Citrus Framework - Reference Documentation

Version 1.1

Preface

Integration testing can be very hard, especially when there is no sufficient tool support. Unit testing is flavored with fantastic tools and APIs like JUnit, TestNG, and EasyMock which support you in writing automated tests. A tester who is in charge of integration testing may lack of tool support for automated integration testing. In a typical Service-oriented Architecture (SOA) with enterprise applications the test team has to deal with different interfaces and various transport protocols. Without sufficient tool support the automated integration testing of message-based interactions between interface partners is exhausting and sometimes barely possible.

The tester is forced to simulate several interface partners trying to simulate a chain of actions and messages. Simulating the behavior of interface partners in a automated test causes severe problems regarding maintainability and interoperability. It is not unusual that a tester simulates external applications manually during the test, because of lacking tool support. The manual test approach is very time consuming and error prone, because the process is not repeatable and the tester manually validates the success.

The Citrus framework gives a complete test automation tool for integration testing of enterprise applications in a Service.oriented Architecture. Every time a change was made in the application all automated Citrus tests ensure the stability of interfaces and message communication. Regression testing and continuous integration testing is very easy. Citrus is designed to provide fully automated integration testing of whole end-to-end use cases with effective simulation of interface partners across various messaging transports and automated validation and functionality checks.

This documentation provides a reference guide to all features of the Citrus test framework. It gives a detailed picture of effective integration testing with automated integration test environments. Since this document is considered to be under construction, please do not hesitate to give any comments or requests to us using our user or support mailing lists.

Chapter 1. Introduction

Enterprise applications in a Service-oriented Architecture usually communicate with different interface partners over loosely coupled message-based communication. The interaction of several interface partners needs to be tested in integration testing.

In integration tests we need to simulate the communication over various transports. How can we test use case scenarios that include several interface partners interacting with each other? How can somebody ensure that the software component is working correctly with the other software systems. How can somebody run positive and negative integration test cases in an automated reproducible way? Citrus tries to answer these questions offering automated integration testing of message-based software components.

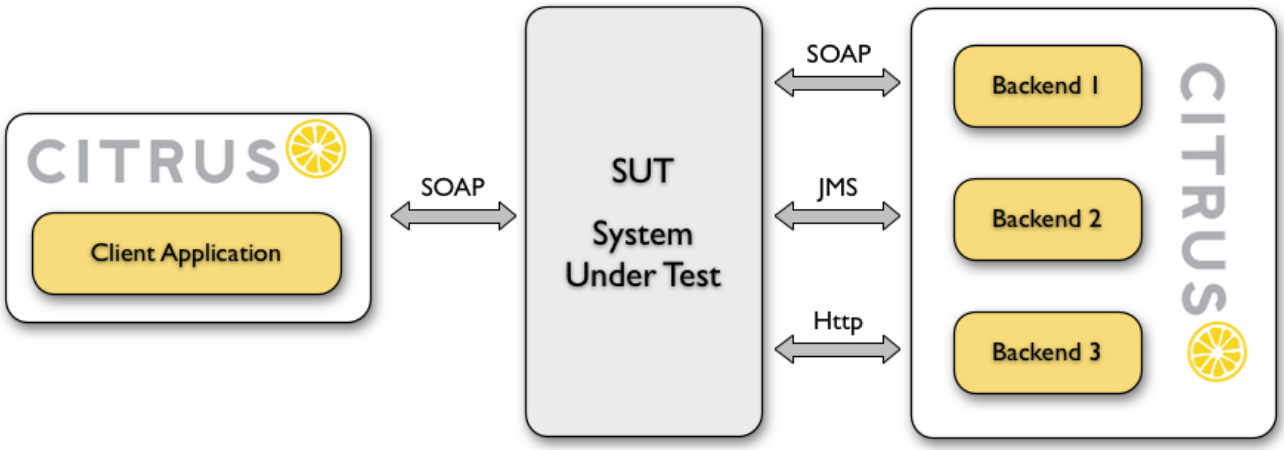
1.1. Overview

Citrus aims to strongly support you in simulating interface partners across different messaging transports. You can easily subscribe and publish to a wide range of protocols like HTTP, JMS, TCP/IP and SOAP. The framework can send and receive messages in order to create a communication chain that simulates a whole use case scenario. In each communication step Citrus is able to validate message contents. In addition to that the framework offers a wide range of actions to take control of the process flow during a test (e.g. iterations, system availability checks, database validation, parallelism, delaying, error simulation, scripting, and many more).

The basic goal in Citrus test cases is to describe a whole use case scenario including several interface partners that exchange messages with each other. You can set up complex message and data flows with several test steps as a pure XML file. The test description can be executed multiple times as automated integration test where Citrus simulated all surrounding interface partners that are not present in the test environment. With easy definition of expected message contents via XML Citrus is able to validate the data flow and the end-to-end use case processing.

1.2. Usage scenarios

If you are in charge of an enterprise application in a service-oriented Architecture with message interfaces to other software components you should use Citrus. In case your project interacts with other software systems over different messaging transports and you need to simulate your interface partners for extended end-to-end testing you should use Citrus. In case you need to increase the software stability dealing with change requests, bugfixing or regression testing you should use Citrus. In case you simply want to increase your test coverage with automated integration tests, in order to feel comfortable regarding the next software release you should definitely use Citrus.



This test set up is typical for Citrus use cases. In a such a test scenario we have a system under test (SUT) with several message interfaces to other applications. A client application invokes services on the SUT application. The SUT is linked to several backend applications over various messaging transports (here SOAP, JMS, and Http). Interim message notifications and final responses are sent back to the client application. This generates a bunch of messages that are exchanged throughout the applications involved.

In the automated integration test Citrus needs to send and receive those messages over different transports. Citrus takes care of all interface partners (ClientApplication, Backend1, Backend2, Backend3) and simulates their behavior by sending proper response messages in order to keep the message flow alive.

Each communication step comes with message validation and XML tree comparison towards an expected message template. Besides messaging actions Citrus is also able to perform plenty of other test actions. You could have a look at the database in order to check the persistence layer for instance.

The Citrus test case runs fully automated as Java application. Step by step the whole use case scenario is performed like in a real production environment. The Java test is repeatable and is included into the software building process (e.g. using Maven) like a normal unit test case would do. This gives you the automated integration testing to ensure your interface stability to external partners.

The following reference guide walks through all Citrus functionalities and shows how to set up a great integration test with Citrus.

Chapter 2. Setup

This chapter will tell you how to get started with Citrus. It deals with the installation and set up of the framework, so you are ready to start writing test cases after reading this chapter.

Usually you would use Citrus as a library in your project or as a standalone Java application. Following these two setup variations Citrus can be invoked either by using Ant or Maven. This chapter describes the Citrus project setup possibilities, choose one of them that fits best to include Citrus into your project.

2.1. Using Maven

Citrus uses Maven internally as a project build tool and provides extended support for Maven projects. Maven will ease up your life as it manages project dependencies and provides extended build lifecycles and conventions for compiling, testing, packaging and installing your Java project. Therefore it is recommended to use the Citrus Maven project setup. In case you already use Maven it is most suitable for you to include Citrus as a test-scoped dependency.

As Maven handles all project dependencies automatically you do not need to download any Citrus project artifacts in advance. If you are new to Maven please refer to the official Maven documentation to find out how to set up a Maven project. Once there is a proper Maven project available we just add some Citrus project dependencies in our Maven pom.xml like follows.

- We add Citrus as test-scoped project dependency to the project POM (pom.xml)

```
<dependency>
  <groupId>com.consol.citrus</groupId>
  <artifactId>citrus-core</artifactId>
  <version>1.1</version>
  <scope>test</scope>
</dependency>
```

- Add the citrus Maven plugin to your project

```
<plugin>
  <groupId>com.consol.citrus.mvn</groupId>
  <artifactId>citrus-maven-plugin</artifactId>
  <version>1.1</version>
  <configuration>
    <author>Donald Duck</author>
    <targetPackage>com.consol.citrus</targetPackage>
  </configuration>
</plugin>
```

Now that we have added Citrus to our Maven project we can start writing new test cases with the Citrus Maven plugin:

```
mvn citrus:create-test
```

Once you have written the Citrus test cases you can execute them automatically in your Maven software build lifecylce. The tests will be included into your projects integration-test phase using the Maven surefire plugin. Here is a sample surefire configuration for Citrus.

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.4.3</version>
```

```
  <configuration>
    <skip>true</skip>
  </configuration>
  <executions>
    <execution>
      <id>citrus-tests</id>
      <phase>integration-test</phase>
      <goals>
        <goal>test</goal>
      </goals>
      <configuration>
        <skip>false</skip>
      </configuration>
    </execution>
  </executions>
</plugin>
```

The Citrus source directories are defined as test sources like follows:

```
<testSourceDirectory>src/citrus/java</testSourceDirectory>
<testResources>
  <testResource>
    <directory>src/citrus/java</directory>
    <includes>
      <include>**</include>
    </includes>
    <excludes>
      <exclude>*.java</exclude>
    </excludes>
  </testResource>
  <testResource>
    <directory>src/citrus/tests</directory>
    <includes>
      <include>**/*</include>
    </includes>
    <excludes>
    </excludes>
  </testResource>
</testResources>
```

Now everything is set up and you can call the usual Maven *install* goal (mvn clean install) in order to build your project. The Citrus integration tests are executed automatically during the build process. Besides that you can call the Maven integration-test phase explicitly to execute all Citrus tests or a specific test by its name:

```
mvn integration-test
mvn integration-test -Dtest=MyFirstCitrusTest
```

Note

If you need additional assistance in setting up a Citrus Maven project please visit the official Maven tutorial on http://www.citfrusframework.org.

2.2. Using Ant

Ant is a very popular way to compile, test, package and execute Java projects. The Apache project has effectively become a standard in managing Java projects. You can run Citrus test cases with Ant as Citrus is nothing but a Java application. This section describes the steps to setup a proper Citrus Ant project.

2.2.1. Preconditions

Before we start with the Citrus setup be sure to meet the following precondintions. The following software should be installed on your computer, in order to use the Citrus framework:

- Java 5.0 or higher

  Installed JDK 5.0 or higher plus JAVA_HOME environment variable set up and pointing to your Java installation directory

- Java IDE (optional)

  A Java IDE will help you to manage your Citrus project (e.g. creating and executing test cases). You can use the any Java IDE (e.g. Eclipse or IntelliJ IDEA) but also any convenient XML Editor to write new test cases.

- Ant 1.7.0 or higher

  Ant (http://ant.apache.org/) will run tests and compile your Citrus code extensions if necessary.

2.2.2. Download

First of all we need to download the latest Citrus release archive from the official website http://www.citrusframework.org

Citrus comes to you as a zipped archive in one of the following packages:

- *citrus-x.x-with-dependencies*
- *citrus-x.x-src*

Usually you would choose the archive including all dependency libraries to start using Citrus. Besides that the package includes the Citrus binaries as well as the reference documentation and some sample applications.

In case you want to get in touch with developing and debugging Citrus you can also go with the source archive which gives you the complete Citrus Java code sources. The whole Citrus project is also accessible for you on http://github.com/christophd/citrus. This open git repository on GitHub enables you to build Citrus from scratch with Maven and contribute code changes.

2.2.3. Installation

After downloading the Citrus archives we extract those into an appropriate location on the local storage. We are seeking for the Citrus project artefacts coming as normal Java archives (e.g. citrus-core.jar, citrus-ws.jar, etc.)

You have to include those Citrus Java archives as well as all dependency libraries to your Ant Java classpath. Usually you would copy all libraries into your project's lib directory and declare those libraries in the Ant build file. See the following sample Ant build script which uses the Citrus project artefacts in combination with the Citrus Ant tasks to setup and run the tests.

```
<project name="citrus-sample" basedir="." default="citrus.run.tests">

  <property file="src/citrus/resources/citrus.properties"/>

  <path id="citrus-classpath">
```

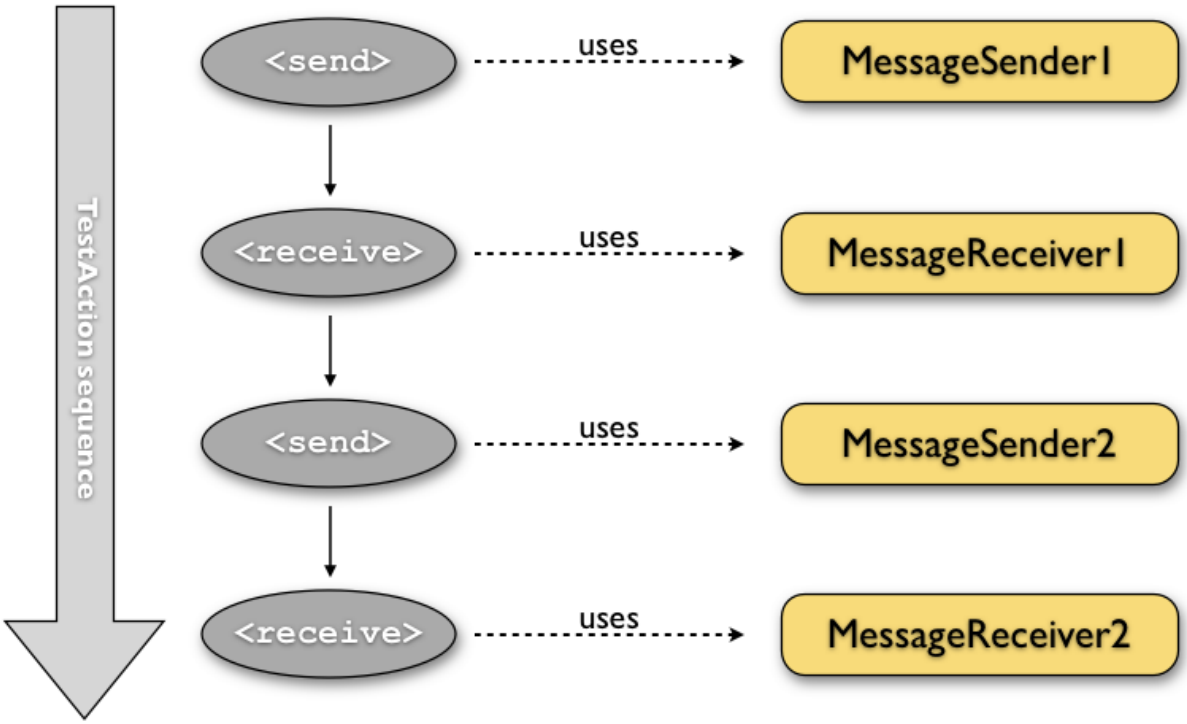
```
    <fileset dir="lib">
      <include name="*.jar"/>
    </fileset>
  </path>

  <typedef resource="citrustasks" classpath="lib/citrus-ant-tasks-1.0.jar"/>

  <target name="compile.tests">
    <javac srcdir="src/citrus/java" classpathref="citrus-classpath"/>
    <javac srcdir="src/citrus/tests" classpathref="citrus-classpath"/>
  </target>

  <target name="create.test" description="Creates a new empty test case">
    <input message="Enter test name:" addproperty="test.name"/>
    <input message="Enter test description:" addproperty="test.description"/>
    <input message="Enter author's name:" addproperty="test.author" defaultvalue="${default.test.author}"/>
    <input message="Enter package:" addproperty="test.package" defaultvalue="${default.test.package}"/>
    <input message="Enter framework:" addproperty="test.framework" defaultvalue="testng"/>

    <java classname="com.consol.citrus.util.TestCaseCreator">
      <classpath refid="citrus-classpath"/>
      <arg line="-name ${test.name} -author ${test.author} -description ${test.description}
                 -package ${test.package} -framework ${test.framework}"/>
    </java>
  </target>

  <target name="citrus.run.tests" depends="compile.tests" description="Runs all Citrus tests">
    <citrus suitename="citrus-samples" package="com.consol.citrus.*"/>
  </target>

  <target name="citrus.run.single.test" depends="compile.tests" description="Runs a single test by name">
    <touch file="test.history"/>
    <loadproperties srcfile="test.history"/>

    <echo message="Last test executed: ${last.test.executed}"/>
    <input message="Enter test name or leave empty for last test executed:" addproperty="testclass"
                 defaultvalue="${last.test.executed}"/>

    <propertyfile file="test.history">
        <entry key="last.test.executed" type="string" value="${testclass}"/>
    </propertyfile>

    <citrus suitename="citrus-samples" test="${testclass}"/>
  </target>

</project>
```

Note

If you need detailed assistance for building Citrus with Ant do also visit our tutorials section on http://www.citrusframework.org. There you can find a tutorial which describes the Citrus Java project set up with Ant from scratch.

Chapter 3. Test case

Now let us start writing test cases! A test case describes the steps for a certain use case. The test case holds a sequence of different test actions that are executed during the test. Typically in message-based applications sending and receiving messages will be the main actions inside a test. But you will learn that Citrus is able to add much more logic in a test case in order to accomplish very complex use case tests. Step by step we will introduce all possible actions inside a test case and explain how to connect to other systems by sending and receiving messages over various transports.



The figure above describes a typical test action sequence in Citrus. A list of test actions that send and receive messages with various MessageSender and MessageReceiver components. So how do we define those test logic? Citrus specifies test cases through simple XML files. The whole test case description will take place in one single XML file. This chapter will introduce the custom XML schema language that defines a test cases.

3.1. Defining a test case

Clearly spoken a test case is nothing but a simple Spring XML configuration file. So using the Spring XML configuration syntax you are able to write fully compatible test cases for the Citrus framework.

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="myFirstTest"
             class="com.consol.citrus.model.TestCase">
        <property name="actions">
            <!-- actions of this test go here -->
        </property>
    </bean>
</beans>
```

Citrus can execute these test case as normal test cases - no problem, but this XML usual syntax is not the best way to describe a test case in Citrus, especially when test scenarios get more complex and the test cases grow in size. Therefore Citrus provides a custom XML schema definition for writing test cases.

The custom XML schema aims to reach the convenience of Domain specific languages (DSL). Let us have a look at the Citrus test describing DSL by introducing a first very simple test case definition:

```
<spring:beans
    xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.citrusframework.org/schema/testcase
    http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd">

    <testcase name="myFirstTest">
        <description>
            First example showing the basic test case definition elements!
        </description>
        <variables>
            <variable name="text" value="Hello Test Framework"/>
        </variables>
        <actions>
            <echo>
                <message>${test}</message>
            </echo>
        </actions>
    </testcase>
</spring:beans>
```

We do need the `<spring:beans>` root element as the XML file is read by the Spring IoC Container. Inside this root element the Citrus specific namespace definitions take place. In the following we use the customized XML schema language in Citrus to describe the test case.

The test case itself gets a mandatory name that must be unique throughout all test cases in a project. You will receive errors when using duplicate test names. The name must not contain any whitespaces but does support special characters like '-', '.', '_'. The <testcase> element holds several child elements. These basic test elements are described in the following sections.

3.1.1. Description

The test case description points out the purpose and gives a short introduction to the intended use case scenario that will be tested. The user should get a first impression what the test case is all about. You can use free text in order to describe the test. But be aware of the XML validation rules of well formed XML (e.g. special character escaping, use of CDATA sections)

3.1.2. Variables

```
<variables>
    <variable name="text" value="Hello Test Framework"/>
    <variable name="customerId" value="123456789"/>
</variables>
```

The test variables are valid for the whole test case. You can reference them several times using a variables expression `"${variable-name}"`. It is good practice to provide all important entities as test variables. This makes the test easier to maintain and more flexible. All essential entities are present right at the beginning of the test, which may also give the opportunity to easily create test variants by

simply changing the variable values.

The name of the variable is arbitrary. Feel free to specify any name you can think of. Of course you need to be careful with special characters and reserved XML entities like '&', '<', '>'. If you are familiar with Java or any other programming language simply think of the naming rules there and you will be fine with working on Citrus variables too. The value of a variable can be any character sequence. But again be aware of special XML characters like "<" that need to be escaped ("<") when used in variable values.

The advantage of variables is obvious. Once declared the variables can be referenced many times in the test. This makes it very easy to vary different test cases by adjusting the variables for different means (e.g. use different error codes in test cases).

3.1.3. Global variables

The last section told us to use variables as they are very useful and extend the maintainability of test cases. Now that every test case defines local variables which are valid inside the test you can also define global variables. The global variables are valid in all tests by default. This is a good opportunity to declare constant values for all tests in global variables. See the following ways to add global variables to Citrus:

```
<bean class="com.consol.citrus.variable.GlobalVariables">
    <property name="variables">
        <map>
            <entry key="projectName" value="Citrus Integration Testing"/>
            <entry key="userName" value="TestUser"/>
        </map>
    </property>
</bean>
```

Add this Spring bean to the *'citrus-context.xml'* application context file in order to have the global variables available in all tests. The bean just receives a map of key-value-pairs where the keys represent the variable names and the values the respective values, of course.

Another possibility to set global variables is to load those from external property files. This may give you more powerful global variables with user specific properties and so on. See how to load properties as global variables in this example:

```
<bean class="com.consol.citrus.variable.GlobalVariablesPropertyLoader">
    <property name="propertyFiles">
        <list>
            <value>classpath:global-variable.properties</value>
        </list>
    </property>
</bean>
```

Again we just place a Spring bean in the application context and everything is done. Citrus loads the properties as name-value-pairs which represent the global variables.

3.1.4. Actions

A test case defines a sequence of actions that will take place during the test. The actions are executed sequentially as they are defined in the test case definition.

```
<actions>
    <action>[...]</action>
    <action>[...]</action>
    <action>[...]</action>
```

```
    <action>[...]</action>
</actions>
```

All actions have individual names and properties that define the action behavior. Citrus offers a wide range of test actions from scratch, but you are also able to write your own test actions in Java or Groovy and execute them during a test. Chapter 6, *Test actions* gives you a brief description of all available actions that can be part of a test case execution.

The actions are combined in free sequence to each other so that the tester is able to declare a special action chain inside the test. These actions can be sending or receiving messages, delaying the test, validating the database and so on. Step by step the test proceeds the action chain. Usually the tester tries to fit the designed use case scenario with the action chain.

3.1.5. Cleanup

The finally element also contains a list of test actions. These actions will be executed at the very end of the test case even if errors did occur during the execution before. This is the right place to tidy up things that were previously created by the test like cleaning up the database for instance. The finally section is described in more detail in Chapter 9, *Finally section*

```
<finally>
    <action>[...]</action>
    <action>[...]</action>
    <action>[...]</action>
    <action>[...]</action>
</finally>
```

3.2. Meta information

The user can provide some basic information about the test case. The meta-info element at the very beginning of the test case holds information like the author of the test or the creation date. In detail the meta information is specified like this:

```
<testcase name="metaInfoTest">
    <meta-info>
        <author>Christoph Deppisch</author>
        <creationdate>2008-01-11</creationdate>
        <status>FINAL</status>
        <last-updated-by>Christoph Deppisch</last-updated-by>
        <last-updated-on>2008-01-11T10:00:00</last-updated-on>
    </meta-info>
    <description>
        ...
    </description>
    <actions>
        ...
    </actions>
</testcase>
```

The status allows following values: DRAFT, READY_FOR_REVIEW, DISABLED, FINAL. The meta-data information to a test is quite important to give the reader a first information about the test. It is also possible to generate test documentation using this meta-data information. The built-in Citrus documentation generates HTML or Excel documents that list all tests with their metadata information and description.

Note

Tests in status DISABLED will not be executed during a test suite run. So someone can just start adding planned test cases that are not finished yet in status DRAFT. In case a test is not runnable yet because not finished entirely someone may disable a test temporarily to avoid causing failures during a test run. Using these different status one can easily set up test plans and review the progress of test coverage by comparing the amount of test in status DRAFT to those in FINAL state.

Chapter 4. Running tests

Citrus test cases consist of two parts. On the one hand the XML part where you define what should happen in the test case. On the other hand the Java part which is responsible for test execution. In the following sections we concentrate on the Java part and the test execution mechanism.

So if you create new test cases in Citrus - for instance via Maven plugin or ANT build script - Citrus generates both parts in your test directory. Let's have an example: If you create a new test named *MyFirstCitrusTest* you will find those two files as a result:

```
src/citrus/tests/com/consol/citrus/MyFirstCitrusTest.xml
```

```
src/citrus/java/com/consol/citrus/MyFirstCitrusTest.java
```

With this test creation we already have made a very important decision. During creation wizard Citrus asks you which execution framework should be used for this test. There are basically three options available: `testng`, `junit3` and `junit4`.

So why is Citrus related to Unit tests although it is intended to be a framework for integration testing? The answer to this question is quite simple: This is because Citrus wants to benefit from both JUnit and TestNG for Java test execution. Both JUnit and TestNG Java APIs offer various ways of execution and both are widely supported by other tools and frameworks.

Plus users might already know one of these frameworks and the chances are good that you are familiar with them. Everything you can do with JUnit and TestNG test cases you can do with Citrus tests. Include them into your Maven build lifecycle. Execute tests from your IDE (Eclipse, IDEA or NetBeans). Include them into a continuous build tool (e.g. Bamboo or Hudson). Generate test execution reports and test coverage reports. The possibilities with JUnit and TestNG are amazing.

So let us have a closer look at the Citrus TestNG and JUnit integration.

4.1. Run with TestNG

TestNG stands for next generation testing and has had a great influence in adding Java annotations to the unit test community. Citrus is able to generate TestNG Java classes that are executable as test cases. See the following standard template that Citrus will generate when having new test cases:

```
package com.consol.citrus.samples;

import org.testng.ITestContext;
import org.testng.annotations.Test;

import com.consol.citrus.testng.AbstractTestNGCitrusTest;

/**
 * TODO: Description
 *
 * @author Unknown
 * @since 2010-06-05
 */
public class TestNGSampleTest extends AbstractTestNGCitrusTest {
    @Test
    public void sampleTest(ITestContext testContext) {
        executeTest(testContext);
    }
}
```

If you are familiar with TestNG you will see that the Citrus generated Java class is nothing but a normal TestNG test class. The good news is that we can still use the fantastic TestNG features in our test. You can think of parallel test execution, test groups, setup and tear down operations and many more. Just to give an example we can simply add a test group to our test like this:

```
@Test(groups = {"long-running"})
```

For more information on TestNG please visit the official homepage where you also find a reference documentation on all features.

4.2. Run with JUnit

JUnit is a very popular unit test framework for Java applications widely accepted and widely supported by many tools and frameworks. Citrus offers complete JUnit support for both major versions 3.x and 4.x. If you choose *junit3* as execution framework Citrus generates a Java file that looks like this:

```
package com.vodafone.uc.il.itest.service;

import com.consol.citrus.junit.AbstractJUnit38CitrusTest;

/**
 * TODO: Description
 *
 * @author Unknown
 * @since 2010-06-05
 */
public class JUnit3SampleTest extends AbstractJUnit38CitrusTest {
    public void testSampleTest() {
        executeTest();
    }
}
```

For *junit4* the respective file looks like this:

```
package com.vodafone.uc.il.itest.service;

import org.junit.Test;
import com.consol.citrus.junit.AbstractJUnit4CitrusTest;

/**
 * TODO: Description
 *
 * @author Unknown
 * @since 2010-06-10
 */
public class SampleTest extends AbstractJUnit4CitrusTest {
    @Test
    public void sampleTest() {
        executeTest();
    }
}
```

The different JUnit versions reveal slight differences, but the idea is the same. We are still able to extend the generated Java files with JUnit features. These Java files are responsible for executing the Citrus test cases. For detailed information about JUnit and the fantastic ways to include those tests into your project please visit the official JUnit homepage.

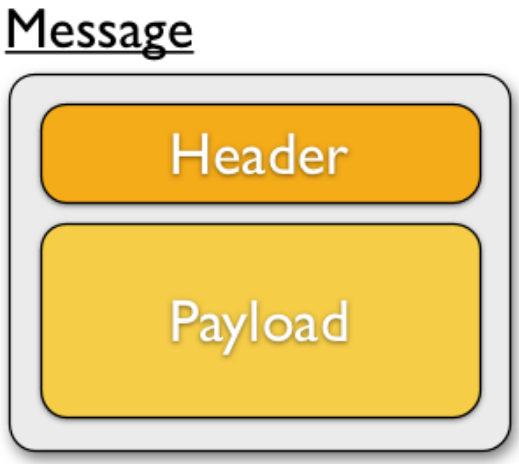
Tip

So now we know both TestNG and JUnit support in Citrus. Which framework should someone choose? To be honest, there is no easy answer to this question. The basic features are equal to each other, where TestNG offers better possibilities for designing more complex test setup with test groups and tasks before and after a group of tests. This is why TestNG is the default option in Citrus. But at the end you have to decide on your own which framework fits best for your project.

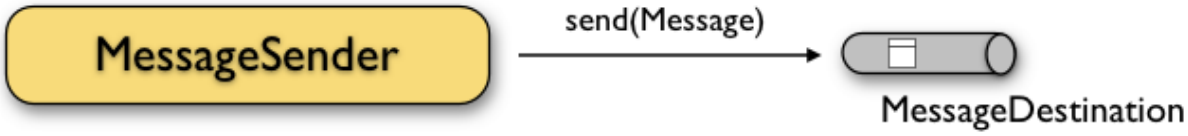
Chapter 5. Messaging

In the previous chapter the basic test case structure was introduced with *variables* and *test actions*. The <actions> element contains a list of actions that take place during the test case. Each test action is executed in sequential order. Citrus offers several built-in test actions that the user can choose from to construct a complex test workflow. The available test actions are discussed in the next sections.

As sending and receiving messages is essential in integration testing of message-based architectures we will handle these actions in first place. But first of all lets have a look at the common message interface in Citrus:



A message consists of a message header (name-value pairs) and a message payload. Later in this document we will see how a test constructs several messages with payload and header values. But first of all let's concentrate on sending and receiving messages to/from various transports. MesageSender and MessageReceiver components play a significant role in this respect. The next figure shows a typical MessageSender component in Citrus:



The MessageSender publishes messages to a destination. This destination can be a JMS queue/topic, a SOAP WebService endpoint, a Http URL and many more. The MessageSender just takes a previously defined message and sends it to the message destination.

Similar to that Citrus defines the MessageReceiver component to consume messages from destinations. This can be a simple subscription on message channels and queues/topics. In case of SOAP WebServices and Http GET/POST things are more complicated as we have to provide a server component in combination with a MessageReceiver. We will handle this server related communication later in this document. For now a MessageReceiver component is defined like this:

In the next sections you learn how a test case uses those MessageSenders and MessageReceivers for publishing and consuming messages.

5.1. Sending messages

The <send> action publishes messages to a destination. The transport that is used does - for now - not matter to the test case. The test case simply defines the message contents and uses a predefined message sender to actually publish the constructed message. There are several message sender implementations in Citrus available representing different transport protocols like JMS, SOAP, HTTP, TCP/IP and many more.

Again the type of transport is not specified in the test case directly, but in the message sender definitions. The separation of concerns (test case/message sender transport) gives us a good flexibility of our test cases. The test case does not know anything about connection factories, queue names or endpoint urls. The transport underneath a sending action can change easily without affecting the test case definition. We will see later in this document how to create different message senders for various transports in Citrus. For now we concentrate on constructing the message content to be sent.

We assume that the message's payload will be plain XML format. Citrus supports XML payloads as default payload format. Citrus is not limited to XML message format though; you can always add message converters and marshallers. But to be honest XML is a very popular message format in Enterprise Applications and messaging solutions. I do not see the point why you should not use XML unless you deal with legacy interfaces or strongly need to avoid the XML message overhead. Anyway Citrus works best on XML payloads and you will see a lot of example code in this document using XML. Finally let us have a look at a first example how a sending action is defined in the test.

```
<testcase name="sendMessageTest">
    <description>
        Send message example
    </description>
    <variables>
        <variable name="requestTag" value="Rx123456789"/>
        <variable name="correlationId" value="Cx1x123456789"/>
    </variables>
    <actions>
        <send with="getCustomerRequestMessageSender">
            <message>
                <data>
                    <![CDATA[
                    <RequestMessage>
                        <MessageHeader>
                            <CorrelationId>${correlationId}</CorrelationId>
                            <Timestamp>2001-12-17T09:30:47.0Z</Timestamp>
                            <RequestTag>_</RequestTag>
                            <VersionId>2</VersionId>
                        </MessageHeader>
                        <MessageBody>
                            <Customer>
                                <Id>1</Id>
                            </Customer>
                        </MessageBody>
                    </RequestMessage>
                    ]]>
                </data>
                <element path="/MessageHeader/RequestTag" value="${requestTag}"/>
            </message>
            <header>
                <element name="Operation" value="GetCustomer"/>
                <element name="RequestTag" value="${requestTag}"/>
            </header>
        </send>
    </actions>
</testcase>
```

The test uses two variable definitions (*requestTag* and *correlationId*), so first of all let us refresh in mind what variables do. Test variables are defined at the very beginning of the test case and are

valid throughout all actions that take place in the test. This means that actions can simply reference a variable by the expression `${variable-name}`.

Tip

Use variables wherever you can! At least the important entities of a test should be defined as variables at the beginning. The test case improves maintainability and flexibility when using variables.

Now lets have a closer look at the sending action. The 'with' attribute might catch someone's attention in first place. So what does the 'with' attribute do? This attribute references a message sender definition by name. As previously mentioned the message sender definition lies in a separate configuration file and contains the actual message transport configurations. In this example the *"getCustomerRequestMessageSender"* is used to send the message over JMS to a destination queue. The test case is not aware of these transport details, because it does not have to. The advantages are obvious: On the one hand many test cases may reuse a single message sender definition in order to send messages of type 'getCustomerRequest'. Secondly test cases are independent of message transport details. Connection factories, user credentials, endpoint urls are not present in the test cases but are configured in a central place - the actual message sender configuration.

In other words the attribute "with" in the <send> element specifies which message sender definition to use for sending the message. Once again all available message senders are configured in a separate Spring configuration file. We will come to this later. Be sure to always pick the right message sender type in order to publish your message to the right destination endpoint.

Tip

It is good practice to follow a naming convention when choosing names for message senders and receivers. The intended purpose of the message sender as well as the sending/receiving actor should be clear when choosing the name. For instance messageSender1, messageSender2 will not give you much hints to the purpose of the message sender.

Now that the used message sender is clear, we finally can specify the message content to be sent. The message content is declared with these elements:

- *message*: This element constructs the message to be sent. There are several child elements available:
  - *data*: Inline CDATA definition of the message payload (instead of <resource> element)
  - *resource*: External file resource holding the message payload (instead of <data> element)

    The syntax would be: <resource file="file:xmlData/NumberDeallocationRequest.xml" />

    The file path prefix indicates the resource type, so the file location is resolved either as file system resource (file:) or classpath resource (classpath:).
  - *element*: Explicitly overwrite values in the XML message payload using XPath. You can replace message content with dynamic values before sending. Each <element> entry provides a "path" and "value" attribute. The "path" gives a XPath expression evaluating to a XML node element or

attribute in the message. The "value" can be a variable expression or any other static value. Citrus will replace the value before sending the message.

- *header*: Defines a header for the message (e.g. JMS header information or SOAP header):
  - *element*: Each header receives a "name" and "value". The "name" will be the name of the header entry and "value" its respective value. Again the usage of variable expressions as value is supported here, too.

The most important thing when dealing with sending actions is to prepare the message payload and header. You are able to construct the message payload either by inline CDATA (<data>) or external file (<resource>). Before sending takes place you can explicitly replace some message values. You can think of overwriting specific message elements with variable values for instance. The example above uses the variable ${correlationId} directly in the XML payload definition. In addition to that you can use XPath expressions for overwriting message contents before sending. The "/MessageHeader/RequestTag" XPath expression for instance overwrites the respective request tag value in the message. The two approaches of overwriting message elements before sending can coexist simultaneously.

The message header is manipulated too. So Citrus uses name-value pairs like "Operation" and "RequestTag" in the example above to set the message's header entries.

This is how you send messages in Citrus. The test case is responsible for constructing the message content while predefined message senders are reused to publish the messages. The variable support in message payload and message header enables you to add dynamic values.

5.2. Receiving messages

Now after sending a message with Citrus we would like to receive a message inside the test. Let us again have a look at a simple example showing how it works.

```
<receive with="getCustomerResponseReceiver">
    <selector>
        <value>operation = 'GetCustomer'</value>
    </selector>
    <message>
        <data>
            <![CDATA[
            <RequestMessage>
                <MessageHeader>
                    <CorrelationId>${correlationId}</CorrelationId>
                    <Timestamp>2001-12-17T09:30:47.0Z</Timestamp>
                    <RequestTag>_</RequestTag>
                    <VersionId>2</VersionId>
                </MessageHeader>
                <MessageBody>
                    <Customer>
                        <Id>1</Id>
                    </Customer>
                </MessageBody>
            </RequestMessage>
            ]]>
        </data>
        <element path="//MessageHeader/RequestTag" value="${requestTag}"/>
    </message>
    <header>
        <element name="Operation" value="GetCustomer"/>
        <element name="RequestTag" value="${requestTag}"/>
    </header>
    <extract>
        <header name="Operation" variable="${operation}"/>
        <message path="//MessageBody/Customer/Id" variable="${customerId}"/>
```

```
    </extract>
</receive>
```

Knowing the send action of the previous chapter we can identify some common mechanisms that apply for both sending and receiving actions. This time the test uses a predefined message receiver in order to receive the message over a certain transport. Again the test is not aware of the transport details (e.g. JMS connection factory, queue names, etc.) but the message receiver does know this information.

While the action tries to receive a message the whole test execution will be delayed. This is important to ensure the step by step test workflow processing. The receiver will only wait a given amount of time for the message to arrive. A timeout exception fails the test in case the message does not arrive in time.

Once the message has arrived, the content can be validated in various ways. On the one hand you can specify a whole XML message template that you expect. In this case the received XML structure will be compared to the expected XML message element by element. On the other hand you can use explicit element validation where only a small subset of message elements is included into validation.

Besides this message payload validation the framework can also validate the message header values where variable usage is supported as usual during the whole validation process.

In general the validation component (validator) in Citrus works hand in hand with a message reveiving component as the following figure shows:

The message receiving component passes the message to the validator where the validation steps take place. Let us have a closer look at the validation options and features step by step.

5.2.1. Validate message content

Once Citrus has received a message the tester can validate the message contents in various ways. First of all the tester can compare the whole message payload to a predefined control message template.

The receiving action offers following elements for control message templates:

- *<data>*: Defines an inline XML message template as nested CDATA
- *<resource>*: Defines an expected XML message template via external file resources

Both ways inline CDATA XML or external file resource give us a control message template. Citrus uses this control template for extended XML tree comparison. All elements, namespaces, attributes and node values are validated in XML DOM tree comparison. Only in case received message and control message are equal to each other the message validation will succeed. In case differences occur Citrus gives detailed error messages and the test fails.

Now up to now the control message template is very static. Message comparison in this high extend has to be much more robust. This is why Citrus supports various ways to add dynamic message content and ignored elements to the XML tree validation. The tester can enrich the expected

message template with test variables or some elements can be completely ignored in validation.

5.2.2. Dynamic message values

Some elements in our message payload might be of variable nature. Just think of identifiers that should not be static in our expected message template. Instead of repeating the ids several times hardcoded in our test we overwrite those elements with variable values. This can be done with XPath or inline variable declarations. Lets have a look at a example listing showing both ways to overwrite message tempalte content before validation:

```
<message>
    <data>
        <![CDATA[
        <RequestMessage>
            <MessageHeader>
                <CorrelationId>${correlationId}</CorrelationId>
                <Timestamp>2001-12-17T09:30:47.0Z</Timestamp>
                <RequestTag>_</RequestTag>
                <VersionId>2</VersionId>
            </MessageHeader>
            <MessageBody>
                <Customer>
                    <Id>1</Id>
                </Customer>
            </MessageBody>
        </RequestMessage>
        ]]>
    </data>
    <element path="//MessageHeader/RequestTag" value="${requestTag}"/>
</message>
```

The program linsting shows both ways of setting variable values inside a message template. First of all you can simply place variable expressions inside the message (see how ${correlationId} is used). Secondly you can also use XPath expressions to explicitly overwrite message elements before validation.

```
<element path="//MessageHeader/RequestTag" value="${requestTag}"/>
```

The XPath expression evaluates to the message template, searches for the right element and replaces the element value. Of course this works with attributes too.

Both ways via XPath or inline variable expressions are equally to each other. Choose one way that fits best for you. This is how we can add dynamic variable values to the control template in order to increase maintainability and robustness of message validation.

5.2.3. Ignore message elements

Some elements in the message payload might not apply for validation at all. Just think of communication timestamps an dynamic values inside a message:

```
[...]
    <Timestamp>2001-12-17T09:30:47.0Z</Timestamp>
[...]
```

The timestamp value will dynamically change from test run to test run and is hardly predictable for the tester, so lets ignore it in validation.

```
 <message>
    <data>
        <![CDATA[
```

```
        <RequestMessage>
            <MessageHeader>
                <CorrelationId>${correlationId}</CorrelationId>
                <Timestamp>@ignore@</Timestamp>
                <RequestTag>${requestTag}</RequestTag>
                <VersionId>2</VersionId>
            </MessageHeader>
            <MessageBody>
                <Customer>
                    <Id>1</Id>
                </Customer>
            </MessageBody>
        </RequestMessage>
        ]]>
    </data>
    <ignore path="//ResponseMessage/MessageHeader/Timestamp"/>
</message>
```

If you do not like XPath you couls also use another possibility to ignore message contents. The next example uses the special *@ignore@* placeholder directly in the message content:

```
<message>
    <data>
        <![CDATA[
        <RequestMessage>
            <MessageHeader>
                <CorrelationId>${correlationId}</CorrelationId>
                <Timestamp>@ignore@</Timestamp>
                <RequestTag>${requestTag}</RequestTag>
                <VersionId>2</VersionId>
            </MessageHeader>
            <MessageBody>
                <Customer>
                    <Id>1</Id>
                </Customer>
            </MessageBody>
        </RequestMessage>
        ]]>
    </data>
</message>
```

The ignored message elements are automatically skipped when Citrus compares and validates message contents.

5.2.4. Explicit message element validation

In the previous sections we have seen how to validate whole XML fragments with control message templates. In some cases this approach might be too extensive. Imagine the tester only needs to validate a small subset of message elements. The definition of control templates in combination with several ignore statements is not appropiate in this case. You would rather want to use explicit element validation.

```
<message>
    <validate path="//MessageHeader/RequestTag"
                value="${requestTag}"/>
    <validate path="//CorrelationId"
                value="${correlationId}"/>
    <validate path="//MessageBody/Number"
                value="123456789"/>
</message>
```

Instead of comparing the whole message some message elements are validated explicitly over XPath. Citrus evaluates the XPAth expression and and compares its value to the control value. The basic message structure as well as other message elements are not included into validation.

Note

If this type of element validation is choosen neither <data> nor <resource> template definitions are allowed.

Tip

Citrus offers an alternative dot-notated syntax in order to walk through XML trees. In case you are not familiar with XPath or simply need a very easy way to find your element inside the XML tree you might use this way. Every element hierarchy in the XML tree is represented with a simple dot - for example:

```
message.body.text
```

The expression will search the XML tree for the respective <message><body><text> element. Attributes are supported too. In case the last element in the dot-notated expression is a XML attribute the framework will automatically find it.

Of course this dot-notated syntax is very simple and might not be applicable for more complex tree walkings. XPath is much more powerful - no doubt. However the dot-notated syntax might help those of you that are not familiar with XPath. So the dot-notation is supported wherever XPath expressions might apply.

5.2.5. Validate the message header

Now that we have validated the message payload in various ways in the previous sections we are also interested in validating the message header. Just add the following header validation to your receiving action.

```
<header>
    <element name="Operation" value="GetCustomer"/>
    <element name="RequestTag" value="${requestTag}"/>
</header>
```

Message headers often occur as name-value pairs. So each header element that we specify in validation has to be present in the received message. In addition to that the header value has to fit the control value. If one header is not found or the value does not fit Citrus will raise errors and the test case will fail.

Note

Sometimes message headers may not apply to the name-value pair pattern. For example SOAP headers can also contain XML fragments. Citrus supports these kind of headers too. Please see the SOAP chapter for more details on this.

5.2.6. Saving message content to variables

Imagine you receive a message in your test that contains a generated message id. You have no chance to predict the id because it was generated at runtime. In many cases you need to return this id in the respective response message to meet the requirements. So Citrus needs to offer a way to save dynamic message content for reuse in later test steps. The solution is simple: we can extract

those dynamic values to test variables.

```
<extract>
    <header name="Operation" variable="operation"/>
    <message name="//MessageBody/Customer/Id" variable="customerId"/>
</extract>
```

As you can see Citrus is able to store both header and message payload content into test variables. It does not matter if you use new test variables or existing ones as target. The extraction will automatically create a new variable in case it does not exist. Once the variable is announced to the test all following test actions can access the variables as usual. So you could reuse the variable value in response messages or other test steps ahead.

5.2.7. Message selectors

The <*selector*> element inside the receiving action defines key-value pairs in order to filter the messages being received. The key value pairs apply to the message headers. This means that a receiver will only accept messages that meet the key-value pairs in its header. Using this mechanism you can explicitly listen for messages that belong to your test. This is very helpful to avoid receiving messages from other tests that are still available on the message destination.

Lets say the tested software application keeps sending messages that belong to previous test cases. This could happen in retry situations where the application's error handling automatically tries to solve a communication problem that occurred during previous test cases. As a result the EAI application keeps sending messages that are not valid any more for the currently running test case. The test case might fail because the received message does not apply to the currently tested use case. The messages received are simply failing because the message content does not fit the expected one (e.g. correlation-ids, customer informations etc.).

Now we have to find a way to avoid these problems. The test could filter the messages on a destination to only receive messages that apply for the use case that is being tested. The Java Messaging System (JMS) came up with a message header selector that will only accept messages that fit the expected header values.

Let us have a closer look at a message selector inside a receiving action:

```
<selector>
    <element> name="correlationId" value="Lx1x123456789"</element>
    <element> name="operation" value="getOrders"</element>
</selector>
```

This example shows how selectors work. The selector will only accept messages that meet the correlation id and the operation in the header values. All other messages are ignored. The selector elements are associated to each other using logical AND (e.g. where correlationId = 'Lx1x123456789' AND operation = 'getOrders').

You can also define a selector string yourself that gives you more power in constructing the selection logic.

```
<selector>
    <value>
        correlationId = 'Cx1x123456789' OR correlationId = 'Cx1x987654321'
    </value>
</selector>
```

Important

In case you want to run tests in parallel you will need to specify message selectors, otherwise the different tests running at the same time will steal messages from each other. In parallel test execution several test cases will listen for messages at the same time.

At this point you know the two most important test actions in Citrus. Sending and receiving actions will become the main components of your integration tests when dealing with loosely coupled message based components in a SOA. It is very easy to create message flows, meaning a sequence of sending and receiving messages in your test case. You can replicate use cases and test your message exchange with extended message validation possibilities.

Chapter 6. Test actions

This chapter gives a brief description to all test actions that a tester can incorporate into the test case. Besides sending and receiving messages the tester may access these actions in order to build a more complex test scenario that fits the desired use case.

6.1. Connecting to the database

In many cases it is necessary to access the database during a test. This enables a tester to also validate the persistent data in a database. It might also be helpful to prepare the database with some test data before running a test. You can do this using the two database actions that are described in the following sections.

6.1.1. Updating the database

The <sql> action simply executes a group of SQL statements in order to change data in a database. Typically the action is used to prepare the database at the beginning of a test or to clean up the database at the end of a test. You can specify SQL statements like INSERT, UPDATE, DELETE, CREATE TABLE, ALTER TABLE and many more.

On the one hand you can specify the statements as inline SQL or stored in an external SQL resource file as shown in the next two examples.

```
<actions>
    <sql datasource="someDataSource">
        <statement>DELETE FROM CUSTOMERS</statement>
        <statement>DELETE FROM ORDERS</statement>
    </sql>

    <sql datasource="myDataSource">
        <resource file="file:tests/unit/resources/script.sql"/>
    </sql>
</actions>
```

The first action uses inline SQL statements defined directly inside the test case. The next action uses an external SQL resource file instead. The file resource can hold several SQL statements separated by new lines. All statements inside the file are executed sequentially by the framework.

Important

You have to pay attention to some rules when dealing with external SQL resources.

- Each statement should begin in a new line
- It is not allowed to define statements with word wrapping
- Comments begin with two dashes "--"

Note

The external file is referenced either as file system resource or class path resource, by using the "file:" or "classpath:" prefix.

Both examples use the "datasource" attribute. This value defines the database data source to be used. The connection to a data source is mandatory, because the test case does not know about user credentials or database names. The 'datasource' attribute references predefined data sources that are located in a separate Spring configuration file.

6.1.2. Verifying data from the database

The <sql> action is specially designed to execute SQL queries (SELECT * FROM). So the test is able to get data from a database. The query results are validated against expected data as shown in the next example.

```
<sql datasource="testDataSource">
    <statement>select NAME from CUSTOMERS where ID='${customerId}'</statement>
    <statement>select count(*) from ERRORS</statement>
    <statement>select ID from ORDERS where DESC LIKE 'Def%'</statement>
    <statement>select DESCRIPTION from ORDERS where ID='${id}'</statement>

    <validate column="ID" value="1"/>
    <validate column="NAME" value="Deppisch"/>
    <validate column="COUNT(*)" value="${rowsCount}"/>
    <validate column="DESCRIPTION" value="null"/>
</sql>
```

The action <sql> offers a wide range of validating functionality for database result sets. First of all you have to select the data via SQL statements. Here again you have the choice to use inline SQL statements or external file resource pattern.

The result sets are validated through <validate> elements. It is possible to do a detailed check on every selected column of the result set. Simply refer to the selected column name in order to validate its value. The usage of test variables is supported as well as database expressions like count(), avg(), min(), max().

You simply define the <validate> entry with the column name as the "column" attribute and any expected value expression as expected "value". The framework then will check the column to fit the expected value and raise validation errors in case of mismatch.

Looking at the first SELECT statement in the example you will see that test variables are supported in the SQL statements. The framework will replace the variable with its respective value before sending it to the database.

In the validation section variables can be used too. Look at the third validation entry, where the variable "${rowsCount}" is used. The last validation in this example shows, that `NULL` values are also supported as expected values.

If a single validation happens to fail, the whole action will fail with respective validation errors.

Note

When validating database query result sets Citrus can only handle single rows. In case a result set holds several rows only the first row is evaluated.

6.1.3. Read data from database

Now the validation of database entries is a very powerful feature but sometimes we simply do not

know the persisted content values. The test may want to read database entries into test variables without validation. Citrus is able to do that with the following <extract> expressions:

```
<sql datasource="testDataSource">
    <statement>select ID from CUSTOMERS where NAME='${customerName}'</statement>
    <statement>select STATUS from ORDERS where ID='${orderId}'</statement>

    <extract column="ID" variable="${customerId}"/>
    <extract column="STATUS" variable="${orderStatus}"/>
</sql>
```

We can save the database column values directly to test variables. Of course you can combine the value extraction with the normal column validation described earlier in this chapter. Please keep in mind that we can not use these operations on result sets with multiple rows. Citrus will always use the first row in a result set.

6.2. Sleep

This action shows how to make the test framework sleep for a given amount of time. The attribute 'time' defines the amount of time to wait in seconds. As shown in the next example decimal values are supported too. When no waiting time is specified the default time of 5.0 seconds applies.

```
<testcase name="sleepTest">
    <actions>
        <sleep time="3.5"/>

        <sleep/>
    </actions>
</testcase>
```

When should somebody use this action? To us this action was always very useful in case the test needed to wait until an application had done some work. For example in some cases the application took some time to write some data into the database. We waited then a small amount of time in order to avoid unnessesary test failures, because the test framework simply validated the database too early. Or as another example the test may wait a given time until retry mechanisms are triggered in the tested application and then proceed with the test actions.

6.3. Java

The test framework is written in Java and runs inside a Java virtual machine. The functionality of calling other Java objects and methods in this same Java VM through Java Reflection is self-evident. With this action you can call any Java API available at runtime through the specified Java classpath.

The action syntax looks like follows:

```
<java class="com.consol.citrus.test.util.InvocationDummy">
    <constructor>
        <argument type="">Test Invocation</argument>
    </constructor>
    <method name="invoke">
        <argument type="String[]">1,2</argument>
    </method>
</java>

<java class="com.consol.citrus.test.util.InvocationDummy">
    <constructor>
        <argument type="">Test Invocation</argument>
    </constructor>
    <method name="invoke">
```

```
        <argument type="int">4</argument>
        <argument type="String">Test Invocation</argument>
        <argument type="boolean">true</argument>
    </method>
</java>

<java class="com.consol.citrus.test.util.InvocationDummy">
    <method name="main">
        <argument type="String[]">4,Test,true </argument>
    </method>
</java>
```

The Java class is specified by fully qualified class name. Constructor arguments are added using the <constructor> element with a list of <argument> child elements. The type of the argument is defined within the respective attribute "type". By default the type would be String.

The invoked method on the Java object is simply referenced by its name. Method arguments do not bring anything new after knowing the constructor argument definition, do they?.

Method arguments support data type conversion too, even string arrays (useful when calling CLIs). In the third action in the example code you can see that colon separated strings are automatically converted to string arrays.

Simple data types are defined by their name (int, boolean, float etc.). Be sure that the invoked method and class constructor fit your arguments and vice versa, otherwise you will cause errors at runtime.

6.4. Expect timeouts on a destination

In some cases it might be necessary to validate that a message is *not* present on a destination. This means that this action expects a timeout when receiving a message from an endpoint destination. For instance the tester intends to ensure that no message is sent to a certain destination in a time period. In that case the timeout would not be a test aborting error but the expected behavior. And in contrast to the normal behavior when a message is received in the time period the test will fail with error.

In order to validate such a timeout situation the action <expectTimout> shall help. The usage is very simple as the following example shows:

```
<testcase name="receiveJMSimeoutTest">
    <actions>
        <expect-timeout message-receiver="myMessageReceiver" wait="500"/>
    </actions>
</testcase>
```

The action offers two attributes:

- *message-receiver*: Reference to a message receiver that will try to receive messages.
- *wait*: Time period to wait for messages to arrive

TODO: describe receive selected!

6.5. Echo

The <echo> action prints messages to the console/logger. This functionality is useful when debugging test runs. The property "message" defines the text that is printed. Tester might use it to print out debug messages and variables as shown the next code example:

```
<testcase name="echoTest">
    <variables>
        <variable name="date" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <echo>
            <message>Hello Test Framework</message>
        </echo>

        <echo>
            <message>Current date is: ${date}</message>
        </echo>
    </actions>
</testcase>
```

Result on the console:

```
Hello Test Framework
Current time is: 05.08.2008
```

6.6. Time measurement

Time measurement during a test can be very helpful. The <trace-time> action creates and monitors multiple timelines. The action offers the attribute "id" to identify a time line. The tester can of course use more than one time line with different ids simultaneously.

Read the next example and you will understand the mix of different time lines:

```
<testcase name="timeWatcherTest_new">
    <actions>
        <trace-time/>

        <trace-time id="time_line_id"/>

        <sleep time="3.5"/>

        <trace-time id=" time_line_id "/>

        <sleep time="5.0"/>

        <trace-time/>

        <trace-time id=" time_line_id "/>
    </actions>
</testcase>
```

The test output looks like follows:

```
Starting TimeWatcher:
Starting TimeWatcher: time_line_id
TimeWatcher time_line_id after 3.5 seconds
TimeWatcher after 8.5 seconds
TimeWatcher time_line_id after 8.5 seconds
```

Note

In case no time line id is specified the framework will measure the time for a default time line.

To print out the current elapsed time for a time line you simply have to place the <trace-time> action into the action chain again and again, using the respective time line identifier. The elapsed time will be printed out to the console every time.

6.7. Create variables

As you know variables usually are defined at the beginning of the test case (Section 3.1.2, “Variables”). It might also be helpful to reset existing variables as well as to define new variables during the test. The action <create-variables> is able to declare new variables or overwrite existing ones.

```
<testcase name="createVariablesTest">
    <variables>
        <variable name="myVariable" value="12345"/>
        <variable name="id" value="54321"/>
    </variables>
    <actions>
        <echo>
            <message>Current variable value: ${myVariable} </message>
        </echo>

        <create-variables>
            <variable name="myVariable" value="${id}"/>
            <variable name="newVariable" value="'this is a test'"/>
        </create-variables>

        <echo>
            <message>Current variable value: ${myVariable} </message>
        </echo>

        <echo>
            <message>
              New variable 'newVariable' has the value: ${newVariable}
            </message>
        </echo>
    </actions>
</testcase>
```

The new variables are valid for the rest of the test. Actions reference them as usual through a variable expression.

6.8. Trace variables

You already know the <echo> action that prints messages to the console or logger. The <trace-variables> action is specially designed to trace all currently valid test variables to the console. This was mainly used by us for debug reasons. The usage is quite simple:

```
<testcase name="traceVariablesTest">
    <variables>
        <variable name="myVariable" value="12345"/>
        <variable name="nextVariable" value="54321"/>
    </variables>
    <actions>
        <trace-variables>
            <variable name="myVariable"/>
            <variable name="nextVariable"/>
        </trace-variables>

        <trace-variables/>
    </actions>
</testcase>
```

Simply add the <trace-variables> action to your action chain and all variables will be printed out to

the console. You are able to define a special set of variables by using the <variable> child elements. See the output that was generated by the test example above:

```
Current value of variable myVariable = 12345
Current value of variable nextVariable = 54321
```

6.9. Groovy support

Groovy is an agile dynamic language for the Java Platform. Groovy ships with a lot of very powerful features and fits perfectly with Java as it is based on Java and runs inside the JVM.

The Citrus Groovy support might be the entrance for you to write customized test actions. You can easily execute Groovy code inside a test case, just like a normal test action. The whole test context with all variables is available to the Groovy action. This means someone can change variable values or create new variables very easily.

Lets have an example to show the possible Groovy code interactions in Citrus:

```
<testcase name="groovyTest">
    <variables>
        <variable name="time" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <groovy>
            println 'Hello TestSuite'
        </groovy>
        <groovy>
            println 'The variable is: ${time}'
        </groovy>
        <groovy>
            <![CDATA[
                import com.consol.citrus.*
                import com.consol.citrus.variable.*
                import com.consol.citrus.context.TestContext
                import com.consol.citrus.script.GroovyAction.ScriptExecutor

                public class GScript implements ScriptExecutor {
                    public void execute(TestContext context) {
                        println context.getVariable("time")
                    }
                }
            ]]>
        </groovy>
        <groovy resource="classpath:com/consol/citrus/script/example.groovy"/>
    </actions>
</testcase>
```

As you can see it is possible to write Groovy code directly into the test case. Citrus will execute the Groovy code and handle variable references automatically. The tester is also able to write whole Groovy classes. The ScriptExecutor interface defines the execute method and provides access to the TestContext and its test variables. Citrus will take care of the TestContext injection as a method parameter.

The last test action in our example refers to an external Groovy file resource. The external resource holds the Groovy code that is included into the test action chain.

6.10. Failing the test

The fail action will generate an exception in order to terminate the test case with error. The test case will therefore not be successful in the reports.

The user can specify a custom error message for the exception in order to describe the error cause. Here is a very simple example to clarify the syntax:

```
<testcase name="failTest">
    <actions>
        <fail message="Test will fail with custom message"/>
    </actions>
</testcase>
```

Test results:

```
Execution of test: failTest failed! Nested exception is:
com.consol.citrus.exceptions.TestSuiteException:
Test will fail with custom message

[...]

TEST RESULTS coreTestSuite

failTest          : failed - Exception is: Test will fail with custom message

Found 1 test cases to execute
Skipped 0 test cases (0.0%)
Executed 1 test cases, containing 3 actions
Tests failed:        1 (100.0%)
Tests successfully:  0 (0.0%)
```

6.11. Input

During the test case execution it is possible to read some user input from the command line. The test execution will stop and wait for keyboard inputs over the standard input stream. The user has to type the input and end it with the return key.

The user input is stored to the respective variable value.

```
<testcase name="inputTest">
    <variables>
        <variable name="userinput" value=""></variable>
        <variable name="userinput1" value=""></variable>
        <variable name="userinput2" value="y"></variable>
        <variable name="userinput3" value="yes"></variable>
        <variable name="userinput4" value=""></variable>
    </variables>
    <actions>
        <input/>
        <echo><message>user input was: ${userinput}</message></echo>

        <input message="Now press enter:" variable="userinput1"/>
        <echo><message>user input was: ${userinput1}</message></echo>

        <input message="Do you want to continue?"
                  valid-answers="y/n" variable="userinput2"/>
        <echo><message>user input was: ${userinput2}</message></echo>

        <input message="Do you want to continue?"
                  valid-answers="yes/no" variable="userinput3"/>
        <echo><message>user input was: ${userinput3}</message></echo>

        <input variable="userinput4"/>
        <echo><message>user input was: ${userinput4}</message></echo>
    </actions>
</testcase>
```

The input action has following attributes:

- *message* -> message displayed to the user

- *valid-answers* -> optional slash separated string containing the possible answers
- *variable* -> destination variable to store the user input (default = ${userinput})

Note

When user input is restricted to a set of valid answers the input validation of course can fail due to mismatch. In this case the user is again asked to provide the input until a valid answer is given.

Note

User inputs may not fit to automatic testing in terms of continuous integration testing where no user is present to type in the correct answer over the keyboard. In this case you can always skip the user input in advance by specifying a variable that matches the user input variable name. As the user input variable is then already present the user input is missed out and the test proceeds automatically.

6.12. Load

You are able to load properties from external property files and store them as test variables. The action will require a file resource either from class path or file system in order to read the property values.

Let us look at an example to get an idea about this action:

Content of load.properties:

```
username=Mickey Mouse
greeting.text=Hello Test Framework
```

```
<testcase name="loadPropertiesTest">
    <actions>
        <load>
            <properties file="file:tests/resources/load.properties"/>
        </load>

        <trace-variables/>
    </actions>
</testcase>
```

Output:

```
Current value of variable username = Mickey Mouse
Current value of variable greeting.text = Hello Test Framework
```

The action will load all available properties in the file load.properties and store them to the test case as local variables.

Important

Existing variables are overwritten!

6.13. Purging JMS destinations

Purging JMS destinations during the test run is quite essential. Different test cases can influence each other when sending messages to the same JMS destinations. A test case should only receive those messages that actually belong to it. Therefore it is a good idea to purge all JMS queue destinations between the test cases. Obsolete messages that are stuck in a JMS queue for some reason are then removed so that the following test case is not offended.

So we need to purge some JMS queues in out test case. This can be done with following action definition:

```
<testcase name="purgeTest">
  <actions>
      <purge-jms-queues>
          <queue name="Some.JMS.QUEUE.Name"/>
          <queue name="Another.JMS.QUEUE.Name"/>
          <queue name="My.JMS.QUEUE.Name"/>
      </purge-jms-queues>

      <purge-jms-queues connection-factory="connectionFactory">
          <queue name="Some.JMS.QUEUE.Name"/>
          <queue name="Another.JMS.QUEUE.Name"/>
          <queue name="My.JMS.QUEUE.Name"/>
      </purge-jms-queues>
  </actions>
</testcase>
```

Purging the JMS queues in every test case is quite exhausting because every test case needs to define a purging action at the very beginning of the test. Fortunately the test suite definition offers tasks to run before, between and after the test cases which should ease up this tasks a lot. The test suite offers a very simple way to purge the destinations between the tests. See Section 16.1, “Tasks before, between and after the test run” for more information about this.

When using the special tasks between a test case you might define a normal Spring bean definition that is referenced then. The 'com.consol.citrus.actions.PurgeJmsQueuesAction' action offers the property "queueNames" to hold all destination names that are supposed to be purged. As you can see in the next example it is quite easy to specify a group of destinations in the Spring configuration. This purging bean is then added to the test suite in the tasks between section.

```
<bean id="purgeJmsQueues"
    class="com.consol.citrus.actions.PurgeJmsQueuesAction">
    <property name="connectionFactory" ref="jmsQueueConnectionFactory"/>
    <property name="queueNames">
        <list>
            <value>${jms.queue.hello.request.in}</value>
            <value>${jms.queue.hello.response.out}</value>
            <value>${jms.queue.echo.request}</value>
            <value>${jms.queue.echo.response}</value>
            <value>JMS.Queue.Dummy</value>
        </list>
    </property>
</bean>
```

So now we are able to purge JMS destinations with given destination names. But sometimes we do not want to rely on queue or topic names as we retrieve destinations over JNDI for instance. We can deal with destinations coming from JNDI lookup like follows:

```
<jee:jndi-lookup id="jmsQueueHelloRequestIn" jndi-name="jms/jmsQueueHelloRequestIn"/>
<jee:jndi-lookup id="jmsQueueHelloResponseOut" jndi-name="jms/jmsQueueHelloResponseOut"/>

<bean id="purgeJmsQueues"
    class="com.consol.citrus.actions.PurgeJmsQueuesAction">
    <property name="connectionFactory" ref="jmsQueueConnectionFactory"/>
```

```
    <property name="queues">
        <list>
            <ref bean="jmsQueueHelloRequestIn"/>
            <ref bean="jmsQueueHelloResponseOut"/>
        </list>
    </property>
</bean>
```

We just use the property *'queues'* instead of *'queueNames'* and Citrus will be able to receive bean references that resolve to JMS destinations. We can purge these destination references in a test case, too. Just use the *'ref'* attribute instead of already known *'name'* attribute:

```
<testcase name="purgeTest">
  <actions>
      <purge-jms-queues>
          <queue ref="jmsQueueHelloRequestIn"/>
          <queue ref="jmsQueueHelloResponseOut"/>
      </purge-jms-queues>
  </actions>
</testcase>
```

6.14. Assert failure

Citrus test actions fail with Java exceptions and error messages. This gives you the opportunity to expect an action to fail during test execution. You can simple assert a Java exception to be thrown during execution. See the example for an assert action definition in a test case:

```
<testcase name="assertFailureTest">
    <actions>
        <assert exception="com.consol.citrus.exceptions.CitrusRuntimeException"
                message="Unknown variable ${date}">
            <echo>
                <message>Current date is: ${date}</message>
            </echo>
        </assert>
    </actions>
</testcase>
```

Note

Note that the assert action requires an exception. In case no exception is thrown by the embedded test action the assertion and the test case will fail!

The assert action always wraps a single test action, which is then monitored for failure. In case the nested test action fails with error you can validate the error in its type and error message (optional). The failure has to fit the expected one exactly otherwise the assertion fails itself.

Important

Important to notice is the fact that asserted exceptions do not cause failure of the test case. As you except the failure to happen the test continues with its work once the assertion is done successfully.

6.15. Catch exceptions

In the previous chapter we learned how to expect failures in Citrus with assert action. Now the assert

action is designed for one single action to be monitored for failures. The *'catch'* action in contrary will hold several nested test actions and catch possible errors. The nested actions are error proof for the chosen exception type, but only for the chosen exception type other exception types may cause the test to fail.

```
<testcase name="catchExceptionTest">
    <actions>
        <catch exception="com.consol.citrus.exceptions.CitrusRuntimeException">
            <echo>
                <message>Current date is: ${date}</message>
            </echo>
        </catch>
    </actions>
</testcase>
```

Important

Note that there is no validation available in a catch block. So catching exceptions is just to make a test more stable towards errors that can occur. The caught exception does not cause any failure in the test. The test case may continue with execution as if there was not failure. Also notice that the catch action is also happy when no exception at all is caught. In contrary to the assert action which requires the failure the catch block is not failing in positive processing.

Catching exceptions like this may only fit to very error prone action blocks where failures do not harm the test case success. Otherwise a failure in a test action should always reflect to the whole test case to fail with errors.

6.16. Including own actions

The generic <action> element references Spring beans that implement the Java interface `com.consol.citrus.TestAction`. This is a very fast way to add your own action implementations to a test case. You can implement own actions in Java and include them into a test case.

```
<testcase name="actionReferenceTest">
    <actions>
        <action reference="cleanUpDatabase"/>
        <action reference="mySpecialAction"/>
    </actions>
</testcase>
```

In the example above the called actions are special database cleanup implementations. The actions are defined as Spring beans in the Citrus configuration and get referenced by their bean name or id.

Chapter 7. Templates

Templates group action sequences to a logical unit. You can think of templates as reusable components that are used in several tests. The maintenance is much more effective because the templates are referenced several times.

The template always has a unique name. Inside a test case we call the template by this unique name. Have a look at a first example:

```
<template name="doCreateVariables">
    <create-variables>
        <variable name="var" value="123456789"/>
    </create-variables>

    <call-template name="doTraceVariables"/>
</template>

<template name="doTraceVariables">
    <echo>
        <message>Current time is: ${time}</message>
    </echo>

    <trace-variables/>
</template>
```

The code example above describes two template definitions. Templates hold a sequence of test actions or call other templates themselves as seen in the example above.

Note

The <call-template> action calls other templates by their name. The called template not necessarily has to be located in the same test case XML file. The template might be defined in a separate XML file other than the test case itself:

```
<testcase name="templateTest">
    <variables>
        <variable name="myTime" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <call-template name="doCreateVariables"/>

        <call-template name="doTraceVariables">
            <parameter name="time" value="${myTime}">
        </call-template>
    </actions>
</testcase>
```

There is an open question when dealing with templates that are defined somewhere else outside the test case. How to handle variables? A templates may use different variable names then the test and vice versa. No doubt the template will fail as soon as special variables with respective values are not present. Unknown variables cause the template and the whole test to fail with errors.

So a first approach would be to harmonize variable usage across templates and test cases, so that templates and test cases do use the same variable naming. But this approach might lead to high calibration effort. Therefore templates support parameters to solve this problem. When a template is called the calling actor is able to set some parameters. Let us discuss an example for this issue.

The template "doDateCoversion" in the next sample uses the variable ${date}. The calling test case can set this variable as a parameter without actually declaring the variable in the test itself:

```
<call-template name="doDateCoversion">
    <parameter name="date" value="${sampleDate}">
</call-template>
```

The variable *sampleDate* is already present in the test case and gets translated into the *date* parameter. Following from that the template works fine although test and template do work on different variable namings.

With template parameters you are able to solve the calibration effort when working with templates and variables. It is always a good idea to check the used variables/parameters inside a template when calling it. There might be a variable that is not declared yet inside your test. So you need to define this value as a parameter.

Template parameters may contain more complex values like XML fragments. The call-template action offers following CDATA variation for defining complex parameter values:

```
<call-template name="printXMLPayload">
    <parameter name="payload">
      <value>
        <![CDATA[
          <HelloRequest xmlns="http://www.consol.de/schemas/samples/sayHello.xsd">
            <Text>Hello South ${var}</Text>
          </HelloRequest>
        ]]>
      </value>
    </parameter>
</call-template>
```

Important

When a template works on variable values and parameters changes to these variables will automatically affect the variables in the whole test. So if you change a variable's value inside a template and the variable is defined inside the test case the changes will affect the variable in a global context. We have to be careful with this when executing a template several times in a test, especially in combination with parallel containers (see Section 8.2, “Parallel”).

```
<parallel>
    <call-template name="print">
        <parameter name="param1" value="1"/>
        <parameter name="param2" value="Hello Europe"/>
    </call-template>
    <call-template name="print">
        <parameter name="param1" value="2"/>
        <parameter name="param2" value="Hello Asia"/>
    </call-template>
    <call-template name="print">
        <parameter name="param1" value="3"/>
        <parameter name="param2" value="Hello Africa"/>
    </call-template>
</parallel>
```

In the listing above a template *print* is called several times in a parallel container. The parameter values will be handled in a global context, so it is quite likely to happen that the template instances influence each other during execution. We might get such print messages:

```
2. Hello Europe
2. Hello Africa
3. Hello Africa
```

Index parameters do not fit and the message *'Hello Asia'* is completely gone. This is because templates overwrite parameters to each other as they are executed in parallel at the same time. To avoid this behavior we need to tell the template that it should handle parameters as well as variables in a local context. This will enforce that each template instance is working on a dedicated local context. See the *global-context* attribute that is set to *false* in this example:

```
<template name="print" global-context="false">
    <echo>
        <message>${param1}.${param2}</message>
    </echo>
</template>
```

After that template instances won't influence each other anymore. But notice that variable changes inside the template then do not affect the test case neither.

Chapter 8. Containers

Similar to templates a container element holds one to many test actions. In contrast to the template the container appears directly inside the test case action chain, meaning that the container is not referenced by more than one test case.

Containers execute the embedded test actions in specific logic. This can be an execution in iteration for instance. Combine different containers with each other and you will be able to generate very powerful hierarchical structures in order to create a complex execution logic. In the following sections some predefined containers are described.

8.1. Sequential

The sequential container executes the embedded test actions in strict sequence. Readers now might search for the difference to the normal action chain that is specified inside the test case. The actual power of sequential containers does show only in combination with other containers like iterations and parallels. We will see this later when handling these containers.

For now the sequential container seems not very sensational - one might say boring - because it simply groups a pair of test actions to sequential execution.

```
<testcase name="sequentialTest">
    <actions>
        <sequential>
            <trace-time/>
            <sleep/>
            <echo>
                <message>Hallo TestFramework</message>
            </echo>
            <trace-time/>
        </sequential>
    </actions>
</testcase>
```

8.2. Parallel

Parallel containers execute the embedded test actions concurrent to each other. Every action in this container will be executed in a separate Java Thread. Following example should clarify the usage:

```
<testcase name="parallelTest">
    <actions>
        <parallel>
            <sleep/>

            <sequential>
                <sleep/>
                <echo>
                    <message>1</message>
                </echo>
            </sequential>

            <echo>
                <message>2</message>
            </echo>

            <echo>
                <message>3</message>
            </echo>

            <iterate condition="i lt= 5"
                        index="i">
```

```
                <echo>
                    <message>10</message>
                </echo>
            </iterate>
        </parallel>
    </actions>
</testcase>
```

So the normal test action processing would be to execute one action after another. As the first action is a sleep of five seconds, the whole test processing would stop and wait for 5 seconds. Things are different inside the parallel container. Here the descending test actions will not wait but execute at the same time.

Note

Note that containers can easily wrap other containers. The example shows a simple combination of sequential and parallel containers that will archive a complex execution logic. Actions inside the sequential container will execute one after another. But actions in parallel will be executed at the same time.

8.3. Iterate

Iterations are very powerful elements when describing complex logic. The container executes the embedded actions several times. The container will continue with looping as long as the defined breaking condition string evaluates to `true`. In case the condition evaluates to `false` the iteration will break an finish execution.

```
<testcase name="iterateTest">
    <actions>
        <iterate index="i" condition="i lt 5">
            <echo>
                <message>index is: ${i}</message>
            </echo>
        </iterate>
    </actions>
</testcase>
```

The attribute "index" automatically defines a new variable that holds the actual loop index starting at "1". This index variable is available as a normal variable inside the iterate container. Therefore it is possible to print out the actual loop index in the echo action as shown in the above example.

The condition string is mandatory and describes the actual end of the loop. In iterate containers the loop will break in case the condition evaluates to `false`.

The condition string can be any Boolean expression and supports several operators:

- lt (lower than)
- lt= (lower than equals)
- gt (greater than)
- gt= (greater than equals)
- = (equals)

- and (logical combining of two Boolean values)
- or (logical combining of two Boolean values)
- () (brackets)

Important

It is very important to notice that the condition is evaluated before the very first iteration takes place. The loop therefore can be executed 0-n times according to the condition value.

8.4. Repeat until true

Quite similar to the previously described iterate container this repeating container will execute its actions in a loop according to an ending condition. The condition describes a Boolean expression using the operators as described in the previous chapter.

Important

The loop continues its work until the provided condition evaluates to true. It is very important to notice that the repeat loop will execute the actions before evaluating the condition. This means the actions get executed 1-n times.

```
<testcase name="iterateTest">
    <actions>
        <repeat-until-true index="i" condition="(i = 3) or (i = 5)">
            <echo>
                <message>index is: ${i}</message>
            </echo>
        </repeat-until-true>
    </actions>
</testcase>
```

8.5. Repeat on error until true

The next looping container is called repeat-on-error-until-true. This container repeats a group of actions in case one embedded action failed with error. In case of an error inside the container the loop will try to execute `all` embedded actions again in order to seek for overall success. The execution continues until all embedded actions were processed successfully `or` the ending condition evaluates to true and the error-loop will lead to final failure.

```
<testcase name="iterateTest">
    <actions>
        <repeat-onerror-until-true index="i" condition="i = 5">
            <echo>
                <message>index is: ${i}</message>
            </echo>
            <fail/>
        </repeat-onerror-until-true>
    </actions>
</testcase>
```

In the code example the error-loop continues four times as the <fail> action definitely fails the test.

During the fifth iteration The condition "i=5" evaluates to true and the loop breaks its processing leading to a final failure as the test actions were not successful.

Note

The overall success of the test case depends on the error situation inside the repeat-onerror-until-true container. In case the loop breaks because of failing actions and the loop will discontinue its work the whole test case is failing too. The error loop processing is successful in case all embedded actions were not raising any errors during an iteration.

The repeat-on-error container also offers an automatic sleep mechanism. This auto-sleep property will force the container to wait a given amount of time before executing the next iteration. We used this mechanism a lot when validating database entries. Let's say we want to check the existence of an order entry in the database. Unfortunately the system under test is not very performant and may need some time to store the new order. This amount of time is not predictable, especially when dealing with different hardware on our test environments (local testing vs. server testing). Following from that our test case may fail unpredictable only because of runtime conditions.

We can avoid unstable test cases that are based on these runtime conditions with the auto-sleep functionality.

```
<repeat-onerror-until-true auto-sleep="1" condition="i = 5" index="i">
    <echo>
        <sql datasource="testDataSource">
            <statement>
              SELECT COUNT(1) AS CNT_ORDERS
              FROM ORDERS
              WHERE CUSTOMER_ID='${customerId}'
            </statement>
            <validate column="CNT_ORDERS" value="1"/>
        </sql>
    </echo>
</repeat-onerror-until-true>
```

We surrounded the database check with a repeat-onerror container having the auto-sleep property set to 1 second. The repeat container will try to check the database up to five times with an automatic sleep of 1 second before every iteration. This gives the system under test up to five seconds time to store the new entry to the database. The test case is very stable and just fits to the hardware environment. On slow test environments the test may need several iterations to successfully read the database entry. On very fast environments the test may succeed right on the first try.

So fast environments are not slowed down by static sleep operations and slower environments are still able to execute this test case with high stability.

Chapter 9. Finally section

This chapter deals with a special section inside the test case that is executed even in case errors did occur during the test. Lets say you have started a Jetty web server instance at the beginning of the test case and you need to shutdown the server when the test has finished its work. Or as a second example imagine that you have prepared some data inside the database at the beginning of your test and you want to make sure that the data is cleaned up at the end of the test case.

In both situations we might run into some problems when the test failed. We face the problem that the whole test case will terminate immediately in case of errors. Cleanup tasks at the end of the test action chain may not be executed correctly.

Dirty states inside the database or still running server instances then might cause problems for following test cases. To avoid this problems you should use the finally block of the test case. The <finally> section contains actions that are executed even in case the test fails. Using this strategy the database cleaning tasks mentioned before will find execution in every case (success or failure).

The following example shows how to use the finally section at the end of a test:

```
<testcase name="finallyTest">
    <variables>
        <variable name="orderId" value="1"/>
        <variable name="date" value="citrus:currentDate('dd.MM.yyyy')"/>
    </variables>
    <actions>
        <sql datasource="testDataSource">
            <statement>
                INSERT INTO ORDERS VALUES (${orderId}, 1, 1, '${date}')
            </statement>
        </sql>

        <echo>
            <message>
                ORDER creation time: ${date}
            </message>
        </echo>
    </actions>
    <finally>
        <sql datasource="testDataSource">
            <statement>
              DELETE FROM ORDERS WHERE ORDER_ID='${orderId}'
            </statement>
        </sql>
    </finally>
</testcase>
```

In the example the first action creates an entry in the database using an `INSERT` statement. To be sure that the entry in the database is deleted after the test, the finally section contains the respective `DELETE` statement that is always executed regardless the test case state (successful or failed).

Chapter 10. Using XPath

Some time ago in this document we have seen how XML elements are manipulated using XPath expressions when sending and receiving messages. Now using XPath might raise some problems regarding namespaces that we want to deal with now.

XPath is a very powerful technology for walking XML trees. This W3C standard stands for advanced XML tree handling using a special syntax as query language. The test framework supports this XPath syntax in the following fields:

- <message><element path="[XPath-Expression]"></message>
- <extract><message path="[XPath-Expression]"></extract>
- <ignore path="[XPath-Expression]"/>
- <validate path="[XPath-Expression]"/>

The next program listing indicates the power in using XPath with Citrus:

```
<message>
    <validate path="//User/Name" value="John"></validate>
    <validate path="//User/Address[@type='office']/Street" value="Companystreet 21"></validate>
    <validate path ="//User/Name" value="${userName}"></validate>
    <validate path ="//User/@isAdmin" value="${isAdmin}"></validate>
    <validate path ="//*[.='search-for']" value="searched-for"></validate>
</message>
```

10.1. Handling XML namespaces

When it comes to XML namespaces you have to be careful with your XPath expressions. Lets have a look at an example message that uses XML namespaces:

```
<ns1:RequestMessage xmlns:ns1="http://testsuite/default">
    <ns1:MessageHeader>
        <ns1:CorrelationId>_</ns1:CorrelationId>
        <ns1:Timestamp>2001-12-17T09:30:47.0Z</ns1:Timestamp>
        <ns1:RequestTag>_</ns1:RequestTag>
        <ns1:VersionId>2</ns1:VersionId>
    </ns1:MessageHeader>
    <ns1:MessageBody>
        <ns1:Customer>
            <ns1:Id>1</ns1:Id>
        </ns1:Customer>
    </ns1:MessageBody>
</ns1:RequestMessage>
```

Now we would like to validate some elements in this message using XPath

```
<message>
    <validate path="//RequestMessage/MessageHeader/RequestTag" value="${requestTag}"/>
    <validate path="//RequestMessage/MessageHeader/CorrelationId" value="${correlationId}"/>
</message>
```

The validation will fail although the XPath expression looks correct regarding the XML tree. Because the message uses the namespace `xmlns:ns1="http://testsuite/default"` with its prefix `ns1` our XPath expression is not able to find the elements. The correct XPath expression uses the

namespace prefix as defined in the message.

```
<message>
    <validate path="//ns1:RequestMessage/ns1:MessageHeader/ns1:RequestTag" value="${requestTag}"/>
    <validate path="//ns1:RequestMessage/ns1:MessageHeader/ns1:CorrelationId" value="${correlationId}"/>
</message>
```

Now the expressions work fine and the validation is successful. But this is quite error prone. This is because the test is now depending on the namespace prefix that is used by some application. As soon as the message is sent with a different namespace prefix (e.g. ns2) the validation will fail again.

You can avoid this effect when specifying your own namespace context and your own namespace prefix during validation.

```
<message>
    <validate path="//pfx:RequestMessage/pfx:MessageHeader/pfx:RequestTag" value="${requestTag}"/>
    <validate path="//pfx:RequestMessage/pfx:MessageHeader/pfx:CorrelationId" value="${correlationId}"/>
    <namespace prefix="pfx" value="http://testsuite/default"/>
</message>
```

Now the test in independent from any namespace prefix in the received message. The namespace context will resolve the namespaces and find the elements although the message might use different prefixes. The only thing that matters is that the namespace value (http://testsuite/default) matches.

10.2. Handling default namespaces

In the previous section we have seen that XML namespaces can get tricky with XPath validation. Default namespaces can do even more! So lets look at the example with default namespaces:

```
<RequestMessage xmlns="http://testsuite/default">
    <MessageHeader>
        <CorrelationId>_</CorrelationId>
        <Timestamp>2001-12-17T09:30:47.0Z</Timestamp>
        <RequestTag>_</RequestTag>
        <VersionId>2</VersionId>
    </MessageHeader>
    <MessageBody>
        <Customer>
            <Id>1</Id>
        </Customer>
    </MessageBody>
</RequestMessage>
```

The message uses default namespaces. The following approach in XPath will fail due to namespace problems.

```
<message>
    <validate path="//RequestMessage/MessageHeader/RequestTag" value="${requestTag}"/>
    <validate path="//RequestMessage/MessageHeader/CorrelationId" value="${correlationId}"/>
</message>
```

Even default namespaces need to be specified in the XPath expressions. Look at the following code listing that works fine with default namespaces:

```
<message>
    <validate path="//:RequestMessage/:MessageHeader/:RequestTag" value="${requestTag}"/>
    <validate path="//:RequestMessage/:MessageHeader/:CorrelationId" value="${correlationId}"/>
</message>
```

Tip

It is recommended to use the namespace context as described in the previous chapter when validating. Only this approach ensures flexibility and stable test cases regarding namespace changes.

Chapter 11. Connecting with JMS

Citrus provides support for sending and receiving JMS messages. We have to separate between synchronous and asynchronous communication. So in this chapter we explain how to setup JMS message senders and receivers for synchronous and asynchronous outbound and inbound communication

Note

Citrus provides a "citrus" configuration namespace and schema definition. Include this namespace into your Spring configuration in order to use the Citrus configuration elements. The namespace URI and schema location are added to the Spring configuration XML file as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus="http://www.citrusframework.org/schema/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.citrusframework.org/schema/config
       http://www.citrusframework.org/schema/config/citrus-config.xsd">

    [...]

</beans>
```

After that you are able to use customized Citrus XML elements in order to define the Spring beans.

11.1. JMS message sender

First of all we deal with asynchronous message senders, which means that Citrus is publishing messages to a JMS destination (queue or topic). The test case itself does not know about JMS transport details like queue names or connection credentials. This information is stored in the basic Spring configuration. So let us have a look at a simple JMS message sender configuration in Citrus.

```
<bean id="connectionFactory"
         class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<citrus:jms-message-sender id="getOrdersRequestSender"
                           destination-name="Citrus.JMS.Order.Queue.Out"/>
```

The JMS connection factory is responsible for connecting to a JMS message broker. In this example we use the Apache ActiveMQ connection factory implementation as we use a ActiveMQ message broker.

Tip

Spring makes it very easy to connect to other JMS broker implementations too (e.g. TIBCO Enterprise Messaging Service, IBM Websphere MQ). Just substitute the implementing class in the connectionFactory bean.

Note

All of the JMS senders and receivers that require a reference to a JMS connection factory will automatically look for a bean named "connectionFactory" by default. That is why you don't see a "connection-factory" attribute in many of the examples. However, you can use the "connection-factory" attribute in order to use other connection factory instances with different bean names.

```
<citrus:jms-message-sender id="getOrdersRequestSender"
                           destination-name="Citrus.JMS.Order.Queue.Out"
                           connection-factory="myConnectionFacotry"/>
```

Alternatively you may want to directly specify a Spring jmsTemplate.

```
<citrus:jms-message-sender id="getOrdersRequestSender"
                           destination-name="Citrus.JMS.Order.Queue.Out"
                           jms-template="myJmsTemplate"/>
```

The message sender is now ready for usage inside a test. Many sending actions and test cases reference the message sender. The message sender will simply publish the message to the defined JMS destination. The communication is supposed to be asynchronous, which means that the sender will not wait for a synchronous response. The sender fires and forgets the message immediately.

11.2. JMS message receiver

Now lets deal with receiving an async message over JMS. The message receiver definition is located again in the Spring configuration files. We assume that a connection factory has been configured as shown in the previous section.

```
<citrus:jms-message-receiver id="getOrdersResponseReceiver"
                             destination-name="Citrus.JMS.Order.Queue.In"/>
```

The receiver acts as a message driven listener. This means that the message receiver connects to the given destination and waits for messages to arrive.

Note

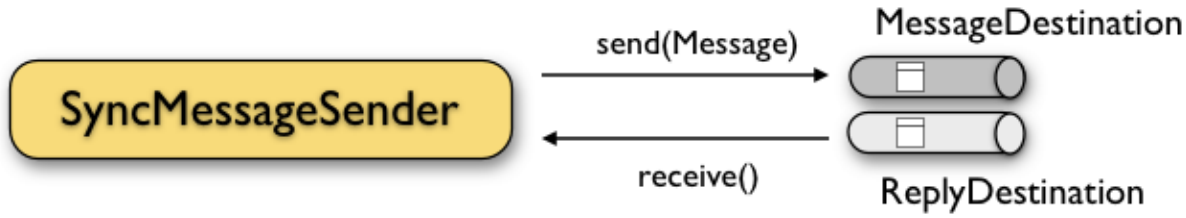
Besides the destination-name attribute you can also provide a reference to a Destination implementation.

```
<citrus:jms-message-receiver id="getOrdersResponseReceiver"
                             destination="orderInboundQueue"/>
```

This destination reference applies to all JMS aware message sender and receiver implementations.

11.3. JMS synchronous message sender

When using synchronous message senders Citrus will define a reply-to-queue destination in the message header and wait synchronously for the response on this destination.

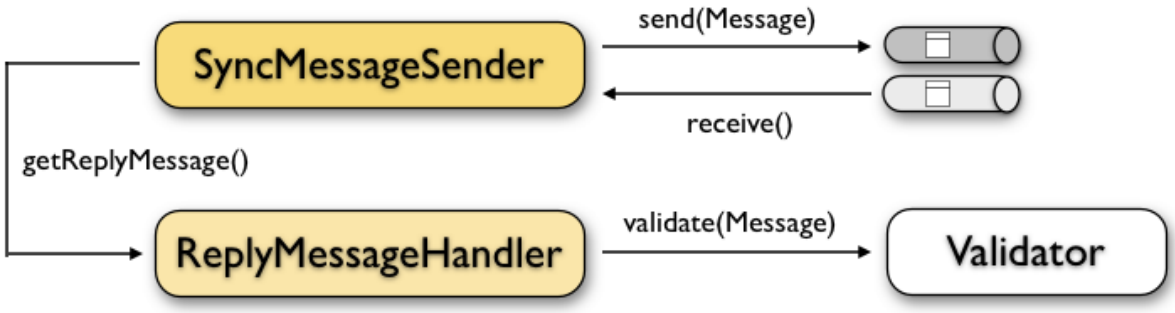


In the Spring XML configuration the synchronous message senders are quite similar to the asynchronous brothers.

```
<citrus:jms-sync-message-sender id="getCustomerRequestSender"
                           destination-name="Citrus.JMS.Customer.Queue.Out"
                           reply-handler="getCustomerReplyHandler"
                           reply-timeout="1000"/>

<citrus:jms-reply-message-handler id="getCustomerReplyHandler"/>
```

To build synchronous outbound communication we need both a synchronous message sender and a reply handler. Both are defined in the Spring configuration. The sender sends the message and waits for the response synchronously. Once the reply has arrived the reply handler is invoked with the respective reply message. The second possibility would be a timeout while receiving the reply.

See the following figure which tries to explain the handshake between synchronous sender component and synchronous reply handler. The synchronous sender receives the reply message and provides it to the reply handler. Once the reply handler has received the reply message it can be handed over to a validator component as usual for extended message validation.



The respective test case makes use of both synchronous message sender and reply handler in order to complete the synchronous communication steps.

```
<testcase name="syncMessagingTest">
    <actions>
        <send with="mySyncMessageSender">
            <message>
                <data>
                    [...]
                </data>
            </message>
        </send>

        <receive with="myReplyMessageHandler">
            <message>
                <data>
                    [...]
                </data>
            </message>
        </receive>
    </actions>
</testcase>
```

Note

The message sender creates a temporary JMS reply destination by default in order to receive the reply. The temporary destination name is stored to the JMS replyTo message header. You can also define a static reply destination like follows.

```
<citrus:jms-sync-message-sender id="getCustomerRequestSender"
                destination-name="Citrus.JMS.Customer.Queue.Out"
                reply-destination-name="Citrus.JMS.Customer.Queue.Reply"
                reply-handler="getCustomerReplyHandler"
                reply-timeout="1000"/>
```

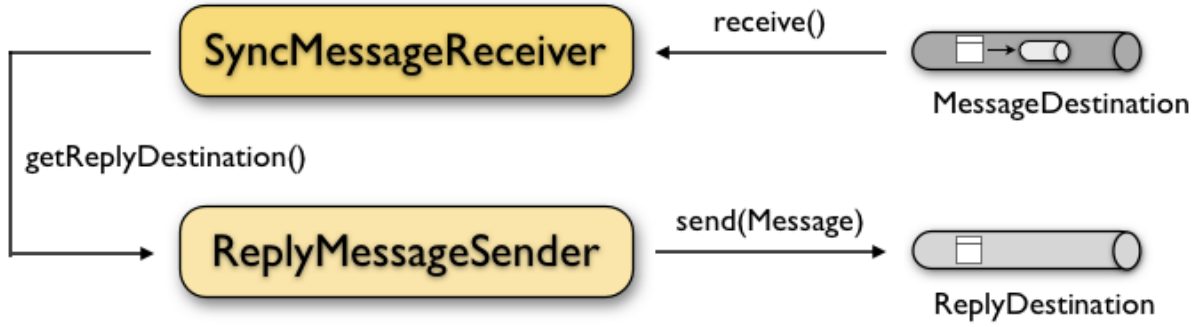
Instead of using the *reply-destination-name* feel free to use the destination reference *reply-destination*

Important

Be aware of permissions that are mandatory for creating temporary destinations. Citrus tries to create temporary queues on the JMS message broker. Following from that the Citrus JMS user has to have the permission to do so. Be sure that the user has the sufficient rights when using temporary reply destinations.

11.4. JMS synchronous message receiver

What is missing is the situation that Citrus receives a JMS message where a temporary reply destination is set. When dealing with synchronous JMS communication the requestor will store a dynamic JMS queue destination into the JMS header in order to receive the synchronous answer on this dynamic destination. So Citrus has to send the reply to the temporary destination, which is dynamic of course. You can handle this with the synchronous message receiver in combination with a reply sender.




```
<citrus:jms-sync-message-receiver id="getOrderRequestReceiver"
                    destination-name="Citrus.JMS.Order.Queue.In"/>

<citrus:jms-reply-message-sender id="getOrderReplySender"
                    reply-destination-holder="getOrderRequestReceiver"/>
```

In first sight the synchronous message receiver has no difference to a normal receiver, but the difference comes in combination with a synchronous reply sender. The reply sender need to know the dynamic reply destination, so it desires a reference to a reply-destination-holder, which is our jms-sync-message-receiver.

11.5. JMS Topics

Up to now we dealt with JMS queue destinations in this chapter. Citrus is also able to connect to JMS topic destinations. In contrary to JMS queues which uses a point-to-point communication JMS topics use publish-subscribe mechanism in order to spread messages over JMS. A JMS topic sender publishes messages to the topic, while the topic accepts multiple message subscriptions and delivers the message to all subscribers.

The Citrus JMS components (sender/receiver) offer the attribute *'pub-sub-domain'*. Once this attribute is set to *true* Citrus will use JMS topics instead of queue destinations. See the following example where the publish-subscribe attribute is set to true in JMS message sender and receiver components.

```
<citrus:jms-message-sender id="helloTopicRequestSender"
                           destination-name="Citrus.JMS.Topic.Hello.Request"
                           pub-sub-domain="true"/>

<citrus:jms-message-receiver id="helloTopicRequestReceiver"
                           receive-timeout="5000"
                           destination-name="Citrus.JMS.Topic.Hello.Request"
                           pub-sub-domain="true"/>
```

When using JMS topics you will be able to subscribe several test actions to the topic destiantion and receive a message multiple times as all subscribers will receive the message.

Important

It is very important to keep in mind that Citrus does not deal with durable subscribers, yet. This means that message that were sent in advance to the message subscription are not delivered to the message receiver. So racing conditions may cause problems when using JMS topic consumers in Citrus. Be sure to let Citrus subscribe to the topic before messages are sent. Otherwise you may loose some messages that were sent in advance to the subscription.

The *'pub-sub-domain'* attribute is also available for synchronous communication sender and receiver components in Citrus. Just add this attribute in order to switch to JMS topics in a publish-subscribe domain.

Chapter 12. Http Support

Citrus is able to connect with Http services and simulate Http servers. In the next sections you will learn how to invoke services using Http messaging. And you will see how to accept client requests and provide proper Http responses.

Note

Similar to the JMS specific configuration schema, Citrus provides a customized Http configuration schema that is used in Spring configuration files. Simply include the http-config namespace in the configuration XML as follows.

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:citrus="http://www.citrusframework.org/schema/config"
   xmlns:citrus-http="http://www.citrusframework.org/schema/http/config"
   xsi:schemaLocation="
   http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
   http://www.citrusframework.org/schema/config
   http://www.citrusframework.org/schema/config/citrus-config.xsd
   http://www.citrusframework.org/schema/http/config
   http://www.citrusframework.org/schema/http/config/citrus-http-config.xsd">

    [...]

</beans>
```

Now you are ready to use the customized http configuration elements using the citrus-http namespace prefix.

12.1. Http message sender

Citrus can invoke any Http service and wait for the response. After that the response goes through the validation process as usual. Let us see how a message sender for Http works:

```
<citrus-http:message-sender id="httpMessageSender"
                            request-url="http://localhost:8090/test"
                            request-method="POST"
                            reply-handler="httpResponseHandler"/>

<citrus-http:reply-message-handler id="httpResponseHandler"/>
```

As Http communication is always synchronous we need a reply message handler besides the http message-sender. It is not very surprising that the sender also needs the *request-url* and a *request-method*. The sender will build a Http request and send it to the Http endpoint. The Http response is then provided to the reply handler.

12.2. Http server

Sending Http messages was quite easy and straight forward. Receiving Http messages is more complicated, because Citrus has to provide a Http server that is listening on a port for client connections. Once a client connection is accepted the Http server must also provide a proper Http response to the client. Citrus ships with an embedded Http server implementation that listens on a port for client connections.

```
<citrus-http:server id="simpleHttpServer"
                port="8090"
                uri="/test"
                deamon="false"
                message-handler="emptyResponseProducingMessageHandler"
                auto-start="true"/>

<bean id="emptyResponseProducingMessageHandler"
 class="com.consol.citrus.http.handler.EmptyResponseProducingMessageHandler"/>
```

The Http server implementation in the example will automatically startup on application loading and will listen on the URL `'http://localhost:8090/test'` for requests. What also is very important is the message handler definition. Once a client request was accepted the message handler is responsible for generating a proper response to the client.

Citrus provides several message handler implementations. Let's have a look at them in the following sections.

12.2.1. Empty response producing message handler

This is the simplest message handler you can think of. It simply provides an empty success response using the Http response code 202. In the introducing example this message handler was used to provide response messages to the calling client. The handler does not need any configurations or properties as it simply responds with an empty Http response.

```
<bean id="emptyResponseProducingMessageHandler"
  class="com.consol.citrus.http.handler.EmptyResponseProducingMessageHandler"/>
```

12.2.2. Static response producing message handler

The next more complex message handler will always return a static response message

```
<bean
  class="com.consol.citrus.adapter.handler.StaticResponseProducingMessageHandler">
    <property name="messagePayload">
        <value>
        <![CDATA[
            <ns0:Response
                xmlns:ns0="http://www.consol.de/schemas/samples/sample.xsd">
                <ns0:MessageId>123456789</ns0:MessageId>
                <ns0:CorrelationId>CORR123456789</ns0:CorrelationId>
                <ns0:Text>Hello User</ns0:Text>
            </ns0:Response>
        ]]>
        </value>
    </property>
    <property name="messageHeader">
      <map>
        <entry key="{http://www.consol.de/schemas/samples}ns0:Operation"
                  value="sayHelloResponse"/>
        <entry key="{http://www.consol.de/schemas/samples}ns0:Request"
                  value="HelloRequest"/>
      </map>
    </property>
</bean>
```

The handler is configured with a static message payload and static response header values. The

response to the client is therefore always identical.

12.2.3. Xpath dispatching message handler

The idea behind the xpath-dispatching-message-handler is that the incoming requests are dispatched to several message handlers according to an element value inside the message payload. The XPath expression will evaluate and call the respective message handler. The message handler mapping is done by their names inside a message handler Spring configuration context. The separate context is loaded in advance.

```
<bean id="xpathDispatchingHandler"
  class="com.consol.citrus.http.handler.XpathDispatchingMessageHandler">
    <property name="xpathMappingExpression" value="//MessageBody/Operation"/>
    <property name="messageHandlerContext" value="message-handler-context.xml"/>
</bean>
```

The handler receives a XPath mapping expression as well as a Spring ApllicationContext file resource. The message handlers are mapped to the different values via their names. For instance a incoming request with `//MessageBody/Operation = "getOrders"` is handled by the message handler named "getOrders". The available message handlers are configured in the message-handler-context (e.g. EmptyResponseProducingMessageHandler, StaticResponseProducingMessageHandler, ...).

12.2.4. JMS connecting message handler

The most powerful message handler is the JMS connecting message handler. Indeed this handler also provides the most flexibility. This handler will forward incoming request to a JMS destination and waiting for a proper response on a reply destination. A configured JMS message receiver can read this forwarded request internally over JMS and provide a proper response on the reply destination.

```
<bean id="jmsForwardingMessageHandler"
         class="com.consol.citrus.http.handler.JmsConnectingMessageHandler">
    <property name="destinationName" value="JMS.Queue.Requests.In"/>
    <property name="replyDestinationName" value="JMS.Queue.Response.Out"/>
    <property name="connectionFactory">
        <bean class="org.apache.activemq.ActiveMQConnectionFactory">
            <property name="brokerURL" value="tcp://localhost:61616" />
        </bean>
    </property>
    <property name="replyTimeout" value="2000"/>
</bean>
```

Tip

The samples section may help you get in touch with the http configuration and the JMS forwarding stategy (Appendix A, *Citrus Samples*)

Chapter 13. SOAP WebServices

In case you need to connect to a SOAP WebService you can use the built-in WebServices support in Citrus. Similar to the Http support Citrus is able to send and receive SOAP messages during a test.

Note

In order to use the SOAP WebService support you need to include the specific XML configuration schema provided by Citrus. See following XML definition to find out how to include the citrus-ws namespace.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:citrus="http://www.citrusframework.org/schema/config"
       xmlns:citrus-ws="http://www.citrusframework.org/schema/ws/config"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
       http://www.citrusframework.org/schema/config
       http://www.citrusframework.org/schema/config/citrus-config.xsd
       http://www.citrusframework.org/schema/ws/config
       http://www.citrusframework.org/schema/ws/config/citrus-ws-config.xsd">

    [...]

</beans>
```

Now you are ready to use the customized WebService configuration elements - all using the citrus-ws prefix - in your Spring configuration.

13.1. SOAP message sender

Citrus can call any SOAP WebService and validate its response message. Let us see how a message sender for SOAP WebServices looks like in the Spring configuration:

```
<citrus-ws:message-sender id="soapMessageSender"
                          request-url="http://localhost:8090/test"
                          reply-handler="soapResponseHandler"/>

<citrus-ws:reply-message-handler id="soapResponseHandler"/>
```

SOAP WebServices always use synchronous communication, so we need a reply message handler. The message sender uses the *request-url* and and calls the WebService. The sender will automatically build a SOAP request message including a SOAP header and the message payload as SOAP body. As the WebService response arrives it is passed to the given reply handler.

Important

The SOAP WebService message sender uses a SoapMessageFactory implementation in order to create the SOAP messages. Just add a bean to the Citrus Spring application context. Spring offers several reference implementations, choose on of them.

```
<bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>
```

By default Citrus will search for a bean with id *'messageFactory'*. In case you intend to use different identifiers you need to tell the SOAP message sender which message factory to

use:

```
<citrus-ws:message-sender id="soapMessageSender"
            request-url="http://localhost:8090/test"
            reply-handler="soapResponseHandler"
            message-factory="mySepcialMessageFactory"/>
```

13.2. SOAP message receiver

Receiving SOAP messages requires a web server instance listening on a port. Citrus is using an embedded Jetty server instance in combination with the Spring WebService project in order to accept SOAP request calls. See how the Jetty server is configured in the Spring configuration.

```
<citrus-ws:jetty-server id="simpleJettyServer"
            port="8091"
            auto-start="true"
            context-config-location="classpath:citrus-ws-servlet.xml"
            resource-base="src/citrus/resources"/>
```

The Jetty server is able to startup automatically during application startup. In the example above the Server is listening on the port `8091` for SOAP requests. The context-config-location attribute defines a further Spring application context. In this application context the request mapping is configured. See the example below.

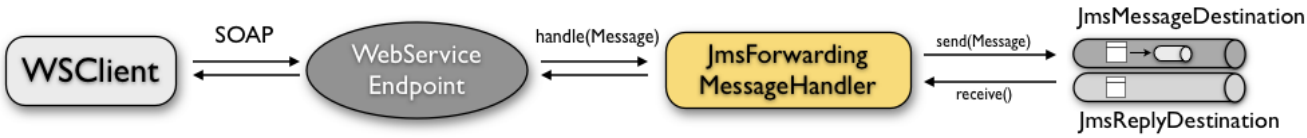
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="loggingInterceptor"
    class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor">
      <description>
          This interceptor logs the message payload.
      </description>
  </bean>

  <bean id="helloServicePayloadMapping"
    class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
      <property name="mappings">
          <props>
              <prop
                  key="{http://www.consol.de/schemas/sayHello}HelloStandaloneRequest">
                  helloServiceEndpoint
              </prop>
          </props>
      </property>
      <property name="interceptors">
          <list>
              <ref bean="loggingInterceptor"/>
          </list>
      </property>
  </bean>

  <bean id="helloServiceEndpoint"
       class="com.consol.citrus.ws.WebServiceEndpoint">
      <property name="messageHandler">
        <bean class="com.consol.citrus.adapter.handler.StaticResponseProducingMessageHandler">
            <property name="messagePayload">
                <value>
                <![CDATA[
                    <ns0:HelloStandaloneResponse
                        xmlns:ns0="http://www.consol.de/schemas/sayHello">
                        <ns0:MessageId>123456789</ns0:MessageId>
                        <ns0:CorrelationId>CORR123456789</ns0:CorrelationId>
                        <ns0:User>WebServer</ns0:User>
                        <ns0:Text>Hello User</ns0:Text>
                    </ns0:HelloStandaloneResponse>
```

```
                ]]>
                </value>
            </property>
            <property name="messageHeader">
                <map>
                    <entry key="{http://www.consol.de/schemas/sayHello}ns0:Operation"
                        value="sayHelloResponse"/>
                    <entry key="{http://www.consol.de/schemas/sayHello}ns0:Request"
                        value="HelloRequest"/>
                    <entry key="citrus_soap_action"
                        value="sayHello"/>
                </map>
            </property>
        </bean>
      </property>
  </bean>
</beans>
```

The program listing above describes a normal request mapping. The mapping is responsible to forward incoming requests to an endpoint which will handle the request and provide a response. First of all Spring's logging interceptor is added to the context. Then we use a payload mapping (PayloadRootQNameEndpointMapping) in order to map all incoming `'HelloStandaloneRequest'` SOAP messages to the `'helloServiceEndpoint'`. Endpoints are of essential nature in Citrus SOAP WebServices implementation. They are responsible for processing a request in order to provide a proper response message that is sent back to the calling client. Citrus uses the endpoint in combination with a message handler implementation.

The enpoint works together with a message handler that is responsible for providing a response message for the client. The various message handler implementations in Citrus were already discussed in Chapter 12, *Http Support*.

In this example the `'helloServiceEndpoint'` uses the `'StaticResponseProducingMessageHandler'` which is always returning a static response message. In most cases static responses will not fit the test scenario and you will have to respond more dynamically. Following from that forwarding to a JMS message destination might fit your needs for more powerful response generation out of a test case. The setup looks like this:



Regardless of which message handler setup you are using in your test case the endpoint transforms the response into a proper SOAP message. You can add as many request mappings and endpoints as you want to the server context configuration. So you are able to handle different request types with one single Jetty server instance.

Have a look at the Chapter 12, *Http Support* in order to find out how the other available message handler work.

That's it for connecting with SOAP WebServices! We saw how to send and receive SOAP messages with Jetty and Spring WebServices. Have a look at the samples coming with your Citrus archive in order to learn more about the SOAP message handlers.

13.3. SOAP headers

SOAP defines several header variations that we discuss in this section. First of all we deal with the special *SOAPAction* header. If we want to set this SOAPAction header when sending WebService messages we simply need to use the special header key called `citrus_soap_action` in our test. This is because in general the sending test action in Citrus is generic for all transport types, but the SOAP action header is specific for the SOAP transport. The special header key in combination with a underlying WebService message sender constructs the SOAPAction in the SOAP message as intended.

```
<header>
    <element name="citrus_soap_action" value="sayHello"/>
</header>
```

Secondly a SOAP message is able to contain customized SOAP headers. These are key-value pairs where the key is a qualified name (QName) and the value a normal String value.

```
<header>
    <element name="{http://www.consol.de/sayHello}ns0:Operation" value="sayHello"/>
    <element name="{http://www.consol.de/sayHello}ns0:Request" value="HelloRequest"/>
</header>
```

Last not least a SOAP header can contain whole XML fragment values. The next example shows how to set these XML framgents as SOAP header:

```
<header>
    <data>
      <![CDATA[
          <ns0:User
              xmlns:ns0="http://www.consol.de/schemas/sayHello">
              <ns0:UserId>123456789</ns0:UserId>
              <ns0:Handshake>S123456789</ns0:Handshake>
          </ns0:User>
      ]]>
    </data>
</header>
```

You can also use external file resources to set this SOAP header XML fragment as shown in this last example code:

```
<header>
    <resource file="classpath:request-soap-header.xml"/>
</header>
```

13.4. SOAP faults

SOAP faults describe a failed communication in SOAP WebServices world. Citrus is able to send and receive SOAP fault messages. On server side Citrus can simulate SOAP faults with fault-code, fault-reason-string and fault-detail. On client side Citrus is able to handle and validate SOAP faults in response messages. The next section describes how to deal with SOAP faults in Citrus.

13.4.1. SOAP fault simulation

As Citrus simulates WebService endpoints you also need to think about simulating SOAP faults. In case Citrus receives a SOAP request you can respond with a proper SOAP fault if necessary.

Please keep in mind that we use the citrus-ws extension for sending SOAP faults in our test case, as shown in this very simple example:

```
<ws:send-fault with="webServiceResponseSender">
    <ws:fault>
        <ws:fault-code>{http://www.citrusframework.org/faults}citrus:TEC-1000</ws:fault-code>
        <ws:fault-string>Invalid request</ws:fault-string>
        <ws:fault-detail>
            <![CDATA[
                <ns0:FaultDetail xmlns:ns0="http://www.consol.de/schemas/samples/sayHello.xsd">
                    <ns0:MessageId>${messageId}</ns0:MessageId>
                    <ns0:CorrelationId>${correlationId}</ns0:CorrelationId>
                    <ns0:ErrorCode>TEC-1000</ns0:ErrorCode>
                    <ns0:Text>Invalid request</ns0:Text>
                </ns0:FaultDetail>
            ]]>
        </ws:fault-detail>
    </ws:fault>
    <ws:header>
        <ws:element name="citrus_soap_action" value="sayHello"/>
    </ws:header>
</ws:send-fault>
```

The example generates a simple SOAP fault that is sent back to the calling client. The fault-detail is optional as well as the soap action declared in the special Citrus header `citrus_soap_action`. You can also set the fault-detail via external file resource. Just use `<ws:fault-detail file="classpath:myFaultDetail.xml"/>` instead of the inline CDATA definition.

The generated SOAP fault results in a SOAP message like follows:

```
HTTP/1.1 500 Internal Server Error
Accept: text/xml, text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
SOAPAction: "sayHello"
Content-Type: text/xml; charset=utf-8
Content-Length: 680
Server: Jetty(7.0.0.pre5)

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <SOAP-ENV:Fault>
            <faultcode xmlns:citrus="http://www.citrusframework.org/faults">citrus:TEC-1000</faultcode>
            <faultstring xml:lang="en">Invalid request</faultstring>
            <detail>
                <ns0:FaultDetail xmlns:ns0="http://www.consol.de/schemas/samples/sayHello.xsd">
                    <ns0:MessageId>9277832563</ns0:MessageId>
                    <ns0:CorrelationId>4346806225</ns0:CorrelationId>
                    <ns0:ErrorCode>TEC-1000</ns0:ErrorCode>
                    <ns0:Text>Invalid request</ns0:Text>
                </ns0:FaultDetail>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Important

Notice that the send action uses a special XML namespace (ws:send). This ws namespace belongs to the Citrus WebService extension and adds SOAP specific features to the normal send action. When you use such ws extensions you need to define the additional namespace in your test case. This is usually done in the root *<spring:beans>* element where we simply declare the citrus-ws specific namespace like follows.

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:ws="http://www.citrusframework.org/schema/ws/testcase"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.citrusframework.org/schema/testcase
    http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd
```

```
    http://www.citrusframework.org/schema/ws/testcase
    http://www.citrusframework.org/schema/ws/testcase/citrus-ws-testcase.xsd">
```

13.4.2. SOAP fault validation

In case you receive SOAP messages from a WebService endpoint you may also want to validate special SOAP faults in error situations. Citrus can validate SOAP faults with SOAP fault code and fault string values.

By default the sending action in Citrus may throw a specific exception when the SOAP response contains a SOAP fault element (SoapFaultClientException). A tester can assert this kind of exception in a test case in order to expect the SOAP error.

```
<assert class="org.springframework.ws.soap.client.SoapFaultClientException">
    <send with="webServiceHelloRequestSender">
        <message>
            <data>
                <![CDATA[
                    <ns0:SoapFaultForcingRequest
                    xmlns:ns0="http://www.consol.de/schemas/soap">
                        <ns0:Message>This is invalid</ns0:Message>
                    </ns0:SoapFaultForcingRequest>
                ]]>
            </data>
        </message>
    </send>
</assert>
```

The SOAP message sending action is surrounded by a simple assert action. The asserted exception class is the SoapFaultClientException. This means that the test expects the exception to be thrown during the communication. Otherwise the test is failing.

This exception assertion can not offer direct SOAP fault code and fault string validation, because we do not have access to the SOAP fault elements. But we can use a special assert implementation for this.

```
<ws:assert fault-code="{http://www.citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
    <send with="webServiceHelloRequestSender">
        <message>
            <data>
                <![CDATA[
                    <ns0:SoapFaultForcingRequest
                    xmlns:ns0="http://www.consol.de/schemas/soap">
                        <ns0:Message>This is invalid</ns0:Message>
                    </ns0:SoapFaultForcingRequest>
                ]]>
            </data>
        </message>
    </send>
</ws:assert>
```

The special assert action offers two attributes *"fault-code"* and *"fault-string"*, where *fault-code* is defined as a QName string and is mandatory for the validation. The fault assertion also supports test variable replacement as usual (e.g. fault-code="{http://www.citrusframework.org/faults}${myFaultCode}").

The time you use SOAP fault validation you need to tell Citrus how to validate the SOAP faults. Citrus needs an instance of SoapFaultValitator that we need to place into the *'citrus-context.xml'* Spring application context. By default Citrus is searching for a bean with the id *'soapFaultValidator'*.

```
<bean id="soapFaultValidator" class="com.consol.citrus.ws.validation.SimpleSoapAttachmentValidator"/>
```

Citrus offers reference implementations for SOAP fault validation such as *com.consol.citrus.ws.validation.SimpleSoapAttachmentValidator*. Please see the API documentation for other available reference implementations. Of course you can define your own validator bean or several validator beans. In the test case you can explicitly choose the validator to use:

```
<ws:assert fault-code="{http://www.citrusframework.org/faults}TEC-1001"
              fault-string="Invalid request"
              fault-validator="mySpecialSoapFaultValidator">
   [...]
</ws:assert>
```

Important

Another important thing to notice when asserting SOAP faults is the fact, that Citrus needs to have a SoapMessageFactory available in the Spring application context. If you deal with SOAP messaging in general you will already have such a bean in the context.

```
<bean id="messageFactory" class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>
```

Choose one of Spring's reference implementations or some other implementation as SOAP message factory. Citrus will search for a bean with id *'messageFactory'* by default. In case you have other beans with different identifiers please choose the messageFactory in the test case assert action:

```
<ws:assert fault-code="{http://www.citrusframework.org/faults}TEC-1001"
              fault-string="Invalid request"
              message-factory="mySpecialMessageFactory">
   [...]
</ws:assert>
```

Important

Notice the ws specific namespace that belongs to the Citrus WebService extensions. As the ws:assert action uses SOAP specific features we need to refer to the citrus-ws namespace. You can find the namespace declaration in the root element in your test case.

```
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:ws="http://www.citrusframework.org/schema/ws/testcase"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.citrusframework.org/schema/testcase
    http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd
    http://www.citrusframework.org/schema/ws/testcase
    http://www.citrusframework.org/schema/ws/testcase/citrus-ws-testcase.xsd">
```

Citrus is also able to validate SOAP fault details. See the following example for understanding how to do it:

```
<ws:assert fault-code="{http://www.citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
    <ws:fault-detail>
      <![CDATA[
          <ns0:FaultDetail xmlns:ns0="http://www.consol.de/schemas/soap">
              <ns0:ErrorCode>TEC-1000</ns0:ErrorCode>
              <ns0:Text>Invalid request</ns0:Text>
```

```
            </ns0:FaultDetail>
        ]]>
    </ws:fault-detail>
    <send with="webServiceHelloRequestSender">
        <message>
            <data>
                <![CDATA[
                    <ns0:SoapFaultForcingRequest
                    xmlns:ns0="http://www.consol.de/schemas/soap">
                        <ns0:Message>This is invalid</ns0:Message>
                    </ns0:SoapFaultForcingRequest>
                ]]>
            </data>
        </message>
    </send>
</ws:assert>
```

The expected SOAP fault detail content is simply added to the ws:assert action. The SoapFaultValidator implementation is responsible for validation the SOAP fault detail offering a validation algorithm. It depends on the chosen validator implementation how the detail is compared to the expected template. Possible algorithms are pure String comparison and XML tree comparison as already known from the normal message payload validation. Please see the Citrus API documentation for available validator implementations and validation algorithms.

13.5. SOAP attachment support

Citrus is able to add attachments to a SOAP request. In return you can also receive SOAP messages with attachments and validate their content. The next chapters describe how to handle SOAP attachments in Citrus.

13.5.1. Send SOAP attachments

As client Citrus is able to add attachments to SOAP messages. I think it is best to look at an example in order to understand how it works.

```
<ws:send with="webServiceRequestSender">
    <message>
        <data>
            <![CDATA[
                <ns0:SoapMessageWithAttachment xmlns:ns0="http://www.consol.de/schemas/samples/sample.xsd">
                    <ns0:Operation>Read the attachment</ns0:Operation>
                </ns0:SoapMessageWithAttachment>
            ]]>
        </data>
    </message>
    <ws:attachment content-id="MySoapAttachment" content-type="text/plain">
        <ws:resource file="classpath:com/consol/citrus/ws/soapAttachment.txt"/>
    </ws:attachment>
</ws:send>
```

Note

In the previous chapters you may have already noticed the citrus-ws namespace that stands for the WebService extensions in Citrus. Please include the citrus-ws namespace in your testcase as described earlier in this chapter, in order to use the attachment support.

We need to use the Citrus ws extension namespace in our test case wich offers a special send action that is aware of SOAP attachments. The attachment content usually consists of a content-id a

content-type and the actual content as plain text or binary content. Inside the test case you can use external file resources or inline CDATA to specify the attachment content. As you are familiar with Citrus you may know this already from other actions.

Citrus will construct a SOAP message with the SOAP attachment. Currently only one attachment per message is supported, which will fullfill the needs of almost every application.

13.5.2. Receive and validate SOAP attachments

When Citrus calls SOAP WebServices as a client we may receive SOAP responses with attachments. The tester can validate received SOAP messages with attachment content quite easy. As usual let us have a look at an example first.

```
<ws:receive with="webServiceRequestReceiver">
    <message>
        <data>
            <![CDATA[
                <ns0:SoapMessageWithAttachmentRequest xmlns:ns0="http://www.consol.de/schemas/samples/sample.xsd">
                    <ns0:Operation>Read the attachment</ns0:Operation>
                </ns0:SoapMessageWithAttachmentRequest>
            ]]>
        </data>
    </message>
    <ws:attachment content-id="MySoapAttachment"
                   content-type="text/plain"
                   validator="mySoapAttachmentValidator">
        <ws:resource file="classpath:com/consol/citrus/ws/soapAttachment.txt"/>
    </ws:attachment>
</ws:receive>
```

Again we use the Citrus ws extension namespace for a specific receive action that is aware of SOAP attachment validation. The tester can validate the content-id, the content-type and the attachment content. Instead of using the external file resource you could also define an expected attachment template directly in the test case as CDATA inline element.

Note

The ws:attachment element specifies a validator instance. This validator determines how to validate the attachment content. SOAP attachments are not limited to XML content. Plain text content and binary content is possible, too. So each SOAP attachment validating action can use a different SoapAttachmentValidator instance which is responsible for validating and comparing received attachments to expected template attachments. In the Citrus configuration the validator is set as normal Spring bean with the respective identifier.

```
<bean id="mySoapAttachmentValidator" class="com.consol.citrus.ws.validation.SimpleSoapAttachmentValidator"/>
<bean id="soapAttachmentValidator" class="com.consol.citrus.ws.validation.SimpleSoapAttachmentValidator"/>
```

You can define several validator instanes in the Citrus configuration. The validator with the general id="soapAttachmentValidator" is the default validator for all actions that do not explicitly set a validator instance. Citrus offers a set of validator implementations. The SimpleSoapAttachmentValidator will use a simple plain text comparison. Of course you are able to add individual validator implementations, too.

As usual a special endpoint implementation receives the SOAP requests and delegates those requests to a MessageHandler implementation as described in chapter Section 13.2, “SOAP

message receiver". The SOAP attachment validation in its curent nature does require the JmsConnectingMessageHandler implementation where the Citrus endpoint will forward incoming requests to a JMS queue. The SOAP attachment is converted to special JMS message headers and are ready for validation when received by the test case. See the following example to clear the boundaries.

```
<bean id="webServiceEndpoint" class="com.consol.citrus.ws.WebServiceEndpoint">
    <property name="messageHandler">
        <bean class="com.consol.citrus.adapter.handler.JmsConnectingMessageHandler">
            <property name="destinationName" value="JMS.Soap.RequestQueue"/>
            <property name="connectionFactory">
                <bean class="org.apache.activemq.ActiveMQConnectionFactory">
                    <property name="brokerURL" value="tcp://localhost:61616" />
                </bean>
            </property>
            <property name="messageCallback">
                <bean class="com.consol.citrus.ws.message.SoapAttachmentAwareJmsMessageCallback"/>
            </property>
            <property name="replyTimeout" value="5000"/>
        </bean>
    </property>
</bean>
```

The endpoint in the example uses the JmsConnectingMessageHandler in combination with the SoapAttachmentAwareJmsMessageCallback, which takes care of SOAP attachments in incoming requests. This mechanism allows test cases to receive messages over JMS with SOAP attachments encoded in the JMS message header. Fortunately you do not need to worry about the JMS header encoding done in the SoapAttachmentAwareJmsMessageCallback, because the ws:attachment extension will do all magic for you. Just use a extended message receiving action as shown in the example at the beginning of this chapter and you are able to validate the SOAP attachment data.

Chapter 14. Message channel support

Spring Integration (http://www.springsource.org/spring-integration) provides support for messaging solutions in Spring-based applications meeting the famous Enterprise Integration patterns and best practices. Citrus itself uses a lot of Spring APIs, especially those from Spring Integration.

The conclusion is that Citrus supports the sending and receiving of messages to/from Spring Integration message channel components.

Note

Citrus message channel connecting components use the same "citrus" configuration namespace and schema definitions in Spring context files as already described in chapter Chapter 11, *Connecting with JMS*. You always have to include this configuration namespace in order to use the Citrus configuration elements.

14.1. Message channel sender

You can access message channels directly with a *message-channel-sender* component. The message channel sender configuration is quite simple and receives a target channel as reference:

```
<citrus:message-channel-sender id="orderRequestSender" channel="orderChannel"/>

<si:channel id="orderChannel"/>
```

Note

The Spring Integration configuration components use a specific namespace that has to be included into your Spring application context. You can use the following template which holds all necessary namespaces and schema locations:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:citrus="http://www.citrusframework.org/schema/config"
    xmlns:si="http://www.springframework.org/schema/integration"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
            http://www.citrusframework.org/schema/config
            http://www.citrusframework.org/schema/config/citrus-config.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/spring-integration-1.0.xsd">
</beans>
```

The Citrus message-channel-sender also supports a customized message channel template that will actually send the messages. The customized template might give you access to special configuration possibilities. However it is optional, so if no message channel template is defined in the configuration Citrus will create a default template.

```
<citrus:message-channel-sender id="statusRequestSender"
                           channel="orderChannel"
                           message-channel-template="myMessageChannelTemplate"/>
```

The message sender is now ready to publish messages to the defined channel. The communication

is supposed to be asynchronous, so the sender is not able to process a reply message. We will deal with synchronous communication and reply messages later in this chapter. The message sender just publishes messages to the channel.

14.2. Message channel receiver

Citrus is able to receive messages from Spring Integration message channel destinations. Again the message-channel-receiver needs nothing but a reference to a message channel in its simplest configuration.

```
<citrus:message-channel-receiver id="ordersResponseReceiver"
                                 channel="orderChannel"
                                 receive-timeout="5000"/>
```

As usual the receiver connects to the message destination and waits for messages to arrive. The user can set a receive timeout which is set to 5000 milliseconds by default. In case no message was received in this time frame the receiver raises timeout errors and the test fails.

Similar to the previously described *message-channel-sender* the *message-channel-receiver* supports a message-channel-template that is used for receiving messages.

```
<citrus:message-channel-receiver id="ordersResponseReceiver"
                                 channel="orderChannel"
                                 receive-timeout="5000"
                                 message-channel-template="myMessageChannelTemplate"/>
```

14.3. Synchronous message channel sender

The synchronous message sender publishes messages and waits synchronously for the response to arrive on some reply channel destination. The reply channel name is set in the message's header attributes so the counterpart in this communication can send its reply to that channel. The basic configuration for a synchronous send-receive message channel sender looks like follows:

```
<citrus:sync-message-channel-sender id="customerRequestSender"
                            channel="customerRequestChannel"
                            reply-handler="customerReplyMessageHandler"
                            reply-timeout="1000"/>

<citrus:message-channel-reply-handler id="customerReplyMessageHandler"/>
```

Synchronous message channel senders usually go with a reply message handler that handles the reply messages. Once the synchronous reply message was consumed from the reply channel the reply-message-handler implementation is responsible for further processing (e.g. message validation).

Note

Reply channels are always of dynamic temporary nature. The temporary reply channels are only used once for a single communication handshake. After that the reply channel is deleted again. Static reply channels are not supported because they would receive multiple reply messages at the same time and the reply message handlers need to filter (select) the message from that channel, which is not in scope yet.

14.4. Synchronous message channel receiver

In the last section we saw that synchronous communication is based on reply messages on temporary reply channels. We saw that Citrus is able to publish messages to channels and wait for reply messages to arrive on temporary reply channels. This section deals with the same synchronous communication over reply messages, but now Citrus has to send dynamic reply messages to temporary channels.

The scenario we are talking about is that Cittrus receives a message and we need to reply to a temporary reply channel that is stored in the message header attributes.

We handle this synchronous communication with the synchronous message receiver in combination with a reply sender. The configuration looks like follows:

```
<citrus:sync-message-channel-receiver id="orderRequestReceiver" channel="order"/>

<citrus:message-channel-reply-sender id="orderReplySender"
                    reply-channel-holder="orderRequestReceiver"/>
```

The synchronous message channel receiver will store dynamic reply channel destinations and provide those dynamic channel names to a reply message sender. Those two components *sync-message-channel-receiver* and *message-channel-reply-sender* are always working together in order to realize incoming synchronous request with reply messages sent by Citrus.

14.5. Connecting with Spring Integration Adapters

We have seen asynchronous and synchronous communication with Spring Integration message channels in this chapter. With this message channel connection Citrus is able to use the various Spring Integration Adapter implementations, which is a fantastic way to extend Citrus for additional transports. The following chapter tries to give an example how to use Spring Integration with Citrus.

We want to use the Spring Integration file adapter that is able to read/write files from/to a storage directory. Citrus can easily connect to this file adapter implementation with its message channel support. Citrus message sender and receiver speak to message channels that are connected to the Spring Integration file adapter.

```
<citrus:message-channel-sender id="fileSender" channel="fileOutboundChannel"/>

<file:outbound-channel-adapter id="filesOut"
                  channel="fileOutboundChannel"
                  directory="file:${some.directory.property}"/>

<si:channel id="fileOutboundChannel"/>
```

The configuration above describes a Citrus message channel sender in combination with a Spring Integration outbound file adapter that writes messages to a storage directory. With this combination you are able to write files to a directory in your Citrus test case.

Note

The Spring Integration file adapter configuration components add a new namespace to our Spring application context. See this template which holds all necessary namespaces and schema locations:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:citrus="http://www.citrusframework.org/schema/config"
    xmlns:si="http://www.springframework.org/schema/integration"
    xmlns:file="http://www.springframework.org/schema/integration/file"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
            http://www.citrusframework.org/schema/config
            http://www.citrusframework.org/schema/config/citrus-config.xsd
            http://www.springframework.org/schema/integration
            http://www.springframework.org/schema/integration/spring-integration-1.0.xsd
            http://www.springframework.org/schema/integration/file
            http://www.springframework.org/schema/integration/file/spring-integration-file-1.0.xsd">
</beans>
```

The next program listing shows a possible inbound file communication. So Spring's file inbound adapter will read files from a storage directory and publish the file contents to a message channel. Citrus can then receive those files as messages in a test case and validate the file contents for instance.

```
<file:inbound-channel-adapter  id="fileIn"
                channel="fileInputChannel"
                directory="file:${some.directory.property}">
    <si:poller>
        <si:interval-trigger time-unit="MILLISECONDS" interval="100" />
    </si:poller>
</file:inbound-channel-adapter>

<si:channel id="fileInputChannel">
    <si:queue capacity="25"/>
    <si:interceptors>
        <bean class="org.springframework.integration.transformer.MessageTransformingChannelInterceptor">
            <constructor-arg>
                <bean class="org.springframework.integration.file.transformer.FileToStringTransformer"/>
            </constructor-arg>
        </bean>
    </si:interceptors>
</si:channel>

<citrus:message-channel-receiver id="fileReceiver" channel="fileInputChannel"/>
```

Important

The file inbound adapter constructs Java file objects as the message payload by default. Citrus can only work on String message payloads. So we need a file transformer that converts the file objects to String payloads representing the file's content.

This file adapter example shows how easy Citrus can work hand in hand with Spring Integration adapter implementations. The message channel support is a fantastic way to extend the transport and protocol support in Citrus by connecting with the very good Spring Integration adapter implementations. Have a closer look at the Spring Integration project for more details and other adapter implementations that you can use with Citrus integration testing.

Chapter 15. Functions

The test framework will offer several functions that are useful throughout the test execution. The functions will always return a string value that is ready for use as variable value or directly inside a text message.

A set of functions is usually combined to a function library. The library has a prefix that will identify the functions inside the test case. The default test framework function library uses a default prefix (citrus). You can write your own function library using your own prefix in order to extend the test framework functionality whenever you want.

The library is built in the Spring configuration and contains a set of functions that are of public use.

```
<bean id="testFrameworkFunctionLibrary"
         class="com.consol.citrus.functions.FunctionLibrary">
    <property name="name" value="testFrameworkFunctionLibrary"/>
    <property name="prefix" value="citrus:"/>
    <property name="members">
        <map>
          <entry key="randomNumber">
              <bean class="com.consol.citrus.functions.RandomNumberFunction"/>
          </entry>
          <entry key="randomString">
              <bean class="com.consol.citrus.functions.RandomStringFunction"/>
          </entry>
          ...
        </map>
    </property>
</bean>
```

In the next chapters the default functions offered by the framework will be described in detail.

15.1. citrus:concat()

The function will combine several string tokens to a single string value. This means that you can combine a static text value with a variable value for instance. A first example should clarify the usage:

```
<testcase name="concatFunctionTest">
    <variables>
        <variable name="date" value="citrus:currentDate(yyyy-MM-dd)" />
        <variable name="text" value="Hello Test Framework!" />
    </variables>
    <actions>
        <echo>
            <message>
                citrus:concat('Today is: ', ${date}, ' right!?')
            </message>
        </echo>
        <echo>
            <message>
                citrus:concat('Text is: ', ${text})
            </message>
        </echo>
    </actions>
</testcase>
```

Please do not forget to mark static text with single quote signs. There is no limitation for string tokens to be combined.

```
citrus:concat('Text1', 'Text2', 'Text3', ${text}, 'Text5', … , 'TextN')
```

The function can be used wherever variables can be used. For instance when validating XML

elements in the receive action.

```
<message>
    <validate path="//element/element" value="citrus:concat('Cx1x', ${generatedId})"/>
</message>
```

15.2. citrus:substring()

The function will have three parameters.

1. String to work on

2. Starting index

3. End index (optional)

Let us have a look at a simple example for this function:

```
<echo>
    <message>
        citrus:substring('Hello Test Framework', 6)
    </message>
</echo>
<echo>
    <message>
        citrus:substring('Hello Test Framework', 0, 5)
    </message>
</echo>
```

Function output:

Test Framework

Hello

15.3. citrus:stringLength()

The function will calculate the number of characters in a string representation and return the number.

```
<echo>
    <message>citrus:stringLength('Hello Test Framework')</message>
</echo>
```

Function output:

20

15.4. citrus:translate()

This function will replace regular expression matching values inside a string representation with a specified replacement string.

```
<echo>
    <message>
        citrus:translate('H.llo Test Fr.mework', '\.', 'a')
    </message>
```

```
</echo>
```

Note that the second parameter will be a regular expression. The third parameter will be a simple replacement string value.

Function output:

Hello Test Framework

15.5. citrus:substringBefore()

The function will search for the first occurrence of a specified string and will return the substring before that occurrence. Let us have a closer look in a simple example:

```
<echo>
    <message>
        citrus:substringBefore('Test/Framework', '/')
    </message>
</echo>
```

In the specific example the function will search for the ‘/’ character and return the string before that index.

Function output:

Test

15.6. citrus:substringAfter()

The function will search for the first occurrence of a specified string and will return the substring after that occurrence. Let us clarify this with a simple example:

```
<echo>
    <message>
        citrus:substringAfter('Test/Framework', '/')
    </message>
</echo>
```

Similar to the substringBefore function the ‘/’ character is found in the string. But now the remaining string is returned by the function meaning the substring after this character index.

Function output:

Framework

15.7. citrus:round()

This is a simple mathematic function that will round decimal numbers representations to their nearest non decimal number.

```
<echo>
    <message>citrus:round('3.14')</message>
</echo>
```

Function output:

3

15.8. citrus:floor()

This function will round down decimal number values.

```
<echo>
    <message>citrus:floor('3.14')</message>
</echo>
```

Function output:

3.0

15.9. citrus:ceiling()

Similar to floor function, but now the function will round up the decimal number values.

```
<echo>
    <message>citrus:ceiling('3.14')</message>
</echo>
```

Function output:

4.0

15.10. citrus:randomNumber()

The random number function will provide you the opportunity to generate random number strings containing positive number letters. There is a singular Boolean parameter for that function describing whether the generated String should have leading zero letters or not. Default value for this padding flag will be true.

Next example will show the function usage:

```
<variables>
    <variable name="rndNumber1" value="citrus:randomNumber(10)"/>
    <variable name="rndNumber2" value="citrus:randomNumber(10, true)"/>
    <variable name="rndNumber3" value="citrus:randomNumber(10, false)"/>
</variables>
```

Function output:

8954638765

0006387650

45638765

15.11. citrus:randomString()

This function will generate a random string representation with a defined length. A second parameter for this function will define the case of the generated letters (UPPERCASE, LOWERCASE, MIXED)

```
<variables>
    <variable name="rndString0" value="${citrus:randomString(10)}"/>
    <variable name="rndString1" value="citrus:randomString(10)"/>
    <variable name="rndString2" value="citrus:randomString(10, UPPERCASE)"/>
    <variable name="rndString3" value="citrus:randomString(10, LOWERCASE)"/>
    <variable name="rndString4" value="citrus:randomString(10, MIXED)"/>
</variables>
```

Function output:

Hr546dfA65

Ag3876det5

4SDF87TR

4tkhi7uz

Vt567JkA32

15.12. citrus:randomEnumValue()

This function returns one of its supplied arguments. Furthermore you can specify a custom function with a preconfigured list of values (the enumeration). The funciton will randomly return an entry when called without arguments. This promotes code reuse and facilitates refactoring.

In the next sample the function is used to set a httpStatusCode variable to one of the given HTTP status codes (200, 401, 500)

```
<variable name="httpStatusCode" value="citrus:randomEnumValue('200', '401', '500')" />
```

As mentioned before you can define a custom function for your very specific needs in order to easily manage a list of predefined values like this:

```
<bean id="myCustomFunctionLibrary" class="com.consol.citrus.functions.FunctionLibrary">
<property name="name" value="myCustomFunctionLibrary" />
<property name="prefix" value="custom:" />
<property name="members">
  <map>
    <entry key="randomHttpStatusCode">
      <bean class="com.consol.citrus.functions.core.RandomEnumValueFunction">
        <property name="values">
          <list>
            <value>200</value>
            <value>500</value>
            <value>401</value>
          </list>
        </property>
      </bean>
    </entry>
  </map>
</property>
</bean>
```

We have added a custom function library with a custom function definition. The custom function

"randomHttpStatusCode" randomly chooses an HTTP status code each time it is called. Inside the test you can use the function like this:

```
<variable name="httpStatusCode" value="custom:randomHttpStatusCode()" />
```

15.13. citrus:currentDate()

This function will definitely help you when accessing the current date. Some examples will show the usage in detail:

```
<echo><message>citrus:currentDate()</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd'T'hh:mm:ss')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1y')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1M')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1d')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1h')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1m')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '+1s')</message></echo>
<echo><message>citrus:currentDate('yyyy-MM-dd HH:mm:ss', '-1y')</message></echo>
```

Note that the currentDate function provides two parameters. First parameter describes the date format string. The second will define a date offset string containing year, month, days, hours, minutes or seconds that will be added or subtracted to or from the actual date value.

Function output:

01.09.2009

2009-09-01

2009-09-01 12:00:00

2009-09-01T12:00:00

15.14. citrus:upperCase()

This function converts any string to upper case letters.

```
<echo>
    <message>citrus:upperCase('Hello Test Framework')</message>
</echo>
```

Function output:

HELLO TEST FRAMEWORK

15.15. citrus:lowerCase()

This function converts any string to lower case letters.

```
<echo>
    <message>citrus:lowerCase('Hello Test Framework')</message>
</echo>
```

Function output:

hello test framework

15.16. citrus:average()

The function will sum up all specified number values and divide the result through the number of values.

```
<variable name="avg" value="citrus:average('3', '4', '5')"/>
```

avg = *4.0*

15.17. citrus:minimum()

This function returns the minimum value in a set of number values.

```
<variable name="min" value="citrus:minimum('3', '4', '5')"/>
```

min = *3.0*

15.18. citrus:maximum()

This function returns the maximum value in a set of number values.

```
<variable name="max" value="citrus:maximum('3', '4', '5')"/>
```

max = *5.0*

15.19. citrus:sum()

The function will sum up all number values. The number values can also be negative.

```
<variable name="sum" value="citrus:sum('3', '4', '5')"/>
```

sum = *12.0*

15.20. citrus:absolute()

The function will return the absolute number value.

```
<variable name="abs" value="citrus:absolute('-3')"/>
```

abs = *3.0*

15.21. citrus:mapValue()

This function implementation maps string keys to string values. This is very helpful when the used key is randomly chosen at runtime and the corresponding value ist not defined during the design time.

The following function library defines a custom function for mapping HTTP status codes to the corresponding messages:

```
<bean id="myCustomFunctionLibrary" class="com.consol.citrus.functions.FunctionLibrary">
  <property name="name" value="myCustomFunctionLibrary" />
  <property name="prefix" value="custom:" />
  <property name="members">
    <map>
      <entry key="getHttpStatusMessage">
        <bean class="com.consol.citrus.functions.core.MapValueFunction">
          <property name="values">
            <map>
              <entry key="200" value="OK" />
              <entry key="401" value="Unauthorized" />
              <entry key="500" value="Internal Server Error" />
            </map>
          </property>
        </bean>
      </entry>
    </map>
  </property>
</bean>
```

In this example the function sets the variable httpStatusMessage to the 'Internal Server Error' string dynamically at runtime. The test only knows the HTTP status code and does not care about spelling and message locales.

```
<variable name="httpStatusCodeMessage" value="custom:getHttpStatusMessage('500')" />
```

Chapter 16. Testsuite

Tests often need to be grouped to test suites. Test suites also provide the functionality to do some work before and after the tests are run. Database preparing and cleaning tasks or server starting and stopping fit well into these initialization and cleanup phases of a test suite. In Citrus a test suite is typically defined as Spring bean inside a XML configuration file.

16.1. Tasks before, between and after the test run

A tester can influence the behavior of a test suite before, after and during the test run. See the next code example to find out how it works:

```
<bean name="integrationTests"
         class="com.consol.citrus.TestSuite">
    <property name="tasksBefore">
        <list>

        </list>
    </property>

    <property name="tasksBetween">
        <list>

        </list>
    </property>

    <property name="tasksAfter">
        <list>

        </list>
    </property>

</bean>
```

The test suite bean is of the type com.consol.citrus.TestSuite and offers the following properties to affect the basic behavior:

- tasksBefore: List of actions that will be executed before the first test is run
- tasksBetween: List of actions that will be executed in between every test
- tasksAfter: List of actions that will be executed after the last test has ended

The three task-sections before, in between and after the execution of tests are supposed to be used for initializing and finishing tasks. All these tasks can easily be adjusted by adding or removing beans inside the <list> element.

Tip

The framework offers special startup and shutdown actions that may start and stop server implementations. This might be helpful when dealing with Http servers or WebService containers like Jetty. You can also think of starting/stopping a JMS broker.

A test suite run may require the test environment to be clean. Therefore it is a good idea to purge all JMS destinations or clean up the database to avoid errors caused by obsolete data from previous test runs.

Between the tests it also might sound reasonable to purge all JMS queues. In case a test

fails the use case processing stops and some messages might be left in some JMS queues. The next test then will be confronted with these invalid messages. Purging all JMS destinations between the tests is therefore a good idea.

16.2. Multiple testsuites

Sometimes it is useful to configure more than one of these test suite instances. You can think of test suites for unit testing, integration testing, performance testing and so on. Unit testing might require different actions before the test run than integration testing and vice versa.

You can simply define several beans of test suites in the Spring configuration. There could be one test suite instance for unit testing and one for integration testing, with their individual configuration like separate 'tasksBefore' and different including patterns for tests.

By default Citrus will start all available test suite instances in sequence. If you want to start a certain instance explicitly you have to declare the name of the instance when starting Citrus.

Chapter 17. TIBCO support

The Citrus framework was designed to also connect to several TIBCO software components. The outcome is a group of special test actions that enable the tester to connect to the TIBCO components and execute operations towards TIBCO.

17.1. Connecting with TIBCO Hawk

TIBCO uses the enterprise monitoring tool Hawk to manage and supervise all TIBCO applications. Citrus can access Hawk to affect the TIBCO service instances. You can start or stop BusinessWorks process archives and process starters for instance.

Tip

Use TIBCO Hawk in order to prepare the TIBCO environment according to the test that will be executed (unit or integration tests).

The next example shows how the Hawk agent can be used in the test suite setup.

```
<bean parent="tibcoHawkAgent">
    <property name="methodName"
                value="ResumeProcessStarter"/>
    <property name="methodParameters">
        <list>
            <bean class="COM.TIBCO.hawk.talon.DataElement">
                <constructor-arg value="ProcessDefinition"/>
                <constructor-arg value="Test/UnitTest/BusinessServiceTester.process"/>
            </bean>
        </list>
    </property>
</bean>
```

The example shows a method call to a Hawk agent. The method is declared through the methodName property. In the above example the Hawk agent will resume a particular process starter. The methodParameters property will cause the process starter to resume.

In general the HawkAgent has the following properties:

- microAgent -> `COM.TIBCO.ADAPTER.bwengine.<hawk.domain>.<hawk.service.instance>.<hawk.process.archive>`
- hawkDomain
- rvService (e.g. 57500)
- rvNetwork
- rvDaemon (e.g. tcp:57500)

The microAgentID consists of the Hawk domain, the service instance and the process archive. With the right setting of the domain the service and the daemon you can now use all methods offered by the specific MicroAgent.

17.2. Connecting with TIBCO Collaborator Workflow Server

Similar to the Hawk agent it is useful to start and stop certain workflow jobs in the TIBCO Collaborator Workflow Server, especially for clean up reasons. Left over workflows from previous test runs will then get terminated correctly.

The test framework offers a special action implementation, in order to clean up all available workflow jobs on the Workflow Collaborator Server.

The bean looks like follows:

```
<bean id="cleanIcJobs"
    class="com.consol.citrus.actions.CleanIcJobsBean">
    <property name="userName" value="${ic.username}"/>
    <property name="password" value="${ic.password}"/>
    <property name="serverName" value="${ic.servername}"/>
    <property name="service" value="${ic.tibrv.service}"/>
    <property name="network" value="${ic.tibrv.network}"/>
    <property name="daemon" value="${ic.tibrv.daemon}"/>
    <property name="queueCount" value="${ic.tibrv.queue.count}"/>
    <property name="serverDiscoveryTimeout" value="${ic.tibrv.server.discovery.timeout}"/>
</bean>
```

The properties of that bean do all refer to the connection credentials of the Workflow InConcert Server instance.

Chapter 18. XML schema validation

There are several possibilities to describe the structure of XML documents. The two most popular ways are DTD (Document type definition) and XSD (XML Schema definition). Once a XML document has decided to be classified using a schema definition the structure of the document has to fit the predefined rules inside the schema definition. XML document instances are valid only in case they meet all these structure rules defined in the schema definition. Currently Citrus can validate XML documents using the schema languages DTD and XSD.

18.1. XSD validation

Citrus handles XML schema definitions in order to validate incoming XML documents. Consequential the message receiving actions have to know the respective XML schema (*.xsd) file resources to do so. This is done through a central schema repository which holds all available XML schema files for a project.

```
<bean id="schemaRepository"
        class="com.consol.citrus.xml.XsdSchemaRepository">
    <property name="schemas">
        <list>
            <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
                <property name="xsd"
                     value="classpath:citrus/flightbooking/TravelAgencySchema.xsd"/>
            </bean>
            <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
                <property name="xsd"
                     value="classpath:citrus/flightbooking/AirlineSchema.xsd"/>
            </bean>
        </list>
    </property>
</bean>
```

By convention the schema repository instance is defined in the Citrus Spring configuration with the id "schemaRepository". Spring is then able to inject this schema repository instance into every message receiving action. The receiving action receives XML messages and will ask the repository for a matching schema definition file in order to validate the document structure.

The connection between XML messages and xsd schema files is done with the target namespace that is defined inside the schema definition. The target namespace of the schema definition will match the namespace of the root element in the received XML message. Using this central schema repository you do not have to wire XML messages and schema files together every time. This is done automatically over the target namespace.

Important

In case Citrus receives a classified XML message using namespaces Citrus will try to validate the structure of the message by default. Consequently you will also get errors in case no matching schema definition file is found inside the schema repository. So if you explicitly do not want to validate the XML schema for some reason you have to disable the validation explicitly in your test.

```
<receive with="getCustomerRequestReceiver">
    <message schema-validation="false">
        <validate path="//ns2:RequestMessage/ns2:MessageHeader/ns2:RequestTag"
             value="${requestTag}"/>
        <validate path="//ns2:RequestMessage/ns2:MessageHeader/ns2:CorrelationId"
             value="${correlationId}"/>
        <namespace prefix="ns2" value="http://testsuite/default"/>
```

```
    </message>
    <header>
        <element name="Operation" value="GetCustomer"/>
        <element name="RequestTag" value="${requestTag}"/>
    </header>
</receive>
```

This way might sound annoying for you but in our opinion it is very important to validate the structure of the received XML messages, so disabling the schema validation should not be the standard for all tests. Disabling automatic schema validation should only apply to special situations.

18.2. DTD validation

XML DTD (Document type definition) is another way to validate the structure of a XML document. Many people say that DTD is deprecated and XML schema is the much more efficient way to describe the rules of a XML structure. We do not disagree with that, but we also know that legacy systems might still use DTD. So in order to avoid validation errors we have to deal with DTD validation as well.

First thing you can do about DTD validation is to specify an inline DTD in your expected message template.

```
<receive with="getTestMessageReceiver">
    <message schema-validation="false">
        <data>
        <![CDATA[
            <!DOCTYPE root [
                <!ELEMENT root (message)>
                <!ELEMENT message (text)>
                <!ELEMENT text (#PCDATA)>
                ]>
            <root>
                <message>
                    <text>Hello TestFramework!</text>
                </message>
            </root>
        ]]>
        <data/>
    </message>
</receive>
```

The system under test may also send the message with a inline DTD definition. So validation will succeed.

In most cases the DTD is referenced as external .dtd file resource. You can do this in your expected message template as well.

```
<receive with="getTestMessageReceiver">
    <message schema-validation="false">
        <data>
        <![CDATA[
            <!DOCTYPE root SYSTEM
                        "com/consol/citrus/validation/example.dtd">
            <root>
                <message>
                    <text>Hello TestFramework!</text>
                </message>
            </root>
        ]]>
        <data/>
    </message>
</receive>
```

Chapter 19. Customize meta information

Test cases in Citrus are usually provided with some meta information like the author's name or the date of creation. In Citrus you are able to extend this test case meta information with your own very specific criteria.

By default a test case comes shipped with meta information that looks like this:

```
<testcase name="PwdChange_OK_1_Test">
    <meta-info>
        <author>Christoph</author>
        <creationdate>2010-01-18</creationdate>
        <status>FINAL</status>
        <last-updated-by>Christoph</last-updated-by>
        <last-updated-on>2010-01-18T15:00:00</last-updated-on>
    </meta-info>

    [...]
</testcase>
```

You can quite easily add data to this section in order to meet your individual testing strategy. Let us have a simple example to show how it is done.

First of all we define a custom XSD schema describing the new elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:tns="http://www.citrusframework.org/samples/my-testcase-info"
        targetNamespace="http://www.citrusframework.org/samples/my-testcase-info"
        elementFormDefault="qualified">

    <element name="requirement" type="string"/>
    <element name="pre-condition" type="string"/>
    <element name="result" type="string"/>
    <element name="classification" type="string"/>
</schema>
```

We have four simple elements (*requirement*, *pre-condition*, *result* and *classification*) all typed as string. These new elements later go into the test case meta information section.

After we added the new XML schema file to the classpath of our project we need to announce the schema to Spring. As you might know already a Citrus test case is nothing else but a simple Spring configuration file with customized XML schema support. If we add new elements to a test case Spring needs to know the XML schema for parsing the test case configuration file. See the *spring.schemas* file usually placed in the META-INF/spring.schemas in your project.

The file content for our example will look like follows:

```
http://www.citrusframework.org/samples/my-testcase-info/my-testcase-info.xsd=com/consol/citrus/schemas/my-testcase-i
```

So now we are finally ready to use the new meta-info elements inside the test case:

```
<?xml version="1.0" encoding="UTF-8"?>
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:custom="http://www.citrusframework.org/samples/my-testcase-info"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
      http://www.citrusframework.org/schema/testcase
      http://www.citrusframework.org/schema/testcase/citrus-testcase-1.1.xsd
      http://www.citrusframework.org/samples/my-testcase-info
      http://www.citrusframework.org/samples/my-testcase-info/my-testcase-info.xsd">
```

```
    <testcase name="PwdChange_OK_1_Test">
        <meta-info>
            <author>Christoph</author>
            <creationdate>2010-01-18</creationdate>
            <status>FINAL</status>
            <last-updated-by>Christoph</last-updated-by>
            <last-updated-on>2010-01-18T15:00:00</last-updated-on>
            <custom:requirement>REQ10001</custom:requirement>
            <custom:pre-condition>Existing user, sufficient rights</custom:pre-condition>
            <custom:result>Password reset in database</custom:result>
            <custom:classification>PasswordChange</custom:classification>
        </meta-info>

        [...]
    </testcase>
</spring:beans>
```

Note

We use a separate namespace declaration with a custom namespace prefix “custom” in order to declare the new XML schema to our test case. Of course you can pick a namespace url and prefix that fits best for your project.

As you see it is quite easy to add custom meta information to your Citrus test case. The customized elements may be precious for automatic reporting. XSL transformations for instance are able to read those meta information elements in order to generate automatic test reports and documentation.

You can also declare our new XML schema in the Eclipse preferences section as user specific XML catalog entry. Then even the schema code completion in your Eclipse XML editor will be available for our customized meta-info elements.

Chapter 20. Debugging received messages

Citrus will receive a lot of messages during a test run. The user may want to persist these messages to the filesystem for further investigations.

Citrus offers an easy way to debug all received messages to the file system. You need to enable a specific aspect in the Citrus Spring configuration.

```
<bean class="com.consol.citrus.aop.StoreMessageInterceptorAspect"/>
```

Just add this bean to the Spring configuration and Citrus will debug received messages to the file system by generating files containing the message header and message body content.

For example:

```
logs/debug/messages/message1.header

logs/debug/messages/message1.body

logs/debug/messages/message2.header

logs/debug/messages/message2.body
```

The framework uses a simple counter that is increased whenever a message is written to the file system. Citrus separates message header and message body into extra files with respective extension (".header" and ".body"). By default the debug directory is `logs/debug/messages/` relative to the project root directory. But you can set your own debug directory in the configuration.

```
<bean class="com.consol.citrus.aop.StoreMessageInterceptorAspect">
  <property name="debugDirectory" value="debugging/messages"/>
</bean>
```

Note

As the file counter is always reset to 0 after a test run the message files may be overwritten. So you eventually need to save the generated message debug files before running another group of test cases.

Chapter 21. Reporting and test results

The framework generates different reports and results after a test run for you. These report and result pages will help you to get an overview of the test cases that were executed and which one were failing.

21.1. Console logging

During the test run the framework will provide a huge amount of information that is printed out to the console. This includes current test progress, validation results and error information. This enables the user to quickly supervise the test run progress. Failures in tests will be printed to the console just the time the error occurred. The detailed stack trace information and the detailed error messages are helpful to get the idea what went wrong.

As the console output might be limited to a defined buffer limit, the user may not be able to follow the output to the very beginning of the test run. Therefore the framework additionally prints all information to a log file according to the logging configuration.

The logging mechanism uses the SLF4J logging framework. SLF4J is independent of logging framework implementations on the market. So in case you use Log4J logging framework the specified log file path as well as logging levels can be freely configured in the respective log4j.xml file in your project. At the end of a test run the combined test results get printed to both console and log file. The overall test results look like following example:

```
TEST RESULTS citrus-default-testsuite

  [...]
  HelloService_Ok_1             : successfull
  HelloService_Ok_2             : successfull
  EchoService_Ok_1              : successfull
  EchoService_Ok_2              : successfull
  EchoService_TempError_1       : successfull
  EchoService_AutomacticRetry_1 : successfull
  [...]

  Found 175 test cases to execute
  Skipped 0 test cases (0.0%)
  Executed 175 test cases
  Tests failed:         0 (0.0%)
  Tests successfully: 175 (100.0%)
```

Failed tests will be marked as failed in the result list. The framework will give a short description of the error cause while the detailed stack trace information can be found in the log messages that were made during the test run.

```
HelloService_Ok_3 : failed - Exception is Action timed out
```

21.2. JUnit reports

As tests are executed as TestNG test cases, the framework will also generate JUnit compliant XML and HTML reports. JUnit test reports are very popular and find support in many build management and development tools. In general the Citrus test reports give you an overall picture of all tests and tell you which of them were failing.

Build management tools like Hudson, Bamboo or CruiseControl can easily import and display the generated JUnit XML results. Please have a look at the TestNG and JUnit documentation for more information about this topic as well as the build management tools (Hudson, Bamboo, CruiseControl, etc.) to find out how to integrate the tests results.

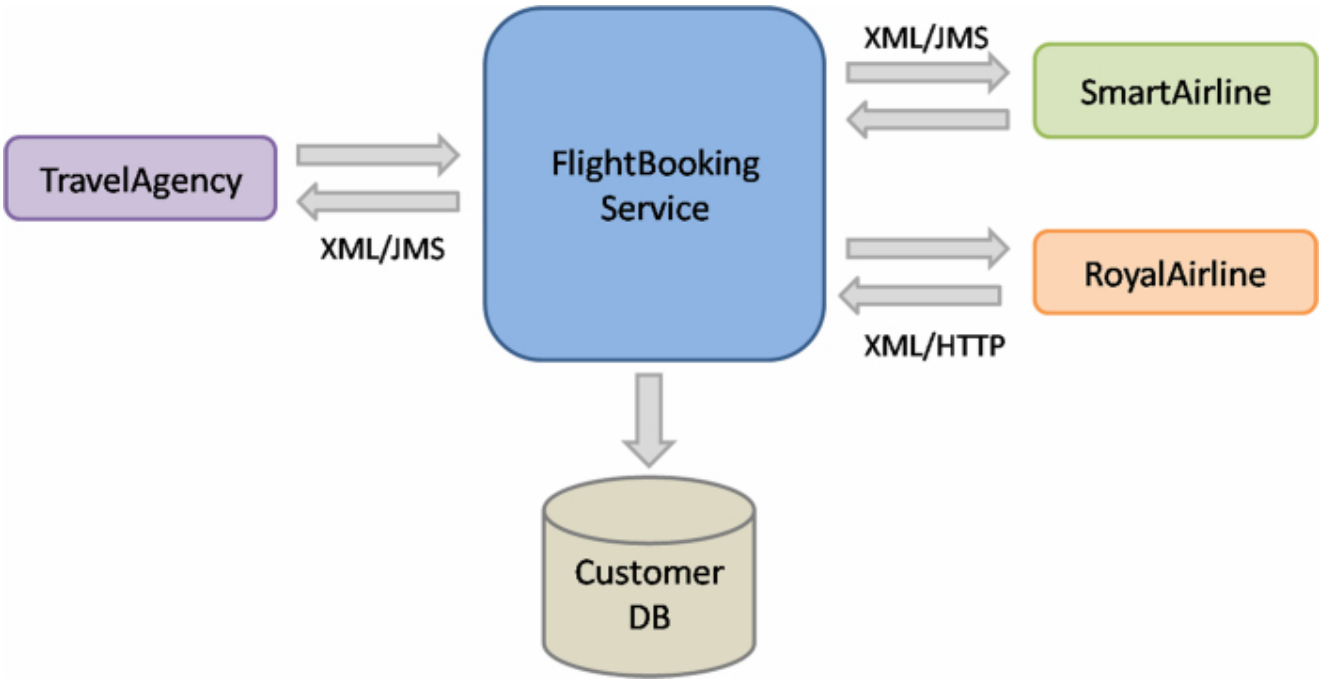
Appendix A. Citrus Samples

This part will show you some sample applications that are tested using Citrus integration tests. See below a list of all samples.

A.1. The FlightBooking sample

A simple project example should give you the idea how Citrus works. The system under test is a flight booking service that handles travel requests from a travel agency. A travel request consists of a complete travel route including several flights. The FlightBookingService application will split the complete travel booking into separate flight bookings that are sent to the respective airlines in charge. The booking and customer data is persisted in a database.

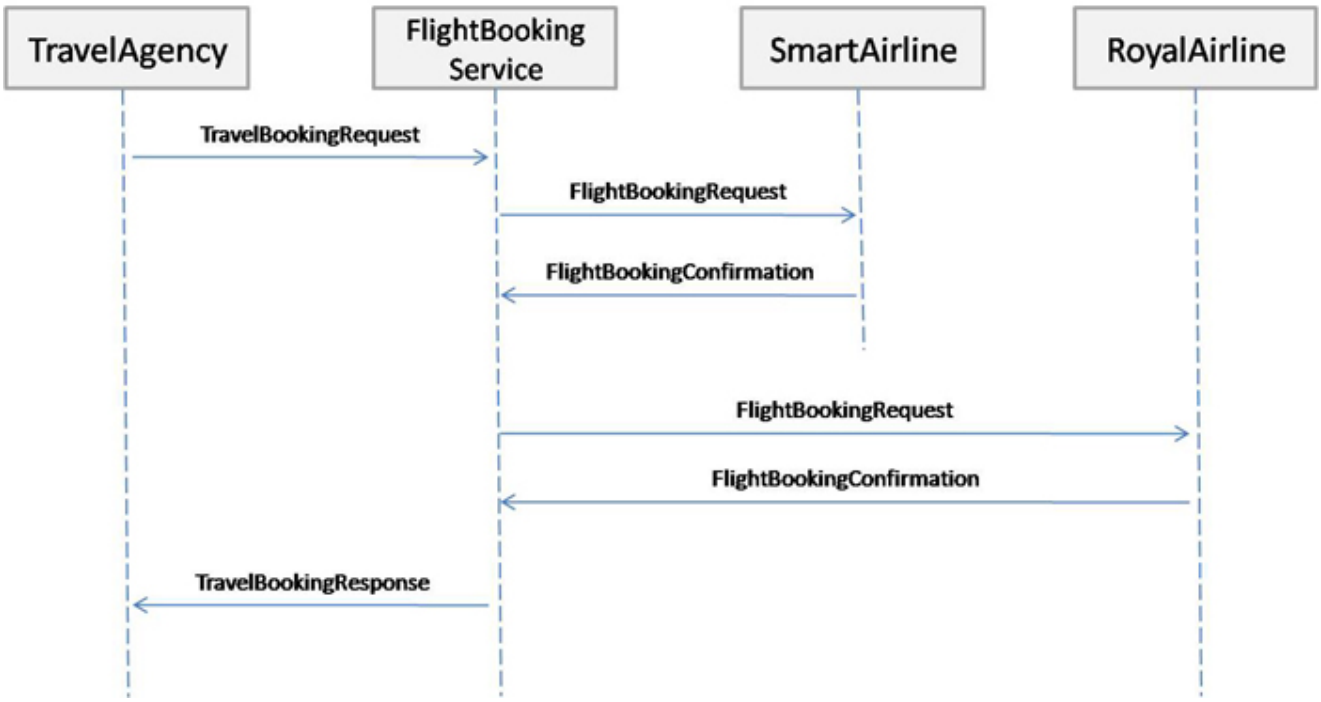
The airlines will confirm or deny the flight bookings. The FlightBookingService application consolidates all incoming flight confirmations and combines them to a complete travel confirmation or denial that is sent back to the travel agency. Next picture tries to put the architecture into graphics:



In our example two different airlines are connected to the FlightBookingService application: the SmartAriline over JMS and the RoyalAirline over Http.

A.1.1. The use case

The use case that we would like to test is quite simple. The test should handle a simple travel booking and expect a positive processing to the end. The test case neither simulates business errors nor technical problems. Next picture shows the use case as a sequence diagram.



The travel agency puts a travel booking request towards the system. The travel booking contains two seperate flights. The flight requests are published to the airlines (SmartAirline and RoyalAirline). Both airlines confirm the flight bookings with a positive answer. The consolidated travel booking response is then sent back to the travel agency.

A.1.2. Configure the simulated systems

Citrus simulates all surrounding applications in their behavior during the test. The simulated applications are: TravelAgency, SmartAirline and RoyalAirline. The simulated systems have to be configured in the Citrus configuration first. The configuration is done in Spring XML configuration files, as Citrus uses Spring to glue all its services together.

First of all we have a look at the TravelAgency configuration. The TravelAgency is using JMS to connect to our tested system, so we need to configure this JMS connection in Citrus.

```
<bean name="connectionFactory"
        class="org.apache.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<citrus:jms-message-sender id="travelAgencyBookingRequestSender"
                      destination-name="${travel.agency.request.queue}"/>

<citrus:jms-message-receiver id="travelAgencyBookingResponseReceiver"
                      destination-name="${travel.agency.response.queue}"/>
```

This is all Citrus needs to send and receive messages over JMS in order to simulate the TravelAgency. By default all JMS message senders and receivers need a connection factory. Therefore Citrus is searching for a bean named "connectionFactory". In the example we connect to a ActiveMQ message broker. A connection to other JMS brokers like TIBCO EMS or IBM Websphere MQ is possible too by simply changing the connection factory implementation.

The identifiers of the message senders and receivers are very important. We should think of suitable ids that give the reader a first hint what the sender/receiver is used for. As we want to simulate the TravelAgency in combination with sending booking requests our id is "travelAgencyBookingRequestSender" for example.

The sender and receivers do also need a JMS destination. Here the destination names are provided by property expressions. The Spring IoC container resolves the properties for us. All we need to do is publish the property file to the Spring container like this.

```
<bean name="propertyLoader"
   class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>citrus.properties</value>
        </list>
    </property>
    <property name="ignoreUnresolvablePlaceholders" value="true"/>
</bean>
```

The citrus.properties file is located in our project's resources folder and defines the actual queue names besides other properties of course:

```
#JMS queues
travel.agency.request.queue=Travel.Agency.Request.Queue
travel.agency.response.queue=Travel.Agency.Response.Queue
smart.airline.request.queue=Smart.Airline.Request.Queue
smart.airline.response.queue=Smart.Airline.Response.Queue
royal.airline.request.queue=Royal.Airline.Request.Queue
```

What else do we need in our Spring configuration? There are some basic beans that are commonly defined in a Citrus application but I do not want to bore you with these details. So if you want to have a look at the "citrus-context.xml" file in the resources folder and see how things are defined there.

We continue with the first airline to be configured the SmartAirline. The SmartAirline is also using JMS to communicate with the FlightBookingService. So there is nothing new for us, we simply define additional JMS message senders and receivers.

```
<citrus:jms-message-receiver id="smartAirlineBookingRequestReceiver"
                      destination-name="${smart.airline.request.queue}"/>

<citrus:jms-message-sender id="smartAirlineBookingResponseSender"
                      destination-name="${smart.airline.response.queue}"/>
```

We do not define a new JMS connection factory because TravelAgency and SmartAirline are using the same message broker instance. In case you need to handle multiple connection factories simply define the connection factory with the attribute "connection-factory".

```
<citrus:jms-message-receiver id="smartAirlineBookingRequestReceiver"
                          destination-name="${smart.airline.request.queue}"
                          connection-factory="smartAirlineConnectionFactory"/>

<citrus:jms-message-sender id="smartAirlineBookingResponseSender"
                        destination-name="${smart.airline.response.queue}"
                        connection-factory="smartAirlineConnectionFactory"/>
```

A.1.3. Configure the Http adapter

The RoyalAirline is connected to our system using Http request/response communication. This means we have to simulate a Http server in the test that accepts client requests and provides proper responses. Citrus offers a Http server implementation that will listen on a port for client requests. The adapter forwards incoming request to the test engine over JMS and receives a proper response that is forwarded as a Http response to the client. The next picture shows this mechanism in detail.

The RoyalAirline adapter receives client requsts over Http and sends them over JMS to a message receiver as we alredy know it. The test engine validates the received request and provides a proper response back to the adapter. The adapter will transform the response to Http again and publishes it to the calling client. Citrus offers these kind of adapters for Http and SOAP communication. By writing your own adapters like this you will be able to extend Citrus so it works with protocols that are not supported yet.

Let us define the Http adapter in the Spring configuration:

```
<citrus-http:server id="royalAirlineHttpServer"
                      port="8091"
                      uri="/flightbooking"
                      message-handler="jmsForwardingMessageHandler"/>

<bean id="jmsForwardingMessageHandler"
        class="com.consol.citrus.http.handler.JmsConnectingMessageHandler">
    <property name="destinationName" value="${royal.airline.request.queue}"/>
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="replyTimeout" value="2000"/>
</bean>

<citrus:jms-sync-message-receiver id="royalAirlineBookingRequestReceiver"
                           destination-name="${royal.airline.request.queue}"/>

<citrus:jms-reply-message-sender id="royalAirlineBookingResponseSender"
                           reply-destination-holder="royalAirlineBookingRequestReceiver"/>
```

We need to configure a Http server instance with a port, a request URI and a message handler. We define the JMS forwarding message handler to handle request as described. In Addition to the message handler we also need synchrpnous JMS message sender and receiver instances. That's it! We are able to receive Http request in order to simulate the RoyalAirline application. What is missing now? The test case definition itself.

A.1.4. The test case

The test case definition is also a Spring configuration file. Citrus offers a customized XML syntax to define a test case. This XML test defining language is supposed to be easy to understand and more specific to the domain we are dealing with. Next listing shows the whole test case definition. Keep in mind that a test case defines every step in the use case. So we define sending and receiving actions of the use case as described in the sequence diagram we saw earlier.

```
<?xml version="1.0" encoding="UTF-8"?>
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
              xmlns:spring="http://www.springframework.org/schema/beans"
              xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xsi:schemaLocation="http://www.springframework.org/schema/beans
              http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
              http://www.citrusframework.org/schema/testcase
              http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd">
    <testcase name="FlightBookingTest">
        <meta-info>
            <author>Christoph Deppisch</author>
            <creationdate>2009-04-15</creationdate>
            <status>FINAL</status>
            <last-updated-by>Christoph Deppisch</last-updated-by>
            <last-updated-on>2009-04-15T00:00:00</last-updated-on>
        </meta-info>
```

```
        <description>
            Test flight booking service.
        </description>
        <variables>
            <variable name="correlationId"
                value="citrus:concat('Lx1x', 'citrus:randomNumber(10)')"/>
            <variable name="customerId"
                value="citrus:concat('Mx1x', citrus:randomNumber(10))"/>
        </variables>
        <actions>
            <send with="travelAgencyBookingRequestSender">
                <message>
                    <data>
                      <![CDATA[
                        <TravelBookingRequestMessage
                          xmlns="http://www.consol.com/schemas/TravelAgency">
                          <correlationId>${correlationId}</correlationId>
                          <customer>
                            <id>${customerId}</id>
                            <firstname>John</firstname>
                            <lastname>Doe</lastname>
                          </customer>
                          <flights>
                            <flight>
                              <flightId>SM 1269</flightId>
                              <airline>SmartAirline</airline>
                              <fromAirport>MUC</fromAirport>
                              <toAirport>FRA</toAirport>
                              <date>2009-04-15</date>
                              <scheduledDeparture>11:55:00</scheduledDeparture>
                              <scheduledArrival>13:00:00</scheduledArrival>
                            </flight>
                            <flight>
                              <flightId>RA 1780</flightId>
                              <airline>RoyalAirline</airline>
                              <fromAirport>FRA</fromAirport>
                              <toAirport>HAM</toAirport>
                              <date>2009-04-15</date>
                              <scheduledDeparture>16:00:00</scheduledDeparture>
                              <scheduledArrival>17:10:00</scheduledArrival>
                            </flight>
                          </flights>
                        </TravelBookingRequestMessage>
                      ]]>
                    </data>
                </message>
                <header>
                    <element name="correlationId" value="${correlationId}"/>
                </header>
            </send>

            <receive with="smartAirlineBookingRequestReceiver">
                <message>
                    <data>
                      <![CDATA[
                        <FlightBookingRequestMessage
                          xmlns="http://www.consol.com/schemas/AirlineSchema">
                          <correlationId>${correlationId}</correlationId>
                          <bookingId>???</bookingId>
                          <customer>
                            <id>${customerId}</id>
                            <firstname>John</firstname>
                            <lastname>Doe</lastname>
                          </customer>
                          <flight>
                            <flightId>SM 1269</flightId>
                            <airline>SmartAirline</airline>
                            <fromAirport>MUC</fromAirport>
                            <toAirport>FRA</toAirport>
                            <date>2009-04-15</date>
                            <scheduledDeparture>11:55:00</scheduledDeparture>
                            <scheduledArrival>13:00:00</scheduledArrival>
                          </flight>
                        </FlightBookingRequestMessage>
                      ]]>
                    </data>
                    <ignore path="//:FlightBookingRequestMessage/:bookingId"/>
                </message>
                <header>
                    <element name="correlationId" value="${correlationId}"/>
                </header>
                <extract>
```

```
            <message path="//:FlightBookingRequestMessage/:bookingId"
                     variable="${smartAirlineBookingId}"/>
        </extract>
    </receive>

    <send with="smartAirlineBookingResponseSender">
        <message>
            <data>
              <![CDATA[
                <FlightBookingConfirmationMessage
                  xmlns="http://www.consol.com/schemas/AirlineSchema">
                  <correlationId>${correlationId}</correlationId>
                  <bookingId>${smartAirlineBookingId}</bookingId>
                  <success>true</success>
                  <flight>
                    <flightId>SM 1269</flightId>
                    <airline>SmartAirline</airline>
                    <fromAirport>MUC</fromAirport>
                    <toAirport>FRA</toAirport>
                    <date>2009-04-15</date>
                    <scheduledDeparture>11:55:00</scheduledDeparture>
                    <scheduledArrival>13:00:00</scheduledArrival>
                  </flight>
                </FlightBookingConfirmationMessage>
              ]]>
            </data>
        </message>
        <header>
            <element name="correlationId" value="${correlationId}"/>
        </header>
    </send>

    <receive with="royalAirlineBookingRequestReceiver">
        <message>
            <data>
              <![CDATA[
                <FlightBookingRequestMessage
                  xmlns="http://www.consol.com/schemas/FlightBooking/AirlineSchema">
                  <correlationId>${correlationId}</correlationId>
                  <bookingId>???</bookingId>
                  <customer>
                      <id>${customerId}</id>
                      <firstname>John</firstname>
                      <lastname>Doe</lastname>
                  </customer>
                  <flight>
                    <flightId>RA 1780</flightId>
                    <airline>RoyalAirline</airline>
                    <fromAirport>FRA</fromAirport>
                    <toAirport>HAM</toAirport>
                    <date>2009-04-15</date>
                    <scheduledDeparture>16:00:00</scheduledDeparture>
                    <scheduledArrival>17:10:00</scheduledArrival>
                  </flight>
                </FlightBookingRequestMessage>
              ]]>
            </data>
            <ignore path="//:FlightBookingRequestMessage/:bookingId"/>
        </message>
        <header>
            <element name="correlationId" value="${correlationId}"/>
        </header>
        <extract>
            <message path="//:FlightBookingRequestMessage/:bookingId"
                     variable="${royalAirlineBookingId}"/>
        </extract>
    </receive>

    <send with="royalAirlineBookingResponseSender">
        <message>
            <data>
              <![CDATA[
                <FlightBookingConfirmationMessage
                  xmlns="http://www.consol.com/schemas/AirlineSchema">
                  <correlationId>${correlationId}</correlationId>
                  <bookingId>${royalAirlineBookingId}</bookingId>
                  <success>true</success>
                  <flight>
                    <flightId>RA 1780</flightId>
                    <airline>RoyalAirline</airline>
                    <fromAirport>FRA</fromAirport>
                    <toAirport>HAM</toAirport>
```

```
                        <date>2009-04-15</date>
                        <scheduledDeparture>16:00:00</scheduledDeparture>
                        <scheduledArrival>17:10:00</scheduledArrival>
                      </flight>
                    </FlightBookingConfirmationMessage>
                  ]]>
                </data>
            </message>
            <header>
                <element name="correlationid" value="${correlationId}"/>
            </header>
        </send>

        <receive with="travelAgencyBookingResponseReceiver">
            <message>
                <data>
                  <![CDATA[
                    <TravelBookingResponseMessage
                      xmlns="http://www.consol.com/schemas/TravelAgency">
                      <correlationId>${correlationId}</correlationId>
                      <success>true</success>
                      <flights>
                        <flight>
                          <flightId>SM 1269</flightId>
                          <airline>SmartAirline</airline>
                          <fromAirport>MUC</fromAirport>
                          <toAirport>FRA</toAirport>
                          <date>2009-04-15</date>
                          <scheduledDeparture>11:55:00</scheduledDeparture>
                          <scheduledArrival>13:00:00</scheduledArrival>
                        </flight>
                        <flight>
                          <flightId>RA 1780</flightId>
                          <airline>RoyalAirline</airline>
                          <fromAirport>FRA</fromAirport>
                          <toAirport>HAM</toAirport>
                          <date>2009-04-15</date>
                          <scheduledDeparture>16:00:00</scheduledDeparture>
                          <scheduledArrival>17:10:00</scheduledArrival>
                        </flight>
                      </flights>
                    </TravelBookingResponseMessage>
                  ]]>
                </data>
            </message>
            <header>
                <element name="correlationId" value="${correlationId}"/>
            </header>
        </receive>

    </actions>
  </testcase>
</spring:beans>
```

Similar to a sequence diagram the test case describes every step of the use case. At the very beginning the test case gets name and its meta information. Following with the variable values that are used all over the test. Here it is the correlationId and the customerId that are used as test variables. Inside message templates header values the variables are referenced several times in the test

```
<correlationId>${correlationId}</correlationId>
```

```
<id>${customerId}</id>
```

The sending/receiving actions use a previously defined message sender/receiver. This is the link between test case and basic Spring configuration we have done before.

```
<send with="travelAgencyBookingRequestSender">
```

The sending action chooses a message sender to actually send the message using a message transport (JMS, Http, SOAP, etc.). After sending this first "TravelBookingRequestMessage" request the test case expects the first "FlightBookingRequestMessage" message on the SmartAirline JMS

destination. In case this message is not arriving in time the test will fail with errors. In positive case our FlightBookingService works well and the message arrives in time. The received message is validated against a defined expected message template. Only in case all content validation steps are successful the test continues with the action chain. And so the test case proceeeds and works through the use case until every message is sent respectively received and validated. The use case is done automatically without human interaction. Citrus simulates all surrounding applications and provides detailed validation possibilities of messages.