SIEMENS

Teamcenter Gateway for Enterprise Applications Connectivity Guide

Contents

Messaging

EA Connection Handling

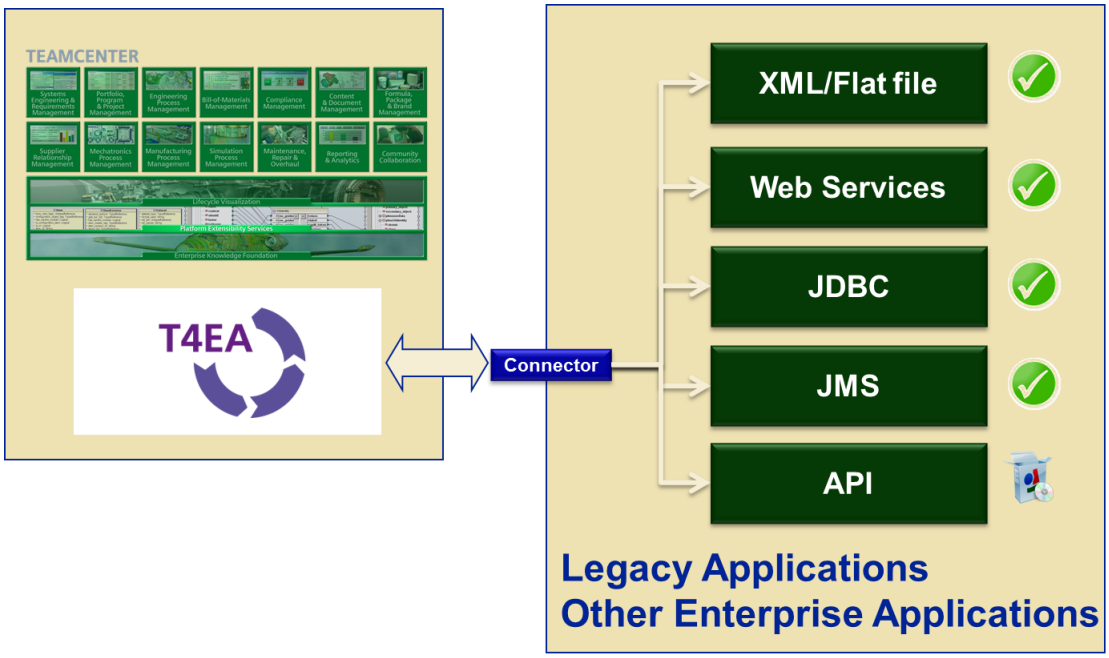
Preface

This documentation cannot be used as a substitute for consulting advice, because it can never consider the individual business processes and configuration. Despite our best efforts it is probable that some information about functionality and coherence may be incomplete.

Issue: July 2018

1. Introduction

This document explains the technical communication channels for connection T4x to Enterprise Applications. While for T4S and T4O the communication adapters are "out of the box", for the generic T4EA the communication layer has to be implemented depending on the requirements of the Enterprise Application. The T4x architecture for generic communication can be illustrated as follows (shown for T4EA exemplarily):



T4EA adds the connector to other Enterprise applications and ERP systems, based on different technologies like file exchange, web services, database connectors or messaging. APIs in different programming languages can theoretically be connected on the basis of the T4x framework, but customer specific software development is required for integration of a new API, so this happens very rarely and API integration is not covered in this document. Please contact Siemens Industry Software via GTAC if you have such requirements.

2. Synchronous vs. Asynchronous

Definition:

- Synchronous communication: The calling party requests a service, and waits for the service to complete. Only when it receives the result of the service it continues with its work. A timeout may be defined, so that if the service does not finish within the defined period the call is assumed to have failed and the caller continues.

- Asynchronous communication: The calling party initiates a service call, but does not wait for the result. The caller immediately continues with its work without caring for the result. If the caller is interested in the result there are mechanisms which we'll discuss in the next paragraphs.

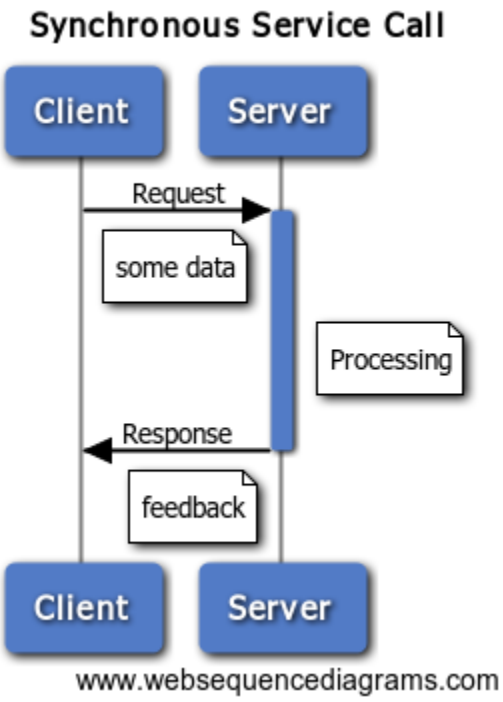
Be aware that the distinction between synchronous and asynchronous is highly dependent on the viewpoint. Often asynchronous is used in the sense of “the user interface must stay responsive all the time”. This interpretation often leads to the wrong conclusion: “...and therefore every communication must be asynchronous”. A non-blocking GUI usually has nothing to do with the low-level communication contracts and can be achieved by different means, e.g. parallel processing of the interactive and communication tasks. The truth is that synchronous communication on a certain level of abstraction can be implemented with asynchronous interfaces on another level of abstraction and vice versa, if needed.

File based communication is often considered to be asynchronous. One party writes a file but does not care if the other party is active, fetches the file or is able to process it. However it is possible to implement another layer of functionality so that the second (reading) party gives feedback, e.g. by writing a short result file, so that the first (writing) party can wait and poll for the result of the file processing. This layer introduces a synchronous communication over file exchange.

Communication over a database often is implemented by one party writing execution orders into a special table and the other party reads this table periodically and processes new entries, marking them as “done” or “failed” after execution. So far this is an asynchronous communication pattern. As soon as the first party waits for the result of the execution, this second layer introduces a synchronous communication pattern again.

The following chapters explain synchronous and asynchronous communication patterns in more detail, using web services as an example. The scenarios can also be used for other connectivity types.

Synchronous services are easy to implement, since they keep the complexity of the communication low by providing immediate feedback. They avoid the need to keep the context of a call on the client and server side, including e.g. the caller’s address or a message id, beyond the lifetime of the request.

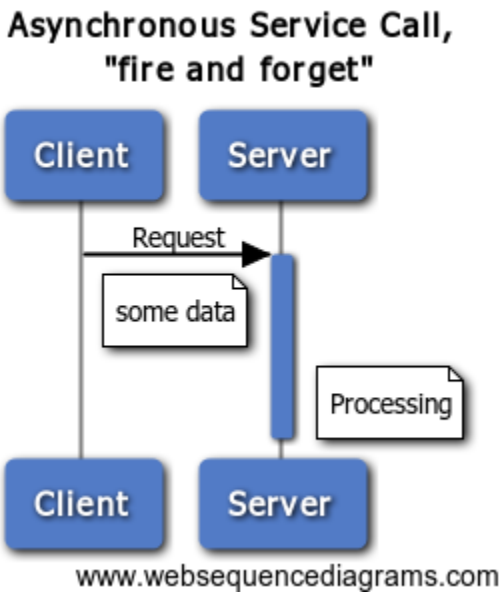


Nevertheless some circumstances may require implementing asynchronous calls:

- Expected round-trip durations are beyond time limits of the connection infrastructure (e.g. some web proxies close TCP connections after 2 minutes idle time)
- Connections with a lack of stability (e.g. a dial-in network connection is not available all the time).
- The caller is not interested in the result of the call or cannot wait for the result for some reason, e.g. it must free its resources.

In some cases the delivery of the asynchronous request can be assured by some other mechanism, e.g. message queuing, storing the request in a file system or creation of a T4x job.

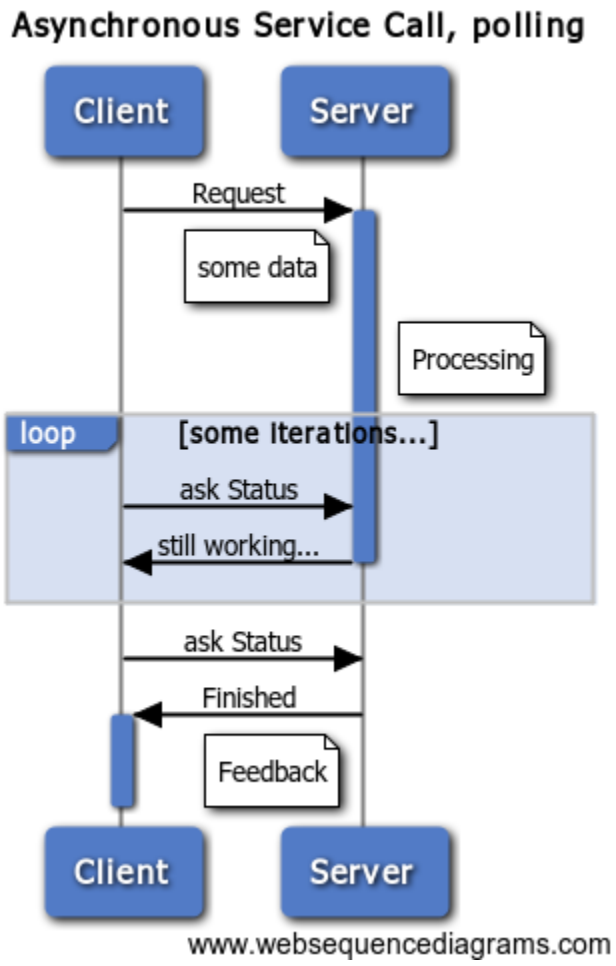
The simplest asynchronous message exchange pattern is called fire-and-forget and means that a message is sent but no feedback is required (at least on that level of abstraction!).



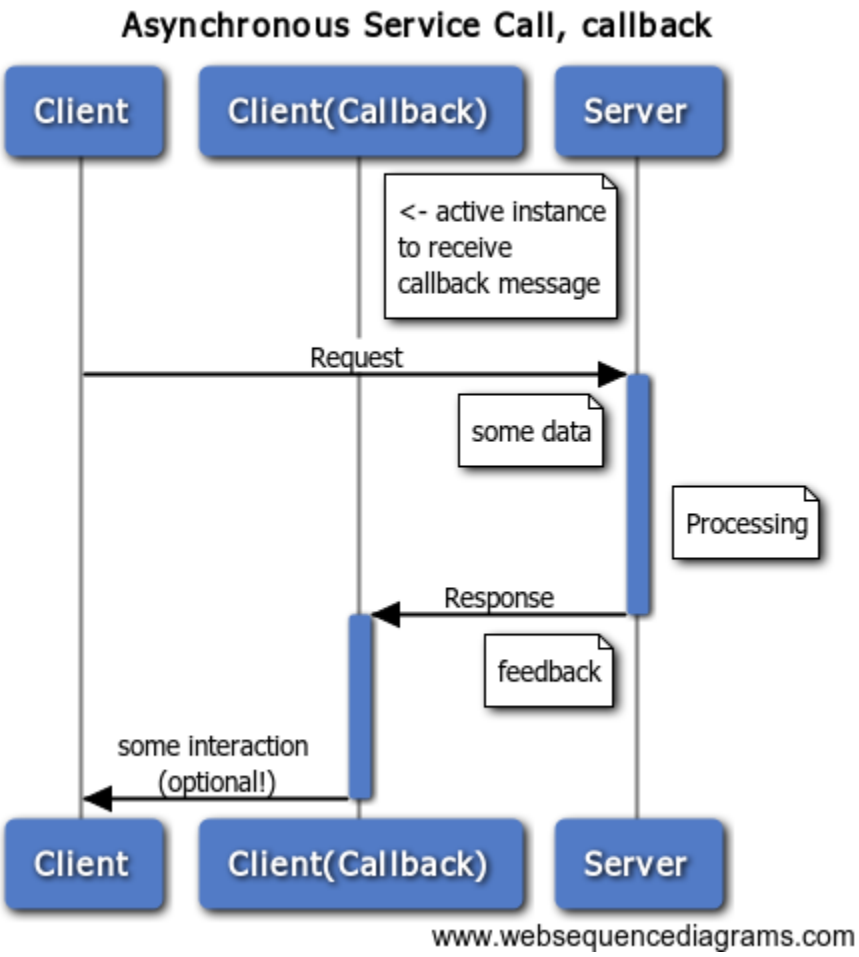
The only possible feedback can come from the communication layer in case of an error in processing or sending the request, but never from the processing of the server.

If feedback on the server-side processing is required using a fire-and-forget transmission, some higher level implementation must add the necessary logic and data to establish a kind of “session” to link the feedback to the request. There are two possible patterns to implement this: Either the client repeatedly asks for the result of the processing on the server (polling) or the server calls a service of the client to report the feedback after it has finished processing (callback).

Polling causes potentially high network loads and is therefore not recommended. Nevertheless it has the advantage that the service provider (server) does not need to know about its clients and that no client needs to provide a service by itself.



On the contrary for the callback pattern, the receiver of the request (server) must by some means know how to send the feedback message and must know how to address the correct client (this information can be passed in the request or be stored statically). To collect the feedback some active instance on the caller's side must listen to receive the feedback message (which in turn can be a fire-and-forget message). So the caller must become a service provider ("server") by itself. Usually the client continues with its work after the request was fired instead of waiting. So there can be some interaction between the client and the callback instance to notify the client or the user of the arrival of the feedback. This interaction happens entirely on the client and is usually not a communication issue (instead you can imagine sending notification emails, notifying the GUI, push a workflow task to the user's inbox or similar actions).



As previously stated the implementation of the message transfer may use synchronous or asynchronous transfers on a lower level. In the fire-and-forget example, the request might be transferred via TCP, which implicitly acknowledges each message. Even if the acknowledgment is being implemented, higher levels might not be interested in it. For the callback and polling scenarios, each message might be acknowledged, but from a high level perspective, there are only fire-and-forget messages.

Asynchronous behavior can be implemented for a T4x server by writing the message (input parameters) to the file system (where the T4x scheduler will poll for it) or by creating a job at the T4x job server. If the caller wants to be informed about the execution's result, the T4x server needs to store the caller's response address and some context information to be able to report the result back. This has to be done in the service implementation. T4x as consumer can handle this by providing a callback service. Another possibility is the caller periodically polling for the execution result, e.g. by looking for a result file in the file system or by asking the T4x job server for the result of a job identified by the job id. This can easily be done by providing an additional service asking for the job result.

3. File Exchange

Implementing file based integration usually includes that one component writes to a commonly used (shared) file system and another component watches this file system and starts to process files whenever they arrive.

If T4x is in charge of importing the file, you can use the T4x scheduler to periodically check the file system for existence of a file and start an import job for this file.

The following sample code reads a comma separated file named `filename` and transforms its content into a list of key/value pairs named `InputDat` (a TCL associative array could also be used), indexed with the line number and field number:

```
if {[catch [list open $filename r] fId]} {
  tpwrite "Cant open file $filename"
  return ERROR
}
tpwrite "File $filename opened"
set LineNo 0
set InputDat {}
while {![eof $fId]} {
  set LineContent [gets $fId]
  set FieldNo 0
  foreach Value [split $LineContent ","] {
    lappend InputDat [list ${LineNo}:$FieldNo $Value]
  }
}
close $fId
tpwrite "File $fname closed"
```

You can use this function in a scheduler script. For details on the scheduler configuration, see chapter **Script Based Triggers** in the **Teamcenter Gateway - Generic Configuration Guide**. To start the item import job, pass the variable `InputDat` to the following call (`ObjectKey` should contain the external ID of the object to be imported):

```
catch {set Status [::T4X::BATCHJOB::IMPORT::createBatchjob \
         T4EA_IMPORT_OBJECT $ObjectKey TC_IMPORT NONE TestMode \
         $InputDat]}
```

Of course, depending on your use case you might want to start a separate import job per line or implement another semantic interpretation of the file.

Note:

- If T4x writes the file, the enterprise application is responsible for watching the directory, so the scheduling is not covered here.
- To write a file from a transfer mapping, write the file in the performTransfer procedure of the mapping. Read the contents of the `ERPOutputDat` array (or any other internal variable), that was previously filled with values in the `TC_Object2EA_Object` procedure of the mapping. See variable `::WriteTransferFile2ERPTransferArea` in `t4ea_custom_services_demoerp.sd`, which is used to trigger writing the generated XML to a file.

The following code snippet writes the content of the list variable `OutputDat` to a single line in the comma separated file named `filename`:

```
if {[catch [list open $filename w] fId]} {
  tpwrite "Cant open file $filename"
  return ERROR
}
tpwrite "File $filename opened for write"

puts $fId "[join $OutputDat {,}]\n"
close $fId
tpwrite "File $fname closed"
```

If the file contains XML see the chapter **XML Processing Samples**.

Caution:

- Consider using directories ("`new`", "`inwork`", "`error`", "`processed`" or similar) to be able to distinguish between jobs yet to process, jobs in progress (working), failed and completed jobs.

- Make sure that the file import starts only once by moving the file to a "working" directory before reading or make the read/import process idempotent (make sure the result is identical even when run multiple times with the same input file). It might happen that the read/import gets started several times for example after a restart of the scheduler or if the processing takes longer than the time between two scheduler runs.

- In order to avoid reading an incomplete file, you have to implement additional synchronization mechanisms, like another "flag" file indicating that the file is properly written or by moving the file from a temporary directory to the exchange directory as soon as it is completely written (assuming that "move" is an atomic operation within a single file system, I.e: the operation will either complete successfully or fail without changing anything).

- If your file system does not guarantee an atomic move operation, all participating parties must implement and respect a common locking protocol (e.g. using a zero byte lock file) to make sure no partial files are read!

- When handling binary or non-US-ASCII files you should consider using the T4x API procedures `tpco_b64tobinfile` and `tpco_filetob64` and the encoding procedures `::base64::encode` and `::base64::decode`, because they will not destroy your multi-byte characters or break the file content at a binary "0". This especially makes sense in context of web services with files being received or transferred base64-encoded. See the **T4EA API Reference** for more details and examples of this.

4. Database Access

T4x uses the JDBC technology to access databases and provides an interface that makes the JDBC calls available in TCL. This allows integrating database access smoothly into the T4x import and transferring use cases. Installing a database is beyond the focus of this manual.

> Caution:
>
> Only in case a direct JDBC driver is not available, consider using the JDBC/ODBC Bridge (JDBC type 1 driver) because it has some drawbacks on portability and performance.

The T4x delivery does not contain any JDBC driver. You must download each required driver on your own and accept the licensing agreements. See the **Teamcenter Gateway - Installation Guide** for details on this. After having installed a JDBC driver, proceed like this:

- Activate the driver in the T4x mapping:

```
# Import the ::T4X::OBJECTS::* namespace
namespace import ::T4X::OBJECTS::*
# Load the JDBC module – add your driver's Jar name below! --
set Module [tpmodule de.tesis.plmware.objects.module.jdbc \
  com.whatever.database.Driver]
```

The last parameter to tpmodule may be a single class name of a driver or a TCL list of class names of drivers. The latter case allows to use drivers of databases from different vendors at the same time. When opening a connection (see next example) the correct driver will automatically be selected based on the connect string.

Open a connection and log in

```
set username "root"
set password "geheim"
set dbServerHost "localhost"
set database world
# replace "whatever" by your driver and review connection string!
set connUrl "jdbc:whatever://${dbServerHost}/${database}"
tplet Conn $Module getConnection $connUrl $username $password
```

Now you can execute SQL statements using the connection in your TCL mapping code:

```
# Sending an SQL query to the DBMS
tplet statement $Conn prepareStatement "select * from country where name
like ?"
tpwith $statement setString 1 "ger%"
tplet CollectionsResultSet $statement executeQuery
```

Caution:

- In order to avoid memory growth of the JDBC adapter, you have to close the statement and the result set after use:

```
tpwith $CollectionsResultSet close
unset CollectionsResultSet
tpwith $statement close
unset statement
```

- Whenever two or more transactions write to a database in parallel, deadlocks may occur. Usually deadlocks are at least reported by the database driver and will cause the T4x JDBC statement to fail. Even if T4x is involved in a deadlock situation, deadlocks can definitely not be prevented by T4x. Instead either your database or your integration mapping code must take precautions. See **https://en.wikipedia.org/wiki/Deadlock** and especially consider deadlock detection and prevention.

5. XML Processing Samples

5.1 Building XML Payload

When preparing the payload of an XML based service in TCL code, often TCL dicts (TCL associative arrays) of a bunch of discrete variables is used to store the values. This paragraph shows several ways to implement the transformation from the TCL variables to XML.

The decision for the right parser/generator depends on the structure of the XML schema. If your service was generated from a high-level programming language using a generic serializer for collection types, you will probably see semantic-free tag names like “item”, “array” and the like. We call this payload type “anonymous tags”. They may make it difficult to create an associative structure and pass the values to TCL while maintaining the semantics of the payload. If on the other hand the schema was created “by hand” you will most probably find tag names carrying names with high semantics like “costOfProduct”, “amountValue” and so on. These “named tags” make it easier to parse the XML and you can use simpler builders and parsers for such XML.

When building XML payload, remember that you have to escape certain characters (especially “<”) in your attribute or tag values! The easiest way to achieve this is using the standard function `::xmlgen::esc`.

The code samples in the following paragraphs build payloads suitable to be passed to the T4x generic service caller to call a web service that determines its operation from the outermost tag in the soap body (which is true for almost each web service). Of course it may be necessary to adapt the structure of the created XML, the parameters or namespaces.

> Caution:
>
> Make sure that error conditions do not invalidate the XML during generation. For example: if an error occurs after some opening XML tags have been written, sent or stored, you have to make sure that either the closing tags are written despite of the error or the whole result is replaced by some valid XML. Otherwise the communication partner will raise an XML parsing error. (For clarity this kind of error handling is dropped from the following code samples.)

Simple builder for XML with named tags

The simplest way to generate XML is to write a TCL procedure taking a set of parameters and returning the XML snippet:

```
proc buildPayload {ChangeStateID ProjectID} {

  set requestPayload "<q0:SomeUpdateRequest_sync xmlns:q0=\"http://mynamespace.com\">
    <Project>
      <ID>[::xmlgen::esc $ProjectID]</ID>
      <ChangeStateID>[::xmlgen::esc $ChangeStateID]</ChangeStateID>
    </Project>
  </q0: SomeUpdateRequest _sync>"
```

```
  return $requestPayload
}
```

If you have stored all parameters in a TCL associative array, you may want to pass the array to the function. This solution puts all key/value pairs of the array to the XML. It does not guarantee the sequence of the tags, which might cause collisions with the schema definition on the server side.

```
proc buildGenericPayload { Operation Namespace ArrayName} {
  upvar $ArrayName MyArray

  set requestPayload "<q0:${Operation} xmlns:q0=\"${Namespace}\">\n"
  foreach Key [array names MyArray] {
    append requestPayload "<${Key}>[::xmlgen::esc $MyArray($Key)]</$
{Key}>\n"
  }
  append requestPayload "</q0:${Operation}>"
  return $requestPayload
}
```

Generic builder for XML with anonymous tags

There is no general solution for generating XML with anonymous tags from a TCL array. The sample code here transforms a TCL list which contains key/value pairs (represented by lists with two elements: “{{key1}{value1}...}”) to a two-dimensional XML structure with the form “<item><item>key1</item><item>value1</item>...</item>”, as used by JAX-WS to serialize the input of a Java function with a single `String[][]` parameter:

```
proc build2DPayload { Operation Namespace ListOfPairs} {
  set payload "\
    <ns0:$Operation xmlns:ns0=\"${Namespace}\">”
      <arg0>"
  foreach KeyValuePair $ListOfPairs {
    append payload "<item><item>[::xmlgen::esc [lindex $KeyValuePair
0]]</item>”
    append payload "<item>[::xmlgen::esc [lindex $KeyValuePair 1]]</
item></item>"
  }
  append payload "
      </arg0>
    </ns0:$Operation>"
  return $payload
}
```

Use of the TCL standard ::xmlgen

For more sophisticated XML generation, you can use the TCL module ::xmlgen, which is available in the T4x environment. See the documentation: **http://tclxml.sourceforge.net/xmlgen.html**. Here is an example:

```
namespace import ::xmlgen::*
set xmlPayload {}
buffer xmlPayload {
  doTag Record xmlns=http://www.namespace.com/mySchema ! {
    foreach key [dict keys $propertyDict] {
      doTag Property name=$key - [esc [dict get $propertyDict $key]]
    }
  }
}
```

This will create an XML similar to the following in the string variable `xmlPayload`:

```
<Record xmlns="http://www.namespace.com/mySchema">
 <Property name="ArticleNo">4356872</Property>
 <Property name="Weight">7.5kg</Property>
 <Property name="Creator">Engineer</Property>
</Record>
```

5.2 Parsing XML Input

The input to services implemented in T4x or the response of services called by T4x is often XML. In TCL it is often easier to handle a TCL associative array or TCL dict instead of an XML string. In these cases we need a parser that extracts the relevant values from the XML.

> Caution:
>
> Never use pattern based matching to analyze or parse XML files! XML allows several notations of tags, several whitespace interpretations and some encodings and escape sequences for string values and characters. These will cause problems in your pattern matching algorithm. The parsers in the following sections are able to handle XML correctly and will avoid such problems.

`::TPXmlParse::str2array` interprets its first argument as XML and parses it to the TCL array named by the second argument. `::TPXmlParse::printXmlArray` logs the contents of the array to the T4x syslogfile for debugging purposes. The 8-digit number starting the XML key name is a serial number that is incremented with every new XML event. You can see the opening “return” tag represented by a separate number (“00000001”) followed by the navigation path (“test.echoResponse.return”) and the value being a list with the XML attributes of the tag (in key/value pairs in a single list, here: the type) and then pairs of XML specific additions (here: the namespace). If the tag had no attributes and no namespace prefix, the value would be empty “{{}}”. The value between opening and closing XML tag is represented by another entry (numbered 00000003) with the key ending with “.DATA”. Note that the value consists of a list since the XML value may be split in several parts, e.g. by newlines or embedded other tags.

Simple parser for XML

T4x contains a parser for simple XML structures which “serializes” XML to an array by generating a numbered key from the tag path. This results in a rather large TCL array but allows easy access to certain

keys and their values without the need to implement a specific parser. This parser is appropriate with tag names carrying semantics and where the XML does not contain large collections of tags.

See the following XML sample (a complete SOAP response):

```
<?xml version="1.0"  standalone="no"?>
<SOAP-ENV:Envelope
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:T4SSoap="urn:T4SSoap"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <ns:test_echoResponse xmlns:ns="T4SSoap"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                <return xsi:type="xsd:string">::T4STestSOAPServer::echo
                  just a test</return>
              </ns:test_echoResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Parse this with the following code snippet (note that only the content of the body (from `<SOAP-ENV:Body>` to `</SOAP-ENV:Body>`) is passed to the parser, see `var/test/Samples/t4x_service_test.tcl` for the source code):

```
::TPXmlParse::str2array $Response ::XIData
::TPXmlParse::printXmlArray ::XIData
```

This leads to the following output in the T4x logfile:

```
::XIData(00000000.test_echoResponse)= {{SOAP-ENV:encodingStyle \
  http://schemas.xmlsoap.org/soap/encoding/} -namespace T4SSoap -namespacedecls \
  {http://schemas.xmlsoap.org/soap/envelope/ SOAP-ENV T4SSoap ns}}
::XIData(00000001.test_echoResponse.return)= {{xsi:type xsd:string} -namespacedecls \
  {http://www.w3.org/2001/XMLSchema-instance xsi}}
::XIData(00000002.test_echoResponse.return.DATA)= {{::T4STestSOAPServer::echo
  just a test}}
```

Event based parser for XML

If the XML to parse is large, consists of tag names without semantics (as created from automatic serializers like JAX-WS or Axis) or if the XML contains collections of tags and you need to know the position of individual tags in the collection, a specific parser implementation might be required. You can implement such a parser using the event based `::xml::parser` from the TclXML package (this is a "SAX" parser, for details see: **http://tclxml.sourceforge.net/**). With this parser you can register several of your own TCL procedures which the parser calls when it reads certain XML constructs from the input, e.g. an opening or closing tag or the text content of a tag. Your functions are called with the necessary values and you can build your own structure as needed. Since the parser scans through the file only

once, it is fast and has a low memory footprint. It is up to your implementation to ignore or skip certain events and remember some data for other events. This allows you to build a small in-memory representation of only the relevant contents of the XML input. In many cases a representation as TCL associative array or TCL dict with the XML tags as (indexed) keys is easy to handle.

TCL array based parser for XML

Another way to parse the result of a web service or any other source providing an XML payload is the built in "XML to TCL array" converter provided by T4x. This approach can be used for XML payloads containing simple or structured attributes like a BOM structure. Example:

```
<t4e:GetProductStructureDataResponse>
<ID>123</ID> (simple attribute "ID")
<RevisionID>A</RevisionID>  (simple attribute "RevisionID")
  <Positions>
    <Position>
       <PositionNumber>1</PositionNumber> (structured attribute \
         " Positions/Position/ PositionNumber" )
       <ArticleID>4711</ArticleID> (structured attribute \
         " Positions/Position/ ArticleID" )
       <Quantity>1</Quantity> (structured attribute " Positions/Position/ Quantity" )
    </Position>
    <Position>
       <PositionNumber>2</PositionNumber> (structured attribute \
         " Positions/Position/ PositionNumber" )
       <ArticleID>4712</ArticleID> (structured attribute \
         " Positions/Position/ ArticleID" )
       <Quantity>2</Quantity> (structured attribute " Positions/Position/ Quantity" )
    </Position>
    <Position>
       <PositionNumber>19</PositionNumber> (structured attribute \
         " Positions/Position/ PositionNumber" )
       <ArticleID>4713</ArticleID> (structured attribute
         " Positions/Position/ ArticleID" )
       <Quantity>1</Quantity> (structured attribute " Positions/Position/ Quantity" )
    </Position>
  </Positions>
</t4e:GetProductStructureDataResponse>
```

In order to convert such a payload into TCL standard elements, the following built-in command can be used:

```
::T4X::XML::PARSER::getResultArrayInfo $xmlPayload ::BomResultDat
```

- `$xmlPayload` → XML Payload to be converted
- `::BomResultDat` → Global TCL array name which will contain the converted result

The following "global" TCL array will be created by the parser:

```
::BomResultDat(ID) = 123
::BomResultDat(RevisionID) = A
```

```
::BomResultDat(Positions: Position:PositionNumber:1) = 1
::BomResultDat(Positions: Position: ArticleID:1) = 4711
::BomResultDat(Positions: Position: Quantity:1) = 1
::BomResultDat(Positions: Position:PositionNumber:2) = 2
::BomResultDat(Positions: Position: ArticleID:2) = 4712
::BomResultDat(Positions: Position: Quantity:2) = 2
::BomResultDat(Positions: Position:PositionNumber:3) = 19
::BomResultDat(Positions: Position: ArticleID:3) = 4713
::BomResultDat(Positions: Position: Quantity:3) = 1
```

All "simple" attributes will be added to the TCL array by just using the name of the attribute. All "structured" attributes will be added using the complete path and an index representing the position within the former XML payload.

Using standard TCL commands it is now very easy to loop over this array in order to access each element in it:

```
#
# Process simple attributes
#
if { [info exists ::BomResultDat(ID)] } {
  set ID $::BomResultDat(ID)
}
if { [info exists ::BomResultDat(RevisionID)] } {
  set RevisionID $::BomResultDat(RevisionID)
}
#
# Process structured attributes
#
foreach Line [::T4X::CORE::sortIndexedInterfaceTable [array
names ::BomResultDat \
    Positions:Position:ArticleID:*]] {
  #
  set LineList [split $Line :]
  set LineIndex [lindex $LineList end]
  #
  set ArticleID $::BomResultDat(Positions:Position:ArticleID:$LineIndex)
  set PositionNumber
$::BomResultDat(Positions:Position:PositionNumber:$LineIndex)
  set Quantity $::BomResultDat(Positions:Position:Quantity:$LineIndex)
  #
  # Now we can process ArticleID, PositionNumber and Quantity of this
BOM line
  #
}
```

TCL tDom parser for XML

The most advanced parser is the TCL-internal tDom parser. This parser allows to access the XML in a object-oriented manner and even supports XPath expressions. However it requires some basic understanding of the "DOM" concept in parsing XML. If you are used to these concepts from other programming languages you will be most productive using this parser. See **http://docs.activestate.com/activetcl/8.5/tcl/tdom/tdomcmd.html** for a detailed description or **http://wiki.tcl.tk/8984**for a tutorial and further readings.

Note:

- It is not necessary to include (with TCL `package require`) any additional component to use this parser - T4x already has everything on board.
- Because of internal namespace issues, the tDom parse command must be called using `xdom parse` (see example below). All other tDom commands can be used 1:1.
- If you need to pass literal strings to tDom texts or attributes from TCL source code, all non-7Bit ASCII characters have to be escaped "JavaScript"-like, e.g. instead of `$someNode createTextNode "€"`, use `$someNode createTextNode "\u20AC"` instead. There are online converters like e.g. **http://r12a.github.io/apps/conversion/**.

The following example extracts some fields from a WSDL in `$filename`:

```
# load xml file with correct encoding,
# variable encStr will contain the xml file's encoding!
set val [tDOM::xmlReadFile $filename encStr]

#parse xml into variable wsdldom
xdom parse $val wsdldom

set defs [$wsdldom getElementsByTagName wsdl:definitions]
set tcon [$defs getElementsByTagName wsdl:types]
set mess [$defs getElementsByTagName wsdl:message]

foreach mess [$defs getElementsByTagName wsdl:message] {
     puts [$mess getAttribute name]
```

6. Web Services

6.1 Introduction

From a high-level developer's point of view a web service can be seen as a remote procedure call or a remote API. T4x can provide web services (T4x acts as the server and the enterprise application calls some service on the T4x system) and call (consume) web services (T4x acts as a client and calls a service running on the enterprise application host).

There are numerous interpretations of the term "web services". Almost all interpretations can be handled with T4x.

T4x has the concept of server instances. A server instance represents a port (like e.g. 11301) and its configuration regarding SSL and authentication on the GS. One or several services can be assigned to an instance. If you want to provide services, you have to make sure your services are assigned to the right instances and are activated. Services can be assigned to none, one or several instances. More details can be found in the online help of the GS Admin UI.

6.2 Assign and Withdraw Access Rights for a Service

By default any service (SOAP or REST) provided by the T4x custom mapping will be available on every server instance where the service is assigned. Whether a remote user can access the service depends on the configuration of the server instance. If the instance requires authentication is editable in the instance's configuration, see "Basic Authentication" On/Off. This setting applies to all users configured on the BGS (Configuration/User Management).

If you need to assign certain services to certain users only (and disallow the service for other users), you can do this with the command:

```
::SYSUtils::bindProcToUser procname user
```

Where `procname` is the name of the implementing TCL procedure (not the service name) and `user` is the name of the user. This call can be placed in `t4ea_mapping_config.sd` for example. If the corresponding instance requires authentication, the configured user will still have to authenticate each HTTP call to the service, but no other user can access the service, even if authenticated correctly.

7. Simple Web Services

7.1 Introduction

Simple services in the CGI style (HTTP/HTTPS transfer with GET and parameters encoded in the URL or POST with XML or other payload) are provided and received by the T4x Gateway Service by creating a proxy function or by directly calling the TCL HTTP client.

Note that this enables you to call and provide typical REST services. Simply speaking REST is an abstraction of communication systems and requires to address resources via URL, defines different actions represented by the HTTP actions (GET, PUT, POST, DELETE and others) and the service response may optionally direct the client to other useful URLs. The World Wide Web and its HTTP/HTML standards are typical samples for a REST based communication system. In a weaker sense REST is often used as a synonym to HTTP based communication using XML payloads.

PXML (for “Plain XML”) is just another T4x-internal synonym for this kind of communication.

7.2 T4EA PXML server (T4EA provides a service)

The following TCL procedure implements a HTTP(S) service that just concatenates its three input parameters (`arg1, arg2, arg3`) and the posted data and returns the result in an XML structure:

```
proc sampleService {args} {
  # set default values
  set result ""
  set Status OK
  set post ""
  set arg1 ""
  set arg2 ""
  set arg3 ""

  # parse for supported parameters
  foreach arg $args {
    switch [lindex $arg 0] {
      post         {set post         [binary format H* [lindex $arg 1]]}
      arg1         {set arg1         [lindex $arg 1]}
      arg2         {set arg2         [lindex $arg 1]}
      arg3         {set arg3         [lindex $arg 1]}
      default      {set Status "ERROR"}
    }
  }
  # handle mandatory arguments, if needed
  if {[string length $post] == 0} {
    tpwrite "Error: no data posted"
    set Status ERROR
  } else {
```

```
    # do something with the values $post, arg1, arg2, arg3
    set result "Result: arg1: >$arg1< arg2: >$arg2< arg3: >$arg3< post: >
$post<"
  }
  # create the http answer
  set response [::TPHAP::XMLResHeader]
  append response "<result>$result</result>\n"
  append response "<status>$Status</status>\n"
  append response [::TPHAP::XMLResFooter]
  return [::TPHAP::createHeader "" "" "text/xml" [string bytelength
$response] \
    "" ""]$response
}; ::ProxyGen::createServer ::PXML::sampleservice ::T4EA::sampleService
```

The call to `::ProxyGen::createServer` creates a PXML server from the shown TCL function under the name “sampleService”. The function gets all URL-encoded HTTP parameters as key-value pairs in the input parameters. The posted value is passed to the function as if it were another HTTP parameter named “post”, but additionally you have to convert it from its hexed format (tpco_formatHEX16 or other functions which are able to work with HEX16- encoded binary data). The response of the function can contain any data – the sample shows a useful example how to return XML data.

If you need more control over the response, like returning HTTP error code 500 or additional HTTP headers, please review `::TPHAP::createGenericHeader` in the **T4EA API Reference**.

Caution:

- The service name must be all lower case. So use `::PXML::sampleservice` instead of `::PXML::sampleService`!
- You must assign your service to a server instance via the configuration in the GS Admin UI to be accessible.

7.3 T4EA HTTP client (T4EA consumes a service)

You can call HTTP services by creating an HTTP Client. You can use this client then from TCL like any other TCL procedure. To create the client, use the function `tpco_httpClient`. For details on the usage see the **T4EA API Reference**.

Here is an example to get the Siemens home page HTML via a proxy called proxy and port 80 into a string variable:

```
set HomePage [tpco_httpClient -proxy proxy.siemens.com:80 http://
siemens.com/ ]
```

If you want to post or put XML content, remember to set the HTTP Content Type attribute:

```
tpco_httpClient -user *** -pass *** -exheader [list "Content-Type: text/
xml"] \
  -post $xmlContent http://someurl.com/test.xml
```

Whenever you need to add parameters to the URL, you must escape the strings for parameter *names and values*. Otherwise any space, "/", "&"; "=" and probably more will be misinterpreted. T4x has a specific conversion for this: `tpco_urlencode`. For example instead of using

```
set URL "http://host/myservice?sentence=this is my
sentence&date=2015/03/20"
```

you should do this:

```
set URL "http://host/myservice?sentence=[tpco_urlencode \
  {this is my sentence}]\&date=[tpco_urlencode {2015/03/20}]"
```

The resulting URL will look like this:

```
http://host/myservice?sentence=this%20is%20my
%20sentence&date=2015%2F03%2F20
```

Never escape the whole URL as this will also escape the "/", "&" and "=" characters which are a necessary part of the URL!

If you are interested in the HTTP response code or want to parse the result headers, look at this sample:

```
set ret [tpco_httpClient -HINFO $URL]
puts ***PAYLOAD***
puts [lindex $ret 0]
puts ***HEADER***
puts [lindex $ret 1]

set hd [::SYSUtils::parseHttpHeader [lindex $ret 1]]
foreach key [dict keys $hd] {
  puts "$key \t=> [string trim [dict get $hd $key]]"
}

if {[dict exists $hd URL]} {
  set URLList [split [dict get $hd URL]]
  puts "URL Protocol: [lindex $URLList 0]"
  puts "URL Code:     [lindex $URLList 1]"
  puts "URL Status:   [lrange $URLList 2 end]"
}
```

Note that the option `-HINFO` makes `tpco_httpClient` return a second argument with the original HTTP header as received "on the line". `::SYSUtils::parseHttpHeader` is used to parse this header

into a TCL dictionary with the special key “URL” created for the initial header line, which contains the HTTP response code by definition.

Executing this sample code with the URL variable set to "**http://www.siemens.de**" gives the following result:

```
***PAYLOAD***
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.siemens.com/entry/de/de/">here</a>.</p>
</body></html>
***HEADER***
HTTP/1.1 301 Moved Permanently
Location: http://www.siemens.com/entry/de/de/
Content-Length: 243
Content-Type: text/html; charset=iso-8859-1
Connection: keep-alive
URL         => HTTP/1.1 301 Moved Permanently
Location     =>  http://www.siemens.com/entry/de/de/
Content-Length    =>  243
Content-Type     =>  text/html; charset=iso-8859-1
Connection     =>  keep-alive
URL Protocol:     HTTP/1.1
URL Code:         301
URL Status:       Moved Permanently
```

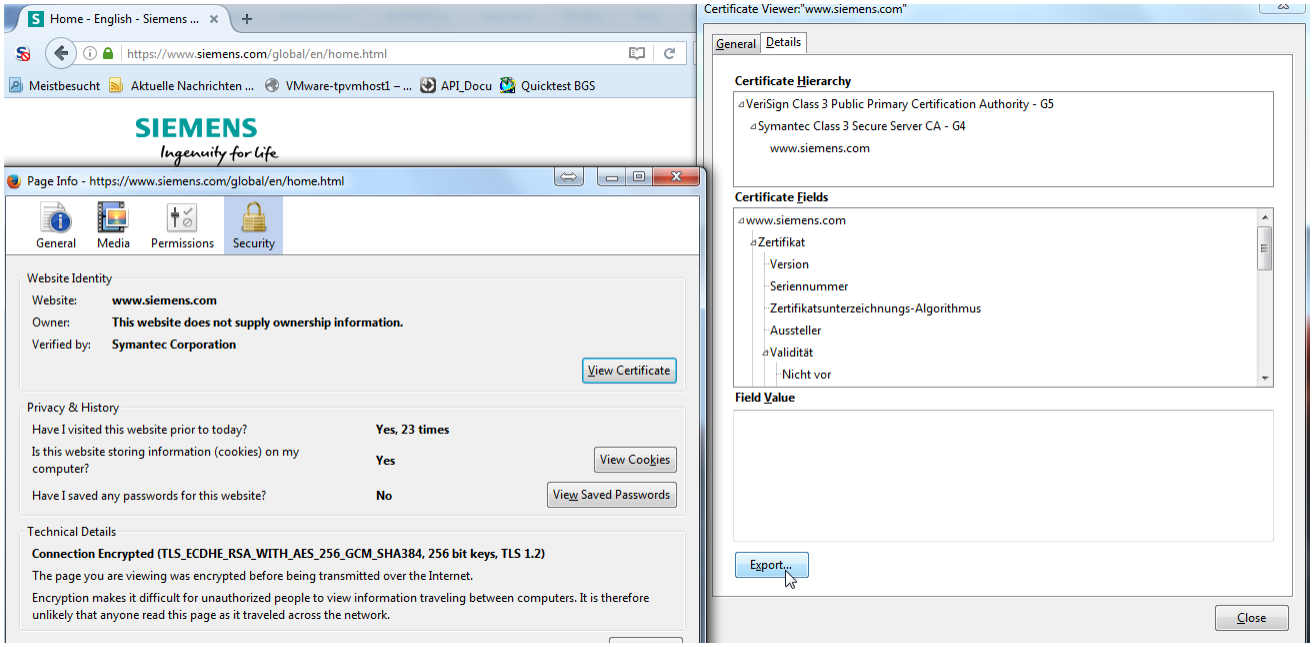
All payload transferred via HTTP post, put, etc. will be encoded in UTF-8 automatically (given the payload is correctly created in the TCL layer!). Currently there is no way to specify another payload encoding.

7.4 SSL support in the T4EA HTTP client

By default `tpco_httpClient` does not evaluate any SSL certificate presented to the client. However the options `-cert` and `-clientcert` can be used to force the check. If you need to use HTTPS or SSL, you should make sure to check the certificate, otherwise your connection is not secure!

T4x in this version does not manage certificates in a store like browsers or the Java VM do. Instead you must pass a certificate with every call to `tpco_httpClient`. Note that there are several formats of certificates. You probably have to convert certificates. This conversion is outside the scope of this document; the openssl libraries usually do a good job for several tasks with SSL and TLS security.

The usual HTTPS/SSL communication as implemented in the browsers is called one-way SSL, because the server has a certificate which the client must trust. You can download such a certificate using a web browser (e.g. in Firefox, click right mouse button on any HTTPS page, select “Page Info”/”Security”/”View Certificate”, change to “Details” tab and click “Export”, use pem format and choose to download the certificate with chain):



A call to a typical SSL server in a tpshell prompt looks like this:

```
tpco_httpClient -cert etc/cert/siemens.pem https://www.siemens.com
```

The client evaluated the downloaded certificate by comparing the received certificate with the given file and if they match, it trusts the connection and establishes it.

In server-server-communication it is not unusual to have so-called two-way SSL. Here the server and the client both have a certificate generated for their hosts and when establishing the communication, both the client and the server check if they trust in the certificate presented by each other. In such a case `tpco_httpClient` must be called with two certificates: the server certificate with `-cert` and the client certificate (a certificate created for the T4x host!) with `-clientcert`:

```
tpco_httpClient -cert etc/cert/server.pem -clientcert etc/cert/
client.pem \
  https://someserver/myservice
```

Caution:

- SSL/TLS communication must be handled with extreme care in order not to open any security issues!
- Make sure you use the right certificates only and take care not to pass your private keys!

8. SOAP Support

8.1 Introduction

T4x supports providing SOAP services and consuming SOAP services as client with two different technologies:

- “TCL to SOAP”: A direct TCL implementation for providing SOAP services, where the WSDL is generated from TCL code. On arrival of a SOAP message T4x calls the TCL implementation procedure for an operation with a TCL dictionary containing the parsed XML payload of the request. The implementation procedure returns a dictionary, which is mapped to the XML response. This solution is limited in its standard conformance (e.g. no MTOM, WS-*, etc) and is deprecated since T4x 11.2.2. New features (protocols, standards, extensions, configuration options) will no longer be implemented in this solution. However to be able to continue with existing integrations the solution will be available for some future releases.
  For consuming SOAP services T4x provides a (deprecated) Java adapter that uses the Apache AXIS2 framework. The procedure `::T4X::SOA::performGenericWebServiceCall` takes an XML payload and sends it to the server via a separate Java pipe process attached to the current worker. This approach misses some features (like MTOM, WS-*, etc.) and has a relatively high memory load. It is deprecated since T4x 11.2.3, but will be maintained until further notice.

- “WSDL to T4x”: A mixed Java/TCL implementation where large part of the SOAP standards are handled by the embedded Apache CXF framework. Here the WSDL must be generated externally and has to be imported to T4x with a T4x command line utility. New implementations for SOAP services should use this approach since it offers most control over the SOAP messages and has a considerable lower memory footprint. New feature requests (new protocols, standards, extensions, configuration options) will only be handled for this approach.
  On arrival of a SOAP message T4x calls the embedded CXF module which in turn calls the TCL implementation procedure with XML strings representing body and headers of the request. The implementation procedure returns an XML string which goes into the response.
  For consuming SOAP services this approach should be used, too. Use wsdl2t4x to import the server's WSDL and create some TCL code examples.

Both approaches can be used in parallel, but it is recommended to move existing integrations from the direct TCL implementation "TCL to SOAP" to the CXF solution "WSDL to T4x". To achieve this, the WSDLs generated in the “TCL to SOAP” approach or provided by the external SOAP servers must be imported with the “WSDL to T4x” tool chain and the service implementation procedures and the client procedures must be adapted since the signatures and the XML handling are different.

8.2 T4EA Provides SOAP Services

8.2.1 WSDL Download

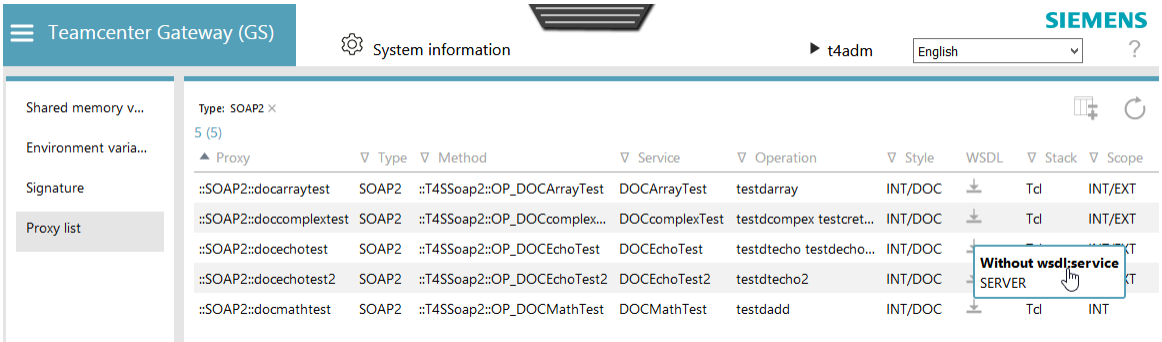
With both SOAP server approaches “TCL to SOAP” and “WSDL to T4x”, you can see the service and download the WSDL from the Admin UI. T4x automatically generates a tailored WSDL for each service

provided. In the T4x Gateway Admin UI, select “System Information/Proxy List”. In the “type” column select to display only “SOAP2” proxies. You will see all SOAP2 services listed, i.e. all services provided by the “TCL to SOAP” and “WSDL to T4x” approach. Click on the download link (a small downward arrow in the table) of a service and a popup will appear, showing several options:

- “Without wsdl:service” provides a link to download the so-called “design” WSDL, i.e. a WSDL that does not contain any information on the endpoint URL, bindings and ports. This WSDL will contain only the schema part of the WSDL and is useful for creating clients in an early development phase. If the service is not bound (assigned) to any server instance only this entry will be available in the popup.

- Zero, one or several entries with the name of an existing server instance will provide links to WSDLs including endpoint URLs tailored for the selected server instance.

Here is an example of this screen showing the popup for a service bound to the SERVER instance:



Note:

- T4x always exposes services via HTTP or HTTPS depending on the instance, the services are assigned to. See the online help in the Admin UI for details on server instances.

- The WSDL URLs provided by the Admin UI differ from the usual WSDL URLs:

  Admin UI WSDL URL:

  https://<T4x-host>:11321/pxml/getsoapwsdl?soapstack=SOAP2&service=DOCEchoTest2

  Canonical WSDL URL:

  http://<T4x-host>:11321/soap2/docechotest2?wsdl

  The difference is imposed by the concept of several server instances in a GS server that bind to different ports and need their own separate authentication. The Admin UI presents the WSDL URL in a convenience way to avoid that the user has to enter his/her credentials a second time, just to view the WSDL. However the canonical URL is always available as well. When downloading the WSDL programmatically, e.g into SoapUI, you can and should use the canonical URL. The content of the canonical WSDL URL is identical to the content of the instance specific download URL. So the canonical URL points to a WSDL with the correct endpoint definition for the used URL.

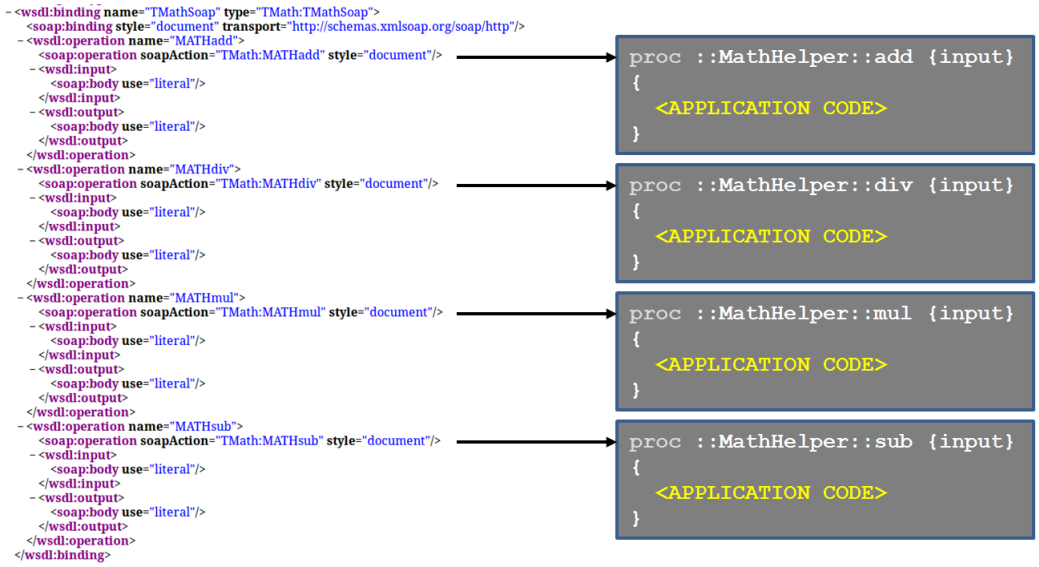
> Caution:
>
> Make sure your service is assigned to at least one server instance in the function binding in “Configuration/Server Instances” in the GS Admin UI to be accessible!

8.2.2 TCL to SOAP Mapping

8.2.2.1 Introduction

> Note:
>
> This technology is deprecated. If you are starting the implementation of a new SOAP service with T4x you should skip this chapter and proceed with the chapter **WSDL to T4EA**. Existing services based on the TCL to SOAP mapping will continue to work.

The function `::ProxyGen::createServer` binds a TCL server function to an operation into a SOAP service. T4x generates the SOAP service and the WSDL automatically. The WSDL is provided with an URL like this: **http://<T4x-host>:11301/soap2/<servicename_in_lowercase_letters>?wsdl**.



A SOAP service can contain many operations and therefore many bindings to TCL functions.

> Note:
>
> In the following chapters SOAP1 and SOAP2 do not denote any standards but are different implementations of the TCL to SOAP implementation. SOAP1 (TCL-Namespaces `::T4SSoap` and `::SOAP`) supports the older RFC-style encoding, SOAP2 (TCL-Namespace `::T4SSOAP2` and `::SOAP2`) supports the newer document-style encoding (recommended).

8.2.2.2 SOAP1 RPC style interface definition

For SOAP, T4x supports the data type's string, int and double. To define the data types for the input and output interface, these will be passed as a TCL list.

```
[list string string int double]
```

The interface expects four arguments with the data types `string, string, int, double`.

To define the SOAP service interfaces, the `-atype` and `-rtype` arguments can be used.

- `-atype` defines the input interface
- `-rtype` defines the output interface

Example

```
proc ::MathHelper::add {a b} {
    return [expr $a + $b]
}; ::ProxyGen::createServer ::SOAP2::MATHadd ::MathHelper::add \
                            -atypes [list double double] \
                            -rtypes [list double] \
                            -service TMath
```

This implementation defines a SOAP operation with two input arguments (`double, double`) and with one output argument (`double`). The service name is `TMath`.

URL: **http://<T4x-host>:11301/soap2/tmath**

8.2.2.3 SOAP2 document style additional interface definition

The SOAP2 document style interface definition has the additional possibilities to define records and arrays. For the definition of records, the function `::T4SSoap2::typeDef` can be used.

A record is defined with a TCL list:

```
{<argname0> {type <type0>} <argname1> {type <type1> … }
```

Records can also contain records (nesting).

Example

```
::T4SSoap2::typeDef HostS Login [list user        [list type string] \
                                      accessmask  [list type int] \
                                      logincnt    [list type int ]]
```

```
::T4SSoap2::typeDef HostS Host  [list hostname    [list type string] \
                                      account     [list type Login]]
```

The record `Login` contains three entries. The entry `Host` is also a record and contains two entries.

The declaration `<type>()` defines an array. This must contain at least one entry. The number of the entries is infinite. To define an array with boundaries, the boundaries have to be inserted in the declaration.`<type>(min,max)`.

- min 0 = means the array is optional > 0 minimum number of entries
- max -1 = means infinite number of entries > min maximum number of entries

Example

```
::T4SSoap2::typeDef HostS Login [list user        [list type string] \
                                      accessmask  [list type int] \
                                      logincnt    [list type int ]]

::T4SSoap2::typeDef HostS Host  [list hostname    [list type string] \
                                      account     [list type
Login(1,255)]]

proc store {args} {
  # APPLICATION CODE
  return 0
}; ::ProxyGen::createServer ::SOAP2::netstore ::Network::store \
                            -atypes [list Host] \
                            -rtypes [list int] \
                            -service HostS
```

The array `account` in the record `Host` is defined to min=1 and max=255.

```
- <xs:complexType name="Host">
  - <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" name="hostname" type="xs:string"/>
      <xs:element maxOccurs="255" minOccurs="1" name="account" type="HostS:Login"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="Login" type="HostS:Login"/>
- <xs:complexType name="Login">
  - <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" name="user" type="xs:string"/>
      <xs:element maxOccurs="1" minOccurs="1" name="accessmask" type="xs:int"/>
      <xs:element maxOccurs="1" minOccurs="1" name="logincnt" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="netstoreRequest" type="HostS:netstoreRequest"/>
- <xs:complexType name="netstoreRequest">
  - <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" name="v0" type="HostS:Host"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="netstoreResponse" type="HostS:netstoreResponse"/>
- <xs:complexType name="netstoreResponse">
  - <xs:sequence>
      <xs:element maxOccurs="1" minOccurs="1" name="netstoreResponse" type="xs:int"/>
    </xs:sequence>
  </xs:complexType>
```

This WSDL (typedef) is generated by the T4x server.

8.2.2.4 SOAP to TCL argument/return mapping

T4x maps a SOAP message in a TCL list structure {KEY VALUE}.

- SOAP1: In the T4x SOAP1 implementation, the top-level arguments have no names. They are mapped by the order.
- SOAP2: T4x insert in the SOAP2 implementation an automatic generated name (v0 ... vn) for the top-level arguments into the WSDL. In the TCL function interface, T4x maps the SOAP entries to the function arguments and hide the top-level names.

Record argument mapping:

```
{<recordname> {<key0> {type <type0>}}
   {<key1> {type <type1>}}
     {<key2> {type <type2>}}
       ...
       }
```

mapped to

```
{<recordname> {<key0> <value0>} {<key1> <value1>} {<key2> <value2>} ...}
```

Array argument mapping:

```
<key> {type <type([min,max])>}
```

mapped to

```
{<key> {<val0> <val1> <val2> …}}
```

8.2.2.5 T4EA as SOAP server

SOAP services can easily be provided by the T4x Gateway Service. The T4x Gateway Service supports SOAP RPC and DOCUMENT style. `::ProxyGen::createServer` exposes any TCL function as a SOAP service. The first parameter to `createServer` is the name for the generated proxy/operation function. For SOAP services this function must be in the `::SOAP[2]` namespace. The second parameter is the fully qualified name of the implementing function. For details on the usage see the **T4EA API Reference**.

The following sample function implements a simple SOAP service. The plain TCL function `echosif` takes three parameters (a string, an integer and a double), concatenates them and returns the resulting string.

```
proc echosif {echo_string echo_int echo_double} {
  return "[namespace current]::echosif $echo_string $echo_int
$echo_double"
};
```

RPC Style:

```
::ProxyGen::createServer ::SOAP::test_esif :: MyNameSpace::echosif \
  -atypes [list string int double]
```

DOCUMENT Style:

```
::ProxyGen::createServer ::SOAP2::test_esif :: MyNameSpace::echosif \
  -atypes [list string int double]
```

Here is an example for DOCUMENT Style and arrays. `test_array` is a array of strings and cnt is of type int.

```
proc echoarray {test_array cnt} {
  return "$test_array -- $cnt"
};
::ProxyGen::createServer ::SOAP2::echoarray ::MyNameSpace::echoarray \
  -atypes [list "string()" int]
```

The function `::T4SSoap2::typeDef` (see **T4EA API Reference**) can be used to define SOAP records (DOCUMENT Style only).

Here is an example that uses arrays and structs. First, we define an input data record (`InputDat`).

```
::T4SSoap2::typeDef TestService InputDat [list description [list type
string] \
                                                value       [list type
float] \
                                                counter     [list type
int]]
```

Then we define an output data record (`OutputDat`).

```
::T4SSoap2::typeDef TestService OutputDat [list echo       [list type
string] \
                                                 ctime      [list type
int]]
```

The service function copies the `input` into the `echo` variable and sets the `ctime` variable.

```
proc echorecord {input} {
  return [list echo $input ctime [clock seconds]]
};
```

Finally, we define the SOAP service. The `InputDat` record is defined as an array.

```
::ProxyGen::createServer ::SOAP2::echorecord ::MyNameSpace::echorecord \
  -atypes [list InputDat()] -rtypes [list OutputDat] -service TestService
```

Caution:

- Although the parameter `-host` for the call `::ProxyGen::createServer ::SOAP2` is optional in theory, you *must* use it if you want to create a WSDL that will be published to clients and used in several versions or on different servers – i.e. probably *in all cases*! The auto generated namespace will contain the current GS's hostname and port. A client generated for the resulting WSDL will not be able to communicate successfully if you transport the unchanged mapping to a different server or change the port of the service (server instance). The naming of this parameter is misleading as it really denotes the XML and WSDL namespace the new service will use.

- For more details on XML namespaces, see e.g. **https://en.wikipedia.org/wiki/XML_namespace**.

- As a simple recommendation you can use the parameter and value `-host {http://<customer_domain>/ActiveIntegration/<service>}` or `-host {http://<customer_domain>/soap2/<service>}` where `customer_domain` stands for your customer or company domain and `service` denotes the service name or the integration use case.

- If your service has more than a single operation, in all `createServer` calls for this service, the parameters `-host` and `-service` must be identical, *including case*!

- The value for the `-service` parameter must be identical to the name used in the corresponding `::T4SSoap2::typeDef` call, *including case*!

- The service name must be all lower case. So use `::SOAP2::sampleservice` instead of `::SOAP2::SampleService` in the `::ProxyGen::createServer` call!

8.2.3 WSDL to T4EA

This approach is geared towards the typical „WSDL-to-XYZ"-tool chains used for code generation from a given WSDL in Java or .NET environments. The generated code in Java or .NET allows parsing and creating the payload as a structure of objects with typed attributes. TCL however does not provide object-orientation and so the generated procedures are only stubs and are less restrictive (as they assume passing the request and response payload as XML strings instead of typed objects). The WSDL to T4x approach assumes and requires the WSDL to be generated outside of the T4x environment (or by the deprecated "TCL to SOAP" mapping). The WSDL must then be imported to T4x which generates stub implementations and adapts and stores the WSDL internally. This chapter handles the SOAP server implementation; a similar approach is also available for SOAP clients, see chapter **T4EA Consumes SOAP services**

Note:

- The T4x SOAP server always returns UTF-8 encoded responses, regardless of the request encoding.
- The SOAP server only supports SOAP 1.1 and 1.2, XML over HTTP. No other protocols (like REST or JMS) are supported.
- The JVM (JDK, see next “Caution box”) only starts up after the first SOAP call comes in. This may take some seconds causing the first SOAP call after the T4x GS startup to last considerably longer.
- If you need to create a WSDL, you can use lots of different programming languages and their web service toolkits or use the Eclipse WSDL editor: **http://wiki.eclipse.org/Introduction_to_the_WSDL_Editor**

Caution:

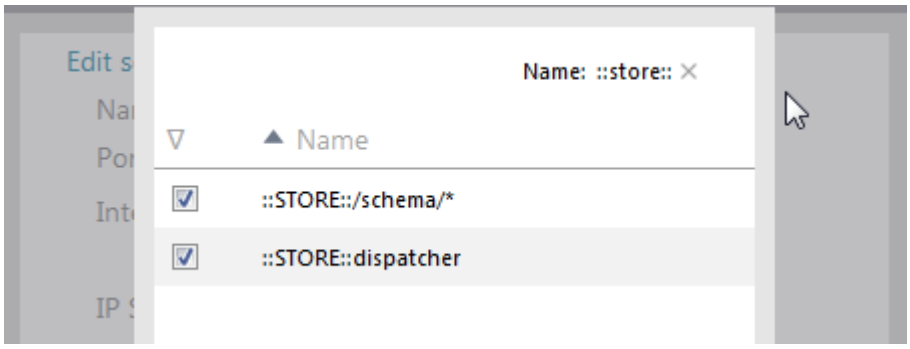
- The WSDL to T4x SOAP server requires that a JDK (Java Development Kit Standard Edition) is set as the Java VM in the Admin UI for either generally for "Configuration/Java" or specifically for the "SOAP server" option in "Configuration/Java/Advanced". The SOAP server does not work with a JRE (Java Runtime Environment).

- The WSDL to T4x server also requires Apache CXF 3.1.11 libraries to be available in `%TP_T4XINSTL%\lib\apache-cxf\lib`. These are not part of the T4x GS package and have to be copied on site! See the **Teamcenter Gateway - Installation Guide** for more information on this.

- A WSDL may only contain a single <wsdl:service> tag to be imported correctly. If your WSDL contains several services, you must split the WSDL into separate WSDL files, each containing exactly one service, before importing them.

- An imported WSDL may contain *schema references*, which are URLs. T4x will download these referenced schema files, import them and adapt the links in the generated WSDL. If your WSDL is on a local file system and the schema URLs in the WSDL are relative, you must either download the schema files and place them in the according (sub-)directories of your WSDL or you must edit the schema references in the WSDL to point to valid absolute URLs. If you want to download the WSDL directly from the web service or any other URL, then relative and absolute URLs should work. In this case you may pass the URL of the WSDL to the import tool instead of a file name.

  In order to successfully download such a WSDL with its referenced schema files, the functions `::STORE::dispatcher` and `::STORE::/schema/*` must be allowed and assigned to the same server instance(s) the service runs on: In the GS Admin UI, select "Configuration/Server Instances", then select the server instance of your service and click the edit (pencil) link, then click "Edit Function Binding". Make sure both entries are enabled (checked):



Example walkthrough

For this example we will use the WSDL provided in the T4EA `var/template/t4eademo` folder. We will import the WSDL to T4EA and then implement one of the operations in TCL and verify it in SoapUI. The following examples assume a Windows platform.

Prerequisites:

- T4EA GS is installed and configured.
- In GS Admin UI you have explicitly configured a JAVA binary from a JDK (JRE or implicit “default” configuration is not sufficient!) in the Configuration/Java settings.
- Apache CXF libraries are available in `%TP_T4XINSTL%\lib\apache-cxf\lib`.
- (optionally) SoapUI version 5.1.x. The installation path of SoapUI is configured in `etc/tpds`. This is only necessary if you want to create default responses for your service operations.

Steps to execute:

1. Set path to SoapUI (optional):
   add the following line you your `%TP_T4XINSTL%\etc\t4xcust.bat` to be able to generate default XML responses:

   ```
   set TP_SOAPUIPATH=C:/Progra~1/SmartBear/SoapUI-5.1.3
   ```

2. Import WSDL and generate TCL stubs:
   in a command window, change to the T4x installation directory (`%TP_T4XINSTL%`) and call the following command with adapted parameters:

   ```
   bin64\wsdl2t4x.exe \
     -wsdl %TP_T4XINSTL%/var/template/t4eademo/T4EADemoERP.wsdl \
     -outns http://www.siemens.com/plm/ActiveIntegration \
     -out var/mmap/t4ea_mapping_config/myservices.sd \
     -genxml on \
     -target server
   ```

3. Review and implement services in the stubs:
   The generated file contains an internal, unreadable representation of the imported WSDL and stubs for all operations. The above example already generates default XML responses for all operations in myservices.sd. You can review these operations and add your real service implementation here, returning valid response XML. As an alternative you can also call the wsdl2t4x with the option –`mapns` and direct the generated stub calls directly to procedures with identical names and signatures in a different namespace. Wsdl2t4x will not touch this namespace – but will preserve and call your implementation. As long as the list of operations stays identical, this approach allows adapting to a new version of the WSDL without having to adapt implementations for unchanged operations.

4. Deploy the changed mapping using the hot deploy script or the manual mmap and restart sequence.

5. Verify the service availability:

In the GS Admin UI, change to “System Information/Proxy List” and choose “SOAP2” as a filter in the “Type” column. You should see a line containing the service “T4EADemoERPService” and can download it’s adapted WSDL, containing the correct endpoint, in the “WSDL” column.
If needed, assign the service to a specific instance in “Configuration/Server Instances”, e.g. to assign a specific port to a service or to enable/disable basic HTTP authentication.
Call the service from e.g. SoapUI by creating a new SOAP project, importing the WSDL (either from a downloaded file or via URL, default **http://<T4x-host>:11301/soap2/t4eademoerpservice?wsdl**), and issuing a sample request.

6. On changes of the WSDL you can re-execute the `wsdl2t4x` utility and hot deploy the changes. When deleting WSDLs (and their service implementations), you must also remove the corresponding generated mapping files (.sd) from the mapping directory and from `t4ea_mapping_config.sd` and all files in `%TP_T4XINSTL%/var/store/wsdl`.

Considerations on WS-*

WS-* is a group of standards implemented as extensions of the SOAP standard. You can implement a correct SOAP server without using any WS-* standard. So if none of the WS-* standards is required for your integration, you can safely skip the rest of this chapter.

Some WS-* configurations can be handled by pure XML configuration changes in the CXF configuration file. This file must be placed directly in the T4x installation directory.

But since T4x uses dynamic configuration and builds the CXF-internal “bus” programmatically, all settings related to beans, interceptor selection and bus or distinct services cannot be used. All these settings usually require some Java custom classes and therefore are not supported OOTB. Please ask Siemens PLM Software if you need support on this.

T4x specific WS-* Configuration File

T4x provides a specific XML configuration file for WS-* related settings. The file must be placed in var/conf of the T4x installation directory and must have the name t4x_cxfproperties.xml. Generally speaking all entries concerning WS-Security are prefixed with *servicename*.wss4j and a direction in for incoming requests and out for outgoing responses. Since the T4x SOAP module only uses the tags behind this prefix and the specified values to set them as properties for the CXF WS-Security interceptors, many combinations of values as e. g. defined in the Java class `org.apache.wss4j.dom.handler.WSHandlerConstants` may make sense. We will just give some examples for WS-Security configuration with user name and token where

servicename is the fully qualified name of the service as specified in the WSDL, i.e. e. g. if the targetNamespace is given as http://www.example.com/Product/Version3 and the `<wsdl:service name="CreateProduct">`, then the fully qualified service name would be http://www.example.com/Product/Version3/CreateProduct.

The key `passwordType` allows values `PasswordText` and `PasswordDigest`. `userID` is a user name and `handlerClass` is the name of a Java class implementing the interface `javax.security.auth.callback.CallbackHandler`. `passwordType PasswordText` means

that the password contained in the security header is clear text, whereas with PasswordDigest it is hashed and digested with timestamp and nonce.

All entries in the file are optional and the file does not have to exist for the T4x SOAP server to work.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
  <comment>CXF properties for T4x services</comment>
    <entry key="servicename.wss4j.in.passwordType">passwordType</entry>
    <entry key="servicename.wss4j.in.action">UsernameToken</entry>
    <entry key="servicename.wss4j.out.user">userID</entry>
    <entry key="servicename.wss4j.out.action">UsernameToken</entry>
    <entry key="servicename.wss4j.out.passwordType">passwordType</entry>
    <entry
key="servicename.wss4j.in.passwordCallbackClass">handlerClass</entry>
</properties>
```

Note:

- Specifying a user name in the configuration makes only sense for outgoing responses since with incoming requests, the user name is contained in the security header.
- If you really need to configure a handler class, this class must be in the classpath at runtime. Please contact Siemens PLM Software if you need a handler class.
- After changing the configuration file you must restart the T4x Gateway Server for the changes to get effective.

WS-Security UserNameToken

With this standard the user credentials are passed in the SOAP header of the request. T4x provides an implementation that allows verifying the credentials against the T4x internal user database.

Note:

By default T4x instances require authentication on the HTTP layer already. To use WS-Security features it is useful to create a separate server instance in the Admin UI of the GS in “Configuration/Server Instances”, assign all services to which you need to apply WS-Security to the new instance and assign “Basic Authentication: Off” to the instance. Otherwise the service clients need to authenticate twice.

To enable this feature, you must enter the service name in a special notation in a T4x-specific configuration file `%TP_T4XINSTL%\var\conf\t4x_cxfproperties.xml` (continued lines ending with "↩" must be merged to be valid!):

```
<entry key="http://t4xsoapcxf.splm.siemens.com/t4xsoaptest/
T4xTestService11↩
  .wss4j.in.passwordType">PasswordText</entry>
<entry key="http://t4xsoapcxf.splm.siemens.com/t4xsoaptest/
T4xTestService11↩
.wss4j.in.action">UsernameToken</entry>
```

In the example the original namespace was http://t4xsoapcxf.splm.siemens.com/t4xsoaptest/ and the servicename was “T4xTestService11”. The added `wss4j.in.passwordType` and “wss4j.in.action” denotes the CXF configuration keys to apply WS-Security to the service.

When you send a WS-Security augmented request to that service, the T4x server will check the delivered password against the T4x user database. The password must be delivered in an MD5 hashed, HEX16 encoded manner. The following is an example of a valid request to the example service, which will authenticated by the server (continued lines ending with "↩" must be merged to be valid!):

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:t4x="http://t4xsoapcxf.splm.siemens.com/t4xsoaptest/">
   <soapenv:Header>
      <wsse:Security
  xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-secext-1.0.xsd"
  xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd">
         <wsse:UsernameToken wsu:Id="UsernameToken-39FB2CB3CFA9222F15144802233652720">
            <wsse:Username>t4adm</wsse:Username>
            <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401↩
  -wss-username-token-profile-1.0#PasswordText">E8636EA013E682FAF61F56CE1CB1AB5C</
wsse:Password>
         </wsse:UsernameToken>
      </wsse:Security>
   </soapenv:Header>
   <soapenv:Body>
      <t4x:soapTest>
         <Input>?</Input>
      </t4x:soapTest>
   </soapenv:Body>
</soapenv:Envelope>
```

Note that this request by itself is not safe against replay attacks. You should always apply transport security (SSL, HTTPS) to the communication to make it safer. Another possibility is to use the password type “PasswordDigest” in the configuration file (and on the client). In this case the request must additionally contain nonce and timestamp. The T4x server will then verify the digest automatically and allow access only if the password matches and nonce and timestamp can be verified – so T4x detects and refuses a replay of that request.

WS-Security Policy

With WS-SecurityPolicy you can force SOAP clients to use a certain security protocol by specifying it in the WSDL. T4x supports WS-SecurityPolicy versions 1.1 and later. WSDL files using WS-SecurityPolicy must include the following namespace definitions (continued lines ending with "↩" must be merged to be valid!):

```
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:wsu=
"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"
```

WS-SecurityPolicy 1.1 is specified in **http://specs.xmlsoap.org/ws/2005/07/securitypolicy/ws-securitypolicy.pdf** and the namespace for WS-SecurityPolicy 1.1 is: `xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"`

WS-SecurityPolicy 1.2 is specified in **http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/ws-securitypolicy-1.2-spec-os.html** and the namespace for WS-SecurityPolicy 1.2 is:`xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"`

Security policies can be defined in one place in the WSDL. References to certain policies can be used for each incoming and outgoing message. Here is an examples snippet from a WSDL:

```
....
xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
xmlns:sp="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/
oasis-200401-wss-wssecurity-utility-1.0.xsd"
....
<wsdl:operation name="doHello1">
   <soap:operation soapAction="" style="document"/>
   <wsdl:input name="doHello1">
     <wsp:PolicyReference URI="#HelloBindingPolicy" />
     <soap:body use="literal"/>
   </wsdl:input>
   <wsdl:output name="doHello1Response">
     <wsp:PolicyReference URI="#HelloBindingPolicy" />
     <soap:body use="literal"/>
   </wsdl:output>
</wsdl:operation>
....
<wsp:Policy wsu:Id="HelloBindingPolicy">
    <sp:SupportingTokens>
        <wsp:Policy>
          <sp:UsernameToken
  sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/
IncludeToken/AlwaysToRecipient">
             <wsp:Policy>
               <sp:PasswordHash/>
             </wsp:Policy>
           </sp:UsernameToken>
        </wsp:Policy>
    </sp:SupportingTokens>
</wsp:Policy>
```

This policy enforces the client to use a hashed password, implying that a timestamp and a nonce have to be contained, too.

> Note:
>
> Since the security information is forced by the policy in the WSDL, there is no need to additionally specify security settings in the configuration file t4x_cxfproperties.xml and you should not do so for this service!

8.3 T4EA Consumes SOAP services

8.3.1 Introduction

To consume SOAP services, T4x offers two approaches:

- The Java adapter that uses the Apache AXIS2 framework. The procedure `::T4X::SOA::performGenericWebServiceCall` takes an XML payload and sends it to the server via a separate Java pipe process attached to the current worker. This approach misses some features (like MTOM, WS-*, etc.) and has a relatively high memory load. It is deprecated with T4x 11.2.3, but will be maintained until further notice.

- The CXF based Java adapter which runs in the same single JVM as the CXF-based T4x SOAP server. It uses the same utility `wsdl2t4x` to import WSDLs as the SOAP server. New implementations should be made using this approach.

Both implementations can use HTTP and HTTPS transports.

Caution:

- Using the AXIS2 Java adapter: Surround the user name and the password by `#!!` or use the TCL function `::T4X::SOA::encryptString4PipeProtocol`. This ensures that the credentials do not appear in the log files, but does not really hide them in the source code.

- Using the CXF-based Java adapter: Do not surround user name and the password by `#!!` and do not use the TCL function `::T4X::SOA::encryptString4PipeProtocol`. The adapter handles credentials correctly without the need to handle them separately, so they will not show up in the log files.

- You can encrypt any TCL command (e.g. setting the credentials variables) using the Teamcenter Gateway script **T4S EvalCrypt** and execute the encrypted command with `::SYSBase::evalCryptCode`. In this case the mapping source code no longer contains the clear text credentials.

- SSL requires that the client trusts the server. Usually this is achieved by a list of trusted root certificates in the client. Any SSL certificate with a certification path derived from one of the trusted certificates is trusted by client. If such a path cannot be established, the following exception appears in the logfile:

  ```
  javax.net.ssl.SSLHandshakeException:
    sun.security.validator.ValidatorException:
    PKIX path building failed:
    sun.security.provider.certpath.SunCertPathBuilderException:
    unable to find valid certification path to requested target.
  ```

  This exception can result from a self-signed certificate, a certificate signed from an untrusted or unknown certification provider or a *man-in-the-middle-attacker*!

- To avoid this exception you can import and trust the certificate by one of the following methods:

  - Use Java keytool, for details see **http://docs.oracle.com/javase/6/docs/technotes/tools/solaris/keytool.html**.

  - Use the T4EA script “T4EA SSL certificate import” (`t4ea_import_certificate.tcl`), which wraps the Java keytool in a simpler user interface, allowing to enter the https server's URL.

  However in each case you *have to make sure* that the server’s certificate is really the one you want to trust in! Both methods do not verify or validate the certificate, they just import it and persistently manifest your trust!

- More configuration for CXF and its keystore can be configured in `cxf.xml` as described in **http://cxf.apache.org/docs/client-http-transport-including-ssl-support.html# \ ClientHTTPTransport(includingSSLsupport)-ConfiguringSSLSupport**

  For example: To get a client to work with a server certificate containing "Common Name" (CN) attributes, add this to the configuration:

  ```
  <http:conduit name=
  "{http://www.siemens.com/plm/ActiveIntegration/soap2/
  DOCMathTest}DOCMathTestSoap.http-conduit"
  ```

8.3.2 Consuming SOAP via the AXIS2 Java Adapter

AXIS2 as an open source Java library provides a wide range of support for different web service flavors and can be configured and extended. Instead of calling the adapter directly you should use the T4EA TCL function `::T4X::SOA::performGenericWebServiceCall` (see the **T4EA API Reference** for details). This function starts a synchronous call. You can transfer the message asynchronously with this procedure instead `::T4X::SOA::performGenericWebServiceCallFireAndForget`.

This adapter cannot automatically generate a TCL proxy from a WSDL. Creating the XML payload, choosing the communication protocols and endpoints has to be implemented manually.

Generally the adapter uses HTTP simple authentication with user and password encoded in a SHA5 digest. Using a secured transport layer like SSL in a HTTPS communication this provides a simple yet secure authentication. If you do not want to send any authentication, leave both user and password empty.

The adapter has a built-in timeout of two minutes. If T4x does not receive an answer from the server within this timeframe, the call to `::T4X::SOA::performGenericWebServiceCall` will return an error. The default timeout can be changed by setting the environment variable `TP_AXIS_WS_TIMEOUT` to the number of seconds of the desired timeout. Note that in case of a timeout it is impossible for T4x to determine the reason: It can be any of the following: wrong server names, slow network, network overload, network failure, firewalls, server faults, slow servers, etc. To investigate on the reason, try other servers, change the payload or watch the server log file.

8.3.3 Consuming SOAP via the CXF-based Java adapter

In contrast to the AXIS2-based adapter, which potentially starts a separate Java process for each worker, this adapter runs in a single, shared JVM, significantly reducing overall memory consumption. All restrictions and recommendations mentioned in the chapter “WSDL to T4x” apply correspondingly.

The approach makes use of two interfaces: the command line utility `wsdl2t4x`, which includes the WSDL into the mapping and generates sample stub code and the TCL procedure `::T4X::SOAP::CLIENT::callSoapOperation`, which takes an XML SOAP envelope, checks and transforms it, sends it to a configurable endpoint and returns the response and HTTP information.

Example walkthrough

In the rest of this chapter this process will be explained.

For prerequisites consult section "Example Walkthrough" in chapter **WSDL to T4EA**

Read WSDL with Wsdl2t4x

For generating client stub examples, call `wsdl2t4x` with the arguments “`-target client`” or “`-target both`”.

```
%TP_T4XINSTL%\bin64\wsdl2t4x.exe -wsdl var/template/t4eademo/
T4EADemoERP.wsdl \
  -target client -genxml on -out stubs.sd
```

The generated TCL file will contain the WSDL in encoded form (shortened for readability) and an example (commented out) for setting an optional timeout for the service:

```
# store the WSDL into the local shared memory
::T4X::SOAP::CLIENT::deployShmemVariable WSDLS T4EADEMOERPSERVICE \
    eF7tXGtv2zgW/T7A/AeuMcC2…
# Response timeout in seconds; default is 60s. To set a timeout,
# remove comment and set the needed value.
# ::T4X::SOAP::CLIENT::deployShmemVariable TIMEOUT T4EADEMOERPSERVICE 60
```

Calling the service operation

The generated code will also contain a runnable stub example similar to the following:

```
proc call_Echo {} {
  # sample SOAP request
  set SOAP_request {
      <soapenv:Envelope
             xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
             xmlns:t4e="http://www.tesis.de/plmware/T4EADemoERP/">
         <soapenv:Header/>
         <soapenv:Body>
            <t4e:Echo>
               <in>?</in>
            </t4e:Echo>
         </soapenv:Body>
      </soapenv:Envelope>
  }
      # >>>>> START HERE GENERATING THE REQUEST >>>>>
  xdom parse $SOAP_request reqdom
  set reqroot [$reqdom documentElement]
  set reqbody [$reqroot selectNodes /soapenv:Envelope/soapenv:Body]
      # <<<<< END HERE GENERATING THE REQUEST <<<<<
  set response [::T4X::SOAP::CLIENT::callSoapOperation \
       -service T4EADemoERPService \
       -envelope [$reqroot asXML] ]
  set responseXml [lindex $response 0]
  set httpHeader  [lindex $response 1]
  set httpCode    [lindex $response 2]
  set contentType [lindex $response 3]
      # >>>>> RESPONSE PROCESSING >>>>>
  # SOAP response template; for documentation only.
  # Please note that the real response may use different namespaces.
  set SOAP_response_body {
```

```
        <soapenv:Envelope
               xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:t4e="http://www.tesis.de/plmware/T4EADemoERP/">
            <soapenv:Header/>
            <soapenv:Body>
                 <t4e:EchoResponse>
                    <Result>?</Result>
                    <!--Optional:-->
                    <Message>?</Message>
                    <!--Optional:-->
                    <Code>?</Code>
                 </t4e:EchoResponse>
            </soapenv:Body>
        </soapenv:Envelope>
    }
    xdom parse $responseXml resdom
    set resroot [$resdom documentElement]
    set resbody [$resroot selectNodes /soapenv:Envelope/soapenv:Body]
        # >>>>> START RESPONSE PROCESSING HERE >>>>>
        # <<<<< END RESPONSE PROCESSING HERE <<<<<
        return [$resroot asXML]
}
```

Calling the procedure `call_Echo` will call the “Echo” operation of the service with a default request generated by SoapUI based on the imported WSDL and return the body of the response.

Note:

- Due to CXFs error handling, callSoapOperation will return HTTP code 500 also for many client side problems (like non-existent WSDL, XML errors, connection problems, etc.). This is why it does not return a separate status code.
- In the CXF-based SOAP *server*, the *generated procedure* contains a `createServer` call and so *gets called by the T4x server*. The generated code or a modified version of it *must* be included in the mapping for the SOAP server to work.
- In *contrast* for the CXF-based SOAP *client* the generated stub does not contain a `createServer` call and *it is just an example* to show how to use callSoapOperation and how to handle the envelopes and there is actually *no use* in embedding the generated procedure in your mapping.
- However you must make sure that the *WSDL part* of the generated code gets *included in your mapping*, and that the mapping is deployed, otherwise calling the service will result in an error.
- The response processing in the client stub is also just an example. Of course a real implementation must use the response from the called service instead.

Caution:

- If you want to redeploy a changed WSDL to a T4x instance that has already called the service with the old WSDL in place, you must:
  - Redeploy the mapping (with mmap or the hot deploy script)
  - Remove all files in `%TP_T4XINSTL%\var\store\wsdl`
  - Restart the CXF based SOAP adapter by restarting the T4x GS
- Certain errors in the XML lead to problems when CXF converts the XML from TCL into its internal object representation. Not all of these errors cause a real error message and abort the operation. Instead it may happen that parts of the payload miss in the sent request. So please check the code generating the payload and the payload you are sending very carefully! You can use tools like SoapUI and Fiddler to verify your payload.

Basic HTTP Authentication Policy

If your service requires basic HTTP authentication but no corresponding policy is contained in the WSDL file (and this situation is quite common), no further configuration has to be done. Just pass the username and password to `::T4X::SOAP::CLIENT::callSoapOperation`.

However your WSDL might contain a policy requesting basic HTTP authetication like in this example:

```
<wsp:Policy wsu:Id="BasicHttpBinding_BillsofMaterialsService_policy">
  <wsp:ExactlyOne>
    <wsp:All>
      <http:BasicAuthentication xmlns:http=
      "http://schemas.microsoft.com/ws/06/2004/policy/http"/>
    </wsp:All>
  </wsp:ExactlyOne>
</wsp:Policy>
```

In this case you must adapt the cxf.xml of your CXF installation (cxf.xml must reside e.g. in <T4x_GS_ROOT>):

```
<bean class="com.siemens.splm.t4x.soapcxf.T4xAssertionBuilder">
  <constructor-arg>
    <list>
      <bean class="javax.xml.namespace.QName">
        <constructor-arg value=
          "http://schemas.microsoft.com/ws/06/2004/policy/http"/>
        <constructor-arg value="BasicAuthentication"/>
      </bean>
    </list>
```

```
    </constructor-arg>
  </bean>
```

WS-Security UserNameToken and WS-SecurityPolicy

For details on UsernameToken, please see also the section "WS-Security UserNameToken" in chapter **WSDL to T4EA**.

The T4x SOAP client detects if a service enforces UserNameToken authentication by Policy. If username and password is passed to callSoapOperation, CXF will automatically add headers accordingly. In this case there is no need for additional mapping or configuration changes.

Note:

- `callSoapOperation` will pass the username and password as basic HTTP and as WS-Security addition in the SOAP header, if UserNameToken authentication is in place. There is no way to directly suppress either of these.
- If UserNameToken is specified in the T4EA specific WS-* configuration file (see below), then the adapter suppresses basic authentication.

If your server's WSDL does not contain the policy statement, but you nevertheless want to authenticate using UserNameToken, you can specify that in the T4x specific SOAP configuration file (see also section "T4x specific WS-* Configuration File" in chapter **WSDL to T4EA**). To enable authentication in the request, set the "out" properties in `%TP_T4XINSTL%\var\conf\t4x_cxfproperties.xml`:

```
<properties>
  <entry key=
"http://www.tesis.de/plmware/T4EADemoERP/T4EADemoERPService.wss4j.out.passwordType"
    >PasswordText</entry>
  <entry key=
"http://www.tesis.de/plmware/T4EADemoERP/T4EADemoERPService.wss4j.out.action"
    >UsernameToken</entry>
</properties>
```

Kerberos HTTP Authentication

With Kerberos, the client needs to add a ticket to the HTTP header. Client and server must connect to the same Kerberos authority to retrieve and validate the ticket. This chapter describes only HTTP Kerberos authentication, WSS-based authentication with Kerberos has not been tested.

For the T4x client to use Kerberos you need to configure the Java Development Kit used for the SOAP client *and* in the CXF configuration. For details about Kerberos and CXF, see **http://cxf.apache.org/docs/client-http-transport-including-ssl-support.html**

In file *<JDK>/lib/security/java.security* configure the login configuration file (in this case *${user.home}/.java.login.config*):

```
#
# Default login configuration file
#
login.config.url.1=file:${user.home}/.java.login.config
```

In the specified login module file (i. e. *<user home>/.java.login.config*) specify the login module:

```
JaasSample {
  com.sun.security.auth.module.Krb5LoginModule required storeKey=true \
  useTicketCache=true useSubjectCredsOnly=false useKeyTab=false
debug=true;
};
```

API description for the available java classes and their configuration options can be found in **https://docs.oracle.com/javase/7/docs/jre/api/security/jaas/spec/index.html** in the package com.sun.security.auth.module.

The option `debug=true` can be omitted or set to `false` and should not be used on a productive system, but it may be useful for tests and it causes some extra logging to the T4x system log file `ecmdgjvm.log`.

An addition to the JDK configuration, CXF also needs to be configured. The name `"JaasSample"` above can be freely chosen, but must be identical to the name referenced in `cxf.xml` as follows:

```
<http:conduit name="[fully qualified service name].http-conduit">
    <http:authorization>
        <sec:AuthorizationType>Negotiate</sec:AuthorizationType>
        <sec:Authorization>JaasSample</sec:Authorization>
    </http:authorization>
</http:conduit>
```

with the "fully qualified service name" consisting of the target namespace specified in the WSDL between curly braces `{}` and the service port name (not the service name!) as specified in the WSDL. Here is an example:

```
<http:conduit name="{http://tempuri.org/}BasicHttpBinding_IDemoService.http-conduit">
```

The HTTP header by the T4x SOAP client will contain something like:

```
Authorization=[Negotiate YIIV1gYGKwYBBQUCoIIVyj...
```

Caution:

- Kerberos authentication requires the target URL to be in the format corresponding to the Kerberos server's configuration. So host names like `localhost` or `127.0.0.1` may not work. In such cases you may receive the error message:

```
An error occurred in service invocation:
org.ietf.jgss.GSSException :: No valid credentials provided
(Mechanism level: No valid credentials provided
(Mechanism level: Server not found in Kerberos database (7)))
```

- The User-ID specified with `callSoapOperation` must not have a domain prefix like e.g. `test\tstusr1`, but has to be specified similar to a mail address with the domain name separated by `'@'`, e. g. `tstusr1@TEST.LOCAL`. If server and client are in the same domain a user name without domain name can be used.

- With Kerberos authentication, Java uses an environment variable `USERDNSDOMAIN` containing the domain name to locate the domain server. The variable normally exists in the environment for a domain user, but access is also possible from a local user on a machine that belongs to a different domain if the variable is set. Without this variable, the client call fails with `"krbexception: cannot locate default realm"`.

- Microsoft based web services often specify a security policy in the WSDL like e. g. for a Kerberos or NTLM secured service:

```
<wsp:Policy wsu:Id="BasicHttpBinding_IDemoService_policy">
    <wsp:ExactlyOne>
        <wsp:All>
            <http:NegotiateAuthentication xmlns:http=
  "http://schemas.microsoft.com/ws/06/2004/policy/http"/>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

  Note that with the namespace `xmlns:http="http://schemas.microsoft.com/ws/06/2004/policy/http"` this is not a standard WS-Security policy (namespace `xmlns:sp="http://schemas.xmlsoap.org/ws/2005/07/securitypolicy"`) and hence is not supported by CXF. Instead this informs the client how to authenticate. Without any further precautions this will cause a runtime error message: `org.apache.cxf.ws.policy.PolicyException: None of the policy alternatives can be satisfied.`

  To avoid this error either remove the policy (and its references) from the WSDL before generating the client stub in T4x or add the following entry in *cxf.xml*

```
<bean class="com.siemens.splm.t4x.soapcxf.T4xAssertionBuilder">
    <constructor-arg>
        <list>
            <bean class="javax.xml.namespace.QName">
                <constructor-arg value=
  "http://schemas.microsoft.com/ws/06/2004/policy/http"/>
                <constructor-arg value="NegotiateAuthentication"/>
```

NTLM Authentication

The CXF-based T4x SOAP client on Windows supports NTLM authentication. One possibility to do so is the above shown Kerberos configuration. CXF "negotiates" with the server on the authentication method and uses the logged in Windows user to transfer the necessary NTLM artifacts. For more details on NTLM or "Windows authentication", see: **https://msdn.microsoft.com/en-us/library/cc236621**

8.4 TLS configuration for SOAP services

8.4.1 Configure SOAP Service for 1-way SSL

The context of this topic is providing SOAP services in PL4x within an instance with encryption, i. e. https access, and calling a SOAP service that uses 1-way SSL. 1-way SSL is a concept that is explained e.g in **http://www.ossmentor.com/2015/03/one-way-and-two-way-ssl-and-tls.html**

1. Configure your server instance: Set encryption on and specify server certificate

2. Configure the corresponding communication channel: Set transport mode Encrypted socket and specify CA certificate

3. Save configuration and restart GS

4. Store CA certificate in your JVM like this:

```
cd C:\Program Files\Java\<my Java version>\jre\bin>
keytool -keystore ..\lib\security\cacerts -importcert -alias
<myalias> -file C:\Users\myuser\cert\<mycertificate_name_root>.pem
Enter keystore password: <enter changeit>
Certificate was added to keystore
keytool -keystore ..\lib\security\cacerts -importcert -alias
<myalias> -file C:\Users\myuser\cert\<mycertificate_name_child>.pem
Enter keystore password: <enter changeit>
Certificate was added to keystore
```

If a service endpoint URL with "localhost" is supposed to work, add the following entries to cxf.xml:

```
<beans ...
   xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
   ...
   xsi:schemaLocation="...
      http://cxf.apache.org/transports/http/configuration http://
cxf.apache.org/schemas/configuration/http-conf.xsd
   ...">

   <http-conf:conduit name="{http://www.siemens.com/
T4x/}T4xGetItemBothSOAP.http-conduit">
      <!-- deactivate HTTPS url hostname verification (localhost,
```

```
etc)     -->
      <!-- WARNING ! disableCNcheck=true should NOT be used in
production -->
      <http-conf:tlsClientParameters disableCNCheck="true" />
   </http-conf:conduit>

   ...
</beans>
```

Note that http://www.siemens.com/T4x/ is the targetNamespace from the WSDL and T4xGetItemBothSOAP is the WSDL port name (as found in the wsdl:service section of the WSDL).

If the server certificate chain is not placed in the default truststore of the JVM (jssecacerts, if it exists, cacerts otherwise, see **https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html#Customization**), a truststore can be configured in cxf.xml:

```
<beans ...
   xmlns:http-conf="http://cxf.apache.org/transports/http/configuration"
   xmlns:sec="http://cxf.apache.org/configuration/security"
   ...
   xsi:schemaLocation="...
      http://cxf.apache.org/transports/http/configuration http://
cxf.apache.org/schemas/configuration/http-conf.xsd
      http://cxf.apache.org/configuration/security http://cxf.apache.org/
schemas/configuration/security.xsd
   ...">

    <http-conf:conduit name="{http://www.siemens.com/
T4x/}T4xGetItemBothSOAP.http-conduit">
      <http-conf:tlsClientParameters>

        <sec:trustManagers>
          <sec:keyStore type="JKS" password="passwd"
                        file="my_truststore.jks"/>
        </sec:trustManagers>

      </http-conf:tlsClientParameters>
    </http-conf:conduit>

   ...
</beans>
```

8.4.2 Configure SOAP for 2-way SSL

The context of this topic is providing SOAP services in PL4x within an instance with encryption, i. e. https access, and calling a SOAP service that uses 2-way SSL. 2-way SSL is a concept that is explained e.g in **http://www.ossmentor.com/2015/03/one-way-and-two-way-ssl-and-tls.html**

In order to configure your SOAP client for 2-way SSL calls you first need to import the certificates into your Java VM keystore. The following example shows how to do that:

```
keytool -importkeystore -deststorepass changeit -destkeypass <mypassword
(default:changeit)> -destkeystore my-keystore.jks -srckeystore C:\Users
\myuser\cert\<mycertificate_name>.p12 -srcstoretype PKCS12 -srcstorepass
<mycertificate_password>
Entry for alias <mycertificate_name> successfully imported.
Import command completed:  1 entries successfully imported, 0 entries
failed or cancelled
```

Once the certificate is imported, you need to configure the client in the cxf.xml like in the following example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:cxf="http://cxf.apache.org/core"
      xmlns:jaxws="http://cxf.apache.org/jaxws"
      xmlns:http-conf="http://cxf.apache.org/transports/http/
configuration"
      xmlns:sec="http://cxf.apache.org/configuration/security"
      xsi:schemaLocation="
            http://cxf.apache.org/core http://cxf.apache.org/schemas/
core.xsd
            http://www.springframework.org/schema/beans http://
www.springframework.org/schema/beans/spring-beans-2.0.xsd
            http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/
jaxws.xsd
            http://cxf.apache.org/transports/http/configuration http://
cxf.apache.org/schemas/configuration/http-conf.xsd
            http://cxf.apache.org/configuration/security http://
cxf.apache.org/schemas/configuration/security.xsd">

    <http-conf:conduit name="{http://www.siemens.com/
T4x/}T4xGetItemBothSOAP.http-conduit">

        <http-conf:tlsClientParameters>

          <sec:keyManagers keyPassword="keypass">
            <sec:keyStore type="JKS" password="storepass"
                          file="keystore"/>
          </sec:keyManagers>

        </http-conf:tlsClientParameters>
    </http-conf:conduit>

</beans>
```

Note that `http://www.siemens.com/T4x/` is the targetNamespace from the WSDL and `T4xGetItemBothSOAP` is the port name.

"keypass" and "storepass" must, of course, be the respective passwords for the keystore and with file=... a full path name can be specified to make sure that the file is found.

More details of what can be configured with this file must be taken from the Apache CXF documentation **http://cxf.apache.org/docs/index.html**

8.5 MTOM Support

MTOM is a standard for transferring potentially large files over SOAP connections. The files may have different types and may contain binary content. Instead of encoding the file contents, e.g. in base64 and including them in an XML tag, the transferred XML "on the wire" is embedded in a MIME message and contains just an "identifier" and client and server stream the contents in parts of the MIME encoded message. This keeps the XML compact and avoids having to load the file content into main memory on either client or server, thus preventing the hosts to run out of memory. Currently there is no known limitation on the file sizes as long there is enough free storage on the file system and the network is fast enough to finish the transfer before any timeouts occur.

The CXF based SOAP server and client of T4x support MTOM according to parts of the WS-I basic profile 1.2 and 2.0. The older and less specific standard SwA (SOAP with Attachments) is not supported.

For both incoming and outgoing MTOM files, the interface between TCL (your mapping code) and Java (CXF adapter) is a standardized additional tag, pointing to the path of the file. Here is an example payload:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
envelope/"
           xmlns:t4e="http://www.tesis.de/plmware/T4EADemoERP/">
   <soapenv:Header/>
   <soapenv:Body>
      <t4e:DocumentUpdate>
         <ID>1234</ID>
         <RevisionID>A</RevisionID>
         <Documents>
            <Document>
               <File>my_example.txt</File>
               <Type>Text</Type>
               <FileContent>
                 <xop:Include href=file:///C:/Temp/my_example.txt
                   xmlns:xop="http://www.w3.org/2004/08/xop/include"/>
               </FileContent>
            </Document>
         </Documents>
      </t4e:DocumentUpdate>
```

```
    </soapenv:Body>
  </soapenv:Envelope>
```

> Note:
>
> The xop:Include tag must contain:
>
> - The absolute path of the local file with forward slashes as separators (“/”, also on Windows) in the href attribute
> - The “file:”protocol specifier, followed by 3 slashes (“///”).
> - The namespace declaration xmlns:xop="http://www.w3.org/2004/08/xop/include"

The T4x SOAP adapter will make sure this local file gets translated into a MIME encoded message containing the file content when sending this request or will make sure the file content from an incoming message will be stored in the named file before passing control to the TCL implementation.

For this to work correctly, the WSDL of that service must specify that the tag for the file content is of type xsd:base64 and has a list of allowed MIME types assigned. Here is an example excerpt:

```
<xsd:element name="FileContent" type="xsd:base64Binary"
  xmime:expectedContentTypes="application/octet-stream"
  xmlns:xmime="http://www.w3.org/2005/05/xmlmime"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
<xsd:element>
```

The xsd, xmime and xop namespaces can of course also be declared on the top level XML element, if more appropriate. One of the specified content types must match the actual MIME type of the transferred file, otherwise content might get lost without notice.

> Tip:
>
> - In order to transfer files with different or unknown file types and to avoid Java’s automatic interpretation of MIME types, we highly recommend to keep the contract “generic” and use MIME type “application/octet-stream” in the WSDL! This MIME type may hold text or binary files of any type.
> - Information regarding file type and file name can be transferred in additional attributes. The xop:Inlcude tag contains only the file content, not any meta data.

When passing files from TCL to MTOM, the files may reside anywhere on the file system. The `xop:Include` tag must contain the absolute path in URL notation.

When TCL receives files from the T4x SOAP implementation, the files will be in the directory indicated by the default value of the Java system property java.io.tmpdir (usually `%TMP%` for Windows, `/tmp` for Unix).

Caution:

- In the following SOAP MTOM situations it is the responsibility of the TCL code to care for deletion of the files:
  - Files passed to the request in a client implementation
  - Files received from a call in a client
  - Files received in the request of a server implementation.
- In a *SOAP server* implementation, if *files are passed from the TCL layer into the response* (e.g. a “download” service implementation):
  - The request must be in MIME encoded format (not plain XML), even if it does not contain any MTOM file. In SoapUI for example, you can achieve this by setting the request properties “Enable MTOM” and “Force MTOM”.
  - The T4x CXF SOAP server will automatically *delete the files* passed in the payload.
- *Reason for the deletion:* When the TCL implementation is finished, control is passed to the CXF server and no TCL code is called afterwards that could delete the files. So the server implementation takes care of the deletion.
- In most cases this should be the desired behavior as files are usually prepared solely for the purpose of being transferred and are no longer needed after the service call has finished.
- But if your *file must be kept* after the call has finished, you *must* pass a copy of the file via the `xop:Include` tag!

9. Messaging

9.1 Introduction

With messaging we mean the ability of T4x to communicate with “Message Oriented Middleware” (MOM). This pattern of distributed system design allows decoupling the information sender (producer of messages) from the message receiver. The only knowledge they share is the structure of the message and the destination (queue) to store messages in and read messages from. The knowledge of the direct communication paths from sender to receiver, their addresses or implementation is hidden from each other. Many messaging systems work with a central component collecting, storing and distributing messages, called “broker” or “provider”.

Messaging is by default asynchronous (fire-and-forget). Other message exchange patterns, like request-response have to be implemented in the application layer on top of messaging.

9.2 Java Message Service

One important standard in the messaging area is JMS (Java Message Service), version 1.1, a Java API supported by many messaging systems. T4EA can work with any JMS system, if the JMS 1.1 implementation Jars are available and accessible by T4x.

In JMS messages contain any kind of data; the JMS API requires no interpretation of the message body. The message body has to be interpreted by the application layer and can be anything from XML or pure text to serialized Java objects or even binary data.

JMS 1.1 has two major concepts of message exchange:

- point-to-point: Messages are stored in several queues identified by a name. Several producers may add messages to a queue. Once a receiver reads messages from a queue, they get removed from the queue. Each message gets processed by only one receiver, but many receivers may read from a single queue.

- publish-subscribe: Several producers write messages to a topic. A topic is like a named queue and topics may be organized in a hierarchy. Receivers may register for topics. If a message arrives for a topic, all registered receivers get notified and can read the message. So a single message gets delivered to none, one or many receivers without the sender having to care for the distribution.

To be able to configure details of the connections and the brokers, JMS proposes the use of JNDI (Java Naming and Directory Interface). JNDI allows to specify provider-specific settings in a neutral, pure text based format. The simplest provider of a JNDI service is a flat text file with an implementing Java class, LDAP being more advanced. JMS uses some standard keys to search in a JNDI provider for queues, servers, etc.

> Caution:
>
> T4x supports point-to-point messaging only.

9.3 Preparing the T4x GS installation

T4x uses the JMS API to implement messaging. This requires that the customer provides the implementation JAR of the messaging system or broker they want to use. For details on the installation process please see the **Teamcenter Gateway Installation Manual**.

9.4 Example code snippets

T4x does not expose the JMS API to the mapping. Instead you can use messaging by an abstract TCL API.

To use messaging, you must first configure a connection. T4x accepts the connection details as a TCL dictionary.

This example builds a connection dictionary for an IBM MQ connection:

```
set connection_mq1 [dict create t4x.connectionFactoryName "myFactory"]
dict set connection_mq1 t4x.queueName "Q1"
dict set connection_mq1 java.naming.provider.url "file://\\\\c:\\\
\JMSBindings"
dict set connection_mq1 java.naming.factory.initial \
  "com.sun.jndi.fscontext.RefFSContextFactory"
dict set connection_mq1 t4x.acknowledgeMode "AUTO_ACKNOWLEDGE"
```

The first line denotes a so-called “factory”, a Java object built dynamically in the JMS client and containing vendor-specific and configuration-specific parameters to build a connection. The next line denotes a JMS queue name, also called a “destination”. The factory and the destination must be created, named and provided by an IBM MQ administrator, e.g using the IBM MQ Explorer by creating corresponding “JMS administered objects”. This action creates a so-called “Bindings” file, which can be transported from the server (or a client) to any client requiring the configuration. For T4x this bindings file must be specified in the third line. Since the file name gets interpreted twice in TCL and Java, the Windows backslashes must be double-escaped, that is why 4 backslashes must be used in TCL source. The fourth line sets the Java class implementing the reader for the bindings file. This class is contained in the vendor-specific JMS implementation Jars. The last line is only necessary if this connection is used for reading (consuming) messages and specifies the acknowledgment mode to use. For details see the **T4EA API Reference**.

Here is another example configuring an in-memory broker for Apache ActiveMQ:

```
set connection_amq1 [dict create t4x.connectionFactoryName
"ConnectionFactory"]
dict set connection_amq1 t4x.queueName "queue/simple"
dict set connection_amq1 t4x.queueValue "test.queue.simple"
```

```
dict set connection_amq1 java.naming.factory.initial \
  "org.apache.activemq.jndi.ActiveMQInitialContextFactory"
dict set connection_amq1 java.naming.provider.url \
  "vm://localhost?broker.persistent=false"
dict set connection_amq1 t4x.acknowledgeMode "CLIENT_ACKNOWLEDGE"
```

As this broker will run in the same Java virtual machine as the JMS client, we need to specify not only the JMS queue name (destination), but also a name for the physical queue with the key "`queueValue`". Instead of a file path for the JNDI bindings, we provide a URI that specifies the parameters for the broker.

Next we open a connection, using the previously generated dictionary and give it a name (which can be used in the managed T4EA connection handing, if needed). The first example opens the IBM MQ connection for creating messages ("`PRODUCER`"), the second example opens an ActiveMQ connection for reading ("`CONSUMER`"):

```
set Status [::T4X::MESSAGING::startConnection "IBM_MQ_Producer1"
"PRODUCER" \
  $connection_mq1 ]
set Status [::T4X::MESSAGING::startConnection "ActiveMQ_Consumer1"
"CONSUMER" \
  $connection_amq1]
```

Next we send a "Hello World!" message to IBM MQ and receive and print a message from ActiveMQ:

```
set Status [::T4X::MESSAGING::sendStringMessage "IBM_MQ_Producer1" "" \
  "Hello World!"]
set Status [::T4X::MESSAGING::receiveMessage "ActiveMQ_Consumer1"
"NON_BLOCK"]
if {[lindex $Status 0] eq "OK"} {
  puts "MessageID:  [lindex $Status 1]"
  puts "Properties: [lindex $Status 2]"
  puts "Body:       [lindex $Status 3]"
  puts "Headers     [lindex $Status 4]"
  puts "isBinary:   [lindex $Status 5]"
  set Status [::T4X::MESSAGING::acknowledgeMessage "ActiveMQ_Consumer1" \
    [lindex $Status 1]]
}
```

Note that there are two different procedures for sending string and binary messages (sendStringMesssage and sendBinaryMessage), while there is only one procedure to receive a message, which returns a flag if the message is binary or not. The parameter "`NON_BLOCK`" in the `receiveMessage` call tells the messaging client to look for a message and return it, if a message is available. However if no message is available, the call will also return immediately but will not return a message instead of waiting until a message is available. This non-blocking behavior is recommended for mappings as a blocking call bears the risk to make the Teamcenter RAC UI unresponsive (in case of workflow transfers) or to cause timeouts (in case of T4x job execution or when embedded in other actions). The call returns a list containing the status, message ID, properties, body, headers and an indicator if the message must be interpreted as binary. If no message is available, the status will be `OK`,

but message ID, properties and body are empty. The call to `acknowledgeMessage` finally confirms to the broker that we successfully received the message. This call is only necessary when using "`CLIENT_ACKNOWLEDGE`" when opening the connection. If it misses in this case, the broker will try to re-deliver the message until some client finally acknowledges it.

Finally you can close the connections you used or close all connections with a single call:

```
set Status [::T4X::MESSAGING::closeConnection "IBM_MQ_Producer1"]
set Status [::T4X::MESSAGING::closeAll]
```

9.5 Messaging and Jobs

Messaging and T4x job management both address similar requirements: both decouple the producer from the consumer and both introduce non-synchronicity. They can be used alternatively or in conjunction. Using both in combination may have some advantages when receiving messages from a JMS broker: Currently T4x cannot listen continuously for arrival of messages and trigger some T4x transaction. Instead a T4x scheduler script can be used to receive all available messages in a loop. The minimum period for a scheduler script is one minute. Therefore the script should not execute the T4x transactions (e.g. import/export) associated with the messages directly, as this could cause the execution time of a single script invocation to exceed the scheduled period. Instead the script should just create T4x jobs for all available messages (read by a non-blocking `receiveMessage` call). These jobs can then be executed asynchronously with all scalability options of the T4x framework.

10. EA Connection Handling

10.1 Introduction

When T4EA accesses Enterprise Applications it has to use a physical connection. A connection is specified by the connection parameters, e.g. a web service URL or a database access string and the corresponding credentials, e.g. username and password. T4EA offers two fundamentally different ways of handling EA connections:

1. *Plain EA connection handling*: The custom mapping files use connection parameters that are "hard coded" within the mapping. The Teamcenter user does not see any connection properties and does not have to care about the connection to use. Instead the mapping "knows" all necessary parameters and credentials to connect.

2. *Managed EA connection handling*: T4EA manages the state of EA connections and the connection parameters and credentials for each named EA connection. The user may be given the possibility to choose between different connections for a certain action and T4EA can display a login dialog for a chosen connection, if needed.

The decision to use one of the alternatives can be taken either for the whole project or only for single connections, based on your requirements. The following chapters explain the alternatives together with the necessary configurations.

Note:

- T4EA EA connection handling only covers outbound service connections. Inbound services (web services) and the Teamcenter connection are not covered.

- Direct SAP and Oracle EBS connections must be handled by the T4S and T4O connection handling, which is very similar to the way T4EA handles managed connections, but different regarding the details. See the corresponding manuals.

10.2 Plain EA Connection Handling

This alternative is easy to implement since every time you access an Enterprise Application in the mapping, you just include the parameters in the mapping. The examples in this document mostly use this alternative.

Use of this alternative is recommended in the following situation:

- You want to hide the connection details from the Teamcenter users.

- The type of transfer uniquely defines the connection to use (n:1 relation from transfer types to connections) or you only have a single connection to a single EA system.

- The EA system can be accessed using a single technical user (instead of user dependent credentials).
- You are setting up a demo or prototype configuration of T4EA.
- You are migrating from pre-10.1 versions of T4EA and do not want to change the connection configuration.

If you want to disable managed connection handling generally, the following configuration is necessary:

- Pass the connection string (URL or database access string) directly to the TCL procedure executing the call, e.g.: `tpco_httpClient http://siemens.com/`.
- Set the site preference `T4EA.EaConnections` to the single value “default”, to indicate that there are no EA connections configured.
- Set the site preference `T4EA.Actions.ShowCustomData.RequiresEaConnection` to the single value “false” to indicate that the EA data view should not open a login dialog.
- Remove the values “CONNECT” and “EaConnections” from the site preference `T4EA.PredefinedMenuItems`, if present, to disable the connection dialog in the T4EA GUI - or - disable the T4EA menu completely by setting the site preference `T4EA.UI.GatewayMenu.Hide` to “true”.
- Implement the procedures `::T4EA::CONNECTION2EA::CUSTOM::MAPPING::checkConnection2EA4Session` and `::T4EA::CONNECTION2EA::CUSTOM::MAPPING::checkConnection2EA4Transaction` (both in t*4ea_connection_mapping_template.sd*) to always return `LOGIN_OK`.
- Delete the procedures `checkActiveEASystemForTargetTypeName`, `connectEA` and `getTargetTypeNames4Connection` in namespace `::T4EA::CONNECTION2EA::CUSTOM::MAPPING` (in *t4ea_connection_mapping_template.sd*), if present.

10.3 Managed EA Connection Handling

This alternative allows to access connections as named entities. T4EA is able to store and manage all the properties and credentials of connections of different types, provides UI dialogs to manage connections and login dialogs, but requires that the selection logic for assigning transfer types to connections is configured in the custom mapping. Since not all configured connections may be applicable to all transfer types, the mapping has the responsibility to decide and choose in this situation. For example an export configured to store item meta data in a database may be used with several database instances, but not with a web service connection. In another scenario your integration may require accessing two different Oracle databases, but each with a different schema, so these two connections are not considered equivalent.

Use managed EA connection handling in the following situation:

- Your scenario requires connections to several EA systems, especially if at least one of your transfers accesses several EA systems (at least a single 1:n relation between transfer type and EA connection, several EA connections are considered equivalent from a technical point of view).
- You want the Teamcenter user to explicitly choose a connection for certain actions.
- The EA system requires user dependent credentials (instead of a technical user) and the Teamcenter user must enter them.
- You are setting up a new T4EA integration project or are migrating from pre-10.1 T4EA and want to change the connection handling.

To use managed EA connection handling, the following configuration is necessary:

- Set the site preference `T4EA.EaConnections` to the names of all connections. This allows the T4EA GUI to display the table of connections in the connection dialog.
- Add the value "CONNECT" to the site preference `T4X.PredefinedMenuItems` (if not already present) to enable the connection dialog in the T4EA GUI.
- Set the site preference `T4EA.Actions.ShowCustomData.RequiresEaConnection` to the single value "true" to indicate that the EA data view should open a login dialog if necessary.
- Set or adapt the site preferences `T4EA.UI.PromptEaConnectionData.Attributes` to configure the connection dialog according to your needs. There are a lot more preferences for the connection dialog described in the **Teamcenter Gateway - Generic Configuration Guide**.
- Implement the procedures `::T4EA::CONNECTION2EA::CUSTOM::MAPPING::checkConnection2EA4Session` and `::T4EA::CONNECTION2EA::CUSTOM::MAPPING::checkConnection2EA4Transaction` (both in *t4ea_connection_mapping_template.sd*). These functions must check if the current active connection is applicable to the given session identifier or transfer type and can change the active connection, if required.
- Implement the procedure `::T4EA::CONNECTION2EA::CUSTOM::MAPPING::checkActiveEASystemForTargetTypeName` (in *t4ea_connection_mapping_template.sd*). It must check, if the current active connection is applicable for the given session identifier and transfer type.
- Implement the procedure `::T4EA::CONNECTION2EA::CUSTOM::MAPPING::connectEA` (in *t4ea_connection_mapping_template.sd*). This procedure must connect to the Enterprise Application using the given parameters and credentials. For session based connections (e.g. JDBC connections), the connection must be opened and kept open. For stateless connections (typically web services), this

procedure can just check, if the credentials are correct by calling a check function or simply return OK without a check.

- Implement the procedure `::T4EA::CONNECTION2EA::CUSTOM::MAPPING::getTargetTypeNames4Connection` (in *t4ea_connection_mapping_template.sd*). This procedure must return all the applicable transfer type names for the given connection identifier. T4EA uses this procedure to restrict the connections displayed to the Teamcenter user depending on the action in progress.

- Configure each named EA connection e.g. in *t4ea_mapping_config.sd*. To configure a connection for automatic login, use `::T4EA::CONNECTION2EA::setConnectionInfoPlain <EASystem> <EAConnectString> <EAUser> <EAPassword>`. Such a connection cannot be closed (disconnected) by the Teamcenter user via the T4EA GUI connect dialog. You can use `::T4EA::CONNECTION2EA::setConnectionInfo` instead to configure a connection using encrypted username and password. To just configure a connection to use a certain connect string but request user name and password from the Teamcenter user, use: `::T4EA::CONNECTION2EA::setConnectString <EASystem> <EAConnectString>`. Such a connection cannot be used in batch/job processing. The Teamcenter user must explicitly login before using this connection and can disconnect this connection via the T4EA GUI connection dialog.

- Optionally set active connections for all sessions (session identifier "`*`") or only special session identifiers using `::T4EA::CONNECTION2EA::selectActiveConnection2EA`.

- If you want job imports to use the connection, that was active at the time of creation of the job, set the parameter SessionId to a value identifying the active connection. E.g. the 6th parameter of the procedure `::T4X::BATCHJOB::IMPORT::createBatchjob` can be set to `"BATCH:[::T4EA::CONNECTION2EA::getCurrentEAInfo EASystem]"`. If the active connection at the time of the creation was e.g. "`Oracle1`" and additionally the active connection for the session "`BATCH:Oracle1`" was set to "`Oracle1`" (`set Status [::T4EA::CONNECTION2EA::selectActiveConnection2EA "BATCH:Oracle1" "Oracle1"]` in *t4ea_mapping_config.sd*), then the job will automatically use "Oracle1" as its active connection, regardless of the delay and the executing gateway service.

- Add the "EA logon task" to your Teamcenter workflows if they contain T4EA transfer handlers. If you want the task to check if the current active connection matches the intended action (TargetTypeName), add the handler `T4X-copy-Attributes-to-Task` and add an argument `-eax2TargetTypeName=<TransferType>`. If you want the workflow to use a certain connection, add `-use_EA_system=<EASystem>` to the arguments of the handler `T4EA-validate-EALogon`. If this handler is not present, the login dialog of the task will offer all configured connections, restricted only by your connection mapping logic based on the TargetTypeName given to the `T4X-copy-Attributes-to-Task` handler, if present.

- Finally in the mappings, retrieve the connection string (URL or database access string) of the current active connection from the T4EA managed EA connection handling and pass it to the TCL procedure executing the call. E.g.:

```
tpco_httpClient \
"[::T4EA::CONNECTION2EA::getCurrentEAInfo EAConnectString]?
param1=x&param2=y"
```

The procedure `::T4EA::CONNECTION2EA::getCurrentEAInfo` will return information on the current active EA connection and accepts the following parameters:

- *EAUser* will return the unencrypted (cleartext) user name (login name)
- *EAPassword* will return the unencrypted (cleartext) password
- *EASystem* will return the name of current active EA connection
- *EAConnectString* will return the connect string (e.g. URL, database access string)
- *EALanguage* will return the chosen language – can also be used for different purposes.

Caution:

Each connection contained in the GUI preference `T4EA.EaConnections` must also be configured by
either `::T4EA::CONNECTION2EA::setConnectionInfoPlain`, `::T4EA::CONNECTION2EA::setConnectionInfo` or `::T4EA::CONNECTION2EA::setConnectString`, otherwise undetermined behavior will occur.

10.4 Combining Plain and Managed EA Connection Handling

It is possible to use the managed EA connection handling for some of the configured connections, while using plain connection handling for others. This may give the user the possibility to select one of several equivalent connections for certain actions, while the system uses a hard coded connection for other actions, where no user interaction is necessary. Actions that use plain connection handling do not have to deal with connection handling and do not make use of any managed connection handling functionality. To bypass these checks you have to explicitly implement the bypass for these TargetTypeNames in the procedures `checkConnection2EA4Session`, `checkConnection2EA4Transaction` and `checkActiveEASystemForTargetTypeName` (in the `::T4EA::CONNECTION2EA::CUSTOM::MAPPING` namespace) in this case. Instead the action must use “hard coded” connection parameters and credentials.

A. Glossary

A

Admin
is the term used in this document for people who install and configure Teamcenter and its components. This is in contrast to the “user” role.

Apps
See "GS".

B

BGS
Basic Gateway Service.

BMIDE
Teamcenter Business Modeler IDE (Integrated Development Environment).

BOM
A Bill Of Materials is a list of the parts or components and their quantities that are required to build a product.

BOP
The Bill Of Process describes a manufacturing process and lists the operations and steps with all their instructions, consumed materials, resources, work places and machines.

D

Dataview mark-up
is the language understood by the Dataview. The Dataview receives messages written in this language from the T4x server. Such messages can be formatted as XML or JSON. Normally users do not see such messages. They may however appear in log files or error messages. The so called prop mapping (e.g. *t4s_prop_mapping_template.sd*) contains TCL commands that compose messages in the Data View mark-up.

E

EA
stands for Enterprise Application, any software or set of computer programs used by business users to perform various business functions in context of current integration's portfolio with Teamcenter.

ECN
The Engineering Change Notice can also be called an Engineering Change Note, Engineering Change Order (ECO), or just an Engineering Change (EC).

EPM
Enterprise Process Modeling.

G

GRM
The Generic Relationship Management provides a general way in which two objects can be associated via a relationship.

GS
Gateway Service, manages the communication between Teamcenter and the Enterprise Application.

GUI
Graphical user interface.

I

IDGEN
The IDGEN is a mechanism to get an external ID from the ERP system when assigning a Teamcenter ID.

ITK
The Integration Toolkit (ITK) is a set of software tools provided by Siemens PLM Software that you can use to integrate third-party or user-developed applications with Teamcenter.

J

JDBC
Java Database Connectivity is an application programming interface (API) for the programming language Java, which defines how a client may access a database.

L

LOV
List of Values.

M

MFK
Multi-key functionality in Teamcenter.

O

OOTB
Out of the box.

R

RAC
stands for Rich Application Client also referred to as rich client or portal.

S

SSL
Secure Sockets Layer.

T

T4x
The entire Teamcenter Gateway product family.

TC
Teamcenter

TCL
is a high-level, general-purpose, interpreted, dynamic programming language.

TEM
Teamcenter Environment Manager.

U

UOM
UOM stands for Unit of Measure.

URI
Unified Resource Identifier: a generalized from of a resource locator (URL) and resource name (URN), which just identifies a resource, but is not necessarily sufficient to locate (find) the resource. URIs are

often used to identify configurations in Java and other languages. See **https://en.wikipedia.org/wiki/Uniform_Resource_Identifier** for more details.

URL
Unified Resource Locator: a string with a certain format, allowing to load a resource from a network. URLs are a specific form or URNs.

X

XML
Extensible Markup Language is designed to store and transport data in a format that is both human- and machine-readable.

XRT
stands for XML Rendering Template, also known as XML Rendering Stylesheet. These are XML documents stored in datasets that define how parts of the Teamcenter user interface are rendered. They are used for the Rich Client as well as the Active Workspace.

Z

Z-Table
"Z" is the prefix name for custom tables well-known in SAP world.

Siemens Industry Software

Headquarters
Granite Park One
5800 Granite Parkway
Suite 600
Plano, TX 75024
USA
+1 972 987 3000

Americas
Granite Park One
5800 Granite Parkway
Suite 600
Plano, TX 75024
USA
+1 314 264 8499

Europe
Stephenson House
Sir William Siemens Square
Frimley, Camberley
Surrey, GU16 8QD
+44 (0) 1276 413200

Asia-Pacific
Suites 4301-4302, 43/F
AIA Kowloon Tower, Landmark East
100 How Ming Street
Kwun Tong, Kowloon
Hong Kong
+852 2230 3308

About Siemens PLM Software

Siemens PLM Software, a business unit of the Siemens Industry Automation Division, is a leading global provider of product lifecycle management (PLM) software and services with 7 million licensed seats and 71,000 customers worldwide. Headquartered in Plano, Texas, Siemens PLM Software works collaboratively with companies to deliver open solutions that help them turn more ideas into successful products. For more information on Siemens PLM Software products and services, visit www.siemens.com/plm.