Citrus Framework - Reference Documentation

Version 1.3

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. What's new?!

The new Citrus release introduces new features and changes that we discuss in the following sections. We want you to get an overview of what has happened in the last release so you can spot your favorite.

1.1. Java DSL

Citrus users, in particular those with development experience, do often tell me about the nasty XML code they have to deal with for writing Citrus test definitions. Developers want to write Java code rather than XML. Although I personally do like the Citrus XML test syntax we have introduced a Java DSL language for writing Citrus tests with Java only.

We have introduced the Java DSL to all major test action features in Citrus so you can switch without having to worry about loosing functionality. Details can be seen in the test action section where we added Java DSL examples almost everywhere (Chapter 11, *Test actions*). The basic Java DSL setup is described in Chapter 4, *Test case*.

1.2. XHTML message validation

Message validation for Html code was not really comfortable as Html does not confirm to be wellformed and valid XML syntax. XHTML tries to close this gap by automatically resolving all Html specific XML syntax rule violations. With Citrus 1.3 we introduced a XHTML message validator which does the magic for converting Html code to proper wellformed and valid XML. In a test case you can then use the full XML validation power in Citrus in order to validate incoming Html messages. Section Section 8.2, “XHTML message validation” deals with the new validation capabilities for Html.

1.3. Multiple SOAP fault detail support

SOAP fault messages can hold many SOAP fault detail elements. Citrus was able to handle a single SOAP fault detail on sending and receiving test actions from the very beginning but multiple SOAP fault detail elements were not supported. Fortunately things are getting better and you can send and receive as many fault detail elements as you like in Citrus 1.3. For each SOAP fault detail you can specify individual validation rules and expectations. See Section 18.5, “SOAP faults” for detailed description.

1.4. Jetty server security handler

With our Jetty server component you can set up Http mock servers very easy. The server is automatically configured to accept Http client connections and to load a Spring application context on startup. Now you can also set some more details on this automatic server configuration (e.g. server context path, servlet names or servlet mappings). In addition to that you can access the security context of the web container. This enables you to set up security constraints such as basic authentication on server resources. Clients are then forced to authenticate properly when accessing the server. Unauthorized users will get *401 access denied* errors immediately. See Section 16.6, “Server basic authentication” for details. Of course this also applies to our SOAP WebService Jetty

server components (Section 18.9, “Basic authentication on the server”).

1.5. Test actors

We introduced a new concept of test actors for sending and receiving test actions. This enables you to link a test actor (e.g. interface partner application, backend application) to a test action in your test. Following from that you can enable/disable test actors and all linked test actions very easy. This enables us to reuse Citrus test cases in end-to-end test scenarios where not all interface partners get simulated by Citrus. If you like to read more about this concept follow the detailed instruction in Chapter 5, *Test actors*.

1.6. Simulate Http error codes with SOAP

Citrus provides SOAP WebServices server simulation with clients connecting to the server sending SOAP requests. As a server Citrus is now able to simulate Http error codes like *404 Not found* and *500 Internal server error*. Before that the Citrus SOAP server had to always respond with a proper SOAP response or SOAP fault. See Section 18.6, “Simulate Http error codes with SOAP” for details.

1.7. SSH server and client

The Citrus family has raised a new member in adding SSH connectivity. With the new SSH module you are able to provide a full stack SSH server. The SSH server accepts client connections and you as a tester can simulate any SSH server functionality with proper validation as it is known to Citrus SOAP and HTTP modules. In addition to that you can also use the Citrus SSH client in order to connect to an external SSH server. You can execute SSH commands on the SSH server and validate the respective response data. The full description is provided in Chapter 17, *SSH support*.

1.8. ANT run test action

With this new test action you can call ANT builds from your test case. The action executes one or more ANT build targets on a build.xml file. You can specify build properties that get passed to the ANT build and you can add a custom build listener. In case the ANT build run fails the test fails accordingly with the build exception. See Section 11.18, “Running ANT build targets” for details.

1.9. Polling interval for reply handlers

With synchronous communication in Citrus we always have a combination of a synchronous message sender and a reply handler component. These two perform a handshake when passing synchronous reply messages to the test for further processing such as message validation. While the sender is waiting for the synchronous response to arrive the reply handler polls for the reply message. This polling for reply messages was done in a static way which often led to time delays according to long polling intervals. Now with Citrus 1.3 you can set the polling-interval for the reply handler as you like. This setting is valid for all reply handler components in Citrus (citrus-jms, citrus-http, citrus-ws, citrus-channel, citrus-shh, and so on).

1.10. Bugfixes

Bug fixing is part of our daily business. Finding and solving issues makes Citrus better every day. For a detailed bugfix listing please refer to the complete changes log in JIRA (https://citrusframework.atlassian.net).

1.11. Upgrading from version 1.2

If you are coming from Citrus 1.2 you may have to look at the following points in order to have a smooth upgrade to the new release version.

- *Jetty version upgrade* We are using Jetty a lot for starting Http server mocks within Citrus. In order to stay up to date we upgraded to Jetty 8.1.7 version with this Citrus release. This implies that package names did change for Jetty API. In general there is no conflict for you as a Citrus user, but you may want to adjust your logging configuration according to new Jetty packages. Jetty package names did change from *ord.mortbay* to *org.eclipse.jetty*.
- *Spring version upgrade* Staying up to date with the versions of 3rd library dependencies is quite important for us. So we upgrade our dependencies to newer versions with each release. As we did only upgrade minor versions there is no significant change or problems to be expected. However you may take care on versions and release changes in the Spring world for details and migration.
- *TIBCO module* We decided to put the TIBCO module separately as it is a very special connectivity adapter for TIBCO software only. So you will not find the TIBCO module within the Citrus distribution anymore. We will maintain a TIBCO connectivity adapter separately in the future.

Chapter 2. 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.

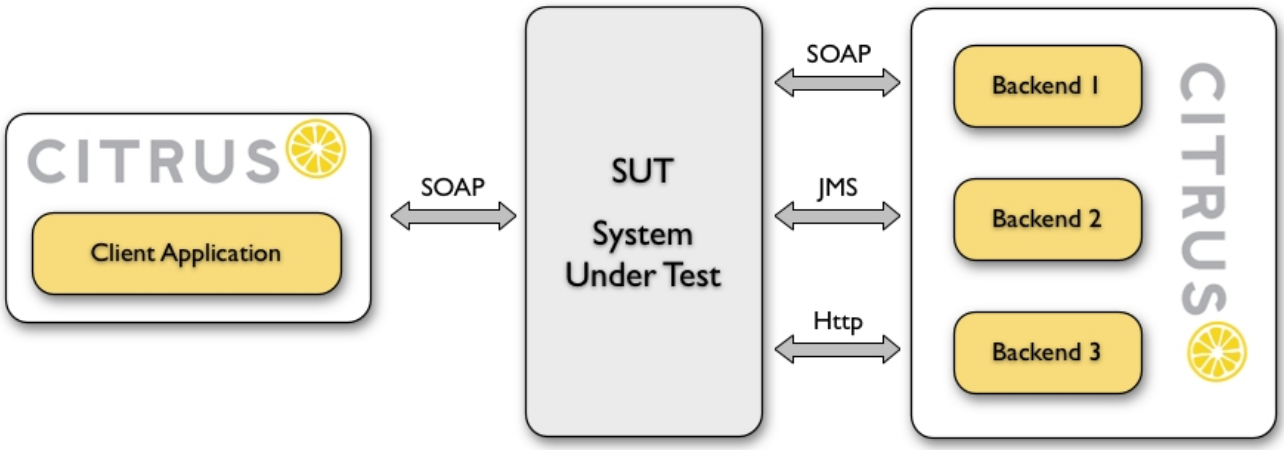
2.1. Overview

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.

2.2. 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, bug fixing 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 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 against an expected message template. Besides messaging actions Citrus is also able to perform arbitrary other test actions. The client can, for example, perform a database query between requests to ensure that the previous test step stored what was expected.

The Citrus test case runs fully automated as a 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 build process (e.g. using Maven) like a normal unit test case would be. This gives clients automated integration testing to help ensure 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 3. 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.

3.1. 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 life cycles 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.

3.1.1. Use Citrus Maven archetype

If you start from scratch or in case you would like to have Citrus operating in a separate Maven module you can use the Citrus Maven archetype to create a new Maven project. The archetype will setup a basic Citrus project structure with basic settings and files.

```
mvn archetype:generate -DarchetypeCatalog=http://citrusframework.org

Choose archetype:
1: http://citrusframework.org -> citrus-archetype (Basic archetype for Citrus integration test project)
Choose a number: 1

Define value for groupId: com.consol.citrus.samples
Define value for artifactId: citrus-sample
Define value for version: 1.0-SNAPSHOT
Define value for package: com.consol.citrus.samples
```

We load the archetype information from "http://citrusframework.org" and choose the Citrus basic archetype. Now you have to define several values for your project: the groupId, the artifactId, the package and the project version. After that we are done! Maven created a Citrus project structure for us which is ready for testing. You should see the following basic project folder structure.

```
citrus-sample
  |   + src
  |   |   + main
  |   |    |   + java
  |   |    |   + resources
  |   |   + citrus
  |   |    |   + java
  |   |    |   + resources
  |   |    |   + tests
  pom.xml
```

The Citrus project is absolutely ready for testing. With Maven we can build, package, install and test

our project right away without any adjustments. Try to execute the following commands:

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

Note

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

3.1.2. Add Citrus to existing Maven project

In case you already have a proper Maven project you can also integrate Citrus with it. Just add the Citrus project dependencies in your Maven pom.xml as a dependency 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>1.3</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>1.3</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 our Maven setup tutorial on http://www.citfrusframework.org.

3.2. 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.

3.2.1. Preconditions

Before we start with the Citrus setup be sure to meet the following preconditions. 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.

3.2.2. 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.

3.2.3. 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-1.3.jar"/>

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

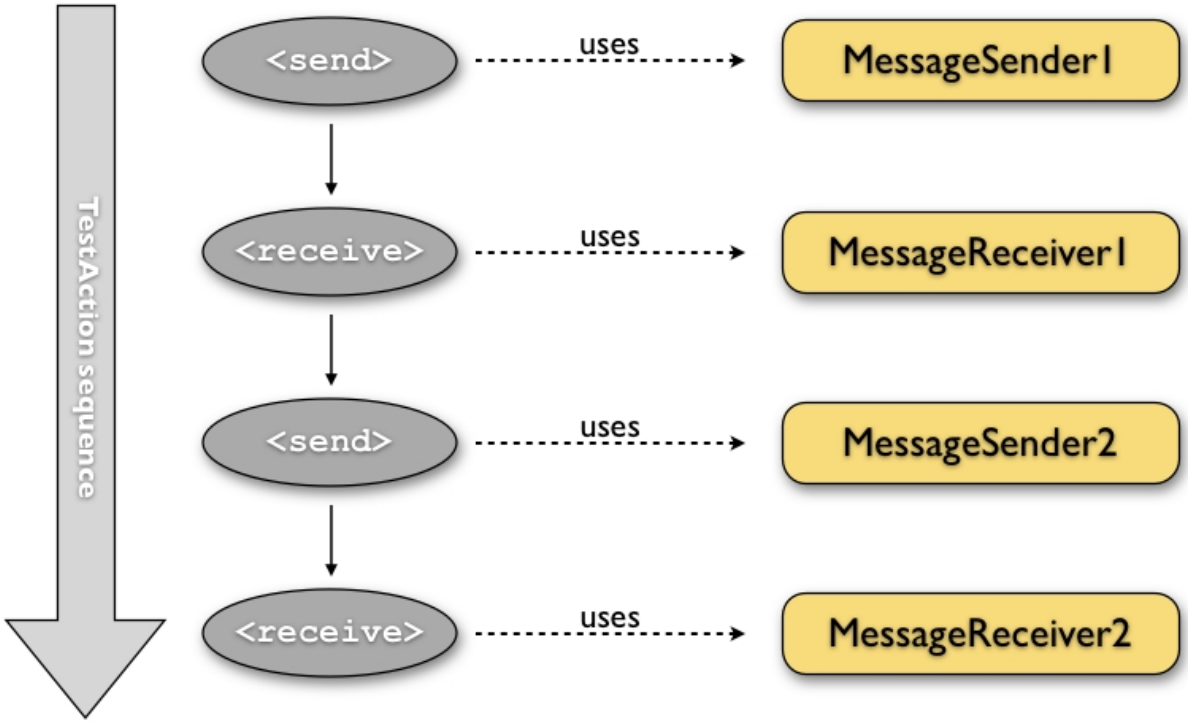
```
  <target name="create.test" description="Creates a new empty test case">
    <input message="Enter test name:" addproperty="test.name"/>
    <input message="Enter test description:" addproperty="test.description"/>
    <input message="Enter author's name:" addproperty="test.author" defaultvalue="${default.test.author}"/>
    <input message="Enter package:" addproperty="test.package" defaultvalue="${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 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 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 4. Test case

Now let us start writing test cases! A test case in Citrus describes all steps for a certain use case. The Citrus integration test holds different test actions that are executed in sequence during the test. Typically with message-based enterprise applications the sending and receiving of messages represent the main actions inside a test. But you will learn that Citrus is able to add much more logic to your test, such as connecting to the database, transforming data, adding loops and iterations. With the default Citrus action set you can accomplish very complex use case integration tests. We will introduce all possible actions step by step and learn how to use various message transports within the test.



The figure above describes a typical test action sequence in Citrus. A list of test actions with purpose to send and receive messages using predefined Citrus MessageSender and MessageReceiver components. So how do we define those test cases? In general Citrus specifies test cases as Java classes. With TestNG or JUnit you can execute the Citrus tests within your Java runtime as you would do within unit testing. You can code the Citrus test in a single Java class doing assertions and using Spring's dependency injection mechanisms. If you are not familiar to writing Java code you can also write Citrus tests as XML files. Whatever test language you choose for Citrus the whole test case description takes place in one single file (Java or XML). This chapter will introduce the custom XML schema language as well as the Java domain specific language so you will be able to write Citrus test cases no matter what knowledge base you belong to.

4.1. Defining a test case

Put simply, a Citrus test case is nothing but a simple Spring XML configuration file. The Spring framework has become a state of the art development framework for enterprise Java applications. As you work with Citrus you will also learn how to use the Spring Ioc (Inversion of control) container and the concepts of dependency injection. So let us have a look at the pure Spring XML configuration syntax first. You are free to write fully compatible test cases for the Citrus framework just using this syntax.

Spring bean definition syntax

```
<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.xsd">

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

Citrus can execute these Spring bean definitions as normal test cases - no problem, but the pure Spring XML syntax is very verbose and probably not the best way to describe a test case in Citrus. In particular you have to know a lot of Citrus internals such as Java class and property names. In addition to that as test scenarios get more complex and the test cases grow in size we need a more effective way of writing tests. Therefore Citrus provides a custom XML schema definition for writing test cases which is much more adequate for our testing purpose.

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

XML DSL

```
<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.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.

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 test name has to follow the common Java naming conventions and rules for Java classes. This means names must not contain any whitespace characters but characters like '-', '.', '_' are supported. For example, `TestFeature_1` is valid but `TestFeature 1` is not.

Before we go into more details on the attributes and actions that take place within a test case we just have a look at the Java capabilities. Citrus works with Java and uses the well known JUnit and TestNG framework benefits that you may be used to as a tester. Therefore each Citrus test comes with a Java class. When you use the XML test syntax mentioned above you may not change the Java class at all. You do not need to write Java code.

On the other hand many users may prefer to write Java code instead of the verbose XML syntax. Therefore you have another possibility for writing Citrus tests in pure Java. Let us have a look at a first Java test example:

Java DSL

```
import org.testng.ITestContext;
import org.testng.annotations.Test;
import com.consol.citrus.dsl.TestNGCitrusTestBuilder;

public class MyFirstTest extends TestNGCitrusTestBuilder {
    protected void configure() {
        description("First example showing the basic test case definition elements!");

        variable("text", "Hello Test Framework");

        echo(${test});
    }

    @Test
    public void doTest(ITestContext testContext) {
        executeTest(testContext);
    }
}
```

Citrus provides a base builder Java class that provides all capabilities for you in form of builder pattern methods. You are forced to implement the configure() method which explains the test logic. The test actions defined inside the configure() method will be called later on during test runtime sequence. For completion we need a TestNG annotated method that executes the test case for us.

This is the basic test Java class pattern used in Citrus. You as a tester with development background can easily extend this pattern for customized logic. Again if you are coming without coding experience do not worry this Java code is optional. You can do exactly the same with the XML syntax only.

Note

Please choose whatever code language type you want (Java, XML, Spring bean syntax) in order to write Citrus test cases. Developers may choose Java, testers without coding experience may run best with the XML syntax. We are constantly working on even more test writing language support such as Groovy, Scala, Xtext, and so on. In general you can mix the different language types just as you like within your Citrus project which gives you the best of flexibility.

Now let us continue with the basic test case attributes and properties that you will come accross while writing new tests. A typical test case defines these basic attributes that we are going to discuss in detail in the next sections.

4.1.1. Description

In the first examples you may have noticed that a tester can give a detailed test description. The test

case description clarifies the testing purpose and perspectives. The description should give 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 as well as special information to understand the test scenario. You can use free text in your test description no limit to the number of characters. But be aware of the XML validation rules of well formed XML when using the XML test syntax (e.g. special character escaping, use of CDATA sections may be required)

4.1.2. Variables

XML DSL

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

Java DSL

```
variable("text", "Hello Test Framework");
variable("customerId", "123456789");
```

The concept of test variables is essential when writing complex tests with lots of identifiers and semantical data. Test variables are valid for the whole test case. You can reference them several times using a common variable 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 and identifiers 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 for other test scenarios.

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).

You can also use a script to create variable values. This is extremely handy when you have very complex variable values. Just code a small Groovy script for instance in order to define the variable value. A small sample should give you the idea how that works:

```
<variables>
    <variable name="avg">
        <value>
            <script type="groovy">
                <![CDATA[
                    a = 4
                    b = 6
                    return (a + b) / 2
                ]]>
            </script>
        </value>
    </variable>
    <variable name="sum">
        <value>
```

```
            <script type="groovy">
                <![CDATA[
                    5 + 5
                ]]>
            </script>
        </value>
    </variable>
</variables>
```

We use the script code right inside the variable value definition. The value of the variable is the result of the last operation performed within the script. For longer script code the use of `<![CDATA[ ]]>` sections is recommended.

Citrus uses the javax ScriptEngine mechanism in order to evaluate the script code. By default Groovy is supported in any Citrus project. So you can add additional ScriptEngine implementations to your project and support other script types, too.

4.1.3. 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. As these variables are global we need to add those to the basic Spring application context file. The following example demonstrates how 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 make 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 global test variables.

Note

You can use variables and functions in the external property files, too. It is possible to use previously defined global variables in the values, like in this example:

```
user=Citrus
greeting=Hello ${user}!
date=citrus:currentDate('yyyy-MM-dd')
```

4.1.4. Actions

Now we get close to the main part of writing an integration test. A Citrus test case defines a sequence of actions that will take place during the test. Actions by default are executed sequentially in the same order as they are defined in the test case definition.

XML DSL

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

All actions have individual names and properties that define the respective 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 11, *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 through the action chain. In case one single action fails by reason the whole test case is red and declared not successful.

4.1.5. Finally cleanup section

Java developers might be familiar with the concept of doing something in the finally code section. The `finally` section contains a list of test actions that will be executed guaranteed 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 14, *Finally section*. However here is the basic syntax inside a test.

XML DSL

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

Java DSL

```
protected void configure() {
    echo("Hello Test Framework");

    doFinally(
        echo("Do finally - regardless of any error before");
    );
}
```

4.2. Meta information

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

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    description("This is a Test");
    author("Christoph");
    status(Status.FINAL);

    echo("Hello Citrus!");
}
```

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 with the 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 it is not finished, someone may disable a test temporarily to avoid causing failures during a test run. Using these different statuses one can easily set up test plans and review the progress of test coverage by comparing the number of DRAFT tests to those in the FINAL state.

Chapter 5. Test actors

The concept of test actors came to our mind when reusing Citrus test cases in end-to-end test scenarios. Usually Citrus simulates all interface partners within a test case which is great for continuous integration testing. In end-to-end integration test scenarios some of our interface partners may be real and alive. Some other interface partners still require Citrus simulation logic.

It would be great if we could reuse the Citrus integration tests in this test setup as we have the complete test flow of messages available in the Citrus tests. We only have to remove the simulated send/receive actions for those real interface partner applications which are available in our end-to-end test setup.

With test actors we have the opportunity to link test actions, in particular send/receive message actions, to a test actor. The test actor can be disabled in configuration very easy and following from that all linked send/receive actions are disabled, too. One Citrus test case is runnable with different test setup scenarios where different partner applications on the one hand are available as real life applications and on the other hand my require simulation.

5.1. Define test actors

First thing to do is to define one or more test actors in Citrus configuration. A test actor represents a participating party (e.g. interface partner, backend application). We write the test actors into the central citrus-context.xml. We can use a special Citrus Spring XML schema so definitions are quite easy:

```
<citrus:actor id="travelagency" name="TRAVEL_AGENCY"/>
<citrus:actor id="royalairline" name="ROYAL_AIRLINE"/>
<citrus:actor id="smartariline" name="SMART_AIRLINE"/>
```

The listing above defines three test actors participating in our test scenario. A travel agency application which is simulated by Citrus as a calling client, the smart airline application and a royal airline application. Now we have the test actors defined we can link those to message sender/receiver instances and/or test actions within our test case.

5.2. Link test actors

We need to link the test actors to message send and receive actions in our test cases. We can do this in two different ways. First we can set a test actor reference on a message sender and message receiver.

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

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

Now all test actions that are using these message receiver and message sender instances are linked to the test actor. In addition to that you can also explicitly link test actions to test actors in a test.

```
<receive with="royalAirlineBookingRequestReceiver" actor="royalairline">
    <message>
```

```
        [...]
    </message>
</receive>

<send with="royalAirlineBookingResponseSender" actor="royalairline">
    <message>
        [...]
    </message>
</send>
```

This explicitly links test actors to test actions so you can decide which link should be set without having to rely on the message receiver and sender configuration.

5.3. Disable test actors

Usually both airline applications are simulated in our integration tests. But this time we want to change this by introducing a royal airline application which is online as a real application instance. So we need to skip all simulated message interactions for the royal airline application in our Citrus tests. This is easy as we have linked all send/receive actions to one of our test actors. So wen can disable the royal airline test actor in our configuration:

```
<citrus:actor id="royalairline" name="ROYAL_AIRLINE" disabled="true"/>
```

Any test action linked to this test actor is now skipped. As we introduced a real royal airline application in our test scenario the requests get answered and the test should be successful within this end-to-end test scenario. The travel agency and the smart airline still get simulated by Citrus. This is a perfect way of reusing integration tests in different test scenarios where you enable and disable simulated participating parties in Citrus.

Important

Server ports may be of special interest when dealing with different test scenarios. You may have to also disable a Citrus embedded Jetty server instance in order to avoid port binding conflicts and you may have to wire endpoint URIs accordingly before executing a test. The real life application may not use the same port and ip as the Citrus embedded servers for simulation.

Chapter 6. Running tests

Citrus test cases are nothing but Java classes that get executed within a Java runtime environment. Each Citrus test therefore relates to a Java class representing a JUnit or TestNG unit test. As optional add on a Citrus test can have a XML test declaration file. This is for those of you that do not want to code Java. In this case the XML part holds all actions to tell Citrus what should happen in the test case. The Java part will then just be responsible for test execution and is not likely to be changed at all. In the following sections we concentrate on the Java part and the test execution mechanism.

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. For example: if you create a new test named *MyFirstCitrusTest* you will find these two files as a result:

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

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

Note

If you prefer to just write Java code you can throw away the XML part immediately and continue working with the Java part only. In case you are familiar with writing Java code you may just skip the test template generation via Maven or ANT and preferably just create new Citrus Java test classes on your own.

With the creation of this test we have already made a very important decision. During creation, 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 the JUnit and TestNG Java APIs offer various ways of execution and both frameworks are widely supported by other tools (e.g. continuous build, build lifecycle, development IDE).

Users might already know one of these frameworks and the chances are good that they are familiar with at least one of them. Everything you can do with JUnit and TestNG test cases you can do with Citrus tests also. 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. Jenkins, Bamboo or Hudson). Generate test execution reports and test coverage reports with Sonar, Cobertura and so on. The possibilities with JUnit and TestNG are amazing.

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

6.1. 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 find complete reference documentation.

6.2. 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

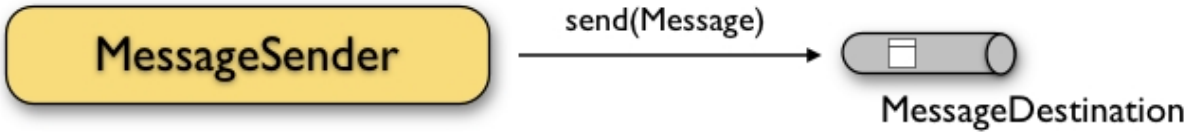
So now we know both TestNG and JUnit support in Citrus. Which framework should someone choose? To be honest, there is no easy answer to this question. The basic features are equivalent, but TestNG offers better possibilities for designing more complex test setup with test groups and tasks before and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

Chapter 7. Messaging

In one of the previous chapters we have discussed the basic test case structure as we introduced *variables* and *test actions*. The <actions> section contains a list of test actions that take place during the test case. Each test action is executed in sequential order by default. Citrus offers several built-in test actions that the user can choose from to construct a complex testing workflow without having to code everything from scratch. In particular Citrus aims to provide all the test actions that you need as predefined components ready for you to use. The goal is to minimize the coding effort for you so you can concentrate on the test logic itself. All 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, a FTP folder destination and many more. The MessageSender component just takes a previously defined message definition (header and payload) and sends it to the message destination.

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

In the next sections you will learn how a test case uses those MessageSender and MessageReceiver components for publishing and consuming messages.

7.1. Sending messages

The <send> action publishes messages to a destination. The message transport which is used under the hood does not matter to the test case - for now. The test case simply defines the message contents and uses a predefined message sender components to actually publish the constructed message to a destination. 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 to use is not specified inside the test case but in the message sender definition. 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, connection timeouts and so on. The transport internals underneath a sending test 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 uses XML as the default data format for message payload data. But Citrus is not limited to XML message format though; you can always define other message data formats such as JSON, plain text, CSV. As XML is still a very popular message format in enterprise applications and message-based solution architectures we have this as a default format. 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.

XML DSL

```
<testcase name="SendMessageTest">
    <description>Basic 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 recap 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 lives 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 any 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.

If you do not like the XML language you can also use pure Java code to define the same test. In Java aou would also make use of the message sender pattern. The same test as shown above in Java DSL looks like this:

Java DSL

```
import org.testng.ITestContext;
import org.testng.annotations.Test;
import com.consol.citrus.dsl.TestNGCitrusTestBuilder;

public class SendMessageTest extends TestNGCitrusTestBuilder {
    protected void configure() {
        description("Basic send message example");

        variable("requestTag", "Rx123456789");
        variable("correlationId", "Cx1x123456789");

        send("getCustomerRequestMessageSender")
            .payload("<RequestMessage>" +
                        "<MessageHeader>" +
                            "<CorrelationId>${correlationId}</CorrelationId>" +
                            "<Timestamp>2001-12-17T09:30:47.0Z</Timestamp>" +
                            "<RequestTag>${requestTag}</RequestTag>" +
                            "<VersionId>2</VersionId>" +
```

```
                        "</MessageHeader>" +
                        "<MessageBody>" +
                            "<Customer>" +
                                "<Id>1</Id>" +
                            "</Customer>" +
                        "</MessageBody>" +
                    "</RequestMessage>")
            .header("Operation", "GetCustomer")
            .header("RequestTag", "${requestTag}");
    }

    @Test
    public void doTest(ITestContext testContext) {
        executeTest(testContext);
    }
}
```

The same message sender is referenced within the sending message action. The payload is constructed as plain Java character sequence which is a bit verbose. We will se later on how we can improve this. For now it is important to understand the combination of send test action and a message sender.

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 message sender pattern is clear we can concentrate on how to specify the message content to be sent. There are several possibilities for you to define message content in Citrus:

- *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 overwrite some message values explicitly. 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 part of our duty of defining proper messages, too. So Citrus uses name-value pairs like "Operation" and "RequestTag" in the example above to set message header entries. Depending on what message sender is used and which message transport underneath the header values will be shipped in different ways. In JMS the headers go to the header section of the message, in Http we set mime headers accordingly, in SOAP we can access the SOAP header elements and so on. Citrus aims to do the hard work for you. So Citrus knows how to set headers on different message transports.

Note

In case you don't like to define the message payload in a simple CDATA section, you also have the possibility to use explicit payload setting with XML schema validation. In this case you need to use the `<payload>` instead of the `<data>` element. With this alternative you can validate the payload with XML schema during designtime. With XML editors you also have XSD auto completion when constructing your payload. See following example for usage explanation:

```
<testcase name="sendMessageTest">
    <actions>
        <send with="helloRequestSender">
            <message>
                <payload>
                    <hlo:HelloRequest xmlns:hlo="http://www.consol.de/schemas/samples/sayHello.xsd">
                        <hlo:MessageId>${messageId}</hlo:MessageId>
                        <hlo:CorrelationId>${correlationId}</hlo:CorrelationId>
                        <hlo:User>${user}</hlo:User>
                        <hlo:Text>Hello TestFramework</hlo:Text>
                    </hlo:HelloRequest>
                </payload>
            </message>
            ...
        </send>
    </actions>
</testcase>
```

This alternative works for receiving message action, too. We will see this in the next section when dealing with message receipt within Citrus.

This is basically how you send messages in Citrus. The test case is responsible for constructing the message content while the predefined message senders hold transport specific settings. All test cases reused the predefined message sender components to publish the messages to a destination. The variable support in message payload and message header enables you to add dynamic values.

7.2. 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.

XML DSL

```
<receive with="getCustomerResponseReceiver">
    <message>
        <data>
            <![CDATA[
            <ResponseMessage>
                <MessageHeader>
                    <CorrelationId>${correlationId}</CorrelationId>
                    <Timestamp>2001-12-17T09:30:47.0Z</Timestamp>
                    <RequestTag>_</RequestTag>
                    <VersionId>2</VersionId>
                </MessageHeader>
                <MessageBody>
                    <Customer>
                        <Id>1</Id>
                    </Customer>
                </MessageBody>
            </ResponseMessage>
            ]]>
        </data>
        <element path="//MessageHeader/RequestTag" value="${requestTag}"/>
    </message>
    <header>
        <element name="Operation" value="GetCustomer"/>
        <element name="RequestTag" value="${requestTag}"/>
    </header>
</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 component does know this information.

Before go into detail on validating the received message we have a quick look at the Java DSL variant on the receive action. As you are familiar with Java and the code completion capabilities of your IDE you will not have problems to code the test action.

Java DSL

```
protected void configure() {
    receive("getCustomerResponseReceiver")
        .payload("<ResponseMessage>" +
                    "<MessageHeader>" +
                        "<CorrelationId>${correlationId}</CorrelationId>" +
                        "<Timestamp>2001-12-17T09:30:47.0Z</Timestamp>" +
                        "<RequestTag>${requestTag}</RequestTag>" +
                        "<VersionId>2</VersionId>" +
                    "</MessageHeader>" +
                    "<MessageBody>" +
                        "<Customer>" +
                            "<Id>1</Id>" +
                        "</Customer>" +
                    "</MessageBody>" +
                "</ResponseMessage>")
        .header("Operation", "GetCustomer")
        .header("RequestTag", "${requestTag}");
}
```

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. At this time the test case waits for the message to arrive. Of course you can specify message timeouts so the receiver will only wait a given amount of time. Following from that a timeout exception fails the test in case the message does not arrive in time. Citrus defines default timeout settings for all message receiving tasks.

In best case the message has arrived in time and the content can be validated as next step. This

validation can be done in various ways. On the one hand you can specify a whole XML message that you expect as control template. In this case the received XML structure will be compared to the expected XML message content 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 the message payload Citrus will also perform validation on the received message header values. Test variable usage is supported as usual during the whole validation process for payload and header checks.

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 individual validation steps are performed. Let us have a closer look at the validation options and features step by step.

7.2.1. 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 XML 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.

The Java DSL has a significant disadvantage. Java does not support multiline character sequence values as Strings. Thisb is why you have to use verbose String concatenation when constructing XML message payload content. In addition to that reserved characters like quotes must be escaped and line breaks must be explicitly added. All these impediments let me suggest to use external file resources in Java DSL when dealing with large complex XML data.

Java DSL

```
protected void configure() {
    receive("getCustomerResponseReceiver")
        .payload(new ClassPathResource("com/consol/citrus/message/data/GetCustomerResponse.xml"))
```

```
        .header("Operation", "GetCustomer")
        .header("RequestTag", "${requestTag}");
}
```

7.2.2. 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:

XML DSL

```
<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 listing 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 equal to each other. With respect to the complexity of XML namespaces and XPath you may find the inline variable expression more comfortable to use. Anyway feel free to choose the 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.

Tip

Validation matchers put validation mechanisms to a new level offering dynamic assertion statements for validation. Have a look at the possibilities with assertion statements in Chapter 21, *Validation matchers*

7.2.3. 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.

XML DSL

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

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

XML DSL

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

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

When using Java as the test case DSL the *@ignore@* placeholder as well as XPath expression ignore can be used seamlessly. Here is an examnple if you need clarification:

Java DSL

```
protected void configure() {
```

```
    receive("getCustomerResponseReceiver")
        .payload(new ClassPathResource("com/consol/citrus/message/data/GetCustomerResponse.xml"))
        .header("Operation", "GetCustomer")
        .header("RequestTag", "${requestTag}")
        .ignore("//ResponseMessage/MessageHeader/Timestamp");
}
```

7.2.4. 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 appropriate in this case. You would rather want to use explicit element validation.

XML DSL

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

Java DSL

```
protected void configure() {
    receive("getCustomerResponseReceiver")
        .validate("//MessageHeader/RequestTag", "${requestTag}")
        .validate("//CorrelationId", "${correlationId}")
        .validate("//MessageBody/Number", "123456789")
        .header("Operation", "GetCustomer")
        .header("RequestTag", "${requestTag}");
}
```

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

Note

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

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.

7.2.5. Validate the message header

Now that we have validated the message payload in various ways we are now interested in validating the message header. This is simple as you have to define the header name and the control value that you expect. Just add the following header validation to your receiving action.

XML DSL

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

Java DSL

```
protected void configure() {
    receive("getCustomerResponseReceiver")
        .header("Operation", "GetCustomer")
        .header("RequestTag", "${requestTag}");
}
```

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 given control value. If a header entry is not found by its name or the value does not fit accordingly Citrus will raise validation 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 specifica.

7.2.6. Saving message content to variables

Imagine you receive a message in your test holding some generated message identifier. You have no chance to predict the identifier value because it was generated at runtime by a foreign application. You can ignore the value in order to not break your validation, but in many cases you might need to return this identifier in the respective response message. So Citrus needs to offer a way to save dynamic message content that is not known to the tester for reuse in later test steps. The solution is simple and very powerful. We can extract dynamic values from received messages and save those to test variables. Add this code to your message receiving action.

XML DSL

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

Java DSL

```
protected void configure() {
    receive("getCustomerResponseReceiver")
        .extractFromHeader("Operation", "operation")
        .extractFromPayload("//MessageBody/Customer/Id", "customerId");

    echo("Extracted operation from header is: ${operation}");
    echo("Extracted customerId from header is: ${customerId}");
}
```

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 variables as target. The extraction will automatically create a new variable in case it does not exist. The time the variable was created from message extraction 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.

7.2.7. 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 a message destination (e.g. a JMS message queue) contains 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 information 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:

XML DSL

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

Java DSL

```
protected void configure() {
    receive("getCustomerResponseReceiver")
        .selector("correlationId='Lx1x123456789' AND operation='getOrders'");
}
```

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 on the message destination

are ignored. The selector elements are automatically associated to each other using the logical AND operator. This means that the message selector string would look like this: *correlationId = 'Lx1x123456789' AND operation = 'getOrders'*.

Instead of using several elements in the selector you can also define a selector string directly which gives you more power in constructing the selection logic yourself. This way you can use *AND* logical operators yourself.

```
<selector>
    <value>
        correlationId = 'Cx1x123456789' AND messageId = '987654321'
    </value>
</selector>
```

Important

In case you want to run tests in parallel message selectors become essential in your test cases. The different tests running at the same time will steal messages from each other when you lack of message selection mechanisms.

Important

Previously only JMS message destinations offered support for message selectors! With Citrus version 1.2 we introduced message selector support for Spring Integration message channels, too (see Section 19.5, “Message selectors on channels”).

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.

7.3. Validation callback

The Java DSL offers some additional validation tricks and functionalities when dealing with messages that are sent and received over Citrus. One of them is the validation callback functionality. With this piece you can marshal received message payloads and code validation steps on Java objects.

Java DSL

```
protected void configure() {
    receive(bookResponseMessageReceiver)
      .validationCallback(new MarshallingValidationCallback<AddBookResponseMessage>() {
          @Override
          public void validate(AddBookResponseMessage response, MessageHeaders headers) {
              Assert.isTrue(response.isSuccess());
          }
      });
}
```

By default the validation callback needs a XML unmarshaller implementation for transforming the XML payload to a Java object. Citrus will automatically search for a Unmarshaller bean in your Spring application context if nothing specific is set. Of course you can also set the unmarshaller explicitly.

Java DSL

```
@Autowired
private Unmarshaller unmarshaller;

protected void configure() {
    receive(bookResponseMessageReceiver)
        .validationCallback(new MarshallingValidationCallback<AddBookResponseMessage>(unmarshaller) {
            @Override
            public void validate(AddBookResponseMessage response, MessageHeaders headers) {
                Assert.isTrue(response.isSuccess());
            }
        });
}
```

Obviously working on marshalled Java objects is much more comfortable than using the XML String concatenation. This is why you can also use this feature when sending messages.

Java DSL

```
@Autowired
private Marshaller marshaller;

protected void configure() {
    send(bookRequestMessageSender)
            .payload(createAddBookRequestMessage("978-citrus:randomNumber(10)"), marshaller)
            .header("citrus_soap_action", "addBook");
}

private AddBookRequestMessage createAddBookRequestMessage(String isbn) {
    AddBookRequestMessage requestMessage = new AddBookRequestMessage();
    Book book = new Book();
    book.setAuthor("Foo");
    book.setTitle("FooTitle");
    book.setIsbn(isbn);
    book.setYear(2008);
    book.setRegistrationDate(Calendar.getInstance());
    requestMessage.setBook(book);
    return requestMessage;
}
```

7.4. Groovy utils for send and receive

Groovy offers some nice and easy to use features for working with XML structures. You can use those in Citrus within the send and receive test actions in order to construct message payloads via Groovy scripts.

7.4.1. Groovy MarkupBuilder

With the Groovy MarkupBuilder the tester may build XML message payloads in a simple way, without having to write the typical XML tag overhead. For example we use the code listing from Section 7.1, “Sending messages”, but now we use a Groovy script instead of plain CDATA XML section. The Groovy MarkupBuilder generates the XML message payload with exactly the same result:

XML DSL

```
<send with="getCustomerRequestMessageSender">
    <message>
        <builder type="groovy">
            markupBuilder.RequestMessage{
                MessageHeader{
                    CorrelationId('${correlationId}')
                    Timestamp('2001-12-17T09:30:47.0Z')
                    RequestTag('_')
```

```
                    VersionId('2')
                }
                MessageBody{
                    Customer{
                        Id('1')
                    }
                }
            }
        </builder>
        <element path="/MessageHeader/RequestTag"
                    value="${requestTag}"/>
    </message>
    <header>
        <element name="Operation" value="GetCustomer"/>
        <element name="RequestTag" value="${requestTag}"/>
    </header>
</send>
```

The MarkupBuilder code is directly written into the well-known message element. As you can see from the example above, the MarkupBuilder syntax is very easy and follows the simple rule: *markupBuilder.ROOT-ELEMENT{ CHILD-ELEMENTS }*. However the tester has to follwow some simple rules and naming conventions when using the Citrus MarkupBuilder extension:

- The MarkupBuilder is accessed within the script over an object named `markupBuilder`. The name of the custom root element follows with all its child elements.
- Child elements may be defined within curly brackets after the root-element (the same applies for further nested child elements)
- Attributes and element values are defined within round brackets, after the element name
- Attribute and element values have to stand within apostrophes (e.g. attribute-name: 'attribute-value')

The Groovy MarkupBuilder script may also be used within receive actions as shown in the following listing:

XML DSL

```
<send with="helloRequestSender">
    <message>
        <builder type="groovy" file="classpath:com/consol/citrus/groovy/helloRequest.groovy"/>
    </message>
</send>

<receive with="helloResponseReceiver" timeout="5000">
    <message>
        <builder type="groovy">
            markupBuilder.HelloResponse(xmlns: 'http://www.consol.de/schemas/samples/sayHello.xsd'){
                MessageId('${messageId}')
                CorrelationId('${correlationId}')
                User('HelloService')
                Text('Hello ${user}')
            }
        </builder>
    </message>
</receive>
```

As you can see it is also possible to define the script as external file resource. In addition to that namespace support is given as normal attribute definition within the round brackets after the element name.

The MarkupBuilder implementation in Groovy opens great possibilities in defining message payloads in Citrus test cases. We do not need to write XML tag overhead and we can construct complex

message payloads with Groovy logic like iterations and conditions. For detailed MarkupBuilder descriptions please see the official Groovy documentation.

7.4.2. Groovy XmlSlurper

With the Groovy XmlSlurper you can easily validate XML message payloads without having to deal directly with XML. People who do not want to deal with XPath may also like this validation alternative. The tester directly navigates through the message elements and uses simple code assertions in order to control the message content. Here is an example how to validate messages with Groovy script:

XML DSL

```
<receive with="helloResponseReceiver" timeout="5000">
    <message>
        <validate>
            <script type="groovy">
              assert root.children().size() == 4
              assert root.MessageId.text() == '${messageId}'
              assert root.CorrelationId.text() == '${correlationId}'
              assert root.User.text() == 'HelloService'
              assert root.Text.text() == 'Hello ' + context.getVariable("user")
            </script>
        </validate>
    </message>
    <header>
        <element name="Operation" value="sayHello"/>
        <element name="CorrelationId" value="${correlationId}"/>
    </header>
</receive>
```

Java DSL

```
protected void configure() {
    receive("helloResponseReceiver")
        .validateScript("assert root.MessageId.text() == '${messageId}';" +
                        "assert root.CorrelationId.text() == '${correlationId}';")
        .header("Operation, "sayHello")
        .header("CorrelationId", "${correlationId}")
        .timeout(5000L);
}
```

The Groovy XmlSlurper validation script goes right into the message-tag instead of a XML control template or XPath validation. The Groovy script supports Java `assert` statements for message element validation. The element navigation starts with the root element reffered to as `root`. Based on this root element you can also access child elements and attributes. Just use the element names seperated by a simple dot. Very easy! If you need the list of child elements use the `children()` function on any element. With the `text()` function you get access to the element's text-value. The `size()` is very useful for validating the number of child elements which completes the basic validation statements.

As you can see from the example, we may use test variables within the validation script, too. Also not very surprising you may use external file resources as validaton scripts. The syntax looks like follows:

XML DSL

```
<receive with="helloResponseReceiver" timeout="5000">
    <message>
        <validate>
          <script type="groovy" file="classpath:validationScript.groovy"/>
        </validate>
    </message>
    <header>
```

```
        <element name="Operation" value="sayHello"/>
        <element name="CorrelationId" value="${correlationId}"/>
    </header>
</receive>
```

Java DSL

```
protected void configure() {
    receive("helloResponseReceiver")
        .validateScript(new FileSystemResource("validationScript.groovy"))
        .header("Operation, "sayHello")
        .header("CorrelationId", "${correlationId}")
        .timeout(5000L);
}
```

Note

You can use the Groovy validation script in combination with other validation types like XML tree comparison and XPath validation.

Tip

For further information on the Groovy XmlSlurper please see the official Groovy website and documentation

Chapter 8. Message validation

In the previous sections we have seen the powerful Citrus XML message validation capabilities. The tester is able to define expected message behaviour on header and payload level. Later in this documentation we will also deal with XPath and XML schema validation capabilities. Now we deal with other message formats than XML.

Message formats such as JSON have become very poular in the Java enterprise world. More and more interfaces are built with JSON message format. Just think of RESTful WebServices and JavaScript using JSON as the message format to go for. Citrus is able to expect and validate JSON messages and other formats such as Html, CSV or plaintext.

8.1. JSON message validation

By default Citrus will use XML message formats when sending and receiving messages. This also reflects to the message validation logic Citrus uses for incoming messages.

However other message formats are supported, too. And it is quite easy to tell Citrus not to use XML but JSON for instance. First of all we have intriduce a JSON message validator implementation to the citrus-context.xml

```
<bean id="jsonTextMessageValidator" class="com.consol.citrus.validation.json.JsonTextMessageValidator"/>
```

We just add the bean in the citrus-context.xml and Citrus is now aware of JSON message format validation.

Citrus provides several default message validator implementations for JOSN message format:

- com.consol.citrus.validation.json.JsonTextMessageValidator: Basic JSON message validator implementation compares JSON objects (expected and received). The order of JSON entries can differ as specified in JSON protocol. Tester defines an expected control JSON object with test variables and ignored entries. JSONArray as well as nested JSONObjects are supported, too.
- com.consol.citrus.validation.script.GroovyJsonMessageValidator: Extended groovy message validator provides specific JSON slurper support. With JSON slurper the tester can validate the JSON message payload with closures for instance.

Important

It is very important to set a proper message validator in the citrus-context. If no proper validator is present for a message format Citrus might fall back to some other validator implementation (e.g. plaintext) or skip the message validation completely.

What is now to do? We have to tell the test case receiving action that we expect a message format other than XML.

```
<receive with="httpMessageReceiver">
    <message type="json">
        <data>
          {
            "type" : "read",
            "mbean" : "java.lang:type=Memory",
```

```
            "attribute" : "HeapMemoryUsage",
            "path" : "@equalsIgnoreCase('USED')@"
            "value" : "${heapUsage}"
            "timestamp" : "@ignore@"
          }
        </data>
    </message>
</receive>
```

The message receiving action in our test case specifies a message format type *type="json"*. This tells Citrus to look for some message validator implementation capable of validating JSON messages. As we have added the proper message validator to the citrus-context.xml Citrus will pick the right validator and JSON message validation is performed on this message. As you can see you we can use the usual test variables and the ignore element syntax here, too. Citrus is able to handle different JSON element orders when comparing received and expected JSON object.

We can also use JSON arrays and nested objects. Citrus is very powerful in comparing JSON objects.

Important

When using several message validators for one single message format in the citrus-context.xml you may have to pick a implementation for each receive action in your test cases. Otherwise Citrus does not know which of the applying message validators to choose.

Lets have a look at the Groovy JSON message validator example:

```
<receive with="httpMessageReceiver">
    <message type="json" validator="groovyJsonMessageValidator">
        <validate>
            <script type="groovy">
                <![CDATA[
                  assert json.type == 'read'
                  assert json.mbean == 'java.lang:type=Memory'
                  assert json.attribute == 'HeapMemoryUsage'
                  assert json.value == '${heapUsage}'
                ]]>
            </script>
        </validate>
    </message>
</receive>
```

In this example we use the *groovyJsonMessageValidator* explicitly which was added to the citrus-context.xml as bean before. The Groovy JSON slurper is automatically passed to the validation script. This way you can access the JSON object elements in your code doing some assertions.

By the way sending JSON messages in Citrus is also very easy. Just use JSON message payloads in your sending message action.

```
<send with="httpMessageSender">
    <message>
        <data>
          {
            "type" : "read",
            "mbean" : "java.lang:type=Memory",
            "attribute" : "HeapMemoryUsage",
            "path" : "used"
          }
        </data>
    </message>
</send>
```

8.2. XHTML message validation

When Citrus receives plain Html messages we likely want to use the powerful XML validation capabilities such as XML tree comparison or XPath support. Unfortunately Html messages do not ollow the XML well formed rules very strictly. This implies that XML message validation will fail because of non well formed Html code.

XHTML closes this gap by automatically fixing the most common Html XML incompatible rule violations such as missing end tags (e.g.
).

Let's try this with a simple example. Again we have to add a proper XHTML message validator implementation to the citrus-context first.

```
<bean id="xhtmlMessageValidator" class="com.consol.citrus.validation.xhtml.XhtmlMessageValidator"/>
```

Now we can tell the test case receiving action that we want to use the XHTML message validation in our test case.

```
<receive with="httpMessageReceiver">
    <message type="xhtml">
        <data>
          <![CDATA[
            <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" "org/w3c/xhtml/xhtml1-strict.dtd">
            <html xmlns="http://www.w3.org/1999/xhtml">
            <head>
              <title>Citrus Hello World</title>
            </head>
            <body>
              <h1>Hello World!</h1>
              <br/>
              <p>This is a test!</p>
            </body>
          ]]>
        </data>
    </message>
</receive>
```

The message receiving action in our test case has to specify a message format type *type="xhtml"*. As you can see the Html message payload get XHTML specific DOCTYPE processing instruction. The *xhtml1-strict.dtd* is mandatory in the XHTML message validation. For better convenience all XHTML dtd files are packaged within Citrus so you can use this as a relative path.

The incoming Html message is automatically converted into proper XHTML code with well formed XML. So now the XHTML message validator can use the XML message validation mechanism of Citrus for comparing received and expected data. As usual you can use test variables, ignore element expressions and XPath expressions.

8.3. Plain text message validation

Plain text message validation is the easiest validation in Citrus that you can think of. This validation just performs an exact Java String match of received and expected message payloads.

First thing to do is to add a plain text message validator implementation to the citrus-context.xml

```
<bean id="plainTextMessageValidator" class="com.consol.citrus.validation.text.PlainTextMessageValidator"/>
```

In the test case receiving action we tell Citrus to use plain text message validation.

```
<receive with="httpMessageReceiver">
    <message type="plaintext">
        <data>Hello World!</data>
    </message>
</receive>
```

With the message format type *type="plaintext"* set Citrus performs String equals on the message payloads (received and expected). Only exact match will pass the test.

By the way sending plain text messages in Citrus is also very easy. Just use the plain text message payload data in your sending message action.

```
<send with="httpMessageSender">
    <message>
        <data>Hello World!</data>
    </message>
</send>
```

Of course test variables are supported in the plain text payloads. The variables are replace by the referenced values before sending or receiving the message.

Chapter 9. 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.

9.1. XSD schema 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 a mapping strategy which we will discuss later in this chapter. By default Citrus picks the right schema using the target namespace that is defined inside a schema definition. The target namespace of the schema definition has to 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 within your test. This is done automatically within the schema repository. All you need to do is defin all schema definition files inside the schema repository.

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="${correlationId}"/>
        <namespace prefix="ns2" value="http://citrus/default"/>
    </message>
    <header>
        <element name="Operation" value="GetCustomer"/>
        <element name="RequestTag" value="${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.

9.2. Schema mapping strategy

The schema repository in Citrus holds one to many schema definition files and dynamically picks up the right one according to the validated message payload. The repository needs to have some strategy for deciding which schema definition to choose. See the following schema mapping strategies and decide which of them is suitable for you.

9.2.1. Target Namespace Mapping Strategy

This is the default schema mapping strategy. Schema definitions usually define some target namespace which is valid for all elements and types inside the schema file. The target namespace is also used as root namespace in XML message payloads. According to this information Citrus can pick up the right schema definition file in the schema repository. You can set the schema mapping strategy as property in the configuration files:

```
<bean id="schemaRepository"
         class="com.consol.citrus.xml.XsdSchemaRepository">
    <property name="schemaMappingStrategy">
        <bean class="com.consol.citrus.xml.schema.TargetNamespaceSchemaMappingStrategy"/>
    </property>
    <property name="schemas">
        <list>
            <bean class="org.springframework.xml.xsd.SimpleXsdSchema">
                <property name="xsd"
                     value="classpath:citrus/samples/sayHello.xsd"/>
            </bean>
        </list>
    </property>
</bean>
```

The *sayHello.xsd* schema file defines a target namespace (http://consol.de/schemas/sayHello.xsd):

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
     xmlns="http://consol.de/schemas/sayHello.xsd"
     targetNamespace="http://consol.de/schemas/sayHello.xsd"
     elementFormDefault="qualified"
     attributeFormDefault="unqualified">

</xs:schema>
```

Incoming request messages should also have the target namespace set in the root element and this is how Citrus matches the right schema file in the repository.

```
<HelloRequest xmlns="http://consol.de/schemas/sayHello.xsd">
```

```
   <MessageId>123456789</MessageId>
   <CorrelationId>1000</CorrelationId>
   <User>Christoph</User>
   <Text>Hello Citrus</Text>
</HelloRequest>
```

9.2.2. Root QName Mapping Strategy

The next possibility for mapping incoming request messages to a schema definition is via the XML root element QName. Each XML message payload starts with a root element that usually declares the type of a XML message. According to this root element you can set up mappings in the schema repository.

```
<bean id="schemaRepository"
         class="com.consol.citrus.xml.XsdSchemaRepository">
    <property name="schemaMappingStrategy">
        <bean class="com.consol.citrus.xml.schema.RootQNameSchemaMappingStrategy">
            <property name="mappings">
                <map>
                    <entry key="HelloRequest" value="helloSchema"/>
                    <entry key="GoodbyeRequest" value="goodbyeSchema"/>
                </map>
            </property>
        </bean>
    </property>
    <property name="schemas">
        <list>
            >ref bean="helloSchema"/>
            >ref bean="goodbyeSchema"/>
        </list>
    </property>
</bean>

<bean id="helloSchema" class="org.springframework.xml.xsd.SimpleXsdSchema">
    <property name="xsd"
         value="classpath:citrus/samples/sayHello.xsd"/>
</bean>

<bean id="goodbyeSchema" class="org.springframework.xml.xsd.SimpleXsdSchema">
    <property name="xsd"
         value="classpath:citrus/samples/sayGoodbye.xsd"/>
</bean>
```

The programlisting above defines two root qname mappings - one for *HelloRequest* and one for *GoodbyeRequest* message types. An incoming message of type <HelloRequest> is then mapped to the respective schema and so on. With this dedicated mappings you are able to control which schema is used on a XML request, regardless of target namespace definitions.

9.2.3. Schema mapping strategy chain

Let's discuss the possibility to combine several schema mapping strategies in a logical chain. You can define more than one mapping strategy that are evaluated in sequence. The first strategy to find a proper schema definition file in the repository wins.

```
<bean id="schemaRepository"
         class="com.consol.citrus.xml.XsdSchemaRepository">
    <property name="schemaMappingStrategy">
        <bean class="com.consol.citrus.xml.schema.SchemaMappingStrategyChain">
            <property name="strategies">
                <list>
                    <bean class="com.consol.citrus.xml.schema.RootQNameSchemaMappingStrategy">
                        <property name="mappings">
                            <map>
                                <entry key="HelloRequest" value="helloSchema"/>
                            </map>
                        </property>
                    </bean>
```

```
                    <bean class="com.consol.citrus.xml.schema.TargetNamespaceSchemaMappingStrategy"/>
                </list>
            </property>
        </bean>
    </property>
    <property name="schemas">
        <list>
            >ref bean="helloSchema"/>
            >ref bean="goodbyeSchema"/>
        </list>
    </property>
</bean>
```

So the schema mapping chain uses both *RootQNameSchemaMappingStrategy* and *TargetNamespaceSchemaMappingStrategy* in combination. In case the first root qname strategy fails to find a proper mapping the next target namespace strategy comes in and tries to find a proper schema.

9.3. Schema definition overruling

Now it is time to talk about schema definition settings on test action level. We have learned before that Citrus tries to automatically find a matching schema definition in some schema repository. There comes a time where you as a tester just have to pick the right schema definition by yourself. You can overrule all schema mapping strategies in Citrus by directly setting the desired schema in your receiving message action.

```
<receive with="getCustomerRequestReceiver">
    <message schema="helloSchema">
        <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://citrus/default"/>
    </message>
</receive>

<bean id="helloSchema" class="org.springframework.xml.xsd.SimpleXsdSchema">
    <property name="xsd"
         value="classpath:citrus/samples/sayHello.xsd"/>
</bean>
```

In the example above the tester explicitly sets a schema definition in the receive action (schema="helloSchema"). The attribute value refers to named schema bean somewhere in the applciation context. This overrules all schema mapping strategies used in the central schema repository as the given schema is directly used for validation. This feature is helpful when dealing with different schema versions at the same time where the schema repository can not help you anymore.

Another possibility would be to set a custom schema repository at this point. This means you can have more than one schema repository in your Citrus project and you pick the right one by yourself in the receive action.

```
<receive with="getCustomerRequestReceiver">
    <message schema-repository="mySpecialSchemaRepository">
        <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://citrus/default"/>
    </message>
</receive>
```

The *schema-repository* attribute refers to a bean of type *com.consol.citrus.xml.XsdSchemaRepository* somewhere in the applicaiton context.

Important

In case you have several schema repositories in your project do always define a default repository (name="schemaRepository"). This helps Citrus to always find at least one repository to consult with.

9.4. 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 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://citrus/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://citrus/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://citrus/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://citrus/default) matches.

Tip

Instead of this namespace context on validation level you can also have a global namespace context which is valid in all test cases. We just add a bean in the basic citrus-context configuration which defines global namespace mappings.

```
<bean id="namespaceContextBuilder" class="com.consol.citrus.xml.namespace.NamespaceContextBuilder">
    <property name="namespaceMappings">
        <props>
            <prop key="def">http://www.consol.de/samples/sayHello</prop>
        </props>
    </property>
</bean>
```

Once defined the *def* namespace prefix is valid in all test cases and all XPath expressions. This enables you to free your test cases from namespace prefix bindings that might be broken with time. You can use these global namespace mappings wherever XPath expressions are valid inside a test case (validation, ignore, extract).

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://citrus/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. 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.

11.1. 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.

11.1.1. 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.

XML DSL

```
<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>
```

Java DSL

```
@Autowired
@Qualifier("myDataSource")
private DataSource dataSource;

protected void configure() {
    sql(dataSource)
            .statement("DELETE FROM CUSTOMERS")
            .statement("DELETE FROM ORDERS");

    sql(dataSource)
            .sqlResource("file:tests/unit/resources/script.sql");
}
```

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.

11.1.2. Verifying data from the database

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

XML DSL

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

Java DSL

```
@Autowired
@Qualifier("testDataSource")
private DataSource dataSource;

protected void configure() {
    query(dataSource)
            .statement("select NAME from CUSTOMERS where CUSTOMER_ID='${customerId}'")
            .statement("select COUNT(1) as overall_cnt from ERRORS")
            .statement("select ORDER_ID from ORDERS where DESCRIPTION LIKE 'Migrate%'")
            .statement("select DESCRIPTION from ORDERS where ORDER_ID = 2")
            .validate("ORDER_ID", "1")
            .validate("NAME", "Christoph")
            .validate("OVERALL_CNT", "${rowsCount}")
            .validate("DESCRIPTION", "NULL");
}
```

The action 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.

Important

The validation with `"<validate column="..." value="..."/>"` meets single row result sets as you specify a single column control value. In case you have multiple rows in a result set you rather need to validate the columns with multiple control values like this:

```
<validate column="someColumnName">
    <values>
        <value>Value in 1st row</value>
        <value>Value in 2nd row</value>
        <value>Value in 3rd row</value>
        <value>Value in x row</value>
    </values>
</validate>
```

Within Java you can pass a variable argument list to the validate method like this:

```
query(dataSource)
    .statement("select NAME from WEEKDAYS where NAME LIKE 'S%'")
    .validate("NAME", "Saturday", "Sunday")
```

Next example shows how to work with multiple row result sets and multiple values to expect within one column:

```
<sql datasource="testDataSource">
    <statement>select WEEKDAY as DAY, DESCRIPTION from WEEK</statement>
    <validate column="DAY">
        <values>
            <value>Monday</value>
            <value>Tuesday</value>
            <value>Wednesday</value>
            <value>Thursday</value>
            <value>Friday</value>
            <value>@ignore@</value>
            <value>@ignore@</value>
        </values>
    </validate>
```

```
      <validate column="DESCRIPTION">
          <values>
              <value>I hate Mondays!</value>
              <value>Tuesday is sports day</value>
              <value>The mid of the week</value>
              <value>Thursday we play chess</value>
              <value>Friday, the weekend is near!</value>
              <value>@ignore@</value>
              <value>@ignore@</value>
          </values>
      </validate>
  </sql>
```

For the validation of multiple rows the `<validate>` element is able to host a list of control values for a column. As you can see from the example above, you have to add a control value for each row in the result set. This also means that we have to take care of the total number of rows. Fortunately we can use the ignore placeholder, in order to skip the validation of a specific row in the result set. Functions and variables are supported as usual.

Important

It is important, that the control values are defined in the correct order, because they are compared one on one with the actual result set coming from database query. You may need to add "order by" SQL expressions to get the right order of rows returned. If any of the values fails in validation or the total number of rows is not equal, the whole action will fail with respective validation errors.

11.1.3. Groovy SQL result set validation

Groovy provides great support for accessing Java list objects and maps. As a Java SQL result set is nothing but a list of map representations, where each entry in the list defines a row in the result set and each map entry represents the columns and values. So with Groovy's list and map access we have great possibilities to validate a SQL result set - out of the box.

XML DSL

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

    <validate-script type="groovy">
        assert rows.size() == 2
        assert rows[0].ID == '1'
        assert rows[1].STATUS == 'in progress'
        assert rows[1] == [ORDERTYPE:'SampleOrder', STATUS:'in progress']
    </validate-script>
</sql>
```

Java DSL

```
query(dataSource)
    .statement("select ORDERTYPE, STATUS from ORDERS where ID='${orderId}'")
    .validateScript("assert rows.size == 2;" +
            "assert rows[0].ID == '1';" +
            "assert rows[0].STATUS == 'in progress';", "groovy");
```

As you can see Groovy provides fantastic access methods to the SQL result set. We can browse the result set with named column values and check the size of the result set. We are also able to search for an entry, iterate over the result set and have other helpful operations. For a detailed description of

the list and map handling in Groovy my advice for you is to have a look at the official Groovy documentation.

Note

In general other script languages do also support this kind of list and map access. For now we just have implemented the Groovy script support, but the framework is ready to work with all other great script languages out there, too (e.g. Scala, Clojure, Fantom, etc.). So if you prefer to work with another language join and help us implement those features.

11.1.4. 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:

XML DSL

```
<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>
```

Java DSL

```
query(dataSource)
    .statement("select STATUS from ORDERS where ID='${orderId}'")
    .extract("STATUS", "orderStatus");
```

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.

11.2. 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.

XML DSL

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

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

Java DSL

```
protected void configure() {
    sleep(3500L); // sleep 3.5 seconds

    sleep(); // sleep default time
}
```

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 unnecessary 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.

11.3. 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.

Besides instantiating a fully new object instance for a class how about reusing a bean instance available in Spring bean container. Simply use the *ref* attribute and refer to an existing bean in Spring application context.

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

<bean id="invocationDummy" class="com.consol.citrus.test.util.InvocationDummy"/>
```

The method is invoked on the Spring bean instance. This is very useful as you can inject other objects (e.g. via Autowiring) to the Spring bean instance before method invocation in test takes place. This enables you to execute any Java logic inside a test case.

11.4. 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:

XML DSL

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

Java DSL

```
@Autowired
@Qualifier("myMessageReceiver")
private MessageReceiver myMessageReceiver;

protected void configure() {
    expectTimeout(myMessageReceiver)
        .timeout(500);
}
```

The action offers two attributes:

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

Sometimes you may want to add some selector on the timeout receiving action. This way you can very selective check on a message to not be present on a message destination. This is possible with

defining a message selector on the test action as follows.

XML DSL

```
<expect-timeout message-receiver="myMessageReceiver" wait="500">
  <select>MessageId='123456789'<select/>
<expect-timeout/>
```

Java DSL

```
protected void configure() {
    expectTimeout(myMessageReceiver)
        .selector("MessageId = '123456789'")
        .timeout(500);
}
```

11.5. 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:

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    variable("date", "citrus:currentDate()");

    echo("Hello Test Framework");
    echo("Current date is: ${date}");
}
```

Result on the console:

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

11.6. 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:

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    stopTime();
    stopTime("time_line_id");
    sleep(3500L); // do something
    stopTime("time_line_id");
    sleep(5000L); // do something
    stopTime();
    stopTime("time_line_id");
}
```

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.

11.7. Create variables

As you know variables usually are defined at the beginning of the test case (Section 4.1.2, “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.

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    variable("myVariable", "12345");
    variable("id", "54321");

    echo("Current variable value: ${myVariable}");

    variable("myVariable", "${id}");
    variable("newVariable", "this is a test");

    echo("Current variable value: ${myVariable}");

    echo("New variable 'newVariable' has the value: ${newVariable}");
}
```

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

11.8. 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:

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    variable("myVariable", "12345");
    variable("nextVariable", "54321");

    traceVariables("myVariable", "nextVariable");
    traceVariables();
}
```

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
```

11.9. Transform

The `<transform>` action transforms XML fragments with XSLT in order to construct various XML representations. The transformation result is stored into a test variable for further usage. The property *xml-data* defines the XML source, that is going to be transformed, while *xslt-data* defines the XSLT transformation rules. The attribute *variable* specifies the target test variable which receives the transformation result. The tester might use the action to transform XML messages as shown in the next code example:

XML DSL

```
<testcase name="transformTest">
    <actions>
        <transform variable="result">
            <xml-data>
                <![CDATA[
                    <TestRequest>
                        <Message>Hello World!</Message>
                    </TestRequest>
                ]]>
            </xml-data>
            <xslt-data>
                <![CDATA[
                    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
                    <xsl:template match="/">
                        <html>
                            <body>
                                <h2>Test Request</h2>
                                <p>Message: <xsl:value-of select="TestRequest/Message"/></p>
                            </body>
                        </html>
                    </xsl:template>
                    </xsl:stylesheet>
                ]]>
            </xslt-data>
        </transform>
        <echo>
            <message>${result}</message>
        </echo>
    </actions>
</testcase>
```

The transformation above results to:

```
<html>
    <body>
        <h2>Test Request</h2>
        <p>Message: Hello World!</p>
```

```
      </body>
  </html>
```

In the example we used CDATA sections to define the transformation source as well as the XSL transformation rules. As usual you can also use external file resources here. The transform action with external file resources looks like follows:

```
  <transform variable="result">
      <xml-resource file="classpath:transform-source.xml"/>
      <xslt-resource file="classpath:transform.xslt"/>
  </transform>
```

The Java DSL alternative for transforming data via XSTL in Citrus looks like follows:

Java DSL

```
protected void configure() {
    transform()
        .source("<TestRequest>" +
                    "<Message>Hello World!</Message>" +
                "</TestRequest>")
        .xslt("<xsl:stylesheet version=\"1.0\" xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\">\n" +
                    "<xsl:template match=\"/\">\n" +
                    "<html>\n" +
                        "<body>\n" +
                            "<h2>Test Request</h2>\n" +
                            "<p>Message: <xsl:value-of select=\"TestRequest/Message\"/></p>\n" +
                        "</body>\n" +
                    "</html>\n" +
                    "</xsl:template>\n" +
                "</xsl:stylesheet>")
        .result("result");

    echo("${result}");

    transform()
        .source(new ClassPathResource("com/consol/citrus/actions/transform-source.xml"))
        .xslt(new ClassPathResource("com/consol/citrus/actions/transform.xslt"))
        .result("result");

    echo("${result}");
}
```

Defining multi-line Strings with nested quotes is no fun in Java. So you may want to use external file resources for your scripts as shown in the second part of the example. In fact you could also use script languages like Groovy or Scala that have much better support for multi-line Strings.

11.10. 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.

Let's have a look at some examples in order to understand the possible Groovy code interactions in Citrus:

XML DSL

```
<testcase name="groovyTest">
  <variables>
    <variable name="time" value="citrus:currentDate()"/>
  </variables>
  <actions>
    <groovy>
        println 'Hello Citrus'
    </groovy>
    <groovy>
        println 'The variable is: ${time}'
    </groovy>
    <groovy resource="classpath:com/consol/citrus/script/example.groovy"/>
  </actions>
</testcase>
```

Java DSL

```
protected void configure() {
    groovy("println 'Hello Citrus'");
    groovy("println 'The variable is: ${time}'");

    groovy(new ClassPathResource("com/consol/citrus/script/example.groovy"));
}
```

As you can see it is possible to write Groovy code directly into the test case. Citrus will interpret and execute the Groovy code at runtime. As usual nested variable expressions are replaced with respective values. In general this is done in advance before the Groovy code is interpreted. For more complex Groovy code sections which grow in lines of code you can also reference external file resources.

After this basic Groovy code usage inside a test case we might be interested accessing the whole TestContext. The TestContext Java object holds all test variables and function definitions for the test case and can be referenced in Groovy code via simple naming convention. Just access the object reference 'context' and you are able to manipulate the TestContext (e.g. setting a new variable which is directly ready for use in following test actions).

XML DSL

```
<testcase name="groovyTest">
  <actions>
    <groovy>
      context.setVariable("greetingText","Hello Citrus")
      println context.getVariable("greetingText")
    </groovy>
    <echo>
      <message>New variable: ${greetingText}</message>
    </echo>
  </actions>
</testcase>
```

Note

The implicit TestContext access that was shown in the previous sample works with a default Groovy script template provided by Citrus. The Groovy code you write in the test case is automatically surrounded with a Groovy script which takes care of handling the TestContext. The default template looks like follows:

```
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) {
        @SCRIPTBODY@
    }
}
```

Your code is placed in substitution to the `@SCRIPTBODY@` placeholder. Now you might understand how Citrus handles the context automatically. You can also write your own script templates making more advanced usage of other Java APIs and Groovy code. Just add a script template path to the test action like this:

```
<groovy script-template="classpath:my-custom-template.groovy">
  [...]
</groovy>
```

On the other hand you can disable the automatic script template wrapping in your action at all:

```
<groovy use-script-template="false">
  println 'Just use some Groovy code'
</groovy>
```

The next example deals with advanced Groovy code and writing whole classes. We write a new Groovy class which implements the ScriptExecutor interface offered by Citrus. This interface defines a special execute method and provides access to the whole TestContext for advanced test variables access.

```
<testcase name="groovyTest">
  <variables>
    <variable name="time" value="citrus:currentDate()"/>
  </variables>
  <actions>
    <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>
  </actions>
</testcase>
```

Implementing the ScriptExecutor interface in a custom Groovy class is applicable for very special test context manipulations as you are able to import and use other Java API classes in this code.

11.11. 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:

XML DSL

```
<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.CitrusRuntimeException:
Test will fail with custom message

[...]

CITRUS TEST RESULTS

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%)
```

When using the Java DSL someone could think of throwing Java exceptions in the middle of configuring the test case. But this is not possible as the configure method execution time is not the runtime of the Java test. During configuration you specify all test actions and the test execution time is done to some later state.

Java DSL

```
protected void configure() {
    // some test actions

    throw new ValidationException("This test should fail now"); // does not work as expected
}
```

The validation exception above is directly raised before the test is able to start as the configure method does not represent the test runtime. Instead of this use the fail action as follows:

Java DSL

```
protected void configure() {
    // some test actions

    fail("This test should fail now"); // fails at test runtime as expected
}
```

11.12. 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.

XML DSL

```
<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>
```

As you can see the input action is customizable with a prompt message that is displayed to the user and some valid answer possibilities. The user input is stored to a test variable for further use in the test case. In detail the input action offers following attributes:

- *message* -> message displayed to the user
- *valid-answers* -> optional slash separated string containing the possible valid answers
- *variable* -> result variable name holding the user input (default = ${userinput})

The same action in Java DSL now looks quite familiar to us although attribute naming is slightly different:

Java DSL

```
protected void configure() {
    variable("userinput", "");
    variable("userinput1", "");
    variable("userinput2", "y");
    variable("userinput3", "yes");
    variable("userinput4", "");

    input();
    echo("user input was: ${userinput}");
    input().message("Now press enter:").result("userinput1");
    echo("user input was: ${userinput1}");
    input().message("Do you want to continue?").answers("y", "n").result("userinput2");
    echo("user input was: ${userinput2}");
    input().message("Do you want to continue?").answers("yes", "no").result("userinput3");
    echo("user input was: ${userinput3}");
    input().result("userinput4");
    echo("user input was: ${userinput4}");
}
```

When the user input is restricted to a set of valid answers the input validation of course can fail due to mismatch. This is the case when the user provides some input not matching the valid answers given. In this case the user is again asked to provide valid input. The test action will continue to ask for valid 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.

11.13. 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
```

XML DSL

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

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

Java DSL

```
protected void configure() {
    load("file:tests/resources/load.properties");

    traceVariables();
}
```

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

Please be aware of the fact that existing variables are overwritten!

11.14. 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 our test case. This can be done with following action definition:

XML DSL

```
<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>
```

Java DSL

```
@Autowired
@Qualifier("connectionFactory")
private ConnectionFactory connectionFactory;

protected void configure() {
    purgeQueues()
        .queue("Some.JMS.QUEUE.Name")
        .queue("Another.JMS.QUEUE.Name");

    purgeQueues(connectionFactory)
        .timeout(150L) // custom timeout in ms
        .queue("Some.JMS.QUEUE.Name")
        .queue("Another.JMS.QUEUE.Name");
}
```

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 22.3, “Before test” 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:

XML DSL

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

Of course you can use queue object references also in Java DSL test cases. Here we easily can use Spring's dependency injection with autowiring to get the object references from the container.

Java DSL

```
@Autowired
@Qualifier("jmsQueueHelloRequestIn")
private Queue jmsQueueHelloRequestIn;

@Autowired
@Qualifier("jmsQueueHelloResponseOut")
private Queue jmsQueueHelloResponseOut;

protected void configure() {
    purgeQueues()
        .queue(jmsQueueHelloRequestIn)
        .queue(jmsQueueHelloResponseOut);
}
```

Note

You can mix queue name and queue object references as you like within one single purge queue test action.

11.15. Purging message channels

Message channels define central messaging destinations in Citrus. These are namely in memory message queues holding messages for test cases. These messages may become obsolete during a test run, especially when test cases fail and stop in their message consumption. Purging these message channel destinations is essential in these scenarios in order to not influence upcoming test cases. Each test case should only receive those messages that actually refer to the test model. Therefore it is a good idea to purge all message channel destinations between the test cases. Obsolete messages that get stuck in a message channel destination for some reason are then removed so that upcoming test case are not broken.

Following action definition purges all messages from a list of message channels:

XML DSL

```
<testcase name="purgeChannelTest">
  <actions>
      <purge-channels>
          <channel name="someChannelName"/>
          <channel name="anotherChannelName"/>
      </purge-channels>

      <purge-channel>
          <channel ref="someChannel"/>
          <channel ref="anotherChannel"/>
      </purge-channel>
  </actions>
</testcase>
```

As you can see the test action supports channel names as well as channel references to Spring bean instances. When using channel references you refer to the Spring bean id or name in your application context.

The Java DSL works quite similar as you can read from next examples:

Java DSL

```
@Autowired
@Qualifier("channelResolver")
private ChannelResolver channelResolver;

protected void configure() {
    purgeChannels()
        .channelResolver(channelResolver)
        .channelNames("ch1", "ch2", "ch3")
        .channel("ch4");
}
```

The channel resolver reference is optional. By default Citrus will automatically use a Spring application context channel resolver so you just have to use the respective Spring bean names that are configured in the Spring application context. However setting a custom channel resolver may be adequate for you in some special cases.

While speaking of Spring application context bean references the next example uses such bean references for channels to purge.

Java DSL

```
@Autowired
@Qualifier("channel1")
private MessageChannel channel1;
```

```
@Autowired
@Qualifier("channel2")
private MessageChannel channel2;

@Autowired
@Qualifier("channel3")
private MessageChannel channel3;

protected void configure() {
    purgeChannels()
        .channels(channel1, channel2)
        .channel(channel3);
}
```

Message selectors enable you to selectively remove messages from the destination. All messages that pass the message selection logic get deleted the other messages will remain unchanged inside the channel destination. The message selector is a Spring bean that implements a special message selector interface. A possible implementation could be a selector deleting all messages that are older than five seconds:

```
import org.springframework.integration.Message;
import org.springframework.integration.core.MessageSelector;

public class TimeBasedMessageSelector implements MessageSelector {

    public boolean accept(Message<?> message) {
        if (System.currentTimeMillis() - message.getHeaders().getTimestamp() > 5000) {
            return false;
        } else {
            return true;
        }
    }

}
```

Note

The message selector returns *false* for those messages that should be deleted from the channel!

You simply define the message selector as a new Spring bean in the Citrus application context and reference it in your test action property.

```
<bean id="specialMessageSelector"
    class="com.consol.citrus.special.TimeBasedMessageSelector"/>
```

Now let us have a look at how you reference the selector in your test case:

XML DSL

```
<purge-channels message-selector="specialMessageSelector">
          <channel name="someChannelName"/>
          <channel name="anotherChannelName"/>
      </purge-channels>
```

Java DSL

```
@Autowired
@Qualifier("specialMessageSelector")
private MessageSelector specialMessageSelector;

protected void configure() {
    purgeChannels()
```

```
        .channelNames("ch1", "ch2", "ch3")
        .selector(specialMessageSelector);
}
```

Purging channels in each test case every time is quite exhausting because every test case needs to define a purging action at the very beginning of the test. A more straight forward approach would be to introduce some purging action which is automatically executed before each test. Fortunately the Citrus test suite offers a very simple way to do this. It is described in Section 22.3, “Before test”.

When using the special action sequence between test cases you must define a normal Spring bean definition first. The 'com.consol.citrus.actions.PurgeMessageChannelAction' bean offers the respective properties for setting channel destinations to be purged. See the upcoming example to find out how the action is defined in the Spring configuration application context.

```
<bean id="purgeMessageChannels"
    class="com.consol.citrus.actions.PurgeMessageChannelsAction">
    <property name="channelNames">
        <list>
            <value>fooChannel</value>
            <value>barChannel</value>
        </list>
    </property>
</bean>
```

Use this Spring bean definition in the action sequence before each test and your test do not influence each other with obsolete messages still waiting on the message channels for consumption.

11.16. 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:

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    assertException(echo("Current date is: ${date}"))
        .exception(com.consol.citrus.exceptions.CitrusRuntimeException.class)
        .message("Unknown variable ${date}");
}
```

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.

11.17. Catch exceptions

In the previous chapter we have seen how to expect failures in Citrus with assert action. Now the assert action is designed for single actions to be monitored and for failures to be expected in any case. The *'catch'* action in contrary can hold several nested test actions and exception failure is optional.

The nested actions are error proof for the chosen exception type. This means possible exceptions are caught and ignored - the test case will not fail for this exception type. But only for this particular exception type! Other exception types that occur during execution do cause the test to fail as usual.

XML DSL

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

Java DSL

```
protected void configure() {
    catchException(CitrusRuntimeException.class, echo("Current date is: ${date}"));
}
```

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 raised. In contrary to that the assert action requires the exception and an assert action is 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.

Note

Java developers might ask why not use try-catch Java block instead? The answer is simple yet very important to understand. The configure() method is called by the Java DSL test case builder not at test runtime but before that. This means that a try-catch block within the configure() method will never perform during the test run. Only adding the catch test action as part of the test case will result in expected behavior.

11.18. Running ANT build targets

The <ant> action loads a build.xml ANT file and executes one or more targets in the ANT project. The target is executed with optional build properties passed to the ANT run. The ANT build output is logged with Citrus logger and the test case success is bound to the ANT build success. This means in case the ANT build fails for some reason the test case will also fail with build exception accordingly.

See this basic ANT run example to see how it works within your test case:

XML DSL

```
<testcase name="AntRunTest">
    <variables>
        <variable name="today" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <ant build-file="classpath:com/consol/citrus/actions/build.xml">
            <execute target="sayHello"/>
            <properties>
                <property name="date" value="${today}"/>
                <property name="welcomeText" value="Hello!"/>
            </properties>
        </ant>
    </actions>
</testcase>
```

Java DSL

```
protected void configure() {
    variable("today", "citrus:currentDate()");

    antrun("classpath:com/consol/citrus/actions/build.xml")
        .target("sayHello")
        .property("date", "${today}")
        .property("welcomeText", "$Hello!");
}
```

The respective build.xml ANT file must provide the target to call. For example:

```
<project name="citrus-build" default="sayHello">
    <property name="welcomeText" value="Welcome to Citrus!"></property>

    <target name="sayHello">
        <echo message="${welcomeText} - Today is ${date}"></echo>
    </target>

    <target name="sayGoodbye">
        <echo message="Goodbye everybody!"></echo>
    </target>
</project>
```

As you can see you can pass custom build properties to the ANT build execution. Existing ANT build

properties are replaced and you can use the properties in your build file as usual.

You can also call multiple targets within one single build run by using a comma separated list of target names:

XML DSL

```
<testcase name="AntRunTest">
    <variables>
        <variable name="today" value="citrus:currentDate()"/>
    </variables>
    <actions>
        <ant build-file="classpath:com/consol/citrus/actions/build.xml">
            <execute targets="sayHello,sayGoodbye"/>
            <properties>
                <property name="date" value="${today}"/>
            </properties>
        </ant>
    </actions>
</testcase>
```

Java DSL

```
protected void configure() {
    variable("today", "citrus:currentDate()");

    antrun("classpath:com/consol/citrus/actions/build.xml")
        .targets("sayHello", "sayGoodbye")
        .property("date", "${today}");
}
```

The build properties can live in external file resource as an alternative to the inline property definitions. You just have to use the respective file resource path and all nested properties get loaded as build properties.

In addition to that you can also define a custom build listener. The build listener must implement the ANT API interface *org.apache.tools.ant.BuildListener*. During the ANT build run the build listener is called with several callback methods (e.g. buildStarted(), buildFinished(), targetStarted(), targetFinished(), ...). This is how you can add additional logic to the ANT build run from Citrus. A custom build listener could manage the fail state of your test case, in particular by raising some exception forcing the test case to fail accordingly.

XML DSL

```
<testcase name="AntRunTest">
    <actions>
        <ant build-file="classpath:com/consol/citrus/actions/build.xml"
                build-listener="customBuildListener">
            <execute target="sayHello"/>
            <properties file="classpath:com/consol/citrus/actions/build.properties"/>
        </ant>
    </actions>
</testcase>
```

Java DSL

```
@Autowired
private BuildListener customBuildListener;

protected void configure() {
    antrun("classpath:com/consol/citrus/actions/build.xml")
        .target("sayHello")
        .propertyFile("classpath:com/consol/citrus/actions/build.properties")
        .listener(customBuildListener);
}
```

The *customBuildListener* used in the example above should reference a Spring bean in the Citrus application context. The bean implements the interface *org.apache.tools.ant.BuildListener* and controls the ANT build run.

11.19. Including custom test actions

Now we have a look at the opportunity to add custom test actions to the test case flow. Let us start this section with an example:

XML DSL

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

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 Citrus test case. This way you can easily implement your own actions in Java and include them into the test case.

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.

```
<bean id="cleanUpDatabase" class="my.domain.citrus.actions.SpecialDatabaseCleanupAction">
    <property name="dataSource" ref="testDataSource"/>
</bean>
```

The Spring application context holds your custom bean implementations. You can set properties and use the full Spring power while implementing your custom test action in Java. Let us have a look on how such a Java class may look like.

```
import com.consol.citrus.actions.AbstractTestAction;
import com.consol.citrus.context.TestContext;

public class SpecialDatabaseCleanupAction extends AbstractTestAction {

    @Autowired
    private DataSource dataSource;

    @Override
    public void doExecute(TestContext context) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.execute("...");
    }

}
```

All you need to do in your Java class is to implement the Citrus `com.consol.citrus.TestAction` interface. The abstract class `com.consol.citrus.actions.AbstractTestAction` may help you to start with your custom test action implementation as it provides basic method implementations so you just have to implement the `doExecute()` method.

When using the Java test case DSL you are also quite comfortable with including your custom test actions.

```
@Autowired
private SpecialDatabaseCleanupAction cleanUpDatabaseAction;

protected void configure() {
    echo("Now let's include our special test action");

    action(cleanUpDatabaseAction);

    echo("That's it!");
}
```

Using anonymous class implementations is also possible.

```
protected void configure() {
    echo("Now let's call our special test action anonymously");

    action(new AbstractTestAction() {
        public void doExecute(TestContext context) {
            // do something
        }
    });

    echo("That's it!");
}
```

Chapter 12. 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:

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    variable("myTime", "citrus:currentDate()");

    template("doCreateVariables");

    template("doTraceVariables")
        .parameter("time", "${myTime}");
}
```

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 13.3, “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 13. 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.

13.1. 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.

XML DSL

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

Java DSL

```
protected void configure() {
    sequential(
        stopTime(),
        sleep(1.0),
        echo("Hello Citrus"),
        stopTime()
    );
}
```

13.2. Conditional

Now we deal with conditional executions of test actions. Nested actions inside a conditional container are executed only in case a booleand expression evaluates to true. Otherwise the container execution is not performed at all.

See some example to find out how it works with the conditional expression string.

XML DSL

```
<testcase name="conditionalTest">
    <actions>
        <conditional expression="${value} = 5">
            <sleep time="10"/>
        </conditional>
    </actions>
</testcase>
```

Java DSL

```
protected void configure() {
    conditional(
        sleep(10000L)
    ).when("${value} = 5");
}
```

The nested sleep action is executed in case the variable ${value} is equal to the value '5'. This conditional execution of test actions is useful when dealing with different test environments such as different operating systems for instance.

13.3. 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:

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    parallel(
        sleep(),
        sequential(
            sleep(),
            echo("1")
        ),
        echo("2"),
        echo("3"),
```

```
        iterate(
            echo("10")
        ).condition("i lt= 5").index("i")
    );
}
```

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.

13.4. 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.

XML DSL

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

Java DSL

```
protected void configure() {
    iterate(
        echo("index is: ${i}")
    ).condition("i lt 5").index("i");
}
```

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.

13.5. 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.

Note

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.

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    repeat(echo("index is: ${i}"))
        .until("(i gt 5) or (i = 3)")
        .index("i");
}
```

13.6. 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.

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    repeatOnError(
        echo("index is: ${i}"),
        fail("Force loop to fail!")
    ).until("i = 5").index("i");
}
```

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.

XML DSL

```
<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>
```

Java DSL

```
protected void configure() {
    repeatOnError(
        query(testDataSource)
            .statement("SELECT COUNT(1) AS CNT_ORDERS FROM ORDERS WHERE CUSTOMER_ID='${customerId}'")
            .validate("CNT_ORDERS", "1")
    ).until("i = 5").index("i").autoSleep(1L);
}
```

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 14. 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:

XML DSL

```
<testcase name="finallyTest">
    <variables>
        <variable name="orderId" value="citrus:randomNumber(5)"/>
        <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).

Of course you can also use the finally block in the Java test case DSL. Find following example to see how it works:

Java DSL

```
protected void configure() {
    variable("orderId", "citrus:randomNumber(5)");
    variable("date", "citrus:currentDate('dd.MM.yyyy')");

    sql(dataSource)
        .statement("INSERT INTO ORDERS VALUES (${orderId}, 1, 1, '${date}')");
```

```
    echo("ORDER creation time: citrus:currentDate('dd.MM.yyyy')");

    doFinally(
        sql(dataSource).statement("DELETE FROM ORDERS WHERE ORDER_ID='${orderId}'")
    );
}
```

Note

Java developers might ask why not use try-finally Java block instead? The answer is simple yet very important to understand. The configure method is called by the Java DSL test case builder not at test runtime but before that. This means that a try-finally block within the configure() method will never perform the finally during the test run. Only adding the finally section as part of the test case with doFinally() will result in expected behavior.

Chapter 15. JMS support

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.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.

15.1. 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. Apache ActiveMQ, 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.

15.2. 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

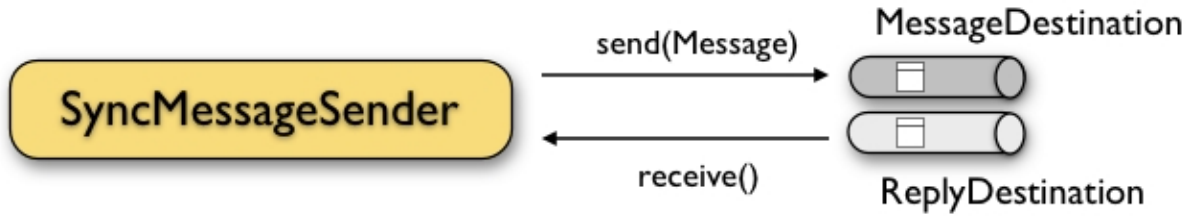
Besides the destination-name attribute you can also provide a reference to a Destination implementation.

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

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

15.3. 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.

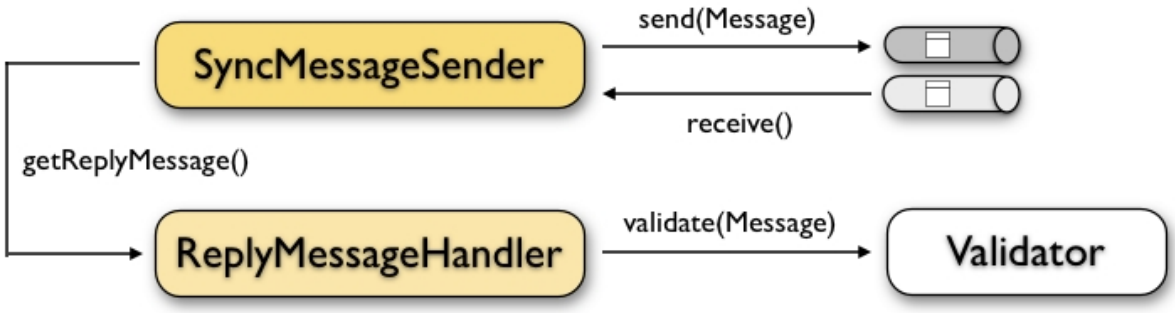


In the Spring XML configuration the synchronous message senders are quite similar to the asynchronous brothers.

```
<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"
                                polling-interval="1000"/>
```

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 component sends the message and waits synchronously for the response message to arrive. While the sender component is waiting for the response the reply handler is polling for the response several times (depends on polling interval and reply timeout settings). When the reply message arrives the reply handler takes over and performs further processing steps such as message validation and so on. In cas no reply message arrives in time a message timeout error is raised respectively.

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 further 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"/>
```

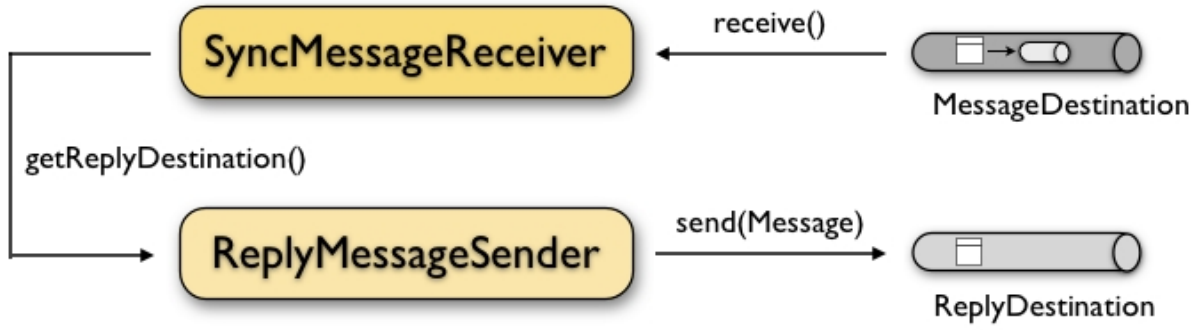
Instead of using the *reply-destination-name* feel free to use the destination reference *reply-destination*

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.

15.4. 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 message producer stores a dynamic JMS queue destination to 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"
                    receive-timeout="5000"/>

<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.

15.5. 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 destination 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.

15.6. JMS message headers

The JMS specification defines a set of special message header entries that can go into your JMS message. These JMS headers are stored differently in a JMS message header than other custom header entries do. Therefore these special header values should be set in a special syntax that we discuss in the next paragraphs.

```
<header>
    <element name="jms_correlationId" value="${correlationId}"/>
    <element name="jms_messageId" value="${messageId}"/>
    <element name="jms_redelivered" value="${redelivered}"/>
    <element name="jms_timestamp" value="${timestamp}"/>
</header>
```

As you see all JMS specific message headers use the *jms_* prefix. This prefix comes from Spring Integration message header mappers that take care of setting those headers in the JMS message

header properly.

Typing of message header entries may also be of interest in order to meet the JMS standards of typed message headers. For instance the following message header is of type double and is therefore transfered via JMS as a double value.

```
<header>
    <element name="amount" value="19.75" type="double"/>
</header>
```

Chapter 16. Http support

Citrus is able to connect with Http services and simulate RESTful WebServices servers. In the next sections you will learn how to invoke Http services and how to handle REST Http requests in a test case. We deal with setting up a Http server in order to accept client requests and provide proper Http responses with GET, PUT, DELETE or POST request method.

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 files 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.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 with the citrus-http namespace prefix.

16.1. Http message sender

Citrus is able to invoke a Http service as client waiting for the response message from server. After that the response message 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"
                            content-type="application/xml"
                            reply-handler="httpResponseHandler"/>

<citrus-http:reply-message-handler id="httpResponseHandler"
                                   polling-interval="1000"/>
```

As Http communication is always synchronous we need a reply message handler in combination with the Http message-sender. It is not very surprising that the sender also needs the *request-url* and a *request-method* as parameter. In addition to that we can specify the content-type of the request we are about to send. The message sender builds the Http request and sends it to the Http server. While the sender component is waiting for the synchronous Http response to arrive the reply handler polls several times for the response message. The reply handler polling interval is set as well as the reply message timeout in order to handle situations where messages arrive late or not in time. In case the reply message comes in too late according to the timeout settings a respective timeout error is raised.

The request method is statically set to *POST* in the example above. You can also set/overwrite the Http request method inside the sending test action which gives more flexibility. Use something like this in your test:

```
<send with="httpMessageSender">
    <message>
        <data>
            <![CDATA[
                <testRequestMessage>
                    <text>Hello HttpServer</text>
                </testRequestMessage>
            ]]>
        </data>
    </message>
    <header>
        <element name="citrus_http_method" value="PUT"/>
    </header>
</send>
```

Tip

Citrus uses Spring's REST template mechanism for sending Http requests. This means you have great customizing opportunities with a special REST template configuration. You can think of basic Http authentication, read timeouts and special message factory implementations. Just use the custom REST template attribute in message sender like this:

```
<citrus-http:message-sender id="httpMessageSender"
                            request-url="http://localhost:8090/test"
                            request-method="POST"
                            content-type="text/plain"
                            rest-template="customizedRestTemplate"
                            reply-handler="httpResponseHandler"/>

<!-- Customized rest template -->
<bean name="customizedRestTemplate" class="org.springframework.web.client.RestTemplate">
  <property name="messageConverters">
    <util:list id="converter">
      <bean class="org.springframework.http.converter.StringHttpMessageConverter">
        <property name="supportedMediaTypes">
          <util:list id="types">
            <value>text/plain</value>
          </util:list>
        </property>
      </bean>
    </util:list>
  </property>
  <property name="errorHandler">
    <!-- Custom error handler -->
  </property>
  <property name="requestFactory">
    <bean class="org.springframework.http.client.HttpComponentsClientHttpRequestFactory">
      <property name="readTimeout" value="9000" />
    </bean>
  </property>
</bean>
```

Tip

And another tip for you regarding dynamic endpoint URI in Http message sender. Similar to the endpoint resolving mechanism in SOAP sending action you can dynamically set the called endpoint URI on Http message sender. By default Citrus will check a specific header entry for dynamic endpoint URI which is simply defined for each message sending action inside the test. See the next short example and you will see that it is very simple.

```
<header>
    <element name="citrus_endpoint_uri" value="http://localhost:8080/customers/${customerId}"/>
</header>
```

The specific send action above will send its message to the dynamic endpoint (http://localhost:8080/customers/${customerId}) which is set in the header *citrus_endpoint_uri*. As you can see the endpoint contains a variable entry so you can reuse the same Http message sender with different endpoint URI. This is essential when calling RESTful WebServices where the URI contains parameters, identifiers and modifiers.

16.2. Http server

Sending Http messages was quite easy and straight forward. Receiving Http messages is a little bit more complicated, because Citrus has to provide server functionality listening on a local 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, which is a preconfigured Jetty server instance with Spring web application context support.

```
<citrus-http:server id="simpleHttpServer"
                port="8090"
                auto-start="true"
                context-config-location="classpath:com/consol/citrus/http/citrus-servlet-context.xml"
                resource-base="src/citrus/resources"/>
```

The Http Jetty server will automatically start when the Spring application context is loaded (auto-start="true"). The basic connector is listening on port 8090 for requests. The server automatically uses Spring application context loading on startup. The servlet context file is specified via classpath and hold all custom configurations for the server. Here is a sample servlet context with some basic Spring MVC components and the central HttpMessageController which is responsible for handling incoming requests (GET, PUT, DELETE, POST, etc.).

```
<bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"/>

<bean class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">
  <property name="messageConverters">
    <util:list id="converters">
      <bean class="org.springframework.http.converter.StringHttpMessageConverter">
        <property name="supportedMediaTypes">
          <util:list>
            <value>text/xml</value>
          </util:list>
        </property>
      </bean>
    </util:list>
  </property>
</bean>

<bean class="com.consol.citrus.http.controller.HttpMessageController">
  <property name="messageHandler">
      <bean id="emptyResponseProducingMessageHandler"
       class="com.consol.citrus.adapter.handler.EmptyResponseProducingMessageHandler"/>
  </property>
</bean>
```

The beans above are responsible for proper Http server configuration. In general you do not need to adjust those beans, but we have the possibility to do so which gives us a great customization and

extension points. The important part is the message handler definition inside the HttpMessageController. 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 for different simulation strategies. With these message handler implementations you should be able to generate proper response messages for the client in a test case. Let's have a look at them in the following sections.

16.2.1. 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.adapter.handler.EmptyResponseProducingMessageHandler"/>
```

16.2.2. 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.

16.2.3. Xpath dispatching message handler

The idea behind the xpath-dispatching-message-handler is that the incoming requests are dispatched to several message handlers depending on the existence of a specific node inside the message payload. The XPath expression will evaluate to the parent of the deciding node and call the

respective message handler. The message handler mapping is done by mapping the deciding node name to a bean name in a message handler Spring configuration context. The separate context is loaded in advance. Let's see an example.

```
<bean id="xpathDispatchingHandler"
  class="com.consol.citrus.adapter.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 ApplicationContext file resource. The message handlers in this ApplicationContext are mapped to the different values via their names. For instance an incoming request with `//MessageBody/Operation/GetOrders` would be handled by the message handler named "GetOrders". The available message handlers are configured in the separate message-handler-context (e.g. EmptyResponseProducingMessageHandler, StaticResponseProducingMessageHandler, ...). Here is an example.

```
<bean name="GetOrders"
  class="com.consol.citrus.adapter.handler.StaticResponseProducingMessageHandler">
    <property name="messagePayload" value="Your orders"/>
</bean>
<bean name="SaveOrder"
  class="com.consol.citrus.adapter.handler.StaticResponseProducingMessageHandler">
    <property name="messagePayload" value="Order saved"/>
</bean>
```

Tip

It is possible to ommit the xpath expression in the xpath dispatching message handler bean definition. If no xpath is provided, Citrus will take the first element node in the request into account.

16.2.4. 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.adapter.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 strategy (Appendix A, *Citrus Samples*)

16.3. Http headers

When dealing with Http request/response communication we always deal with Http specific headers. The Http protocol defines a group of header attributes that both client and server need to handle. You can set and validate these Http headers in Citrus quite easy. Let us have a look at a client operation in Citrus where some Http headers are explicitly set before the request is sent.

```
<send with="httpMessageSender">
    <message>
        <data>
            <![CDATA[
                <testRequestMessage>
                    <text>Hello HttpServer</text>
                </testRequestMessage>
            ]]>
        </data>
    </message>
    <header>
        <element name="CustomHeaderId" value="${custom_header_id}"/>
        <element name="Content-Type" value="text/xml"/>
        <element name="Accept" value="text/xml,*/*"/>
    </header>
</send>
```

We are able to set custom headers (*CustomHeaderId*) that go directly into the Http header section of the request. In addition to that testers can explicitly set Http reserved headers such as *Content-Type*. Fortunately you do not have to set all headers on your own. Citrus will automatically set the required Http headers for the request. So we have the following Http request which is sent to the server:

```
POST /test HTTP/1.1
Accept: text/xml, */*
Content-Type: text/xml
CustomHeaderId: 123456789
Accept-Charset: macroman
User-Agent: Jakarta Commons-HttpClient/3.1
Host: localhost:8091
Content-Length: 175
<testRequestMessage>
    <text>Hello HttpServer</text>
</testRequestMessage>
```

On server side testers are interested in validating the Http headers. Within Citrus receive action you simply define the expected header entries. The Http specific headers are automatically available for validation as you can see in this example:

```
<receive with="httpRequestReceiver">
    <message>
        <data>
            <![CDATA[
                <testRequestMessage>
                    <text>Hello HttpServer</text>
                </testRequestMessage>
            ]]>
        </data>
    </message>
    <header>
        <element name="CustomHeaderId" value="${custom_header_id}"/>
        <element name="Content-Type" value="text/xml"/>
        <element name="Accept" value="text/xml,*/*"/>
    </header>
</receive>
```

The test checks on custom headers and Http specific headers to meet the expected values.

Now that we have accepted the client request and validated the contents we are able to send back a

proper Http response message. Same thing here with Http specific headers. The Http protcol defines several headers marking the success or failure of the server operation. In the test case you can set those headers for the response message with convention header names. See the following example to find out how that works for you.

```
<send with="httpResponseSender">
    <message>
        <data>
            <![CDATA[
                <testResponseMessage>
                    <text>Hello Citrus Client</text>
                </testResponseMessage>
            ]]>
        </data>
    </message>
    <header>
        <element name="CustomHeaderId" value="${custom_header_id}"/>
        <element name="Content-Type" value="text/xml"/>
        <element name="citrus_http_status_code" value="200"/>
        <element name="citrus_http_reason_phrase" value="OK"/>
    </header>
</send>
```

Once more we set the custom header entry (*CustomHeaderId*) and a Http reserved header (*Content-Type*) for the response message. On top of this we are able to set the response status for the Http response. We use the reserved header names *citrus_http_status_code* in order to mark the success of the server operation. With this mechanism we can easily simulate different server behaviour such as Http error response codes (e.g. 404 - Not found, 500 - Internal error). Let us have a closer look at the generated response message:

```
HTTP/1.1 200 OK
Content-Type: text/xml;charset=UTF-8
Accept-Charset: macroman
Content-Length: 205
Server: Jetty(7.0.0.pre5)
<testResponseMessage>
    <text>Hello Citrus Client</text>
</testResponseMessage>
```

Tip

You do not have to set the reason phrase all the time. It is sufficient to only set the Http status code. Citrus will automatically add the proper reason phrase for well known Http status codes.

The only thing that is missing right now is the validation of Http status codes when receiving the server response in a Citrus test case. It is very easy as you can use the Citrus reserved header names for validation, too.

```
<receive with="httpResponseReceiver">
    <message>
        <data>
            <![CDATA[
                <testResponseMessage>
                    <text>Hello Test Framework</text>
                </testResponseMessage>
            ]]>
        </data>
    </message>
    <header>
        <element name="CustomHeaderId" value="${custom_header_id}"/>
        <element name="citrus_http_status_code" value="200"/>
        <element name="citrus_http_version" value="HTTP/1.1"/>
        <element name="citrus_http_reason_phrase" value="OK"/>
    </header>
```

```
</receive>
```

Up to now we have used some of the basic Citrus reserved Http header names (citrus_http_status_code, citrus_http_version, citrus_http_reason_phrase). In Http RESTful WebServices some more header names are essential ofr validation. These are request attributes like query parameters, context path and request URI. The Citrus server side REST message controller will automatically add all this information to the message header for you. So all you need to do is validate the header entries in your test.

The next example receives a Http GET method request on server side. Here the GET request does not have any message payload, so the validation just works on the information given in the message header. We assume the client to call *http://localhost:8080/app/users?id=123456789*. As a tester we need to validate the request method, request URI, context path and the query parameters.

```
<receive with="httpRequestReceiver">
    <message>
        <data></data>
    </message>
    <header>
        <element name="Host" value="localhost:8080"/>
        <element name="Content-Type" value="text/html"/>
        <element name="Accept" value="text/xml,*/*"/>
        <element name="citrus_http_method" value="GET"/>
        <element name="citrus_http_request_uri" value="/app/users"/>
        <element name="citrus_http_context_path" value="/app"/>
        <element name="citrus_http_query_params" value="id=123456789"/>
    </header>
</receive>
```

Tip

Be aware of the slight differences in request URI and context path. The context path gives you the web application context path within the servlet container for your web application. The request URI always gives you the complete path that was called for this request.

Important

Another important thing to notice is the usage of multiple query parameters that are put together using '&' characters (e.g. http://localhost:8080/app/users?id=123456789&name=foo). As the Citrus test case is written in XML we have to escape the reserved '&' with & entity (e.g. value="id=123456789&name=test").

As you can see we are able to validate all parts of the initial request endpoint URI the client was calling. This completes the Http header processing within Citrus. On both client and server side Citrus is able to set and validate Http specific header entries which is essential for simulating Http communication.

16.4. Http error handling

So far we have received response messages with Http status code *200 OK*. How to deal with server errors like *404 Not Found* or *500 Internal server error*? The default Http message client error strategy is to propagate server error response messages to the reply message handler for validation. We simply check on Http status code and status text for error validation.

```
<send with="httpMessageSender">
    <message>
        <data>
            <![CDATA[
                <testRequestMessage>
                    <text>Hello HttpServer</text>
                </testRequestMessage>
            ]]>
        </data>
    </message>
</send>

<receive with="httpResponseReceiver">
    <message>
        <data><![CDATA[]]></data>
    </message>
    <header>
        <element name="citrus_http_status_code" value="403"/>
        <element name="citrus_http_reason_phrase" value="FORBIDDEN"/>
    </header>
</receive>
```

The message data can be empty depending on the server logic for these error situations. If we receive additional error information as message payload just add validation assertions as usual.

Instead of receiving such empty messages with checks on Http status header information we can change the error strategy in the message sender component in order to automatically raise exceptions on response messages other than *200 OK*. Therefore we go back to the Http message sender configuration for changing the error strategy.

```
<citrus-http:message-sender id="httpMessageSender"
                            request-url="http://localhost:8080/test"
                            reply-handler="httpResponseReceiver"
                            error-strategy="throwsException"/>

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

Now we expect an exception to be thrown because of the error response. Following from that we have to change our test case. Instead of receiving the error message with receive action we assert the client exception and check on the Http status code and status text.

```
<assert exception="org.springframework.web.client.HttpClientErrorException"
          message="403 Forbidden">
    <send with="httpMessageSender">
        <message>
            <data>
                <![CDATA[
                    <testRequestMessage>
                        <text>Hello HttpServer</text>
                    </testRequestMessage>
                ]]>
            </data>
        </message>
    </send>
</assert>
```

Both ways of handling Http error messages on client side are valid for expecting the server to raise Http error codes. Choose the prefered way according to your test project requirements.

16.5. Client basic authentication

As client you may have to use basic authentication in order to access a resource on the server. In most cases this will be username/password authentication where the credentials are transmtted in the request header section as base64 encoding. As Citrus uses Spring's REST support with the

RestTemplate and ClientHttpRequestFactory the basic authentication is already covered. You simply have to set the user credentials on the HttpClient which is used inside the RestTemplate.

Citrus gives a even more comfortable way to configure the basic authentication credentials on the RestTemplate's ClientHttpRequestFactory. Just see the following example and learn how to do that.

```
<citrus-http:message-sender id="testHttpMessageSender"
                            request-method="POST"
                            request-url="http://localhost:8080/test"
                            reply-handler="testHttpReplyMessageHandler"
                            request-factory="basicAuthFactory"/>

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

<bean id="basicAuthFactory" class="com.consol.citrus.http.client.BasicAuthClientHttpRequestFactory">
  <property name="authScope">
      <bean class="org.apache.http.auth.AuthScope">
        <constructor-arg value="localhost"/>
        <constructor-arg value="8072"/>
        <constructor-arg value=""/>
        <constructor-arg value="basic"/>
      </bean>
  </property>
  <property name="credentials">
    <bean class="org.apache.http.auth.UsernamePasswordCredentials">
        <constructor-arg value="someUsername"/>
        <constructor-arg value="somePassword"/>
    </bean>
  </property>
</bean>
```

Important

Since Citrus has upgraded to Spring 3.1.x the Jakarta commons Http client is deprecated with Citrus version 1.2. The formerly used UserCredentialsClientHttpRequestFactory is therefore also deprecated and will not continue with next versions. Please update your configuration if you are coming from Citrus 1.1 or earlier versions.

The above configuration results in Http client requests with authentication headers properly set for basic authentication. The client request factory takes care on adding the proper basic authentication header to each request that is sent with this Citrus message sender. Citrus uses preemtive authentication. The message sender only sends a single request to the server with all authentication information set in the message header. The request which determines the authentication scheme on the server is skipped. This is why you have to add some auth scope in the client request factory so Citrus can setup an authentication cache within the Http context in order to have preemtive authentication.

See the following example request that is created by the Citrus message sender and have a look at the *Authorization* header which gets automatically set for all requests.

```
POST /test HTTP/1.1
Accept: text/xml, */*
Content-Type: text/xml
Accept-Charset: iso-8859-1, us-ascii, utf-8
Authorization: Basic c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA==
User-Agent: Jakarta Commons-HttpClient/3.1
Host: localhost:8080
Content-Length: 175
<testRequestMessage>
  <text>Hello HttpServer</text>
</testRequestMessage>
```

Tip

Of course you can also set the *Authorization* header on your own for each request in your send action definition. Be aware of using the correct basic authentication header syntax with base64 encoding for the username:password phrase.

```
<header>
    <element name="Authorization" value="Basic c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA==" />
</header>
```

For base64 encoding you can also use a Citrus function, see Section 20.23, “citrus:encodeBase64()”

16.6. Server basic authentication

Citrus as a server can also set basic authentication so clients need to authenticate properly when accessing server resources.

```
<citrus-http:server id="basicAuthHttpServer"
                port="8090"
                auto-start="true"
                context-config-location="classpath:com/consol/citrus/http/citrus-servlet-context.xml"
                resource-base="src/citrus/resources"
                security-handler="basicSecurityHandler" />

<bean id="securityHandler" class="com.consol.citrus.http.security.SecurityHandlerFactory">
    <property name="users">
        <list>
            <bean class="com.consol.citrus.http.security.User">
                <property name="name" value="citrus" />
                <property name="password" value="secret" />
                <property name="roles" value="CitrusRole" />
            </bean>
        </list>
    </property>
    <property name="constraints">
        <map>
            <entry key="/foo/*">
                <bean class="com.consol.citrus.http.security.BasicAuthConstraint">
                    <constructor-arg value="CitrusRole" />
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

We have set a security handler on the server web container with a constraint on all resources with */foo/**. Following from that the server requires basic authentication for these resources. The granted users and roles are specified within the security handler bean definition. Connecting clients have to set the basic auth Http header properly using the correct user and role for accessing the Citrus server now.

You can customize the security handler for your very specific needs (e.g. load users and roles with JDBC from a database). Just have a look at the code base and inspect the settings and properties offered by the security handler interface.

Tip

This mechanism is not restricted to basic authentication only. With other settings you can also set up digest or form-based authentication constraints very easy.

Chapter 17. SSH support

In the spirit of other Citrus mock services, there is support for simulating an external SSH server as well as for connecting to SSH servers as a client during the test execution. Citrus translates SSH requests and responses to simple XML documents for better validation with the common Citrus mechanisms.

This means that the Citrus test case does not deal with pure SSH protocol commands. Instead of this we use the powerful XML validation capabilities in Citrus when dealing with the simple XML documents that represent the SSH request/response data.

Let us clarify this with a little example. Once the real SSH server daemon is fired up within Citrus we accept a SSH EXEC request for instance. The request is translated into a XML message of the following format:

```
<ssh-request>
  <command>cat - | sed -e 's/Hello/Hello SSH/'</command>
  <stdin>Hello World</stdin>
</ssh-request>
```

This message can be validated with the usual Citrus mechanism in a receive test action. If you do not know how to do this, please read one of the sections about XML message validation in this reference guide first. Now after having received this request message the respective SSH response should be provided as appropriate answer. This is done with a message sending action on a reply handler as it is known from synchronous http message communication in Citrus for instance. The SSH XML representation of a response message looks like this:

```
<ssh-response>
  <stdout>Hello SSH World</stdout>
  <stderr></stderr>
  <exit>0</exit>
</ssh-response>
```

Besides simulating a full featured SSH server, Citrus also provides SSH client functionality. This client uses the same request message pattern, which is translated into a real SSH call to an SSH server. The SSH response received is also translated into a XML message as shown above so we can validate it with known validation mechanisms in Citrus.

Similar to the other Citrus modules (http, soap), a Citrus SSH server and client is configured in Citrus Spring application context. There is a dedicated *ssh* namespace available for all ssh Citrus components. The namespace declaration goes into the context top-level element as usual:

```
<beans
    [...]
    xmlns:citrus-ssh="http://www.citrusframework.org/schema/ssh/config"
    [...]
    xsi:schemaLocation="
        [...]
        http://www.citrusframework.org/schema/ssh/config
        http://www.citrusframework.org/schema/ssh/config/citrus-ssh-config.xsd
        [...] ">
   [...]
</beans>
```

Both, SSH server and client along with their configuration options are described in the following two sections.

17.1. SSH Client

A Citrus SSH client is useful for testing against a real SSH server. So Citrus is able to invoke SSH commands on the external server and validate the SSH response accordingly. The test case does not deal with the pure SSH protocol within this communication. The Citrus SSH client component expects a customized XML representation and automatically translates these request messages into a real SSH call to a specific host. Once the synchronous SSH response was received the result gets translated back to the XML response message representation. On this translated response we can easily apply the validation steps by the usual Citrus means.

The SSH client components receive its configuration in the Spring application context as usual. We can use the special SSH module namespace for easy configuration:

```
<citrus-ssh:client id="sshClientSender"
           port="9072"
           user="roland"
           private-key-path="classpath:com/consol/citrus/ssh/test_user.priv"
           strict-host-checking="false"
           host="localhost"
           reply-handler="sshResponseReceiver"/>
```

The SSH client receives several attributes, these are:

- *id:* Id identifying the bean and used as reference from with test descriptions. (e.g. id="sshClient")
- *host:* Host to connect to for sending an SSH Exec request. Default is 'localhost' (e.g. host="localhost")
- *port* Port to use. Default is 2222 (e.g. port="9072")
- *private-key-path:* Path to a private key, which can be either a plain file path or an class resource if prefixed with 'classpath' (e.g. private-key-path="classpath:test_user.priv")
- *private-key-password:* Optional password for the private key (e.g. password="s!cr!t")
- *user:* User used for connecting to the SSH server (e.g. user="roland")
- *password:* Password used for password based authentication. Might be combined with "private-key-path" in which case both authentication mechanism are tried (e.g. password="ps!st)
- *strict-host-checking:* Whether the host key should be verified by looking it up in a 'known_hosts' file. Default is false (e.g. strict-host-checking="true")
- *known-hosts-path:* Path to a known hosts file. If prefixed with 'classpath:' this file is looked up as a resource in the classpath (e.g. known-hosts-path="/etc/ssh/known_hosts")
- *command-timeout:* Timeout in milliseconds for how long to wait for the SSH command to complete. Default is 5 minutes (e.g. command-timeout="300000")
- *connection-timeout:* Timeout in milliseconds for how long to for a connectiuon to connect. Default is 1 minute (e.g. connection-timeout="60000")
- *reply-handler:* Reference to reply handler which receives the SSH answer as a message. This reply handler should be specified with *<citrus-ssh:reply-handler>*. The reply handler follows the typical Citrus semantics. (e.g. reply-handler="sshResponseReceiver")

- *actor*: Actor used for switching groups of actions (e.g. actor="ssh-mock")

Once defines as client component in the Spring application context test cases can reference the client in every send test action.

```
<send with="sshClientSender">
  <message>
    <data>
      <![CDATA[
          <ssh-request>
            <command>shutdown</command>
            <stdin>input</stdin>
          </ssh-request>
      ]]>
    </data>
  </message>
</send>

<receive with="sshResponseReceiver">
  <message>
    <data>
      <![CDATA[
          <ssh-response>
              <stdout>Hello Citrus</stdout>
              <stderr/>
              <exit>0</exit>
          </ssh-response>
      ]]>
    </data>
  </message>
</receive>
```

As you can see we use usual send and receive test actions. The XML SSH representation helps us to specify the request and response data for validation. This way you can call SSH commands against an external SSH server and validate the response data.

17.2. SSH Server

Now that we have used Citrus on the client side we can also use Citrus SSH server module in order to provide a full stacked SSH server deamon. We can accept SSH client connections and provide proper response messages as an answer.

Given the above SSH module namespace declaration, adding a new SSH server is quite simple:

```
<citrus-ssh:server id="sshServer"
            allowed-key-path="classpath:com/consol/citrus/ssh/test_user_pub.pem"
            user="roland"
            port="9072"
            auto-start="true"
            message-handler-ref="sshMessageHandler"/>
```

message-handler-ref is the handler which receives the SSH request as messages (in the request format described above). Message handler implementations are fully described in Section 16.2, “Http server” All message handlers described there are supported in SSH server module, too.

The *<citrus-ssh:server>* supports the following attributes:

SSH Server Attributes:

- *id:* Name of the SSH server which identifies it unique within the Citrus Spring context (e.g. id="sshServer")

- *host-key-path:* Path to PEM encoded key pair (public and private key) which is used as host key. By default, a standard, pre-generate, fixed keypair is used. The path can be specified either as an file path, or, if prefixed with *classpath:* is looked up from within the classpath. The path the is relative from to the top-level package, so no leading slash should be used (e.g. hist-key-path="/etc/citrus_ssh_server.pem)
- *user:* User which is allowed to connect (e.g. user="roland")
- *allowed-key-path:* Path to a SSH public key stored in PEM format. These are the keys, which are allowed to connect to the SSH server when publickey authentication is used. It seves the same purpose as `authorized_keys` for standard SSH installations. The path can be specified either as an file path, or, if prefixed with *classpath:* is looked up from within the classpath. The path the is relative from to the top-level package, so no leading slash should be used (e.g. allowed-key-path="classpath:test_user_pub.pem)
- *password:* Password which should be used when password authentication is used. Both publickey authentication and password based authentication can be used together in which case both methods are tried in turn (e.g. password="s!cr!t")
- *host:* Host address (e.g. localhost)
- *port:* Port on which to listen. The SSH server will bind on localhost to this port (e.g. port="9072")
- *auto-start:* Whether to start this SSH server automatically. Default is *true*. If set to *false*, a test action is responsible for starting/stopping the server (e.g. auto-start="true")
- *message-handler-ref:* Bean reference to a message handler which processes the incoming SSH request. The message format for the request and response are described above (e.g. message-handler-ref="sshMessageHandler")

Once the SSH server component is added to the Spring application context with a proper message handler like the MessageChannel forwarding message handler we can receive incoming requests in a test case and provide a respone message for the client.

```
<receive with="sshRequestReceiver">
  <message>
    <data>
      <![CDATA[
          <ssh-request>
             <command>shutdown</command>
             <stdin>input</stdin>
          </ssh-request>
       ]]>
    </data>
  </message>
</receive>

<send with="sshResponseSender">
  <message>
    <data>
      <![CDATA[
          <ssh-response>
              <stdout>Hello Citrus</stdout>
              <exit>0</exit>
          </ssh-response>
        ]]>
    </data>
  </message>
</send>
```

Chapter 18. 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.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.

18.1. 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"
                                 polling-interval="1000"/>
```

SOAP WebServices usually use synchronous communication as we use the Http transport protocol. Following from that we need a reply message handler. The message sender component uses the *request-url* and calls the WebService as a client. The sender will automatically build a SOAP request message including a SOAP header and the message payload as SOAP body. This means that you as a tester do not care about SOAP envelope specific logic in the test case. The SOAP message components do add SOAP elements automatically. As soon as the SOAP response arrives it is passed to the given reply handler. In detail the reply handler polls several times for the reply message while the message sender is waiting. The poll interval is an optional setting so you can manage the frequency of message handshake attempts. In case no response message is available in time according to the timeout settings we raise a timeout error and the test will fail.

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"/>
```

Tip

Up to now we have used a static endpoint request url for the SOAP message sender. Besides that we can use dynamic endpoint uri in configuration. We just use an endpoint resolver bean instead of the static request url like this:

```
<citrus-ws:message-sender id="soapMessageSender"
            endpoint-resolver="dynamicEndpointResolver"
            reply-handler="soapResponseHandler"
            message-factory="mySepcialMessageFactory"/>

<bean id="dynamicEndpointResolver"
       class="com.consol.citrus.adapter.common.endpoint.MessageHeaderEndpointUriResolver"/>
```

The *dynamicEndpointResolver* bean must implement the EndpointUriResolver interface in order to resolve dynamic endpoint uri values. Citrus offers a default implementation, the *MessageHeaderEndpointUriResolver*, which uses a specific message header for setting dynamic endpoint uri. The message header needs to specify the header *citrus_endpoint_uri* with a valid request uri.

```
<header>
  <element name="citrus_endpoint_uri"
           value="http://localhost:${port}/${context}" />
</header>
```

18.2. 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 start automatically during application startup. In the example above the server is listening for requests on port `8091`. This is the standard connector configuration for the Jetty

server. For detailed customization the Citrus Jetty server configuration also supports explicit connector configurations (@connector and @connectors attributes). For more information please see the Jetty Connector implementations.

Now let us have a closer look at the context-config-location attribute. This configuration defines the Spring application context file for endpoints, request mappings and other SpringWS specific information. Please see the official SpringWS documentation for details on this Spring based configuration. You can also just copy the following example application context which should work for you in general.

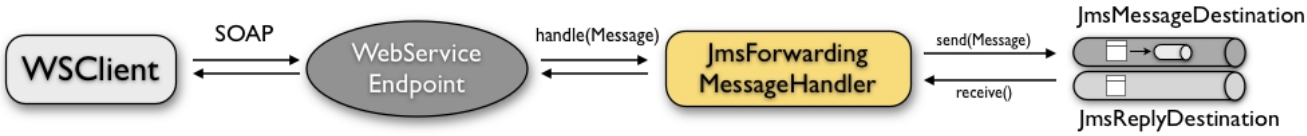
```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd">

  <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 SpringWS request mapping with endpoint configurations. The mapping is responsible to forward incoming requests to the endpoint which will

handle the request and provide a proper response message. 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 16, *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 16, *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.

18.3. SOAP headers

SOAP defines several header variations that we discuss in the following sections. First of all we deal with the special *SOAP action* header. In case we need to set this SOAP action header 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 SOAP action 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>
```

This completes the SOAP header possibilities for sending SOAP messages with Citrus. Of course you can also use these variants in SOAP message header validation. You define expected SOAP headers, SOAP action and XML fragments and Citrus will match incoming request to that. Just use *citrus_soap_action* header key in your receiving message action and you validate this SOAP header accordingly.

When validating SOAP header XML fragments you need to define the whole XML header fragment as expected header data like this:

```
<receive with="soapMessageHandler">
    <message>
        <data><![CDATA[
                <c:ResponseMessage xmlns:c="http://citrusframework.org/schema">
                  <c:resultCode>OK</c:resultCode>
                </c:ResponseMessage>
        ]]></data>
    </message>
    <header>
        <data>
            <![CDATA[
                <SOAP-ENV:Header
                    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
                    <c:customHeader xmlns:c="http://citrusframework.org/headerschema">
                        <c:correlationId>${correlationId}</c:correlationId>
                        <c:applicationId>${applicationId}</c:applicationId>
                        <c:trackingId>${trackingId}</c:trackingId>
                        <c:serviceId>${serviceId}</c:serviceId>
                        <c:interfaceVersion>1.0</c:interfaceVersion>
                        <c:timestamp>@ignore@</c:timestamp>
                    </c:customHeader>
                </SOAP-ENV:Header>
            ]]>
        </data>
        <element name="citrus_soap_action" value="doResponse"/>
    </header>
</receive>
```

As you see the SOAP XML header validation can combine header element and XML fragement validation. This is also likely to be used when dealing with WS-Security message headers.

18.4. Http mime headers

Besides the SOAP specific header elements the Http mime headers (e.g. Content-Type, Content-Length, Authorization) might be candidates for validation, too. When using Http as transport layer the SOAP message may define those mime headers. The tester is able to send and validate these headers inside the test case, although these Http headers are located outside of the SOAP envelope. Let us first of all speak about validating the Http mime headers. This feature is not enabled by default. We have to set a special flag in our SOAP WebService endpoint.

```
<bean id="webServiceEndpoint" class="com.consol.citrus.ws.WebServiceEndpoint">
    <property name="handleMimeHeaders" value="true"/>
    [...]
</bean>
```

With this configuration Citrus will handle all available mime headers passing those to the test case for normal header validation.

```
<ws:receive with="webServiceRequestReceiver">
    <message>
        <data>
            <![CDATA[
                <ns0:SoapMessageRequest xmlns:ns0="http://www.consol.de/schemas/sample.xsd">
                    <ns0:Operation>Validate mime headers</ns0:Operation>
                </ns0:SoapMessageRequest>
            ]]>
        </data>
    </message>
    <header>
        <element name="Content-Type" value="text/xml; charset=utf-8"/>
    </header>
</ws:receive>
```

As you can see the validation is quite simple as soon as we have enabled the mime header handling in the WebService endpoint. The transport (http) headers go into the header list just like the normal SOAP header elements do. So you can validate the headers as usual.

So much for receiving and validating Http messsage headers with SOAP communication. Now we want to send special mime headers on client side. We overwrite or add mime headers to our sending action. We mark some headers with following prefix `"citrus_http_"`. This tells the SOAP message sender to add these headers to the Http header section outside the SOAP envelope. Keep in mind that header elements without this prefix go right into the SOAP header section by default.

```
<ws:send with="webServiceRequestSender">
  [...]
  <header>
    <element name="citrus_http_operation" value="foo"/>
  </header>
  [...]
</ws:send>
```

The listing above defines a Http mime header *operation*. The header prefix `citrus_http_` is cut off before the header goes into the Http header section. With this feature we can decide where exactly our header information is located in our resulting client message.

18.5. 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, fault-actor 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.

18.5.1. 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-actor>SERVER</ws:fault-actor>
        <ws:fault-detail>
            <![CDATA[
                <ns0:FaultDetail xmlns:ns0="http://www.consol.de/schemas/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-actor and the fault-detail elements are optional. Same with the soap action declared in the special Citrus header `citrus_soap_action`. In the sample above the fault-detail data is placed inline as XML data. As an alternative to that 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/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.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">
```

18.5.2. 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. We can use a special assert implementation especially designed for SOAP faults in this case.

```
<ws:assert fault-code="{http://www.citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
           fault-actor="SERVER">
    <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 several attributes to specify the expected SOAP fault. Namely these are *"fault-code"*, *"fault-string"* and *"fault-actor"*. The *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*
- *com.consol.citrus.ws.validation.SimpleSoapFaultValidator*
- *com.consol.citrus.ws.validation.XmlSoapFaultValidator*

Please see the API documentation for details on the available reference implementations. Of course you can also define your own SOAP validator logic (would be great if you could share your ideas!). 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.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 defined in the *citrus-context.xml* is responsible for checking the SOAP fault detail with validation algorithm. The validator implementation checks the detail content to meet the expected template. Citrus provides some default SoapFaultValidator implementations. Supported algorithms are pure String comparison (*com.consol.citrus.ws.validation.SimpleSoapFaultValidator*) as well as XML tree walk-through (com.consol.citrus.ws.validation.XmlSoapFaultValidator).

When using the XML validation algorithm you have the complete power as known from normal message validation in receive actions. This includes schema validation or ignoring elements for instance. On the fault-detail element you are able to add some validation settings such as *schema-validation=enabled/disabled*, custom *schema-repository* and so on.

```
<ws:assert fault-code="{http://www.citrusframework.org/faults}TEC-1001"
           fault-string="Invalid request">
    <ws:fault-detail schema-validation="false">
      <![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">
        [...]
    </send>
```

```
</ws:assert>
```

Please see also the Citrus API documentation for available validator implementations and validation algorithms.

So far we used assert action wrapper in order to catch SOAP fault exceptions and validate the SOAP fault content. Now we have an alternative way of handling SOAP faults in Citrus. With exceptions the send action aborts and we do not have a receive action for the SOAP fault. This might be inadequate if we need to validate the SOAP message content (SOAPHeader and SOAPBody) coming with the SOAP fault. Therefore the web service message sender component offers several fault strategy options. In the following we discuss the propagation of SOAP fault as messages to the reply message handler as we would do with normal SOAP messages.

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

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

We have configured a fault strategy *propagateError* so the message sender will not raise client exceptions but inform the reply message handler with SOAP fault message contents. By default the fault strategy raises client exceptions (fault-strategy=*throwsException*).

So now that we do not raise exceptions we can leave out the assert action wrapper in our test. Instead we simply use a receive action and validate the SOAP fault like this.

```
<send with="soapMessageSender">
    <message>
        <data>
            <![CDATA[
                <ns0:SoapFaultForcingRequest xmlns:ns0="http://www.consol.de/schemas/sample.xsd">
                    <ns0:Message>This is invalid</ns0:Message>
                </ns0:SoapFaultForcingRequest>
            ]]>
        </data>
    </message>
</send>

<receive with="soapResponseHandler" timeout="5000">
    <message>
        <data>
            <![CDATA[
                <SOAP-ENV:Fault xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
                    <faultcode xmlns:CITRUS="http://citrus.org/soap">CITRUS:${soapFaultCode}</faultcode>
                    <faultstring xml:lang="en">${soapFaultString}</faultstring>
                </SOAP-ENV:Fault>
            ]]>
        </data>
    </message>
</receive>
```

So choose the prefered way of handling SOAP faults either by asserting client exceptions or propagating fault messages to reply message handler receive action.

18.5.3. Multiple SOAP fault details

SOAP fault details can hold one or more SOAP fault detail elements. In the previous sections we have used SOAP fault details in sending and receiving actions as single element. In order to meet the SOAP specification Citrus is also able to handle multiple SOAP fault detail elements in a message. You just use many fault-detail elements in your test action like this:

```
<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-actor>SERVER</ws:fault-actor>
        <ws:fault-detail>
            <![CDATA[
                <ns0:FaultDetail xmlns:ns0="http://www.consol.de/schemas/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-detail>
            <![CDATA[
                <ns0:ErrorDetail xmlns:ns0="http://www.consol.de/schemas/sayHello.xsd">
                    <ns0:ErrorCode>TEC-1000</ns0:ErrorCode>
                </ns0:ErrorDetail>
            ]]>
        </ws:fault-detail>
    </ws:fault>
    <ws:header>
        <ws:element name="citrus_soap_action" value="sayHello"/>
    </ws:header>
</ws:send-fault>
```

This will result in following SOAP envelope message:

```
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/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>
                <ns0:ErrorDetail xmlns:ns0="http://www.consol.de/schemas/sayHello.xsd">
                    <ns0:ErrorCode>TEC-1000</ns0:ErrorCode>
                </ns0:ErrorDetail>
            </detail>
        </SOAP-ENV:Fault>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Of course we can also expect several fault detail elements when receiving a SOAP fault.

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

```
    <send with="webServiceHelloRequestSender">
        [...]
    </send>
</ws:assert>
```

As you can see we can individually use validation settings for each fault detail. In the example above we disabled schema validation for the first fault detail element.

18.6. Simulate Http error codes with SOAP

The SOAP server logic on Citrus is able to simulate pure Http error codes such as 404 "Not found" or 500 "Internal server error". The good thing is that the Citrus server is able to receive a request for proper validation in a receive action and then simulate Http errors on demand.

The mechanism on Http error code simulation is not different to the usual SOAP reuest/response handling in Citrus. We receive the request as usual and we provide a response with a reply message sender. The Http error situation is simulated according to the special Http header *citrus_http_status* in the Citrus SOAP response definition. In case this header is set to a value other than 200 OK the Citrus SOAP server sends an empty SOAP response with Http error status code set accordingly.

```
<receive with="soapMessageReceiver">
  <message>
      <data>
          <![CDATA[
              <ns0:Message xmlns:ns0="http://consol.de/schemas/sample.xsd">
                  <ns0:Text>Hello SOAP server</ns0:Text>
              </ns0:Message>
          ]]>
      </data>
  </message>
</receive>

<send with="soapResponseSender">
    <message>
        <data></data>
    </message>
    <header>
        <element name="citrus_http_status_code" value="500"/>
    </header>
</send>
```

The SOAP response must be empty and the Http status code is set to a value other than 200, like 500. This results in a Http error sent to the calling client with error 500 "Internal server error".

18.7. 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.

18.7.1. 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://consol.de/schemas/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 which 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 fulfill the needs of almost every application.

18.7.2. 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://consol.de/schemas/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="soapAttachmentValidator" class="com.consol.citrus.ws.validation.SimpleSoapAttachmentValidator"/>
<bean id="mySoapAttachmentValidator" class="com.company.ws.validation.MySoapAttachmentValidator"/>
```

You can define several validator instances 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 18.2, “SOAP message receiver”. The SOAP attachment validation in its current 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.

18.8. Basic authentication with SOAP

As a SOAP client you often have to use basic authentication in order to access a service. Basic authentication via Http stands for username/password authentication where the credentials are transmtted in the Http request header section as base64 encoded entry. As Citrus uses the WebService stack of the Spring framework we have to setup basic authentication there. We set the

user credentials on the HttpClient message sender which is used inside Spring's WebServivceTemplate.

Citrus provides a comfortable way to set the Http message sender with basic authentication credentials on the WebServiceTemplate. Just see the following example and learn how to do that.

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

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

<bean id="basicAuthMessageSender" class="org.springframework.ws.transport.http.HttpComponentsMessageSender">
  <property name="authScope">
      <bean class="org.apache.http.auth.AuthScope">
        <constructor-arg value="localhost"/>
        <constructor-arg value="8090"/>
        <constructor-arg value=""/>
        <constructor-arg value="basic"/>
      </bean>
  </property>
  <property name="credentials">
    <bean class="org.apache.http.auth.UsernamePasswordCredentials">
        <constructor-arg value="someUsername"/>
        <constructor-arg value="somePassword"/>
    </bean>
  </property>
</bean>
```

The above configuration results in SOAP requests with authentication headers properly set for basic authentication. The special message sender takes care on adding the proper basic authentication header to each request that is sent with this Citrus message sender. By default preemtive authentication is used. The message sender only sends a single request to the server with all authentication information set in the message header. The request which determines the authentication scheme on the server is skipped. This is why you have to add some auth scope so Citrus can setup an authentication cache within the Http context in order to have preemtive authentication.

Tip

You can also skip the message sender configuration and set the *Authorization* header on each request in your send action definition on your own. Be aware of setting the header as Http mime header using the correct prefix and take care on using the correct basic authentication with base64 encoding for the username:password phrase.

```
<header>
    <element name="citrus_http_Authorization" value="Basic c29tZVVzZXJuYW1lOnNvbWVQYXNzd29yZA=="/>
</header>
```

For base64 encoding you can also use a Citrus function, see Section 20.23, “citrus:encodeBase64()”

18.9. Basic authentication on the server

When providing SOAP WebService server functionality Citrus can also set basic authentication so clients need to authenticate properly when accessing the server.

```
<citrus-ws:jetty-server id="simpleJettyServer"
```

```
            port="8091"
            auto-start="true"
            context-config-location="classpath:citrus-ws-servlet.xml"
            resource-base="src/citrus/resources"
            security-handler="basicSecurityHandler"/>

<bean id="securityHandler" class="com.consol.citrus.ws.security.SecurityHandlerFactory">
    <property name="users">
        <list>
            <bean class="com.consol.citrus.ws.security.User">
                <property name="name" value="citrus"/>
                <property name="password" value="secret"/>
                <property name="roles" value="CitrusRole"/>
            </bean>
        </list>
    </property>
    <property name="constraints">
        <map>
            <entry key="/foo/*">
                <bean class="com.consol.citrus.ws.security.BasicAuthConstraint">
                    <constructor-arg value="CitrusRole"/>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

We have set a security handler on the server web container with a constraint on all resources with */foo/**. Following from that the server requires basic authentication for these resources. The granted users and roles are specified within the security handler bean definition. Connecting clients have to set the basic auth Http header properly using the correct user and role for accessing the Citrus server now.

You can customize the security handler for your very specific needs (e.g. load users and roles with JDBC from a database). Just have a look at the code base and inspect the settings and properties offered by the security handler interface.

Tip

This mechanism is not restricted to basic authentication only. With other settings you can also set up digest or form-based authentication constraints very easy.

18.10. WS-Addressing support

The web service stack offers a lot of different technologies and standards within the context of WebServices. Some people speak of WS-* specifications. One of these specifications deals with addressing WebServices. On client side you may add wsa header information to the request in order to give the server instructions how to deal with SOAP faults for instance.

In Citrus WebServiceMessageSender you can add those header information using the common configuration like this:

```
<citrus-ws:message-sender id="soapMessageSender"
                          request-url="http://localhost:8090/test"
                          reply-handler="soapResponseHandler"
                          addressing-headers="wsAddressing200408"/>

<bean id="wsAddressing200408" class="com.consol.citrus.ws.addressing.WsAddressingHeaders">
    <property name="version" value="VERSION200408"/>
    <property name="action"
                value="http://citrus.sample/sayHello"/>
    <property name="to"
                value="http://citrus.sample/server"/>
    <property name="from">
```

```
        <bean class="org.springframework.ws.soap.addressing.core.EndpointReference">
            <constructor-arg value="http://citrus.sample/client"/>
        </bean>
    </property>
    <property name="replyTo">
        <bean class="org.springframework.ws.soap.addressing.core.EndpointReference">
            <constructor-arg value="http://citrus.sample/client"/>
        </bean>
    </property>
    <property name="faultTo">
        <bean class="org.springframework.ws.soap.addressing.core.EndpointReference">
            <constructor-arg value="http://citrus.sample/fault/resolver"/>
        </bean>
    </property>
</bean>
```

Note

The WS-Addressing specification knows several versions. Supported version are VERSION10 (WS-Addressing 1.0 May 2006) and VERSION200408 (August 2004 edition of the WS-Addressing specification).

The addressing headers find a place in the SOAP message header with respective namespaces and values. A possible SOAP request with ws addressing headers looks like follows:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">
        <wsa:To SOAP-ENV:mustUnderstand="1">http://citrus.sample/server</wsa:To>
        <wsa:From>
            <wsa:Address>http://citrus.sample/client</wsa:Address>
        </wsa:From>
        <wsa:ReplyTo>
            <wsa:Address>http://citrus.sample/client</wsa:Address>
        </wsa:ReplyTo>
        <wsa:FaultTo>
            <wsa:Address>http://citrus.sample/fault/resolver</wsa:Address>
        </wsa:FaultTo>
        <wsa:Action>http://citrus.sample/sayHello</wsa:Action>
        <wsa:MessageID>urn:uuid:4c4d8af2-b402-4bc0-a2e3-ad33b910e394</wsa:MessageID>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <cit:HelloRequest xmlns:cit="http://citrus/sample/sayHello">
            <cit:Text>Hello Citrus!</cit:Text>
        </cit:HelloRequest>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Important

The message id property is automatically generated for each request. If you need to set a static message id you can do so in citrus-context message sender configuration.

18.11. Synchronous SOAP fork mode

SOAP over HTTP uses synchronous communictaion by nature. This means that sending a SOAP message in Citrus over HTTP will automatically block further test actions until the synchronous HTTP response has been received. In test cases this synchronous blocking might cause problems for several reasons. A simple reason would be that you need to do further test actions in parallel to the synchronous HTTP SOAP communication (e.g. simulate another backend system in the test case).

You can separate the SOAP send action from the rest of the test case by using the *"fork"* mode. The

SOAP message sender will automatically open a new Java Thread for the synchronous communication and the test is able to continue with execution although the synchronous HTTP SOAP response has not arrived yet.

```
<ws:send with="webServiceRequestSender" fork="true">
  <message>
    <data>
      <![CDATA[
        <ns0:SoapMessageWithAttachmentRequest xmlns:ns0="http://www.consol.de/schemas/sample.xsd">
          <ns0:Operation>Read the attachment</ns0:Operation>
        </ns0:SoapMessageWithAttachmentRequest>
      ]]>
    </data>
  </message>
</ws:send>
```

With the *"fork"* mode enabled the test continues with execution while the sending action waits for the synchronous response in a separate Java Thread. You could reach the same behaviour with a complex <parallel>/<sequential> container construct, but forking the send action is much more straight forward.

Important

It is highly recommended to use a proper *"timeout"* setting on the SOAP receive action when using fork mode. The forked send operation might take some time and the corresponding receive action might run into failure as the response was has not been received yet. The result would be a broken test because of the missing response message. A proper *"timeout"* setting for the receive action solves this problem as the action waits for this time period and occasionally repeatedly asks for the SOAP response message. The following listing sets the receive timeout to 10 seconds, so the action waits for the forked send action to deliver the SOAP response in time.

```
<ws:receive with="webServiceReplyHandler" timeout="10000">
  <message>
    <data>
      <![CDATA[
        <ns0:SoapMessageWithAttachmentResponse xmlns:ns0="http://www.consol.de/schemas/sample.xsd">
          <ns0:Operation>Read the attachment</ns0:Operation>
          <ns0:Success>true</ns0:Success>
        </ns0:SoapMessageWithAttachmentResponse>
      ]]>
    </data>
  </message>
</ws:receive>
```

Chapter 19. 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 15, *JMS support*. You always have to include this configuration namespace in order to use the Citrus configuration elements.

19.1. 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.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.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.

19.2. 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"/>
```

19.3. 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"
                                      polling-interval="1000"/>
```

Synchronous message channel senders usually go with a reply message handler which is responsible for processing the reply messages. While the synchronous message sender component is waiting for the reply message to arrive the reply handler polls for the reply message several times. The poll interval is an optional setting in order to manage the amount of reply message handshake attempts. Once the sender was able to receive the reply message synchronously the reply handler takes over with further steps such as message validation. In case all message handshake attempts do fail because the reply message is not available in time we raise some timeout error and the test will fail.

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.

19.4. 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 Citrus 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.

19.5. Message selectors on channels

Unfortunately Spring Integration message channels do not support message selectors on header values as described in Section 7.2.7, “Message selectors”. With Citrus version 1.2 we found a way to also add message selector support on message channels. We had to introduce a special queue message channel implementation. So first of all we use this new message channel implementation in our configuration.

```
<citrus:message-channel id="orders" capacity="5"/>
```

The Citrus message channel implementation extends the queue channel implementation from Spring Integration. So we can add a capacity attribute for this channel. That's it! Now we use the message channel that supports message selection. In our test we define message selectors on header values as described in Section 7.2.7, “Message selectors” and you will see that it works.

In addition to that we have implemented other message filter possibilities on message channels that we discuss in the next sections.

19.5.1. Root QName Message Selector

You can use the XML root QName of your message as selection criteria. Let's see how this works in a small example:

We have two different XML messages on a message channel waiting to be picked up by a consumer.

```
<HelloMessage xmlns="http://citrusframework.org/schema">Hello Citrus</HelloMessage>
<GoodbyeMessage xmlns="http://citrusframework.org/schema">Goodbye Citrus</GoodbyeMessage>
```

We would like to pick up the *GoodbyeMessage* in our test case. The *HelloMessage* should be left on the message channel as we are not interested in it right now. We can define a root qname message selector in the receive action like this:

```
<receive with="channelReceiver">
    <selector>
        <element name="root-qname" value="GoodbyeMessage"/>
    </selector>
    <message>
        <payload>
            <GoodbyeMessage xmlns="http://citrusframework.org/schema">Goodbye Citrus</GoodbyeMessage>
        </payload>
    </message>
</receive>
```

The Citrus receiver picks up the *GoodbyeMessage* from the channel selected via the root qname of the XML message payload. Of course you can also combine message header selectors and root qname selectors as shown in this example below where a message header *sequenceId* is added to the selection logic.

```
<selector>
    <element name="root-qname" value="GoodbyeMessage"/>
    <element name="sequenceId" value="1234"/>
</selector>
```

As we deal with XML qname values, we can also use namespaces in our selector root qname selection.

```
<selector>
    <element name="root-qname" value="{http://citrusframework.org/schema}GoodbyeMessage"/>
</selector>
```

19.5.2. XPath Evaluating Message Selector

It is also possible to evaluate some XPath expression on the message payload in order to select a message from a message channel. The XPath expression outcome must match an expected value and only then the message is consumed form the channel.

The syntax for the XPath expression is to be defined as the element name like this:

```
<selector>
    <element name="xpath://Order/status" value="pending"/>
</selector>
```

The message selector looks for order messages with *status="pending"* in the message payload. This means that following messages would get accepted/declined by the message selector.

```
<Order><status>pending</status></Order> = ACCEPTED
<Order><status>finished</status></Order> = NOT ACCEPTED
```

Of course you can also use XML namespaces in your XPath expressions when selecting messages from channels.

```
<selector>
    <element name="xpath://ns1:Order/ns1:status" value="pending"/>
</selector>
```

Namespace prefixes must match the incoming message - otherwise the XPath expression will not work as expected. In our example the message should look like this:

```
<ns1:Order xmlns:ns1="http://citrus.org/schema"><ns1:status>pending</ns1:status></ns1:Order>
```

Knowing the correct XML namespace prefix is not always easy. If you are not sure which namespace prefix to choose Citrus ships with a dynamic namespace replacement for XPath expressions. The XPath expression looks like this and is most flexible:

```
<selector>
    <element name="xpath://{http://citrus.org/schema}:Order/{http://citrus.org/schema}:status"
                value="pending"/>
</selector>
```

This will match all incoming messages regardless the XML namespace prefix that is used.

19.6. 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.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.xsd
            http://www.springframework.org/schema/integration/file
            http://www.springframework.org/schema/integration/file/spring-integration-file.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 fixed-rate="100"/>
</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 20. 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.

20.1. 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>
```

20.2. 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

20.3. 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

20.4. 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

20.5. 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

20.6. 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

20.7. 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

20.8. citrus:floor()

This function will round down decimal number values.

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

Function output:

3.0

20.9. 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

20.10. 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 number should have exactly the amount of digits. 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="rndNumber2" value="citrus:randomNumber(10, false)"/>
    <variable name="rndNumber3" value="citrus:randomNumber(3, false)"/>
</variables>
```

Function output:

8954638765

5003485980

6387650

65

20.11. 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). The last parameter allows also digit characters in the generated string. By default digit charaters are not allowed.

```
<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)"/>
    <variable name="rndString4" value="citrus:randomString(10, MIXED, true)"/>
</variables>
```

Function output:

HrGHOdfAer

AgSSwedetG

JSDFUTTRKU

dtkhirtsuz

Vt567JkA32

20.12. 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 function 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()" />
```

20.13. 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

20.14. 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

20.15. 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

20.16. 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*

20.17. 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*

20.18. 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*

20.19. 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*

20.20. citrus:absolute()

The function will return the absolute number value.

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

abs = *3.0*

20.21. 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 is 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')" />
```

20.22. citrus:randomUUID()

The function will generate a random Java UUID.

```
<variable name="uuid" value="citrus:randomUUID()"/>
```

uuid = *98fbd7b0-832e-4b85-b9d2-e0113ee88356*

20.23. citrus:encodeBase64()

The function will encode a string to binary data using base64 hexadecimal encoding.

```
<variable name="encoded" value="citrus:encodeBase64('Hallo Testframework')"/>
```

encoded = *VGVzdCBGcmFtZXdvcms=*

20.24. citrus:decodeBase64()

The function will decode binary data to a character sequence using base64 hexadecimal decoding.

```
<variable name="decoded" value="citrus:encodeBase64('VGVzdCBGcmFtZXdvcms=')"/>
```

decoded = *Hallo Testframework*

20.25. citrus:escapeXml()

If you want to deal with escaped XML in your test case you may want to use this function. It automatically escapes all XML special characters.

```
<echo>
    <message>
        <![CDATA[
            citrus:escapeXml('<Message>Hallo Test Framework</Message>')
        ]]>
    </message>
</echo>
```

<Message>Hallo Test Framework</Message>

20.26. citrus:cdataSection()

Usually we use CDATA sections to define message payload data inside a testcase. We might run into problems when the payload itself contains CDATA sections as nested CDATA sections are prohibited by XML nature. In this case the next function ships very usefull.

```
<variable name="cdata" value="citrus:cdataSection('payload')"/>
```

cdata = *<![CDATA[payload]]>*

20.27. citrus:digestAuthHeader()

Digest authentication is a commonly used security algorithm, especially in Http communication and SOAP WebServices. Citrus offers a function to generate a digest authentication principle used in the Http header section of a message.

```
<variable name="digest"
  value="citrus:digestAuthHeader('username', 'password', 'authRealm', 'acegi',
                            'POST', 'http://127.0.0.1:8080', 'citrus', 'md5')"/>
```

A possible digest authentication header value looks like this:

```
<Digest username=foo,realm=arealm,nonce=MTMzNT,
uri=http://127.0.0.1:8080,response=51f98c,opaque=b29a30,algorithm=md5>
```

You can use these digest headers in messages sent by Citrus like this:

```
<header>
  <element name="citrus_http_Authorization"
    value="vflig:digestAuthHeader('${username}','${password}','${authRealm}',
                            '${nonceKey}','POST','${uri}','${opaque}','${algorithm}')"/>
</header>
```

This will set a Http Authorization header with the respective digest in the request message. So your test is ready for client digest authentication.

20.28. citrus:localHostAddress()

Test cases may use the local host address for some reason (e.g. used as authentication principle). As the tests may run on different machines at the same time we can not use static host addresses. The provided function localHostAddress() reads the local host name dynamically at runtime.

```
<variable name="address" value="citrus:localHostAddress()"/>
```

A possible value is either the host name as used in DNS entry or an IP address value:

address = <*192.168.2.100*>

Chapter 21. Validation matchers

Message validation in Citrus is essential. The framework offers several validation mechanisms for different message types and formats. With test variables we are able to check for simple value equality. We ensure that message entries are equal to predefined expected values. Validation matchers add powerful assertion functionality on top of that. You just can use the predefined validation matcher functionalities in order to perform more complex assertions like *contains* or *isNumber* in your validation statements.

The following sections describe the Citrus default validation matcher implementations that are ready for usage. The matcher implementations should cover the basic assertions on character sequences and numbers. Of course you can add custom validation matcher implementations in order to meet your very specific validation assertions, too.

First of all let us have a look at a validation matcher statement in action so we understand how to use them in a test case.

```
<message>
    <data>
        <![CDATA[
        <RequestMessage>
            <MessageBody>
                <Customer>
                    <Id>@greaterThan(0)@</Id>
                    <Name>@equalsIgnoreCase('foo')@</Name>
                </Customer>
            </MessageBody>
        </RequestMessage>
        ]]>
    </data>
</message>
```

The listing above describes a normal message validation block inside a receive test action. We use some inline message payload template as CDATA. As you know Citrus will compare the actual message payload to this expected template in DOM tree comparison. In addition to that you can simply include validation matcher statements. The message element *Id* is automatically validated to be a number greater than zero and the *Name* character sequence is supposed to match 'foo' ignoring case spelling considerations.

Please note the special validation matcher syntax. The statements are surrounded with '@' markers and are identified by some unique name. The optional parameters passed to the matcher implementation state the expected values to match.

Tip

You can use validation matchers with all validation mechanisms - not only with XML validation. Plaintext, JSON, SQL result set validation are also supported.

See now the following sections describing the default validation matchers in Citrus.

21.1. matchesXml()

The XML validation matcher implementation is the possibly most exciting one, as we can validate nested XML with full validation power (e.g. ignoring elements, variable support). The matcher checks

a nested XML fragment to compare against expected XML. For instance we receive following XML message payload for validation:

```
<GetCustomerMessage>
      <CustomerDetails>
          <Id>5</Id>
          <Name>Christoph</Name>
          <Configuration><![CDATA[
            <config>
                <premium>true</premium>
                <last-login>2012-02-24T23:34:23</last-login>
                <link>http://www.citrusframework.org/customer/5</link>
            </config>
          ]]></Configuration>
      </CustomerDetails>
</GetCustomerMessage>
```

As you can see the message payload contains some configuration as nested XML data in a CDATA section. We could validate this CDATA section as static character sequence comparison, true. But the <last-login> timestamp changes its value continuously. This breaks the static validation for CDATA elements in XML. Fortunately the new XML validation matcher provides a solution for us:

```
<message>
    <data>
        <![CDATA[
            <GetCustomerMessage>
                  <CustomerDetails>
                      <Id>5</Id>
                      <Name>Christoph</Name>
                      <Configuration>citrus:cdataSection('@matchesXml('<config>
                            <premium>${isPremium}</premium>
                            <last-login>@ignore@</last-login>
                            <link>http://www.citrusframework.org/customer/5</link>
                        </config>')@')</Configuration>
                  </CustomerDetails>
            </GetCustomerMessage>
        ]]>
    </data>
</message>
```

With the validation matcher you are able to validate the nested XML with full validation power. Ignoring elements is possible and we can also use variables in our control XML.

Note

Nested CDATA elements within other CDATA sections are not allowed by XML standard. This is why we create the nested CDATA section on the fly with the function cdataSection().

21.2. equalsIgnoreCase()

This matcher implementation checks for equality without any case spelling considerations. The matcher expects a single parameter as the expected character sequence to check for.

```
<value>@equalsIgnoreCase('foo')@</value>
```

21.3. contains()

This matcher searches for a character sequence inside the actual value. If the character sequence is

not found somewhere the matcher starts complaining.

```
<value>@contains('foo')@</value>
```

The validation matcher also exist in a case insensitive variant.

```
<value>@containsIgnoreCase('foo')@</value>
```

21.4. startsWith()

The matcher implementation asserts that the given value starts with a character sequence otherwise the matcher will arise some error.

```
<value>@startsWith('foo')@</value>
```

21.5. endsWith()

Ends with matcher validates a value to end with a given character sequence.

```
<value>@endsWith('foo')@</value>
```

21.6. matches()

You can check a value to meet a regular expression with this validation matcher. This is for instance very useful for email address validation.

```
<value>@matches('[a-z0-9]')@</value>
```

21.7. matchesDatePattern()

Date values are always difficult to check for equality. Especially when you have millisecond timestamps to deal with. Therefore the date pattern validation matcher should have some improvement for you. You simply validate the date format pattern instead of checking for total equality.

```
<value>@matchesDatePattern('yyyy-MM-dd')@</value>
```

The example listing uses a date format pattern that is expected. The actual date value is parsed according to this pattern and may cause errors in case the value is no valid date matching the desired format.

21.8. isNumber()

Checking on values to be of numeric nature is essential. The actual value must be a numeric number otherwise the matcher raises errors. The matcher implementation does not evaluate any parameters.

```
<value>@isNumber()@</value>
```

21.9. lowerThan()

This matcher checks a number to be lower than a given threshold value.

```
<value>@lowerThan(5)@</value>
```

21.10. greaterThan()

The matcher implementation will check on numeric values to be greater than a minimum value.

```
<value>@greaterThan(5)@</value>
```

Chapter 22. Actions before/after the test run

A test framework should also provide the functionality to do some work before and after the test run. You could think of preparing/deleting the data in a database or starting/stopping a server in this section before/after a test run. These tasks fit best into the initialization and cleanup phases of Citrus. All you have to do is add some Spring bean definitions to the Citrus application context.

22.1. Before suite

You can influence the behavior of a test run in the initialization phase actually before the tests are executed. See the next code example to find out how it works with actions that take place before the first test is executed:

```
<bean class="com.consol.citrus.container.SequenceBeforeSuite">
      <property name="actions">
          <list>
              <!-- list of actions before suite -->
          </list>
      </property>
  </bean>
```

```
<bean class="com.consol.citrus.container.SequenceBeforeSuite">
      <property name="actions">
          <list>
              <bean id="initDatabase" class="com.consol.citrus.actions.ExecuteSQLAction">
                  <property name="dataSource" ref="testDataSource"/>
                  <property name="statements">
                      <list>
                          <value>CREATE TABLE PERSON (ID integer, NAME char(250))</value>
                      </list>
                  </property>
              </bean>
          </list>
      </property>
  </bean>
```

We access the database and create a table PERSON which is obviously needed in our test cases. You can think of several actions here to prepare the database for instance.

Tip

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

22.2. After suite

A test 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 after the test run in order to avoid errors in follow-up test runs. Just like we prepared some data in actions before suite we can clean up the test run in actions after the tests are finished. The Spring bean syntax here is not significantly different to those in before suite section:

```
<bean class="com.consol.citrus.container.SequenceAfterSuite">
      <property name="actions">
```

```
        <list>
            <!-- list of actions after suite -->
        </list>
    </property>
  </bean>
```

```
<bean class="com.consol.citrus.container.SequenceAfterSuite">
      <property name="actions">
          <list>
              <bean id="initDatabase" class="com.consol.citrus.actions.ExecuteSQLAction">
                  <property name="dataSource" ref="testDataSource"/>
                  <property name="statements">
                      <list>
                          <value>DELETE FROM TABLE PERSON</value>
                      </list>
                  </property>
              </bean>
          </list>
      </property>
  </bean>
```

We just remove all data from the database so we do not influence follow-up tests. Quite simple isn't it!?

22.3. Before test

Before each test is executed it also might sound reasonable to purge all JMS queues for instance. In case a previous test fails some messages might be left in the JMS queues. Also the database might be in dirty state. The follow-up test then will be confronted with these invalid messages and data. Purging all JMS destinations before a test is therefore a good idea. Just like we prepared some data in actions before suite we can clean up the data before a test starts to execute.

```
<bean class="com.consol.citrus.container.SequenceBeforeTest">
      <property name="actions">
          <list>
              <!-- list of actions before test -->
          </list>
      </property>
  </bean>
```

```
<bean class="com.consol.citrus.container.SequenceBeforeTest">
      <property name="actions">
          <list>
              <ref bean="purgeJmsQueues"/>
          </list>
      </property>
  </bean>
```

Chapter 23. 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.xsd
      http://www.citrusframework.org/schema/testcase
      http://www.citrusframework.org/schema/testcase/citrus-testcase.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 24. Tracing incoming/outgoing messages

As we deal with message based interfaces Citrus will send and receive a lot of messages during a test run. Now we may want to see these messages in chronological order as they were processed by Citrus. We can enable message tracing in Citrus in order to save messages to the file system for further investigations.

Citrus offers an easy way to debug all received messages to the file system. You need to enable some specific loggers and interceptors in the Citrus configuration (citrus-context.xml).

```
<bean class="com.consol.citrus.report.MessageTracingTestListener"/>
```

Just add this bean to the Spring configuration and Citrus will listen for sent and received messages for saving those to the file system. You will find files like these in the default test-output folder after the test run:

For example:

```
logs/trace/messages/MyTest.msgs

logs/trace/messages/FooTest.msgs

logs/trace/messages/SomeTest.msgs
```

Each Citrus test writes a *.msgs* file containing all messages that went over the wire during the test. By default the debug directory is set to `logs/trace/messages/` relative to the project test output directory. But you can set your own output directory in the configuration

```
<bean class="com.consol.citrus.report.MessageTracingTestListener">
  <property name="outputDirectory" value="file:/path/to/folder"/>
</bean>
```

Note

As the file names do not change with each test run message tracing files may be overwritten. So you eventually need to save the generated message debug files before running another group of test cases.

Lets see some sample output for a test case with message communication over SOAP Http:

```
Sending SOAP request:
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header>
<Operation xmlns="http://citrusframework.org/test">sayHello</Operation>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
<ns0:HelloRequest xmlns:ns0="http://www.consol.de/schemas/samples/sayHello.xsd">
    <ns0:MessageId>0857041782</ns0:MessageId>
    <ns0:CorrelationId>6915071793</ns0:CorrelationId>
    <ns0:User>Christoph</ns0:User>
    <ns0:Text>Hello WebServer</ns0:Text>
</ns0:HelloRequest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

=======================================================================

Received SOAP response:
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Header/>
```

```
<SOAP-ENV:Body>
<ns0:HelloResponse xmlns:ns0="http://www.consol.de/schemas/samples/sayHello.xsd">
    <ns0:MessageId>0857041782</ns0:MessageId>
    <ns0:CorrelationId>6915071793</ns0:CorrelationId>
    <ns0:User>WebServer</ns0:User>
    <ns0:Text>Hello Christoph</ns0:Text>
</ns0:HelloResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For this message tracing to work we need to add logging listeners to our sender and receiver components accordingly.

```
<citrus-ws:message-sender id="webServiceRequestSender"
    request-url="http://localhost:8071"
    reply-handler="webServiceReplyHandler"
    message-factory="messageFactory"
    interceptors="clientInterceptors"/>

    <util:list id="clientInterceptors">
        <bean class="com.consol.citrus.ws.interceptor.LoggingClientInterceptor"/>
    </util:list>
```

Important

Be aware of adding the Spring *util* XML namespace to the citrus-context.xml when using the *util:list* construct.

Chapter 25. 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.

25.1. 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:

```
CITRUS TEST RESULTS

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

  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
```

25.2. 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.

25.3. HTML reports

Citrus creates HTML reports after each test run. The report provides detailed information on the test run with a summary of all test results. You can find the report after a test run in the `target/test-output/citrus-reports` directory.

The report consists of two parts. The test summary on top shows the total number executed tests. The main part lists all test cases with detailed information. With this report you immediately identify all tests that were failing. Each test case is marked in color according to its result outcome.

The failed tests give detailed error information with error messages and Java StackTrace information. In addition to that the report tries to find the test action inside the XML test part that failed in execution. With the failing code snippet you can see where the test stopped.

Note

JavaScript should be active in your web browser. This is to enable the detailed information which comes to you in form of tooltips like test author or description. If you want to access the tooltips JavaScript should be enabled in your browser.

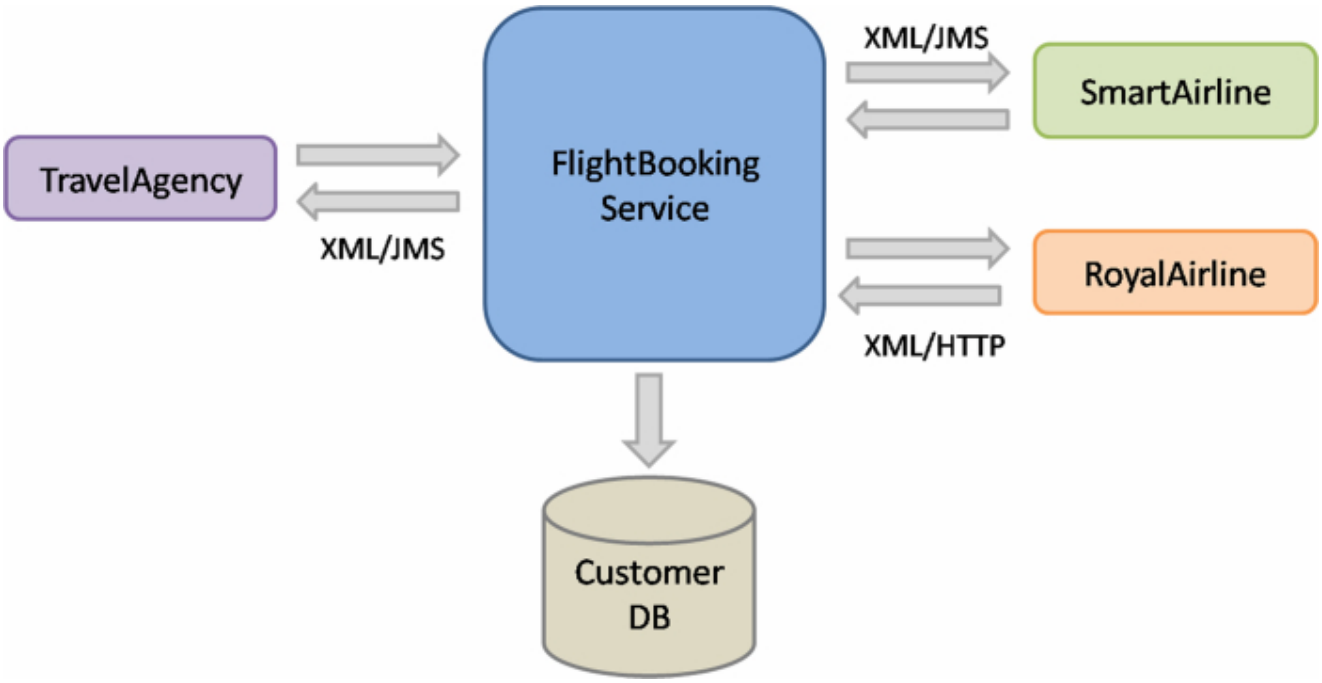
Appendix A. Citrus Samples

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.

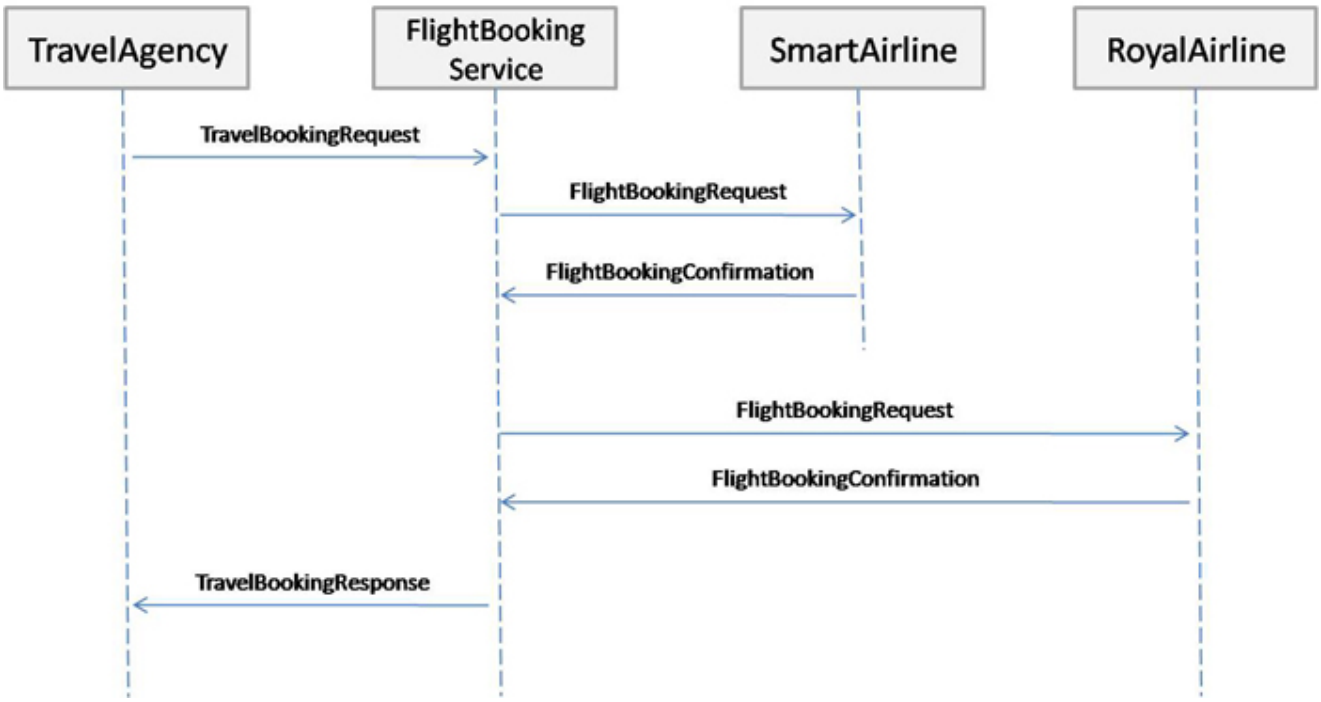
The airlines will confirm or deny the flight bookings. The FlightBookingService application consolidates all incoming flight confirmations and combines them to a complete travel confirmation or denial that is sent back to the travel agency. Next picture tries to put the architecture into graphics:



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 separate 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 Apache ActiveMQ 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 requests over Http and sends them over JMS to a message receiver as we already 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.adapter.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 synchronous 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.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 proceeds 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.