# Manual RelayConnector

## Table of Contents



# 1 Introduction

The basic function of the RelayConnector is to relay method calls to one or more other methods, with some simple transformations.

The RelayConnector can be used for a variety of purposes:

- To make a method that calls a number of different methods (a composite service).

- To wrap an existing method with some simple transformations.
- To provide a set of methods in the same namespace, which have different underlying technical implementations, and are thus implemented on different application connectors (and thus with different namespaces).
- For all kind of simple testing and prototyping purposes, such as a HelloWorld or a Echo service.
- (Planned) To relay method calls to different Cordys organisations

In order to use the RelayConnector, one must define methods by adding a method to a method set, and entering a small XML script in the "implementation" section of this method. The RelayConnector is not meant to be a full scripting language. It is intended that a script should only be 10 or 20 lines long. In fact a macro facility has been removed from the RelayConnector, because this lead to too complex scripts. Instead it is possible to easily define new command using java, which of course is a full blown programming language.

# 2 Fundamental concepts

The fundamental flow of a RelayConnector method is very simple:

- an input message is received
- do some stuff in the scipt
- an output message is returned

The first and last bullet are handled automatically by the Connector. It is up to the implementation script of a method to do some interesting stuff with the input message, and prepare the correct output message.

## 2.1 XML Variables

In a script one can use and manipulate some simple XML variables. For each script two basic variables always exist:

- `/input` contains the input XML request message
- `/output` is used to compose the output XML response message

Additional variables will contain the result of method calls that the script has executed (see Chapter 4.5). Additionally it is possible to define new variables by hand (see the `xml` command in Chapter 4.9), but it is not sure if this feature is really needed.

Each variable has a name and the value of the entire XML or one of it's sub-elements can be referenced using a starting slash, and possibly additional element selectors. When assigning or appending to a XML variable the leading slash is not needed.

## 2.2 Expressions

The expression syntax is still very basic and the parser is very simplistic and will not handle all cases

equally well. It is expected that a new release will make some major improvements, but these might be not compatible with the current syntax.

There are two kind of expressions:

- xml expressions
- text expressions

### 2.2.1 XML expressions

A XML expression starts with a XML variable name (thus with a leading slash), selecting an existing XML variable, or a children using a path like construction. See the example below.

```
<!-- appends the entire input message to output -->
<output xml="/input"/>

<!-- appends a part of the input message to output -->
<output xml="/input/tuple/old"/>
```

These expressions are still very limited. There is no support for recurring elements with the same name, for selecting attributes or for selecting all children without the surrounding element.

### 2.2.2 Text expressions

Text expressions are more flexible, but still very limited, and has some serious limitations of the parser:

- if the expression contains one or more + signs, it is split into several sub expressions, which will be concatenated
- if an expression starts with / it is assumed to be a XML variable.
- if an expression contains :: it is assumed to be calling a static java function
- if an expression is surrounded by ${ and } it is supposed to be a text variable
- if an expression is surrounded by square brackets [ and ] it is supposed to be a constant string
- otherwise the entire text (without +, :: ${, etc symbols) is considered a constant string

A limitation is, that,because the first rule is to split around + signs, it is not possible to use a literal + sign, not even in a constant string.

# 3 XML construction commands

There are several commands that all use the same syntax for building XML:

- The `append` command appends data to existing XML variables.
- The `output` command appends data to the output XML variable.
- The `call` command creates a new SOAP XML request message and appends data to this XML before sending this request.
- The `createXmlVar` command creates a new XML variable and appends data to this variable.

All these commands use the same syntax, and can be used to build complete XML data. The syntax mainly consists of nested elements that help build the XML. There are 5 different elements:

- element, adds a new element
- attribute, adds a new attribute
- text, adds new text
- cdata, adds new cdata section
- include, adds the contents of a xml expression (this name might be changed in future)

A complete overview of all possible syntax is quite complicated. This is because the syntax is very powerful but also meant to be very intuitive to read. The syntax in semi formal notation is shown below. It is shown for the `output` command, but the `call`, `append` and `createXmlVar` command all share this syntax.

```
<output
    [name="element-name"]
    [text="text-expression"]
    [xml="xml-expression"]
    [childrenOf="xml-expression"]
>
  <!-- any of the following elements in any order, as many times as needed -->
  <element
    [name="element-name"]
    [text="text-expression"]
    [xml="xml-expression"]
    [childrenOf="xml-expression"]
  >
    <!-- nested elements, attributes, etc if needed -->
  </element>

  <attribute [name="attribute-name"] [text="text-expression"] />
  <text [name="element-name"] [text="text-expression"]>[fixed-text]</text>
  <cdata [name="element-name"] [text="text-expression"]>[fixed-text]</cdata>
  <include ..../> <!-- synonym for element -->
</output>
```

## 3.1 Attribute shortcuts

There are several attributes possible. Some of these attributes are all shorthand notations, which can also be accomplished by using children elements. The rules are as follows:

- If the name attribute is present, a child element with that name is created, and all content is added under this element.
- If a text attribute is present, the text of that expression is added to the element

An example without a name for a new element

```
<output text="hello world!"/>

<!-- is shorthand for -->

<output>
```

```
  <text text="hello world!"/>
</output>
```

```
<output xml="/input/field1"/>

<!-- is shorthand for -->

<output>
  <include xml="/input/field1"/>
</output>
```

```
<output childrenOf="/input/field1"/>

<!-- is shorthand for -->

<output>
  <include childrenOf="/input/field1"/>
</output>
```

```
<output name="data" .....>
  <!-- possibly extra elements -->
</output>

<!-- is shorthand for -->

<output>
  <element name="" ....>
    <!-- possibly extra elements -->
  </element>
</output>
```

And the same example with a new element name.

```
<output name="data" text="hello world!"/>

<!-- is shorthand for -->

<output>
  <element name="data" text="hello world!"/>
</output>

<!-- is shorthand for -->

<output>
  <element name="data">
    <text text="hello world!"/>
  </element>
</output>
```

## 3.2 examples

The example below shows some the main elements

```
<output name="demo">
  <attribute name="optional" text="true"/>
  <text text="Here one can use a text expression"/>
  <element name="bold" text="New York"/>
  <text>Static text can also be added in the element</text>
</output>
```

This construct will create a new element with the name "demo". In this element a new attribute named "optional" is created with  as value the text "true". Then some text is added, a new subelement named "bold" is created, and finally some more text is appended.

```
<demo optional="true">
  Here one can use a text expression
  <bold>New York</bold>
  Static text can also be added in the element
</demo>
```

Note that t

The `element`, `text` and `cdata` elements all have an optional `name` attribute. If this name attribute is provided, a new element is created and the content is appended to the new element. otherwise the content is appended to the current element. It might seem a bit strange for an element statement to have no name, because it's sole purpose would be to add a new element. However it can be very useful to add the content of an XML variable to an existing element, without adding a new element.

# 4 Command reference

## 4.1 `import` command

```
<import name="commandname" class="classname"/>
```

This command defines a new command with the name *commandname*. The command is implemented in a Java class given by classname. The exact specifications of such a Java class will be described elsewhere.

## 4.2 `default` command

```
<default attribute="attribute-name" value="fixed-text"/>
```

This is a convenience function, that allows to define a default value for certain attributes. Currently the following attributes support this mechanism:

- namespace
- async

- showSoap

## 4.3 `append` command

```
<append to="xml-expression"
    [name="name"]
    [xml="xml-expression"]
    [text="text-expression"]>
  <!-- see chapter 3 -->
  <element ..../>
  <attribute..../>
  <text ..../>
  <cdata..../>
  <include..../>
</append>
```

This command appends XML to an existing XML structure.

```
<append to="/output" name="city" text="New York"/>
```

```
<append to="/output">
  <element name="city" text="New York"/>
</append>
```

## 4.4 `output` command

```
<output name="varname" [xml="xml-expression"] [text="text-expression"]>
  <!-- see chapter 3 -->
  <element ..../>
  <attribute..../>
  <text ..../>
  <cdata..../>
  <include..../>
</output>
```

This is a convenience function that is a synonym for <append to="/output" .....

## 4.5 `call` command

```
<call method="method-name"
  [namespace="namespace"]*
  [async="true|false"]*
  [showSoap="true|false"]*
  [ignoreSoapFault="true|false"]*
  [appendMessagesTo="xml-expression"]*
  [resultVar="result-var-name"]
  [timeout="millisec"]

  [name="element-name"]
  [xml="xml-expression"]
  [text="text-expression"]
```

```
>
  <!-- see chapter 3 -->
  <element ..../>
  <attribute..../>
  <text ..../>
  <cdata..../>
  <include..../>
</call>
```

This command will execute a method call *method-name* in *namespace*. The response of this call will be stored in a variable *result-var-name*. If the `resultVar` attribute is not set, this will default to *method-name*.

### 4.5.1 `async` attribute

If the `async` flag is true, the script sends the request message and will not wait for the response, but will continue processing the next steps of the script. If the script evaluates an expression that refers to *result-var-name*, and this variable is not yet filled, the script will then block until a response message is received. The default value for the `async` attribute is false.

### 4.5.2 `showSoap` attribute

If the `showSoap` flag is true, the resultVar will be filled with the entire SOAP message, including Envelope, Header and Body elements. Because this info is usually not needed, the default behaviour is to set this variable to false, which will discard the SOAP elements and store the first child of the body in the ResultVar.

### 4.5.3 `ignoreSoapFault` attribute

If the `ignoreSoapFault` attribute is set to true, a SOAP:Fault response will not result in an error, but teh SOAP:Fault message will be stored in the resultVar. If this attribute is set to false (the default behaviour), the processing of the script will finish and a SOAP:Fault will be returned as result of the script.

### 4.5.4 `timeout` attribute

The timeout attribute sets an timeout value in milliseconds. If the method call does not answer within this timeout period, the script will stop with a SOAP Fault. If no timeout attribute is specified a system wide default timeout is used. This value is 20 seconds by default, but a different value may be configured (see Chapter 6).

### 4.5.5 `appendMessagesTo` attribute

The appendMessagesTo attribute is mainly meant for debugging purposes. It will append all request and response message to a XML element. Usually you would set this using the default command to "/output", during development, to see what is happening, and remove that default command, once the

function works.

```
<!-- remove the next statement after debugging is finished -->
<default attribute="appendMessagesTo" value="/output"/>

<!-- The input and output of these calls will be shown in the output -->
<call ...>
<call ...>
```

You might also set it to something like "/output/log", but the one should first append a "log" element to the output, e.g.

```
<!-- remove the next two statements after debugging is finished -->
<output name="log">
<default attribute="appendMessagesTo" value="/output/log"/>
```

Note that when using async calls, the order of the response messages is not defined.

### 4.5.6 Using the default mechanism

The `async, showSoap, namespace, ignoreSoapFault` and `appendMessagesTo` attributes will use a default attribute if set with the default command.
If no `async` attribute is set directly or using the default command, it will default to false.
If no `showSOAP` attribute is set directly or using the default command, it will default to false.
If no `ignoreSoapFault` attribute is set directly or using the default command, it will default to false.
If no `appendMessagesTo` attribute is set directly or using the default command, it will default to none, which means that nothing is appended. It is currently not possible to unset this attribute once set using the default mechanism.
If no `namespace` attribute is set directly or using the default command this is an error.

## 4.6 `delete` command

This command delete a specific node in a XML expression.

```
<delete node="xmlExpression"/>
```

## 4.7 `fault` command

This command will abort further execution of the script, and return a SOAP:Fault to the caller. Currently it is only possible to specify a faultcode and a faultstring (called the message). In future it might be possible to add additional fields (such as faultactor and details).

```
<fault code="fixedCode" message="textExpression"/>
```

## 4.8 `sleep` command

This is a very simple command, that just pause the execution of a script for a given interval in milliseconds. This is mainly useful for testing and debugging purposes.

```
<sleep millis="milliseconds"/>
```

## 4.9 `createXmlVar` command

```
<createXmlVar var="varname" value="xml-expression"/>
```

This command defines a new XML variable that can be used in XML expressions. In an older version this command was called xml. This could be confusing, and the new name seems more precise in what it does. It is expected that this command does not need to be used very often, so the slightly longer name is not a problem.

## 4.10 Experimental commands

The commands is this section do work. They might be removed in the future, or be changed in incompatible ways. They are currently not being used in any known methods.

### 4.10.1 `var` command

```
<var name="varname" value="text-expression"/>
```

This command defines a text variable, similar to the `xml` command that defines a XML variable. It is not sure if such variables are really needed, and the mechanism for expressions and variables might be changed in future.

### 4.10.2 `script` command

```
<script>
  <command ...> ...
</script>
```

This building block does not really have a function yet. It groups a number of commands, that are executed. This would be exactly the same if the script tag was omitted. This command is a leftover from the macro facility, and might be useful for conditional or looping constructs.

# 5 Examples

## 5.1 Hello World

```
<implementation type="RelayCall">
  <output text="Hello World!"/>
</implementation>
```

## 5.2 Echo

```
<implementation type="RelayCall">
  <output xml="/input"/>
</implementation>
```

# 6 Configuration

Currently only a single configuration is supported.

## 6.1 `script.cache` configuration

If this variable is set to true, implementation scripts are compiled to an internal representation on the first execution of a script, and this compiled representation is cached for future invocations. In theory this could give a better performance, but some simple tests did not show any noticeable performance improvements. The cache is cleared if the processor is reset.

The default value of this configuration variable is false. In this case for each invocation the script is compiled again. The big advantage of this setting is that during the development one does not need to restart or reset the SOAP processor each time the script has been edited (even though Cordys always warns that this is necessary).

## 6.2 `methodcall.timeout` configuration

This value is used to determine the default timeout in milliseconds when calling a method. The default value is 20000 (20 seconds).