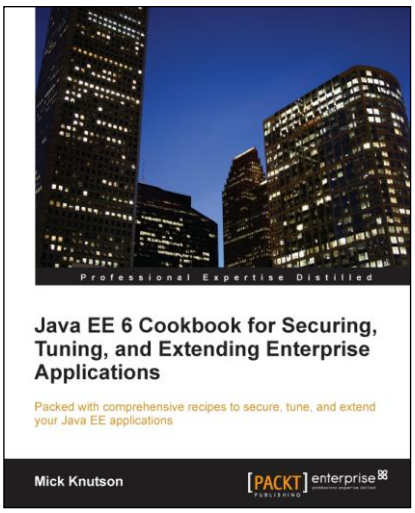
[PACKT] PUBLISHING enterprise professional expertise distilled

Java EE 6 Cookbook for Securing, Tuning, and Extending Enterprise Applications

Mick Knutson



Chapter No. 4
"Enterprise Testing Strategies"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.4 "Enterprise Testing Strategies"

A synopsis of the book's content

Information on where to buy this book

About the Author

Mick Knutson, with nearly two decades of experience working in the IT industry in various roles as Enterprise technology consultant, Java Architect, project leader, Engineer, Designer and Developer, has gained a wide variety of experience in disciplines including Java EE, Web Services, Mobile Computing, and Enterprise Integration Solutions.

Over the course of his career, Mr. Knutson has enjoyed long-lasting partnerships with many of the most recognizable names in the Health Care, Financial, Banking, Insurance, Manufacturing, Telecommunications, Utilities, Product Distribution, Industrial, and Electronics industries employing industry-standard full software lifecycle methodologies, including the Rational Unified Process (RUP), Agile, SCRUM, and Extreme Programming (XP).

Mr. Knutson has led training courses and book publishing engagements, authored technical white papers, and presented at seminars worldwide. As an active blogger and Tweeter, Mr. Knutson has also been inducted in the prestigious DZone.com "Most Valuable Blogger" (MVB) group, and can be followed at `http://baselogic.com`, `http://dzone.com/users/mickknutson` and `http://twitter.com/mickknutson`.

Mr. Knutson is exceptional at team building and motivating both at a peer-to-peer level and in a leadership role. He demonstrates excellent communications skills and the ability to adapt to all environments and cultures with ease.

Mr. Knutson is President of BASE Logic, Inc., a software consulting firm focusing on Java-related technologies and development practices, and training for enterprise development.

Mr. Knutson has been a strategic member of Comcast, for Wayne Ramprashad, helping to design and deploy the next generation IVR to align the One Customer Experience and deflect millions in quarterly operational costs. This opportunity helped foster many real world challenges and solutions used indirectly in many of the recipes included in this book.

Java EE 6 Cookbook for Securing, Tuning, and Extending Enterprise Applications

Java Platform, Enterprise Edition is a widely used platform for enterprise server programming in the Java programming language.

This book covers exciting recipes on securing, tuning, and extending Enterprise Applications using a Java EE 6 implementation.

The book starts with the essential changes in Java EE 6. Then we will dive into the implementation of some of the new features of the JPA 2.0 specification, and look at implementing auditing for relational data stores. There are several additional sections that describe some of the subtle issues encountered, tips, and extension points for starting your own JPA application, or extending an existing application.

We will then look into how we can enable security for our software system using Java EE built-in features as well as using the well-known Spring Security framework. We will then look at recipes on testing various Java EE technologies including JPA, EJB, JSF, and web services.

Next we will explore various ways to extend a Java EE environment with the use of additional dynamic languages as well as frameworks.

The book then covers recipes that touch on the issues, considerations, and options related to extending enterprise development efforts into mobile application development.

At the end of the book, we will cover managing Enterprise Application deployment and configuration, and recipes that will help you debug problems and enhance the performance of your applications.

What This Book Covers

Chapter 1, Out with the Old, In with the New: This chapter is not a tutorial or primer on the various specifications, but rather aimed at giving a high-level summary of the key changes in the Java EE 6 release. The focus will be directed on how these new features will simplify your development, as well as how to improve your application performance.

Chapter 2, Enterprise Persistence: In this chapter, we will dive into the implementation of some of the new features of the JPA 2.0 specification, and look at implementing auditing for relational data stores. There are also several additional sections that describe some typical issues encountered, further tips, and extension points for starting your own JPA application, or extending an existing application.

Chapter 3, Security: In this chapter, we will look into how we can enable security for our software system using Java EE built-in features as well as using the well-known Spring Security framework, which is a widely accepted framework for more fi ne-grained security implementation.

Chapter 4, Enterprise Testing Strategies: This chapter covers a wide range of testing techniques to employ in the Enterprise. We cover testing-related recipes for testing various Java EE technologies, including JPA, EJB, JSF, and web services.

Chapter 5, Extending Enterprise Applications: In this chapter, we will explore various ways to extend a Java EE environment with the use of additional dynamic languages as well as frameworks.

We start with a recipe using Groovy as a dynamic language integrating to existing Java code, then move to examples with Scala, followed by a recipe to integrate AspectJ aspect weaving into an existing application.

We will then end this chapter with two standard Java EE 6 extensions, the Decorator and Interceptor. These are new CDI features that have similar capability and extensibility as we might get from Aspects.

Chapter 6, Enterprise Mobile Device Integration: This chapter will cover recipes that touch on the issues, considerations, and options related to extending Enterprise development efforts into mobile application development.

Chapter 7, Deployment and Configuration: In this chapter, we will cover issues and solutions to application configuration. The solutions described will cover the use of standard Java EE APIs to access external properties files, as well as Groovy-based configuration scripts.

Advanced configuration topics will be covered using the Java Management Extensions (JMX) including detailed configuration and recipes explaining the use of tools to connect to a JMX service.

This chapter will also cover tools to aid in rapid and hot-deployment of Java EE applications through a development IDE or existing build tool such as Apache Ant or Apache Maven.

Chapter 8, Performance and Debugging: This chapter consists of recipes for solving issues related to the performance and debugging of Java EE applications. The solutions described will help in understanding performance-related issues in a Java EE application and ways to identify the cause. Performance topics that will be covered include profiling application memory, TCP connections, server sockets, and threading-related problems that can face any Java application.

This chapter will also cover how to leverage tools for debugging web service payloads as well as ways to extend the capabilities of those tools. Additionally, we will cover leveraging tools to debug network-related issues, including profiling TCP, HTTP, and HTTPS-based connections. We finish the chapter by leveraging tools for application server monitoring to get a better understanding of the health and performance of a live application and the server it runs on.

4
Enterprise Testing Strategies

In this chapter, we will cover:

- Remote debugging of Java EE applications
- Testing JPA with DBUnit
- Using Mock objects for testing
- Testing HTTP endpoints with Selenium
- Testing JAX-WS and JAX-RS with soapUI

Introduction

This chapter is going to cover a wide range of testing techniques to employ in the enterprise. We cover testing-related recipes for testing various Java EE technologies, including JPA, EJB, JSF, and web services.

Remote debugging of Java EE applications

Most Integrated Development Environments have the ability to debug Java applications. Most debugging is done locally while the application is being developed and while unit testing. This is a useful practice, but sometime issues arise when running applications on servers outside the development sandbox. These issues can be caused for various reasons and are usually not reproducible on a local development machine. In these cases, having the ability to debug through an application running on a target remote machine is the only way to deduce application issues.

In this recipe we are going to learn how to attach a remote debugger process to a Maven build running outside of the IDE.

Getting ready

Maven has debugging capabilities built into it as of version 2.0.8. The easiest way to start Maven in debug mode is to set the `surefire debug` option parameter and run your tests:

```
mvn -Dmaven.surefire.debug test
```

The default port will be 5005, and any IDE can attach to it.

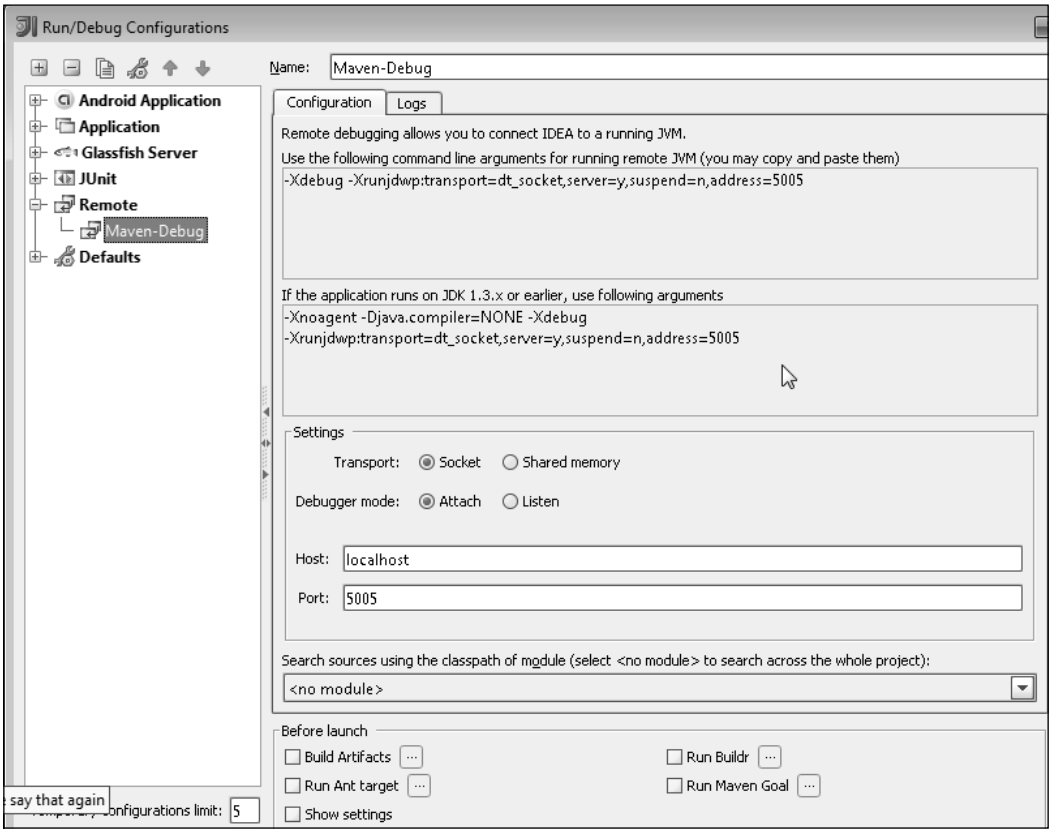
Another option is to explicitly set the debug properties. This is especially helpful if you want to change the debug ports the IDE needs to attach to:

```
mvn -Dmaven.surefire.debug="-Xdebug
  -Xrunjdwp:transport=dt_socket,
  server=y,
  suspend=y,
  address=8000
  -Xnoagent -Djava.compiler=NONE" test
```

After executing Maven with the debug flag enabled, the Maven process opens a debug port. it will appear as though the Maven build has paused, waiting for something to attach to that debug port. The build will not continue unless a debug process connects to the opened debug port. In the following case you see that port 5005 is the socket that Maven is listening to for debug connections to:

```
[INFO] --- maven-surefire-plugin:2.7.1:test (default-test) @ ch03 ---
[INFO] Surefire report directory: C:\usr\SYNCH\PACKT\3166\Chapters_
Code\ch03\target\surefire-reports
Listening for transport dt_socket at address: 5005
```

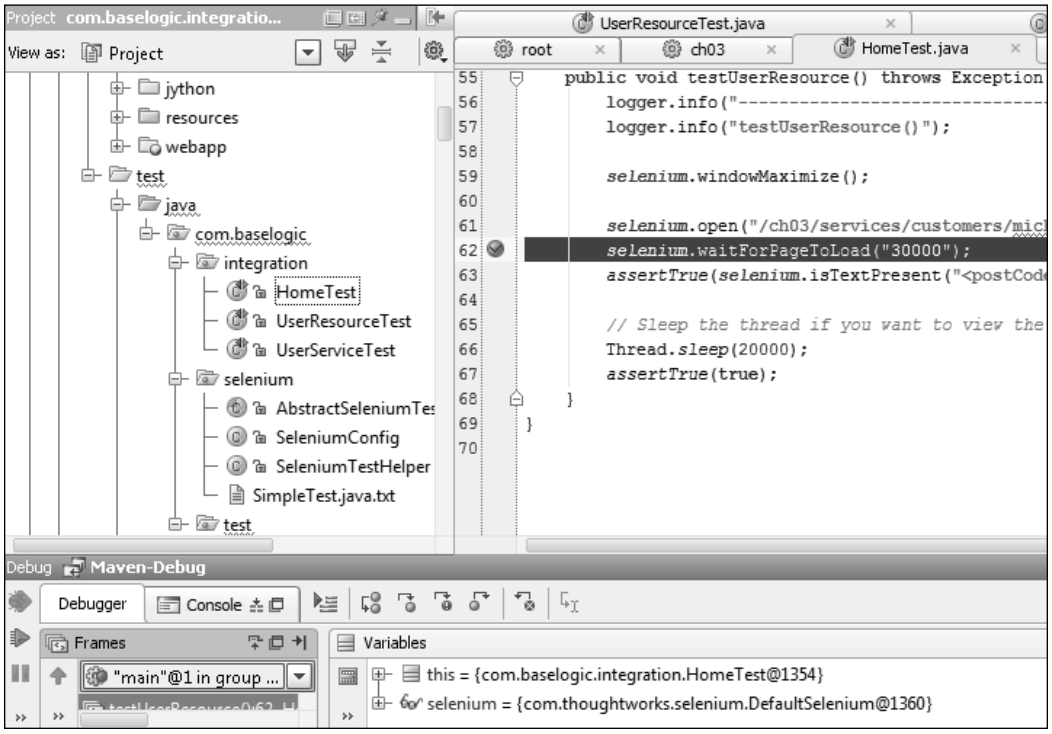
At this point, we will open our IDE and create a new, remote-run configuration which will attach to the port that Maven is listening on:



How to do it...

Now that we have created a remote-run configuration, we can debug that configuration. As soon as the IDE attaches to the port that Maven is listening on, Maven will continue the build process.

As Maven continues the build process and gets to the portion of code that you have set as a breakpoint, you will notice that Maven will stop and your breakpoint will be active for the session you're running currently:



Remote debugging not only works through the Maven test phase, as done in this section, but also works through the entire Maven build lifecycle. This method is extremely helpful when deploying a web application to an embedded container, so that you can debug a running web page within a local build.

How it works...

Remote debugging uses the **Java Platform Debugger Architecture** (**JPDA**) in order to broker information from a running **virtual machine** (**VM**) and a debugging tool, usually an IDE capable of interacting with the **Java Debug Wire Protocol** (**JDWP**).

Adding JVM debug options with Ant

When using Ant to build and run an application, you can add JVM arguments to the `<java>` Ant task to add debugging settings to the running JVM as seen in the following listing:

```
<java fork="on"
      failonerror="true"
      classpath="com.baselogic"
      classname="SomeClass">
   <jvmarg line="-Xdebug
      -Xrunjdwp:transport=dt_socket,
      server=y,
      suspend=y,
      address=4000
      -Xnoagent -Djava.compiler=NONE" />
    <arg line="--arg1 arg2 --arg3 arg4"/>
</java>
```

Starting Gradle in debug mode

Gradle is a build system that uses Groovy to script the build that feels very intuitive for Java developers.

If you are running Linux, you can simply export `GRADLE_OPTS`, as shown in the following code:

```
export GRADLE_OPTS="-Xdebug
    -Xrunjdwp:transport=dt_socket,
    server=y,
    suspend=y,
    address=8000
    -Xnoagent -Djava.compiler=NONE"
```

If you are running Windows, you can simply export `GRADLE_OPTS`, as shown here:

```
set GRADLE_OPTS="-Xdebug
    -Xrunjdwp:transport=dt_socket,
    server=y,
    suspend=y,
    address=8000
    -Xnoagent -Djava.compiler=NONE"
```

Adding debug options to JAVA_OPTS

Besides running in a build system that is starting the Java processes for you, there are instances when you may want to add debugging options to all the running processes that might take advantage of remote debugging.

If you are running Linux, you can simply export `GRADLE_OPTS`, as shown here:

```
export JAVA_OPTS="-Xdebug
    -Xrunjdwp:transport=dt_socket,
    server=y,
    suspend=y,
    address=4000
    -Xnoagent -Djava.compiler=NONE"
```

If you are running Windows, you can simply export `JAVA_OPTS`, as shown in the following code snippet:

```
set JAVA_OPTS="-Xdebug
    -Xrunjdwp:transport=dt_socket,
    server=y,
    suspend=y,
    address=4000
    -Xnoagent -Djava.compiler=NONE"
```

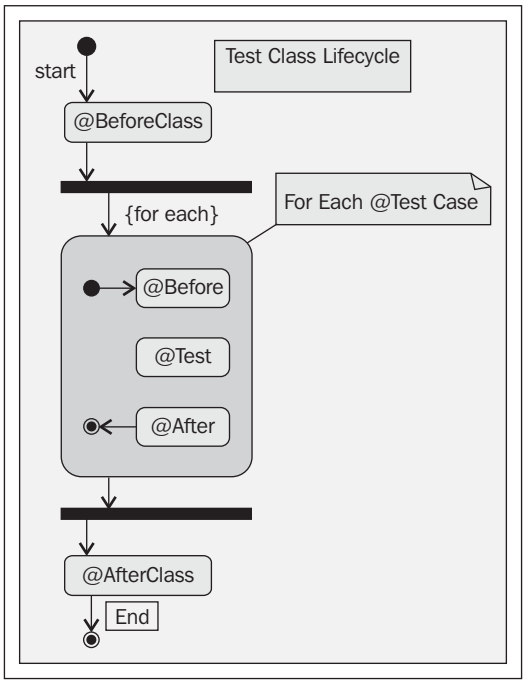
See Also

- *IDEA Debugging*: `http://www.jetbrains.com/idea/documentation/howto_02.html`
- *Jetty Debugging*: `http://docs.codehaus.org/display/JETTY/Debugging+Jetty+with+IntelliJ+IDEA`
- *Maven Debugging*: `http://maven.apache.org/plugins/maven-surefire-plugin/examples/debugging.html`
- *Java Platform Debugger Architecture (JPDA)*: `http://java.sun.com/javase/technologies/core/toolsapis/jpda/index.jsp`
- *JPDA Architecture*: `http://java.sun.com/javase/technologies/core/toolsapis/jpda/index.jsp`

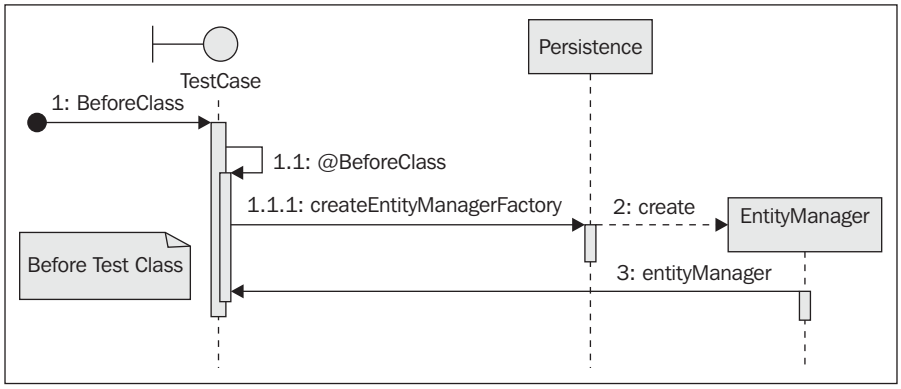
Testing JPA with DBUnit

In *Chapter 2*, *Enterprise Persistence*, we touched on some examples where we were testing our JPA examples. In this recipe, we will take a step-by-step approach on how to use JUnit and DBUnit in your Java EE application. First, I want to review each tool, and the benefits it will provide for this recipe.

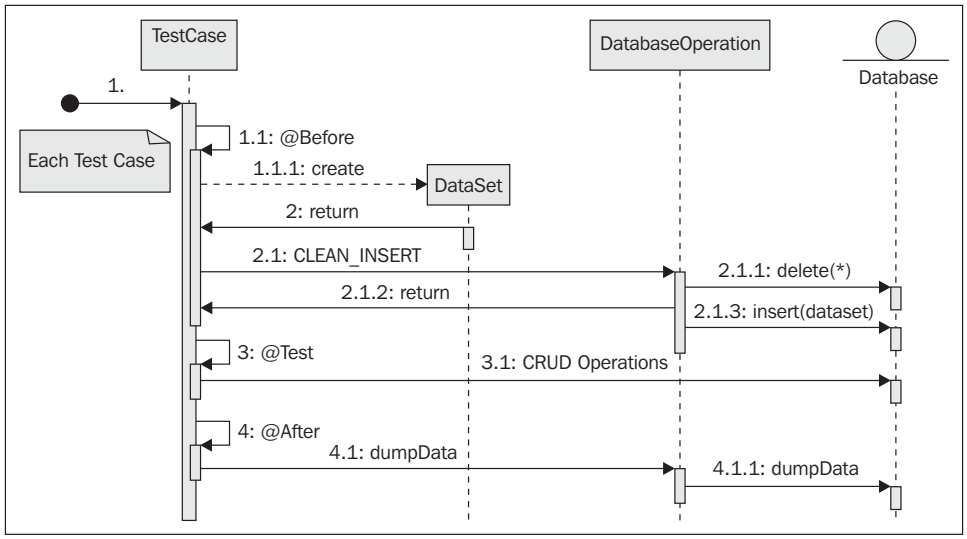
The JUnit lifecyle is a fixed series of method calls to ensure consistency when running unit tests. In the following screenshot, we can see the lifecycle that BDUnit and any other unit tests will follow:



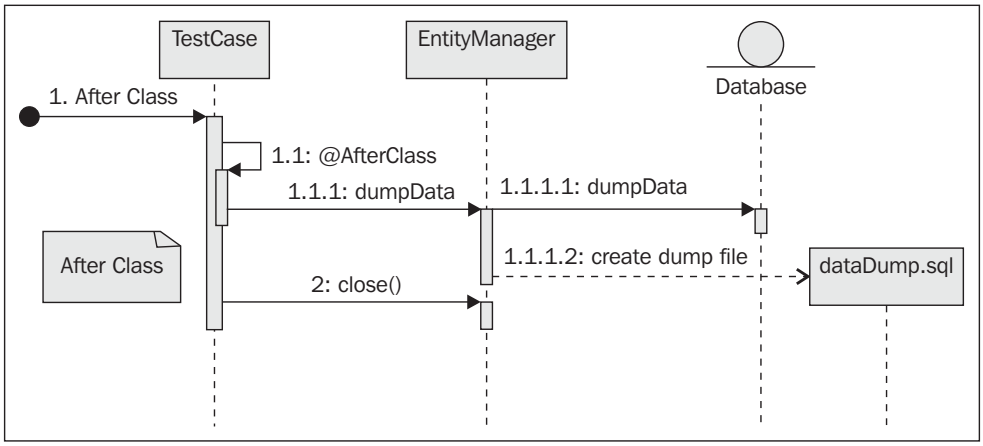
The lifecycle of all unit test classes is exactly the same. You can perform processing before and after the class is created, with additional processing before and after each and every test case:



This allows you to instantiate costly objects such as entity managers and database connections once, and share them for all tests cases:



For each test case that is run, you must ensure you have the same starting point as all the other tests. This means that your tests will always be repeatable, no matter what order they are run in. Before we run a test, we are going to clean the database, and re-insert a test dataset. This way we know what we are testing against. After we have performed pre-processing operations, we can run our test. Then, when the test is complete, we can dump the finished data into an SQL file for later review. This can be quite helpful in order to debug issues wherein your data is not what you expected. I usually write this SQL file into the build's output directory, so it is created each time I run my unit tests, and is also deleted every time I run a clean; this way it is not accidentally checked into source control:



After all the tests have run, you can perform post-processing operations to clean up loose ends. This can include closing entity manager and database connections, creating or deleting SQL dump files, or removing other objects you are finished with.

Getting ready

Getting ready to create a JUnit test and seeding a test database with DBUnit, we need to first import `junit.jar` and `dbunit.jar` into our project. In the case of this recipe, we are going to use the Maven build system:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.dbunit</groupId>
    <artifactId>dbunit</artifactId>
    <version>${dbunit.version}</version>
    <scope>test</scope>
</dependency>
```

Once we have defined the two dependencies in our `pom.xml` file to ensure we have the JUnit and DBUnit libraries, we only need to add our Surefire plugin which will run all tests:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>${plugin.surefire.version}</version>
    <inherited>true</inherited>
</plugin>
```

This simple Surefire plugin will run all tests under:

```
[project_root]/src/test/java/**
```

This is quite simple in Maven by using just the defaults, but if needed, further configuration can be made to do very complex testing patterns.

This recipe builds upon the work done in *Chapter 2*, *Enterprise Persistence*, so we are assuming we already have our domain objects created, and are basically able to create a Customer entity using EclipseLink. If you need help, the source code for *Chapter 2*, *Enterprise Persistence*, actually has the Entities, as well as the test selections for this recipe.

A key feature of EclipseLink from *Chapter 2*, *Enterprise Persistence*, is how it processes the `persistence.xml` file:

```
<property name="eclipselink.ddl-generation" value="drop-and-create-
tables"/>
```

This entry will create your database schema on startup, and drop that database on application shutdown.

DDL generation

In *Chapter 2*, *Enterprise Persistence*, and in this chapter, examples use drop-and-create tables for the DDL generation mechanism. This is only designed to be used for testing and should not be used in production environments, unless you have a specific use case to do so, because all the existing table structures and the data contained in all tables will be lost.

How to do it...

Assuming we have a valid JPA domain object created, and EclipseLink has been properly configured as per *Chapter 2*, *Enterprise Persistence*, we are now ready to start this testing recipe.

To begin, we need to create a Java class called `CustomerTest` and because we are using Maven to compile, build, and run our unit tests, we will put these calls in `src/test/java`. This is going to be a common JUnit test class, and uses annotations and signatures you are already familiar with, if you are writing JUnit test cases in your current project.

Step 1: Imports

With the advent of JUnit 4.x, and specifically 4.5+, there are several imports I like to add to each of my tests to import static references, which will make unit tests easier to read and understand the tests intentions.

The following import allows a shorter version of assertions to be used such as `assertNotNull(obj)` versus the longer version `Assert. assertNotNull(obj)`:

```
import static junit.framework.Assert.*;
```

These two imports will allow for a more readable assertion such as `assertThat( someString, is( "Success"))`.

```
import static org.junit.Assert.assertThat;
import static org.hamcrest.CoreMatchers.is;
```

Look at `org.hamcrest.CoreMatchers` for the various methods available for matching.

Step 2: Attributes

There are several attributes that are used by each test case:

```
private static EntityManagerFactory emf;
private static EntityManager em;
public static final String dataSetFile =
      "./src/test/resources/dataset.xml";
public static final String dataSetOutputFile =
      "./target/test-dataset_dump.xml";
public static final String[] nullPrimaryKeyFilters =
      {"ID", "ADDRESS_KEY", "P_NUMBER", "HOBBY_NAME"};
```

Null Primary Key Filter

An important note for using DBUnit is knowing that, when seeding data for your tests, DBUnit does not work well with tables that do not have explicit primary keys such as new `CollectionTables`. This is easily remedied by adding explicit filters to allow for null primary keys in such cases. We will use these filters in our lifecycle methods later in this chapter.

Lifecycle methods

Next, we will create our lifecycle methods that include all before and after lifecycles. These methods are run before and after every Class instantiation, or test run, which ensures consistent test conditions.

The `@BeforeClass` is called when the class is instantiated before anything else occurs:

```
@BeforeClass
public static void initEntityManager() throws Exception {
      emf = Persistence.createEntityManagerFactory
            (Constants. PERSISTENCEUNIT);
      em = emf.createEntityManager();
}
```

At this stage in the lifecycle, we can create the entity manager, which will be used for all the tests in this class. The creation of the entity manager is based on `PERSISTENCEUNIT`. This means that EclipseLink is going to create an instance of our database without any data and it will now be available to all tests.

When all the tests are complete, we need to clean up the `EntityManager` and any other objects we might have created:

```
@AfterClass
public static void closeEntityManager()
    throws SQLException {
      if (em != null) {
          em.close();
      }
```

```
        if (emf != null) {
            emf.close();
        }
}
```

Before each and every unit test, we need to ensure we have consistent data. So, we need to start by seeding our database:

```
@Before
public void initTransaction()
  throws Exception {
      TestUtils.seedData(em,
            dataSetFile,
            nullPrimaryKeyFilters);
}
```

To keep the unit test classes clean, I re-factored most of the DBUnit helper utilities into a separate TestUtils class:

```
    public static void seedData(EntityManager em,
                                String dataSetFile,
                                String... nullPrimaryKeyFilters)
            throws Exception {
1   em.getTransaction().begin();
2   Connection connection = em.unwrap(java.sql.Connection.class);

    try {
3       IDatabaseConnection dbUnitCon = new DatabaseConnection(connection);

4       dbUnitCon.getConfig().setProperty(DatabaseConfig.PROPERTY_DATATYPE_FACTORY,
                new H2DataTypeFactory());

        if (nullPrimaryKeyFilters != null && nullPrimaryKeyFilters.length > 0) {
            // Set the property by passing the new IColumnFilter
            dbUnitCon.getConfig().setProperty(
5                   DatabaseConfig.PROPERTY_PRIMARY_KEY_FILTER,
                    new NullPrimaryKeyFilter(nullPrimaryKeyFilters));
        }

        IDataSet dataSet = getDataSet(dataSetFile);
6
        DatabaseOperation.CLEAN_INSERT.execute(dbUnitCon, dataSet);
    } catch (Exception exc) {
        exc.printStackTrace();
    } finally {
        em.getTransaction().commit();
7       connection.close();
    }
    }
```

Let's go through this utility to describe in detail the important operation that is initiated:

1. Before we start any database operations, we need to begin a new database transaction.
2. DBUnit requires a `java.sql.Connection`, and we need to get a valid instance from the `EntityManager`.
3. We now create a DBUnit `IDatabaseConnection`, wrapping the `java.sql.Connection`.
4. Based on the database type you are using, we need to create a database type factory for our DBUnit operations. This allows for proper data type conversion for the database you are using.
5. If there are any DBUnit-specific properties that need to be set before the DBUnit start, we need to add them now. In this case, the addition of any Primary Key Filters is needed.
6. Based on the test data file, we need to create an `IDataSet`, and then perform `CLEAN_INSERT`.
7. At the end of seeding our database, we need to commit the transaction and close the `java.sql.Connection` we created.

<dataset>

Now we can start creating a test data to test against:

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <CUSTOMER id='100100' USERNAME="user1" FIRSTNAME="Mick"
LASTNAME="Knutson"/>

    <HOBBIES CUST_ID="100200" HOBBY_NAME="BASE-Jumping"/>
    <HOBBIES CUST_ID="100200" HOBBY_NAME="Skydiving"/>

    <PHONES AREACODE="415" PHONE_NUMBER="5551212" TYPE="WORK" CUST_
ID="100200"/>

    <CUST_ADDRESSES ADDRESS_KEY="PRIMARY" CITY="Exton"
                    POSTCODE="91335"
                    PROVINCE=""
                    STATE="PA" STREET="555 Main Street"
                    STREET2="Suite 101" TYPE="RESIDENTIAL"
                    CUST_ID="100200"/>
</dataset>
```

To create a test data to be inserted by DBUnit, we can start by creating an XML tag per domain object. In this case, there is a single `CUSTOMER` tag with all the fields we want this domain to possess. You can add additional domain objects, and have those additional objects reference parent domain objects. In this way, testing relationships can be quite easy.

<dataset> ordering

It is worth noting that the `<dataset>` domain object ordering is very important, as DBUnit processes this file from top to bottom. Thus, you must define parent objects first, and followed by child objects. The aforementioned code creates a `CUSTOMER` first, then creates the customer address object, which references `CUSTOMER` via `CUST_ID`.

Step 3: Unit testing

We now have everything in place to create unit tests that can create, read, update, and delete data from our database.

To begin database operations, a transaction must begin the initiation CRUD operations, and then commit the transaction:

```
em.getTransaction().begin();

// Creates an instance of Customer
Customer customer = CustomerFixture.createSingleCustomer();

// Persists the Customer to the database
em.persist(customer);
em.getTransaction().commit();

assertNotNull("ID should not be null", customer.getId());
```

From a testing perspective, this gives the tester ability to begin a transaction, perform some database interactions, then commit the changes that should be durable during the scope of this unit test.

There's more...

Similar to seeding data into a database for testing, you can also dump data after a unit test for review and validation:

```
IDataSet dataSet = dbUnitCon.createDataSet();

FileOutputStream fos =
     new FileOutputStream(dataSetOutputFile);
FlatXmlDataSet.write(dataSet, fos);
fos.close();
```

This is especially helpful with operations such as update timestamps, and other insert and update operations, where manual validation can be useful for debugging purposes.

Multiple databases

Sometimes, with testing, it can be useful to test against more than one database:

```
<persistence-unit name="DERBY_PU" ...>
    ...
    <properties>
        <property
            name="eclipselink.target-database"
            value="DERBY"/>
        ...
        <property
            name="javax.persistence.jdbc.driver"
            value="org.apache.derby.jdbc.EmbeddedDriver"/>
        <property
            name="javax.persistence.jdbc.url"
            value="jdbc:derby:memory:chapter02DB;create=true"/>
        <property
            name="javax.persistence.jdbc.user" value="APP"/>
        <property
            name="javax.persistence.jdbc.password" value="APP"/>
    </properties>
</persistence-unit>
```

It is easiest to create another unit test class, and then use `Persistence.createEntityManagerFactory("DERBY_UNIT")` in the `@BeforeClass` initialize, to use this alternative persistence unit.

See also

- *Chapter 2*, *Enterprise Persistence*
- *Apache Maven*: `http://maven.apache.org`
- *DBUnit*: `http://dbunit.org`
- *DBUnit Primary Key Filter*: `http://www.dbunit.org/apidocs/org/dbunit/database/PrimaryKeyFilter.html`

Using Mock objects for testing

In order to properly execute a unit test, you must be able to create an isolated unit of work that can be measured in isolation. When you are attempting to isolate portions of a Java EE application, you quickly find there are many supporting services and external systems that a Java EE application requires, but which can interfere with isolation. In order to enforce test isolation, you can introduce Stubs and Mocks into your tests. In the recipe for DBUnit testing, we saw how there can be many complexities in seeding data for external systems such as databases, to ensure consistent and reliable tests. Introducing Mock objects can aid in reducing the complexity of testing, and help foster isolated testing.

This recipe is going to focus on a pattern to utilize Mock object, or 'Mocks' in order to facilitate test isolation. There are many popular Mock frameworks such as EasyMock, JMock, and many others, but this recipe is going to focus on a framework called Mockito (`http://mockito.org`) as well as Powermock (`http://code.google.com/p/powermock/`) to add support for static and private method testing.

Getting ready

To begin this recipe, you first need to include the Mockito and Powermock JARs into your Maven project's `pom.xml` file, similar to this:

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-api-mockito</artifactId>
    <version>${powermock.version}</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.powermock</groupId>
    <artifactId>powermock-module-junit4</artifactId>
    <version>${powermock.version}</version>
    <scope>test</scope>
</dependency>
```

We are now ready to start writing Mock test cases.

How to do it...

To start from the beginning, create a test class to run our first Mockito JUnit test such as the following:

```
public class OrderMockServiceTests {}
```

The next thing we need to do is add some imports to allow easy use and readability of our tests, as shown here:

```
// Hamcrest_____
import static org.hamcrest.Matchers.*;
import static org.hamcrest.MatcherAssert.assertThat;

// JUnit_____
// use MatcherAssert.assertThat instead
//import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// Mockito_____
import static org.mockito.Matchers.any;
import static org.mockito.Mockito.*;
import org.mockito.runners.MockitoJUnitRunner;
import org.mockito.stubbing.Answer;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.invocation.InvocationOnMock;
import org.mockito.stubbing.Answer;

// PowerMock_____
import org.powermock.api.mockito.PowerMockito;
import org.powermock.core.classloader.annotations.PrepareForTest;
import org.powermock.core.classloader.annotations.
SuppressStaticInitializationFor;
import org.powermock.modules.junit4.PowerMockRunner;
```

Adding static reference to Hamcrest matchers, JUnit assertions, Mockito, and Powermock methods will allow for more legible unit tests.

First we are going to use Mockito as the JUnit runner for this test:

```
@RunWith(MockitoJUnitRunner.class)
public class OrderMockServiceTests {}
```

Next, we need to create an instance of our Mock object; creating a global instance and recreating the Mock for each test ensures our test will always have a fresh Mock:

```
@InjectMocks private OrderServiceImpl classUnderTest;

@Mock private OrderDAO supportingDao;
```

Here, we create a member variable for our class under test `OrderServiceImpl` and annotate it with `@InjectMocks` which tells Mockito that this class will need to have one or more supporting Mock objects injected into it at the beginning of the unit test.

Then, we create a member variable, annotated by `@Mock`, for the Mock that we are using for this unit test, to tell Mockito that this variable is eligible for injection into the `@InjectMocks` class under test.

We then need to tell Mockito to create an instance of the class under test, and inject all Mock objects into it before each unit test by initializing Mocks:

```
@Before
public void setup() {
    MockitoAnnotations.initMocks(this);
}
```

Now that we have set up the class under test and Mock objects, we can begin writing individual test cases.

Let's go through how we will create a simple Mock interaction between our class under test and our Mock.

First, let's understand the role of the Mock by looking at the method we want to isolate and test, in our class under test:

```
public Order placeOrder(Order order){
    return orderDao.placeOrder(order);
}
```

Here, we have a method inside our `OrderServiceImpl` class, which takes an order as a parameter, then calls `OrderDao` with that order, to place the order. The main point is that in isolation, we don't care what actually happens in `OrderDao`, all that we are concerned with is the type—if `OrderDao` accepts any, and what `orderDao` returns. With that information, we can begin to simulate the inputs and expected outputs for the Mock `OrderDao`.

Step 1:

Create an `Order` object which we expect `OrderDao` to receive when it is called:

```
// Control input
Order orderInput = new Order();
orderInput.setDescription("Mick's Order");
```

Step 2:

Create an `Order` object which we expect `OrderDao` to return when it is called:

```
// Control Sample
Order orderOutput = new Order();
orderOutput.setDescription("Someone Else's Order");
```

Step 3:

Define expected behavior for the Mock when the class under test interacts with it. We want to instruct the Mock that it should expect to be called with `Order.class`, then we expect it to return the `orderOutput` object we created as a return control sample:

```
// Create Mock Behavior
when(supportingDao.placeOrder(any(Order.class)))
    .thenReturn(orderOutput);
```

Step 4:

Now we will execute the class under test's place order method, using the `orderInput` object we created as a return control sample:

```
// execute class under test
Order resultOrder =
    classUnderTest.placeOrder(orderInput);
```

Step 5:

We can now perform assertions on the returned object to determine whether the object returned was the expected object:

```
assertThat(resultOrder.getDescription(),
    is("Someone Else's Order"));
```

We expect `orderOutput` to be the order object that was returned.

Step 6:

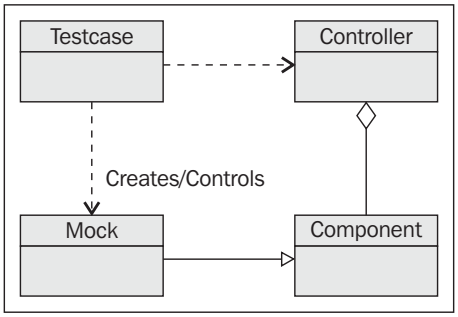
We need to verify the Mock `OrderDao` object was actually called in the manner you expected:

```
// Verify behavior for supporting were executed
verify(supportingDao).placeOrder(any(Order.class));
```

At this point we have successfully written our first Mock test case.

How it works...

What is really happening under the covers? Mockito is creating an instance of the dependent class you want to Mock, and based on the instructions on how the mock should behave, Mockito will play back the instructions you have defined:



Testing successful scenarios is usually the first task for developers; you must also be diligent about testing, this should include testing failure scenarios.

There's more...

As seen above, mocking of custom objects and objects that are not available can simplify testing, and in some cases is the only way testing in isolation is possible.

Mocking a custom object is not the only use case for Mocking and any non-final class or interface can be mocked.

Mocking all object types

Mockito is not limited to just the custom objects that you have created. You can mock any type of object, including Java EE components as shown in the following listing:

```
@Mock HttpServletRequest servletRequest;
@Mock HttpSession httpSession;
when(servletRequest.getSession())
    .thenReturn(httpSession);
```

Here we mock `HttpServletRequest` to send back a mocked HttpSession when called. This is an important concept because many Java EE objects are interfaces, and Mockito will create an implementation of the interface for you.

Simulating service delays

Another common scenario is to simulate the service that delays longer than a given period of time. You can achieve this by adding a custom answer to be returned during the mock interaction with a call to the `thenAnswer()` method:

```
when(mockService.lookupAppointment
    (any(AppointmentRequest.class)))
    .thenAnswer(delayedAnswerWithObject(response, 10000));


Future<Object> future =
    controller.queryAppointment
    ("12345", new Request());
```

You can externalize the creation of a custom `Answer` with a simple static utility:

```
public static Answer delayedAnswerWithObject
  (final Object o, final long delay) {
      return new Answer() {
          @Override
          public Object answer(InvocationOnMock invocation)
            throws Throwable {
              Thread.sleep(delay);
              return o;
            }
      };
}
```

This will simply put the current thread to sleep for a given delay.

```
@Override
public Object answer(InvocationOnMock invocation) throws Throwable {
    try {
          Thread.sleep(delay);
    }
    catch (InterruptedException e) {
          if (throwExceptions) {
```

```
                throw e;
            }
        }
        return o;
}
```

The mock answer that is created might need to catch various exceptions and, as shown in the previous code listing, are re-thrown. It is also possible to consume, or ignore these exceptions in certain situations, and because this answer is a mock, we can control object attributes without changing the class under test.

Partial Mocking

In a scenario wherein a class under test has a supporting method it calls, which you want to isolate from the test, a partial Mock that will only Mock the method described in behavior can be created.

Step 1:

Create a new instance of the class under test, using the Spy feature of Mockito, to allow Mockito to spy into an object and add custom behavior:

```
OrderServiceImpl partialMock = spy(classUnderTest);
```

Step 2:

Define the expected behavior for the partial mock:

```
when(partialMock.getMessage())
    .thenReturn("partially mocked message");
```

Again, we want to control the returned value from a Mock method call with a customer String.

Step 3:

Now we will execute the class under test's place order method using the `orderInput` object we created as a return control sample:

```
String partialResult = partialMock.getProxiedMessage();
```

Step 4:

We can now perform assertions on the returned object to determine whether the object returned was the expected object or not:

```
assertThat(partialResult, is("partially mocked message"));
```

Step 5:

We need to verify the Mock was actually called in the manner you expected:

```
verify(partialMock).getMessage();
```

Mocking exception scenarios

Instructing a Mock to throw an exception is just a matter of instructing the behavior of the Mock to throw an exception instead of returning a value type:

```
when(supportingUtils.nestedFunction())
    .thenThrow(new RuntimeException("mock Exception"));
```

Mocking methods returning void

When a method to be mocked does not return a value it is denoted as returning void, which is `Void.class`. In order to mock this type of method, you can define the Mock behavior to do nothing when it is called:

```
doNothing()
    .when(supportingUtils)
    .voidMethod();
```

You can also add behavior to have a void method throw an exception:

```
doThrow(new RuntimeException
  ("void stubbed method exception"))
      .when(supportingUtils)
      .voidMethod();
```

Multiple interactions with a Mock

Sometimes you have a Mock that will be called multiple times, and each time the Mock is called, we want the Mock to return a different value. This behavior is going to have three different values returned for three subsequent calls to the Mock:

```
when(supportingUtils.nestedFunction())
    .thenReturn("1st mocked message")
    .thenReturn("2nd mocked message")
    .thenReturn("nth mocked message");
```

The first time the Mock is called, it returns "1st mocked message", the second time it is called, it returns "2nd mocked message", and the third and subsequent time this mock is called, it returns "nth mocked message".

Ensuring Mocks called in order

Another common situation involves testing a method which makes multiple, different Mock calls and we want to ensure the Mocks get called in the correct order.

To accomplish this, we set up the Mock behavior as we normally would for our test:

```
when(supportingUtils.nestedFunction())
    .thenReturn("1st mocked message");

when(supportingUtils.nestedFunctionTwo())
    .thenReturn("Another mock being called");
```

Next, we need to create an ordered container, so Mockito can keep track of the ordered calls to the Mock:

```
InOrder inOrder = inOrder(supportingUtils);
```

Then execute the class under test:

```
String result = classUnderTest.callsFunctionInOrder();
```

Now, assert which Mocks should be called in what order:

```
// nestedFunction() should have been called first
    inOrder.verify(supportingUtils).nestedFunction();

// nestedFunctionTwo() should have been called first
    inOrder.verify(supportingUtils).nestedFunctionTwo();
```

Mocking static methods

Another tricky situation is isolating the static method calls in a method under test. Take the following listing for example:

```
public Object staticFunctions(){
    Object object;
    try {

      Object result =
        ExampleUtils.staticFunction(); // Static Call

    } catch (Exception e){
```

```
      logger.error( e.getMessage() );
    }

    return object;
}
```

We see that there is a static call which returns an object, and can also throw an Exception when called.

To mock a static method call, we need to use Powermock to provide the functionality.

It is recommended to keep all tests that require static mocks separate from the ones that only require Mockito. This is because Powermock requires more memory to process static classes than Mockito needs for mocking instance classes. This allows you to easily isolate the types of tests, and manage the test lifecycle in your build.

In order to begin, we need to decorate the unit test with `PowerMockRunner.class` instead of `MockitoJUnitRunner.class`:

```
@RunWith(PowerMockRunner.class)
@PrepareForTest(
  { ExampleUtils.class, OrderServiceImpl.class })
public class OrderPowerMockServiceTests {
...
```

The previous listing tells JUnit to use `PowerMockRunner.class` to run the entire test class of tests. Then the `@PrepareForTest` annotation tells the `PowerMockRunner.class` to prepare a list of classes to have the behavioral instructions for testing added to them. This is different than using the `@Mock` annotation because we will not have an instance of the class available for testing, so the `@PrepareForTest` annotation can be thought of as creating the Mock placeholder.

We then continue to set up Mockito Mock objects in the same way, as in the previous examples.

Let's go step-by-step to see how to mock a static call, and then look at the complete unit test:

Step 1:

We begin with a standard JUnit `@Test` method, afterwhich we need to instruct Powermock to mock the static methods present in the class we want to mock:

```
@Test
 public void staticMock() throws Exception {

PowerMockito.mockStatic(ExampleUtils.class);
```

Step 2:

Create the behavior expected when the Mock is called during the test:

```
when(ExampleUtils.staticFunction())
    .thenReturn("some static mocked value");
```

Step 3:

Execute the class under test:

```
String result = classUnderTest.staticFunctions();
```

Step 4:

Perform any assertions on the values that the class under test returns:

```
assertThat(result, is
    ("OrderServiceImpl:
    function():
    some static mocked value:
    staticFunction"));
```

Step 5:

Lastly, and most importantly, you must have Powermock verify each mocked static call before you can execute another mocked static call:

```
PowerMockito.verifyStatic();
```

This might seem like a limitation if you are confronting a method that has more than one static call in it. In such a situation, we can refactor the method under test into a more cohesive design and avoid this problem altogether.

This is what our final test method would look like:

```
@Test
public void staticMock() throws Exception {

  PowerMockito.mockStatic(ExampleUtils.class);

  when(ExampleUtils.staticFunction())
 .thenReturn("some static mocked value");

  String result =
 classUnderTest.staticFunctions();

  assertThat(result, is
 ("OrderServiceImpl:
 function():
 some static mocked value: staticFunction"));

  PowerMockito.verifyStatic();
}
```

Mocking private methods

One of the most difficult mock testing patterns is isolating private methods. Let's look at a method we want to test, which calls two different private methods:

```
public String executeInternalPrivate(){
  String result = "OrderServiceImpl: executeInternalPrivate()";
  result += ": " + privateFunction();
  result += ": " + privateFunction("privateFunction");
  return result;
}

private String privateFunction() {
  return "OrderServiceImpl: privateFunction";
}

private String privateFunction(String privateString) {
  return "OrderServiceImpl: privateFunction: " + privateString;
}
```

In our test, we want to isolate the following two private method calls:

- `private String privateFunction();`
- `private String privateFunction(String privateString);`

Using the same test class as the previous example, we already have Powermock set up and ready to run our tests. Let's go step-by-step to see how to mock the two private calls, and then look at the complete unit test:

Step 1:

We are going to create a partial mock of the class under test, which we will be mocking:

```
OrderServiceImpl classUnderTest =
  PowerMockito.spy(new OrderServiceImpl());
```

Step 2:

We will then add the expected behavior to a private class called `privateFunction`, which does not take any arguments:

```
PowerMockito.doReturn
  ("private string with no params")
  .when(classUnderTest, "privateFunction");
```

Powermock uses this String-based name for the method we intend to mock, because Powermock will use reflection at runtime to see this private method, and then add the desired behavior.

We also add behavior to a private method which takes any String as an argument:

```
PowerMockito.doReturn
  ("some altered private string")
  .when(classUnderTest, "privateFunction", anyString());
```

Step 3:

Execute the class under test:

```
String result =
  classUnderTest.executeInternalPrivate();
```

Step 4:

Perform any assertions on the values that the class under test returns:

```
assertThat(result, is
  ("OrderServiceImpl:
  executeInternalPrivate():
  private string with no params:
  some altered private string"));
```

Step 5:

Unlike verifying static calls, where we are required to verify one call at a time, mocking a private method is the same as mocking an instance method where we can ask Powermock to verify whether the private method was called:

```
PowerMockito.verifyPrivate
  (classUnderTest, times(1))
  .invoke("privateFunction");
```

We also can verify whether private methods were called with different arguments:

```
PowerMockito.verifyPrivate
  (classUnderTest, times(1))
  .invoke("privateFunction", anyString());
```

The following is what our final test method would look like:

```
@Test
public void privatePartialMock()
  throws Exception {

     OrderServiceImpl classUnderTest =
    PowerMockito.spy(new OrderServiceImpl());

      // use PowerMockito to set up your expectation
```

```
        PowerMockito.doReturn
          ("private string with no params")
          .when(classUnderTest, "privateFunction");

        PowerMockito.doReturn
          ("some altered private string")
          .when(classUnderTest, "privateFunction", anyString());

        // execute your test
        String result =
          classUnderTest.executeInternalPrivate();

        assertThat(result, is
          ("OrderServiceImpl:
          executeInternalPrivate():
          private string with no params:
          some altered private string"));

        // Use PowerMockito.verify() to verify result
        PowerMockito.verifyPrivate
          (classUnderTest, times(1))
          .invoke("privateFunction");

        PowerMockito.verifyPrivate
          (classUnderTest, times(1))
          .invoke("privateFunction", anyString());
    }
```

As you can see, using Mocks can be a powerful tool to help increase test code coverage and isolate components.

See also

- *EasyMock*: `http://easymock.org`
- *Mockito*: `http://mockito.org`
- *Powermock*: `http://code.google.com/p/powermock/`

Testing HTTP endpoints with Selenium

In order to test web application services, such as JSP and JSF pages that are dynamically generated, you need to deploy those artefacts to a Java EE or Servlet container in order to test them. While manual testing of page content is possible, automating this task will greatly increase the development lifecycle, and it will also reduce errors in validating repeated test cycles to ensure consistent test results.

In order to automate web-based testing, you need to employ a mechanism in which a test will simulate web navigation, and input and result validation for given scenarios. The Selenium test runner comes to the rescue. Selenium allows a serial set of web integration instructions to be executed on a given URL, and asserts rules to determine the expected outcomes for a scenario or a functional path defined.

Selenium can also test HTTP endpoints, such as REST services that are local or remote. It is possible to create a suite of tests that perform deployment validation of applications for QA as well as production environment, so you have quite a bit of flexibility.

Getting ready

There are a few steps involved to integrate Selenium into your build. There are several different ways you can run Selenium. Some of the alternative possibilities are to run either an embedded application server within the build process, or have a local or remote application server running with the application under test, and then run the Selenium tests. In this recipe, we will be using an embedded application server running the application under test.

Dependencies

There are several dependencies that will be required in order to run Selenium tests. You will need JUnit for running your tests. Also, you will need `selenium-server`, which creates your remote control test server, and lastly, you will need a Selenium Java client driver, which will specifically allow you to run JUnit-based test cases.

We first start by adding the following dependencies to your Maven `pom.xml` file:

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>${hamcrest.version}</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.seleniumhq.selenium.server</groupId>
    <artifactId>selenium-server</artifactId>
    <version>${selenium.server.version}</version>
    <scope>test</scope>
</dependency>
```

```
<dependency>
    <groupId>org.seleniumhq.selenium.client-drivers</groupId>
    <artifactId>selenium-java-client-driver</artifactId>
    <version>${selenium.client.version}</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.openqa.selenium.server</groupId>
            <artifactId>selenium-server</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

This is all that's required to run Selenium.

Application server

In order for Selenium to test a web application, the application must be running in a web container; Selenium will interact with live pages in order to perform testing. To do this, we can use an embedded application server, which we will start before our integration test phase, and deploy our application to that server. Once the integration test phase is complete, we will underfloor the application and stop the server:

```
<plugin>
    <groupId>org.glassfish</groupId>
    <artifactId>
        maven-embedded-glassfish-plugin
    </artifactId>
    <version>
        ${glassfish.embedded.plugin.version}
    </version>
    <configuration>
        <app>
            ${project.build.directory}/
              ${project.build.finalName}.war
        </app>
        <port>
            ${glassfish.domain.port}
        </port>
        <contextRoot>
            ${project.build.finalName}
        </contextRoot>
        <autoDelete>true</autoDelete>
    </configuration>
    <executions>
```

```
            <execution>
                <id>start-glassfish</id>
                <phase>pre-integration-test</phase>
                <goals>
                    <goal>start</goal>
                    <goal>deploy</goal>
                </goals>
            </execution>
            <execution>
                <id>glassfish-undeploy</id>
                <phase>post-integration-test</phase>
                <goals>
                    <goal>undeploy</goal>
                    <goal>stop</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
```

JUnit

We need to configure Surefire to run Selenium tests only during the integration test phase, so we can ensure our application is ready:

```
    <plugin>
        <groupId>
            org.apache.maven.plugins
        </groupId>
        <artifactId>
            maven-surefire-plugin
        </artifactId>
        <version>
            ${plugin.surefire.version}
        </version>
        <configuration>
            <argLine>
                -Xmx256m -DuseSystemClassLoader=true
            </argLine>
            <testFailureIgnore>
                True
            </testFailureIgnore>
            <excludes>
                <exclude>**/selenium/*</exclude>
            </excludes>
        </configuration>
        <executions>
```

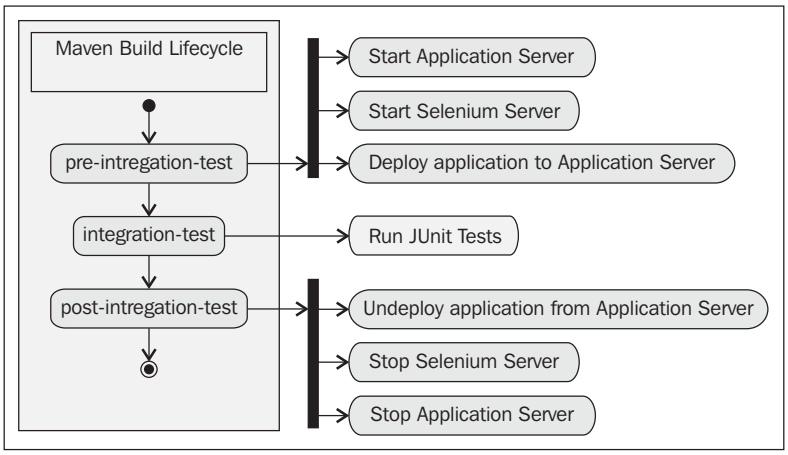
```
            <execution>
                <id>integration-tests</id>
                <phase>integration-test</phase>
                <goals>
                    <goal>test</goal>
                </goals>
                <configuration>
                    <includes>
                        <include>**/selenium/*</include>
                    </includes>
                </configuration>
            </execution>
        </executions>
    </plugin>
```

Selenium

Selenium has a great plugin to run your tests within a defined Maven build lifecycle. The goal of the plugin is to define the execution phase in which you want to run a Selenium server:

```
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>selenium-maven-plugin</artifactId>
        <version>${plugin-selenium-version}</version>
        <executions>
            <execution>
                <phase>pre-integration-test</phase>
                <goals>
                    <goal>start-server</goal>
                </goals>
                <configuration>
                    <background>true</background>
                    <logOutput>true</logOutput>
                </configuration>
            </execution>
            <execution>
                <id>stop</id>
                <phase>post-integration-test</phase>
                <goals>
                    <goal>stop-server</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
```

At this point, we have our dependencies and an application server, a Selenium server, and Surefire configured to run in a pre- and post-integration test, and during the integration test phase of a Maven build:



We can now start writing our first Selenium test cases.

How to do it...

To begin with, we need to create a Java Selenium controller which will interact with the Selenium server that needs to be started. In the following listing, we will create a helper method that will initialize the Selenium server and start the controller, making it available for remote controlling:

```
public static DefaultSelenium init() {
    logger.info("*** Starting selenium client driver ...");

    DefaultSelenium defaultSelenium =
      new DefaultSelenium
      (SeleniumConfig.getSeleniumServerHostName(),
      SeleniumConfig.getSeleniumServerPort(),
      "*" + SeleniumConfig.getTargetBrowser(),
      http://
      + SeleniumConfig.getApplicationServerHostName()
      + ":"
      + SeleniumConfig.getApplicationServerPort()
      + "/");
    defaultSelenium.start();
    return defaultSelenium;
}

public static void destroy(Selenium selenium){
    selenium.stop();
}
```

In order to create a Selenium client driver, we need to know the Selenium server hostname, port, and the target browser to be used for the testing. The Selenium client controls the target application based on the test scenarios. In the previous example, the Selenium configuration is externalized, which will allow more flexibility with configuring the Selenium client driver at runtime. Here is what the Selenium configuration looks like:

```
public class SeleniumConfig {
    private static String seleniumServerHostName = "localhost";
    private static int seleniumServerPort = 4444;
    private static String applicationServerHostName = "localhost";
    private static int applicationServerPort = 8888;
    private static String targetBrowser = "firefox";
```

For each test class, you can create one Selenium client driver and reuse it for every test you are running. To do this, we initialize the client driver in `@BeforeClass` and stop the server in the `@AfterClass` method:

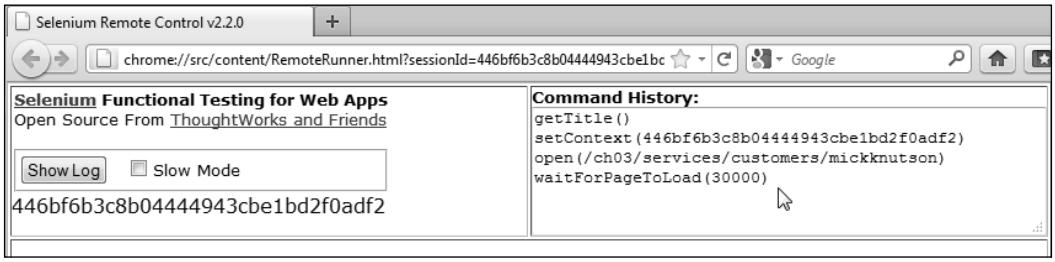
```
protected Selenium selenium;

@BeforeClass
public static void beforeClass() throws Exception{
    selenium = SeleniumTestHelper.init();
}

@AfterClass
public static void destroy(){
    SeleniumTestHelper.destroy(selenium);
}
```

Now, writing the unit test becomes very straightforward. In the following code snippet, we will open an HTTP resource or page, wait for the page to load, then assert a response containing the desired content:

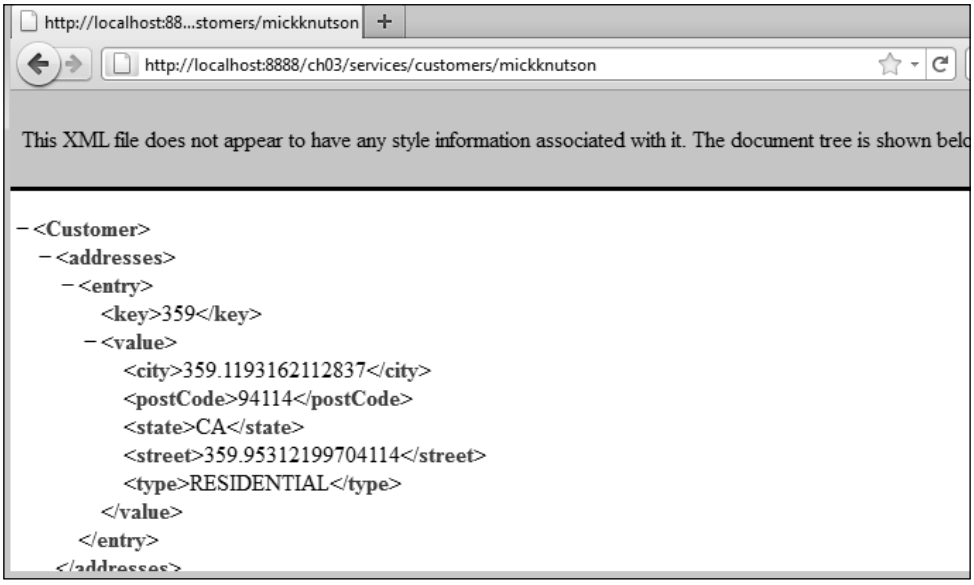
```
@Test
public void testUserResource() throws Exception {
    selenium.open("/ch03/services/customers/mickknutson");
    selenium.waitForPageToLoad("3000");
    assertTrue(selenium.isTextPresent
      ("<postCode>94110</postCode>"));
}
```

In the previous listing, we call `selenium.open()` to the application context to open the page under test. Next we call `selenium.waitForPageToLoad()` and wait 300 milliseconds to ensure the page loads completely before continuing with the test. At this point, there are a multitude of different options for validating the responses for each page, as well as controlling the next interaction.

When Selenium runs, it will open up one browser which is the test overview screen:

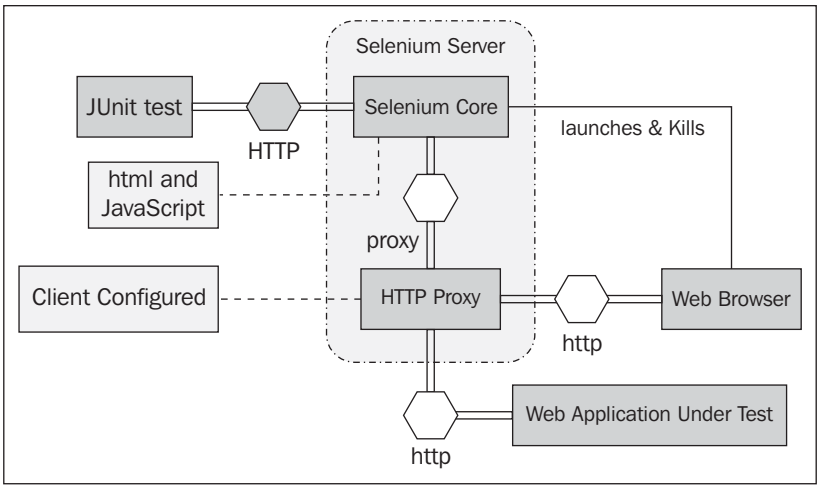


Selenium will open another browser window to execute the commands of your test:



How it works...

In order to run a Selenium test, you need to have a Selenium server running which launches and kills browsers, interprets and runs test commands from your tests, and acts as a proxy to verify the interaction between your tests and your browser commands that are executed. The following diagram shows the interaction between the unit test, the Selenium server, the web application under test, and the web browser that interacts with the test web application:



There's more...

If your goal is to test web-based applications with Selenium, Selenium IDE is a Firefox plugin that will allow you to navigate your page while recording the steps. You can then export those recordings as a Selenium test case. This can be a huge timesaver.

WebDriver integration

Selenium now supports integration with WebDriver, which is a compact tool that allows easier integration with browser-based applications. You can review migration tips and other differences at `http://seleniumhq.org/docs/appendix_migrating_from_rc_to_webdriver.html`.

There are some issues trying to validate XML passed to pages, so this will be more for HTML-based pages.

See also

- *Selenium*: `http://seleniumhq.org/`
- *Documentation*: `http://seleniumhq.org/docs/03_webdriver.html`
- *Javadocs*: `http://selenium.googlecode.com/svn/trunk/docs/api/java/index.html`
- *Selenium IDE*: `http://seleniumhq.org/docs/02_selenium_ide.html`

Testing JAX-WS and JAX-RS with soapUI

soapUI is a cross-platform function testing solution for SOAP, REST, Web, JDBC, and much more. soapUI also has the capability to test for security vulnerabilities such as XML Bomb, SQL Injection, Malformed XML-testing, and many other types of scans. soapUI allows you to create and execute any functional, regression, and loads tests in minutes.

There is an open source edition named "soapUI", and a professional edition named "soapUI Pro". The professional edition has additional wizard forms for the interface, provides test data integration, and extended reporting capabilities.

This recipe will focus on how to use soapUI as a functional testing tool. It will cover testing REST and SOAP services. This recipe will also cover creating test cases and integrating Mocks into tests. Finally, the recipe will cover extending soapUI with Groovy scripting for added functionality and logging.

Getting ready

soapUI is a Java-based application and is available with or without a **Java Runtime Engine** (**JRE**). This recipe assumes the target machine has soapUI or soapUI Pro installed and running.

soapUI works similarly to the Eclipse IDE, where the application has a workspace that can consist of one or more projects. When soapUI is started, choose a workspace location that will be used for the projects in this recipe.

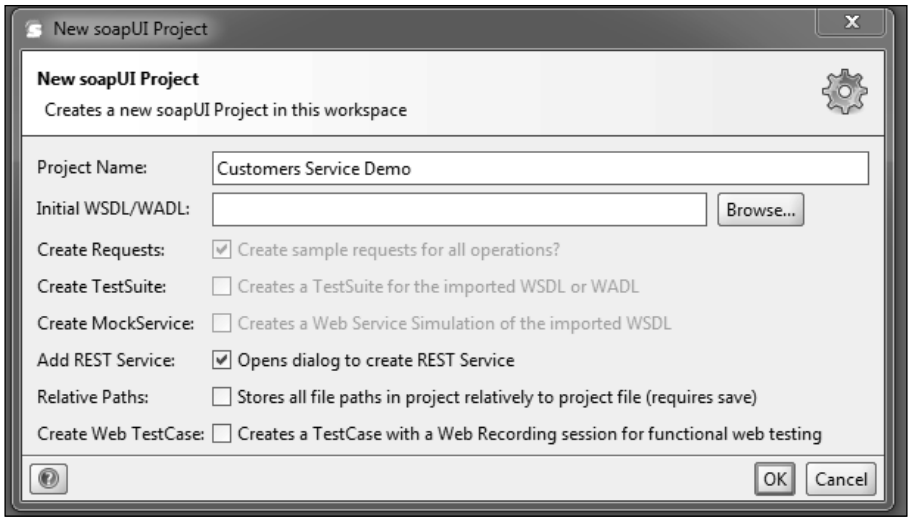
How to do it...

Open soapUI with a new workspace, then we will explore creating a RESTful service first, and continue with a SOAP service next:

Testing RESTful services

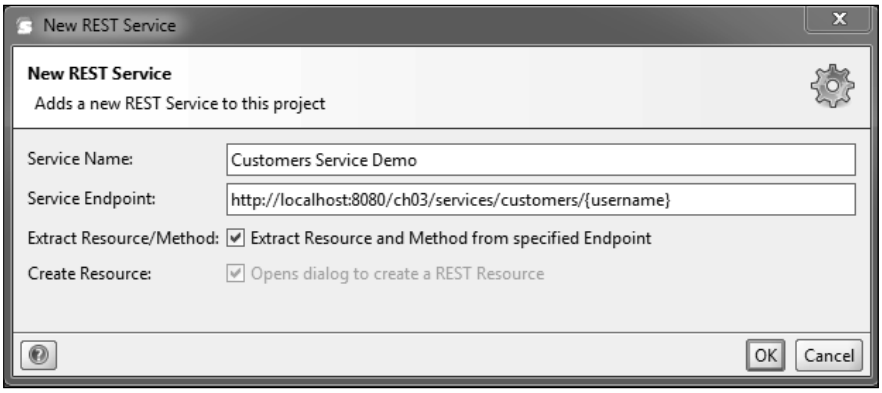
soapUI comes with an extensive support for testing WADL-based RESTFul services. We will begin with testing a REST-based service.

1. First, we create a new project from the context menu located at **File | New soapUI Project**.
2. On the **New soapUI Project** dialog box, enter a unique project name and select the **Add REST Service** option as shown in the following listing:

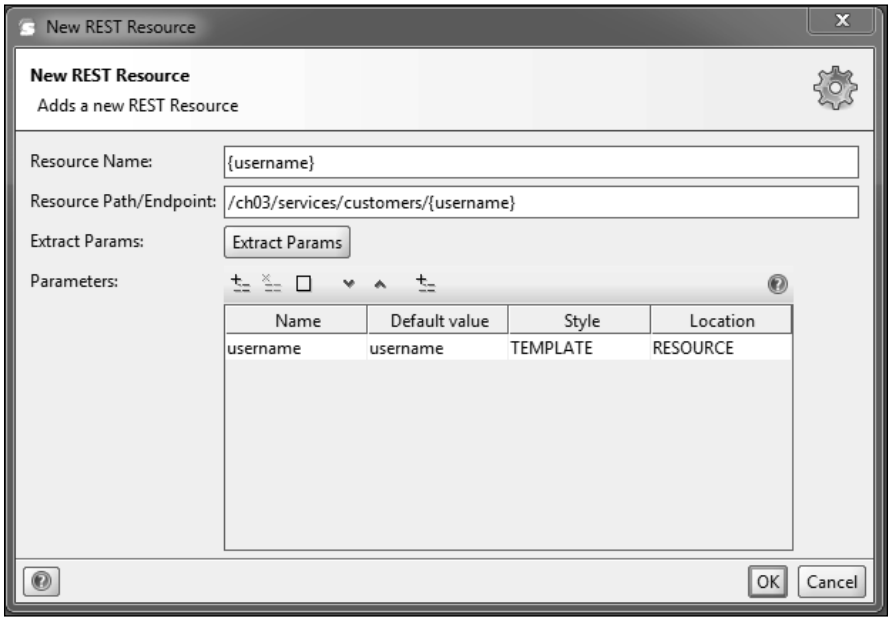


3. Click on the **OK** button to create the new project.
4. After creating the project, a dialog window will appear; it requires input for the REST service this project will be interacting with.
5. Enter a unique service name, and for the service endpoint, we need to enter the fully qualified URI for the REST service; include any optional query parameters inside curly braces.

   The following screenshot depicts a **Customers Service Demo** service, with a **username** query parameter:

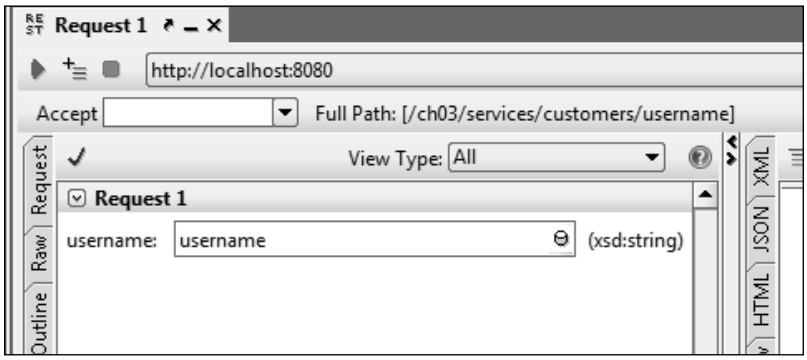


6. Clicking the **OK** button will automatically extract the **username** query argument from the endpoint and display a dialog with the new REST resource for this service:

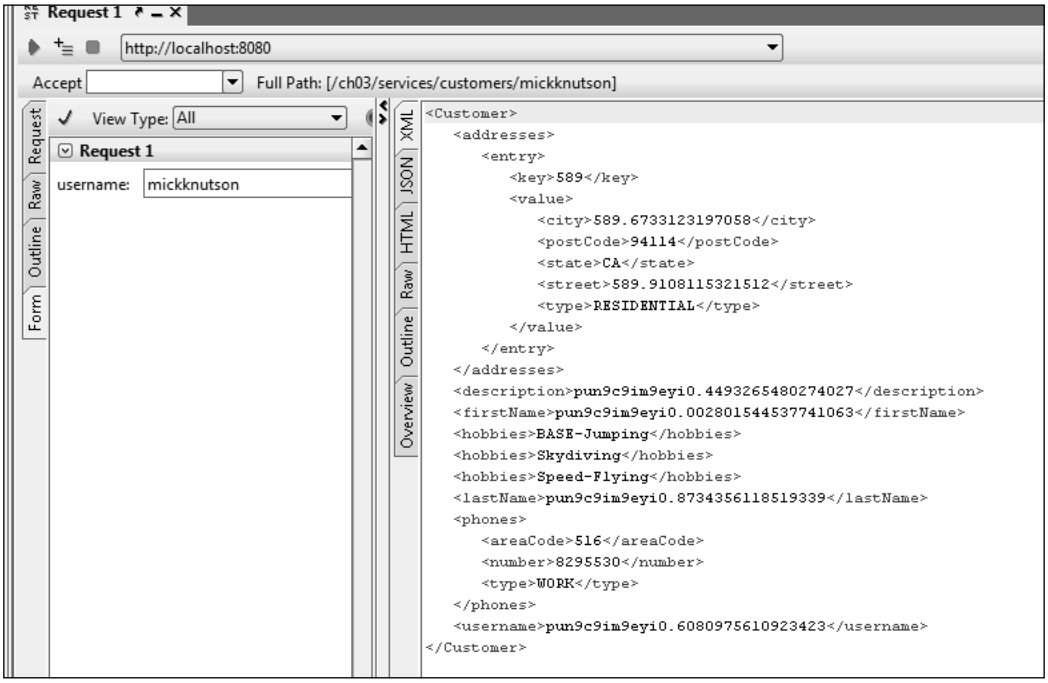


In this example, there are no additional parameters, but if one were required, it can be added.

7. Press the **OK** button to create the initial request to the new REST service and open a request editor form, as shown in the following screenshot:

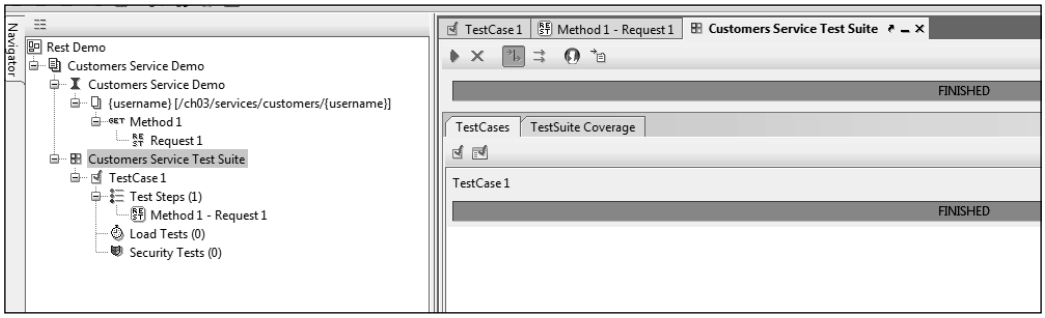


This request form has one parameter for a username which is expecting a String value. In the actual request, type `mickknutson` in the **username** text field and press the **green arrow** at the top-left side of the page, and you can see the XML output returned by the service, as shown in the following screenshot:



8. At this point we have used soapUI as a client to a published REST service. Next, create a new Test Suite by right-clicking on the **Customers Service Demo** project.
9. When prompted, enter **Customers Service Test Suite** for the **TestSuite** name.
10. Next, right-click on **Customers Service Test Suite** and select **New TestCase**.
11. Next, enter a name for the **TestCase** and click the **OK** button.
12. Now you can see the **Customers Service Test Suite** on the left-hand side. Double-click on the test suite and then press the **green arrow** button at the top-left to run test cases of this test suite.

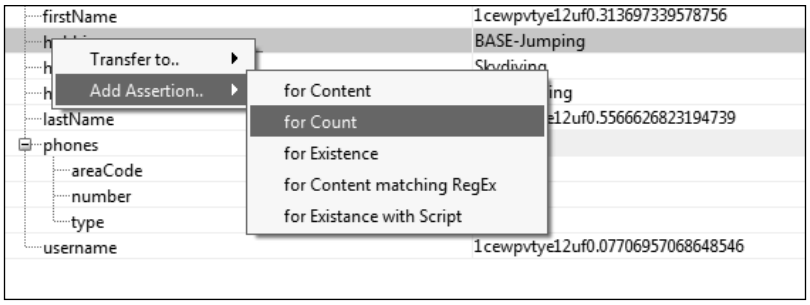
13. The test should complete successfully as depicted in the following screenshot:



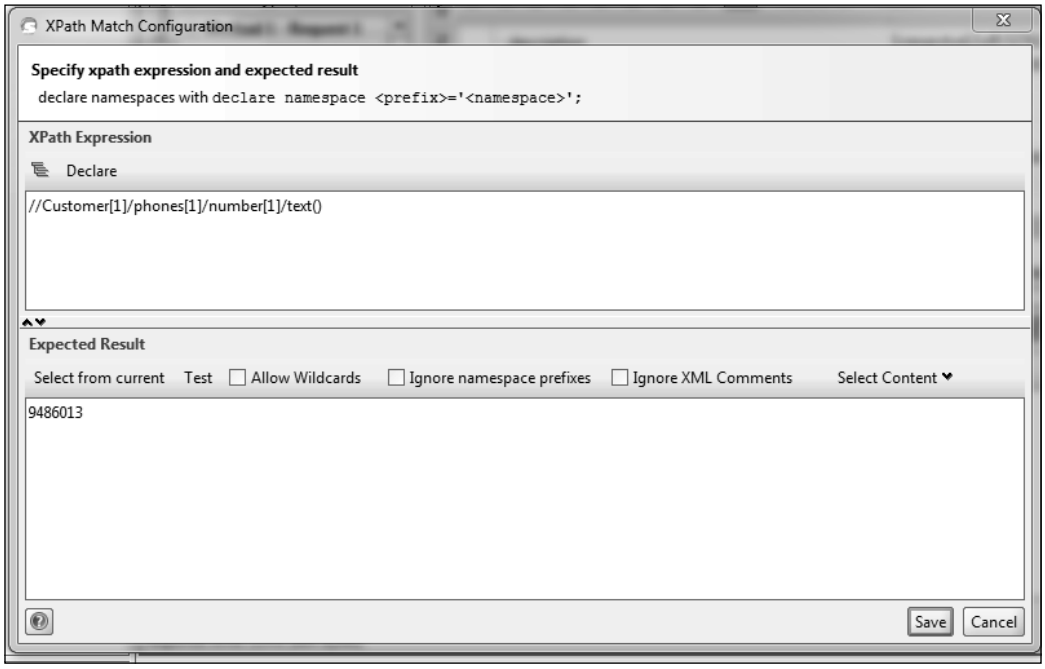
Testing in soapUI is all about the assertions. Assertions are the checks to validate the message received by a TestStep, usually by comparing the parts of the response or the entire response to the expected response. soapUI offers many different built-in assertions. The most common assertions are *contains* (checks for presence of a specified string), *not contains* (checks for the non-existence of a specified string), *XPath* or *XQuery match* (compares the result of an XPath expression to an expected value) and *Scripts* (validates the response using script). Any number of assertions can be added to validate different aspects or content of the response.

We will add an actual assertion to validate the content of the response. In our first assertion, we are just going to check that we get three hobbies back from the service for username `mickknutson`. The response is listed in the right-hand side of the window.

Now select the **Outline** view and right-click on the first **hobbies** item, in the generated XML:

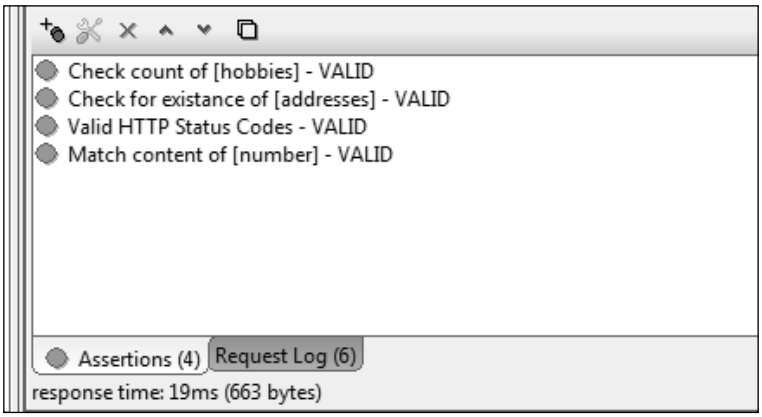


In the second example, we can add another XPath assertion to verify the content of the phone number, as shown in the following screenshot:



Similarly, we can add two more assertions to check the existence of customer address, and the response will be of valid HTTP status code.

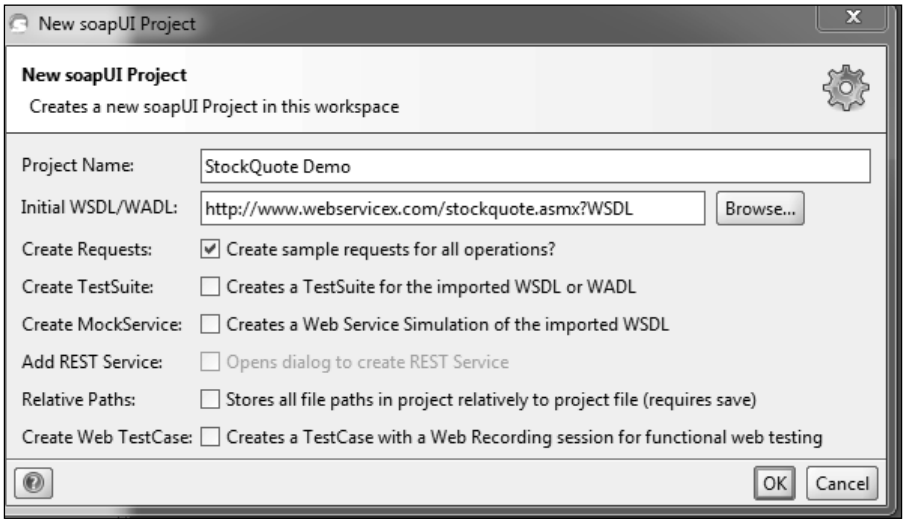
Run the TestCase with the **green arrow** at the top-left side of the window. This will result in the aforementioned output, in the log at the bottom as shown in the following screenshot:



Testing SOAP services

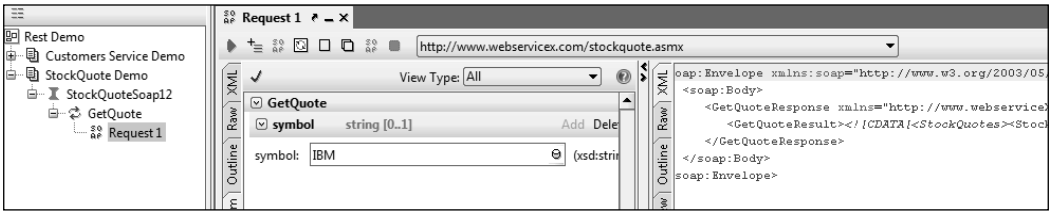
soapUI comes with a very extensive support for testing WSDL and SOAP-based services. The WSDL files are central to testing SOAP-based services. They define the actual contract a service exposes and are required by soapUI to generate and validate tests.

Create another new project in the existing workspace by importing stockQuote public service WSDL as shown in the following screenshot:



After clicking on the **OK** button, soapUI will load the stockQuote WSDL and parse its contents.

The following screenshot depicts the **GetQuote** service being executed with a valid symbol parameter and the SOAP response being returned:



There's more...

soapUI can also help to automate functional tests using the command line tools bundled with soapUI, and can also be run using a Maven build system. In order to have soapUI run during a build process, you will need to first save the soapUI project file in your source tree, such as `/src/soapui/3166_soapui_project.xml`; then add the following Maven plugin to be run during the test phase of the build:

```
<plugin
    <groupId>eviware</groupId
    <artifactId>maven-soapui-plugin</artifactId
    <version>${soapui.plugin.version}</version
    <configuration
        <outputFolder>
          ${project.basedir}/target/soapui/output
        </outputFolder>
        <junitReport>true</junitReport>
    </configuration>
    <executions>
        <execution>
            <id>Customer Service Test Suite</id>
            <phase>test</phase>
            <goals>
                <goal>test</goal>
            </goals>
            <configuration>
                <projectFile>
                  ${project.basedir}/src/test/resources/soapui/3166-
soapui-project.xml
                </projectFile>
            </configuration>
        </execution>
    </executions>
</plugin>
```

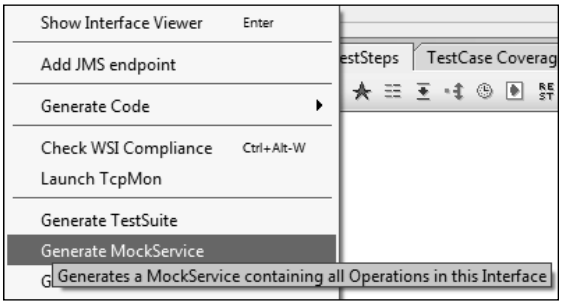
The previous listing will instruct Maven to execute the `maven-soapui-plugin` with the `3166-soapui-project.xml` project file during the test lifecycle phase of the build.

Testing with Mock services

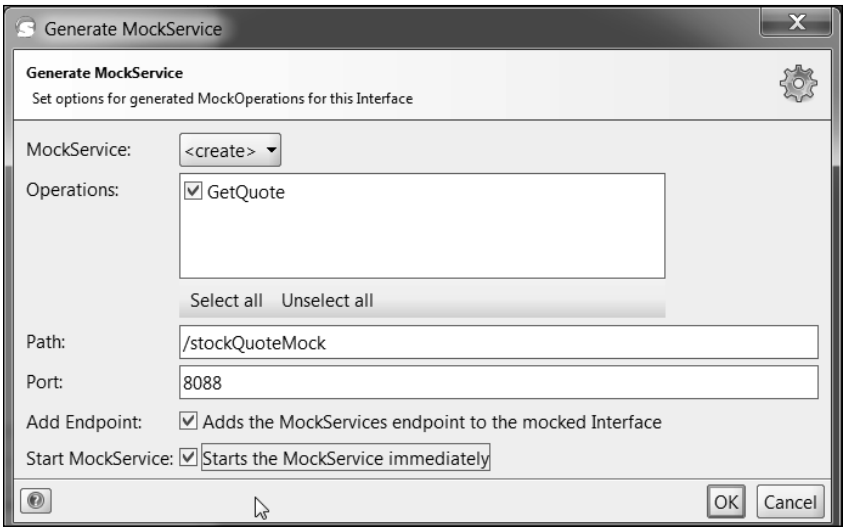
Testing with mock objects using Mockito or EasyMock are commonly used in unit testing. Mock objects simulate real objects, and they can be configured to match the behavior of real objects that are not available or cannot be incorporated into unit tests.

soapUI is capable of mocking any external service that is not available, or that cannot be incorporated into a given test case. The only thing required is the WSDL of the external service to be mocked.

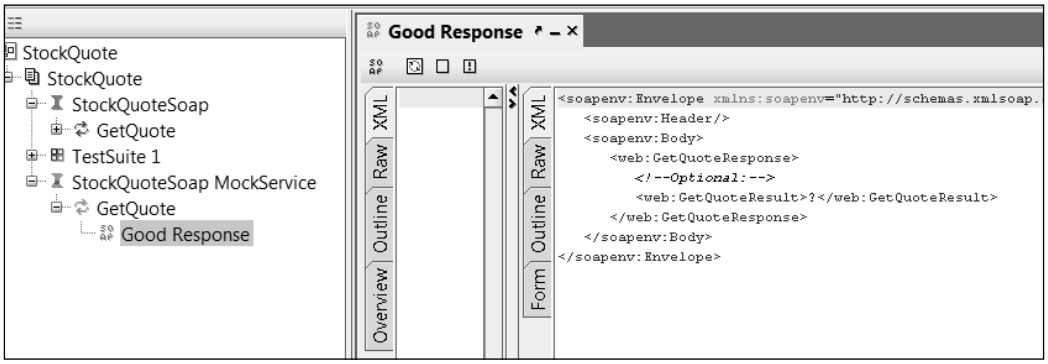
In order to begin, right-click on the project and select **Generate MockService** from the context menu as depicted in the following screenshot:



The next dialog window allows configuration of the services, operations, and URI for the new Mock. The following dialog window depicts the imported WSDL for the stockQuote service:



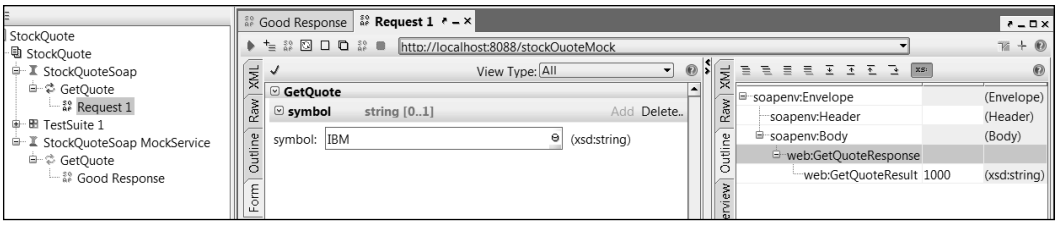
After clicking on **OK**, you will see new **StockQuoteSoap MockService** created with a new Mock response. The SOAP message response can be modified to simulate the response needed for the given test. As depicted in the following screenshot, the response for the **GetQuote** operation for **StockQuoteSoap MockService** will return a valid SOAP response:



The previous mock SOAP response for the **GetQuote** operation returns a result as an XML String, such as the following listing:

```
<web:GetQuoteResult>1000</web:GetQuoteResult>
```

The **StockQuoteSoap MockService** is now ready to handle incoming SOAP requests on the configured endpoint `http://localhost:8088/stockQuoteMock` as shown in the following screenshot:



Numerous Mock response messages can be created for a given Mock service; they can be tailored to return various values for various test outcomes. This can include different quote amounts and various fault scenarios. Creating mock responses can help with code coverage for web services.

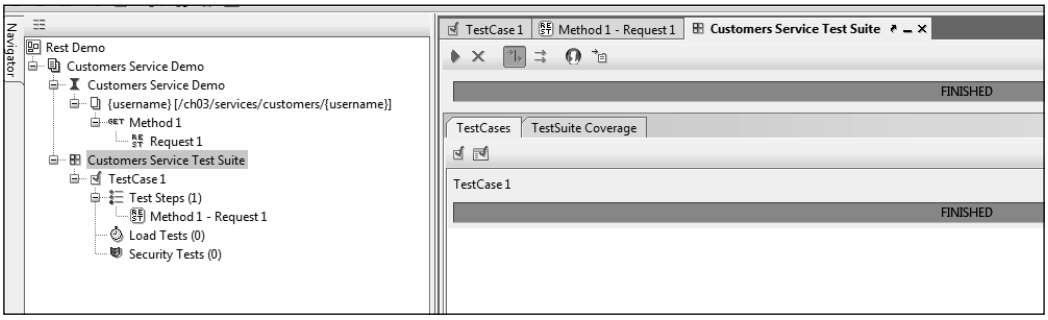
Extending soapUI capabilities with Groovy

soapUI provides extensive and powerful options for scripting using the Groovy language that is accessible from within all soapUI projects.

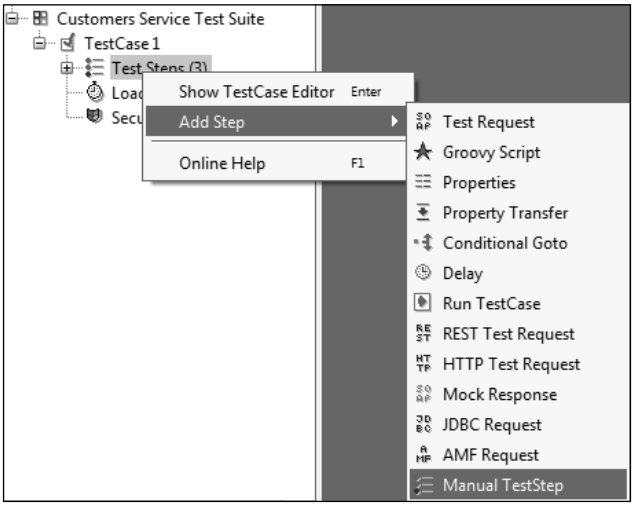
In this recipe, we will explore extending the capabilities of soapUI with Groovy scripting for testing and debugging.

Getting ready

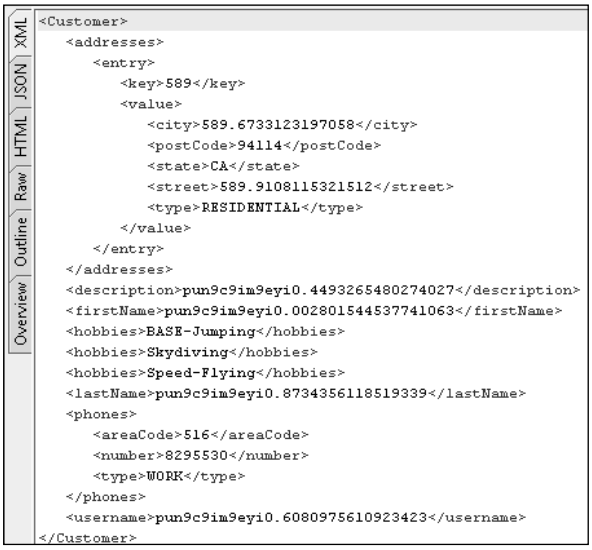
We will begin with the previous soapUI project, named **Customers Service Demo**, which was created earlier in this chapter. The following screenshot depicts the project, and the TestSuite and TestCase for the project:



In a soapUI TestCase, a **Manual TestStep** can be created; we will use this as an assertion placeholder. Later, this manual step will be referenced by our Groovy script. The following screenshot depicts how to add a manual step by right-clicking the **Test Steps** and selecting **Manual TestStep**, which is located in the context menu at **Add Step | Manual TestStep**:



Next, we need to add the expected response for the **Manual TestStep**, which will be the valid Customer as depicted in the following screenshot:

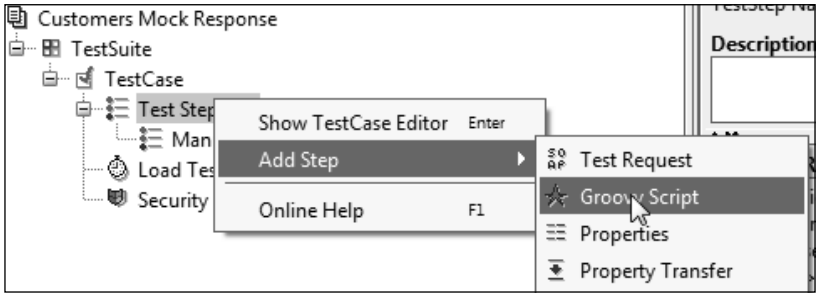


The manual step uses the expected result for assertions. The Groovy script that is created, is going to use the expected result as though it was the actual response. This can be thought of as a mock response to the Groovy script.

How to do it...

At this point the project has everything required to run a successful test, now we want to add a Groovy script to the test step:

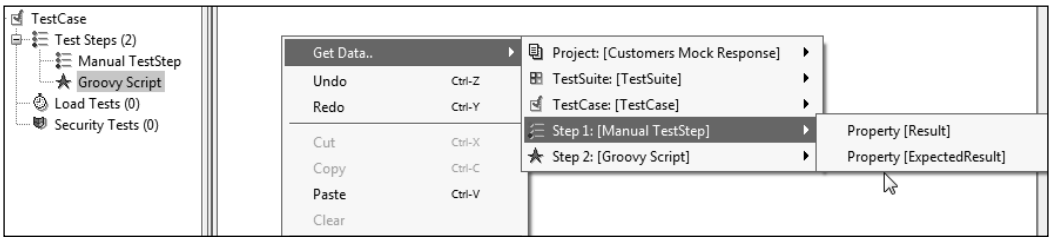
1. To begin with, add a new **Groovy Script** step by right-clicking on **Test Step**, and navigating to **Add Step | Groovy Script**, as depicted in the following screenshot:



This will create a Groovy script under **Test Step**.

2. Double-click the **Groovy Script** to open the file with the embedded Groovy editor in soapUI. In this editor, we will write the Groovy script to parse the Customers service response.

Groovy makes it very easy to inspect or parse anything from the response. The Groovy editor has access to all the context-related variables and log objects. We can retrieve any property from the response using the context variable by right-clicking in the editor and selecting **Get Data** from the menu, as depicted in the following screenshot:

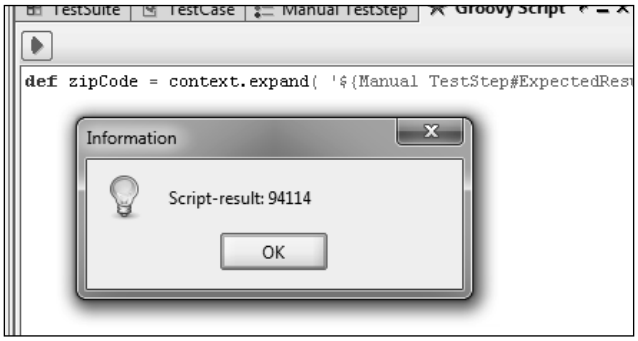


Earlier in this recipe, the **Manual TestStep** was configured with an **ExpectedResult**. The previous screenshot depicts the ExpectedResult property from the Manual TestStep being added to the Groovy script.

Within the editor, we will retrieve **postCode** from the **ExpectedResult** property, with XPath, and add the result to a new variable as shown in the following listing:

```
def zipCode = context.expand(
  '${Manual TestStep#ExpectedResult#//Customer[1]/
  addresses[1]/entry[1]/value[1]/postCode[1]}' )
```

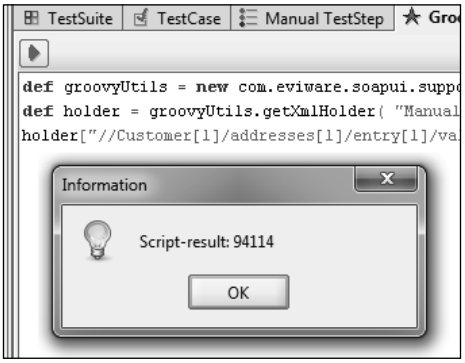
When the test case is run, this code listing will result in the following **Information** dialog window:



We can also parse the response by using soapUI Groovy utilities. The `GroovyUtils` class is used to create an `XMLHolder` to retrieve properties from the response object. The following sample code demonstrates how to create an `XmlHolder` from the soapUI `GroovyUtils` wrapper object, then `XmlHolder` is used to parse the XML and retrieve **postCode** from the **ExpectedResult** with XPath:

```
def groovyUtils =
  new com.eviware.soapui.support.GroovyUtils( context )
def holder =
  groovyUtils.getXmlHolder( "Manual TestStep#ExpectedResult" )
holder["//Customer[1]/addresses[1]/entry[1]/value[1]/postCode[1]"]
```

When the test case is run, this code listing will result in the following **Information** dialog window:



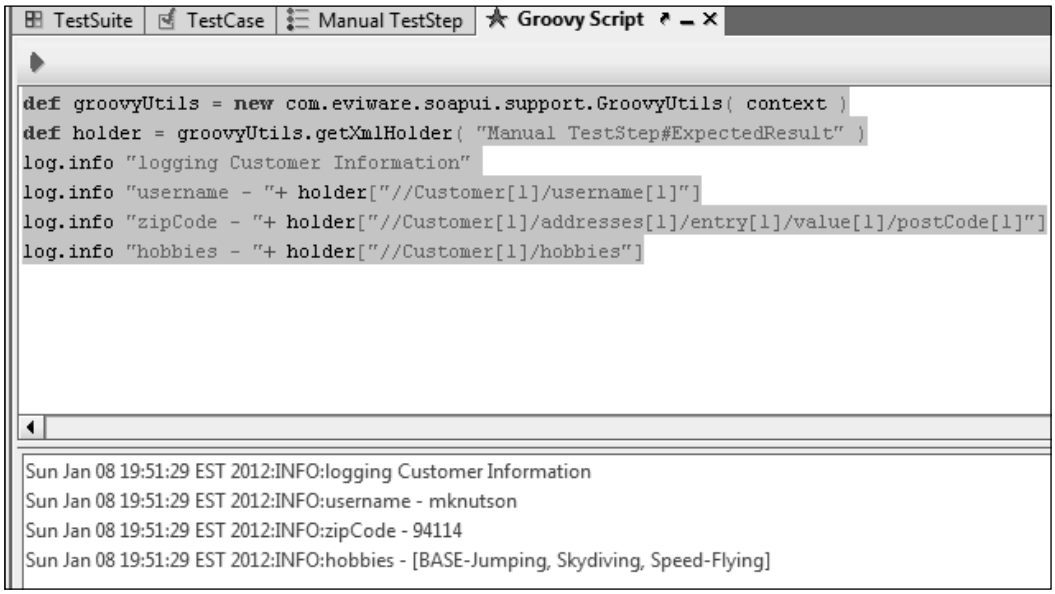
The `XMLHolder` can be used for a variety of other XML-related activities, such as a count on the response properties:

```
def groovyUtils =
  new com.eviware.soapui.support.GroovyUtils( context )
def holder =
  groovyUtils.getXmlHolder( "Manual TestStep#ExpectedResult" )
def numberOfCustomers = holder["count(//Customer)"]
assert numberOfCustomers > 0
```

Groovy can write logs from the response, or other information, using the following `log` object:

```
def groovyUtils =
  new com.eviware.soapui.support.GroovyUtils( context )
def holder =
  groovyUtils.getXmlHolder( "Manual TestStep#ExpectedResult" )
log.info "logging Customer Information"
log.info "username - "+ holder["//Customer[1]/username[1]"]
log.info "zipCode - "+ holder["//Customer[1]/addresses[1]/entry[1]/
value[1]/postCode[1]"]
log.info "hobbies - "+ holder["//Customer[1]/hobbies"]
```

When the test case is run with the log statements, the result will be as shown in the following console window for soapUI:



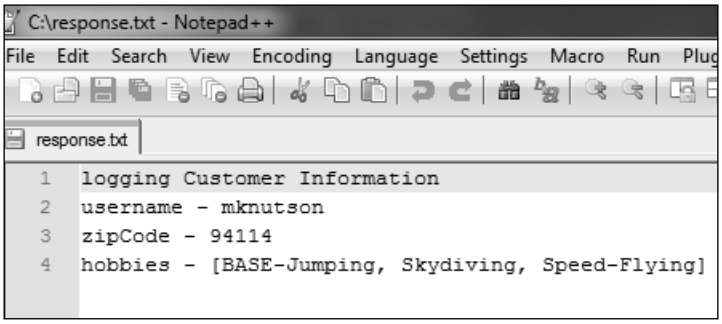
Writing output to file

Response, log, or other information can be written to a file for review and audit. This can be especially helpful if there is a large amount of console data to be captured, or if the project is being executed unattended.

The following listing depicts the creation of a new `File` object, and the addition of data to the file:

```
def groovyUtils =
  new com.eviware.soapui.support.GroovyUtils( context )
def holder =
  groovyUtils.getXmlHolder( "Manual TestStep#ExpectedResult" )
def file = new File( "c:/response.txt" )
file.append "logging Customer Information"
file.append "\nusername - "+ holder["//Customer[1]/username[1]"]
file.append "\nzipCode - "+ holder["//Customer[1]/addresses[1]/
entry[1]/value[1]/postCode[1]"]
file.append "\nhobbies - "+ holder["//Customer[1]/hobbies"]
```

When the test case is run, the output will be written to `C:/response.txt` and the text file can be viewed in a system text editor as seen in the following listing:



See also

- *SOAP UI*: `http://www.soapui.org/`

Where to buy this book

You can buy Java EE 6 Cookbook for Securing, Tuning, and Extending Enterprise Applications from the Packt Publishing website: `http://www.packtpub.com/java-ee6-securing-tuning-extending-enterprise-applications-cookbook/book`.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

www.PacktPub.com

For More Information:
www.packtpub.com/java-ee6-securing-tuning-extending-enterprise-applications-cookbook/book