Neue Serie: Geschichte von NoSQL ▸ 82

S&S

JAVA Mag

5.2015

Java magazin

Java • Architekturen • Web • Agile

www.javamagazin.de

JNI, Rootbeer, CUDA
Parallelisierung auf der Grafikkarte ▸20

Apache Tamaya
Konfiguration in Java ▸91

RESILIENCE

Einführung in Resilient Software Design

©iStockphoto.com/Antrey

Sustainable Service Design: Nachhaltig – oder nur langlebig? ▸62

gulp: Frontend-Build-System auf Node.js-Basis ▸96

Android 5 Lollipop: Ein Blick auf die neuen Nexus-Geräte ▸122

Teil 2: Selenium parallelisieren im Grid

Testen mit Selenium

Die automatisierte Ausführung von Testläufen ist ein wichtiger Schritt im Softwareentwicklungsprozess, um zu gewährleisten, dass Probleme und Auffälligkeiten bei Codeänderungen schnell erkannt werden.

von Roland Rickborn und Tobias Kraft

Nachdem wir im ersten Teil des Artikels hauptsächlich auf die Aufzeichnungen von Tests mit Selenium Builder eingegangen sind, legen wir in diesem Teil den Schwerpunkt auf das Ausführen der Tests. Die bisher gezeigte Möglichkeit des Abspielens von Tests über das Seenium-Builder-Add-on im Firefox ist auf einen Browser beschränkt und muss aktiv durch den Anwender erfolgen. Für eine automatisierte Testdurchführung ist Selenium Builder deshalb eher ungeeignet. Des Weiteren sorgen Browserinkompatibilitäten immer wieder für Probleme, die nur durch Testläufe mit verschiedenen Browsern gefunden werden können.

Die Firma Sauce Labs [1], die wesentlich Selenium Builder entwickelt, bietet beispielsweise über ihre Cloud-Lösung eine entsprechende Infrastruktur an, die die Anforderungen der Automatisierung und des Cross-Browser-Testings erfüllt. Diese Lösung eignet sich aus rechtlichen Gründen für viele Projekte jedoch nicht, da die zu testenden Ressourcen über das Internet erreichbar sein müssen.

Selenium Interpreter mit Java

Mit dem Selenium Interpreter [2] steht eine freie Java-Bibliothek zur Verfügung, mit deren Hilfe aufgezeichnete Selenium-Builder-Skripte abgespielt werden können. Das Ausführen der Testsuite für die im ersten Teil des Artikels vorgestellten Mitteilungen kann mit dem folgenden Befehl auf der Kommandozeile erfolgen:

```
java -jar SeInterpreter.jar createNewsSuite.json
```

Es wird standardmäßig nur ausgegeben, ob die Ausführung erfolgreich verlief. Für detailliertere Informationen zu den abgearbeiteten Schritten muss der Loglevel auf DEBUG umgestellt werden.

Die Bibliothek bietet aktuell die Möglichkeit zur Nutzung von Firefox sowie des Remote-Drivers zum Abspielen der Tests. Mit Webdriver als Basis lassen sich jedoch mit ein wenig Eigenarbeit einfach weitere Browser unterstützen. Hierfür muss die Klasse *WebDriverFactory* implementiert werden. Zur Verwendung von Chrome ist die Umsetzung wie in Listing 1 gezeigt denkbar einfach. Über Webdriver lässt sich ebenfalls PhantomJS [4] einbinden. PhantomJS ist ein Browser ohne Oberfläche, der auf WebKit basiert und sich damit hervorragend zur Ausführung von Tests auf Serverumgebungen eignet. Die Implementierung der *WebDriverFactory* erfolgt analog zu dem vorgestellten Beispiel mit Chrome.

Mit dem Selenium-Interpreter-Parameter *driver* wird definiert, mit welchem Browser die Ausführung des Tests erfolgt. Listing 2 zeigt den Ausschnitt eines Gradle-Skripts, das Testausführungen mit mehreren Browsern ermöglicht, die nacheinander ablaufen. Hierfür werden zunächst die Abhängigkeiten zu den verschiedenen WebDriver-Implementierungen definiert. Für die beiden genannten Browserimplementierungen Chrome und PhantomJS gibt es jeweils eine eigene Gradle-Task, die mit Umgebungsvariablen den Pfad zu den Treibern festlegt. Anschließend erfolgt die Testdurchführung durch Starten des Selenium Interpreters. Die Gradle-Task *runWithAllDrivers* sorgt für den aufeinanderfolgenden Aufruf aller Browserimplementierungen.

Mit wenigen Erweiterungen kann damit für den Selenium Interpreter eine Cross-Browser-Umgebung zum automatisierten Abspielen von aufgezeichneten Selenium-Builder-Tests geschafft werden. Das vollständige Gradle-Projekt für das gezeigte Beispiel ist auf GitHub [3] verfügbar.

Verteiltes Testen mit Selenium Grid

Mit Selenium Grid lässt sich ein dynamischer Zusammenschluss aus Computerknoten einrichten, die Selenium-Testfälle von einem Hub entgegennehmen und

Artikelserie

Teil 1: Funktionale Tests mit Selenium Builder
Teil 2: Selenium parallelisieren im Grid

 © Software & Support Media GmbH

abarbeiten. Es lassen sich Testläufe auf unterschiedlichen Maschinen mit verschiedenen Browsern parallel ausführen. Für den Einsatz von Selenium Grid [5] spricht zum einen die Zeitersparnis durch parallele Abarbeitung der Testläufe. Zum anderen können Testfälle mit mehreren Browsern auf verschiedenen Betriebssystemen ausgeführt werden.

Die Anwendung steht in der Standalone-Variante als JAR-Datei zum Download bereit [6]. Im Selenium Grid gibt es die Rollen Hub und Node, die beim Kommandoaufruf vergeben werden. Ein Hub steuert jeweils ein Grid mit beliebig vielen Nodes. Der Hub hat dabei folgende Aufgaben:

- Verwaltung der angemeldeten Nodes sowie deren verfügbare Fähigkeiten, auch *Capabilities* genannt (Browsertyp, Version, Anzahl Instanzen, Protokoll)
- Entgegennehmen und Verteilen von Testläufen anhand eines *CapabilityMatchers* und der am Grid angemeldeten Nodes
- Verwaltung der lokal eingerichteten Browser, wobei das bereits vorgestellte PhantomJS oder auch HtmlUnit eingesetzt werden kann

Nodes können sich dynamisch am Grid anmelden. Sie übermitteln bei der Anmeldung an einen Hub die Fähigkeiten, die sie zur Verfügung stellen. Ein Node muss mindestens eine und kann beliebig viele Capabilities bereitstellen. Sie bestehen aus folgenden Attributen (siehe auch [6] und [7]):

- *browserName*: Pflichtattribut, das den Namen des Browsers angibt, der im Parameter *binary* referenziert wird. Der angegebene Name kann später beim Aufruf des Selenium Interpreters oder im Selenium Builder verwendet werden und muss zum eingesetzten *CapabilityMatcher* passen. Folgende Namen können vom Standard-*CapabilityMatcher* interpretiert werden:
  - android
  - chrome
  - firefox
  - htmlunit
  - iexplorer
  - iphone
  - opera
- *Version*: Optionales Attribut für die Version des Browsers. Bei Angabe sollte die evtl. vorhandene Autoupdatefunktion des Browsers deaktiviert werden. Die angegebene Versionsnummer ist unab-

Listing 1

```
package com.sebuilder.interpreter.webdriverfactory;

import java.util.HashMap;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

public class Chrome implements WebDriverFactory {
  /**
   * @param config Key/value pairs treated as required capabilities.
   * @return A ChromeDriver.
   */
  @Override
  public RemoteWebDriver make(HashMap<String, String> config) {
    return new ChromeDriver(DesiredCapabilities.chrome());
  }
}
```

Listing 2

```
...
dependencies {
  compile('com.saucelabs:sebuilder-interpreter:1.0.6')
  compile "org.seleniumhq.selenium:selenium-firefox-driver:$seleniumVersion"
  compile "org.seleniumhq.selenium:selenium-chrome-driver:$seleniumVersion"
  compile("com.codeborne:phantomjsdriver:1.2.1") {
    transitive = false
  }
}

task runWithAllDrivers {
  // The drivers we want to use
  def drivers = ["Firefox", "Chrome", "PhantomJs"]
  dependsOn drivers.collect { driver -> "runSeInterpreterWith${driver}" }
}

task(runSeInterpreterWithPhantomJs, dependsOn: ['unzipPhantomJs', 'classes'], type:
                                                                  JavaExec) {
  def phantomJsFilename = Os.isFamily(Os.FAMILY_WINDOWS) ? "phantomjs.exe" : "bin/
                                                                  phantomjs"
  systemProperty "phantomjs.binary.path",
    new File(unzipPhantomJs.outputs.files.singleFile, phantomJsFilename).absolutePath

  main = 'com.sebuilder.interpreter.SeInterpreter'
  classpath = sourceSets.main.runtimeClasspath
  args "src/main/resources/createNewsSuite.json"
  args "--driver=PhantomJs"
}

task(runSeInterpreterWithChrome, dependsOn: ['unzipChromeDriver', 'classes'], type:
                                                                  JavaExec) {
  def chromedriverFilename = Os.isFamily(Os.FAMILY_WINDOWS) ? "chromedriver.exe" :
                                                                  "chromedriver"
  systemProperty "webdriver.chrome.driver",
      new File(unzipChromeDriver.outputs.files.singleFile, chromedriverFilename).
                                                                  absolutePath

  main = 'com.sebuilder.interpreter.SeInterpreter'
  classpath = sourceSets.main.runtimeClasspath
  args "src/main/resources/createNewsSuite.json"
  args "--driver=Chrome"
}
...
```


© Software & Support Media GmbH

Selenium Server Settings

Selenium 2

Host:Port of Selenium Server	myhub:4444
Browser String	iexplorer
Browser Version	11
Platform	WINDOWS

Run Cancel

Abb. 1: Grid-Einstellungen des Selenium Builder

hängig von der tatsächlich in *binary* referenzierten ausführbaren Datei.

- *platform*: Dieses optionale Attribut beschreibt, welche Betriebssystemarchitektur der Node verwendet. Zur Auswahl stehen Windows, Linux und Mac.
- *maxInstances*: Definiert die maximale parallele Anzahl an Instanzen für den angegebenen Browser. Bei Verwendung älterer Versionen des Internet Explorers ist der Wert 1 empfehlenswert, um Probleme beim synchronen Zugriff zu vermeiden. Um die maximale Anzahl gleichzeitig laufender Sessions auf dem Node einzuschränken, steht das globale Attribut *maxSession* zur Verfügung.
- *seleniumProtocol*: Gibt das Protokoll an, das zwischen Node und Hub für die Übermittlung und Bearbeitung des Testfalls verwendet wird. Folgende Protokolle können verwendet werden:
- Selenium (bei Verwendung von Selenium-1-Skripten)
- WebDriver (Standard; bei Verwendung von Selenium-2-Skripten)
- *binary* (teilweise auch *firefox_binary* bzw. *chrome_binary*): definiert den Pfad zur ausführbaren Datei des Browsers.

Ein vom Selenium Grid entgegengenommener Auftrag besteht aus dem Skript mit dem Testfall, also beispielsweise einem JSON-Skript mit Selenium-Code oder einer Testsuite. Außerdem wird der URL und der Port des Hubs benötigt, der den Auftrag entgegennehmen soll. Die Angaben bezüglich der Eigenschaften des Zielhosts, also Browsername, Browserversion und Architektur, sind dagegen freiwillig.

Selenium Grid einrichten

Die Einrichtung des Selenium Grids setzt neben der Java-Laufzeitumgebung die Installation des Selenium-Standalone-Servers [6] voraus. Das JAR-Archiv enthält bereits alle benötigten Komponenten zum sofortigen Start. Ein funktionsfähiges Grid benötigt immer einen Hub. Entweder es wird zusätzlich mindestens ein Node definiert, oder auf dem Hub muss wenigstens ein lokaler Browser konfiguriert sein. Hub und Node können auf einem einzigen Host gestartet werden, in der Regel laufen Hub und Node jedoch auf unterschiedlichen Hosts. Die Kommunikation innerhalb des Grids, die dann über das Netzwerk stattfindet, kann über die Parameter *port* und *hubPort* des Selenium-Servers beeinflusst werden. Standardmäßig lauscht der Hub an Port 4444. Nodes öffnen standardmäßig lokal Port 5555 für die Verbindung zum Hub.

Hub und Nodes lassen sich entweder in der Kommandozeile über Parameter definieren oder mittels Konfigurationsdatei im JSON-Format. Listing 3 zeigt die Datei *hubconfig.json*, die einen Hub definiert. Der zugehörige Kommandozeilenaufruf lautet:

```
java -jar selenium-server-standalone.jar -role hub -hubConfig hubconfig.json
```

Listing 3: hubconfig.json

```
{
  "host": null,
  "port": 4444,
  "newSessionWaitTimeout": -1,
  "servlets" : [],
  "prioritizer": null,
  "capabilityMatcher":"org.openqa.grid.internal.utils.
                          DefaultCapabilityMatcher",
  "throwOnCapabilityNotPresent": true,
  "nodePolling": 5000,
  "cleanUpCycle": 5000,
  "timeout": 300000,
  "browserTimeout": 0,
  "maxSession": 5,
  "jettyMaxThreads":-1
}
```

Listing 4: nodeconfig.json

```
{
 "capabilities":
 [
  {
   "browserName": "firefox",
   "version": "35",
   "platform": "WINDOWS",
   "maxInstances": 2,
   "seleniumProtocol": "WebDriver",
   "binary": "c:\Program Files (x86)\Mozilla
Firefox\firefox.exe"
  } ,
  {
   "browserName": "iexplorer",
   "version": "11",
   "platform": "WINDOWS",
   "maxInstances": 1,
   "seleniumProtocol": "WebDriver"
  }
 ],
 "configuration":
 {
  "nodeTimeout":240,
  "nodePolling":2000,
  "maxSession": 2,
  "timeout":30000,
  "port": 5555,
  "host": mynode,
  "register": true,
  "registerCycle": 5000,
  "cleanUpCycle":2000,
  "hubPort": 4444,
  "hubHost": myhub
 }
}
```


© Software & Support Media GmbH

In Listing 4 wird die Konfigurationsdatei *nodeconfig.json* gezeigt, die einen Node definiert, der maximal zwei Instanzen von Firefox Version 35 und maximal eine Instanz von Internet Explorer 11 anbietet. Außerdem arbeitet der Node unter einer Windows-Architektur, erlaubt maximal zwei gleichzeitige Sessions und nimmt nur Testfälle mit dem Selenium-2-Protokoll entgegen. Der zugehörige Kommandozeilenaufruf lautet:

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

Abb. 2: Selenium Grid im Jenkins

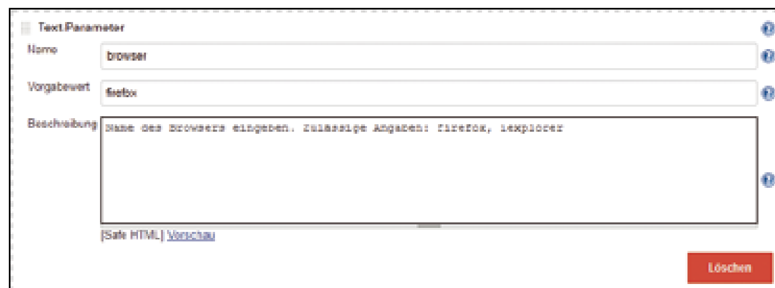

Abb. 3: Definition des Browserparameters im Jenkins

Wurden Hub und Node gestartet, steht das Grid zur Verfügung und ist bereit, Testläufe entgegenzunehmen. Zur manuellen Überprüfung der Funktionsfähigkeit des Grid oder um zu überprüfen, ob sich Testfälle auf anderen Browsern ausführen lassen, bietet sich der Selenium Builder an. Dazu wählt man im Menü Run entweder den Eintrag Run on Selenium Server (für ein einzelnes JSON-Skript) oder Run suite on Selenium Server (für eine Suite von JSON-Skripten). Im nachfolgenden Dialog gibt man die Parameter wie oben definiert ein, vgl. **Abbildung 1**.

Für die automatisierte Ausführung bietet sich die schon vorgestellte Bibliothek Selenium Interpreter an. Die Kommandozeile für das Ausführen der Testsuite aus Teil 1 des Artikels sieht damit wie folgt aus:

```
java -jar SeInterpreter.jar --driver=Remote --driver.browserName='iexplorer'
--driver.version=11 --driver.url=http://myhub:4444/wd/hub/
createNewsSuite.json
```

Selenium Grid im Zusammenspiel mit Jenkins

Wird der Continuous-Integration-Server Jenkins [8] eingesetzt, bietet sich das Selenium-Plug-in [9] an, das den CI-Server auch zu einem Hub eines Selenium Grids macht. Bei entsprechender Konfiguration verwandeln sich die Jenkins-Slaves nach dem Deployment des Plug-ins umgehend in Selenium-Knoten. Die Nodes lassen sich ebenfalls, wie oben beschrieben, manuell starten und dynamisch zu diesem Selenium Grid hinzufügen. In unserem Szenario verwenden wir als Nodes vordefinierte virtuelle Maschinen, z. B. von modern.IE [10], die per Autostart obige Kommandozeile ausführen und sich auf diese Weise beim Hub registrieren. **Abbildung 2** zeigt das Hub Management des Selenium Grids im Jenkins-Server. Am Grid hat sich ein Node registriert, der die Konfiguration aus Listing 4 nutzt.

Die Möglichkeiten, in Jenkins einen Selenium-Testfall automatisch zu starten, sind sehr umfangreich. Mit dem Build-Flow-Plug-in [11] kann beispielsweise ein Projekt definiert werden, bei dem die Applikation zuerst gebaut und gestartet wird und anschließend die Testfälle ausgeführt werden. In unserem Szenario gehen wir davon aus, dass die zu testende Applikation bereits gestartet wurde. Im Jenkins-Projekt müssen deshalb nur Selenium-Testskripte ausgeführt werden. Die Steuerung der Tests, also welches Selenium-Skript verwendet wird und mit welchem Browser es getestet wird, muss dabei über Parameter definiert werden. Dazu setzt man beim Anlegen des neuen Projekts mit dem Namen *SeleniumTest* vom Typ *Free Style* in Jenkins die Option *Dieser Build ist parametrisiert*. Als Parameter definiert man dort z. B. die vier Textparameter *browser*, *version*, *hub* und *script*, ggf. mit Vorgabewerten. In **Abbildung 3** ist die Konfiguration in Jenkins für den Parameter *browser* mit dem Vorgabewert *firefox* zu sehen.

Als Build-Verfahren wählt man *Shell ausführen* und hinterlegt die obige Kommandozeile in angepasster Form:

```
java -jar seInterpreter/SeInterpreter.jar --driver=Remote --driver.url=$hub
--driver.browserName=$browser --driver.version=$version $script
```

Mit dieser Konfiguration lassen sich, dynamisch anpassbar an die erforderliche Umgebung, unterschiedliche Selenium-Testskripte automatisch von Jenkins starten. Dazu muss in Jenkins ein Projekt vom Typ *Build Flow* angelegt werden. Der *Flow* in der Domain-specific Language (DSL) des schon angesprochenen Build-Flow-Plug-ins, kann Listing 5 entnommen werden. Er startet das zuvor erstellte Projekt *SeleniumTest* drei Mal parallel, jeweils mit den angegebenen Parametern der Selenium Nodes.

© Software & Support Media GmbH

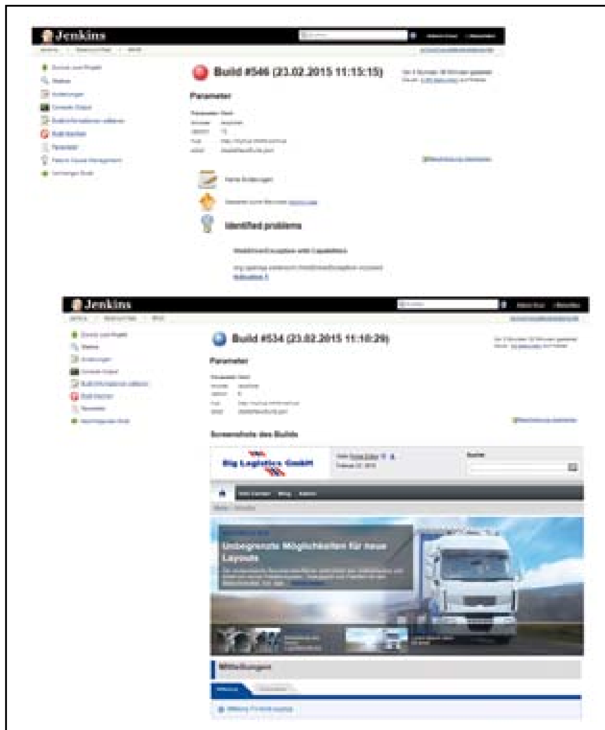

Abb. 4: Reporting in Jenkins

Reporting

Detaillierte Informationen über den Verlauf der Tests in Jenkins sind wichtig. Ein gutes Reporting benötigt aber etwas Vorarbeit. So sollten relevante Schritte des Testfalls im JSON-Skript von einem *print*-Kommando begleitet werden, das den Vorgang kommentiert. Die Ausgaben dieses Kommandos finden sich anschließend im Build-Log von Jenkins. Mit dem Kommando *saveScreenshot* erstellte Bildschirmabzüge kann man von Jenkins automatisch als Artefakte ablegen lassen. Das Image Gallery Plug-in [12] erstellt daraus eine Galerie. Um Fehler schnell erkennen zu können, bietet sich der Einsatz des Build-Failure-Analyzer-Plug-ins [13] an, dessen Fehlerdatenbank allerdings konsequent gepflegt werden muss. In **Abbildung 4** sind zwei Builds dargestellt. Beim erfolgreichen Build ist die Galerie zu sehen, beim anderen wird die Ausgabe eines Fehlers angezeigt.

Fazit

Mit dem vorgestellten Szenario lassen sich funktionale Tests im JSON-Format von Selenium automatisiert über einen CI-Server anstoßen und auf unterschiedlichen Browsern ausführen. Dabei wird sowohl von der schnellen und einfachen Aufzeichnung der Testfälle mit Selenium Builder als auch von den umfangreichen Funktionen von Jenkins profitiert. Durch den geschickten Aufbau der Projekte im Build-Server und die verwendeten Plug-ins erhält man eine Umgebung, mit der effektiv funktionale Tests parallel in unterschiedlichen Browsern ausgeführt werden können. Trotz der Komplexität des Szenarios können Änderungen an den Selenium-Tests einfach über den Builder vorgenommen werden.

Für ein gutes Reporting, vor allem im Fehlerfall, ist allerdings etwas Handarbeit erforderlich. Eine Lösung könnte die Erweiterung der Logging-Funktion des Selenium Interpreters sein.

Listing 5: Jenkins Build Flow

```
parallel (
  { build("SeleniumTest", browser: "iexplorer", version: "8", hub: "http://
                    myhub:4444/wd/hub", script: "createNewsSuite.json") },
  { build("SeleniumTest", browser: "iexplorer", version: "9", hub: "http://
                    myhub:4444/wd/hub", script: "createNewsSuite.json") },
  { build("SeleniumTest", browser: "iexplorer", version: "10", hub:
              "http://myhub:4444/wd/hub", script: "createNewsSuite.json") }
)
```

Tobias Kraft beschäftigt sich bei der exensio GmbH mit der Architektur und Umsetzung von Enterprise-Portalen, Webapplikationen und Suchtechnologien basierend auf dem Java-Stack sowie dem Grails-Framework. Des Weiteren ist er Mitorganisator des Search Meetup Karlsruhe.

tobias.kraft@exensio.de **@tokraft**

Roland Rickborn ist IT-Berater bei der exensio GmbH in Karlsruhe. Als Consultant arbeitet er in Kundenprojekten im Bereich fachliche Konzeption, Projektleitung und automatisierte Tests. Aktuell betreut er die Weiterentwicklung und Optimierung von Intranetportalen bei einem großen Kunden aus der Pharmabranche.

roland.rickborn@exensio.de **@RolandRickborn**

Links & Literatur

[1] Sauce Labs: https://saucelabs.com

[2] Selenium Interpreter: https://github.com/sebuilder/se-builder/wiki/Se-Interpreter

[3] Selenium-Interpreter-Erweiterung: https://github.com/tobiaskraft/SeInterpreterGradle

[4] PhantomJS: http://phantomjs.org

[5] Selenium Grid: http://docs.seleniumhq.org/docs/07_selenium_grid.jsp

[6] Selenium Grid: https://code.google.com/p/selenium/wiki/Grid2

[7] Selenium Grid Properties: https://code.google.com/p/selenium/source/browse/java/server/src/org/openqa/grid/common/defaults/GridParameters.properties

[8] CI-Server Jenkins: http://jenkins-ci.org

[9] Selenium-Plug-in: https://wiki.jenkins-ci.org/display/JENKINS/Selenium+Plugin

[10] modern.IE: https://www.modern.ie

[11] Build-Flow-Plug-in: https://wiki.jenkins-ci.org/display/JENKINS/Build+Flow+Plugin

[12] Image-Gallery-Plug-in: https://wiki.jenkins-ci.org/display/JENKINS/Image+Gallery+Plugin

[13] Build-Failure-Analyzer-Plug-in: https://wiki.jenkins-ci.org/display/JENKINS/Build+Failure+Analyzer

© Software & Support Media GmbH