

コード品質を追求する: Selenium と TestNG を使ったプログラムによるテスト

容易になった自動ユーザー受け入れテスト

Andrew Glover

2007年 4月 03日

Selenium は、Web アプリケーションでのユーザー受け入れテストを簡単に実行できるテスト用フレームワークです。Andrew Glover が今月紹介するのは、この Selenium のテストを、TestNG をテスト・ドライバースとして使い、プログラムによって実行する方法です。TestNG の柔軟なテスト機能 (パラメーターを使ったフィクスチャーを含め) を Selenium 固有のツールキットに加えれば、後は DbUnit と Cargo の助けをほんの少し借りるだけで、完全に自動化した、しかも論理的に反復可能な受け入れテストを作成できます。

Selenium は、Web アプリケーションの検証に新たな手法を確立する Web テスト用フレームワークです。たいていのテスト・ツールは HTTP 要求をシミュレートしようとはしますが、Selenium はそれ自体がブラウザーであるかのように Web テストを処理します。自動化された Selenium テストを実行すると、このフレームワークはブラウザーを起動し、ユーザーのアプリケーション操作とまったく同じようにテストに記述されたステップでブラウザーを操作します。

Selenium は開発者と開発者以外の人々の両方にとってテストが作成しやすいという点でも、多くのアプリケーション・テスト用フレームワークと一線を画します。Selenium でテストを作成するには、プログラムによる方法、あるいは Fit 形式のテーブルを使用する方法を選べます。いずれの方法で作成しても、仕上がったテストは完全に自動化できます。Ant ビルド (一例として) を使って Selenium テスト・スイート全体を実行するのも簡単で、Selenium テストを継続的インテグレーション (CI) 環境で実行することも可能です。

今月は、この Selenium を紹介するとともに、特に TestNG、DbUnit、そして Cargo などと組み合わせさせて Selenium を非常に優れた Web テスト用フレームワークにしている機能を説明します。

受け入れテスト

Selenium はユーザーの行動をモデル化するのに非常に優れているため、通常は受け入れテストに関連付けられています。受け入れテストとは、完成したシステムで実行する一連のテストのことです。受け入れテストを効果的なものにするためには通常、アプリケーション全体を実行中の状態にする必要があります。例えば Web アプリケーションをテストする場合には、アプリケーションのデータベースだけでなく、Web サーバー、コンテナ、そしてアプ

リケーションを実行するために必要なあらゆる構成要素にアクセスできるようにしなければなりません。

Selenium を使ったプログラムによるテスト

Selenium では、お好みの言語を使ってプログラムによるテストを作成することも、Fit 形式のテーブルによってテストを作成することもできます。どちらの作成方法にしても、テストの観点からはプロセスと結果に大した違いはありませんが、この記事ではプログラムによる方法について取り上げます。Selenium を TestNG と組み合わせると、興味深い可能性が生まれてくるからです。

Selenium を TestNG のようなフレームワークと一緒にプログラムで使用する場合の利点としては、インテリジェントなフィクスチャーを作成できるという点が挙げられます。同じことを Fit 形式のテーブルで行おうとしても、そう簡単にはいきません。TestNG がとりわけ Selenium と相性が高い理由は、依存関係を使ったテスト、失敗したテストの再実行、そして別のファイルに定義されたパラメーターを使ったパラメーター・テストの設定など、他のフレームワークでは不可能なことも TestNG では可能だからです。Web アプリケーションのテスト用フレームワークとしては、このような機能をすべて兼ね備えているだけでも注目に値することですが、これから説明するように、これだけの機能を完全に自動化された受け入れテストで使用できるとなると群を抜いた強みになります。

最初のテストを構成する

Selenium のアーキテクチャーは基本的に 2 つの論理エンティティーで構成されます。1 つは作成するテスト・コード、もう 1 つはテスト対象のアプリケーションの操作を容易にする Selenium サーバーです。テストを正常に実行するには、Selenium サーバーのインスタンスとテスト対象のアプリケーションを両方用意して実行する必要があります (テストの結果は、もちろん作成したアプリケーションの出来具合によります)。

幸いなことに、Selenium サーバーは軽量なプロセスなので、実際のテストの制約内でプログラムによって起動、停止することができます。Selenium サーバー (Selenium オブジェクトによって実現) の起動と停止は、フィクスチャーの役目となります。

Selenium サーバーをプログラムによって起動するには、新規 Selenium オブジェクトを作成し、どの互換ブラウザを使用するかを指定します (以下の例で使用するのは Firefox です)。また、サーバー・インスタンスが実行される場所 (一般的には localhost ですが、必須ではありません)、テスト対象アプリケーションの基本 URL を指定する必要があります。

リスト 1 では、ローカルにインストールした Web アプリケーション (<http://localhost:8080/gt15/>) で Firefox を駆動するように、Selenium のローカル・インスタンスを構成しています。引数から想像がつかかもしれませんが、Selenium オブジェクトはテスト対象アプリケーションに対してはプロキシとして機能するため、容易にテストをすることができます。

リスト 1. SeleniumServer の構成

```
Selenium driver =  
    new DefaultSelenium("localhost", SeleniumServer.getDefaultPort(),  
        "firefox", "http://localhost:8080/gt15/");  
  
driver.start();  
//go to web pages and do stuff...  
driver.stop();
```

Selenium のインスタンスを作成すると、実行中にそのインスタンスを開始、停止できるようになります。つまり、Selenium サーバーをプログラムで操作し、テスト・プロシーチャーに従ってサーバーにブラウザーを制御させることができるようになるということです。

アプリケーションの操作

Web ページをプログラムで操作するということは、すなわち論理 ID の使用を実践するということです (TestNG-Abbot を取り上げた 2 月の記事を読んだ読者には、お馴染みのコンセプトです)。ページ要素を操作するための最初のステップは、その要素を見つけることです。それには通常、HTML 要素 ID を使いますが、Selenium ではお望みとあれば、XPath や正規表現、さらには JavaScript を使って特定の要素を見つけることもできます。

リスト 2 に記載する HTML は、Groovlet を使用する単純な Web アプリケーションからの抜粋です。このコードは、1 つの入力とサブミット・ボタンが含まれるフォームを定義しています。Selenium にこのフォームを操作させるには、入力に対する ID とそれに対応する値を指定します。また、サブミット・ボタンの ID も指定して、Selenium がボタンを「クリック」できるようにする必要があります。ボタンをクリックすると、フォームは Groovlet (この例では FindWidget.groovy) にサブミットされます。

リスト 2. 単純な HTML フォーム

```
<form method=post action="./FindWidget.groovy">
<table border="0" style="border-style: dotted">
  <tr>
    <td class="heading">Widget:</td>
    <td class="value"><input type="text" name="widget"></td>
  </tr>
  <tr>
    <td></td>
    <td class="value"><input type="submit" value="Find Description" name="submit"></td>
  </tr>
</table>
</form>
```

これで、ID の widget (値の入力用) と submit (ボタンのクリック用) を使用して、HTML フォームをプログラムで操作できるようになります (リスト 3 を参照)。

リスト 3. 単純な Web ページの操作

```
driver.type("widget", "pg98-01");
driver.click("submit");
driver.waitForPageToLoad("10000");
//assert some return value...
```

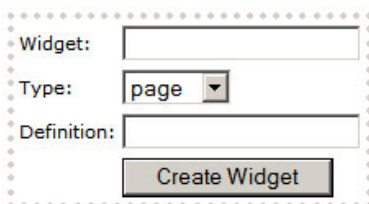
Web ページ要素を操作するための Selenium の API はかなり直観的なものです。入力フィールドには、type() メソッドを使って値を ID に関連付ければ、好きなときにプログラムでボタンを click できるようになります。リスト 3 では、Selenium が 10 秒間待機するようにしています。10 秒あれば、フォーム・サブミット要求の処理は十分完了します。FindWidget.groovy 内のコードがその過程を実行して応答を返すと、その応答を使用して特定のページ要素を検出し、すべてが正常に機能したことを確認できます。

Selenium と TestNG

TestNG はその柔軟なパラメーターを使ったフィクスチャーにより、Selenium による受け入れテストを定義する際に大いに活躍します。TestNG のテスト依存関係を定義する機能、そして失敗したテストを再実行する機能を追加すれば、考えるまでもなく Selenium と TestNG は最強の組み合わせになります。

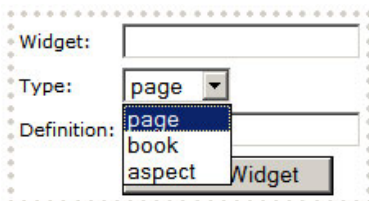
まずは、ユーザーがウィジェットを作成、検索、更新、削除できる Web アプリケーションから取り掛かることにしましょう。ウィジェットを作成するには、名前、型、そして定義という 3 つの属性が必要です。図 1 に、ウィジェットを作成するためのフォームを示します。

図 1. ウィジェット作成の Web フォーム



フォームの Type 要素はドロップダウン・リストで、このリストには 3 つの選択項目が提示されます (図 2 を参照)。

図2. ドロップダウン・リストが含まれる Web フォーム



Create Widget をクリックすると、Groovlet がその要求を処理します。すべてが正しければ (つまり、名前と定義が空白でなく、インスタンスがすでにデータベースにある場合)、Groovlet は新しいウィジェット・インスタンスを作成して、図 3 のようなステータス・ページを返します。

図 3. 返された Web ページに表示されたステータス



Selenium を TestNG と併用すると、単純な Create Widget の使用事例は以下のような扱いやすいステップで検証できます。

1. Selenium サーバーのインスタンスを構成して起動する。
2. Create Widget の Web フォームを操作してサブミットする。
3. 結果のページに、ウィジェットの名前を示す成功メッセージが含まれていることを確認する。
4. Selenium サーバー・インスタンスを停止する。

上記の使用事例の各ステップは、Selenium によって行われることに注意してください。TestNG はいわゆるパイプ役でしかありません。それでは早速、Create Widget のテスト・ケースを試してみましょう。

Create Widget のテスト・ケース

Selenium サーバーの構成には柔軟性を持たせたいので、パラメーター化したフィクスチャー (TestNG-Selenium 形式) を作成し、このフィクスチャーを使用して、さまざまなブラウザー、さまざまなロケーション、さらには多種多様な Web アプリケーション・アドレス (いくつか例を挙げると localhost、実動アプリケーションなど) に対して Selenium サーバーを汎用的に作成できるようにします。この柔軟な Selenium サーバー・フィクスチャーを定義しているのが、リスト 4 です。

リスト 4. 柔軟な Selenium フィクスチャー

```
@Parameters({"selen-svr-addr", "brwsr-path", "aut-addr"})
@BeforeClass
private void init(String selenSvrAddr, String bpath,
    String appPath) throws Exception {
    driver = new DefaultSelenium(selenSvrAddr,
        SeleniumServer.getDefaultPort(), bpath, appPath);
    driver.start();
}
//....
@AfterClass
private void stop() throws Exception {
    driver.stop();
}
```

TestNG の testng.xml ファイルでパラメーター名を値にリンクし、最終的にはリスト 5 に示す 3 つのパラメーター要素を定義します (brwsr-path パラメーターは Firefox にデフォルト設定されるようにしていますが、Internet Explorer を使った新しいテスト・セットを定義するのも同じく簡単です)。

リスト 5. TestNG の testng.xml ファイルでのパラメーター値

```
<parameter name="selen-svr-addr" value="localhost"/>
<parameter name="aut-addr" value="http://localhost:8080/gt15/" />
<parameter name="brwsr-path" value="*firefox"/>
```

次に、リスト 6 のテスト・ケースを定義します。ここでも、テスト対象アプリケーションの基本 URL にはパラメーターを使います。このテストでは、ブラウザーに Web アプリケーション内の特定のページを表示させ、[図 1](#) に示したフォームを操作します。

リスト 6. 正常な場合のテスト・ケース

```
@Parameters({"aut-addr"})
@Test
public void verifyCreate(String appPath) throws Exception {
    driver.open(appPath + "/CreateWidget.html");
    driver.type("widget", "book-01");
    driver.select("type", "book");
    driver.type("definition", "book widget type book");
    driver.click("submit");

    driver.waitForPageToLoad("10000");
    assertEquals(driver.getText("success"),
        "The widget book-01 was successfully created.",
        "test didn't return expected message");
}
```

driver.click("submit") 呼び出しでフォームがサブミットされたら、応答がロードされるまで Selenium を待機させて、それから作成が成功したというメッセージがあることを表明します (応答 Web ページには ID が success となっている要素があることに注意してください)。

以上を組み立てると、2 つのシナリオを検証する柔軟なテスト・クラスになります。検証する 2 つのシナリオは、一方はすべてが順調に運ぶケース、そしてもう一方は定義の指定がないという例外ケースです (リスト 7 を参照)。

リスト 7. TestNG による全体的処理

```
public class CreateWidgetUATest {
    private Selenium driver;

    @Parameters({"selen-svr-addr", "brwsr-path", "aut-addr"})
    @BeforeClass
    private void init(String selenSvrAddr, String bpath,
        String appPath) throws Exception {
        driver = new DefaultSelenium(selenSvrAddr,
            SeleniumServer.getDefaultPort(), bpath, appPath);
        driver.start();
    }

    @Parameters({"aut-addr"})
    @Test
    public void verifyCreate(String appPath) throws Exception {
        driver.open(appPath + "/CreateWidget.html");
        driver.type("widget", "book-01");
        driver.select("type", "book");
        driver.type("definition", "book widget type book");
        driver.click("submit");

        driver.waitForPageToLoad("10000");
        assertEquals(driver.getText("success"),
            "The widget book-01 was successfully created.",
            "test didn't return expected message");
    }

    @Parameters({"aut-addr"})
    @Test
    public void verifyCreationError(String appPath) throws Exception {
        driver.open(appPath + "/CreateWidget.html");
        driver.type("widget", "book-02");
        driver.select("type", "book");
        //definition explicitly set to blank
        driver.type("definition", "");
        driver.click("submit");
    }
}
```



```
driver.waitForPageToLoad("10000");
assertEquals(driver.getText("failure"),
    "There was an error in creating the widget.",
    "test didn't return expected message");
}

@AfterClass
private void stop() throws Exception {
    driver.stop();
}
}
```

ここまでのところで、複数のブラウザーや複数のロケーションを使ったテストを簡単に行えるだけの柔軟性を持つ 2 つの Selenium のテストを定義しました。初心者には上出来です。ですがもう少し踏み込んで、テストのロジックが反復可能かどうかを考えてみたいと思います。例えば、CreateWidgetUATest テストを 2 回連続で実行したら、どうなるでしょう。その場合、ローカル・マシン、あるいはどのマシンでも Web アプリケーションが確実に最新バージョンのコードで実行されるようにするには、どうしたらいいのでしょうか。

反復可能な受け入れテスト

Selenium のテストを実行するときには、検証対象の Selenium サーバーと Web アプリケーションが両方とも実行中でなければなりません。そこには、アプリケーションのアーキテクチャーに関連するすべての依存関係も実行中でなければならないという意味も含まれます。たいていの Java™ Web アプリケーションでは、Servlet コンテナと関連データベースがそれに相当します。

[反復可能なシステム・テスト](#)に関する記事で説明したように、データベースに依存する Web アプリケーションに論理的再現性を実装する技術として私が気に入っているのは DbUnit と Cargo の 2 つです。DbUnit はデータベース内のデータを管理し、Cargo は一般的な方法でコンテナ管理を自動化します。以降のセクションでは、この 2 つの技術が Selenium と TestNG と連携して、受け入れテストの論理的再現性を確実にする仕組みを説明します。

DbUnit によるテスト・ケースの再現

記憶に残っているかもしれませんが、DbUnit はテスト・シナリオのコンテキスト内でデータベースのデータを効果的に管理することにより、データベースの処理を容易にします。DbUnit を使用すれば、既知のデータ・セットをテスト前にデータベースにロードできます。つまり、テストの間中、存在しているそのデータに依存できるということです。さらに、テストが終わったら、テストの結果として作成されたデータをデータベースから削除することもできます。このような作業はすべて、DbUnit が便利なフィクスチャーとして機能することにより容易になります。JUnit にせよ TestNG にせよ、DbUnit は、テスト・データが含まれるシード・ファイルを読み取って、対応するデータベース・テーブルに対してデータの論理的挿入、削除、更新を行うというフィクスチャーとして動作するのです。

Selenium を制御するために私が使っているのは TestNG なので、DbUnit フィクスチャーを作成して、これをテスト・レベルで実行することにします。TestNG では、5 段階の粒度でフィクスチャーを実行できます。下位の 2 段階はメソッドとクラスです。当然この 2 つは、それぞれ各テスト・メソッドごとのフィクスチャー、クラス全体のフィクスチャーに相当します。その他に TestNG が定義するのは、(TestNG 構成ファイル内で定義され、test 要素によって指定される) テス

トの集合に対するフィクスチャー、(テストの集合からなる集合で、suite 要素によって指定される) テスト・スイート全体に対するフィクスチャー、そして (TestNG の Test 注釈内で定義される) テストのグループに対するフィクスチャーです。

テストの詳細

テスト・レベルで実行する DbUnit フィクスチャーを作成するということは、テスト・クラスの集合が同じロジックを共有して、テストが実行される前にデータベースに適切にデータを供給するということです。この例では、論理テストの集合が実行される前に、データベースにクリーンなデータ・セットを持たせることにします。DbUnit の CLEAN_INSERT コマンドを使用すれば、以前のテスト実行時に作成された行が確実に削除されるため、データベースの制約を心配せずに、データを繰り返し作成するテストを再実行することができます。

さらに、フィクスチャーをパラメーターのデータに依存させるようにする必要もあります。これにより、特定のテストを実行する前に、シード・ファイルだけでなく特定のデータベースのロケーションを切り替えられるという柔軟性がもたらされます。パラメーターを TestNG に関連付けるのはこれ以上なく簡単で、フィクスチャーを Parameters 注釈で修飾し、対応するパラメーターをメソッド・シグニチャーで宣言して TestNG の構成ファイルに値を指定するだけのことです。

リスト 8 では、データベースに必要なシード・ファイルを供給する単純な DbUnit フィクスチャーを定義しています。このフィクスチャーでは 5 つのパラメーターを使用していることに注目してください (多過ぎるかもしれませんが、フィクスチャーがパラメーターを持つのは賢いことだと思いますか)。

リスト 8. テストの集合に対する DbUnit フィクスチャー

```
public class DatabaseFixture {

    @Parameters({"seed-path", "db-driver", "db-url", "db-user", "db-psswrд"})
    @BeforeTest
    public void seedDatabase(String seedpath, String driver,
        String url, String user, String pssword) throws Exception {

        IDatabaseConnection conn = this.getConnection(driver, url, user, pssword);
        IDataset data = this.getDataSet(seedpath);

        try {
            DatabaseOperation.CLEAN_INSERT.execute(conn, data);
        } finally {
            conn.close();
        }
    }

    private IDataset getDataSet(String path) throws IOException, DataSetException {
        return new FlatXmlDataSet(new File(path));
    }

    private IDatabaseConnection getConnection(String driver,
        String url, String user, String pssword ) throws ClassNotFoundException,
        SQLException {
        Class.forName(driver);
        Connection jdbcConnection =
            DriverManager.getConnection(url, user, pssword);
        return new DatabaseConnection(jdbcConnection);
    }
}
```


実際の値をリスト 8 のパラメーターに関連付けるには、値を TestNG の testng.xml ファイルに定義します (リスト 9 を参照)。

リスト 9. TestNG の testng.xml ファイルに定義した DbUnit 固有のパラメーター

```
<parameter name="seed-path" value="test/conf/gt15-seed.xml"/>
<parameter name="db-driver" value="org.hsqldb.jdbcDriver"/>
<parameter name="db-url" value="jdbc:hsqldb:hsqldb://127.0.0.1"/>
<parameter name="db-user" value="sa"/>
<parameter name="db-psswr" value=""/>
```

汎用パラメーター値

データベースの状態、そして対応するいくつかのテストを処理する柔軟なフィクスチャーを定義したので、いよいよ TestNG を使ってすべてをつなぎ合わせます。その第 1 のステップは、例によって何が目的なのかを把握することです。この例で達成したい目的は、以下のとおりです。

- テストの論理的集合を実行する前に、DbUnit フィクスチャーがその役目を果たすようにする。
- 同じひとまとまりのテストを 2 回実行する (Firefox に対して 1 回、Internet Explorer に対して 1 回)。

TestNG の parameter 要素のスコープがローカルであることは有利に働きます。なぜなら、汎用パラメーター値を TestNG 構成ファイル内に容易に定義することができ、必要に応じて、それらの値で TestNG の test グループ化要素内の値を書き換えることができるからです。

例えば 2 つのテスト・セットを実行するには、単純に 2 つの test 要素を作成します。それから TestNG の package 要素を使用してフィクスチャーと関連テストを組み込むと、パッケージ構造内のすべてのテスト (フィクスチャー) を検出しやすくなります。次に、定義した 2 つの test グループの Firefox と Internet Explorer に対応する brwsr-path パラメーターを関連付けます。この作業をすべて示しているのが、リスト 10 の testng.xml です。

リスト 10. DbUnit を実行させる柔軟な testng.xml ファイル

```
<suite name="User Acceptance Tests" verbose="1" >

  <!-- required for DbUnit fixture -->
  <parameter name="seed-path" value="test/conf/gt15-seed.xml"/>
  <parameter name="db-driver" value="org.hsqldb.jdbcDriver"/>
  <parameter name="db-url" value="jdbc:hsqldb:hsqldb://127.0.0.1"/>
  <parameter name="db-user" value="sa"/>
  <parameter name="db-psswr" value=""/>

  <!-- required for Selenium fixture -->
  <parameter name="selen-svr-addr" value="localhost"/>
  <parameter name="aut-addr" value="http://localhost:8080/gt15/">

  <test name="GT15 CRUDs- Firefox" >

    <parameter name="brwsr-path" value="*firefox"/>

    <packages>
      <package name="test.com.acme.gt15.Web.selenium" />
      <package name="test.com.acme.gt15.Web.selenium.fixtures" />
    </packages>
  </test>

  <test name="GT15 CRUDs- IE" >
```

```
<parameter name="brwsr-path" value="*iexplore"/>

<packages>
  <package name="test.com.acme.gt15.Web.selenium" />
  <package name="test.com.acme.gt15.Web.selenium.fixtures" />
</packages>
</test>
</suite>
```

ここにめでたく、反復可能な受け入れテスト・スイートを作成するのに必要なほとんどの作業が終わったことを宣言します。残るは、Web アプリケーションのコンテナ自体を処理するだけです。幸い、それには Cargo が活躍してくれます。

Cargo による作業の軽減

Cargo はコンテナ管理を一般的な方法で自動化する革新的なオープン・ソース・プロジェクトで、WAR ファイルを JBoss にデプロイするために使用する API で Tomcat の起動と停止も行うことができます。Cargo では、コンテナを自動的にダウンロードしてインストールすることも可能です。Java コードから Ant タスク、さらには Maven に至るまで、Cargo API の使い道は多岐にわたります。

Cargo のようなツールを使用すると、論理的に反復可能なテスト・ケースを作成する際の主要な課題の一つに対処できます。その課題とは、実行中のコンテナに最新かつ最良のアプリケーション・コードがあるという大前提を設けないということです。さらに、Cargo の機能を利用すれば、以下を自動的に行う (ANT 内にあるような) ビルド・プロセスを構築することができます。

1. 目的のコンテナのダウンロード
2. コンテナのインストール
3. コンテナの起動
4. 選択した WAR または EAR ファイルのコンテナへのデプロイ

以上に加え、選択したコンテナを Cargo で停止させることもできます (言い忘れていましたが、コンテナのダウンロードとインストールに関して心配する必要は一切ありません。正しいバージョンのコンテナがローカル・マシン上にすでにある場合、Cargo はステップ 1 とステップ 2 を省略します)。

以上の理由から、この例では Cargo を使用して、最新かつ最適なバージョンの Web アプリケーションが起動して実行されることを確実にしようと思います。その上、Cargo を使用すれば WAR ファイルをデプロイする場所を心配したり、最新の WAR が使用されていることを確認する必要もありません。何が本当の目的かと言えば、ユーザーの受け入れテストをイベントにせず、uat-test などの 1 つのコマンドを実行して、ただ座って結果を待てばいいだけにすることです。さらに CI 環境では待つ対象さえありません。テストが完了した後のどこかの時点で、通知を受け取るだけのことからです。

テスト・コンテナの管理

Ant 内に Cargo をセットアップするには、まず特定バージョンの Tomcat をダウンロードして、ローカル・マシン上の一時ディレクトリーにインストールするというタスクを定義します。続いて、最新バージョンのコード (WAR ファイルにバンドル) が Tomcat にデプロイされます (リスト 11 を参照)。

リスト 11. Cargo をセットアップするためのタスク

```
<target name="ua-test" depends="compile-tests,war">

  <taskdef resource="cargo.tasks">
    <classpath>
      <pathelement location="${libdir}/${cargo-jar}" />
      <pathelement location="${libdir}/${cargo-ant-jar}" />
    </classpath>
  </taskdef>

  <cargo containerId="tomcat5x" action="start" wait="false" id="${tomcat-refid}">
    <zipurlinstaller installurl="${tomcat-installer-url}" />
    <configuration type="standalone" home="${tomcatdir}">
      <property name="cargo.remote.username" value="admin" />
      <property name="cargo.remote.password" value="" />
      <deployable type="war" file="${wardir}/${warfile}" />
    </configuration>
  </cargo>

  <antcall target="_start-selenium" />

  <cargo containerId="tomcat5x" action="stop" refid="${tomcat-refid}" />
</target>
```

リスト 11 のターゲットは、`antcall` を使用して別のターゲットを呼び出します。基本的には、リスト 11 の最後の `cargo` タスクが `_start-selenium` ターゲットをラップして、テストの実行後に Tomcat を確実に停止するようにしています。

リスト 12 に定義する `_start-selenium` ターゲットでは、Selenium サーバーを起動 (そして後で停止) する必要があります。余談ですが、このサーバー・インスタンスには、私が作成したテストもそれぞれの Selenium フィクスチャーで接続することになります。このターゲットが、どのようにして別のターゲット (`_run-ua-tests`) を参照するかに注目してください ([リスト 13](#) に定義)。

リスト 12. Selenium サーバーの起動と停止

```
<target name="_start-selenium">
  <java jar="${libdir}/${selenium-srvr-jar}" fork="true" spawn="true" />
  <antcall target="_run-ua-tests" />
  <get dest="${testreportdir}/results.txt"
      src="${selenium-srvr-loc}/selenium-server/driver/?cmd=shutDown" />
</target>
```

グループ内の最後のターゲットが、TestNG を介してプログラムによる Selenium テストを実行します。TestNG に `testng.xml` ファイルを使用させるため、リスト 13 の `_run-ua-tests` ターゲットに含まれる `xmlfileset` 要素を使用していることは、注目に値する点です。

リスト 13. TestNG の testng.xml ファイルで検出されるテストの実行

```
<target name="_run-ua-tests">
  <taskdef classpathref="build.classpath" resource="testngtasks" />
  <testng outputDir="${testreportdir}"
        classpath="${testclassesdir};${classesdir}" haltonfailure="true">
    <xmlfileset dir="./test/conf" includes="testng.xml" />
    <classpath>
      <path refid="build.classpath" />
    </classpath>
  </testng>
</target>
```

まとめ

以上の説明からおわかりのように、Selenium は特に TestNG で制御された場合、ユーザー受け入れテストを極めて巧妙に容易なものにします。プログラムによるテストは全員を対象としているわけではありませんが (開発者でない人は、Selenium の Fit 形式テーブルを選ぶでしょう)、TestNG の並外れた機能を使用する手段となります。また、プログラムによるテストでは DbUnit と Cargo を備えた独自のテスト用フレームワークもビルドできるので、テストの論理的再現性が確実にあります。

コード品質の完璧主義者にとっての朗報は、オープン・ソースの Web テスト・フレームワークの発展は決して終わったわけではないということです。Selenium はユーザー受け入れテストを自動化するオープン・ソースのブラウザー駆動型 Web テスト・フレームワークに初めて吹き込まれた新風として、その存在を際立たせています。この記事で実践したように、Selenium をさらに TestNG と組み合わせれば、テストを意のままに操れるだけでなく、依存関係テストやパラメーター・テストの注目に値する利点も手に入れます。Selenium をぜひ TestNG で試してみてください。そうすれば、ユーザーもきっと感謝するはずです。

関連トピック

- 「[TestNG で Java ユニット・テストを楽々行う](#)」 (Filippo Diotalevi 著、developerWorks、2005年1月): TestNG によるユニット・テストを紹介しています。
- 「[コード品質を追及する: 反復可能なシステム・テスト](#)」 (Andrew Glover 著、developerWorks、2006年9月): 論理的に反復可能なテストが必要となる理由を理解してください。
- 「[コード品質を追求する: TestNG-Abbot による GUI テストの自動化](#)」 (Andrew Glover 著、developerWorks、2007年2月): GUI コンポーネントのテスト方法に新風を吹き込むテスト用フレームワーク、TestNG-Abbot について説明しています。
- 「[コード品質を追及する: FIT で解決する](#)」 (Andrew Glover 著、developerWorks、2006年2月): この統合テスト用フレームワークでは、開発者でなくても簡単にテストを作成できます。
- 「[DbUnit と Anthill によるテスト環境の制御](#)」 (Philippe Girolami 著、developerWorks、2004年4月): DbUnit と JUnit を併用すると、継続的インテグレーション・テスト環境でさえも制御できます。
- 「[実用的な Groovy: Groovy でサーバー側に対応する](#)」 (Andrew Glover 著、developerWorks、2005年3月): Groovy は、サーバー側アプリケーションを素早く簡単に開発するための簡略手段です。
- 「[DbUnit を使った効果的なユニットテスト](#)」 (Andrew Glover 著、OnJava、2004年1月): DbUnit を用いたデータベース依存テストを紹介しています。
- 「An interview with Cargo's Vincent Massol」 (Andrew Glover 著、thediscoblog.com、2006年2月): Cargo の創始者、Vincent Massol とのインタビュー記事です。
- 「[Hip system tests with Cargo](#)」 (Andrew Glover 著、thediscoblog.com、2006年4月): OCargo の Java API を紹介しています。
- [developerWorks Java technology ゾーン](#): Java プログラミングのあらゆる側面を網羅した記事が、豊富に用意されています。
- [Selenium のダウンロード](#): IE または Firefox で直接ユーザー受け入れテストを実行してください。
- [TestNG のダウンロード](#): Selenium のテストを操作する場合にも使える柔軟なテスト用フレームワークです。
- [Cargo のダウンロード](#): Web アプリケーションの反復可能なテストを一層簡単にします。
- [DbUnit のダウンロード](#): テストの実行前後でデータベースを既知の状態にします。

© Copyright IBM Corporation 2007

(www.ibm.com/legal/copytrade.shtml)

商標

(www.ibm.com/developerworks/jp/ibm/trademarks/)