Citrus Framework - Reference Documentation

Version 1.0

Table of Contents

Preface

Integration testing can be very hard, especially when there are no sufficient tools that support you. Regarding Java unit testing very good tools and APIs exists (JUnit, TestNG, EasyMock, and many more). The testing tools support you and try to ease up your life as a tester who is in charge of testing a software in various ways. When speaking of integration testing in a service-oriented architecture (SOA) things are getting evil. Besides unit testing efforts especially those Enterprise Application Integration (EAI) projects need to be tested regarding their integration into the existing application environment on the customer side. Without sufficient tool support the integration testing is exhausting and sometimes barely possible.

Testing can then be a hard task, because the tester may be forced to simulate several external interface applications. The simulation of interface partners cause severe problems regarding maintainability and compability. It is not unusual, that a tester has to simulate several applications manually during a test, because of lacking tool support. This manual test approach is very time consuming and error prone. Every time a tester has to manually validate the success of at test case it is difficult to repeat those tests every time a change was made in the source code of the application. Regression testing and continuous testing is very important but not possible the time testers are forced to manually test the applications. Fully automated integration testing of whole use cases in an end-to-end point of view is very difficult using this manual testing approach.

The Citrus test framework aims to fill in this gap. The tool helps you to perform complex integration testing in a message based EAI environment. Citrus is designed to completely test whole use case scenarios in an automated way, where interface partners are simulated in their behaviour and message contents are automatically validated, so the correct processing is always ensured.

This document provides a reference guide to all features of the Citrus test framework and it gives a detailed picture of how to set up an automated integration testing environment. Since this document is considered to be under construction, please do not hesitate to give any comments or requests to us using our user or support mailing lists.

Chapter 1. Introduction

Enterprise Application Integration (EAI) projects usually communicate with different interface partners using a loosely coupled message-based communication. The success of a use case is defined by the cooperation of several applications exchanging information with each other. Therefore it is essential to test this cooperation/communication between the applications in several integration tests.

In software devlopment integration testing aims to test the interaction of software components that depend on each other. In EAI projects these components are loosely coupled sofware systems communicating with each other over various transports exchanging messages and information. The interaction of these components is essential to successfully process a use case scenario.

How can somebody test use case scenarios that include several interface partners interacting with each other? How can somebody ensure that the software component he/she is in charge of is working correctly with the other software systems. How can somebody run positive and negative integration test cases in an automated reproducable way? Citrus tries to answer these questions by offering fully automated integration testing always looking at the whole picture of several systems interacting with each other.

1.1. Overview

Citrus aims to strongly support you in simulating your interface partners. You can easily subscribe and publish to a wide range of protocols like HTTP, JMS, TCP/IP and SOAP in order to test your message based applications. The framework will send/receive messages in order to create a communication as it would be in real production environment. In each step the framework is able to validate the message contents by comparing them to expected contents. In addition to that the framework offers a wide range of functionalities to take control of the process flow during a test (e.g. iterations, system availability checks, database validation, parallelism, delaying, error simulation, and many more).

The basic goal is to test a whole use case scenario including all interface partners. With easy definition of test cases via XML the framework will simulate the message flow step by step and ensure the correct end-to-end processing.

1.2. Usage scenarios

When should someone use the Citrus test framework? If you are in charge of an EAI application and need to test its integration into the existing software landscape. In case your project interacts with a lot of other software systems and you need to simulate those in order to test your end-to-end processing. In case you need to increase the stability of an existing EAI projects dealing with change requests, bugfixes or regression testing. Or if you simply need to increase your test coverage, in order to feel comfortable regarding the next software release.

Chapter 2. Setup

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

After downloading the latest release there are several ways to use Citrus in your project. The next sections will describe to you the Citrus project setup possibilities. Choose on of them and include Citrus into your project.

In case you use Maven in your project you may choose the Maven way described in Section 2.4, “Using Maven”. Others may feel right downloading and unzipping the Citrus archive release manually.

2.1. Download

You can download the latest Citrus release from our homepage 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 one would choose the with-dependencies archive in order to use Citrus, because this archive includes the Citrus binaries plus all needed depedency libraries. In case you want to get in touch with developing Citrus you should go with the project archive. This archive will give you the complete Citrus project sources. Using Maven you will easily be able to build Citrus on your own and extend Citrus with your own code.

The src archive is helpful during extension of the framework as it provides the Java sources for you. You may then be able to debug into Citrus classes and have a look at the sources during development of your own features.

2.2. Prerequisits

You need following software 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 manage your Citrus project and create and execute test cases. Use the Java IDE you like best (e.g. Eclipse or IntelliJ IDEA).

- Maven 2.0.9 or higher (optional)

Maven will ease up your life when it comes to managing project dependencies and project builds. Maven can support you while using Citrus. In case you have a Maven built project it is very easy to include Citrus tests into your existing build lifecycle.

2.3. Installation

Once you have downloaded the latest Citrus archive you need to unzip the archive into a suitable location on your local storage.

You are now able to run the built-in testing samples, which also should give you an idea how to use Citrus with ANT.

2.4. Using Maven

Citrus uses Maven as build tool. Therefore it is also very easy for you to include Citrus into your Maven project.

In case you already use Maven build tool in your project it is most suitable for you to include Citrus as a test-scoped dependency into your Maven project

- Add Citrus as test-scoped project dependency to your projects POM (pom.xml)

```
<dependency>
  <groupId>com.consol.citrus</groupId>
  <artifactId>citrus-core</artifactId>
  <version>0.9</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>0.9</version>
  <configuration>
    <author>Mickey Mouse</author>
    <targetPackage>com.consol.citrus</targetPackage>
  </configuration>
</plugin>
```

- Create tests using the citrus-maven-plugin (goal: citrus:create-new-test)

- Tests will be included into your projects integration-test phase using the surefire plugin

```
<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>
```

- Citrus source directories have to be defined as test sources

```
<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>
```

- Run Maven integration-test phase to execute all tests (goal: integration-test)

Chapter 3. Test case

Now let us start writing test cases! The upcoming guide will introduce all possible actions inside a test case and explain how to connect to other systems by sending and receiving messages over various transports.

Citrus specifies test cases through simple XML files. The whole test case description will take place in one single XML file. This chapter will introduce the custom XML schema language that defines a test cases.

3.1. Defining a test case

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

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

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

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

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

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

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

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

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

3.1.1. Description

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

3.1.2. Variables

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

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

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

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

3.1.3. Actions

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

```
<actions>
```

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

All actions have individual names and properties that define the action behaviour. 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 5, *Test actions* gives you a brief description of all available actions that can be part of a test case execution.

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

3.1.4. Cleanup

The finally element also contains a list of test actions. These actions will be executed at the very end of the test case even if errors did occur during the execution before. This is the right place to tidy up things that were previously created by the test like cleaning up the database for instance. The finally section is described in more detail in Chapter 8, *Finally section*

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

3.2. Meta information

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

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

The status allows following values: DRAFT, READY_FOR_REVIEW, DISABLED, FINAL. The meta-data information to a test is quite important to give the reader a first information about the test.

It is also possible to generate test documentation using this meta-data information. The built-in Citrus documentation generates HTML or Excel documents that list all tests with their metadata information and description.

Note

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

Chapter 4. Send and receive messages

In the previous chapter the basic test case structure was introduced with its <variables> and <actions>. The <actions> element contains all actions that will be executed during the test case in sequential order. The framework offers several built-in actions that the user can choose from. These actions will take our interest in the next sections.

As sending and receiving messages is an essential part in EAI projects we will handle these actions first.

4.1. Sending messages

The <send> action publishes messages to a destination. The transport to be used does - for now - not matter to the test case. The test case simply defines the sending action and uses a predefined message sender to actually publish the defined message. There are several message sender implementations in the framework available representing several ways to publish a message. This means that the transport protocol to be used (JMS, SOAP, HTTP, TCP/IP etc.) is not specified in the test case directly, but in the message sender definitions. We will see later in this document how message senders for various transports are configured in Citrus.

The advantage of this seperation of configuration is definitely the flexibility of test cases. The test case does not know anything about JMS ConnectionFactories, queue names or Http URLs. The transport underneath a sending action can change easily without affecting the test case definition.

Usually the message payload will be plain XML format. The framework is not limited to the XML message format, but to be honest XML is the default message format that is supported out of the box. Let us have a look at a first example how a sending action is defined in the test.

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

The test uses two variable definitions (requestTag and correlationId), so first of all let us refresh in mind what variables do. Variables defined at the very beginning of the test case are valid throughout all actions that take place in the test. This means that actions can simply reference a variable by the variable expression (e.g. ${correlationId}).

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 reaches better maintainability and flexibility using variables.

Now lets have a closer look at the sending action. The 'with' attribute might catch someones attention at first. So what does the 'with' attribute do? This attribute references a message sender definition by name. As previously mentioned the message sender definition lies in a seperate configuration file and contains the actual message transport configurations. In this example the *"getCustomerRequestMessageSender"* is used to send the message over JMS to a destination queue. The test case is not aware of these details, because it does not have to. The advantage is obvious: Many test cases may use the same message sender in order to send messages of type 'getCustomerRequest'.

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

Tip

It is good practice to follow a naming convention when choosing names for message senders and receivers. The intended purpose of the message sender as well as the sending/receiving actor should be clear when choosing the name. For instance messageSender1, messageSender2 will not give any hint to the reader what is actually sent and to which destination.

Now that the actual message sender is clear, the test must specify the message content itself. This is done with these elements:

- *message*: This element defines the message to be sent. There are several child elements available:

    - *data*: Inline definition of the XML message (instead of <resource> element)

    - *resource*: External file holding the XML message to be sent (instead of <data> element)

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

      The location of the file can be declared as file system resource (file:) or as classpath resource (classpath:).

    - *element*: Explicitly overwrite values in the XML message using XPath. XML elements can be replaced explicitly by previously defined test variable values. Each <element> entry provides a "path" and "value" attribute. The "path" should be a valid XPath expression evaluating to a node element or attribute in the message to be sent. The "value" can be a variable or any other static value. The framework will replace the value before sending the message.

- *header*: Defines a header for the message (e.g. JMS header information or SOAP header). This <header> element has child elements to define the header values:

    - *element*: Similar to the <element> tag in the message definition. Here the attributes "name" and "value" define the header entry. The "name" will be the name of the header entry and "value" its respective value. Again the usage of variables is very useful here, too.

The most important thing when dealing with sending actions is to prepare the XML message to be sent. The message content itself is specified by the <data> or the <resource> property. These elements hold the message payload as internal CDATA definition or as external file resource. Usually there are some message values that have to be dynamically set before sending the message (especially when using test variables). This is why you can overwrite specific message elements. The example above uses the variable ${correlationId} directly in the XML payload definition. The framework will replace this variable with the respective value before sending the message. There is also a second approach to overwrite message elements using XPath. The "/MessageHeader/RequestTag" XPath expression for instance overwrites the respective request tag value in the message. The two approaches of setting message elements can coexist simultaneously; however the inline XML content replacement seems less complex than XPath, although XPath may be the right way for powerful replacements inside the message payload.

The <header> property is used to set the values like "Operation" and "RequestTag". The example shows that the use of variables is supported here using the common variables syntax.

4.2. 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 s simple example showing how it works.

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

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

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

Once the message has arrived, the content can be validated in various ways. On the one hand you can specify a whole XML message template that you expect. In this case the received XML structure will be compared to the expected XML message template element by element. On the other hand you can specify some elements that are of interest for the test. These elements will then be explicitly validated. The framework will search for the right element in the received XML structure and compare the value with the expected one.

Besides this message payload validation the framework can also validate the message header values. Simply specify the header name and the expected value inside the receive action. Variable usage is supported as usual. But let us have a closer look at the validation options and the features regarding message receiving step by step.

4.2.1. Messages selectors

The *<selector>* element inside the receiving action defines key-value paris in order to filter the messages beeing received. The key value paris 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 occured during previous test cases. As a result the EAI application keeps sending messages that are not valid any more for the currently running test case. The test case might fail because the received message does not apply to the currently tested use case. The messages received are simply failing because the message content does not fit the expected one (e.g. correlation-ids, customer informations etc.).

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

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

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

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

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

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

Important

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

4.2.2. Expected message templates

Once a message is received a tester may always want to validate the message content. The tester has a expected message in mind that is compared to the actual message that arrived.

A tester can sepcify expected message templates using the following elements inside a receiving action:

- *<data>*: Defines an inline XML message template that is expected
- *<resource>*: Defines an expected XML message via external file resources

Both ways inline CDATA XML or external file resource do specify a expected message template. The framework will use this template to compare the received message to it. Now the message template is very static. Dynamic message contents may cause the validation to fail. Therefore a tester can enrich the expected message template using test variables before the validation or a tester can ignore some elements in validation.

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

The previous program linsting shows the two ways of setting variable values inside a message template. First of all you can simply place variable expressions inside the message data (see how ${correlationId} is used). As a second possibility you can 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.

However despite of adding dynamic variable values to the expected message template some elements inside a message will not be dedicated for validation at all. Communication timestamps inside a message for example as shown here:

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

This timestamp value will dynamically change from test run to test run, so lets ignore it in validation

```
<ignore path="//ResponseMessage/MessageHeader/Timestamp"/>
```

This is how a message template is manipulated before validation. The framework applies a detailed XML tree validation to ensure that the messages (expected and received) are matching. XML attributes, element names, namespaces as well as the number of child elements are validated. In case one single element is not equal to the expected one the validation will raise errors and the test will fail.

4.2.3. Validating message elements

In the previous chapter we learned how to validate the whole XML tree against a expected message template. In some cases this approach might be overkill. In case you as a tester need to validate only a small subset of message elements you might want to use the element validation.

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

Instead of comparing the whole message only the specified message elements are validated. The framework tries to find the elements in the received message via XPath and compares its value to the expected one. Nothing else is validated here.

Note

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

Tip

The test framework 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.

4.2.4. Validate the message header

After we have validated the message payload in the previous chapters we might be interested in validating the message header too.

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

The given values have to be present in the received message header and their value has to fit the expected value otherwise the test case will fail.

4.2.5. Saving dynamic message content

Imagine you receive a message in your test that contains a generated message id. You have no chance to predict the id because it was generated dynamically. But unfortunately you need to return this id in the respective response message to meet the requirements. We need to save the dynamic value into a new test variable.

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

As you can see the framework is able to store content into variables. The content may come from message payload or message header values. You can store the information to new test variables or existing ones. You do not need to create the variables before the extracting will automatically create a new variable in case it does not exist. The upcoming test actions can access these variables as usual.

At this point you know the two most important test actions in the test framework. Sending and receiving actions will become the main components of your integration tests. In most cases a test will create a message flow, which means a sequence of sending and receiving messages in order to replicate a use case.

Chapter 5. 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.

5.1. 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 descirbed in the following sections.

5.1.1. Updating the database

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

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

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

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

The first action uses inline SQL statements defined directly inside the test case. The next action uses an external SQL resource file instead. The file resource can hold several SQL statements seperated 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.

5.1.2. Verifying data from the database

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

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

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

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

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

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

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

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

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

Note

When validating database query result sets Citrus can only handle single

5.2. Sleep

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

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

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

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

5.3. 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.

5.4. 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 behaviour. And in contrast to the normal behaviour when a message is received in the time period the test will fail with error.

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

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

The action offers two attributes:

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

TODO: describe receive selected!

5.5. Echo

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

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

Result on the console:

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

5.6. Time measurement

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

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

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

        <trace-time id="time_line_id"/>

        <sleep time="3.5"/>

        <trace-time id=" time_line_id "/>

        <sleep time="5.0"/>

        <trace-time/>

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

The test output looks like follows:

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

Note

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

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

5.7. Create variables

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

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

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

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

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

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

5.8. Trace variables

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

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

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

Simply add the <trace-variables> action to your action chain and all variables will be printed out to the console. You are able to define a special set of variables by using the <variable> child elements. See the output that was generated by the test example above:

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

5.9. Failing the test

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

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

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

Test results:

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

[...]

TEST RESULTS coreTestSuite

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

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

5.10. Input

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

The user input is stored to the respective variable value.

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

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

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

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

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

The input action has following attributes:

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

Note

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

Note

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

5.11. Load

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

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

Content of load.properties:

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

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

Output:

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

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

Important

Existing variables are overwritten!

5.12. Purging JMS destinations

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

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

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

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

Purging the JMS queues in every test case is quite exhausting because every test case needs to define a purging action at the very beginning of the test. Fortunately the test suite definition offers tasks to run before, between and after the test cases which should ease up this tasks a lot. The test suite offers a very simple way to purge the destinations between the tests. See Section 14.1, “Tasks before, between and after the test run” for more information about this.

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

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

5.13. Including own actions

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

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

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

Chapter 6. Templates

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

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

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

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

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

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

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

Note

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

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

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

There is an open question when dealing with templates that are defined somewhere else outside the test case. How to handle variables? A templates may use different variable names then the test and vice versa. But all variables must be present with respective values as soon as the template gets called. Otherwise the variables are unknown to the template and the test will raise errors.

So a first approach would be to harmonize variable usage accross 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. On the other hand a template can handle parameters to solve this problem. When a template is called the calling actor has to set some parameters. Lets discuss an example for this issue.

The "doTraceVariables" template uses the variable ${time}. The test case must therefore have a variable called time too or declare the variable ${time} as a parameter when calling the template:

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

The variable *myTime* is translated - if you would call it like this - into the desired *time* parameter. So the template works fine referencing the time variable. This means that you always have to check the used variables 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.

Chapter 7. 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.

7.1. 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 very unspectacular - one might say boring - because it simply groups a pair of test actions to sequential execution.

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

7.2. Parallel

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

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

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

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

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

            <iterate condition="i lt= 5"
                        index="i">
                <echo>
                    <message>10</message>
                </echo>
            </iterate>
        </parallel>
    </actions>
</testcase>
```

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

Note

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

7.3. Iterate

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

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

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

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

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

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

- gt (greater than)
- gt= (greater than equals)
- = (equals)
- and (logical combining of two Boolean values)
- or (logical combining of two Boolean values)
- () (brackets)

Important

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

7.4. Repeat until true

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

Important

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

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

7.5. Repeat on error until true

The next looping container is called repeat-on-error-until-true. This container simply repeats group of actions only in case one embedded action has failed with error. In case of such an error inside the container the loop will try again to execute `all` embedded actions. The on-error execution takes place until one of the following conditions evaluates to true:

1. All embedded actions were processed successfully

2. The ending condition evaluates to true

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

In the code example the error loop continues four times as the <fail> action definitely fails the test. During the fifth iteration The condition "i=5" evaluates to true and the loop breaks its processing.

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 only in case all embedded actions were not raising any errors before in one iteration.

Chapter 8. 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 upcoming test cases. To avoid this problems you should use the finally block of the test case. The <finally> section contains actions that are executed even in case the test fails. Using this strategy the database cleaning tasks mentioned before will find execution in every case (success or failure).

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

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

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

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

Chapter 9. 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="//element/elementA"
                 value="text"></validate>
    <validate path="//element/elementA[@attr='A']"
                 value="text"></validate>
    <validate path ="//element/elementB"
                 value="%elementValue"></validate>
    <validate path ="//element/elementB/@attribute"
                 value="%attributeValue"></validate>
    <validate path ="/root/element"
                 value="namespace"></validate>
    <validate path ="/root/element"
                 value="namespace"></validate>
    <validate path ="//*[.='search-for']"
                 value="search-for"></validate>
</message>
```

9.1. Handling XML namespaces

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

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

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

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

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

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

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

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

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

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

9.2. Handling default namespaces

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

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

```
            <Id>1</Id>
        </Customer>
    </MessageBody>
</RequestMessage>
```

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

```

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

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

```

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

Tip

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

Chapter 10. Connecting with JMS

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

Note

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

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

    [...]

</beans>
```

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

10.1. 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 tranport details like queue names or connection credentials. This information is stored in the basic Spring configuration. So let us have a look at a simple JMS message sender configuration in Citrus.

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

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

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

Tip

Spring makes it very easy to connect to other JMS broker implementations too (e.g.

TIBCO Enterprise Messaging Service, IBM Websphere MQ). Just substitute the implementing class in the connectionFactory bean.

Note

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

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

Alternatively you may want to directly specify a Spring jmsTemplate.

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

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

10.2. 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.

10.3. 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.

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

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

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

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.

10.4. JMS synchronous message receiver

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

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

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

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

Chapter 11. Http Support

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

Note

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

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

    [...]

</beans>
```

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

11.1. Http message sender

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

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

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

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

11.2. Http server

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

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

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

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

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

11.2.1. 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 reponse code 202. In the introducing example this message handler was used to provide reponse messages to the calling client. The handler does not need any configurations or properties as it simply responds with an empty Http response.

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

11.2.2. 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/sample.xsd}ns0:Operation"
                      value="sayHelloResponse"/>
            <entry key="{http://www.consol.de/schemas/samples/sample.xsd}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.

11.2.3. Xpath dispatching message handler

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

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

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

11.2.4. JMS connecting message handler

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

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

Tip

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

Chapter 12. SOAP WebServices

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

Note

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

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

    [...]

</beans>
```

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

12.1. SOAP message sender

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

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

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

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

12.2. Receiving SOAP messages

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/it/resources"/>
```

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

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

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

    <bean id="helloServicePayloadMapping"
          class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
        <property name="mappings">
            <props>
                <prop key="{http://www.consol.de/schemas/samples/sayHello.xsd}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/samples/sayHello.xsd">
                          <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/samples/sayHello.xsd}ns0:Operation"
                                value="sayHelloResponse"/>
                      <entry key="{http://www.consol.de/schemas/samples/sayHello.xsd}ns0:Request"
                                value="HelloRequest"/>
                      <entry key="{http://www.consol.de/schemas/samples/sayHello.xsd}ns0:SOAPAction"
                                value="sayHello"/>
                  </map>
              </property>
          </bean>
        </property>
    </bean>
</beans>
```

The programlisting above describes a normal request mapping. The mapping is responsible to forward incoming requests to an endpoint which will handle the request and provide a response. First of all Spring's logging interceptor is added to the context. Then we use a payload mapping (PayloadRootQNameEndpointMapping) in order to map all incoming `'HelloStandaloneRequest'` SOAP messages to the `'helloServiceEndpoint'`. The enpoint definition follows at the very end of the programlisting. Inside the endpoint configuration we can see the well known message handler that is responsible for providing a proper response message for the client. The various message handlers in Citrus were already described in Chapter 11, *Http Support*.

In this example the `'helloServiceEndpoint'` uses the `'StaticResponseProducingMessageHandler'` which is always returning a static response message. The endpoint transforms the static response into a proper SOAP message that is sent back to the calling client as SOAP response. 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 11, *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.

Chapter 13. 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.

13.1. 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="Hallo 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>
```

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

Function output:

Test Framework

Hallo

13.3. citrus:stringLength()

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

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

Function output:

20

13.4. 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:

Hallo Test Framework

13.5. 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

13.6. 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>
```

Analogue 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

13.7. citrus:round()

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

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

Function output:

3

13.8. citrus:floor()

This function will round down decimal number values.

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

Function output:

3.0

13.9. 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

13.10. citrus:randomNumber()

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

Next example will show the function usage:

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

Function output:

8954638765

0006387650

45638765

13.11. citrus:randomString()

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

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

Function output:

Hr546dfA65

Ag3876det5

4SDF87TR

4tkhi7uz

Vt567JkA32

13.12. 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

13.13. citrus:upperCase()

This function converts any string to upper case letters.

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

Function output:

HALLO TEST FRAMEWORK

13.14. citrus:lowerCase()

This function converts any string to lower case letters.

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

Function output:

hallo test framework

13.15. 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*

13.16. 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*

13.17. 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*

13.18. 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*

13.19. citrus:absolute()

The function will return the absolute number value.

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

abs = *3.0*

Chapter 14. Testsuite

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

14.1. Tasks before, between and after the test run

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

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

        </list>
    </property>

    <property name="tasksBetween">
        <list>

        </list>
    </property>

    <property name="tasksAfter">
        <list>

        </list>
    </property>

</bean>
```

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

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

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

Tip

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

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

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

14.2. Include and exclude tests

While using the test framework you will find out that the duration of tests differ according to the purpose of the test. Especially timeout tests take a long time. In some cases the tester might not want to execute those tests, because it would take too long. Another situation would be that the tester intends to test a certain group of tests explicitly and all other tests should not be executed.

In these situations the test suite is very helpful because here you can easily declare include or exclude test cases from a test run.

The following example tries to explain this functionality in detail:

```
<bean name="testSuite"
         class="com.consol.citrus.TestSuite">
    <property name="excludeTests">
        <list>
            <value>*timeout*</value>
            <value>someTestName</value>
            <value>startsWith*</value>
            <value>*endsWith</value>
        </list>
    </property>

    <property name="includeTests">
        <list>
            <value>*timeout*</value>
            <value>someTestName</value>
            <value>startsWith*</value>
            <value>*endsWith</value>
        </list>
    </property>
</bean>
```

Both properties support name patterns. This means that you can use the wildcard "*" to define name patterns as shown in the examples. Be careful to use both including and excluding patterns at the same time, because they might overrule each other.

14.3. Multiple testsuites

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

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

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

Chapter 15. TIBCO support

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

15.1. Connecting with TIBCO Hawk

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

Tip

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

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

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

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

In general the HawkAgent has the following properties:

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

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

15.2. Connecting with TIBCO Collaborator Workflow Server

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

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

The bean looks like follows:

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

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

Chapter 16. 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.

16.1. XSD validation

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

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

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

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

Important

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

```
<receive with="getCustomerRequestReceiver">
    <message schema-validation="false">
        <validate path="//ns2:RequestMessage/ns2:MessageHeader/ns2:RequestTag"
```

```
                     value="${requestTag}"/>
        <validate path="//ns2:RequestMessage/ns2:MessageHeader/ns2:CorrelationId"
                     value="${correlationId}"/>
        <namespace prefix="ns2" value="http://testsuite/default"/>
    </message>
    <header>
        <element name="Operation" value="GetCustomer"/>
        <element name="RequestTag" value="${requestTag}"/>
    </header>
</receive>
```

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

16.2. DTD validation

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

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

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

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

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

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

```
                <message>
                    <text>Hello TestFramework!</text>
                </message>
            </root>
        ]>
        <data/>
    </message>
</receive>
```

Chapter 17. Debugging received messages

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

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

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

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

For example:

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

logs/debug/messages/message1.body

logs/debug/messages/message2.header

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

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

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

Note

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

Chapter 18. 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.

18.1. Console logging

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

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

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

```
TEST RESULTS citrus-default-testsuite

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

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

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

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

18.2. JUnit reports

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

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

Appendix A. Citrus samples

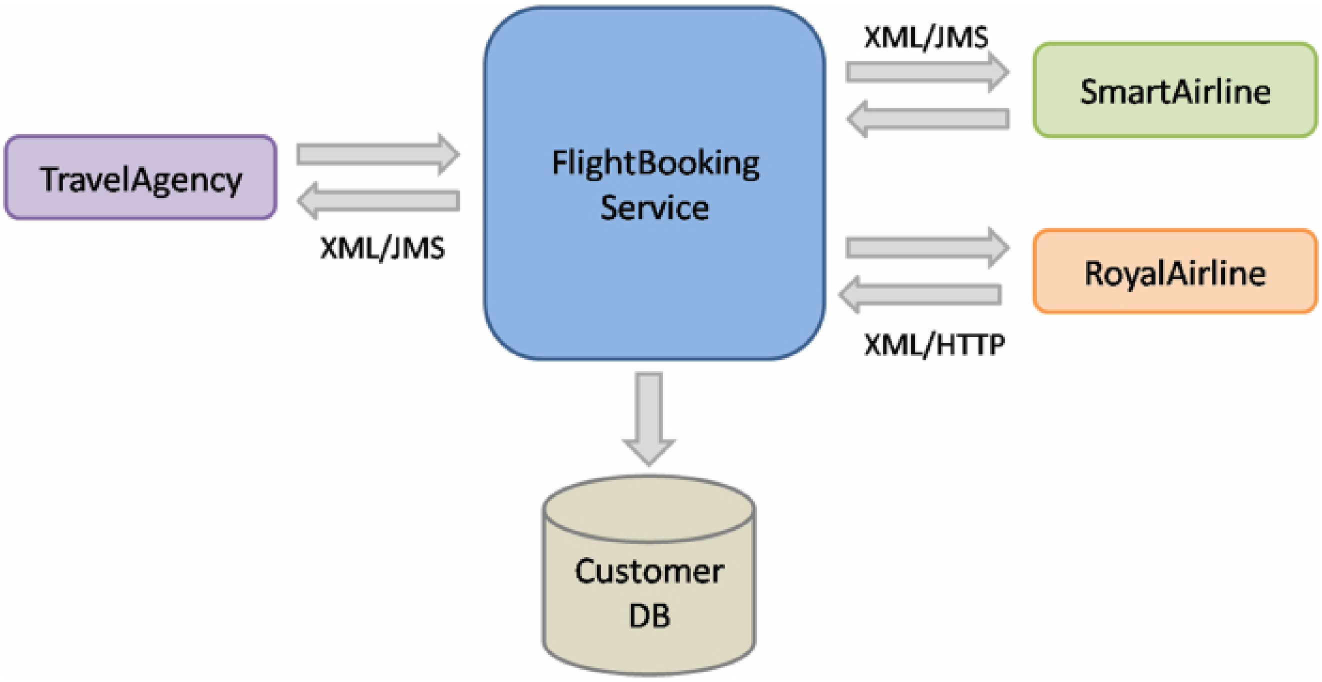
This part will show you some sample applications that are tested using Citrus integration tests. See below a list of all samples.

- Section A.1, “The FlightBooking sample”

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 consits of a complete travel route including several flights. The FlightBookingService application will split the complete travel booking into seperate flight bookings that are sent to the respective airlines in charge. The booking and customer data is persisted in a database.

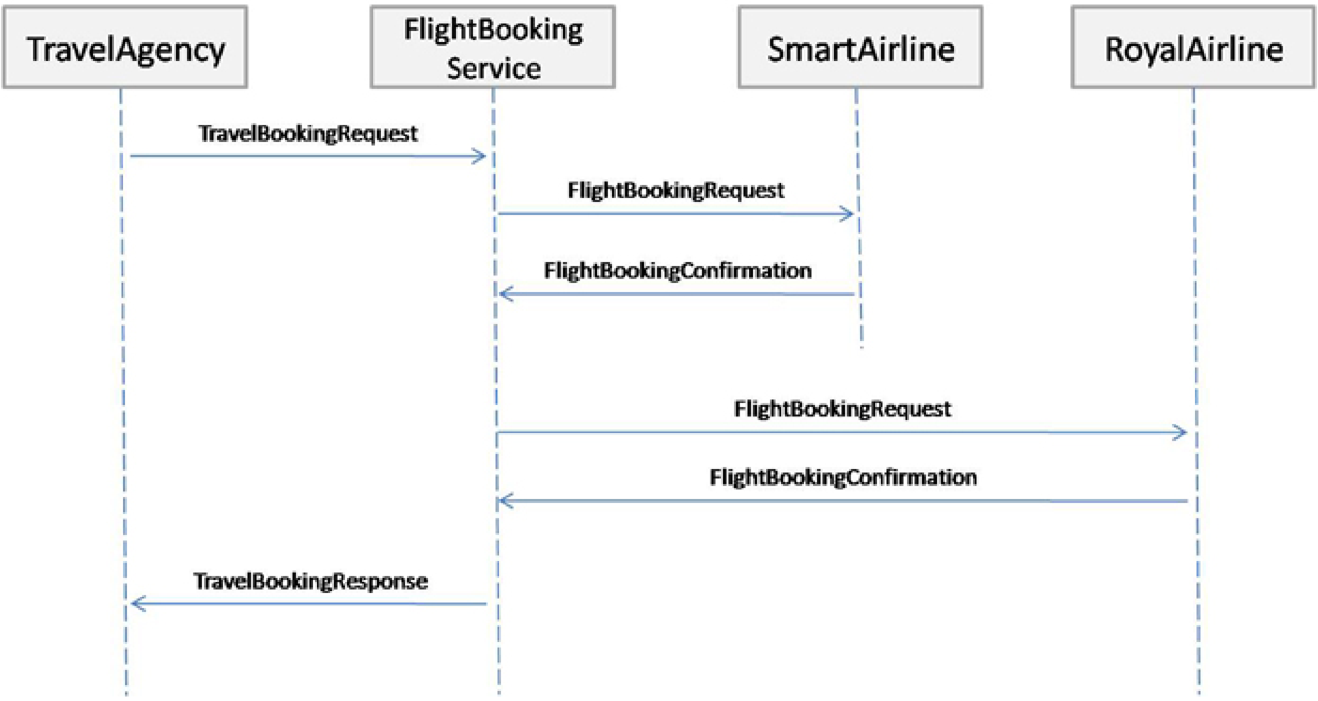
The arilines will confirm or deny the flight bookings. The FlightBookingService application consolidates all incoming flight confirmations and combines them to a complete travel confirmation or denial that is sent back to the travel agency. Next picture tries to put the architecture into graphics:



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

A.1.1. The use case

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



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

A.1.2. Configure the simulated systems

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

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

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

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

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

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

The identifyers of the message senders and receivers are very important. We should think of suitable ids that give the reader a first hint what the sender/receiver is used for. As we want to simulate the

TravelAgency in combination with sending booking requests our id is "travelAgencyBookingRequestSender" for example.

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

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

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

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

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

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

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

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

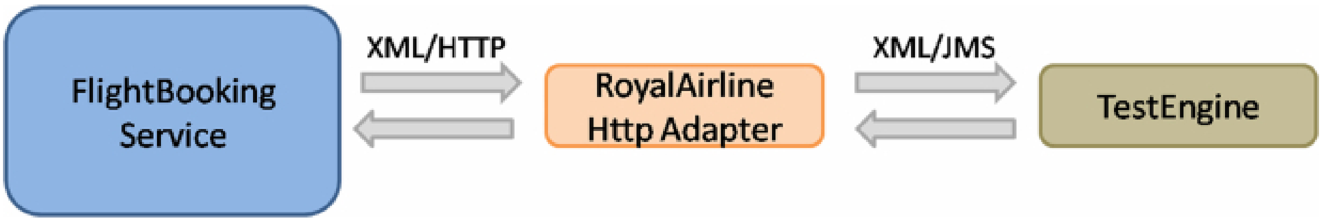
We do not define a new JMS connection factory because TravelAgency and SmartAirline are using the same message broker instance. In case you need to handle multiple connection factories simply define the connection factory with the attribute "connection-factory".

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

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

A.1.3. Configure the Http adapter

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



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

Let us define the Http adapter in the Spring configuration:

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

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

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

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

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

A.1.4. The test case

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

```
<?xml version="1.0" encoding="UTF-8"?>
<spring:beans xmlns="http://www.citrusframework.org/schema/testcase"
                 xmlns:spring="http://www.springframework.org/schema/beans"
                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                 xsi:schemaLocation="http://www.springframework.org/schema/beans
                 http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                 http://www.citrusframework.org/schema/testcase
                 http://www.citrusframework.org/schema/testcase/citrus-testcase.xsd">
    <testcase name="FlightBookingTest">
        <meta-info>
            <author>Christoph Deppisch</author>
            <creationdate>2009-04-15</creationdate>
            <status>FINAL</status>
            <last-updated-by>Christoph Deppisch</last-updated-by>
            <last-updated-on>2009-04-15T00:00:00</last-updated-on>
        </meta-info>
        <description>
            Test flight booking service.
        </description>
        <variables>
            <variable name="correlationId" value="citrus:concat('Lx1x', 'citrus:randomNumber(10)')"/>
            <variable name="customerId" value="citrus:concat('Mx1x', citrus:randomNumber(10))"/>
        </variables>
        <actions>
            <send with="travelAgencyBookingRequestSender">
                <message>
                    <data>
                      <![CDATA[
                        <TravelBookingRequestMessage
                          xmlns="http://www.consol.com/schemas/FlightBooking/TravelAgency/TravelAgencySchema.x
                          <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/FlightBooking/AirlineSchema.xsd">
                          <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/FlightBooking/AirlineSchema.xsd">
                          <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.xsd">
                          <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/FlightBooking/AirlineSchema.xsd">
                          <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/FlightBooking/TravelAgency/TravelAgencySchema.x
                          <correlationId>${correlationId}</correlationId>
                          <success>true</success>
                          <flights>
                              <flight>
                                  <flightId>SM 1269</flightId>
                                  <airline>SmartAirline</airline>
                                  <fromAirport>MUC</fromAirport>
                                  <toAirport>FRA</toAirport>
                                  <date>2009-04-15</date>
                                  <scheduledDeparture>11:55:00</scheduledDeparture>
                                  <scheduledArrival>13:00:00</scheduledArrival>
                              </flight>
                              <flight>
                                  <flightId>RA 1780</flightId>
                                  <airline>RoyalAirline</airline>
                                  <fromAirport>FRA</fromAirport>
                                  <toAirport>HAM</toAirport>
                                  <date>2009-04-15</date>
                                  <scheduledDeparture>16:00:00</scheduledDeparture>
                                  <scheduledArrival>17:10:00</scheduledArrival>
                              </flight>
                          </flights>
                        </TravelBookingResponseMessage>
                      ]]>
                    </data>
                </message>
                <header>
                    <element name="correlationId"
                                value="${correlationId}"/>
```

```
                </header>
            </receive>

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

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

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

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

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

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

The sending action chooses a message sender to actually send the message using a message transport (JMS, Http, SOAP, etc.). After sending this first "TravelBookingRequestMessage" request the test case expects the first "FlightBookingRequestMessage" message on the SmartAirline JMS destination. In case this message is not arriving in time the test will fail with errors. In positive case our FlightBookingService works well and the message arrives in time. The received message is validated against a defined expected message template. Only in case all content validation steps are successful the test continues with the action chain. And so the test case proceeeds and works through the use case until every message is sent respectively received and validated. The use case is done automatically without human interaction. Citrus simulates all surrounding applications and provides detailed validation possibilities of messages.