Distributed Testing with Selenium Grid

In this chapter we will cover:

- Setting up Selenium Grid Server for parallel execution
- Adding Windows/IE node to Selenium Hub
- Adding Mac/Firefox node to Selenium Hub
- Adding iPhone/iWebDriver node to Selenium Hub
- Adding Android node to Selenium Hub
- Creating and executing Selenium script in parallel with TestNG
- Creating and executing Selenium script in parallel with Python
- Setting up Selenium with Jenkins CI for parallel execution

Introduction

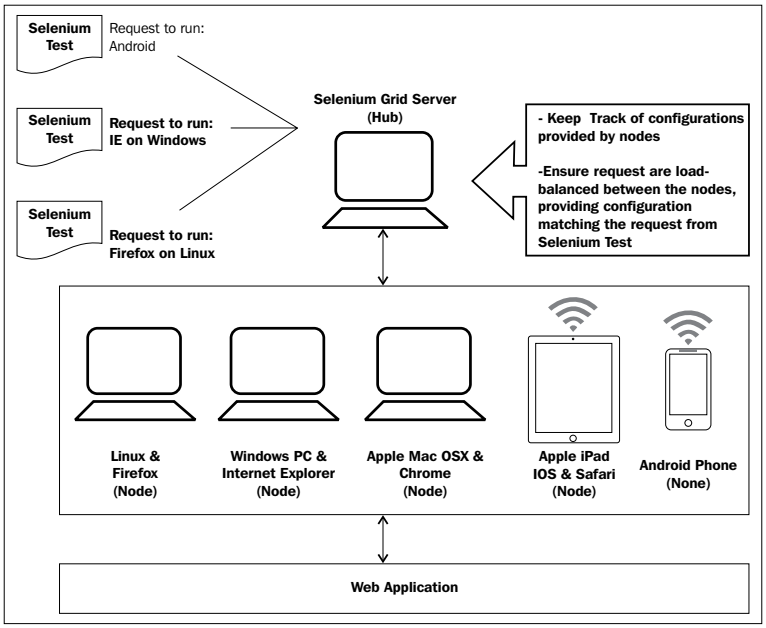
Organizations are adopting virtualization and cloud-based technologies to reduce costs and increase efficiency. Virtualization and cloud-based infrastructure can also be used to scale the test automation by reducing investment in physical hardware needed to set up the test environment. Using these technologies, web applications can be tested on a variety of browser and operating system combinations. Selenium WebDriver has unmatched support for testing applications in virtual environment, executing tests in parallel, reducing costs, and increasing speed and coverage. This chapter will cover recipes to configure and execute Selenium WebDriver tests for parallel or distributed execution.

Running tests in parallel requires two things: an infrastructure to distribute the tests and a framework that will run these tests in parallel in the given infrastructure. In this chapter, we will first create a distributed infrastructure and then create some tests, which will be executed in this distributed test environment.

Selenium Grid

Selenium Grid transparently distributes our tests across multiple physical or virtual machines so that we can run them in parallel, cutting down the time required for running tests. This dramatically speeds up testing, giving us quick and accurate feedback.

With Selenium Grid, we can leverage our existing computing infrastructure. It allows us to easily run multiple tests in parallel, on multiple nodes or clients, in a heterogeneous environment where we can have mixture of OS and Browser support. Here is an example of the Selenium Grid architecture having capabilities to run tests on Linux, Windows, Mac, iOS, and Android platforms:



Selenium Grid allows us to run multiple instances of WebDriver or Selenium Remote Control in parallel. It makes all these nodes appear as a single instance, so tests do not have to worry about the actual infrastructure. Selenium Grid cuts down on the time required to run Selenium tests to a fraction of the time that a single instance of Selenium would take to run and it is very easy to set up and use.

The selenium-server-standalone package includes the Hub, WebDriver, and Selenium RC needed to run the grid.

Testing the framework for parallel execution

In this chapter, we need a testing framework that supports parallel runs of our tests. We will use TestNG (`http://testng.org/`) for running parallel tests with Selenium WebDriver Java bindings. TestNG has a threading model to support running multiple instances of a same test using an XML-based configuration. TestNG is pretty much similar to JUnit.

We will also use a raw method to run tests in parallel for Python bindings using the subprocess module. However, you can also use `nose` for parallel execution in Python.

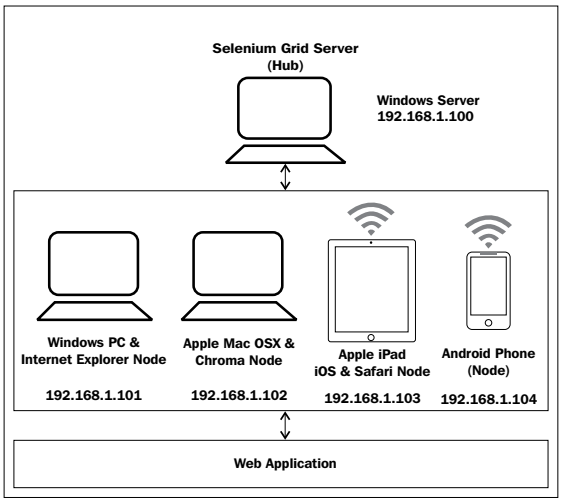
For running test in parallel with .NET bindings, you can use PUnit or MSTEST, and for Ruby, you can use DeepTest.

Setting up Selenium Grid Server for parallel execution

For running Selenium WebDriver tests in parallel, we need to set up the Selenium Grid Server as a hub. This hub will provide the available configurations or capabilities to the Selenium WebDriver tests. The slave machines, also called as **node**, connect to hub for parallel execution. Selenium WebDriver tests use the JSON wire protocol to talk to the Hub |for executing Selenium commands.

The Hub acts like the central point, which will receive the entire test request and distribute it to the right nodes.

In this recipe, we will set up a Selenium Grid Server and then add nodes with different OS and browser combinations, as shown in the following figure. We will then use this setup to run tests in parallel using TestNG.



Getting ready

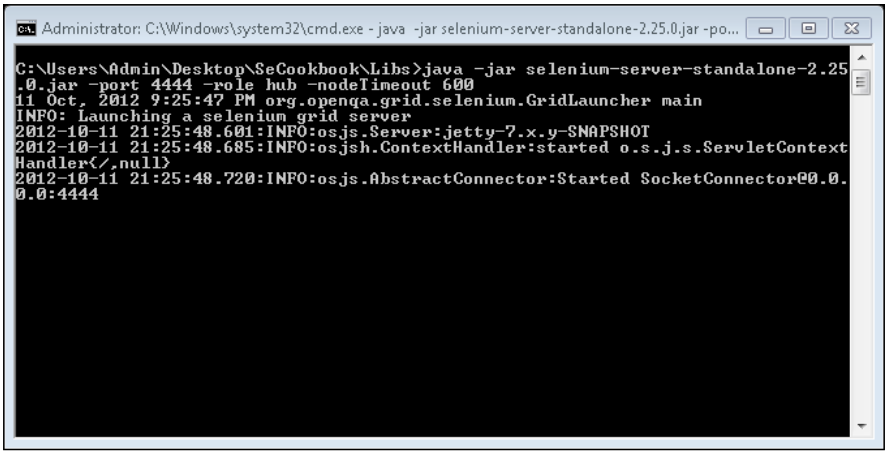
Download the latest Selenium Server standalone JAR file from `http://code.google.com/p/selenium/downloads/list`. For this recipe, selenium-server-standalone-2.25.0 version is used.

How to do it...

Setting up a Hub is a fairly simple job. Launch the Selenium Server using the following command:

```
java -jar selenium-server-standalone-2.25.0.jar -port 4444 -role hub
-nodeTimeout 600
```

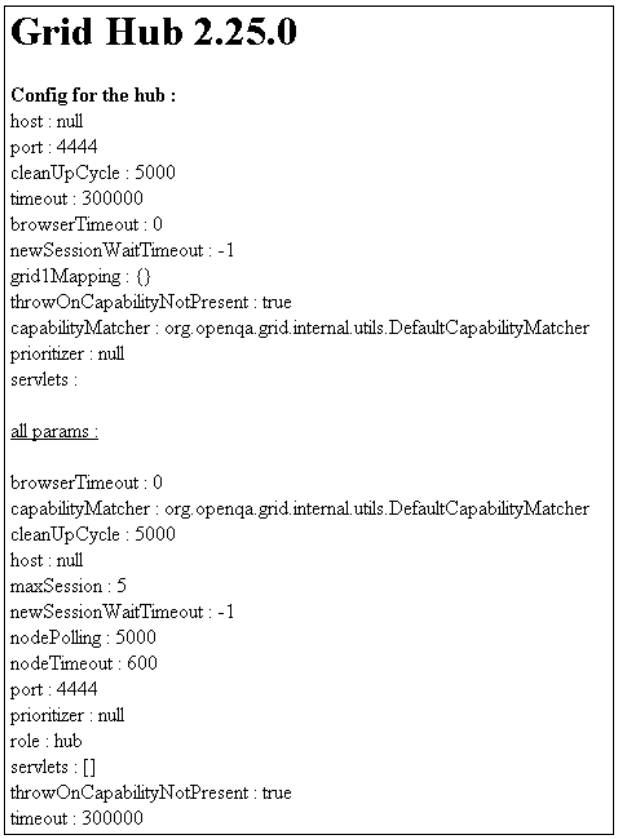
This command will start the Selenium Server in Hub role with the following output on the command prompt:



How it works...

When we launch the Selenium Standalone Server in Hub role, it starts listening to nodes and Selenium tests on port 4444. If you browse to `http://localhost:4444/grid/console` on the Hub machine, it will display the following information in the browser:

After clicking on the **view config** link, it will display the configuration details of the Hub, as shown in the following screenshot:



Grid Hub 2.25.0

Config for the hub :
host : null
port : 4444
cleanUpCycle : 5000
timeout : 300000
browserTimeout : 0
newSessionWaitTimeout : -1
grid1Mapping : {}
throwOnCapabilityNotPresent : true
capabilityMatcher : org.openqa.grid.internal.utils.DefaultCapabilityMatcher
prioritizer : null
servlets :

all params :

browserTimeout : 0
capabilityMatcher : org.openqa.grid.internal.utils.DefaultCapabilityMatcher
cleanUpCycle : 5000
host : null
maxSession : 5
newSessionWaitTimeout : -1
nodePolling : 5000
nodeTimeout : 600
port : 4444
prioritizer : null
role : hub
servlets : []
throwOnCapabilityNotPresent : true
timeout : 300000

By default Hub provides 5 sessions on the server.

There's more...

For running tests with Selenium Grid, Selenium WebDriver tests need to use the instance of the `RemoteWebDriver` and `DesiredCapabilities` classes to define which browser, version, and platform tests will be executed on. Based on preferences set in the `DesiredCapabilities` instance, the Hub will point the test to a node that matches these preferences. In this example, Hub will point the test to a node running on the Windows operating system and Firefox browser with the specified version:

```
DesiredCapabilities cap = new DesiredCapabilities();
          cap.setBrowserName("firefox");
          cap.setVersion("9.0.1");
          cap.setPlatform(org.openqa.selenium.Platform.WINDOWS);
```

These preferences are set using the `setBrowserName()`, `setVersion()`, and `setPlatform()` methods of the `DesiredCapabillities` class.

An Instance of the `DesiredCapabilities` class is passed to `RemoteWebDriver`:

```
driver = new RemoteWebDriver(new
  URL("http://192.168.1.100:4444/wd/hub"),cap);
```

In this example, the test is connecting to the Hub running on `http://192.168.1.100:4444/wd/hub` with the instance of `DesiredCapabilities` named `cap`.

Adding Windows/IE node to Selenium Hub

For running tests in parallel, we will now add nodes to the Selenium Hub that we set up in the previous recipe. It is recommended we test our web applications with major OS platforms and browsers.

With Microsoft Windows being a widely used OS along with Internet Explorer, let us add a node with Microsoft Windows and Internet Explorer capabilities to the Hub, so that tests can provision this combination and test the application.

How to do it...

We can add a Windows/Internet Explorer node to the Hub in two ways. First, supplying all the options as command line parameters described as follows:

1. To start and register a node to the Hub, use the following command by specifying browser type, version, platform, hostname, and port as arguments:

   ```
   java -jar selenium-server-standalone-2.25.0.jar -role webdriver
   -browser "browserName=internet explorer,version=8,maxinstance=1,pl
   atform=WINDOWS" -hubHost localhost –port 8989
   ```

2. Alternatively, a node can be added by creating a configuration file in JSON format and then using the command:

   ```
   {
          "class": "org.openqa.grid.common.RegistrationRequest",
          "capabilities": [
                 {
                        "seleniumProtocol": "WebDriver",
                        "browserName": "internet explorer",
                        "version": "8",
                        "maxInstances": 1,
                        "platform" : "WINDOWS"
                 }
          ],
   ```

```
        "configuration": {
               "port": 8989,
               "register": true,
               "host": "192.168.1.100",
               "proxy": "org.openqa.grid.selenium.proxy.
                  DefaultRemoteProxy",
               "maxSession": 2,
               "hubHost": "192.168.1.100",
               "role": "webdriver",
               "registerCycle": 5000,
               "hub": "http://192.168.1.100:4444/grid/register",
               "hubPort": 4444,
               "remoteHost": "http://192.168.1.101:8989"
        }
}
```

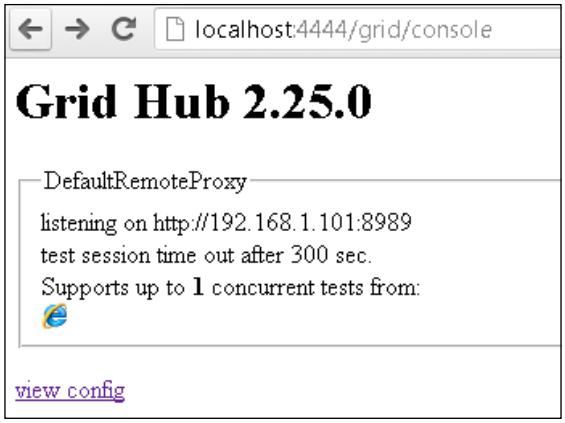
3. We can now pass the selenium-node-win-ie8.cfg.json configuration file through command-line arguments shown as follows:

```
java -jar selenium-server-standalone-2.25.0.jar -role webdriver
-nodeConfig selenium-node-win-ie8.cfg.json
```

How it works...

An instance of Selenium Standalone Server is launched with WebDriver protocol on the port specified in the configuration; in this example node, we will start running on `http://192.168.1.101:8989`.

After a successful start, the node registers itself to the Hub. If you navigate to the Grid's console on the hub, that is, `http://localhost:4444/grid/console`, you will see Internet Explorer in the list of available nodes, as shown in the following screenshot:



To register a node to the Hub, we need to supply browser details such as name of the browser, version, OS platform, port of the node machine, and so on.

When the node is launched, it connects the Hub on port 4444 providing the capabilities based on the configuration. For Internet Explorer, restrict the `maxInstance` property to `1` only.

There's more...

In this example, we registered the node supporting the WebDriver protocol by using the `role` parameter. We can also register this node to the Hub as a legacy Selenium RC node by specifying the value `node` for the `role` parameter:

```
java -jar selenium-server-standalone-2.25.0.jar -role node -browser
"browserName=internet explorer,version=8,maxinstance=1,platform=WINDO
WS" -hubHost localhost –port 8989
```

Adding Mac/Firefox node to Selenium Hub

Similar to the Windows and IE mix, we will add a node combination by using Mac OS X and Firefox and register this to the Hub.

The steps are pretty much similar to what we did in the previous recipe.

How to do it...

Similar to Windows and Internet Explorer, a node for Mac OS X and Firefox can be added to the Hub in two ways. First, supplying all the options as command line parameters as described in the following steps:

1. To start and register the node to the Hub, use the following command-line options:

   ```
   java -jar selenium-server-standalone-2.25.0.jar -role webdriver
   -browser "browserName=firefox,version=9.0.1,maxinstance=1,platform
   =MAC" -hubHost localhost –port 8989
   ```

2. In another way a node can be added by creating a configuration file in JSON format and then using the command:

   ```
   {
          "class": "org.openqa.grid.common.RegistrationRequest",
          "capabilities": [
                 {
                        "seleniumProtocol": "WebDriver",
                        "browserName": "firefox",
                        "version": "9.0.1",
   ```

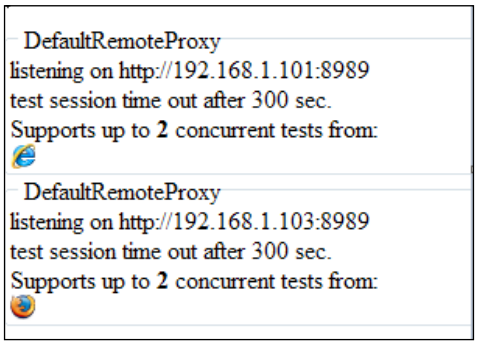
```
                "maxInstances": 1,
                "platform" : "MAC"
            }

        ],
        "configuration": {
            "port": 8989,
            "register": true,
            "host": "192.168.1.100",
            "proxy": "org.openqa.grid.selenium.proxy.
              DefaultRemoteProxy",
            "maxSession": 2,
            "hubHost": "192.168.1.100",
            "role": "webdriver",
            "registerCycle": 5000,
            "hub": "http://192.168.1.100:4444/grid/register",
            "hubPort": 4444,
            "remoteHost": "http://192.168.1.102:8989"
        }
}
```

3. We can now pass the `selenium-node-mac-ff901.cfg.json` configuration file through the following command arguments:

```
java -jar selenium-server-standalone-2.25.0.jar -role webdriver
-nodeConfig selenium-node-mac-ff901.cfg.json
```

How it works...

After a successful start, the node registers itself to the Hub. Navigate to Grid's console on the Hub. An entry for Firefox will appear in the list of available nodes along with the Internet Explorer node added in the previous recipe:



Adding iPhone/iWebDriver node to Selenium Hub

We added nodes for desktop OS and browser combinations. There might be a need to test the application on mobile platforms such as Apple iOS and Android. In this recipe, let's add a node for the iOS Safari browser running on an Apple iPad device.

Let's start this by creating a node for the iOS platform (iPhone or iPad). We can simply add an iOS device or simulator (iPhone or iPad) to the Selenium Grid server by installing the iWebDriver application.

In this recipe, we will add an iPad to the Hub set up in the first recipe.

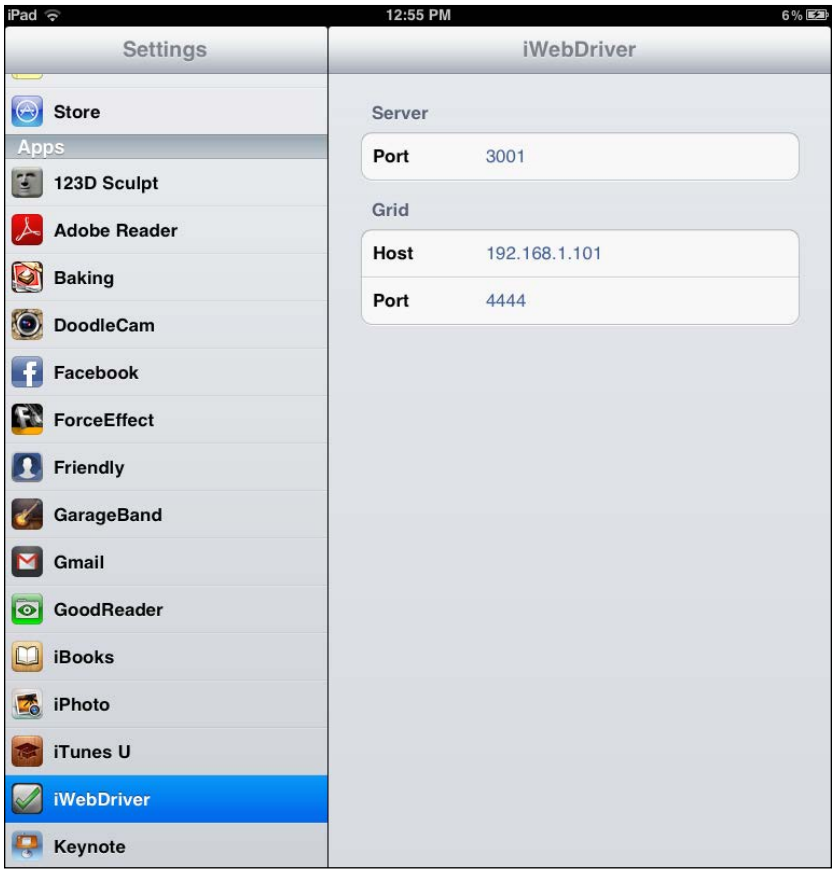
Getting ready

We need to install the iWebDriver application on the iOS device or simulator. See *Chapter 7*, *Testing on Mobile Browsers*, for the steps to install the **iWebDriver** application on an iOS device or simulator.

How to do it...

Setting up an iOS device or a simulator on a Hub is much easier compared to setting up other types of nodes:

1. Close the **iWebDriver** application if it's running.
2. Go to the iOS device or simulator **Settings** from the **Home Screen**.
3. Select **iWebDriver**. The settings for **iWebDriver** will be displayed on the screen.
4. In the **Grid** section, enter the **Host** and **Port** details of the Hub we already set in the first recipe, as shown in the following screenshot:



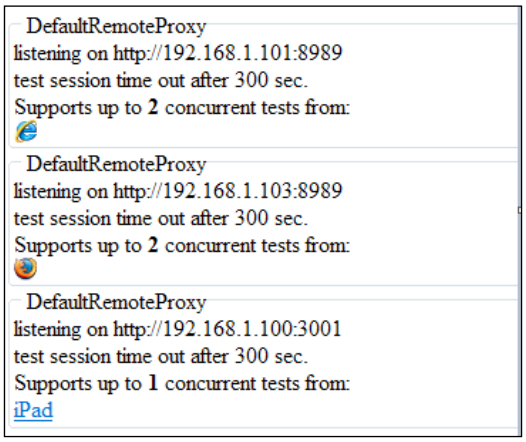
5. Start the **iWebDriver** application.

How it works...

Setting up an iOS device on the simulator on Hub does not require any configuration file or command-line options. It can be set very easily from the settings UI in iOS.

When we enter the **Host** and **Port** details for Hub and start the **iWebDriver** application, it automatically registers itself on the Hub providing the capability to run tests on the iOS Safari browser.

You can navigate to Hub console to check whether iOS device or simulator is added as a node. In the following screenshot, iPad is added to the list of available nodes in the Grid console:



See also

- The *Setting up the iWebDriver App for the iPhone/iPad simulator* recipe in *Chapter 7*, *Testing on Mobile Browsers*
- The *Setting up the iWebDriver App for an iPhone/iPad device* recipe in *Chapter 7*, *Testing on Mobile Browsers*

Adding Android node to Selenium Hub

Now let's also add an Android device or AVD to the Hub for testing applications on the Android browser.

For adding an Android node to the Hub, we need to install a utility written in Python, which will help us to register the Android node to the Hub.

In this recipe, we will add an Android emulator to the Selenium Hub.

Getting ready

We need to install and set up WebDriver APK on an Android device or emulator. See *Chapter 7*, *Testing on Mobile Browsers*, for the steps to install and set up WebDriver APK on an Android device or emulator.

This recipe will also need Python installed on the machine.

How to do it...

To register the Android node on the Hub, we need to install the flynnid (`https://github.com/davehunt/flynnid`) utility in Python.

1. Use the following command to install flynnid in Python:

   ```
   pip install flynnid
   ```

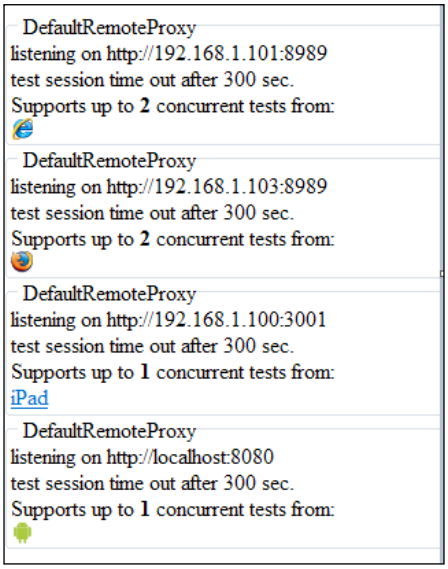
2. Now, let's register the Android node by calling flynnid from the command prompt with the following options:

   ```
   flynnid --nodeport=8080 --browsername=android --browserver=2.20
   --platform=ANDROID
   ```

How it works...

The **flynnid** utility registers the Android node by using configurations provided as command arguments. We need to supply the port of the Android node, browser name, platform Android and version of the Android Server APK installed on the Android device or emulator.

Once the node is registered successfully on the Hub, the Android node will be listed on the Grid console with a tiny Android icon, shown as follows:



See also

- The *Setting up the Android emulator for Selenium* recipe in *Chapter 7*, *Testing on Mobile Browsers*
- The *Setting up the Android device for Selenium* recipe in *Chapter 7*, *Testing on Mobile Browsers*

Creating and executing Selenium script in parallel with TestNG

Now that we have our Selenium Grid Hub and nodes ready and configured from previous recipes, it's time to run tests in parallel with this distributed environment.

For running tests in parallel, we need a test framework that supports distributing tests on these nodes. We will use TestNG for running tests in parallel for Selenium tests developed with Java bindings.

TestNG (`http://testng.org`) is another popular Unit testing framework used by the Java community after JUnit. TestNG allows us to create tests that can be executed in parallel. We can set up a group of tests or test suites and have parameters configured to run these tests in parallel. It uses a multithreading model to run tests in parallel.

TestNG requires that we have an XML configuration file with details of configurations that are needed to run the tests in parallel. Before starting the run, TestNG reads this configuration file to setup the test.

In this recipe, we use TestNG to create tests that are parameterized for parallel runs. This recipe is divided in two parts; in the first part we will create a configuration file based on the Hub and the nodes that we have configured so far and then in the second part we will be creating a test case using TestNG annotations and parameterization features.

Getting ready

Make sure you have TestNG installed on the machine from where the tests will be launched. You can download TestNG from `http://testng.org/doc/index.html`.

How to do it...

Let's create and execute a test in parallel with the following steps:

1. Before we create a test for parallel execution, first we need to create a configuration file, which TestNG will need to run the tests in parallel. We will provide four parameters for tests, platform, browser, version, and URL.

Create a new `conf.xml` file in your projects source directory and add the following code to this file:

```
<!DOCTYPE suite SYSTEM http://testng.org/testng-1.0.dtd>
<suite name="Parallel Tests" verbose="1" thread-count="4"
parallel="tests">
</suite>
```

2. Add the `<test>` node for a test that will be executed on the Windows and Internet Explorer combination in the `conf.xml` file as follows:

```
<tests>
<test name="Windows+IE9 Test">
            <parameters>
                  <parameter name="platform" value="Windows" />
                  <parameter name="browser" value="Internet
                    Explorer" />
                  <parameter name="version" value="8" />
                  <parameter name="url" value="http://
                 dl.dropbox.com/u/55228056/bmicalculator.html"/>
            </parameters>
         <classes>
            <class name="SeGridTest" />
         </classes>
</test>
</tests>
```

3. Add the following combinations to `conf.xml`. It should have the following entries:

```
<suite name="Parallel Tests" verbose="1" thread-count="4"
parallel="tests">
      <tests>
      <test name="Windows+IE9 Test">
                  <parameters>
                        <parameter name="platform"
                          value="Windows" />
                        <parameter name="browser"
                          value="Internet Explorer" />
                        <parameter name="version" value="8" />
                        <parameter name="url" value="http://
                         dl.dropbox.com/u/55228056/
                         bmicalculator.html"/>
                  </parameters>
            <classes>
                  <class name="SeGridTest" />
            </classes>
            </test>
            <test name="Mac+Firefox Test">
```

```
                        <parameters>
                              <parameter name="platform"
                                value="Mac" />
                              <parameter name="browser"
                                value="Firefox" />
                              <parameter name="version"
                                value="9.0.1" />
                              <parameter name="url" value="http://
                               dl.dropbox.com/u/55228056/
                               bmicalculator.html"/>
                        </parameters>
                  <classes>
                        <class name="SeGridTest" />
                  </classes>
                  </test>
                  <test name="iOS Test">
                        <parameters>
                              <parameter name="platform"
                                value="MAC" />
                              <parameter name="browser"
                                value="iPad" />
                              <parameter name="version" value="" />
                              <parameter name="url" value="http://
                                dl.dropbox.com/u/55228056/
                                mobilebmicalculator.html"/>
                        </parameters>
                   <classes>
                        <class name="SeGridTest" />
                  </classes>
                  </test>
                  <test name="Andorid Test">
                        <parameters>
                              <parameter name="platform"
                                value="Android" />
                              <parameter name="browser"
                                value="Android" />
                              <parameter name="version" value="2.6" />
                              <parameter name="url" value="http://
                                dl.dropbox.com/u/55228056/
                                mobilebmicalculator.html"/>
                        </parameters>
                  <classes>
                        <class name="SeGridTest" />
                  </classes>
                  </test>
            </tests>
      </suite>
```

4. Now we need to create a test using TestNG. We will create a parameterized test using TestNG's parameterization features. We will create a test for the BMI calculator application. We will use both the normal web and mobile web version of this application. Create a new test with the name as `SeGridTest` and copy the following code:

```
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.By;

import org.openqa.selenium.remote.DesiredCapabilities;
import org.openqa.selenium.remote.RemoteWebDriver;

import static org.testng.Assert.*;
import org.testng.annotations.AfterTest;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;
import org.testng.annotations.BeforeTest;

import java.net.MalformedURLException;
import java.net.URL;

public class SeGridTest {

  WebDriver driver = null;
  private StringBuffer verificationErrors = new StringBuffer();


  @Parameters({ "platform","browser","version", "url" })
  @BeforeTest(alwaysRun=true)
  public void setup(String platform, String browser, String
    version, String url) throws MalformedURLException
  {
        DesiredCapabilities caps = new DesiredCapabilities();

        //Platforms
        if(platform.equalsIgnoreCase("Windows"))
              caps.setPlatform(org.openqa.selenium.Platform.
                 WINDOWS);

        if(platform.equalsIgnoreCase("MAC"))
              caps.setPlatform(org.openqa.selenium.Platform.MAC);

        if(platform.equalsIgnoreCase("Andorid"))
```

```
                caps.setPlatform(org.openqa.selenium.
                  Platform.ANDROID);

        //Browsers
        if(browser.equalsIgnoreCase("Internet Explorer"))
               caps = DesiredCapabilities.internetExplorer();

        if(browser.equalsIgnoreCase("Firefox"))
               caps = DesiredCapabilities.firefox();

        if(browser.equalsIgnoreCase("iPad"))
               caps = DesiredCapabilities.ipad();

        if(browser.equalsIgnoreCase("Android"))
               caps = DesiredCapabilities.android();

        //Version
        caps.setVersion(version);

        driver = new RemoteWebDriver(new URL("http://
          localhost:4444/wd/hub"), caps);

        // Open the BMI Calculator Application
        driver.get(url);

}

@Test(description="Test Bmi Calculator")
public void testBmiCalculator() throws InterruptedException {

            WebElement height = driver.findElement(By.
              name("heightCMS"));
            height.sendKeys("181");

            WebElement weight = driver.findElement(By.
              name("weightKg"));
            weight.sendKeys("80");

            WebElement calculateButton = driver.findElement(By.
              id("Calculate"));
            calculateButton.click();

            try {
```

```
                        WebElement bmi = driver.findElement(By.
                          name("bmi"));
                        assertEquals(bmi.getAttribute("value"),"24.4");

                        WebElement bmi_category =
                          driver.findElement(By.name("bmi_category"));
                        assertEquals(bmi_category.getAttribute
                          ("value"),"Normal");

                  } catch (Error e) {

                        //Capture and append Exceptions/Errors
                        verificationErrors.append(e.toString());
                  }
      }
            @AfterTest
            public void afterTest() {
                  //Close the browser
                  driver.quit();

                  String verificationErrorString =
                    verificationErrors.toString();
                  if (!"".equals(verificationErrorString)) {
                        fail(verificationErrorString);
                  }

      }
    }
```

How it works...

First, we created a test configuration file listing the combination of tests that we want to run in parallel. Each instance of a test is listed in the `<test>` tag:

```
<test name="Windows+IE9 Test">
          <parameters>
                <parameter name="platform" value="Windows" />
                <parameter name="browser" value="Internet Explorer" />
                <parameter name="version" value="8" />
                <parameter name="url" value="http://dl.dropbox.
                  com/u/55228056/bmicalculator.html"/>
          </parameters>
      <classes>
```

```
            <class name="SeGridTest" />
        </classes>
    </test>
```

In the `<parameters>` section we provided platform, browser, and browser version. Last we added URL as we want to run this test both for normal and mobile web versions of the application. Next, we need to specify the name of the test class in the `<class>` node. If you are storing the test class in a package, then specify the complete path. For example, if we keep `SeGridTest` in the package `com.bmicalcapp.test` then the following will be the value that we will pass:

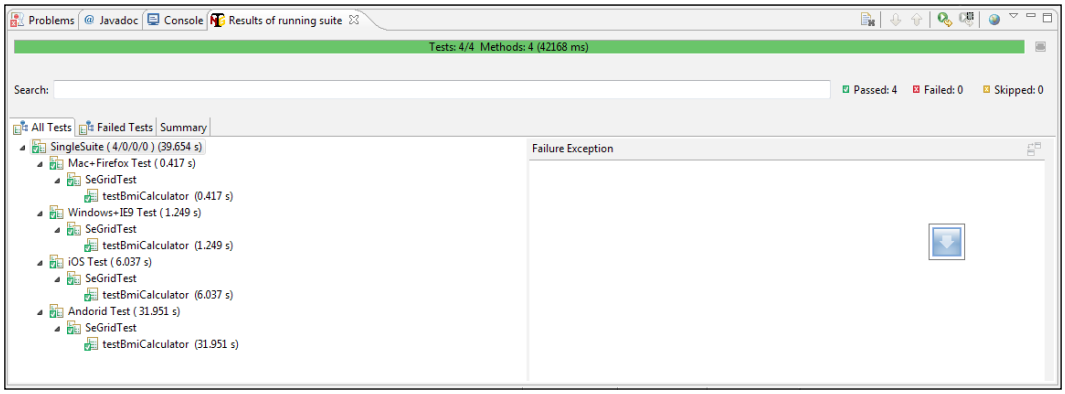
```
        <classes>
            <class name="com.bmicalcapp.test.SeGridTest" />
        </classes>
```

Next, we created a parameterized test in TestNG. We added a `setup()` method, which accepts arguments to set the options for the `DesiredCapabilities` class for running tests for each combination specified in the configuration file:

```
@Parameters({ "platform","browser","version", "url" })
@BeforeTest(alwaysRun=true)
public void setup(String platform, String browser, String version,
String url) throws MalformedURLException
{
    DesiredCapabilities caps = new DesiredCapabilities();

      //Platforms
      if(platform.equalsIgnoreCase("Windows"))
            caps.setPlatform(org.openqa.selenium.Platform.WINDOWS);

      if(platform.equalsIgnoreCase("MAC"))
            caps.setPlatform(org.openqa.selenium.Platform.MAC);

      if(platform.equalsIgnoreCase("Andorid"))
                  caps.setPlatform(org.openqa.selenium.Platform.
                    ANDROID);

      //Browsers
      if(browser.equalsIgnoreCase("Internet Explorer"))
             caps = DesiredCapabilities.internetExplorer();

      if(browser.equalsIgnoreCase("Firefox"))
             caps = DesiredCapabilities.firefox();

      if(browser.equalsIgnoreCase("iPad"))
```

```
            caps = DesiredCapabilities.ipad();

        if(browser.equalsIgnoreCase("Android"))
            caps = DesiredCapabilities.android();

        //Version
        caps.setVersion(version);

        driver = new RemoteWebDriver(new URL("http://localhost:4444/wd/
          hub"), caps);

        // Open the BMI Calculator Application
        driver.get(url);
    }
```

When we execute the tests, we need to tell TestNG to use the `conf.xml` file for test configurations. TestNG will read this file and create a pool of threads (for this example, it will create four threads) for each combination calling the test class. It will then create and assign an instance of test class to each thread by passing the parameters from the configuration file to the `setup()` method. TestNG will run the test concurrently for each combination, monitoring and reporting the test status and final results. The following is a report it generates after running the test for combinations specified in the configuration file:



See also

- The *Creating a data-driven test using TestNG* recipe in *Chapter 4*, *Data-driven Testing*

Creating and executing Selenium script in parallel with Python

As we have seen in the previous recipe, TestNG provides a very powerful execution framework for running tests in parallel for Java bindings.

However, if you are using Python or another binding to develop Selenium WebDriver tests, you can also run tests in parallel for distributed testing.

In this recipe, we will create and execute tests in parallel with Python bindings using the subprocess module. We will use the Hub and nodes configured in earlier recipes to run these tests.

How to do it...

We will create two test scripts for testing the application with Firefox and Internet Explorer using the following steps. You can also create a single test and parameterize it similar to what we did for TestNG:

1. For the first Python test, which will test the application functionality using the Firefox browser, name this test as `testOnFirefox.py` and copy the following code:

```
from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import
DesiredCapabilities

from selenium.webdriver.support.ui import WebDriverWait
import time, unittest

class OnFirefox (unittest.TestCase):
       def setUp(self)     :
              self.driver = webdriver.Remote(
              command_executor='http://localhost:4444/wd/hub',
              desired_capabilities=DesiredCapabilities.Firefox)

       def test_Google_Search_FF(self):
              driver = self.driver
              driver.get("http://www.google.com")
              inputElement = driver.find_element_by_name("q")
              inputElement.send_keys("Cheese!")
              inputElement.submit()
              WebDriverWait(driver, 20).until(lambda driver :
                driver.title.lower().startswith("cheese!"))
```

```
                self.assertEqual("cheese! - Google Search",driver.
                  title)

        def tearDown(self):
                self.driver.quit()

if __name__ == "__main__":
        unittest.main()
```

2. For the second Python test, which will test the application functionality using the Internet Explorer browser, name this test as `testOnIE.py` and copy the following code:

```
from selenium import webdriver
from selenium.webdriver.common.desired_capabilities import
DesiredCapabilities

from selenium.webdriver.support.ui import WebDriverWait
import time, unittest

class OnInternetExplorer (unittest.TestCase):
        def setUp(self)     :
                self.driver = webdriver.Remote(
                command_executor='http://localhost:4444/wd/hub',
                desired_capabilities=DesiredCapabilities.
                  INTERNETEXPLORER)

        def test_Google_Search_IE(self):
                driver = self.driver
                driver.get("http://www.google.com")
                inputElement = driver.find_element_by_name("q")
                inputElement.send_keys("Cheese!")
                inputElement.submit()
                WebDriverWait(driver, 20).until(lambda driver :
                  driver.title.lower().startswith("cheese!"))
                self.assertEqual("cheese! - Google Search",driver.
                  title)

        def tearDown(self):
                self.driver.quit()

if __name__ == "__main__":
        unittest.main()
```

3. Finally, we need to create a Python script, which will use the subprocess module to run these tests concurrently on different nodes. Name this script as `runner.py`:

```
from subprocess import Popen
import glob

tests = glob.glob('test*.py')
processes = []
for test in tests:
    processes.append(Popen('python %s' % test, shell=True))

for process in processes:
    process.wait()
```

How it works...

We need to place all three scripts in the same directory. When we execute `runner.py`, it collects all the tests with names starting as `test` using the `glob` function. Then we create a hash for processes and append each test using the `Popen` function from the subprocess module. The `Popen()` function calls each test using Python as a subprocess of the main process and waits for the script to complete.

There's more...

We can also use `nose` module for Python to run the tests in parallel. We need to install `nose` by using the following command:

```
easy_install nose==0.11 multiprocessing
```

After `nose` is installed, you need to open the folder where all the tests are stored and use the following command:

```
nosetests --processes=2
```

This will call the `nosetests` scripts, which will locate all the files with test prefix in the current directory and start running tests concurrently. In this example, we are running two tests so we need to specify the value for processes argumented as 2. `nose` module internally uses multiprocessing module for concurrent execution.

See also

- The *Creating and executing Selenium script in parallel with TestNG* recipe

Setting up Selenium with Jenkins CI for parallel execution

As continuous integration is a widely accepted practice, it is essential that we run Selenium WebDriver tests as part of the CI process for early feedback and increased confidence. Jenkins is widely used as a continuous integration server tool in Java world.

Instead of having a plain Selenium Grid, Jenkins provides a Selenium Grid that can be very well integrated with the CI infrastructure. We can use Jenkins' powerful distributed model for CI to run our Selenium tests in parallel on a Jenkins cluster. In this recipe, we will set up Jenkins for Selenium Grid.

Getting ready

Download and install Jenkins from `https://jenkins-ci.org/`.

How to do it...

We need to install **Jenkins Selenium Plugin** to add Selenium Grid support in Jenkins. Use the following steps to install and configure **Jenkins Selenium Plugin**:

1. Click **Manage Jenkins** on the Jenkins **Dashboard**.
2. Click on **Manager Plugins** from the **Manage Jenkins** option.
3. Click on the **Available** tab.
4. Locate and select **Jenkins Selenium Plugin** from the list of available plugins.
5. Click on the **Download now and install after restart** button.
6. A new screen will be displayed **Installing Plugins/Upgrades**.
7. After the plugin is downloaded, restart Jenkins. Make sure that no jobs are running while you restart.
8. After Jenkins restarts, a **Selenium Grid** link will appear on the left side navigation pane.
9. Click on **Selenium Grid**, the **Registered Remote Controls** page will be displayed.
10. **Selenium Grid** is now available on `http://localhost:4444/wd/hub` for tests.

How it works...

On Jenkins master, Selenium Grid Hub is started on port 4444. This is where all Selenium WebDriver tests should connect, same as using a normal Selenium Grid.

When we set up a Jenkins slave, necessary binaries are copied and Selenium RCs are started automatically by Jenkins. Selenium running on slaves and the Selenium Grid Hub on master are hooked up together automatically.

Along with the standard platform matching capability offered out of the box by Selenium Grid, Jenkins allows us to specify `jenkins.label` as a capability, whose value is a Boolean expression of labels/slave names to narrow down where to run the tests:

```
DesiredCapabilities cap = DesiredCapabilities.firefox();
cap.setCapability("jenkins.label","redhat5 && amd64");
WebDriver driver = new RemoteWebDriver(new
URL("http://192.168.1.100:4444/wd/hub"), cap);
```

In the previous example, Jenkins Selenium Grid will point the tests to the Jenkins slave with redhat5 and amd64 as a label set in Jenkins cluster.

This capability is automatically added to grid nodes that are launched by Jenkins.

Jenkins Selenium Grid can also accept additional nodes running on a Selenium Standalone Server launched outside of Jenkins.