Übersicht

* Selenium
* Expresso

Selenium

Was ist Selenium?

Selenium ist ein open source Tool um verschieden Browser fernzusteuern. That’s it. Mehr macht es nicht. Was damit gemacht wird, wird dem Benutzer überlassen. Es kann und wird aber hauptsächlich für UI-Tests (Benutzeroberflächentests) eingesetzt, aber auch die automatische Steuerung per "Screen Scraping" ist möglich.

* Functionality Testing which includes Smoke Testing , Regression Testing , and Sanity Testing
* Browser Compatibility Testing (Cross browser and Different versions of same browser)
* UI Testing
* Api Testing

Dabei emuliert es einen User beziehungsweise dessen Verhalten. Selenium ist nur speziell für das Testen von Websiten entwickelt, das bedeutet es gibt keinen Support für Desktop oder Mobile Applikationen.

Fun fact: Der Erfinder von Selenium Jason Huggins hat einmal ein Scherz gerissen das Selenium eine Quecksilbervergiftung heilen kann. Und Mercury Interactive war zu diesem Zeitpunkt die Konkurrenzfirma. Seine Mitarbeiter fanden den Namen gut und haben ihn als Firmennamen genommen.

Selenium ist sehr weit verbreitet, da es Open Source ist und die populärsten Browser, Betriebssysteme und 7 Programmiersprachen unterstützt.

Hier ist eine genaue Auflistung:

Unterstützt

* Sprachen: Java, Python, C#, PHP, Ruby, Perl & .NET
* Betriebssystem: Windows, Linux, Mac
* Browser: Safari, Internet Explorer, Chrome, Mozilla Firefox, Opera

Selenium Suite Of Tools

* Selenium RC (Now deprecated)
* Selenium IDE
* Selenium Grid
* **Selenium WebDriver**

Selenium Remote Control

Stellt einen Selenium Server zur Verfügung. Dieser bekommt Selenium Befehle von unserem Programm über einfache HTTP-GET / POST Request(client libaray), interpretiert sie und schickt unseren Programm die Ergebnisse dieser Tests. Der RC Server bündelt dabei Selenium Core und fügt es automatisch in den Browser ein. Dies geschieht, wenn unser Testprogramm den Browser öffnet. Selenium Core ist ein Set von Javascript-Funktionen und ist dafür zuständig unsere Befehle im Browser auszuführen.

Es wurde vom Selenium WebDriver ersetzt.

Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Selenium IDE

Selenium IDE ist ein Add-on für Mozilla Firefox, mit dem es möglich ist, direkt im Browser durch die Interaktion mit einer Webanwendung Testfälle aufzunehmen und diese im Browser wieder abzuspielen. Neben der reinen Capture-Replay-Funktionalität lassen sich Überprüfungen mittels verify und assert einbinden. Zudem ist das schrittweise Abspielen von Testfällen und das Setzen von Breakpoints zur Überprüfung der Testfälle möglich

Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Selenium Grid:

Selenium Grid ist eine Erweiterung von Selenium RC und Webdriver. Dieser ermöglicht die parallele Ausführung von Tests auf mehreren Rechnern und Brwoser, um die Dauer der Testdurchführung zu verkürzen.

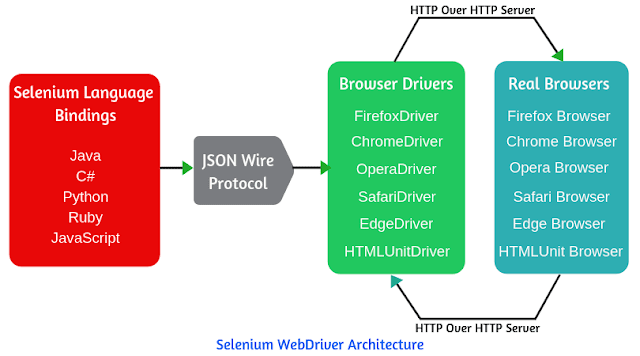
Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Selenium Webdriver?

Ist ein Browser automation Framework, welches es uns erlaubt unsere Tests auf verschiedenen Browser zu testen. Damit wir unseren Browser steuern können brauchen wir einen Browser spezifischen Treiber. Dieser kommuniziert direkt auf OS Ebene mit dem Browser.

Selenium WebDriver Architecture



Der Selenium WebDriver besteht aus 4 Komponenten.

* Selenium Client Library
* JSON Wire Protocol over HTTP
* Browser Drivers
* Browsers

**Selenium Client Libraries/Language Bindings:**

Wir schreiben das Testskript mit einer Programmiersprache wie Java, C #, Python usw. zur Automatisierung von Webanwendungen, die API versteht diese Sprachen jedoch nicht.

Die Selenium-Entwickler haben Language Bindings entwickelt, damit Selenium mehrere Sprachen unterstützen kann. Die Default Language von Selenium ist **Selenese** (Selenium IDE).

**JSON WIRE PROTOCOL Over HTTP Client:**

JSON Wire Protocol ist ein von WebDriver-Entwicklern erstellter Transportmechanismus, der die Daten zwischen einem Server und einem Client im Web überträgt. Es verwendet eine REST-API, um die Informationen zwischen dem HTTP-Server zu übertragen. Jeder BrowserDriver (wie FirefoxDriver, ChromeDriver usw.) verfügt dabei über einen eigenen HTTP-Server

**Browser Drivers:**

Da jeder Browser eine andere Logik hat, um verschiedene Aktionen auszuführen, z. B. Klicken, Öffnen und Schließen und das Problem besteht das nicht jeder Browser open source ist, Chrome zum Beispiel. Wir wissen daher nicht wie der Quellcode aussieht und haben dadurch keinen Einblick in die Logik. Um diese Art von Problemen zu beheben, spielt hier der Browser Treiber eine wichtige Rolle.  
Jeder Browser enthält einen separaten Browsertreiber. Browser-Treiber kommunizieren mit dem jeweiligen Browser, ohne die interne Logik der Browser-Funktionalität preiszugeben. Wenn ein Browsertreiber einen Befehl erhält, wird dieser Befehl im jeweiligen Browser ausgeführt und das Ergebnis wird in Form einer HTTP-Response an den Client zurückgesendet.

Wie funktioniert der WebDriver als Ganzes jetzt?

Wenn man ein Testskript ausführt, werden die folgenden Schritte intern durchgeführt.

1. Wenn das Programm ausgeführt wird, führt die Selenium-Client-Library Selenium-spezifische Befehle aus unserem Programm aus und konvertiert sie in das JSON-Format. Das Ergebnis wird über das JSON Wire Protocol über HTTP an den Browsertreiber gesendet.(z. B. ChromeDriver).

WebDriver browser = new ChromeDriver();

browser.get("http://orteil.dashnet.org/cookieclicker/");

***https://localhost:7705/ {"url": "*** http://orteil.dashnet.org/cookieclicker***"}***

2. Der HTTP-Server führt alle spezifischen Aktionen oder Anweisungen im realen Browser aus.   
  
3. Nachdem alle Anweisungen ausgeführt wurden, wird der Ausführungsstatus an den HTTP-Server zurückgesendet. Der Browsertreiber verwendet erneut den HTTP-Server, um die HTTP-Response zu empfangen und über JSON Wire Protocol an den Client zu senden. Wir erfahren dann ob die Operation erfolgreich war oder nicht.

Chromedriver

https://sites.google.com/a/chromium.org/chromedriver/downloads

**Browser Navigation**

Browser.get(): geht zur einer bestimmten Website, es speichert aber nicht den Verlauf und Cookies. Das bedeutet wir können nicht hin und zurück navigieren. Bei einer SPA würde es außerdem jedesmal die Seite neu laden.

Browser.navigate().to(): wird benutzt um zur einer bestimmten Website mit der URL zu navigieren. Anders wie beim get() wird hier der Browserverlauf und Cookies gespeichert und wir können hin und zurück navigieren.

**Locating Webelements (mit findElements bekommen wir ein Webelement zurück und können diesen dann manipulieren/Aktionen ausführen und Daten auslesen)**

• driver.findElement(By)

– 0 match -> throws exception

– 1 match -> returns a WebElement instance

– 2+ matches -> returns only the first match from web page

• driver.findElements(By)

– 0 match -> returns an empty list

– 1 match -> returns a list with one WebElement

– 2+ matches -> returns list with all matching WebElements

**By (sind unsere Suchkriterien und ermöglichen uns ein Element auf mehrere Arten zu suchen)**

By locating mechanisms

* Id: driver.findElement(By.id(”some\_id”));
* ClassName: driver.findElement(By.className(”some\_class\_name”));
* LinkText: driver.findElement(By.linkText(”Sign in”));
* PartialLinkText: driver.findElement(By.partiallinkText(”Sign”));
* Name:

<input id=”modCustLoginPassword” name=”password”>

driver.findElement(By.name(”password”));

* TagName:

<label>Email address</label> driver.findElement(By.tagName(”label”));

* CssSelector: driver.findElement(By.cssSelector(”.login”));
* XPath: driver.findElement(By.xpath(”//input[@id=’username’]”)

**Manipulation (sind Aktionen )**

Input:

element.sendKeys(”string”)

element.clear();

Buttons, Link, Checkbox, Combobox

element.click()

**Spezielle Aktionen:**

Actions(driver).keyDown(Keys.SHIFT).click(element).keyUp(Keys.SHIFT).build().perform();

Shift + Click

**•** Actions(driver).moveToElement(element).build().perform();

• Actions(driver).contextClick().build().perform();

• Actions(driver).doubleClick().build().perform();

• Actions(driver).clickAndHold().build().perform();

• Actions(driver).release().build().perform();

**Time Management/ Waits**

Da die Test sehr schnell ablaufen, kann es oft passieren, dass die getestete Seite nicht mit dem Rendering oder Laden von Elementen nachkommt. Deshalb muss man oft einige Sekunden warten, bevor der Test fortgesetzt werden kann. In Java wäre das einfach mit einem Thread.Sleep() möglich. Das beschränkt uns aber, weil wir nicht genau wissen wann das Element erscheint oder wie lange ein Task dauert. Das kann dazu führen, das ein Test sehr lange braucht. Um dem entgegen zu steuern gibt es von Selenium bereits vordefinierte Methoden.

**Implicit Wait**

Hier sagen wir dem WebDriver er soll ein die DOM für eine gewisse Zeit untersuchen, wenn versucht wird, ein Element oder Elemente zu finden, wenn diese nicht sofort verfügbar sind. Das ist hilfreich wenn gewisse Elemente auf der Website nicht sofort erhältlich sind und Zeit zum laden brauchen.

In diesem Beispiel öffnen wir eine neue Seite und warten maximal 30 Sekunden lang bis er ein Element mit der Id „item“ findet.

*ChromeDriver browser* = **new** ChromeDriver();

*browser*.navigate().to(**"https://web.whatsapp.com"**);

*browser*.manage().timeouts().implicitlyWait(30, TimeUnit.***SECONDS***);

WebElement searchInput = *browser*.findElement(By.*id(“item”))*

**Explicit Wait**

Ermöglichen uns den Testcase anzuhalten und auf ein oder mehrere Elemente zu warten bis eine gewisse Bedingung erfüllt wird. Diese Bedingung wird in regelmäßigen Abständen bis zum Timeout abgefragt. Explizite Wartezeiten sind auf ein bestimmtes Webelement beschränkt sind.

Explizite Wartezeiten werden mit den Klassen WebDriverWait und ExpectedConditions ausgeführt. Im folgenden Beispiel werden wir bis zu 20 Sekunden warten, bis mehr als 10 Elemente mit der ID „list-item“ sichtbar werden, bevor wir mit dem nächsten Befehl fortfahren. Hier sind die Schritte. Geschieht das nicht innerhalb dieser Zeit wird eine Exception geworfen.

*ChromeDriver browser* = **new** ChromeDriver();

WebDriverWait wait=**new** WebDriverWait(*browser*, 20);  
wait.until(ExpectedConditions.*numberOfElementsToBeMoreThan*(By.*id*(**"list-item"**),10));

**Fluent Wait**

Ähnlich wie bei Explicit Wait hier können wir noch zusätzlich bestimmen wie oft die Condition überprüft werden sollt und welche Exceptions ignoriert werden können.

Wait<WebDriver> wait = **new** FluentWait<WebDriver>(driver) .withTimeout(30, SECONDS) .pollingEvery(5, SECONDS) .ignoring(NoSuchElementException.class); WebElement foo = wait.until(ExpectedConditions);

Espresso

Bevor wir mit Espresso losstarten wäre es gut zu wissen welche Art von Tests in Android unterschieden wird.

In Android unterscheidet man hauptsächlich zwischen Unit und Instrumentation Test.

**Unit Tests**

Sind Tests um die funktionalen Einzelteile (*Units*) von Computerprogrammen zu [testen](https://de.wikipedia.org/wiki/Softwaretest), d. h., sie auf korrekte Funktionalität zu prüfen. Diese sind isolated, das bedeutet sie haben keinen Zugriff auf externe Ressourcen wie zum Beispiel eine Datenbank(Aushilfe mit Mocks) und sind unabhängig von anderen Units.

* Diese rennen direkt im JVM und haben keinen Zugriff auf die Android framework Klassen.
* Sie sind im test/java package
* Man verwendet üblicher Weise Mockito, JUnit für diese Tests

**Instrumentation Tests**

Fallen unter der Kategorie UI-Tests.

* Diese rennen direkt im Android Eemulator und haben direkten Zugriff auf Android classes
* Sie sind im androidTest/java package
* Und man verwendet hauptsächlich Espresso dafür

Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Das Espresso-Testframework bietet APIs zum Schreiben von UI-Tests zur Simulation von Benutzerinteraktionen innerhalb einer einzelnen App. Espresso-Tests werden auf dem tatsächlichen Gerät oder Emulator ausgeführt und verhalten sich so, als ob ein Benutzer die App verwendet. Espresso-Tests können auf Geräten mit Android 2.3.3 (API Level 10) und höher ausgeführt werden.

Was ist das besondere von Instrumentation Tests?

Diese können jedoch sowohl ein Testpaket als auch die App in denselben Prozess laden, was normalerweise nicht möglich ist. Da sich die App Components und die Tests im selben Prozess befinden, können die Tests Methoden in den Components aufrufen und Fields ändern und untersuchen.

Ein weiterer wichtiger Vorteil ist, dass Testaktionen automatisch mit der Benutzeroberfläche der App synchronisiert werden. Espresso erkennt, wenn der Main-Thread idle ist und führt es dann die nächste Operation aus. Durch diese Funktion erspart man sich außerdem schlechte workaround Lösungen wie das Thread.Sleep(). Weil wenn wir unseren Tests künstliche Delays hinzufügen, dauert es länger, bis die Testsuite abgeschlossen ist, und die Tests können trotzdem fehlschlagen, wenn sie auf langsameren Geräten ausgeführt werden.

Schauen wir uns an wie eine Test Klasse aufgebaut ist.

@RunWith(AndroidJUnit4::**class**)  
@MediumTest  
**class** LoginTest {  
@get:Rule  
 **var activityRule**: ActivityTestRule<LoginActivity>  
 = ActivityTestRule(LoginActivity::**class**.*java*)  
 @Test  
 **fun** attemptToLogin () {  
 }  
  
}

@Rule <-Junit spezifisch

Rules sind nichts anderes wie ein @Before und @After bzw. @BeforeClass und @AfterClass, mit denen lassen sich aber mehr Sachen machen und man kann auch seine eigene Rules schreiben.

Wir benutzen die von Espresso bereit gestellte ActivityTestRule. Diese launched im unseren Beispiel vor jeder Test Methode die MainActivity und zerstört sie auch wieder nach jedem Test. Über dieses Field haben wir dann auch Zugriff auf die Properties und Methoden der Activity.

@get:Rule <-für eigene rules

val mActivityRule = ActivityTestRule<>(

MainActivity.class);

Wenn wir das nur einmal ausführen wollen:

Companion object{

@ClassRule

@JvmField

val mActivityRule = ActivityTestRule<>(

MainActivity.class);

}

Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung

Note: The most current JUnit revision is JUnit 5. However for the purposes of using Espresso or UI Automator, version 4.12 is recommended.

@RunWith

Mit der Annotation @RunWith(AndroidJUnit4.class) sagen wir zu Begin welcher TestRunner zum Ausführen der Tests verwendet werden soll. Ein test runner ist eine Library das uns ermöglicht die Tests auszuführen und liefert einen Test Report.

@SmallTest, @MediumTest, and @LargeTest <- sind für den Developer

Mit @SmallTest, @MediumTest, and @LargeTest geben uns als Developer Einsicht welche Resourcen der Test verwendet und wie lange der Test ungefähr dauert. Zum Beispiel @SmallTest würde hier bedeuten dass der Test mit keinen externen Resourcen interagiert.

The following table summarizes what the annotations mean:



Jetzt kommen wir zum eigentlichen Testing.

Was macht einen guten UI-Test aus? Er simuliert genau das Verhalten, das auch ein realer Benutzer an den Tag legen würde: UI-Element auswählen, Aktion auf dem Element ausführen und abschließend das erwartete Ergebnis prüfen. Genau diese drei Elemente finden wir auch in der Espresso-API wieder:

• auswählen: onView(Matcher<View>)

• ausführen: perfom(ViewAction)

• prüfen: check(ViewAssertion)

Ein Bild, das Screenshot enthält.

Mit sehr hoher Zuverlässigkeit generierte Beschreibung