

11

Testing with Groovy

In this chapter, we will cover:

- ▶ Unit testing Java code with Groovy
- ▶ Testing SOAP web services
- ▶ Testing RESTful services
- ▶ Writing functional tests for web applications
- ▶ Writing behavior-driven tests with Groovy
- ▶ Testing the database with Groovy
- ▶ Using Groovy in soapUI
- ▶ Using JMeter and Groovy for load testing

Introduction

This chapter is completely devoted to testing in Groovy. Testing is probably the most important activity that allows us to produce better software and make our users happier. The Java space has countless tools and frameworks that can be used for testing our software. In this chapter, we will direct our focus on some of those frameworks and how they can be integrated with Groovy. We will discuss not only unit testing techniques, but also integration and load testing strategies.

Starting from the king of all testing frameworks, JUnit and its seamless Groovy integration, we move to explore how to test:

- ▶ SOAP and REST web services
- ▶ Code that interacts with databases
- ▶ The web application interface using Selenium

The chapter also covers **Behavior Driven Development (BDD)** with **Spock**, advanced web service testing using **soapUI**, and load testing using **JMeter**.

Unit testing Java code with Groovy

One of the ways developers start looking into the Groovy language and actually using it is by writing unit tests. Testing Java code with Groovy makes the tests less verbose and it's easier for the developers that clearly express the intent of each test method.

Thanks to the Java/Groovy interoperability, it is possible to use any available testing or mocking framework from Groovy, but it's just simpler to use the integrated JUnit based test framework that comes with Groovy.

In this recipe, we are going to look at how to test Java code with Groovy.

Getting ready

This recipe requires a new Groovy project that we will use again in other recipes of this chapter. The project is built using Gradle (see the *Integrating Groovy into the build process using Gradle* recipe in *Chapter 2, Using Groovy Ecosystem*) and contains all the test cases required by each recipe.

Let's create a new folder called `groovy-test` and add a `build.gradle` file to it. The build file will be very simple:

```
apply plugin: 'groovy'
apply plugin: 'java'
repositories {
    mavenCentral()
    maven {
        url 'https://oss.sonatype.org' +
            '/content/repositories/snapshots'
    }
}

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.1.6'
    testCompile 'junit:junit:4.+'
```

The standard source folder structure has to be created for the project. You can use the following commands to achieve this:

```
mkdir -p src/main/groovy
mkdir -p src/test/groovy
mkdir -p src/main/java
```

Verify that the project builds without errors by typing the following command in your shell:

```
gradle clean build
```

How to do it...

To show you how to test Java code from Groovy, we need to have some Java code first!

1. So, this recipe's first step is to create a simple Java class called `StringUtil`, which we will test in Groovy. The class is fairly trivial, and it exposes only one method that concatenates `String` passed in as a `List`:

```
package org.groovy.cookbook;
import java.util.List;
public class StringUtil {
    public String concat(List<String> strings,
                        String separator) {
        StringBuilder sb = new StringBuilder();
        String sep = "";
        for(String s: strings) {
            sb.append(sep).append(s);
            sep = separator;
        }
        return sb.toString();
    }
}
```

Note that the class has a specific package, so don't forget to create the appropriate folder structure for the package when placing the class in the `src/main/java` folder.

2. Run the build again to be sure that the code compiles.
3. Now, add a new test case in the `src/test/groovy` folder:

```
package org.groovy.cookbook.javatesting
import org.groovy.cookbook.StringUtil
class JavaTest extends GroovyTestCase {
    def stringUtil = new StringUtil()
    void testConcatenation() {
        def result = stringUtil.concat(['Luke', 'John'], '-')
        assertEquals('Luke-John', result)
    }
    void testConcatenationWithEmptyList() {
        def result = stringUtil.concat([], ',')
        assertEquals('', result)
    }
}
```

Again, pay attention to the package when creating the test case class and create the package folder structure.

4. Run the test by executing the following Gradle command from your shell:

```
gradle clean build
```

Gradle should complete successfully with the `BUILD SUCCESSFUL` message.

5. Now, add a new test method and run the build again:

```
void testConcatenationWithNullShouldReturnNull() {  
    def result = StringUtil.concat(null, ',')  
    assertEquals('', result)  
}
```

This time the build should fail:

```
3 tests completed, 1 failed  
:test FAILED  
FAILURE: Build failed with an exception.
```

6. Fix the test by adding a null check to the Java code as the first statement of the `concat` method:

```
if (strings == null) {  
    return "";  
}
```

7. Run the build again to verify that it is now successful.

How it works...

The test case shown at step 3 requires some comments. The class extends `GroovyTestCase` is a base test case class that facilitates the writing of unit tests by adding several helper methods to the classes extending it. When a test case extends from `GroovyTestCase`, each test method name must start with `test` and the return type must be `void`. It is possible to use the JUnit 4.x `@Test` annotation, but, in that case, you don't have to extend from `GroovyTestCase`.

The standard JUnit assertions (such as `assertEquals` and `assertNull`) are directly available in the test case (without explicit import) plus some additional assertion methods are added by the super class. The test case at step 3 uses `assertToString` to verify that a `String` matches the expected result. There are other assertions added by `GroovyTestCase`, such as `assertArrayEquals`, to check that two arrays contain the same values, or `assertContains` to assert that an array contains a given element.

There's more...

The `GroovyTestCase` class also offers an elegant method to test for expected exceptions. Let's add the following rule to the `concat` method:

```
if (separator.length() != 1) {
    throw new IllegalArgumentException(
        "The separator must be one char long");
}
```

Place the separator length check just after the null check for the `List`. Add the following new test method to the test case:

```
void testVerifyExceptionOnWrongSeparator() {
    shouldFail IllegalArgumentException, {
        stringUtil(['a', 'b'], ',,')
    }
    shouldFail IllegalArgumentException, {
        stringUtil(['c', 'd'], '')
    }
}
```

The `shouldFail` method takes a closure that is executed in the context of a try-catch block. We can also specify the expected exception in the `shouldFail` method. The `shouldFail` method makes testing for exceptions very elegant and simple to read.

See also

- ▶ <http://junit.org/>
- ▶ <http://groovy.codehaus.org/api/groovy/util/GroovyTestCase.html>

Testing SOAP web services

In the *Issuing a SOAP request and parsing a response* recipe in *Chapter 8, Working with Web Services in Groovy*, we have learned how to execute a request against a SOAP web service. In this recipe, we put into practice the concepts learned in that recipe and combine them with the unit testing capabilities of Groovy.

Additionally, this recipe shows you how to use the JUnit 4 annotations instead of the JUnit 3 API offered by `GroovyTestCase` (see the *Unit testing Java code with Groovy* recipe).

Getting ready

For this recipe, we are going to use a publicly available web service, the US holiday date web service hosted at <http://www.holidaywebservice.com>. The WSDL of the service can be found on the `/Holidays/US/Dates/USHolidayDates.asmx?WSDL` path. We have already encountered this service in the *Issuing a SOAP request and parsing a response* recipe in *Chapter 8, Working with Web Services in Groovy*. Each service operation simply returns the date of a given US holiday such as Easter or Christmas.

How to do it...

We start from the Gradle build that we created in the *Unit testing Java code with Groovy* recipe.

1. Add the following dependency to the `dependencies` section of the `build.gradle` file:

```
testCompile 'com.github.groovy-wslite:groovy-wslite:0.8.0'
```

The `groovy-wslite` library is mentioned in the *Issuing a SOAP request and parsing a response* recipe in *Chapter 8, Working with Web Services in Groovy*. This library is the de facto SOAP library for Groovy.

2. Let's create a new unit test for verifying one of the web service operations. As usual, the test case is created in the `src/test/groovy/org/groovy/cookbook` folder:

```
package org.groovy.cookbook.soap
```

```
import static org.junit.Assert.*
import org.junit.Test
import wslite.soap.*
```

```
class SoapTest {

    ...

}
```

3. Add a new test method to the body of the class:

```
@Test
void testMLKDay() {

    def baseUrl = 'http://www.holidaywebservice.com'
    def service = '/Holidays/US/Dates/USHolidayDates.asmx?WSDL'
    def client = new SOAPClient("${baseUrl}${service}")
    def baseNS = 'http://www.27seconds.com/Holidays/US/Dates/'
    def action = "${baseNS}GetMartinLutherKingDay
```

```

"
    def response = client.send(SOAPAction: action) {
        body {
            GetMartinLutherKingDay('xmlns': baseNS) {
                year(2013)
            }
        }
    }

    def date = Date.parse(
        "yyyy-MM-dd'T'hh:mm:ss",
        response.
            GetMartinLutherKingDayResponse.
            GetMartinLutherKingDayResult.text()
    )

    assertEquals(
        date,
        Date.parse('yyyy-MM-dd', '2013-01-15')
    )

    assertEquals(
        response.httpResponse.headers['X-Powered-By'].value,
        'ASP.NET'
    )
}

```

4. Build the code and execute the test from the command line by executing:

```
gradle -Dtest.single=SoapTest clean test
```

How it works...

The test code creates a new `SOAPClient` with the URI of the target web service. The request is created using Groovy's `MarkupBuilder` (see the *Constructing XML content* recipe in *Chapter 5, Working with XML in Groovy*). The body closure (and if needed, also the header closure) is passed to the `MarkupBuilder` for the SOAP message creation. The assertion code gets the result from the response, which is automatically parsed by `XMLSlurper`, allowing easy access to elements of the response such as the header or the body elements. In the above test, we simply check that the returned Martin Luther King day matches with the expected one for the year 2013.

There's more...

If you require more control over the content of the **SOAP** request, the `SOAPClient` also supports sending the **SOAP** envelope as a `String`, such as in this example:

```
def response = client.send (
    """<?xml version='1.0' encoding='UTF-8'?>
    <soapenv:Envelope
    xmlns:soapenv='http://schemas.xmlsoap.org/soap/envelope/'
    xmlns:dat='http://www.27seconds.com/Holidays/US/Dates/'>
    <soapenv:Header/>
    <soapenv:Body>
    <dat:GetMartinLutherKingDay>
    <dat:year>2013</dat:year>
    </dat:GetMartinLutherKingDay>
    </soapenv:Body>
    </soapenv:Envelope>
    """
)
```

Replace the call to the `send` method in step 3 with the one above and run your test again.

See also

- ▶ <https://github.com/jwagenleitner/groovy-wslite>
- ▶ <http://www.holidaywebservice.com>
- ▶ The *Constructing XML content* recipe in *Chapter 5, Working with XML in Groovy*
- ▶ The *Issuing a SOAP request and parsing a response* recipe in *Chapter 8, Working with Web Services in Groovy*

Testing RESTful services

This recipe is very similar to the previous recipe *Testing SOAP web services*, except that it shows how to test a RESTful service using Groovy and JUnit.

Getting ready

For this recipe, we are going to use a test framework aptly named Rest-Assured. This framework is a simple DSL for testing and validating REST services returning either JSON or XML.

Before we start to delve into the recipe, we need to start a simple REST service for testing purposes. We are going to use the Ratpack framework, which we already encountered in the *Running tasks in parallel and asynchronously* recipe in *Chapter 10, Concurrent programming in Groovy*. The test REST service, which we will use, exposes three APIs to fetch, add, and delete books from a database using JSON as lingua franca.

For the sake of brevity, the code for the setup of this recipe is available in the `rest-test` folder of the companion code for this chapter. The code contains the Ratpack server, the domain objects, Gradle build, and the actual test case that we are going to analyze in the next section.

How to do it...

The test case takes care of starting the Ratpack server and execute the REST requests.

1. Here is the `RestTest` class located in `src/test/groovy/org/groovy/cookbok/rest` folder:

```
package org.groovy.cookbook.rest

import static com.jayway.restassured.RestAssured.*
import static com.jayway.restassured.matcher.
    RestAssuredMatchers.*

import static org.hamcrest.Matchers.*
import static org.junit.Assert.*

import groovy.json.JsonBuilder

import org.groovy.cookbook.server.*
import org.junit.AfterClass
import org.junit.BeforeClass
import org.junit.Test

class RestTest {

    static server
    final static HOST = 'http://localhost:5050'
    ,
    @BeforeClass
    static void setUp() {
        server = App.init()
        server.startAndWait()
    }
}
```

```
@AfterClass
static void tearDown() {
    if(server.isRunning()) {
        server.stop()
    }
}

@Test
void testGetBooks() {
    expect().
        body('author',
            hasItems('Ian Bogost', 'Nate Silver')).
    when().get("${HOST}/api/books")
}

@Test
void testGetBook() {
    expect().
        body('author', is('Steven Levy')).
    when().get("${HOST}/api/books/5")
}

@Test
void testPostBook() {
    def book = new Book()
    book.author = 'Haruki Murakami'
    book.date = '2012-05-14'
    book.title = 'Kafka on the shore'
    JsonBuilder jb = new JsonBuilder()
    jb.content = book
    given().
        content(jb.toString()).
    expect().body('id', is(6)).
    when().post("${HOST}/api/books/new")
}

@Test
void testDeleteBook() {
    expect().statusCode(200).
    when().delete("${HOST}/api/books/1")
    expect().body('id', not(hasValue(1))).
    when().get("${HOST}/api/books")
}
}
```

2. Build the code and execute the test from the command line by typing:

```
gradle clean test
```

How it works...

The JUnit test has a `@BeforeClass` annotated method, executed at the beginning of the unit test, that starts the Ratpack server and the associated REST services. The `@AfterClass` annotated method, on the contrary, shuts down the server when the test is over.

The unit test has four test methods. The first one, `testGetBooks` executes a `GET` request against the server and retrieves all the books.

The rather readable DSL offered by the Rest-Assured framework should be easy to follow. The `expect` method starts building the response expectation returned from the `get` method. The actual assert of the test is implemented via a **Hamcrest** matcher (hence the static `org.hamcrest.Matchers.*` import in the test). The test is asserting that the body of the response contains two books that have the author named `Ian Bogost` or `Greg Grandin`. The `get` method hits the URL of the embedded Ratpack server, started at the beginning of the test.

The `testGetBook` method is rather similar to the previous one, except that it uses the `is` matcher to assert the presence of an author on the returned JSON message.

The `testPostBook` tests that the creation of a new book is successful. First, a new book object is created and transformed into a JSON object using `JsonBuilder` (see the *Constructing JSON messages with JsonBuilder* recipe in *Chapter 6, Working with JSON in Groovy*). Instead of the `expect` method, we use the `given` method to prepare the `POST` request. The `given` method returns a `RequestSpecification` to which we assign the newly created book and finally, invoke the `post` method to execute the operation on the server. As in our book database, the biggest identifier is `8`, the new book should get the `id 9`, which we assert in the test.

The last test method (`testDeleteBook`) verifies that a book can be deleted. Again we use the `expect` method to prepare the response, but this time we verify that the returned HTTP status code is `200` (for deletion) upon deleting a book with the `id 1`. The same test also double-checks that fetching the full list of books does not contain the book with `id` equals to `1`.

See also

- ▶ <https://code.google.com/p/rest-assured/>
- ▶ <https://code.google.com/p/hamcrest/>
- ▶ <https://github.com/ratpack/ratpack>
- ▶ The *Constructing JSON messages with JsonBuilder* recipe in *Chapter 6, Working with JSON in Groovy*

- ▶ The *Issuing a REST request and parsing a response* recipe in *Chapter 8, Working with Web Services in Groovy*

Writing functional tests for web applications

If you are developing web applications, it is of utmost importance that you thoroughly test them before allowing user access. Web GUI testing can require long hours of very expensive human resources to repeatedly exercise the application against a varied list of input conditions. **Selenium**, a browser-based testing and automation framework, aims to solve these problems for software developers, and it has become the *de facto* standard for web interface integration and functional testing.

Selenium is not just a single tool but a suite of software components, each catering to different testing needs of an organization. It has four main parts:

- ▶ **Selenium Integrated Development Environment (IDE)**: It is a Firefox add-on that you can only use for creating relatively simple test cases and test suites.
- ▶ **Selenium Remote Control (RC)**: It also known as Selenium 1, is the first Selenium tool that allowed users to use programming languages in creating complex tests.
- ▶ **WebDriver**: It is the newest Selenium implementation that allows your test scripts to communicate directly to the browser, thereby controlling it from the OS level.
- ▶ **Selenium Grid**: It is a tool that is used with Selenium RC to execute parallel tests across different browsers and operating systems.

Since 2008, Selenium RC and WebDriver are merged into a single framework to form Selenium 2. Selenium 1 refers to Selenium RC.

This recipe will show you how to write a Selenium 2 based test using `HtmlUnitDriver`. `HtmlUnitDriver` is the fastest and most lightweight implementation of `WebDriver` at the moment. As the name suggests, it is based on `HtmlUnit`, a relatively old framework for testing web applications.

The main disadvantage of using this driver instead of a `WebDriver` implementation that "drives" a real browser is the JavaScript support. None of the popular browsers use the JavaScript engine used by `HtmlUnit` (Rhino). If you test JavaScript using `HtmlUnit`, the results may divert considerably from those browsers. Still, `WebDriver` and `HtmlUnit` can be used for fast paced testing against a web interface, leaving more JavaScript intensive tests to other, long running, `WebDriver` implementations that use specific browsers.

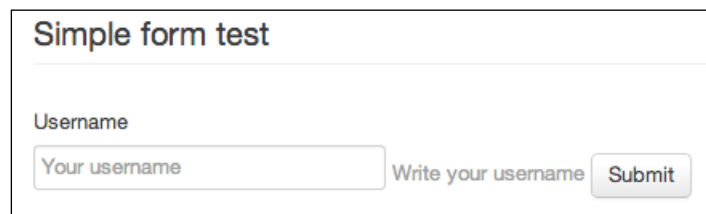
Getting ready

Due to the relative complexity of the setup required to demonstrate the steps of this recipe, it is recommended that the reader uses the code that comes bundled with this recipe. The code is located in the `selenium-test` located in the code directory for this chapter. The source code, as other recipes in this chapter, is built using Gradle and has a standard structure, containing application code and test code.

The web application under test is very simple. It is composed of two pages. A welcome page that looks similar to the following screenshot:



And a single field test form page:

A screenshot of a web form titled "Simple form test" in a bold, dark blue font. Below the title is a horizontal line. Underneath, the label "Username" is followed by a text input field containing the placeholder text "Your username". To the right of the input field is the text "Write your username" and a "Submit" button.

The Ratpack framework is utilized to run the fictional web application and serve the HTML pages along with some JavaScript and CSS.

How to do it...

The following steps will describe the salient points of Selenium testing with Groovy.

1. Let's open the `build.gradle` file. We are interested in the dependencies required to execute the tests:

```
testCompile group: 'org.seleniumhq.selenium',
            name: 'selenium-htmlunit-driver',
            version: '2.32.0'
testCompile group: 'org.seleniumhq.selenium',
            name: 'selenium-support',
            version: '2.9.0'
```

2. Let's open the test case, `SeleniumTest.groovy` located in `test/groovy/org/groovy/cookbook/selenium`:

```
package org.groovy.cookbook.selenium

import static org.junit.Assert.*

import org.groovy.cookbook.server.*
import org.junit.AfterClass
import org.junit.BeforeClass
import org.junit.Test
import org.openqa.selenium.By
import org.openqa.selenium.WebDriver
import org.openqa.selenium.WebElement
import org.openqa.selenium.htmlunit.HtmlUnitDriver
import org.openqa.selenium.support.ui.WebDriverWait

import com.google.common.base.Function

class SeleniumTest {

    static server
    static final HOST = 'http://localhost:5050'
    static HtmlUnitDriver driver

    @BeforeClass
    static void setUp() {
        server = App.init()
        server.startAndWait()
        driver = new HtmlUnitDriver(true)
    }

    @AfterClass
    static void tearDown() {
        if (server.isRunning()) {
            server.stop()
        }
    }

    @Test
    void testWelcomePage() {
        driver.get(HOST)
        assertEquals('welcome', driver.title)
    }
}
```

```
@Test
void testFormPost() {

    driver.get("${HOST}/form")
    assertEquals('test form', driver.title)

    WebElement input =
        driver.findElement(By.name('username'))

    input.sendKeys('test')
    input.submit()

    WebDriverWait wait = new WebDriverWait(driver, 4)
    wait.until(ExpectedConditions.
        presenceOfElementLocated(By.className('hello')))

    assertEquals('oh hello,test', driver.title)

}
}
```

How it works...

The test case initializes the Ratpack server and the `HtmlUnit` driver by passing `true` to the `HtmlUnitDriver` instance. The `boolean` parameter in the constructor indicates whether the driver should support JavaScript.

The first test, `testWelcomePage`, simply verifies that the title of the website's welcome page is as expected. The `get` method executes an HTTP GET request against the URL specified in the method, the Ratpack server in our test.

The second test, `testFormPost`, involves the DOM manipulation of a form, its submission, and waiting for an answer from the server.

The Selenium API should be fairly readable. For a start, the test checks that the page containing the form has the expected title. Then the element named `username` (a form field) is selected, populated, and finally submitted. This is how the HTML looks for the form field:

```
<input type="text"
  name="username"
  placeholder="Your username">
```

The test uses the `findElement` method to select the `input` field. The method expects a `By` object that is essentially a mechanism to locate elements within a document. Elements can be identified by name, id, text link, tag name, CSS selector, or XPath expression.

The form is submitted via AJAX. Here is part of the JavaScript activated by the form submission:

```
complete:function(xhr, status) {
  if (status === 'error' || !xhr.responseText) {
    alert('error')
  } else {
    document.title = xhr.responseText
    jQuery(e.target).
      replaceWith('<p class="hello">' +
        xhr.responseText +
        '</p>')
  }
}
```

After the form submission, the DOM of the page is manipulated to change the page title of the form page and replace the form DOM element with a message wrapped in a paragraph element.

To verify that the DOM changes have been applied, the test uses the `WebDriverWait` class to wait until the DOM is actually modified and the element with the class `hello` appears on the page. The `WebDriverWait` is instantiated with a four seconds timeout.

This recipe only scratches the surface of the Selenium 2 framework's capabilities, but it should get you started to implement your own integration and functional test.

See also

- ▶ <http://docs.seleniumhq.org/>
- ▶ <http://htmlunit.sourceforge.net/>

Writing behavior-driven tests with Groovy

Behavior Driven Development, or simply **BDD**, is a methodology where QA, business analysts, and marketing people could get involved in defining the requirements of a process in a common language. It could be considered an extension of Test Driven Development, although is not a replacement. The initial motivation for BDD stems from the perplexity of business people (analysts, domain experts, and so on) to deal with "tests" as these seem to be too technical. The employment of the word "behaviors" in the conversation is a way to engage the whole team.

BDD states that software tests should be specified in terms of the desired behavior of the unit. The behavior is expressed in a semi-formal format, borrowed from user story specifications, a format popularized by agile methodologies.

For a deeper insight into the BDD rationale, it is highly recommended to read the original paper from Dan North available at <http://dannorth.net/introducing-bdd/>.

Spock is one of the most widely used frameworks in the Groovy and Java ecosystem that allows the creation of BDD tests in a very intuitive language and facilitates some common tasks such as mocking and extensibility. What makes it stand out from the crowd is its beautiful and highly expressive specification language. Thanks to its JUnit runner, Spock is compatible with most IDEs, build tools, and continuous integration servers.

In this recipe, we are going to look at how to implement both a unit test and a web integration test using Spock.

Getting ready

This recipe has a slightly different setup than most of the recipes in this book, as it resorts to an existing web application source code, freely available on the Internet. This is the **Spring Petclinic** application provided by **SpringSource** as a demonstration of the latest Spring framework features and configurations. The web application works as a pet hospital and most of the interactions are typical CRUD operations against certain entities (veterinarians, pets, pet owners). The Petclinic application is available in the `groovy-test/spock/web` folder of the companion code for this chapter.

All Petclinic's original tests have been converted to Spock. Additionally we created a simple integration test that uses Spock and Selenium to showcase the possibilities offered by the integration of the two frameworks.

As usual, the recipe uses Gradle to build the reference web application and the tests.

The Petclinic web application can be started by launching the following Gradle command in your shell from the `groovy-test/spock/web` folder:

```
gradle tomcatRunWar
```

If the application starts without errors, the shell should eventually display the following message:

```
The Server is running at http://localhost:8080/petclinic
```

Take some time to familiarize yourself with the Petclinic application by directing your browser to <http://localhost:8080/petclinic> and browsing around the website:



How to do it...

The following steps will describe the key concepts for writing your own behavior-driven unit and integration tests:

1. Let's start by taking a look at the dependencies required to implement a Spock-based test suite:

```
testCompile 'org.spockframework:spock-core:0.7-groovy-2.0'
testCompile group: 'org.seleniumhq.selenium',
    name: 'selenium-java', version: '2.16.1'
testCompile group: 'junit', name: 'junit', version: '4.10'
testCompile 'org.hamcrest:hamcrest-core:1.2'
testRuntime 'cglib:cglib-nodep:2.2'
testRuntime 'org.objenesis:objenesis:1.2'
```

2. This is what a BDD unit test for the application's business logic looks like:

```
package org.springframework.samples.petclinic.model
import spock.lang.*
class OwnerTest extends Specification {
    def "test pet and owner" () {
        given:
            def p = new Pet()
            def o = new Owner()
        when:
            p.setName("fido")
        then:
```

```

        o.getPet("fido") == null
        o.getPet("Fido") == null
    when:
        o.addPet(p)
    then:
        o.getPet("fido").equals(p)
        o.getPet("Fido").equals(p)
    }
}

```

The test is named `OwnerTest.groovy`, and it is available in the `spock/web/src/test` folder of the main `groovy-test` project that comes with this chapter.

3. The third test in this recipe mixes Spock and Selenium, the web testing framework already discussed in *Writing functional tests for web applications* recipe:

```

package org.cookbook.groovy.spock

import static java.util.concurrent.TimeUnit.SECONDS
import org.openqa.selenium.By
import org.openqa.selenium.WebElement
import org.openqa.selenium.htmlunit.HtmlUnitDriver
import spock.lang.Shared

import spock.lang.Specification

class HomeSpecification extends Specification {

    static final HOME = 'http://localhost:9966/petclinic
    '

    @Shared
    def driver = new HtmlUnitDriver(true)

    def setup() {
        driver.manage().timeouts().implicitlyWait 10, SECONDS
    }

    def 'user enters home page'() {
        when:
            driver.get(HOME)
        then:
            driver.title == 'PetClinic :: ' +
                'a Spring Framework demonstration'
    }

    def 'user clicks on menus'() {

```

```
when:
    driver.get(HOME)
    def vets = driver.findElement(By.linkText('Veterinarians'))
    vets.click()
then:
    driver.currentUrl ==
        'http://localhost:9966/petclinic/vets.html'
}
```

The test above is available in the `spock/specs/src/test` folder of the accompanying project.

How it works...

The first step of this recipe lays out the dependencies required to set up a Spock-based BDD test suite. Spock requires Java 5 or higher, and it's pretty picky with regard to the matching Groovy version to use. In the case of this recipe, as we are using Groovy 2.x, we set the dependency to the `0.7-groovy-2.0` version of the Spock framework.

The full `build.gradle` file is located in the `spock/specs` folder of the recipe's code.

The first test case demonstrated in the recipe is a direct conversion of a JUnit test written by Spring for the Petclinic application. This is the original test written in Java:

```
public class OwnerTests {
    @Test
    public void testHasPet() {
        Owner owner = new Owner();
        Pet fido = new Pet();
        fido.setName("Fido");
        assertNull(owner.getPet("Fido"));
        assertNull(owner.getPet("fido"));
        owner.addPet(fido);
        assertEquals(fido, owner.getPet("Fido"));
        assertEquals(fido, owner.getPet("fido"));
    }
}
```

All we need to import in the Spock test is `spock.lang.*` that contains the most important types for writing specifications. A Spock test extends from `spock.lang.Specification`. The name of a specification normally relates to the system or system operation under test. In the case of the Groovy test at step 2, we reused the original Java test name, but it would have been better renamed to something more meaningful for a specification such as `OwnerPetSpec`. The class `Specification` exposes a number of practical methods for implementing specifications. Additionally, it tells JUnit to run the specification with Sputnik, Spock's own JUnit runner. Thanks to Sputnik, Spock specifications can be executed by all IDEs and build tools.

Following the class definition, we have the feature method:

```
def 'test pet and owner' () {  
    ...  
}
```

Feature methods lie at the core of a specification. They contain the description of the features (properties, aspects) that define the system that is under specification test. Feature methods are conventionally named with `String` literals: it's a good idea to choose meaningful names for feature methods.

In the test above, we are testing that: given two entities, `pet` and `owner`, the `getPet` method of the `owner` instance will return `null` until the `pet` is not assigned to the `owner`, and that the `getPet` method will accept both "fido" and "Fido" in order to verify the ownership.

Conceptually, a feature method consists of four phases:

- ▶ Set up the feature's fixture
- ▶ Provide an input to the system under specification (stimulus)
- ▶ Describe the response expected from the system
- ▶ Clean up

Whereas the first and last phases are optional, the stimulus and response phases are always present and may occur more than once.

Each phase is defined by blocks: blocks are defined by a label and extend to the beginning of the next block or the end of the method. In the test at step 2, we can see 3 types of blocks:

- ▶ In the `given` block, data gets initialized
- ▶ The `when` and `then` blocks always occur together. They describe a stimulus and the expected response. Whereas `when` blocks may contain arbitrary code; `then` blocks are restricted to conditions, exception checking, interactions, and variable definitions.

The first test case has two `when/then` pairs. A `pet` is assigned the name "fido", and the test verifies that calling `getPet` on an `owner` object only returns something if the `pet` is actually "owned" by the `owner`.

The second test is slightly more complex because it employs the Selenium framework to execute a web integration test with a BDD flavor. The test is located in the `groovy-test/spock/specs/src/test` folder. You can launch it by typing `gradle test` from the `groovy-test/spock/specs` folder. The test takes care of starting the web container and run the application under test, `Petclinic`. The test starts by defining a shared Selenium driver marked with the `@Shared` annotation, which is visible by all the feature methods. The first feature method simply opens the `Petclinic` main page and checks that the title matches the specification. The second feature method uses the Selenium API to select a link, click on it, and verify that the link brings the user to the right page. The verification is performed against the `currentUrl` of the browser that is expected to match the URL of the link we clicked on.

See also

- ▶ <http://dannorth.net/introducing-bdd/>
- ▶ http://en.wikipedia.org/wiki/Behavior-driven_development
- ▶ <https://code.google.com/p/spock/>
- ▶ <https://github.com/spring-projects/spring-petclinic/>

Testing the database with Groovy

When developing data-driven applications, which perform operations against a database such as `SELECT` or `DELETE`, we normally unit test these applications by mocking the data layer. The reason is that launching or connecting to a full-fledged database when running a bunch of unit tests slows down the test execution and introduces a strong dependency. Should our database be down, all our tests would fail. Furthermore, testing against a database imposes that the database is in a well-known state, or, alternatively, it is completely empty. This is often not possible.

However, at some point we need to be able to test our data layer, regardless of the technology (JDBC and ORM), and that cannot be done without accessing a real database. Either our SQL queries or our ORM mapping have to be tested for correctness. This will verify that the database schema matches the queries—either generated or hand written.

For many years, the king of the database-driven test frameworks has been DBUnit. DBUnit performs the following operations to help you with database testing:

- ▶ Exporting data from the database to flat XML files: each test can then load data from these files
- ▶ Restoring the original state of the database before executing the test
- ▶ Comparing the actual data in the database and the expected datasets in order to determine the success or failure of a particular test
- ▶ Deleting any changes made after the test is complete

In this recipe, we will explore the DBUnit API and how Groovy can simplify their consumption.

Getting ready

This recipe doesn't ask for much of a setup, as we are reusing the test infrastructure defined in the, *Unit testing Java code with Groovy* recipe. The only required step is adding few dependencies to the `build.gradle` script:

```
testCompile 'org.dbunit:dbunit:2.4.9'
testRuntime 'org.hsqldb:hsqldb:2.3.0'
testRuntime 'org.slf4j:slf4j-api:1.7.5'
```

The first dependency is DBUnit itself. The second dependency is **HSQldb (HyperSQL)**, a Java-based database that will be embedded in the test. We used HSQldb in almost every recipe of *Chapter 7, Working with Databases in Groovy*.

How to do it...

The following steps will present an approach for database testing using DBUnit.

1. Create a new JUnit test in the test folder of the project defined in the first recipe of this chapter. Name the test class `DBUnitTest`. As usual, pay attention to the package name and don't forget to create the proper folders under `src/test/groovy`:

```
package org.groovy.cookbook.dbunit

import static org.junit.Assert.*

import groovy.sql.Sql
import groovy.xml.StreamingMarkupBuilder
import org.dbunit.JdbcDatabaseTester
import org.dbunit.database.*
import org.dbunit.dataset.xml.*
import org.junit.Before
import org.junit.BeforeClass
import org.junit.Test

class DbUnitTest {

    ...

}
```

2. Add the following methods for setting up the database:

```
def tester

final static DB_DRIVER = 'org.hsqldb.jdbcDriver'
final static DB_CONFIG = ['jdbc:hsqldb:db/sample', 'sa', '']

@BeforeClass
static void schema() {
    Sql.newInstance(*DB_CONFIG, DB_DRIVER).execute('''
        drop table if exists post;
        drop table if exists author;
        create table author(
            id integer primary key,
            name varchar(500)
        )
    ''')
```

```
    );
    create table post(
        id integer primary key,
        title varchar(500),
        text longvarchar,
        author integer not null,
        foreign key (author) references author(id)
    );
    '''
}

@Before
void setUp() {
    tester = new JdbcDatabaseTester(DB_DRIVER, *DB_CONFIG)
}
```

3. Finally, let's add two test methods:

```
@Test
void testSelectWithId() {
    tester.dataSet = dataSet {
        author id: 1, name: 'Admin'
        post id: 1,
            author: 1,
            title: 'DBUnit testing for everyone!',
            text: 'How to use DBUnit to rock your tests'
    }
    tester.onSetup()

    def row = firstRow(''
        select id, title, text
        from post where id = ?
    '', [1])

    assertEquals(1, row.id)
    assertEquals('Testing db with dbunit is easy',
        row.title)
}

@Test
void testSelectWithAuthorName() {
    tester.dataSet = dataSet {
        author id: 1, name: 'testName'
        post id: 1,
            author: 1,
```



```

        title: 'Testing a sql join',
        text: 'some text...'
    }
    tester.onSetup()

    def row = firstRow('''
        select p.id, p.title, p.text
        from post p
        left join author a
        on a.id = p.author
        where a.name = ?
    ''', ['testName'])

    assertEquals(1, row.id)
    assertEquals('some text...', row.text)
}

def firstRow(query, List args) {
    Sql.newInstance(DB_DRIVER, *DB_CONFIG).firstRow(query, args)
}

def dataSet(dataClosure) {
    new FlatXmlDataSet(
        new StringReader(
            new StreamingMarkupBuilder().
                bind{dataset dataClosure}.toString()
        )
    )
}

```

How it works...

In step 1, we have a bunch of imports mostly related to `DBUnit`, but also `StreamingMarkupBuilder` that is going to be used later. The `DB_CONFIG` list contains the connection data required to access the database. Using the Groovy spread (`*`) operator, we convert the collection into individual parameters for the `Sql.newInstance` method.

Step 2 contains the methods to initialize the database. The static `schema` method (annotated with `@BeforeClass` gets executed at the beginning of the test) connects to the in-memory `HSQLDB` database and executes the schema creation script. Two tables are created, `post` and `author`—a basic structure for a blogging platform.

The second initialization method (`setup`) executed for each test method creates a new `JdbcDatabaseTester`, a `DBUnit` object responsible for holding the database connection and configuring/executing the datasets.

Let's look at the test methods. Both test methods populate the tables with some data taken from a dataset. A dataset, in DBUnit, is an abstraction to manipulate tabular data, represented by the `org.dbunit.dataset.IDataset` interface. In other words, a dataset contains the data with which we seed the database before running our test in order to place the database into a well-known state.

There are several implementations of the `IDataset` interface; each implementation is specialized in dealing with a specific data format:

- ▶ `FlatXmlDataSet` reads and writes a DBUnit specific XML format
- ▶ `XmlDataSet` reads and writes more verbose XML data
- ▶ `XlsDataSet` reads from Excel files

For a complete list of dataset providers, please check the official DBUnit documentation on datasets: <http://dbunit.sourceforge.net/components.html#IDataset>.

In both tests, we set the dataset by calling the `dataSet` method. This method accepts a closure, which transforms the data structure (basically, `List` of `Map`) defined in the test into XML using the `StreamingMarkupBuilder`. The dataset is transformed from this:

```
author id: 1, name: 'Admin'
post id: 1,
  author: 1,
  title: 'Testing db with dbunit is easy',
  text: 'How to use DBUnit to rock your tests'
```

to this:

```
<author id='1' name='Admin' />
<post id='1'
  author='1'
  title='Testing db with dbunit is easy'
  text='How to use DBUnit to rock your tests'/>
```

Each test method needs to explicitly invoke the `onSetup` method of the `JdbcDatabaseTester` to insert the dataset in the database. Both test methods are similar in the type of test they perform. First, the test inserts a dataset into the database, and then it asserts that a SQL query returns the expected result. In real life, you'd probably have some kind of data layer **Data Access Object (DAO)** that would be the class under test. The data layer would expose methods, such as `saveUser` or `getSalaryByEmployeeId`, which would access the database to perform the requested operations. With DBUnit, it's easy to verify that data are correctly persisted or fetched from the database, and, by using Groovy elegance, we can make the tests less verbose and easy to read.

See also

- ▶ <http://dbunit.sourceforge.net>
- ▶ <http://dbunit.sourceforge.net/components.html#IDataSet>
- ▶ <http://hsqldb.org/>
- ▶ *Chapter 7, Working with Databases in Groovy*

Using Groovy in soapUI

soapUI is a popular tool used for web service testing and, recently, load testing too. It's easy to run, configure, and use; thanks to a polished Java Swing GUI as well as the command line interface.

Out of the box, the open source version of soapUI comes with several features:

- ▶ **WS-* standards support:** Specifically, soapUI supports most of the web service specifications such as WS-Security and WS-Addressing, among others
- ▶ **Service mocking:** With soapUI, it's easy to simulate a web service before it is created, for testing out the clients
- ▶ **Scripting:** soapUI supports Groovy and JavaScript to run before and after processing operations
- ▶ **Functional testing:** soapUI can run functional tests against web services, web applications, and JDBC data sources
- ▶ **Performance testing:** With just a few clicks, you can generate performance and load tests quickly using soapUI
- ▶ **Test automation:** soapUI is easy to integrate into testing frameworks, such as JUnit, and to launch by build tools such as Maven or Gradle

In this recipe, we are going to use soapUI to test the US holiday date web service that we already encountered in the *Testing SOAP web services* recipe. soapUI has support for Groovy scripting, which we will use to make our soapUI test more efficient and useful.

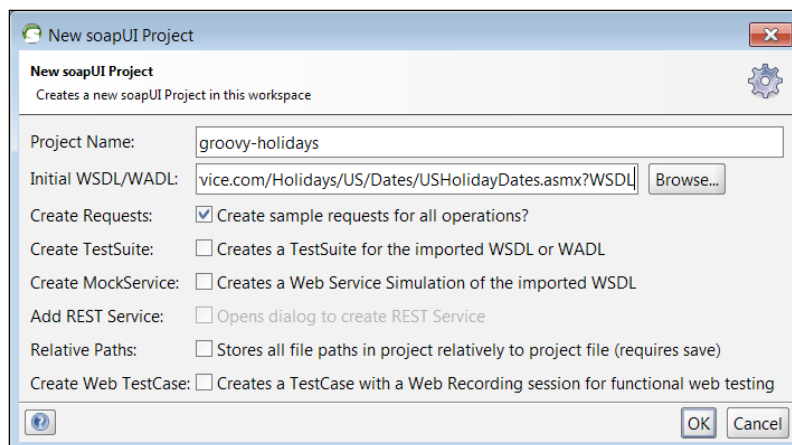
Getting ready

Download and install the open source version of soapUI, available on the product's page website (<http://www.soapui.org/>). Note that this recipe will also work with the Pro version of the product, should you already own a license. At the time of writing, the latest soapUI version is v4.5.2.

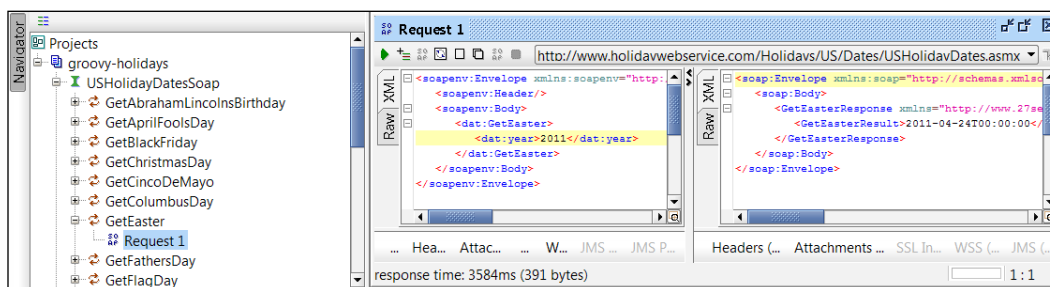
How to do it...

Following the soapUI installation, you should launch it in order to proceed with the following steps:

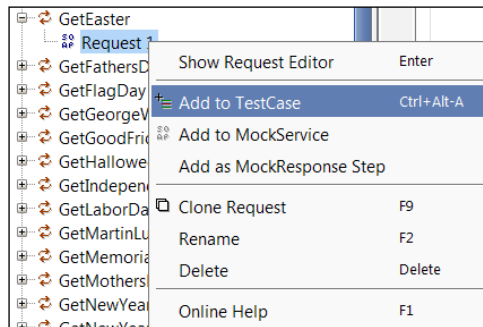
1. From the workspace, create a new soapUI project (**File | New soapUI project**). Assign a name to the project and fill the WADL/WSDL input with the base service URL, `http://www.holidaywebservice.com`, concatenated with the WSDL path - `/Holidays/US/Dates/USHolidayDates.asmx?WSDL`. Leave all the other options as default:



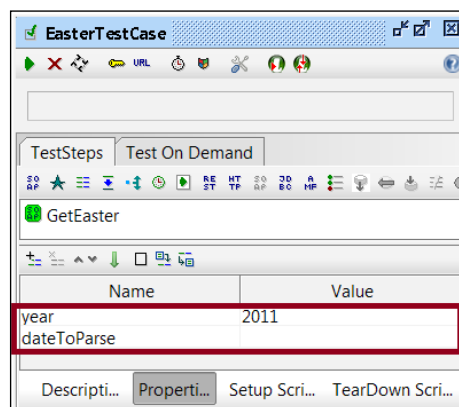
2. soapUI connects to the web service and fetches all the available operations, creating sample requests for each operation. Expand the `GetEaster` operation on the left pane (`USHolidayDateSoap` interface) and double-click on **Request 1**. A SOAP request pane should appear: replace the `? symbol` inside the `<dat:year>` node with a valid year (for example, 1977). Fire the request by clicking on the big green arrow and enjoy the response from the server:



- Now, it's finally time to create the test. Right-click on the **Request 1** element below the **GetEaster** operation and from the pop-up menu, select **Add to TestCase**:



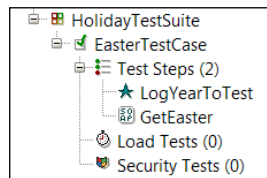
- soapUI requests the name for the test suite, a container for all our test cases: use any meaningful name, for example, `HolidaysTestSuite`. Once we get the test suite name sorted, we will be prompted with the name of the test case; we can fill in the input with `EasterTestCase`.
- Lastly, soapUI will ask for the test method name; we already selected `GetEaster` so we can leave all the options as default. After clicking on **OK**, a new `EasterTestCase` will be added to the tree in the left pane. Expand the node and double-click on `GetEaster` to pull up our test case console. We will use Groovy to process the input and the output of the request by creating more steps.
- From the test console, select the properties tab and click on the small **+** button to add a new property. Name it `year` and set its value to `2011`. Create another property called `dateToParse` and leave its value empty for the moment, we will use it later.



- The `year` property that we just created replaces the input for our SOAP request. Double-click on the **GetEaster** node from the left pane, under **Test Steps**, to show the XML request. Delete any content between the `<dat:year>` tags, click on your mouse's right button, select **Get Data** and **Test Case:[Get Easter], Property[year]**. This will update the request's XML and place a snippet between the `<dat:year>` tags that retrieves the year's value from the property created previously:



- Let's add some Groovy code to the test. Select the **Test Steps** node, click the mouse's right button, select **Add Step** and **Groovy Script**. Name it `LogYearToTest`. If the new step shows up below the **GetEaster** node, drag it above it.



- Double-click on the newly created Groovy step and insert the following Groovy snippet in the code editor:

```
def testCase = testRunner.testCase
def year = testCase.getPropertyValue('year')
log.info("testing with data: ${year}")
```

The console comes with some injected components such as `log` and `testRunner`, which is a controller for our test case. The code above simply extracts the year property we created before and logs it.

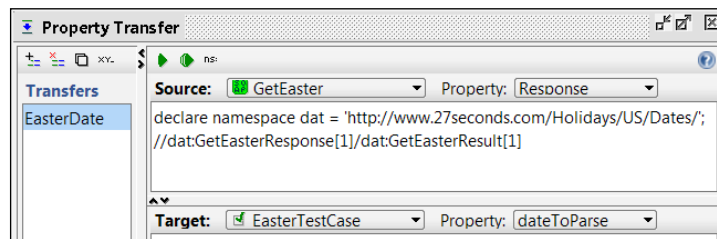
- Let's create a new test step, this time of type **Property Transfer**. This type of test step is required to transfer values between steps. Name it and double-click on the newly created node. The window is roughly divided into two sections: the source (from which we extract the result) and target (the property in which the result will be copied).

- Click on the small **+** icon in the transfer step window and create a new variable name, `EasterDate`. From the **Source** drop-down, select the **GetEaster** SOAP step. Select **Response** from the **Property** drop-down. Finally, add the following XPath snippet in the large text box area below the **Source** drop-down:

```
declare namespace dat =
    'http://www.27seconds.com/Holidays/US/Dates/';
//dat:GetEasterResponse[1]/dat:GetEasterResult[1]
```

The snippet is a simple XPath that identifies the node that is extracted from the response.

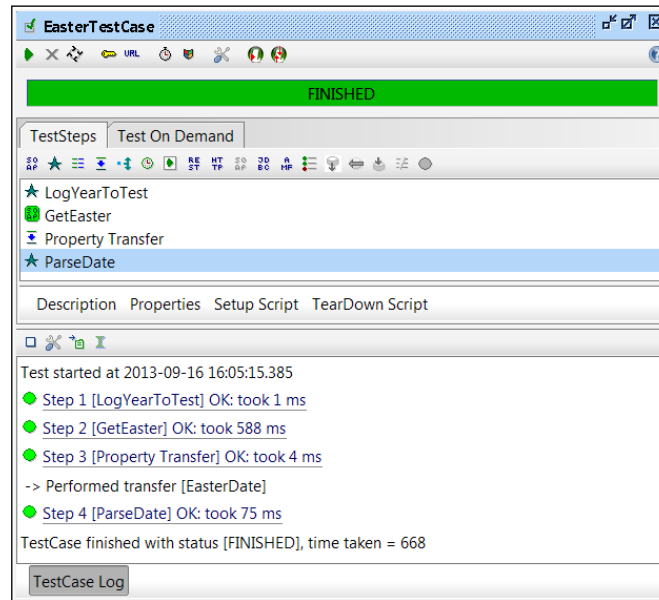
- In the **Target** drop-down, select the `EasterTestCase` test case and pick `dateToParse` in the **Property** drop-down (remember, we created this property at the beginning of the recipe):



- Now, it's time to create the last step of our test case. Under the **Property Transfer** step, let's create a second **Groovy Script** step with the name `ParseDate` and add the following Groovy code:

```
import static java.util.Calendar.*
def testCase = testRunner.testCase
def dateStr = testCase.getPropertyValue('dateToParse')
def date = Date.parse("yyyy-MM-dd'T'HH:mm:ss", dateStr)
int parsedYear = date.toCalendar().get(YEAR)
int currentYear = new Date().toCalendar().get(YEAR)
assert (currentYear - parsedYear == 2)
```

14. You can now run the full test case by double-clicking on the **EasterTestCase** node in the left pane and clicking on the green arrow on the windows that appears. If the test passes, the green dots sitting next to each step should be green. Otherwise, you should see some red dots and a description of the error.



How it works...

soapUI is essentially a test tool. The goal of the recipe was to create a test case for one of the operations, specifically the `GetEaster` method. The method accepts a single parameter, the year, and returns the actual Easter day and month for the given year.

The test case we just created is composed of four steps:

- ▶ The first step logs the value of the property we use in the SOAP request
- ▶ The second step is the actual SOAP request
- ▶ The third step extracts the date value from the SOAP response and transfers it to a property named `dateToParse` using XPath
- ▶ The last step uses Groovy to parse the date returned from the web service and verify that the year returned is 2 years behind the current year

There's more...

With this recipe, we have only scratched the surface of the powerful possibilities of using soapUI with Groovy. It is possible, for instance, to read data from a file or spreadsheet and populate the requests with different information (for example, usernames, passwords, identifiers). More sophisticated assertions can be done using database data to verify that the returned information is correct.

See also

- ▶ <http://www.soapui.org>
- ▶ <http://www.holidaywebservice.com>
- ▶ <http://groovy.codehaus.org/groovy-jdk/java/util/Date.html>
- ▶ <http://www.w3.org/TR/xpath20/>

Using JMeter and Groovy for load testing

Apache JMeter is an open source application, designed to test the performance of client/server applications such as web applications, web service based APIs, or even FTP applications. Thanks to its ease-of-use and wide set of features, JMeter is a very popular testing tool. It is written in Java and exposes API for added extensibility and flexibility.

JMeter acts as the client of the service that has to be tested for performance metrics. It spawns one or more concurrent requests and measures the response time of the server under test. The metrics are not limited to response time, but JMeter can also monitor other resources such as CPU loads or memory usage. By leveraging its scripting capabilities, which we will discuss in this recipe, JMeter can be used effectively as a tool for functional test automation.

In the course of this recipe, we will learn how to execute a Groovy script from JMeter to manipulate and present the information returned from the test. We are going to make several requests to a web service that returns a random joke about *Chuck Norris* in a JSON format. For example:

```
{
  "type": "success",
  "value": {
    "id": 101,
    "joke":
      "Chuck Norris can cut through a hot knife with butter.",
    "categories": [
      "nerdy"
    ]
  }
}
```

We will use Groovy to calculate the frequency of words for each joke. For this recipe, we assume that the reader has some familiarity with JMeter (...and *Chuck Norris!*).

Getting ready

Before starting with the recipe, download the latest version of JMeter (v2.9 at the time of writing) from the product website: https://jmeter.apache.org/download_jmeter.cgi. Unzip the archive and copy the following JAR files from the Groovy 2 distribution folder to the `lib/ext` folder inside the JMeter installation:

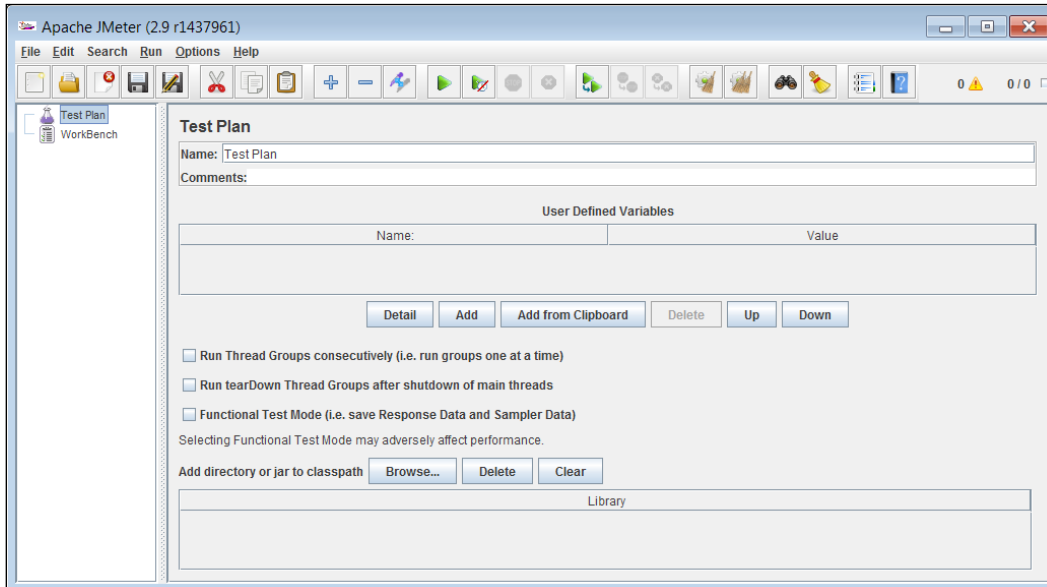
- ▶ `groovy-all-2.1.x.jar`
- ▶ `groovy-jsr223-2.1.x.jar`
- ▶ `asm-4.0.jar`
- ▶ `asm-analysis-4.0.jar`
- ▶ `asm-commons-4.0.jar`
- ▶ `asm-tree-4.0.jar`
- ▶ `asm-util-4.0.jar`
- ▶ `antlr-2.7.7.jar`

This step will enable the Groovy integration in JMeter.

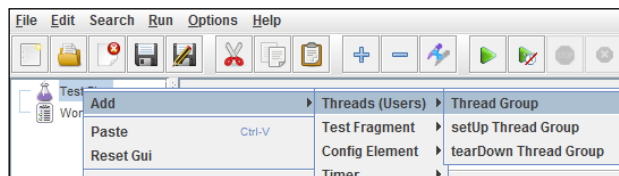
How to do it...

To start JMeter, execute the `jmeter.sh` script on Unix or OSX, or the `jmeter.bat` script under Windows.

1. Let's start with a fresh new project:



2. Right-click on the **Test Plan** element and select the **Add | Threads (Users) | Thread Group**:



- The **Thread Group** allows the configuration of concurrent repetition of child elements. Let's set up the **Thread Group** to run only one thread with the **Loop Count** value set to 50:

The screenshot shows the 'Thread Group' configuration window. It has a title bar 'Thread Group'. Below the title bar, there is a 'Name' field containing 'Thread Group' and a 'Comments' field. A section titled 'Action to be taken after a Sampler error' contains five radio buttons: 'Continue' (selected), 'Start Next Thread Loop', 'Stop Thread', 'Stop Test', and 'Stop Test Now'. Below this is a section titled 'Thread Properties' with four input fields: 'Number of Threads (users):' set to '1', 'Ramp-Up Period (in seconds):' set to '1', 'Loop Count:' with a 'Forever' checkbox and a text field set to '50', and a 'Delay Thread creation until needed' checkbox. At the bottom, there is a 'Scheduler' checkbox.

- Right-click on the **Thread Group** element and select the **Add | Sampler | HTTP Request** menu item.
- Configure the **Sampler** to make the requests to our quote service. Change the **Server name or IP:** field to `api.icndb.com` and the **Path:** field to `/jokes/random`:

The screenshot shows the 'HTTP Request' configuration window. It has a title bar 'HTTP Request'. Below the title bar, there is a 'Name' field containing 'HTTP Request' and a 'Comments' field. A section titled 'Web Server' contains two input fields: 'Server Name or IP:' set to 'api.icndb.com' and 'Port Number:'. To the right, a 'Timeouts (milliseconds)' section contains two input fields: 'Connect:' and 'Response:'. Below this is a section titled 'HTTP Request' with four input fields: 'Implementation:' (a dropdown menu), 'Protocol [http]:', 'Method:' set to 'GET' (a dropdown menu), and 'Content encoding:'. Below this is a 'Path:' field set to '/jokes/random'. At the bottom, there are five checkboxes: 'Redirect Automatically' (unchecked), 'Follow Redirects' (checked), 'Use KeepAlive' (checked), 'Use multipart/form-data for POST' (unchecked), and 'Browser-compatible headers' (unchecked).

- Right-click on the **HTTP Request** and select **Add | Post Processors | JSR223 Postprocessors**.

7. From the **Language** drop-down, select **Groovy** and in the script text area add the following code:

```
import groovy.json.JsonSlurper
import groovy.transform.*

@synchronized
def calculateFrequency() {

    def frequency = vars.getObject('frequency') ?: [:]

    def res = prev.responseDataAsString
    println prev.isStopTest()
    def slurper = new JsonSlurper()
    def result = slurper.parseText(res)

    def joke = result.value.joke
    def words = joke.split(' ')

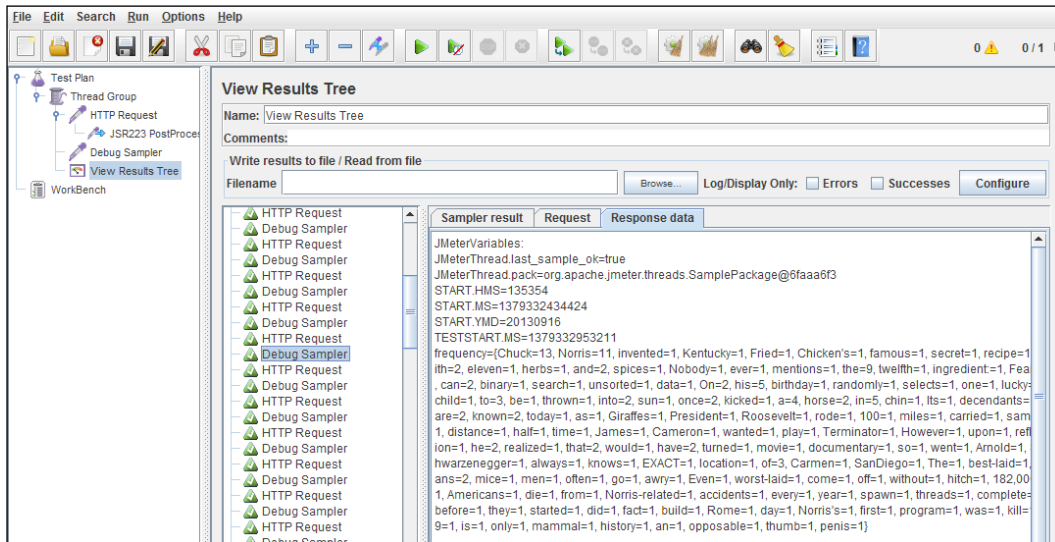
    words.each {
        def w = it.replaceAll( /[,;?.?]*$/, '' ).
            replaceAll( /&quot;/, '' )
        def num = frequency.get(w)
        num ? frequency.put(w, num + 1) : frequency.put(w, 1)
    }

    vars.putObject('frequency', frequency)
}

calculateFrequency()
```

8. Right-click on the **Thread Group** again and select **Add | Sampler | Debug Sampler**. This sampler is used to keep a track of the test variables across the requests.
9. Finally, right-click on the **Thread Group** once again and select **Add | Listener | View Result in Tree**.
10. Select the **View Result Tree** node and launch the test plan by clicking on the green arrow on the top bar.

11. Select any **Debug Sampler** item in the list and click on **Response data**. The **Response data** pane contains a list of JMeter variables, including the variable frequency populated by the Groovy script and the list of words extracted from the jokes and their frequency:



How it works...

Due to the complexity of JMeter, we cannot dig too deep into the large number of components that can be composed to achieve very sophisticated functional and stress tests.

In the JMeter configuration of this recipe, we have three components:

- ▶ An **HTTP Sampler** that runs the requested number of HTTP `GET`, either sequentially or concurrently (that depends on the thread group configuration and the **Number of Threads (users)** property's value).
- ▶ A **JSR223 Post Processor** that is applied after each HTTP request, and that is configured to run the Groovy script displayed at step 7. The JSR-223 is the specification for the Java SE scripting API. It was created as a part of **JCP (Java Community Process)** and added to the core of Java.
- ▶ A **Debug Sampler**, which we use to display the content of the variable populated by the scrip.

The Groovy script that is executed at each HTTP request accesses the global `vars` variable to retrieve a Map containing the word frequency data:

```
def frequency = vars.getObject('frequency')
```

A script can stick any kind of object into the `vars` variable. After figuring out if the `Map` exists or has to be created, the script accesses another variable, named `prev`.

```
def res = prev.responseDataAsString
```

`prev` returns an object of type `org.apache.jmeter.samplers.SampleResult` that is the result of the sample—in our case the HTTP response returned from the HTTP `GET`. We know that the response contains a JSON stream, so we parse the JSON payload, and we run a basic frequency algorithm that splits the words by space and removes some unwanted characters from each word using regular expressions (see the *Searching strings with regular expressions* recipe in *Chapter 3, Using Groovy Language Features*). Finally, the `Map` containing the frequency data is put back into the global `vars` variable.

The script is actually one function annotated with the `@Synchronized` keyword. The annotation creates a lock variable, and the code is synchronized on that lock variable. The synchronization is required because JMeter is normally configured so that several threads simulate realistic user activity. Therefore, we don't want to incur data consistency problems when accessing the global variable.

See also

- ▶ <https://jmeter.apache.org/>
- ▶ <http://www.jcp.org/en/jsr/detail?id=223>
- ▶ The *Embedding Groovy into Java* recipe in *Chapter 2, Using Groovy Ecosystem*

