Builder().forBrowser('chrome')
    .setChromeOptions(options.headless()).build();
await driver.get('https://www.selenium.dev');
let base64 = await driver.printPage();
await fs.writeFile('./test.pdf', base64, 'base64');
```

© JMA 2019. All rights reserved

BUENAS PRACTICAS

© JMA 2019. All rights reserved

Características de una buena prueba unitaria

- Rápida. No es infrecuente que los proyectos maduros tengan miles de pruebas unitarias. Las pruebas unitarias deberían tardar muy poco tiempo en ejecutarse. Milisegundos.
- Aislada. Las pruebas unitarias son independientes, se pueden ejecutar de forma aislada y no tienen ninguna dependencia en ningún factor externo, como sistemas de archivos o bases de datos.
- Reiterativa. La ejecución de una prueba unitaria debe ser coherente con sus resultados, es decir, devolver siempre el mismo resultado si no cambia nada entre ejecuciones.
- Autocomprobada. La prueba debe ser capaz de detectar automáticamente si el resultado ha sido correcto o incorrecto sin necesidad de intervención humana.
- Oportuna. Una prueba unitaria no debe tardar un tiempo desproporcionado en escribirse en comparación con el código que se va a probar. Si observa que la prueba del código tarda mucho en comparación con su escritura, considere un diseño más fácil de probar.

© JMA 2019. All rights reserved

Asignar nombre a las pruebas

- El nombre de la prueba debe constar de tres partes:
 - Nombre del método que se va a probar.
 - Escenario en el que se está probando.
 - Comportamiento esperado al invocar al escenario.
- Los estándares de nomenclatura son importantes porque expresan de forma explícita la intención de la prueba.

© JMA 2019. All rights reserved

Organizar el código de la prueba

- Prepara, actuar, afirmar es un patrón común al realizar pruebas unitarias. Como el propio nombre implica, consta de tres acciones principales:
 - Prepara los objetos, crearlos y configurarlos según sea necesario.
 - Actuar en un objeto.
 - Afirmar que algo es como se espera.
- Separa claramente en secciones lo que se está probando de los pasos preparación y verificación.
- Las secciones solo deben una vez como máximo y en el orden establecido.
- Minimiza la posibilidad de mezclar aserciones con el código para "actuar".

© JMA 2019. All rights reserved

Preparación mínima

- La sección de preparación, con la entrada del caso de prueba, debe ser lo más sencilla posible, lo imprescindible para comprobar el comportamiento que se está probando.
- Las pruebas se hacen más resistentes a los cambios futuros en el código base y más cercano al comportamiento de prueba que a la implementación.
- Las pruebas que incluyen más información de la necesaria tienen una mayor posibilidad de incorporar errores en la prueba y pueden hacer confusa su intención. Al escribir pruebas, el usuario quiere centrarse en el comportamiento. El establecimiento de propiedades adicionales en los modelos o el empleo de valores distintos de cero cuando no es necesario solo resta de lo que se quiere probar.

© JMA 2019. All rights reserved

Actuación mínima

- Al escribir las pruebas hay que evitar introducir condiciones lógicas como if, switch, while, for, etc.
- Minimiza la posibilidad de incorporar un error a las pruebas.
- El foco está en el resultado final, en lugar de en los detalles de implementación.
- Al incorporar lógica al conjunto de pruebas, aumenta considerablemente la posibilidad de agregar un error. Cuando se produce un error en una prueba, se quiere saber realmente que algo va mal con el código probado y no en el código que prueba. En caso contrario, restan confianza y las pruebas en las que no se confía no aportan ningún valor.
- El objetivo de la prueba debe ser único, si la lógica en la prueba parece inevitable, denota que el objetivo es múltiple y hay que considerar la posibilidad de dividirla en dos o más pruebas diferentes.

© JMA 2019. All rights reserved

Sustituir literales por constantes

- La asignación de literales a constantes permite dar nombre a los valores, aportando semántica.
- Evita la necesidad de que el lector de la prueba inspeccione el código de producción con el fin de averiguar lo que significa un valor, que hace el valor sea especial.
`Assert.IsTrue(rslt.Length <= 10)`
- Muestra explícitamente lo que se intenta probar, en lugar de lo que se intenta lograr.
`const string VARCHAR_LEN = 10;`
`Assert.IsTrue(rslt.Length <= VARCHAR_LEN)`
- Los valores literales pueden provocar confusión al lector de las pruebas. Si una cadena tiene un aspecto fuera de lo normal, puede preguntarse por qué se ha elegido un determinado valor para un parámetro o valor devuelto. Esto obliga a un vistazo más detallado a los detalles de implementación, en lugar de centrarse en la prueba.

© JMA 2019. All rights reserved

Evitar varias aserciones

- Al escribir las pruebas, hay que intentar incluir solo una aserción por prueba. Los enfoques comunes para usar solo una aserción incluyen:
 - Crear una prueba independiente para cada aserción.
 - Usar pruebas con parámetros.
- Si se produce un error en una aserción, no se evalúan las aserciones posteriores.
- Garantiza que no se estén declarando varios casos en las pruebas.
- Proporciona la imagen exacta de por qué se producen errores en las pruebas.
- Al incorporar varias aserciones en un caso de prueba, no se garantiza que se ejecuten todas. Es un todas o ninguna, se sabe por cual fallo pero no si el resto también falla o es correcto, proporcionando la imagen parcial.
- Una excepción común a esta regla es cuando la validación cubre varios aspectos. En este caso, suele ser aceptable que haya varias aserciones para asegurarse de que el resultado está en el estado que se espera que esté.

© JMA 2019. All rights reserved

Refactorizar código

- La refactorización del código de prueba favorece la reutilización y la legibilidad, simplifican las pruebas.
- Salvo que todos los métodos de prueba usen los mismos requisitos, si se necesita un objeto o un estado similar para las pruebas, es preferible usar métodos auxiliares a los métodos de instalación y desmontaje (si existen):
 - Menos confusión al leer las pruebas, puesto que todo el código es visible desde dentro de cada prueba.
 - Menor posibilidad de configurar mas o menos de lo necesario para la prueba.
 - Menor posibilidad de compartir el estado entre las pruebas, lo que crea dependencias no deseadas entre ellas.
- Cada prueba normalmente tendrá requisitos diferentes para funcionar y ejecutarse. Los métodos de instalación y desmontaje son únicos, pero se pueden crear tantos métodos auxiliares como escenarios reutilizables se necesiten.

© JMA 2019. All rights reserved

No validar métodos privados

- En la mayoría de los casos, no debería haber necesidad de probar un método privado.
- Los métodos privados son un detalle de implementación.
- Se puede considerar de esta forma: los métodos privados nunca existen de forma aislada. En algún momento, va a haber un método público que llame al método privado como parte de su implementación. Lo que debería importar es el resultado final del método público que llama al privado.

© JMA 2019. All rights reserved

Aislar las pruebas

- Las dependencias externas afectan a la complejidad de la estrategia de pruebas, hay que aislar a las pruebas de las dependencias externas, sustituyendo las dependencias por dobles de prueba, salvo que se este probando específicamente dichas dependencias.
- Siguiendo la misma regla de oro, las pruebas de integración y sistema deben estar aisladas de sus dependencias salvo cuando se estén probando dichas dependencias. Así mismo, el resultado de las pruebas debe ser previsible.
- Entre las ventajas de esta aproximación se encuentran:
 - Devuelven resultados determinísticos
 - Permiten crear o reproducir determinados estados (por ejemplo errores de conexión)
 - Obtienen resultados mucho mas rápidamente y a menor coste, incluso offline.
 - Permiten el inicio temprano de las pruebas incluso cuando las dependencias todavía no están disponibles.
 - Permiten incluir atributos o métodos exclusivamente para el testeo.

© JMA 2019. All rights reserved

Aislar las pruebas

- Comienza cada prueba desde un estado limpio conocido, inicia un nuevo WebDriver para cada prueba, abrirá el navegador limpio. Para evitar fugas de memoria, al terminar la prueba se debe cerrar WebDriver: cierra el navegador, limpia los storages, cookies, ...
- Eliminar las dependencias de servicios externos mediante mocks o proxys mejorará en gran medida la velocidad y la estabilidad de las pruebas.

© JMA 2019. All rights reserved

Cubrir aspectos no evidentes

- Las pruebas no deben cubrir solo los casos evidentes, los correctos, sino que deben ampliarse a los casos incorrectos.
- Un juego de pruebas debe ejercitar la resiliencia: la capacidad de resistir los errores y la recuperación ante los mismos.
- En los cálculos no hay que comprobar solamente si realiza correctamente el calculo, también hay que verificar que es el calculo que se debe realizar.
- Los dominios de los datos determinan la validez de los mismos y fijan la calidad de la información, dichos dominios deben ser ejercitados profundamente.

© JMA 2019. All rights reserved

Respetar los limites de las pruebas

- Las pruebas unitarias ejercitan profundamente los componentes de formar aislada centrándose en la funcionalidad, los cálculos, las reglas de dominio y semánticas de los datos. Opcionalmente la estructura del código, es decir, sentencias, decisiones, bucles y caminos distintos.
- Las pruebas de integración se basan en componentes ya probados (unitaria o integración) o en dobles de pruebas y se centran en la estructura de llamadas, secuencias o colaboración, y la transición de estados.
- Hay muchos tipos de pruebas de sistema y cada uno pone el foco en un aspecto muy concreto, cada prueba solo debe un solo aspecto. Las pruebas funcionales del sistema son las pruebas de integración de todo el sistema centrándose en compleción de la funcionalidades y los procesos de negocio, su estructura, disponibilidad y accesibilidad.

© JMA 2019. All rights reserved

Malas prácticas

- Los CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart o Prueba de Turing Completamente Automática y Pública para diferenciar Ordenadores de Humanos), están explícitamente diseñados para prevenir la automatización, así que se debe intentar automatizarlo. Existen dos estrategias principales para evitar los CAPTCHAs:
 - Deshabilitar los CAPTCHAs en tus entornos de pruebas.
 - Agrega un parámetro que permita que las pruebas hagan un baipás.
- Aunque los servidores informan con códigos HTTP 4xx y 5xx de los errores, las pruebas de WebDriver (al representar al usuario) no deben intentar interceptarlos, deben validar los títulos o algún contenido de confianza de las paginas de error.
- Por múltiples razones, autenticarse en sitios como Gmail y Facebook usando el WebDriver (aparte de estar en contra de los términos y condiciones de estos sitios, te expones a que te cierren la cuenta), no esta recomendado dado es un proceso lento, poco fiable y puede ser considerado un ataque por parte del proveedor.

© JMA 2019. All rights reserved

Limitaciones

- Hay una serie de tipos de pruebas que no es aconsejable realizar con Selenium WebDriver, no porque sea incapaz de ello, si no porque no esta optimizado para ello y es poco probable que se obtengan buenos resultados.
- Específicamente no esta recomendado para:
 - Pruebas de rendimiento, estrés, carga, ...
 - Rastreo de enlaces
- Existen múltiples herramientas especificas, como Jmeter, Google Webmaster Tools o LinkChecker, que permiten realizar dichas pruebas de forma simplificada, siendo mas fiables y eficiente.
- La autenticación de dos factores (2FA) es un mecanismo de autorización en el que se genera una contraseña de un solo uso (OTP) utilizando aplicaciones móviles de "Autenticador" como "Google Authenticator", "Microsoft Authenticator", etc., o por SMS, correo electrónico para autenticar. Hay pocas opciones para sortear los controles 2FA y, por lo tanto, se debe evitar la automatización de 2FA.

© JMA 2019. All rights reserved

Documentar defecto

- Una vez detectado un defecto, en la comunicación es muy útil la siguiente información:
 - Síntomas
 - ¿Cuál es el problema?
 - ¿Sucede cada vez que ejecuta la prueba? Si no, ¿con qué frecuencia?
 - ¿Hay otros experimentando el problema?
 - Un enlace a la esencia de cualquier rastro de pila interesante
 - Lo que has probado hasta ahora
 - ¿Eres capaz de reproducir el error en otros navegadores?
 - ¿Tiene otros cambios en su código que podrían estar afectando esta prueba?
 - Pasos para reproducir
 - ¿Qué sistema operativo, navegador y versión del navegador está utilizando? Esto es vital
 - Idealmente, un caso de prueba reducido.
 - Cómo ejecutar la prueba (si se necesita alguna configuración especial)

© JMA 2019. All rights reserved

RECOMENDACIONES

© JMA 2019. All rights reserved

Patrones de diseño

- Page Objects: una simple abstracción de la interfaz de usuario de su aplicación web.
 - Componente Loadable: Modelado de PageObjects como componentes.
 - BotStyleTests: Uso de un enfoque basado en comandos para automatizar pruebas, en lugar del enfoque basado en objetos que PageObjects fomenta
 - Domain Driven Design: Expresa las pruebas en el idioma del usuario final de la aplicación.
 - Acceptance Test Driven Development (ATDD): técnica conocida también como Story Test-Driven Development (STDD), utiliza las pruebas de aceptación para ayudar a estructurar el trabajo de desarrollo.
-

© JMA 2019. All rights reserved

Lenguaje de dominio específico

- Un lenguaje de dominio específico (DSL) es un sistema que proporciona al usuario un medio expresivo para resolver un problema. Permite a un usuario interactuar con el sistema en sus términos, no solo en jerga del programador.
- A los usuarios, en general, no les importa principalmente cómo se ve su sitio, no están preocupados por la decoración, animaciones o gráficos. Lo que realmente les importa es como el sistema se impulsa a través de los procesos con una mínima dificultad. El trabajo del probador debe acercarse lo más que pueda a “capturar” esta mentalidad utilizando un lenguaje ubicuo (la terminología del usuario). Con eso en mente, los scripts de prueba deberían ser legibles al usuario.
- Con Selenium, el DSL generalmente se representa por métodos, la elección de los sus nombres es de suma importancia, así como la refactorización para ocultar la complejidad.
- Beneficios
 - Legible: Las partes interesadas del negocio pueden entenderlo.
 - Escribible: Fácil de escribir, evita duplicaciones innecesarias.
 - Extensible: Se puede agregar funcionalidad (razonablemente) sin romper los contratos y la funcionalidad existente.
 - Mantenible: Al dejar los detalles de implementación fuera de casos de prueba, está bien aislado contra cambios en el aplicación.

© JMA 2019. All rights reserved

Patrón Page Objects

- <https://martinfowler.com/bliki/PageObject.html>
- Cuando se escriben pruebas de una página web, hay que acceder a los elementos dentro de esa página web para hacer clic en los elementos, teclear entradas y determinar lo que se muestra.
- Sin embargo, si se escriben pruebas que manipulan los elementos HTML directamente, las pruebas serán frágiles ante los cambios en la interfaz de usuario.
- Un objeto de página envuelve una página HTML, o un fragmento, con una API específica de la aplicación, lo que permite manipular los elementos de la página sin excavar en el HTML.

© JMA 2019. All rights reserved

Patrón Page Objects

- La regla básica para un objeto de página es que debe permitir que un cliente de software haga cualquier cosa y vea todo lo que un humano puede hacer.
- El objeto de página debe proporcionar una interfaz que sea fácil de programar y oculta en la ventana.
- Entonces, para acceder a un campo de texto, debe tener métodos de acceso que tomen y devuelvan una cadena, las casillas de verificación deben usar valores booleanos y los botones deben estar representados por nombres de métodos orientados a la acción.
- El objeto de la página debe encapsular los mecanismos necesarios para encontrar y manipular los datos en el propio control GUI.
- A pesar del término objeto de "página", estos objetos no deberían construirse para cada página, sino para los elementos significativos en una página.
- Los problemas de concurrencia y asincronía son otro tema que un objeto de página puede encapsular.

© JMA 2019. All rights reserved

Ventajas del patrón Page Objects

- De acuerdo con patrón Page Object, deberíamos mantener nuestras pruebas y localizadores de elementos por separado, esto mantendrá el código limpio y fácil de entender y mantener.
- El enfoque Page Object hace que el programador de marcos de automatización de pruebas sea más fácil, duradero y completo.
- Otra ventaja importante es que nuestro repositorio de objetos de página es independiente de las pruebas de automatización. Mantener un repositorio separado para los objetos de la página nos ayuda a usar este repositorio para diferentes propósitos con diferentes marcos como, podemos integrar este repositorio con otras herramientas como JUnit / NUnit / PHPUnit, así como con TestNG / Cucumber / etc.
- Los casos de prueba se vuelven cortos y optimizados, ya que podemos reutilizar los métodos de objetos de página.
- Los casos de prueba se centran solamente en el comportamiento.
- Cualquier cambio en la IU se puede implementar, actualizar y mantener fácilmente en los objetos y clases de página sin afectar a los casos de pruebas que no estén implicados.

© JMA 2019. All rights reserved

Page Objects

```
public class LoginPage {
    private WebDriver driver;
    private final By txtUsuarioBy = By.id("txtUsuario");
    private final By txtPasswordBy = By.id("txtPassword");
    private final By btnSendLoginBy = By.id("btnSendLogin");
    private final By userDataBy = By.id("userData");

    public LoginPage(WebDriver driver) {
        this.driver = driver;
        driver.get("http://localhost/login");
    }
    public void ponUsuario(String valor) { driver.findElement(txtUsuarioBy).sendKeys(valor); }
    public void ponPassword(String valor) { driver.findElement(txtPasswordBy).sendKeys(valor); }
    public void enviarLogin() { driver.findElement(btnSendLoginBy).click(); }
    public String textoSaludo() {
        WebElement element = new WebDriverWait(driver,
            Duration.ofSeconds(3).getSeconds()).until(driver -> driver.findElement(userDataBy));
        return element.getText();
    }
}
```

© JMA 2019. All rights reserved

Prueba usando un Page Objects

@Test

```
public void loginTest() {
    LoginPage page = new LoginPage(driver);
    page.ponUsuario("admin");
    page.ponPassword("P@$w0rd");
    page.enviarLogin();
    assertThat(page.textoSaludo(), is("Hola
        Administrador"));
}
```

© JMA 2019. All rights reserved

PageFactory

- La biblioteca de soporte de WebDriver contiene una clase de PageFactory que simplifica la implementación del patrón PageObject.
- La PageObject declara sus campos como WebElements o List<WebElement>:
private WebElement txtUsuario;
- Cuando ejecutamos la factoría, PageFactory buscará un elemento en la página que coincida con el nombre del campo WebElement en la clase e inyectará el elemento en el campo. Para ello, primero busca un elemento con un atributo de ID coincidente. Si esto falla, recurre a la búsqueda de un elemento por el valor de su atributo "name".
LoginPageFactory page = new LoginPageFactory(driver);
PageFactory.initElements(driver, page);
- La anotación @FindBy permite elegir un nombre significativo de campo y cambiar la estrategia utilizada para buscar el elemento:
@FindBy(how = How.CSS, css = "#userData")
private WebElement saludo;

© JMA 2019. All rights reserved

PageFactory

```
public class LoginPageFactory {  
    private WebDriver driver;  
    private WebElement txtUsuario;  
    private WebElement txtPassword;  
    private WebElement btnSendLogin;  
    @FindBy(how = How.CSS, css = "#userData")  
    private WebElement saludo;  
  
    public LoginPageFactory(WebDriver driver) { this.driver = driver; }  
    public void ponUsuario(String valor) { txtUsuario.sendKeys(valor); }  
    public void ponPassword(String valor) { txtPassword.sendKeys(valor); }  
    public void enviarLogin() { btnSendLogin.click(); }  
    public String textoSaludo() { return saludo.getText(); }  
}
```

```
LoginPageFactory page = PageFactory.initElements(driver, LoginPageFactory.class);
```

© JMA 2019. All rights reserved

Pruebas de estilo bot

- Un "bot" es una abstracción orientada a la acción sobre las API de Selenium sin procesar. Permite ocultar la complejidad del API con comandos mayor nivel que son fácil de cambiar.

```
public class ActionBot {  
    private final WebDriver driver;  
    public ActionBot(WebDriver driver) { this.driver = driver; }  
    public void click(By locator) { driver.findElement(locator).click(); }  
    public void submit(By locator) { driver.findElement(locator).submit(); }  
    public void type(By locator, String text) {  
        WebElement element = driver.findElement(locator);  
        element.clear();  
        element.sendKeys(text);  
    }  
}
```
- Una vez que se han construido estas abstracciones y se ha identificado la duplicación en sus pruebas, es posible colocar los PageObjects encima de los bots.

© JMA 2019. All rights reserved

API fluída

- Un API fluida, termino acuñado por Martin Fowler y Eric Evans, permite encadenar varias acciones consecutivas proporcionando una sensación más fluida al código.

```
String saludo = sitio.pedirLogin().rellenarFormulario("admin",  
    "P@$wOrd").enviarFormulario().dameTextoSaludo();
```
- Crear un API fluida significa construirla de tal manera que cumpla con los siguientes criterios:
 - El usuario de la API puede entenderla API muy fácilmente.
 - El API puede realizar una serie de acciones para finalizar una tarea.
 - El nombre de cada método debe utilizar la terminología específica del dominio.
 - La API debe ser lo suficientemente sugerente como para guiar a los usuarios del API sobre qué hacer a continuación y qué posibles operaciones se pueden en un determinado momento.
- Son interfaces o clases que cuando invocamos a un método concreto nos devuelve el mismo objeto modificado u otro objeto. De tal forma que podemos volver a solicitar otro método del mismo objeto, y encadenar más operaciones, o continuar las operaciones con el nuevo objeto.
- Se recomienda considerar el uso del patrón de diseño Fluent API en el objeto de página. El API de Selenium WebDriver suministra la clase base LoadableComponent que tiene como objetivo simplificar la creación de PageObjects fluidos.

© JMA 2019. All rights reserved

LoadableComponent

- LoadableComponent proporciona una forma estándar de garantizar que las páginas se carguen y facilita la depuración de los errores de carga de la página. Ayuda a reducir la cantidad de código repetitivo en las pruebas, lo que a su vez hace que el mantenimiento de las pruebas sea menos agotador.
- La clase cuenta con tres métodos:
 - public T get():
 - Devuelve el PageObject si la página está cargada o cargándola si es necesario.
 - protected abstract void load():
 - Cuando este método termine, el componente modelado por la subclase debería estar completamente cargado. Se espera que esta subclase navegue a una página apropiada si fuera necesario.
 - protected abstract void isLoading() throws Error
 - Determine si el componente está cargado o no, lanzando un Error (no una excepción) cuando no esté cargada u obsoleta. Se carga el componente, este método volverá, pero cuando no se carga, se debe lanzar un. Esto permite la verificación compleja y el informe de errores al cargar una página o cuando una página no se carga.

© JMA 2019. All rights reserved

LoadableComponent

```
public class LoginPageFluent {
    private WebDriver driver;
    private WebElement txtUsuario;
    private WebElement txtPassword;
    private WebElement btnSendLogin;
    @FindBy(how = How.CSS, css = "#userData")
    private WebElement saludo;

    public LoginPageFactory(WebDriver driver) { this.driver = driver; }
    @Override
    protected void load() {
        driver.get("http://localhost/login");
        PageFactory.initElements(driver, this);
    }
    @Override
    protected void isLoading() throws Error {
        if(driver.getTitle() == null || !driver.getTitle().contains("Login"))
            throw new Error();
    }
}
```

© JMA 2019. All rights reserved

LoadableComponent

```
public LoginPageFluent ponUsuario(String valor) {
    txtUsuario.sendKeys(valor); return this;
}
public LoginPageFluent ponPassword(String valor) {
    txtPassword.sendKeys(valor); return this;
}
public LoginPageFluent enviarLogin() {
    btnSendLogin.click(); return this;
}
public String textoSaludo() { return saludo.getText(); }
}

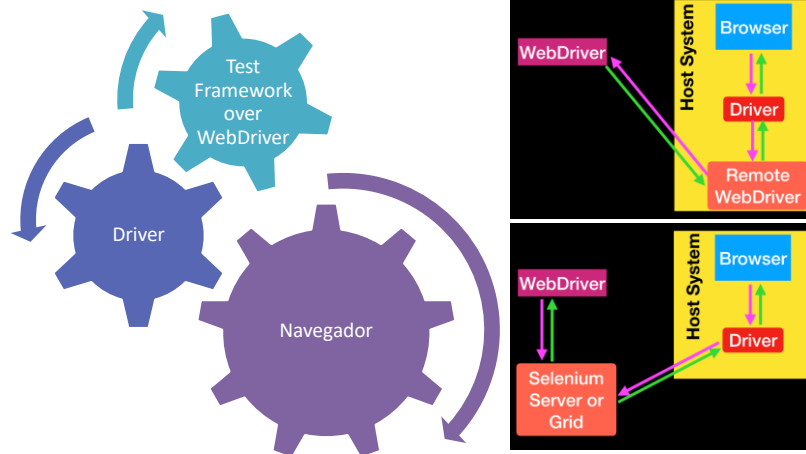
LoginPageFluent page = new LoginPageFluent();
assertEquals("Hola Administrador", page.ponUsuario("admin")
    .ponPassword("P@$w0rd").enviarLogin().textoSaludo());
```

© JMA 2019. All rights reserved

SELENIUM GRID

© JMA 2019. All rights reserved

Interacciones



© JMA 2019. All rights reserved

Remote WebDriver

- Se puede usar WebDriver de forma remota de la misma manera que se usaría localmente. La principal diferencia es que un WebDriver remoto debe ser configurado para que pueda ejecutar las pruebas en una máquina diferente.
- Un WebDriver remoto se compone de dos piezas: un cliente y un servidor. El cliente es la prueba de WebDriver y el servidor es simplemente un servlet Java, que se puede alojar en cualquier servidor moderno de aplicaciones JEE.
- Pros
 - Separa dónde se ejecutan las pruebas desde donde está el navegador.
 - Permite que las pruebas se ejecuten con navegadores no disponibles en el sistema operativo actual
- Contras
 - Requiere que se ejecute un contenedor de servlet externo
 - Puede encontrar problemas con los finales de línea al obtener texto del servidor remoto
 - Introduce latencia adicional a las pruebas, sobre todo con las excepciones.

© JMA 2019. All rights reserved

Servidor

- El servidor siempre se ejecutará en la máquina con el navegador que deseas probar. Debe tener instalados los diferentes driver para los navegadores.
- El servidor se puede usar desde la línea de comandos o mediante configuración de código. Standalone combina todos los componentes de Grid todo en uno en una sola máquina.
`java -jar selenium-server-<version>.jar standalone`
- Para personalizar la configuración por defecto, se utiliza un archivo de configuración JSON:

```
{
  "server": {
    "port": 4449
  },
}
```

`java -jar selenium-server-<version>.jar standalone -config standaloneConfig.json`
- Esta disponible una consola en: <http://localhost:4444/>

© JMA 2019. All rights reserved

Cliente (java)

- Las pruebas que quieran ejecutarse en remoto deben sustituir la instancia de WebDriver por una instancia de RemoteWebDriver.
- El constructor recibe dos parámetros:
 - URL con la dirección del servidor que ejecutara las pruebas.
 - Una instancia de las opciones del navegador que debe usar el servidor remoto (browserName).
- Para implementar la prueba de ejecución remota:

```
FirefoxOptions firefoxOptions = new FirefoxOptions();
WebDriver driver = new RemoteWebDriver(new
    URL("http://localhost:4444"), firefoxOptions);
driver.get("http://www.google.com");
driver.quit();
```

© JMA 2019. All rights reserved

Cliente (js)

- Las pruebas que quieran ejecutarse en remoto deben sustituir la instancia de WebDriver por una instancia de RemoteWebDriver.
- El constructor recibe dos parámetros:
 - URL con la dirección del servidor que ejecutara las pruebas.
 - Una instancia de las opciones del navegador que debe usar el servidor remoto (browserName).
- Para implementar la prueba de ejecución remota:

```
const { Builder, Capabilities } = require("selenium-webdriver");
let driver = new Builder()
    .usingServer("http://localhost:4444")
    .withCapabilities(Capabilities.firefox())
    .build();
await driver.get('http://www.google.com');
await driver.quit();
```

© JMA 2019. All rights reserved

Grid

- Selenium Grid es un servidor inteligente que actúa de proxy lo que permite a los tests de Selenium enrutar sus comandos hacia instancias remotas de navegadores web. La intención es proporcionar una forma sencilla de ejecutar los tests en paralelo en múltiples máquinas.
- Con Selenium Grid un servidor actúa como el centro de actividad (hub) encargado de enrutar los comandos de los tests en formato JSON hacia uno o más nodos registrados en el Grid. Los tests contactan con el hub para obtener acceso a las instancias remotas de los navegadores.
- Selenium Grid permite ejecutar los tests en paralelo en múltiples máquinas y gestionar diferentes versiones de navegadores y configuraciones de manera centralizada (en lugar de hacerlo de manera individual en cada test).

© JMA 2019. All rights reserved

Grid

- Selenium Grid no es una solución mágica para todos los problemas. Permite resolver un subconjunto de problemas comunes de delegación y distribución, pero no administrará su infraestructura y podría no satisfacer necesidades concretas.
- Sus principales características son:
 - Punto de entrada centralizado para todos los tests
 - Gestión y control de los nodos / entornos donde se ejecutan los navegadores
 - Escalado
 - Balanceo de carga
 - Ejecución de los tests en paralelo
 - Testing cruzado entre diferentes sistemas operativos

© JMA 2019. All rights reserved

Casos de uso

- El Grid se usa para acelerar la ejecución de los test usando múltiples máquinas para ejecutarlos en paralelo: en cuatro máquinas tardaría aproximadamente una cuarta parte de lo que tardaría en una sola. Esto permite ejecutar suites muy grandes, y de larga duración (horas), en un tiempo razonable, así como obtener una retroalimentación temprana.
- El Grid permite ejecutar la suite múltiples entornos, especialmente, contra diferentes navegadores y versiones al mismo tiempo, lo cual sería imposible en una sola máquina. Cuando la suite de test es ejecutada, el Selenium Grid recibe cada combinación de test-navegador y lo asigna para su ejecución a la máquina o máquinas con el navegador requerido.
- Remote WebDriver trabaja en escenarios de uso con un cliente para un remoto, Selenium Grid reutiliza la misma infraestructura para que un cliente use múltiples remotos.

© JMA 2019. All rights reserved

Hub

- El Hub es un punto central donde se envían todos los tests. Cada Grid tiene un único hub. El hub necesita ser accesible desde la perspectiva de los clientes (ej. Servidor de la CI, maquina del desarrollador). El hub se conectará a uno o mas nodos en los que los tests serán ejecutados.
- Características:
 - Ejerce como mediador y administrador
 - Acepta peticiones para ejecutar los tests
 - Recoge instrucciones de los clientes y las ejecuta de forma remota en los nodos
 - Gestiona los hilos

© JMA 2019. All rights reserved

Hub

- Para arrancar el Hub:
`java -jar selenium-server-<version>.jar node`
- El hub escuchará al puerto 4444 por defecto. Se puede ver el estado del hub abriendo una ventana del navegador y navegando a <http://localhost:4444/> o consultando <http://localhost:4444/status>.
- Un Hub es la unión de los siguientes componentes:
 - Router
 - Distributor
 - Session Map
 - New Session Queue
 - Event Bus
- Cuando se utiliza un Grid distribuido, cada componente debe iniciarse por separado. Esta configuración es más adecuada para Grids grandes.

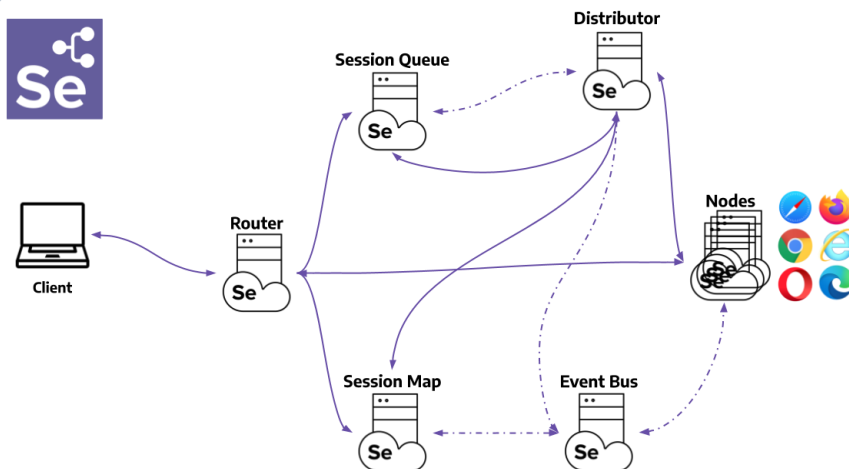
© JMA 2019. All rights reserved

Configuración distribuida

- Enrutador: redirige las solicitudes al componente correcto
 - `java -jar selenium-server-<version>.jar router --sessions http://localhost:5556 --distribuidor http://localhost:5553 --sessionqueue http://localhost:5559`
- Cola de sesiones: agrega la solicitud de nueva sesión a una cola para que el distribuidor la procese
 - `java -jar selenium-server-<version>.jar sessionqueue`
- Mapa de sesión: asigna ID de sesión al nodo donde se ejecuta la sesión
 - `java -jar selenium-server-<version>.jar sessions`
- Distribuidor: asigna un Nodo para una solicitud de sesión. Otros Nodos se registran en el Distribuidor de la misma manera que se habrían registrado en el Hub en una Red no distribuida
 - `java -jar selenium-server-<version>.jar distribuidor --sessions http://localhost:5556 --sessionqueue http://localhost:5559 --bind-bus false`
- Bus de eventos: sirve como ruta de comunicación entre los nodos, el distribuidor, la cola de nuevas sesión y el mapa de sesión.
 - `java -jar selenium-server-<version>.jar event-bus`

© JMA 2019. All rights reserved

Arquitectura



© JMA 2019. All rights reserved

Nodos

- Los nodos son diferentes instancias de Selenium que ejecutarán los tests en sistemas informáticos individuales. Puede haber tantos nodos en un grid como sean necesarios.
- Las maquinas que contienen los nodos no necesitan disponer del mismo sistema operativo o el mismo conjunto de navegadores que el hub o los otros nodos. Un nodo en Windows podría tener la capacidad de ofrecer Internet Explorer como opción del navegador mientras que esto no podría ser posible en Linux o Mac.
- Características:
 - Donde se ubican los navegadores
 - Se registra a si mismo en el hub y le comunica sus capacidades
 - Recibe las peticiones desde el hub y las ejecuta
- Se pueden iniciar uno o más Nodos y el servidor detectará en el PATH los controladores disponibles:
 - `java -jar selenium-server-<version>.jar node`

© JMA 2019. All rights reserved

Nodos

- Al iniciar los nodos deben indicar la url del hub (si no se especifica un puerto a través del parámetro `-port` se elegirá un puerto libre):
`java -jar selenium-server-standalone.jar -role node -hub http://localhost:4444`
- Para personalizar la configuración por defecto, se utiliza un archivo de configuración JSON:

```
{
  "capabilities": [
    { "browserName": "firefox", "maxInstances": 2, "seleniumProtocol": "WebDriver" },
    { "browserName": "chrome", "maxInstances": 3, "seleniumProtocol": "WebDriver" }
  ],
  "proxy": "org.openqa.grid.selenium.proxy.DefaultRemoteProxy",
  "maxSession": 5, "port": -1, "register": true, "registerCycle": 5000,
  "hub": "http://localhost:4444", "nodeStatusCheckTimeout": 5000, "nodePolling": 5000,
  "unregisterIfStillDownAfter": 60000, "downPollingLimit": 2, "debug": false,
  "servlets": [], "withoutServlets": [], "custom": {}
}
```

```
java -jar selenium-server-standalone.jar -role node -nodeConfig nodeConfig.json
```

© JMA 2019. All rights reserved

Seguridad

- El Grid de Selenium debe estar protegido contra accesos externos mediante el uso apropiado de los permisos del firewall.
- Fallar a la hora de proteger el Grid puede ocasionar uno o mas de los siguientes problemas:
 - Permitir acceso abierto a tu infraestructura del Grid.
 - Permitir a terceros el acceso a aplicaciones web y archivos interno.
 - Permitir a terceros ejecutar tus ejecutables.

© JMA 2019. All rights reserved

Cloud

- Para usar Selenium Grid, se necesita mantener una infraestructura para los nodos. Como esto puede suponer un engorro y un gran esfuerzo de tiempo y recursos, se pueden usar proveedores de IaaS (Infraestructura como servicio) en la nube como Amazon EC2 o Google Compute para proveer de la infraestructura. Otras opciones incluyen usar proveedores como Sauce Labs or Testing Bot los cuales proveen Selenium Grid como servicio en la nube.
- Docker provee una forma conveniente de aprovisionar y escalar la infraestructura de Selenium Grid en unidades conocidas como contenedores. Los contenedores son unidades estandarizadas de software que contienen todo lo necesario para ejecutar la aplicación deseada, incluidas todas las dependencias, en un entorno confiable y regenerable en diferentes sistemas.
- El proyecto de Selenium mantiene un conjunto de imágenes Docker, las cuales se pueden descargar y ejecutar para tener un Grid funcionando rápidamente. Los nodos están disponibles para los navegadores Edge, Firefox y Chrome.
 - <https://github.com/SeleniumHQ/docker-selenium>

© JMA 2019. All rights reserved

Standalone con Docker

- Iniciar un contenedor Docker con Firefox
 - docker run -d -p 4444:4444 -p 7900:7900 --shm-size="2g" --name standalone-firefox selenium/standalone-firefox:4.4.0-20220812
- Establecer RemoteWebDriver o usingServer apuntando a <http://localhost:4444>
- Para ver lo que sucede dentro del contenedor:
 - <http://localhost:7900> (la contraseña es secret)
- Al ejecutar docker run para una imagen que contiene un navegador, debería indicarse la opción --shm-size=2g para usar la memoria compartida del host.
- Se debería utilizar siempre una imagen de Docker con una etiqueta completa para anclar un navegador específico y una versión de Grid.

© JMA 2019. All rights reserved

Selenium Hub: docker-compose.yml

```
version: "3"
services:
  chrome:
    image: selenium/node-chrome:4.4.0-20220812
    shm_size: 2gb
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
  edge:
    image: selenium/node-edge:4.4.0-20220812
    shm_size: 2gb
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
  firefox:
    image: selenium/node-firefox:4.4.0-20220812
    shm_size: 2gb
    depends_on:
      - selenium-hub
    environment:
      - SE_EVENT_BUS_HOST=selenium-hub
      - SE_EVENT_BUS_PUBLISH_PORT=4442
      - SE_EVENT_BUS_SUBSCRIBE_PORT=4443
  selenium-hub:
    image: selenium/hub:4.4.0-20220812
    container_name: selenium-hub
    ports:
      - "4442:4442"
      - "4443:4443"
      - "4444:4444"
# docker compose up -d
```

© JMA 2019. All rights reserved