Infor Cloverleaf Application Adaptor Web Services User Guide

Release 2022.x

DRAFT

Important Notices

The material contained in this publication (including any supplementary information) constitutes and contains confidential and proprietary information of Infor.

By gaining access to the attached, you acknowledge and agree that the material (including any modification, translation or adaptation of the material) and all copyright, trade secrets and all other right, title and interest therein, are the sole property of Infor and that you shall not gain right, title or interest in the material (including any modification, translation or adaptation of the material) by virtue of your review thereof other than the non-exclusive right to use the material solely in connection with and the furtherance of your license and use of software made available to your company from Infor pursuant to a separate agreement, the terms of which separate agreement shall govern your use of this material and all supplemental related materials ("Purpose").

In addition, by accessing the enclosed material, you acknowledge and agree that you are required to maintain such material in strict confidence and that your use of such material is limited to the Purpose described above. Although Infor has taken due care to ensure that the material included in this publication is accurate and complete, Infor cannot warrant that the information contained in this publication is complete, does not contain typographical or other errors, or will meet your specific requirements. As such, Infor does not assume and hereby disclaims all liability, consequential or otherwise, for any loss or damage to any person or entity which is caused by or relates to errors or omissions in this publication (including any supplementary information), whether such errors or omissions result from negligence, accident or any other cause.

Without limitation, U.S. export control laws and other applicable export and import laws govern your use of this material and you will neither export or re-export, directly or indirectly, this material nor any related materials or supplemental information in violation of such laws, or use such materials for any purpose prohibited by such laws.

Trademark Acknowledgements

Publication Information

Release: Infor Cloverleaf Application Adaptor-Web Services 2022.x
Publication Date: September 16, 2022
Document code: clfaaws_2022.x_claawsaawsolh__en-us

Disclaimer

This document reflects the direction Infor may take with regard to the specific product(s) described in this document, all of which is subject to change by Infor in its sole discretion, with or without notice to you. This document is not a commitment to you in any way and you should not rely on this document or any of its content in making any decision. Infor is not committing to develop or deliver any specified enhancement, upgrade, product or functionality, even if such is described in this document.

Contents

Contacting Infor....9

CAA-WS....10

Architecture and flow....12

Web Client working modes....13

API....15

Override fields....15

Field modes: SOAP/REST/Raw....16

CAA-WS USERDATA for getting information and setting overrides....17

USERDATA format....17

Provider inbound information....18

Client outbound overrides....20

Client inbound information....25

Provider outbound overrides....28

Open Java API....30

Local Binding....32

CAA-WS IDE properties GUI....34

CAA-Direct Retriever and CAA-Direct Sender....34

CAA-WS Client, CAA-WS RawClient and CAA-WS Server....35

ION Retriever....35

ION Retriever dialog box....36

ION Sender....38

ION Sender dialog box....39

Conduit....40

TLS Secured on the conduit....41

CAA-WS auto-creation of JKS for HTTPS....42

Creating a sample client....44

Creating a new conduit....45
Bus....45
Creating a sample server....46
Logical view....46
Creating an engine....47
Server IP addresses....47
Creating a RAW server....47
Placeholder REST server....48
Logical client items and their fields....48
Logical server items and their fields....57
Message validation check mode....62
Client overrides....63
Server overrides....66
Web Services consumer wizard....69
SOAP, REST and RAW basics....69
User interface....70
Building a SOAP Consumer....71
XSD WSDL tool....72
Selecting a WSDL file as input file....72
Selecting an XSD file as input file....73
WS-Client and WS-Server nodes....73
WS-Client conduit configuration....74
Conduit....75
WS-Client SOAP Consumer configuration....76
WS-Client REST Consumer configuration....77
WS-RawClient wizard flow....78
SPNEGO....80
Scheduler node....81
Web Services security....83
User interface....83
Testing....83
Certificate manager....84
Web services security use case....84
Usage scenario....85

Intended users....85
Basic flow....85
Alternate flow: Normal users....86
Web Service SOAP client: payload....86
Web Service SOAP client: message....86
Web Service RESTful client....86
Web Service Raw client....87
Web Service SOAP server: payload....87
Web Service SOAP server: message....87
Web Service RESTful server....88
Web Service Raw server....88
CAA-WS sample sites....89
REST....90
SOAP....91
SOAP Provider (MESSAGE mode)....91
Editing WS-Addressing....92
SOAP Provider (PAYLOAD mode)....92
SOAP Client....92
Asynchronous SOAP Client....93
Raw....93
Provider (Handler)....93
Raw Client....94
Asynchronous RAW Client....94
ws_more_samples....95
ws_adv_samples....95
Signing/Encryption....96
Understanding the CXF/WSS4J Options....97
FHIR....98
oauth2_sample....99
HL7 FHIR requirements and tools....102
Deploying Cloverleaf FHIR examples BOX....103
Cloverleaf BOX contents....104
Running examples....105
Running the FHIR patient create/update interface....106

Running the FHIR transaction bundle interface....108
Public FHIR test servers and this BOX....111
Updating FHIR schemas....111
Cloverleaf 6.2 translation Include operation....112
Using the Include operation....113
Creating an HTTP outbound web service client thread....114
CAA-WS Swagger....116
Raw Consumer configuration....116
OAuth2 client on the Conduit panel....117
XSD WSDL tool: Client....119
Usage scenario: Accessing a web service in the system....119
Client: Setting up single runs....120
Running XSD WSDL tool: Client version by command line....120
Running the XSD WSDL tool: Client by GUI....120
XSD WSDL tool: Server....122
Usage scenario: Creating a web service with the XSD WSDL tool....123
Server: Setting up single runs....123
Running the XSD WSDL tool: Server from command line....124
Running the XSD WSDL tool: Server from GUI....124
Portecle keystore management tool (third-party)....125
Java....125
Portecle open source GUI....125
Portecle installation....126
Launching Portecle....126
WS-Policy....127
Modifying the WSDL....128
Using the new WSDL....128
Providing valid usernames for server and select username for client....128
Java driver bug....129
Starting and testing threads....129
Policy files....130
Running fail test....130
jaxws:client and jaxws:server configuration properties....131
User properties....131

Callback class and crypto properties....132
Boolean WS-Security configuration tags....132
Non-boolean WS-Security configuration parameters....133
Encryption/signature class files....135
Callback classes....136
User interface for configuration and policy generation....137
Policy properties....138
USERDATA overrides....138
CAA-WS logging....140
Inbound/Outbound message logging....140
Cloverleaf message dump....140
CAA-WS internal logging....141
Output example....141
Enable Jetty access log....142
Updating CAA-WS 1.x sites to 2.0 and later....144
Migrating IHB threads to CAA-WS....146
Differences between IBMIME and CAA-WS messages....146
WSDL files....147
Server URLs....147
Configuration files....147
WS-Security....147
Server thread example....147
WSDL folder and server URLs....149
Removing Tcl procedures and changing routing values....149
Converting to CAA-WS server thread....150
Testing the changes....150
CAA-Direct....152
CAA-Direct architecture and flow....153
POP3/IMAP email retrieval usage....153
SMTP email sending usage....154
CAA-Direct Application Programming Interface (API)....155
CAA-Direct USERDATA for getting information and setting overrides....155
SMTP versus POP3 and IMAP....155

POP3/IMAP inbound information....156
SMTP outbound overrides....158
CAA-Direct IDE Properties GUI....162
Creating a sample sender....162
Sender object's sample site "GreenMailServer" test server....162
Sender's logical view....163
Additional sender configuration items....163
Creating a sample retriever....164
Retriever object's sample site GreenMailServer test server....164
Retriever's logical view....165
Additional retriever configuration items....165
Logical items and their fields....166
CAA-Direct usage scenario....173
Simple Message Sender....173
Message Sender with an attachment....174
Simple message retriever....174
Message retriever for an attachment....174
CAA-Direct sample sites....176
SMTP....176
SMTPS....178
POP3....179
POP3S....180
IMAPS....181
CAA-Direct Portecle Keystore Management tool (third-party)....182
CAA-Direct logging....184
Mail server conversation logging....184
Cloverleaf message dump....186
CAA-Direct internal logging....187
CAA-Direct known issues....189
Log files and troubleshooting....190
Frequently asked questions....191
Index....193

Contacting Infor

If you have questions about Infor products, go to Infor Concierge at https://concierge.infor.com/ and create a support incident.

The latest documentation is available from docs.infor.com or from the Infor Support Portal. To access documentation on the Infor Support Portal, select **Search > Browse Documentation**. We recommend that you check this portal periodically for updated documentation.

If you have comments about Infor documentation, contact documentation@infor.com.

CAA-WS

Cloverleaf Application Adapter Web Services (CAA-WS) is an extension to the core system functionality.

The Generic Java driver enables the engine to associate many threads with the Java protocol. This provides a way for the engine to communicate, by a public supported API, with Java applications running in one or more JVMs.

This foundation provides a platform for users to build custom Java applications that extend the core system engine.

CAA-WS, built using the Java driver, provides support for the prevailing web services paradigms, including:

- SOAP, RESTful, and optionally raw HTTP.
- Client and server.
- Synchronous and asynchronous.
- Advanced topics such as WS-* (for example, WS Security, SAML, mtom w/ attachments, and WS-addressing).

Customization points are for both power users and normal users:

- Normal users are users who follow samples and configuration guidelines. They also apply business logic in typical system application development methodologies, using Tcl API in UPoCs to use existing system functionality.
- Power users are those who must customize processing at the web services protocol level. They also deal with advanced web services topics, such as WS security, policy, and so on. These users are comfortable programming in both Tcl and Java. They have an understanding of how certain open source Java web services technology works. For example, CXF.
- Tcl UPoCs can be used to process WS requests and build a response.
- Message body is passed as a string to/from Tcl UPoCs.
- Metadata is passed as a keyed list to/from Tcl.
- Custom adapter development can be used for power users.

Provided with CAA-WS are:

- GUI tools that can assist with deployment configuration for CXF.
- Tutorials and sample sites.
- Help for existing users migrating from the IHB application to the WS adapter.
- Sample site that contains various examples of client and server applications. These are the starting point for learning how CAA-WS works. You can then modify the samples to create your own applications.

Knowledge levels

For those with a high level knowledge of CXF, these topics help you understand what beans, clients, and endpoints are created. Then, you can configure your own XML files.

For those with an intermediate understanding of CXF, you can use CAA-WS to get your configuration files started. Then, refer to the CXF documentation to configure any interceptors or handlers to add by directly editing the XML.

For those starting with CXF, stay with CAA-WS until you require CXF functionality that this tool does not support. Then, you can study the CXF documentation to modify your XML configuration files.

Architecture and flow

The CAA-WS is a Java Driver application that leverages the Apache Software Foundation's open source CXF web service framework.

Web service server usage

The web service server interacts with an external client.

- CXF comes with an embedded web server, jetty.
- This is an example of how a web service request is processed. This uses the Java Driver thread to route, based on TrxID, to two TCP threads where the Tcl UPoCs apply the business logic.

  This logic also resides in an inbound UPoC within the Java Driver thread without requiring additional TCP threads.
- CXF is based on JAX-WS, a standard Java web services stack. This has the concept of a Provider, or a Java class, that you write and plug into CXF to handle specific requests.
- CAA-WS bundles one or more Providers that do specific tasks. For example, getting the payload of the request and converting them into keyed lists. Then, putting them in USERDATA for subsequent Tcl processing down the message flow, and so on.
- To plug-in providers, specify in the CXF configuration file the Provider Java class. In addition, specify other parameters such as version of SOAP, URL of request, and so on.

  The configuration GUI assists in creating the correct entries in the CXF configuration file. The CXF configuration file can also be manually modified using Spring configuration conventions.
- You can also develop your own custom Provider. The JAX-WS is an open API. The source code for the CAA-WS Providers is available as samples. The CXF configuration is used to plug in the custom Provider.

Web service client usage

With the web service client:

- CXF is used for its client functionality, but the embedded web server, jetty, is not involved.
- CXF uses the concept of a "dispatch", or Java class, that you write and plug into CXF to make requests as a client. The CAA-WS bundles one or more dispatches that have specific tasks. For example, creating the payload of the actual web service request from content in keyed lists in USERDATA. Such keyed lists are from upstream business logic in Tcl.
- You can also develop your own custom dispatch. The JAX-WS is an open API. The source code for the CAA-WS Dispatches is available as samples.
- For example, a web service request can be created using an external TCP thread. This generates the keyed lists, where the Tcl UPoCs apply the business logic. The logic can also reside in an outbound UPoC within the Java Driver thread without requiring additional TCP threads.

To plug in dispatches, specify in the CXF configuration file the dispatch Java class. In addition, specify other parameters such as SOAP or RESTful, version of SOAP, URL of request, and so on.

The configuration GUI assists in creating the correct entries in the CXF configuration file. The CXF configuration file can also be manually modified, using Spring configuration conventions.

Web Client working modes

Working modes of web Clients are:

- Synchronous web Client as outbound
- Asynchronous web Client as outbound
- Scheduled web Client as inbound

Synchronous web Client as outbound

This is the first mode that is available to CAA-WS clients. The working flow of this mode matches the supporting Cloverleaf work flow.

Features include:

- Client requests are message-driven.
- Outbound message reply order is maintained. The next request is sent only when the former request response arrives.
- The request and its response have an internal link in the engine. The outbound await-replies option counts the correct time-out for a pair of "request and response".
- `Send OK` and `Send Fail` procs are called according to whether an expected response comes, or not.
- The MSI Outbound queue reflects the exact number of messages that have not been sent.
- The throughput is relatively small compared to the asynchronous mode.

Asynchronous web Client as outbound

Asynchronous clients are also message-driven.

Differences from synchronous clients include:

- Outbound message replies do not return to the engine in order. They are "first come, first served" by the engine.
- Unless the driver control flags are copied from requests to replies, the engine does not know which request a reply is associated with after the transaction is turned over by the protocol code.
  - Inside the protocol code:

    A reply has a link in code logic to its trigger request. Because this is not a common user point of customization, TCL access is not supported. Some Java customization is possible. For details, see Open Java API on page 30.
  - Outside the protocol code:

When an asynchronous client is in use, `Send OK` and `Send Fail` procs can only check if a message is acceptable to the protocol driver. This gets the service run planned in the thread pool. No response is available on points.

Inbound reply TPS:

The engine handles inbound response messages to `KILL` a message. This triggers time-out handling. The engine cannot resend any requests because there is no request-response link from the protocol driver. Similarly, time-out counting of await replies is incorrect in regards to a logically-linked request-response pair.

- The MSI outbound queue only reflects messages not yet scheduled by the protocol driver. In driver management, the queue is not yet visible to users, but there are copies in the Recovery database. In case of an unexpected blackout, the messages can be resent to the protocol driver.
- The throughput is mostly unhindered.

Scheduled web Client as inbound

A web client can also work as a web hook to retrieve online resources in loops.

- This mode is time-driven instead of message-driven.
- For HTTP actions, there is no requirement for payload. The client configuration is sufficient.
- For HTTP actions that require payload, you can define the `JaxWSMetaRequest` and `RawWSMetaRequest` bean classes in the client configuration to provide the necessary information. For details, see Open Java API on page 30.

API

Application Programming Interface (API) types include:

- Tcl user interface

  This is intended for implementers using a Tcl UPoC to process messages coming into the system from the CAA-WS.

- CXF provider/dispatcher (JAX-WS) API

  This is for users who require custom behavior other than the CAA-WS bundled providers/dispatchers. The source code for the CAA-WS providers/dispatchers is part of the distribution as samples.

  Note: If you develop and configure with CXF custom Providers or Dispatchers, then the Tcl user interface might not apply. This is because that interface is based on the CAA-WS bundled Providers/Dispatchers.

- CXF configuration interface

  This is an XML configuration file that determines the CXF behavior. Users can edit the file using the GUI tool or manually editing the file.

Override fields

For override fields, provide those fields for which the default value is incorrect. All others can be left blank, so defaults are used.

For example, to override only the `messageId` in WS-Addressing, there is no requirement to supply other WS-Addressing fields.

Put the main `wsa` field with the `messageId` that is nested within it. There is no requirement to supply `action` or any others, unless required.

For example:

```
{httpRequestHeaders {{Accept */*} {Cache-Control no-cache} {connection keep-alive} {Content-
Length 1392}
{content-type {application/soap+xml; charset=UTF-8}} {Host localhost:9003} {Pragma no-cache}
{User-Agent {Apache CXF 2.4.2}}}} {httpRequestInfo {{method POST} {requestURL http://local
host:9003/xdsregistryb}
{path /xdsregistryb}}}
```

This example shows:

- The format of the list.

- The nested structure.
- How the `USERDATA` works for a Provider and a Client using the request side.
- First-level keys in the list are `httpRequestHeaders` and `httpRequestInfo`.
- Second-level keys are:
  - `Accept`
  - `Cache-Control`
  - `connection`
  - `Content-Length`
  - `content-type`
  - `Host`
  - `Pragma`
  - `User-Agent`
  - `method`
  - `requestURL`
  - `path`

Inbound messages (Provider) include:

- `httpRequestHeaders` is a list of the HTTP request headers.
- `httpRequestInfo`

  The key values are various bits of information about the request:
  - `method` shows it was an HTTP POST.
  - `requestURL` shows the URL being requested.
  - `path` component of the URL is extracted for simplicity.

Outbound messages (Client) include:

- `httpRequestHeaders` overrides the outbound HTTP request headers that are sent by the system.
- `httpRequestInfo`

  The key values are various overrides:
  - `method` assures the type to be a POST.
  - `requestURL` is an override. When this is passed, the default URL is overridden.
  - `path` is a special case:

    If the `requestURL` override is present, then path is ignored as an override.

    If there is no `requestURL`, then the path appends, not overrides, to the default request URL path.

Field modes: SOAP/REST/Raw

These different modes have commonalities, for example, Providers put HTTP request headers in their information and can override HTTP response headers.

Other items are specific to only one or two modes, for example, WS-Addressing is only applicable to SOAP messages.

If a field is nested within another, then the mode is inherited from the parent unless otherwise specified.

These topics show the modes for which a given field is applicable:

- Provider inbound information
- Client outbound overrides
- Client inbound information
- Provider outbound overrides

CAA-WS USERDATA for getting information and setting overrides

The USERDATA field in the system messages sent to/from the CAA-WS both provides information and permits the setting of overrides.

- For basic web services, ignore this field if the default settings provide what is required.
- More complex web services processing, for example, soap fault handling, require one of these:
  - Reading the USERDATA on an inbound message and implementing different handling logic in UPoC code, depending on the contents.
  - Writing to the USERDATA on an outbound message to set specific HTTP header fields, a specific response code, overriding default WS-Addressing fields, and so on.

For example, USERDATA is used as information and overrides in this context.

The outside client calls the system Provider which processes the message and sends it to the system client which then calls the outside Provider.

Note: A server is often called a web services Provider, or Provider for short.

The response from the outside Provider is then passed back through the system to the outside client. This shows the USERDATA information that a Provider thread attaches to a system message. This is similar to the override USERDATA content that you can send to a client thread.

The reverse is where the response message from the client thread has USERDATA information that is attached to the system message. This is similar to the override USERDATA content that you can send to the Provider thread, instructing it how to reply to the outside client.

USERDATA format

The USERDATA field is used as a Tcl keyed list. A keyed list is a string obeying certain rules, where the list is made up of name/value pair entries.

In CAA-WS, a value is a basic string value, or is another keyed list, also known as nested keyed list. The nesting typically does not go beyond three levels deep.

Most data is at the second level. The first level is general categories.

Inbound messages are processed by the CAA-WS where various message metadata get deposited into the USERDATA field.

Outbound messages are processed by the CAA-WS which takes any information in the USERDATA field and uses it to override default behaviors.

Content from an inbound message to a Provider thread is the same format as the override content that is provided to an outbound client thread. The Provider thread receives a web service request. The client thread creates the web service request.

The same is true for the inbound USERDATA coming from a client thread. This receives a web service response. It can be used without modification to override settings on the outbound message from a Provider thread. This creates a web service response.

Provider inbound information

In the message inbound from the Provider thread, the USERDATA contains information about the request from the outside Client.

This table shows the information request parameters from outside clients.

A "*" in the list is the wildcard. This represents any other key names not listed in the same location of the map.

The "-" prefix in the list represents a sublayer. This indicates that the key following the "-" belongs to a map on a sub-level under the nearest above key with one less "-" prefixed.

Field (key)	Description	Modes
`httpRequestHeaders`	This contains a map of all the HTTP request headers. If the **HTTP request forwarding** option is set, then HTTP headers are moved to `orgHttpRequestHeaders`, unless the header is listed in "HTTP forward headers".	ALL
`-*`	HTTP request headers are named by the Client. These are items such as Content-Length, Content-Type, User-Agent, and Host. They can also contain any arbitrary value sent by the Client. The header name is the key and the header value is the value in the map. For multivalued headers, if they are represented as multiple lines in a request, then that header key can repeat the result map. However, starting from the second time, the header name has a suffix similar to `::{digit}`.	ALL
`httpRequestInfo`	This contains a map of fields giving further information about the HTTP request. If the **HTTP request forwarding** option is set, then this information is moved to `orgHttpRequestInfo`.	ALL
`-method`	The HTTP method. For example, GET, POST, PUT, and so on.	

Field (key)	Description	Modes
-requestURL	Full URL excluding the query string. For example, if a request came in to http://localhost:9005/raw/customer?id=123, then the resulting requestURL is http://localhost:9005/raw/customer.	
-path	The portion of the URL after the port. In the requestURL example, this is "/raw/customer."	
- pathInfo	In Raw mode, the path is further broken down into the portion after the first section. Typically, in a URL such as this, raw would be considered the web app. It is useful to know what the path is after the web app. In the requestURL example, pathInfo would be customer.	Raw
-query	The query string portion of a URL. In the requestURL example this is "id=123."	
-oneWay	This is true/false. This indicates that the request is a one way request and no response should be sent. CAA-WS automatically sends back an HTTP 202 response to the Client.	SOAP
-clientip	The IP address from which the request came.	
-clientport	The port from which the request came.	
-authorization	If HTTP Basic, Digest, or Negotiate Authentication was used to authenticate this request, then this map key is present. Raw mode can handle these types of authentication but the information is not present in USERDATA.	SOAP REST
--user	The user name used to authenticate.	
--type	The type of authentication used–Basic, Digest, or Negotiate.	
attachments	This is a map of all attachments in the message. Raw mode can accept attachments but the entire message, including attachments, is sent to the Handler. Attachments are not broken out separately.	SOAP REST
* (identifier)	Each attachment is itself a nested map of its components where the key is the identifier for the attachment. If the attachment already has a designated ID, for example, a Content-ID header, then that ID is used. If there is no existing ID, then one is automatically generated as (“IB” + unique number).	
--xop	A boolean true/false value to indicate whether the attachment is XOP.	
--headers	A further nested map containing the headers for the attachment.	

Field (key)	Description	Modes
--*	The header names are the keys in this map, with the header values being the values in the map. This is similar to `httpRequestHeaders`.	
--content	A base64 encoded string containing the attachment's content. This is used when the `cloverleaf-attachment-dir` field is missing or blank in the configuration XML.	
wsa	WS-Addressing information. Fields are only populated when they are not null on the inbound message.	SOAP
-action	This required element, whose content is of type xs:anyURI, conveys the value of the [action] property.	
-faultTo	This optional element, of type `wsa:EndpointReferenceType`, provides the value for the [fault endpoint] property.	
-from	This optional element, of type `wsa:EndpointReferenceType`, provides the value for the [source endpoint] property.	
-messageId	This optional element, whose content is of type xs:anyURI, conveys the [message ID] property.	
-relatesTo	This optional, repeating, element information item contributes one abstract [relationship] property value, in the form of an (IRI, IRI) pair. The content of this element, of type `xs:anyURI`, conveys the [message ID] of the related message.	
-replyTo	This optional element, of type `wsa:EndpointReferenceType`, provides the value for the [reply endpoint] property. If this element is not present, then the value of the [address] property of the [reply endpoint] EPR is `http://www.w3.org/2005/08/addressing/anonymous`.	
-to	This optional element, whose content is of type `xs:anyURI`, provides the value for the [destination] property. If this element is not present, then the value of the [destination] property is `http://www.w3.org/2005/08/addressing/anonymous`.	

Client outbound overrides

In the outbound message from the Client thread, `USERDATA` that is passed in overrides default behaviors and values. These `USERDATA` values are almost all exactly the same as the Provider inbound information.

This table shows information on client outbound overrides.

A "*" in the list is the wildcard. This represents any other key names not listed in the same location of the map.

The "-" prefix in the list represents a sublayer. This indicates that the key following the "-" belongs to a map on a sub-level under the nearest above key with one less "-" prefixed.

Field (key)	Description	Modes
`httpRequestHeaders`	This contains a map of all the HTTP request headers	ALL
`-*`	HTTP request headers can sometimes have arbitrary names, and others have well-known special meanings. These are items such as Content-Length, Content-Type, User-Agent, and Host. The header name is the key and the header value is the value in the map. Case does not matter. 3. Notes: • Do not attempt to override the SOAPAction header. This must be completed under **httpRequestInfo > soapAction**. • Setting Content-Type here fails when sending attachments, as CXF controls the Content-Type header in that case. Otherwise, it should work. • Use a comma to combine multivalued HTTP headers into one key-value pair. You can also have multiple pairs, whose keys have a suffix after the second occurrence. For example, `::{digital}`.	
`httpRequestInfo`	This contains a map of fields permitting further overrides regarding the HTTP request.	ALL
`- method`	This is the HTTP method. For example, GET, POST, PUT, and so on. Typically, POST is the default.	
`- requestURL`	Full URL excluding the query string. For example, to change the request to `http://localhost:9005/raw/customer?id=123`, the resulting `requestURL` is `http://localhost:9005/raw/customer`. In this example, exclude the query string here.	
`- path`	The portion of the URL after the port. In the `requestURL` example, this is `/raw/customer`. If a `requestURL` is provided, then this field is ignored. If no `requestURL` is provided, then the path appends to the default request URL path.	
`- pathInfo`	Although this field is present in the information from a Provider, it is ignored by the Client thread. This cannot be used as an override.	N/A
`- query`	The query string portion of a URL. In the `requestURL` example, this is "`id=123`."	

Field (key)	Description	Modes
- `soapAction`	This property sets the SOAPAction HTTP header in SOAP 1.1 messages, and the action attribute of the Content-Type header in SOAP 1.2 message. If WS-Addressing is enabled, then this also sets the Action attribute there, regardless of SOAP version. This is a special override, so it should not be used with HTTP header overrides when using a SOAP Client. This is because CXF performs special operations, such as WS-Addressing/WS-Policy handling, with regard to the soapAction setting. It must be set here and not as an HTTP header override. This is only true when using the SOAP Client. If a SOAP message is sent using a Raw Client or the REST Client, then manually set the SOAPAction header by an HTTP Header override. This applies to the SOAP 1.1 or the Content-Type header's action attribute for SOAP 1.2	SOAP
- `oneWay`	True/False. This indicates that the request is a one way request and no response should be sent. This field is required when communicating with a one way service Provider. Otherwise, the CXF Client expects a response and catches exceptions when the HTTP 202 comes back from the Provider. No Cloverleaf reply message is created in this case.	SOAP
- `tls`	A map of TLS (SSL) overrides. Currently there is only the `certAlias` override.	

Field (key)	Description	Modes
`--certAlias`	The alias of the certificate in the keystore to present that identifies the Client when the TLS connection is being established. This causes an exception when this key is present and TLS is not configured on the Client. **Note:** There are a few issues to be aware of when using this key: • This updates the Conduit object. Any subsequent requests to that Conduit use the same `certAlias`, even when the key is not sent in those subsequent requests. This is contrary to the vast majority of overrides which apply only to that single message. • The Conduit caches the SSL session by default. Subsequent requests within the default time limit use the existing session. To ensure that different `certAlias` values always use their own SSL session, the SSL cache timeout must be set to “0." This indicates it does not cache SSL sessions. This affects performance as it requires a new SSL session to be created for every message. **Note:** Using this mechanism for authorization instead of authentication is not recommended. Use HTTP Basic, Digest/Negotiate, Authentication for authorization purposes, or WS-Security’s `UsernameToken`. For debugging an SSL or TLS handshake that is not working, go to the process’s Java driver configuration and set this user-defined property: `javax.net.debug=ssl:handshake` This logs many SSL handshake information that is useful for debugging SSL or TLS, but can rapidly overwhelm the log file when used in production.	
`-authorization`	If HTTP Basic or Digest Authentication is enabled, then the keys within this map override the configurations on a per-transaction basis. It is acceptable to override as much as required, as fields not specified here use their configured values. If not already enabled, then these keys are used to set up HTTP Authentication.	
`--user`	The user name used to authenticate.	
`--password`	The password belonging to the user.	
`--type`	The authentication type is Basic, Digest, or Negotiate. If not specified, then the default is Basic.	
`attachments`	Map of all attachments to be sent. Raw mode can send attachments, but the entire message, including attachments, is sent as the message body. Attachments are not broken out separately.	

Field (key)	Description	Modes
- * (identifier)	Each attachment is itself a nested map of its components where the key is the identifier for the attachment. The value that is specified is used as the attachment ID. This is set as the Content-ID header.	
- - xop	A boolean true/false value to indicate whether the attachment is to be sent using XOP.	SOAP REST
- - headers	A further nested map containing the headers to override for the attachment.	
- - - *	The header names are the keys in this map, with the header values being the values in the map. This is similar to httpRequestHeaders. Content-Type header defaults to application/octet-stream when no value is provided.	
- - content	Base64 encodes your attachment content to be sent here. This field or the contentFile field is required.	
- - contentFile	File name for a file containing the attachment content. This is used for large attachments to improve performance versus the costs of base64 encoding. The file name is a relative path to the Cloverleaf process directory or an absolute path. This field or content field is required.	
wsa	WS-Addressing information. Descriptions here are the override possibilities. All values are already set based on defaults or other configurations so the necessity to override these should be rare.	SOAP
- action	The **Action** field should be overridden by setting httpRequestInfo->soapAction, which sets this and sets the SOAP Action appropriately for SOAP 1.1 or 1.2. If you require it, then this property can be overridden here.	
- replyTo	This is for making an asynchronous request. You can provide an endpoint here. The recommended approach is to create a CXF HTTP Conduit containing a Client element. Set DecoupledEndpoint to the URL to listen on for asynchronous responses. Setting DecoupledEndpoint in the configuration XML automatically sets this replyTo header to that DecoupledEndpoint value. As with the Action header, you can also override this one here when necessary.	
- to	This is automatically set to the requestURL you are calling, but can be overridden here.	

Field (key)	Description	Modes
-messageId	If not overridden, then this is set with a generated unique value. You can override this when you are making a pass-through type application with the value received by the Provider thread.	
-from	See definition, no specialization here.	
-faultTo	See definition, no specialization here.	
-mustUnderstand	This option overrides the key when any of the other listed keys must be understood by the server. The value for this is a list with a comma. For example: to,action	
trxid	If the settings for this Client tell it that the TrxID on the response should come from USERDATA, then this field is required. The response TrxID is then set as the field value.	ALL
wss	There are many possible child keys here and the operations can be complex. See USERDATA overrides.	SOAP
cookies	This contains a map of all HTTP request cookies.	RAW
-*	The cookies to be added to the requests in key-value pairs. For multivalued cookies, a key can be used multiple times with each key having a different suffix. For example, :: (digit).	
form	This is the information for the client to compose the request, in style of a form.	RAW
-dataMap	This contains the fixed inputs entries for the form.	
--*	Inputs name-value entries for the form. This is similar to cookies and HTTP headers. If a name must be listed more than once, then it must have a suffix. For example, :: (digit).	
-payloadKey	Form requests do not have a HTTP payload or the payload occupied by the form. This key directs the client where Cloverleaf messages are placed inside a form request. When this key is set, the form is appended with a new input. For example, <value of this key>=<Cloverleaf message>.	

Client inbound information

In the message inbound from the Client thread, the USERDATA contains information about the response from the outside Provider.

This table shows the information response parameters from outside providers.

A "*" in the list is the wildcard. This represents any other key names not listed in the same location of the map.

The "-" prefix in the list represents a sublayer. This indicates that the key following the "-" belongs to a map on a sub-level under the nearest above key with one less "-" prefixed.

Field (key)	Description	Modes
`fault`	This field only shows up if the response was a SOAP fault. The message body contains the SOAP fault XML. This field is a place to check if a fault has happened before processing the message body.	SOAP
`-faultActor`	Contains the **Fault Actor** field	
`-code`	Code is a QName field. Here it is represented as a map with the attributes of the QName inside.	
`--namespaceURI`	Namespace of the code QName	
`--localPart`	Local part name of the code QName	
`-faultString`	Explanation of the fault.	
`-faultNode`	SOAP 1.2 only	
`-faultReasonText`	SOAP 1.2 only	
`-faultRole`	SOAP 1.2 only	
`-subCode`	SOAP 1.2 only. First tier of sub-codes.	
`-subCode_1`	SOAP 1.2 only. Second tier of sub-codes and beyond. Labeled as subCode_X, where {X|1..infinity}.	
`httpResponseCode`	Integer value of the HTTP response code. 200 means success in most situations.	ALL
`httpResponseHeaders`	This contains a map of all the HTTP response headers.	
`-*`	HTTP response headers are named by the Provider. These are items such as Content-Length, Content-Type, and Server, but can contain any arbitrary value sent by the Server. The header name is the key and the header value is the value in the map. For multivalued headers, if they are represented as multiple lines in a request, then that header key can repeat the result map. Starting from the second time, the header name contains a suffix. For example, `::{digit}`.	
`attachments`	Map of all attachments in the message. Raw mode can accept attachments but the entire message including attachments, is sent to the Handler. Attachments are not broken out separately.	

Field (key)	Description	Modes
- * (identifier)	Each attachment is itself a nested map of its components where the key is the identifier for the attachment. If the attachment already has a designated ID such as a Content-ID header, then that ID is used here. If there is no existing ID, then one is automatically generated as ("IB" + unique number).	
- - xop	A boolean true/false value to indicate if the attachment was XOP.	SOAP REST
- - headers	A further nested map containing the headers for the attachment.	
- - - *	The header names are the keys in this map, with the header values being the values in the map. Similar to httpResponseHeaders.	
- - content	A base64-encoded string containing the attachment's content. This is used if the cloverleaf-attachment-dir field is missing or blank in the configuration XML.	
- - contentFile	If the cloverleaf-attachment-dir field exists in the configuration XML, then this field is sent instead of the base64-encoded content field. This field contains the file name that holds the attachment content. This is relative to the cloverleaf-attachment-dir that is specified in configuration. It is expected that the Cloverleaf application deletes this file after it has been used. This is useful to improve performance relative to base64-encoding in the case of large attachments. If the cloverleaf-attachment-dir is specified as a relative path, then the contentFile USERDATA value has the correct file name but an incorrect path. This is due to a bug in GJD's handling of Java's working directory. You can do one of these: • Use the file name relative to the cloverleaf-attachment-dir under the process directory, ignoring the path component. or • Specify an absolute path for the cloverleaf-attachment-dir in which case the full path and file name are correct.	

Field (key)	Description	Modes
wsa	WS-Addressing information. Fields are only populated if they are not null on the inbound message. This is the same as Provider inbound, except there is no replyTo header here.	SOAP
- action	This required element, whose content is of type xs:anyURI, conveys the value of the [action] property.	
- faultTo	This optional element, of type wsa:EndpointReferenceType, provides the value for the [fault endpoint] property.	
- from	This optional element, of type wsa:EndpointReferenceType, provides the value for the [source endpoint] property.	
- messageId	This optional element, whose content is of type xs:anyURI, conveys the [message id] property.	
- relatesTo	This optional, repeating, element information item contributes one abstract [relationship] property value, in the form of an (IRI, IRI) pair. The content of this element, of type xs:anyURI, conveys the [message id] of the related message.	
- to	This optional element, whose content is of type xs:anyURI, provides the value for the [destination] property. If this element is not present, then message id value of the [destination] property is http://www.w3.org/2005/08/addressing/anonymous.	

Provider outbound overrides

In the outbound message from the Provider thread, USERDATA that is passed in overrides default behaviors and values. These USERDATA values are almost all exactly the same as the Client Inbound Information.

This table shows provider outbound override information.

A "*" in the list is the wildcard. This represents any other key names not listed in the same location of the map.

The "-" prefix in the list represents a sublayer. This indicates that the key following the "-" belongs to a map on a sub-level under the nearest above key with one less "-" prefixed.

Field (key)	Description	Modes
httpResponseCode	Integer value of the HTTP response code. 200 means success in most situations.	ALL
httpResponseHeaders	This contains a map of all the HTTP response headers.	ALL

Field (key)	Description	Modes
`-*`	HTTP response headers can sometimes have arbitrary names, and others have well-known special meanings. These are items such as Content-Length, Content-Type, and Server. The header name is the key and the header value is the value in the map. Case does not matter. Notes: • Setting Content-Type here fails when sending attachments, as CXF controls the Content-Type header in that instance. Otherwise, it should work. • Setting Content-Length is not advisable unless you require working with another system that has an issue or something you are trying to work around. • You can combine multivalued HTTP headers into one key-value pair by using a comma. You can also have multiple pairs, whose keys have a suffix after the second occurrence. For example, `::{digital}`.	
`attachments`	This is a map of all attachments to be sent. Raw mode can send attachments but the entire message, including attachments, must be sent as the message body. Attachments are not broken out separately.	SOAP REST
`-*` (identifier)	Each attachment is itself a nested map of its components where the key is the identifier for the attachment. The value you specify here is used as the ID for the attachment and is set as the Content-ID header.	
`--xop`	A boolean `true`/`false` value to indicate if the attachment is to be sent using XOP.	
`--headers`	A further nested map containing the headers to override for the attachment.	
`---*`	The header names are the keys in this map, with the header values being the values in the map. This is similar to `httpRequestHeaders`. Content-Type header defaults to "application/octet-stream" if no value is provided here.	
`--content`	Base64-encode your attachment content to be sent here. This field or `contentFile` field is required.	
`--contentFile`	File name for a file containing the attachment content. This is used for large attachments to improve performance versus the costs of base64 encoding. The file name should be a relative path to the Cloverleaf process directory or an absolute path. This field or the content field is required.	

Field (key)	Description	Modes
wsa	WS-Addressing information. Descriptions here are the override possibilities. All values are already set based on defaults or other configurations, so the necessity to override these should be rare.	
- action	In general, this is automatically set from the WSDL, but can be overridden here, if necessary.	
- replyTo	This is not used in the normal Request/Response MEP. If the response is expecting a further message to be sent to the Provider, then this is where you specify the message's destination URL.	
- to	This is automatically set to the requestURL you are calling, but can be overridden here.	
- messageId	If not overridden, then this is set with a generated unique value. You can override this if you are making a pass-through type application with the value received by the client thread.	
- from	This optional element, of type wsa:EndpointReferenceType, provides the value for the [source endpoint] property.	
- faultTo	This optional element, of type wsa:EndpointReferenceType, provides the value for the [fault endpoint] property.	
- relatesTo	Automatically populated with the message ID from the in-bound request to which this response is replying. This would rarely require overriding, except for the Client that can override it for a pass-through application. Even for a pass-through app, this should require overriding on the Provider.	
wss	There are many possible child keys here and the operations can be complex. See USERDATA overrides.	SOAP

Open Java API

In addition to TCL overrides, CAA-WS uses open Java APIs. Details are written in JavaDoc and are located at *HCIROOT*/CAA/ws/OpenJavaAPIDoc.

These APIs are classified in two parts:

- com.infor.cloverleaf.gjdws.message.RawWSMetaRequest and com.infor.cloverleaf.gjdws.message.JaxWSMetaRequest

You can use these two beans to provide metadata for schedule mode web clients in configuration files. Their functionality is almost the same as the TCL APIs in overriding the USERDATA. The difference is that for schedule mode clients, there is no triggering message to flow through any TPS. Details are located in the `JavaDoc`.

- `com.infor.cloverleaf.gjdws.async.AsyncWsClientNoExceptionHandler` and `com.infor.cloverleaf.gjdws.async. AsyncWsRawClientNoExceptionHandler`

  These two classes are the Java access points for asynchronous mode clients. When implemented, these handlers are a supplement for users who have lost track of request-response pairs in regular TPS such as inbound reply TPS in asynchronous mode.

  The `cloverleaf-no-exception` option must be "true" to make the handlers effective. Additionally, CAA-WS has a default handler implementation when the option is "true". Details are located in the `JavaDoc`.

Local Binding

In some instances, it is better to bind a connection to a particular local IP on NIC. This is already performed for the CIS TCP protocol.

For example, in some countries have multiple governmental web services. One of the security layers is where the government web server verifies the IP address that a user is employing. In these instances, users require the option to bind to a certain IP that is known to be safe.

Similar situations also happen in the cloud and HA environments. Without an option to configure a binding to an InetAddress to use within a Cloverleaf HA environment, the connection then comes from a local NODE IP. This is undesirable.

The **Edit** tab of the **Web Services Consumer** contains a text field in which users can configure a particular IP address. The SOAP Consumer, REST Consumer, and RAW Consumer of the WS Consumer, and their edit tabs, all support usage of that particular IP address.

GUI design

The WS Consumer **General** tab contains a **Local Binding Address** text field. Characteristics of **Local Binding Address** include:

- Support for configuring an IP address or a host name.

  This is optional. The default is blank, and connection is made to the server as before.
- The location of **Local Binding Address** is different on different WS Consumers.
  - On the REST/SOAP Consumer, it is below the **Address** field.
  - On the Raw Consumer, it is below the **Default Method** field.

Local Binding Address is validated on the user interface. When the local binding address is invalid, such as "`!!!`", a warning message opens: "`The local binding address is detected as an invalid address. Do you want to continue?`".

Data Flow

The local binding address for GUI data flow is:

The SOAP/REST/RAW Consumer GUI requests the NetConfig Server to get the local binding address from `applicationContext_`*`ThreadName`*`.xml`. It then saves the value of **Local Binding Address** from the GUI into this file.

Testing on the GUI

To test on the GUI:

1. Create a java/ws-client protocol thread and click **Properties**. This opens the **WS Client** dialog box.
2. Click **New** to create a SOAP/REST Consumer.
3. Select the created consumer and configure an IP address or a host name into **Local Binding Address** on the **General** tab.
4. Click the **OK** button to apply the change and then **Save** the NetConfig.
5. Reopen the **WS Client** dialog box and the configured IP address or host name should successfully load. Or, you can check the configuration in `applicatinContext_ThreadName.xml`.
6. Create a java/ws-rawclient protocol thread, then create a RAW Consumer and follow the previous steps 1-5.
7. Use the configured **Local Binding Address** (IP address or host name) to connect to the server. If this is an invalid IP address or host name, then the consumer cannot connect to the server. An exception is logged into `Process.log` located in the *Process* folder.

CAA-WS IDE properties GUI

CAA-WS is built upon the Java driver. When CAA-WS is installed, new protocols that are based upon the Java driver are available in the Cloverleaf IDE.

These are the CAA-WS installed protocols. Clicking the **Properties** button opens a dialog box specific to each of these protocols. For example, when **`ws-client`** is selected, you can create SOAP and REST client configurations. These dialog boxes automatically configure the Java driver and create the XML configuration files that configure the CAA-WS components.

For the XML structure of the configuration files which the GUI creates, refer to the *Apache CXF configuration guide*. See http://cxf.apache.org/docs/configuration.html.

The CAA-WS protocols are configurable similar to other protocols. You select the protocol and use the dialog box that displays by clicking **Properties** to configure it. Click **Apply** to retain your updates and save the NetConfig. This is the same for CAA-WS as for other protocols.

When a new thread is created and a process name is specified, or when you change the thread or process name, you must click **Apply**. This must be finished before clicking **Properties** to edit the configuration in the dialog box. This is because the **Properties** dialog box must know the actual thread/process information to write its configuration files.

For Cloverleaf 6.0.1 users, there is a small bug that requires you to click **OK**, apply changes, and reopen the **Properties** dialog box. This must be finished before configuring SOAP clients or servers. Otherwise, you receive an error when trying to browse for your WSDL. This is fixed in Cloverleaf 6.1 and later versions.

You should not put two separate Java driver threads in the same process, because CAA-WS is based on Java driver. There are some cases where this can work with Java driver, but with CAA-WS it causes trouble. You must ensure they are in separate processes. They can share with other protocols, but not other CAA-WS or other Java driver threads.

Do not change the Java Driver process working directory. When you create one of these product's threads in a given process, the working directory defaults to `$SITEPATH/javadriver/process name`. This should not be changed as it can cause errors. All file relative paths are automatically computed when you select the various **Browse** buttons to select files.

CAA-Direct Retriever and CAA-Direct Sender

CAA-Direct is built upon the Java Driver. When CAA-Direct is installed, protocols that are based upon Java Driver become available in your system IDE.

The Network Configurator's **Protocol** menu lists the CAA-Direct installed protocols.

When **`direct-sender`** is selected, you can create SMTP configurations.

When **`direct-retriever`** is selected, you can create POP3 or IMAP configurations.

CAA-WS Client, CAA-WS RawClient and CAA-WS Server

CAA-WS is an extension to the core system functionality, built upon the Java driver. When CAA-WS is installed, new protocols that are based upon the Java driver are available in the Cloverleaf IDE.

In the Network Configurator, the **Protocol** menu lists the CAA-WS installed protocols:

- When **`ws-client`** is selected, you can create SOAP and REST Client configurations.
  - A SOAP Client does SOAP/HTTP calls to a web server sending/receiving SOAP messages.
  - A REST Client does HTTP calls to a web server sending/receiving XML messages. Other message types are not supported.

When **`ws-rawclient`** is selected, you can configure HTTP calls to a web server.

When **`ws-server`** is selected, you can create SOAP Server, REST Server, and Raw Server configurations.

- A SOAP server receives SOAP/HTTP calls and replies with a SOAP message.
- A REST server listens for HTTP requests containing XML messages. Other message types are not supported.
- A Raw server answers HTTP calls.

These dialog boxes configure the Java driver automatically and create the XML configuration files that configure the CAA-WS components.

ION Retriever

With the CAA-ION adapter, you can send outbound Business Object Documents (BODs) to ION IOBox database tables (IOBox). You can also retrieve inbound BODs from an IOBox. An IOBox is a predefined set of tables in a database that ION knows how to interact and communicate with using predefined BODs.

The GUI is integrated directly inside the Cloverleaf IDE. It writes all the configuration files necessary without the user doing manual edits, except in specialized circumstances. Users have the ability to create custom overrides before inbound/outbound message transmissions.

ION-Retriever configuration is performed through the Network Configurator by selecting the java/ion-retriever protocol. This protocol retrieves and sends BODs and maps to the IOBox COR_INBOX* tables.

For the Retriever, an inbound message on the CAA-ION thread is sent to the outbound threads.

- The Retriever class inside the Java Driver thread initiates a JDBC query (request) to the database server. You can configure this to run as often as necessary.

- The database returns a list of BOD messages in the IOBox's INBOX* tables, which constitutes the JDBC response.
- For each BOD message on the list, the Retriever class sends a Cloverleaf message out through the Java Driver thread. When the message is successfully stored in the recovery database, the BOD message's rows are deleted from the IOBox tables. The delete transaction is committed immediately after successful entry of the message into the recovery database. This repeats until all BODs in the IOBox are processed.

In Server/Retriever/Inbound mode, when the runtime is set to retrieve BODs from an IOBox, it is acting as a server. It does this by placing inbound messages into the Cloverleaf engine. You can set up a server thread by creating a java/ion-retriever protocol thread.

The Retriever queries BODs from IOBox. As such it is not strictly a server because messages are not sent to it from ION directly. Cloverleaf polls the configured IOBox(s) periodically to retrieve any new BODs.

Cloverleaf receives messages from ION and passes them into Cloverleaf as an inbound pre-inbound TPS

The configuration file can use multiple IOBox Retriever configurations. These configurations:

- Provide all necessary fields to make the connection to the IOBox as a single database user.
- Specify which Tenant Id and To Logical Id entries that this Retriever is acquiring. A given set of IOBox tables can be shared between different applications. In this way, a thread should only extract rows from the database that is addressed to its Tenant Id and To Logical Id combination.

The IOBox that is listed in the configuration file is periodically processed. The period between connections is configurable using the Advanced Scheduler.

ION Retriever dialog box

The ION Retriever is configured using the **`java/ion-retriever`** protocol.

This is referred to as server mode, because it acts as a server thread to Cloverleaf. It retrieves messages from an IOBox database and sends them inbound to Cloverleaf.

Selecting the **`java/ion-retriever`** protocol and clicking **Properties** opens the ION Retriever dialog box.

The **Retrieve Interval** field applies to all of the Retriever configurations within the thread. This specifies in seconds how often to check all the IOBox configurations for new BODs. If left blank, then the default is 30 seconds.

This table shows the available fields for individual Retriever configurations:

Field/Option	Description
ID	Optional. This is used to distinguish similar tabs. If an ID is used, then it must be unique with respect to all other IDs in the thread's configuration.
Database Type	Required. Choices are **`SQLSERVER`**, **`Oracle`**, **`DB2`**, or **`DB2_400`**.
Host	Required. This is the IP or DNS name of the database server. Specifying a host automatically fills in the URL field.

Field/Option	Description
Port	Required. This is the port on which the database server is listening. The default SQL Server port is 1433.
Database Name	Required. This is the name of the database schema to connect to.
URL	This is the Database URL.
User, **Password**, and **Confirm Password**	Required. These are the log-in fields to authenticate against the database. The password can be encoded using the standard Java driver encoding by selecting **Password Encoded**. When using an un-encoded password, the engine prints out the encoded password at startup. You can take this encoded password and overwrite the un-encoded password. The difference is that the password is stored in the application context XML file in an encoded form. In this way, users cannot accidentally view other users' passwords.
Tenant Id	This is the tenant value for retrieving BODs. If left blank, then it uses the ION default of infor.
To Logical Id	Required. This is the logical ID value for retrieving BODs. This is so that an application can use the same IOBox as another, yet keep its data separate within the same tenant. This is required because there is no default value.
Cloverleaf TrxId Determination	This is for selecting the method of determining the TrxId. If left blank, then no transaction ID is sent. Options are: • **`TOLOGICALID`**. This uses **To Logical Id** as the TrxId. • **`FROMLOGICALID`**. This uses **From Logical Id** as the TrxId. • **`TENANTID`**. This uses **Tenant Id** as the TrxId. • **`BODTYPE`**. This uses the `Verb.Noun` BOD Type header as the TrxId. • **`VALUE`**. This places the value from **Cloverleaf TrxId Value** as the TrxId. This is useful with multiple Retriever configurations in one thread • **`NULL`**. This explicitly declares the TrxId to not be set.
Cloverleaf TrxId Value	This is used as the TrxId if **Cloverleaf TrxId Determination** is set to **`VALUE`**.

Field/Option	Description
Cloverleaf Log Exceptions	This is used to disabled the logging of exceptions if they become too numerous. For example, if a database server is known to be down during the night for daily maintenance, the logs could become too large. When this is the case, the user might elect to disable these exceptions.

ION Sender

The CAA-ION adapter provides system users the ability to send outbound BODs (Business Object Documents) to ION IOBox database tables (IOBox). It also retrieves inbound BODs from an IOBox. An IOBox is a predefined set of tables in a database that ION knows how to interact and communicate with using predefined BODs.

The GUI is integrated directly inside the Cloverleaf IDE. It writes all the configuration files necessary without the user doing manual edits, except in specialized circumstances. Users have the ability to create custom overrides before inbound/outbound message transmissions.

ION Sender configuration is performed through the Network Configurator by selecting the java/ion-sender protocol. This protocol retrieves and sends BODs and maps to the IOBox COR_OUTBOX* tables.

For the Sender, the outbound message is sent by the CAA-ION thread to the IOBox OUTBOX* tables.

- An inbound Cloverleaf thread sends a message to the Java Driver thread.
- After this, the Sender class within the Java Driver thread sends the message by JDBC to the IOBox's OUTBOX* tables.
- The Sender class is completed after it successfully commits the database transaction. The Cloverleaf thread receives no reply message.

In Client/Sender/Outbound mode, the IOBox Sender thread sends a BOD to an IOBox as a Cloverleaf outbound thread. Sender sends messages out to the preconfigured IOBox. There are no reply messages sent or received by Cloverleaf.

When the runtime is set to send BODs from an IOBox, it is acting as a client. It gets outbound messages from the Cloverleaf engine and sends those messages to an IOBox. You set up a client thread by creating a **`java/ion-sender`** protocol thread.

The configuration file can use multiple IOBox Sender configurations.

Cloverleaf metadata field `USERDATA` are used on outbound Cloverleaf messages to perform overrides. These features are available to override:

- Configuration to use if there are multiple sender configurations present in the config file. If there is only one configuration, then no selection is necessary.
- Default ION headers.
- Inclusion of other ION headers to send in addition to the default header (for example, headers that change with every message).

ION Sender dialog box

The ION Sender is configured using the **`java/ion-sender`** protocol.

This is referred to as client mode, because it acts as a client thread to Cloverleaf. It gets outbound messages from Cloverleaf and sends them to the IOBox database.

Selecting the **`java/ion-sender`** protocol and clicking **Properties** opens the **ION Sender** dialog box.

For sending, there is only one Sender configuration type. A new ION Sender creates additional instances of it.

The **Sender** entry contains connection information for making the database connection. The various default values assist in reducing USERDATA workload by setting defaults for various IOBox column and header fields.

This table shows the available fields:

Field/Option	Description
ID	This is optional if there is only one Sender entry and is used to distinguish similar tabs. If an ID is used, then it must be unique with respect to all other IDs in the thread's configuration. • When there is only one Sender entry, that entry is automatically used to send BODs. • When there is more than one Sender entry, the ID is passed in USERDATA. This is used to distinguish for a given message which sender entry to connect with.
Database Type	Required. Choices are **`SQLSERVER`**, **`Oracle`**, **`DB2`**, or**`DB2_400`**.
Host	Required. This is the IP or DNS name of the database server. Specifying a host automatically fills in the URL field.
Port	Required. This is the port on which the database server is listening. The default SQL Server port is 1433.
Database Name	Required. This is the name of the database schema to connect to.

Field/Option	Description
User, **Password**, and **Confirm Password**	Required. These are the log-in fields to authenticate against the database. The password can be encoded using the standard Java driver encoding by selecting **Password Encoded**. When using an unencoded password, the runtime prints out the encoded password at startup. You can take this encoded password and overwrite the unencoded password. The difference is that the password is stored in the application context XML file in an encoded form. In this way, users cannot accidentally view other users' passwords.
Default Message Priority	This is an integer value for setting message priority. There is no value in the BOD to indicate this. It must be set here or in the USERDATA as an override. The default is 4, with 0 being the lowest priority message and 9 the highest.
Default Tenant ID	This is used if the BOD does not provide a tenant ID.
Default From Logical ID	This is used if the BOD does not provide a logical ID for the sender.
Default Accounting Entity	This is used if the BOD does not provide an accounting entity. This is for message tracking in ION.
Default Location	This is used if the BOD does not provide a location. This is for message tracking in ION.
Enable Message Validation Check	When **true** is selected, this causes various checks to be run against an outbound message, looking for mistakes. This uses extra CPU resources and should be used only in development. For example, this checks if a BOD field value does not match the default value in the configuration or the override value in USERDATA. This is because header data that does not match the BOD could be the source of an error.

Conduit

A conduit is used to set HTTP connection details. For example, a conduit can put TLS on the connection. A conduit can also add HTTP basic authentication, proxy settings, and time-out settings.

A conduit uses a single `name` attribute. This specifies an expression to be matched against the various clients that exist in a configuration file.

The conduit name can be in different formats:

- A regular expression to match a URL.

  For example, `http://somehost:port/url*`. This indicates that the conduit can associate with the clients whose address is `http://somehost:port/url`, or a subset of that path.

- `<soap port>.http-conduit`

  For example: `{urn:ihe:iti:xds-b:2007}DocumentRegistry_Port_Soap12.http-conduit`. This indicates that the conduit associates with the SOAP clients whose port is `{urn:ihe:iti:xds-b:2007}DocumentRegistry_Port_Soap12`

- `*WebClient.http-conduit`

  The conduit associates with all Raw clients.

- `*.http-conduit`

  The conduit associates with all clients.

A REST client follows this order until it locates the matched conduit:

1. Locate a conduit whose name is a URL regular expression and matches the address of the REST client.
2. Locate a conduit whose name is `*.http-conduit`.

A SOAP client follows this order until it locates the matched conduit:

1. Locate a conduit whose name is a URL regular expression and matches the address of the SOAP client.
2. Locate a conduit whose name is a Port format and matches the port of the SOAP client.
3. Locate a conduit whose name is `*.http-conduit`.

A Raw client follows this order until it locates the matched conduit:

1. Locate a conduit whose name is a URL regular expression and matches the address of the Raw client.
2. Locate a conduit whose name is `*WebClient.http-conduit`.
3. Locate a conduit whose name is `*.http-conduit`.

If the current Consumer's URL is an HTTPS type, then it must have a Conduit configured with TLS to access.

If any existing conduit matches, then it can be directly used. You are not required to create a conduit. Otherwise, a new conduit with TLS should be created to ensure the HTTPS Consumer functions properly.

When creating a new conduit, by default, "Name" is filled with the URL from the Consumer's address with an asterisk. For example, `http://somehost:port/url*`.

TLS Secured on the conduit

HTTP addresses can specify TLS Secured on the conduit.

The default value of TLS Secured depends on the URL Consumer type:

- This is cleared and disabled by default when the Consumer address is an HTTP URL.
- This is enabled and selected by default when the address is an HTTPS URL.

CAA-WS auto-creation of JKS for HTTPS

With this feature, you can generate a keystore/truststore without having to use command-line tools.

You can:

- **Generate Keystore**/**Regenerate Keystore**
- **Generate Truststore**/**Update Truststore**

These actions are located at:

- **WS Raw Client/WS Client > Conduit > TLS tab > Keystore/Truststore panel**
- **WS Server > Engine > TLS Secured panel**

To begin the **Conduit Creating** wizard:

1. Click **Add** in the **WS Client/WS Raw Client** dialog box. Then, select **Create Conduit**.
   You can also select **Create Raw/SOAP/REST Consumer** and then select **Create New Conduit** in the **Create Conduit** page.
2. On the **Create Conduit-Basic Settings** page, select **TLS Secured**.
3. On the **Create Conduit-TLS Settings** page, generate the keystore/truststore.

Keystore generation

A **Generate Keystore** button is located on the Keystore panel of the Engine Panel TLS Section, **Conduit TLS Panel**, and **Conduit Creating** wizard. Clicking this opens the **Generate Keystore(JKS) With Certificate** dialog box.

Note: Only JKS is currently supported.

On the **Generate Keystore(JKS) With Certificate** dialog box, you can generate a keystore with a self-signed certificate. To do this, specify the keystore information and certificate information.

Clicking **More...** opens the **Certificate Information** dialog box. In this dialog box, you can specify all information related to certificate generation.

These fields are available for Generate Keystore:

Name	Description
Keystore Name*	Name of the keystore. Only lowercase letters, numbers, dashes, and underscores are allowed. Do not include file name extensions.
Keystore Directory*	Directory of the keystore.
Keystore Password*	Password of the keystore.
Confirm Password*	Confirm password for keystore.
Certificate Password*	Password of the certificate.
Confirm Password*	Confirm password for the certificate.

Name	Description
Alias*	Alias of the certificate. Click **More. . .** to add the certificate information. This opens the **Certificate Information** dialog box. • **Common Name*** • **Country** • **Email** • **Locality** • **Organization** • **State** • **Unit** • **Validity Period (Days)** The default is “3650”.

If the keystore already exists, then this becomes a **Regenerate Keystore**. This opens the **Regenerate Keystore(JKS) With Certificate** dialog box. The keystore Information fields are already populated.

A **Replace Existing Keystore?** warning opens when you reuse the same keystore name and directory.

After generation has completed, these files are located under the specified directory:

- *keystore_name*-cert.der
- *keystore_name*-key.der
- *keystore_name*.jks

Truststore generation

A **Generate Truststore** button is located on the Truststore panel of the **Conduit TLS** panel and **Conduit Creating** wizard.

Click this button to open the **Import Certificate to Generate Truststore(JKS)** dialog box.

Note: Only JKS is currently supported. You cannot generate truststores on the server side **Engine** panel.

You can generate the truststore by importing certs from the server to which you intend to connect.

To do this, specify the truststore and certificate information in "host:port" format. Other required fields pertaining to the truststore must also be specified.

These fields are required to generate a truststore:

Name	Description
Truststore Name*	Name of the truststore. Only lowercase letters, numbers, dashes, and underscores are allowed. Do not include file name extensions.
Truststore Directory*	Directory of the truststore.
Truststore Password*	Password of the truststore.
Confirm Password*	Specify the truststore password again to confirm.

Name	Description
Server (Host:Port)*	Host and port to get certs. Example: localhost:7443

If a truststore already exists, then there is an **Update Truststore** button. This opens the **Import Certificate To Update Truststore(JKS)** dialog box.

- If the Certificate Server(host:port) is the same, then the old certificate in truststore is replaced with the new certificate.
- If the Certificate Server(host:port) is different, then the new certificate is appended after the old certificate.

After generation is done, a *truststore_name*`.jks` file is located under the specified directory.

Creating a sample client

This covers creating a SOAP Client. REST and Raw are similar.

1. Select the **`java/ws-client`** protocol and click **Properties** (click **Apply** after setting the thread/process name for your new thread).
2. Click **New** to start the wizard, then click **Create SOAP Consumer**.
3. Click **Next**. On the Import WSDL page, specify a **WSDL URI***.

   This example uses Weather Service. This WSDL is available at `http://wsf.cdyne.com/WeatherWS/Weather.asmx?WSDL`.

   Note: Using a WSDL at an HTTP address can be dangerous if the WSDL becomes unavailable. This causes the client service to fail to start. It is suggested to save a copy of a WSDL to the local hard disk and access it from there.
4. Click **Next**.
5. Specify this information:

   Address is for address overrides. In this example, it is set to empty to use the address supplied in the WSDL. The address is used when a server implements a widely available WSDL based on a standard where the address in the WSDL is generic. In this case, you must override it with the address of the server with which this is interacting.

   Selecting a **Service** populates the **Port** list. There are multiple ports within this service, so the port is not auto-selected, but would be if there was only one. There are two SOAP ports named WeatherSoap and WeatherSoap12. These are for SOAP 1.1 and SOAP 1.2 ports. In this example, the later SOAP 1.2 is selected.
6. For **Service Mode**, selecting PAYLOAD (recommended) tells the system to send/receive only the contents of the SOAP Body element. CXF handles the SOAP Envelope/Header.

   Use PAYLOAD mode unless there is information in the SOAP Header that is required or must be sent that CXF does not already handle.

   MESSAGE indicates the system user is sending/receiving the entire SOAP Envelope. This permits greater flexibility but is also more complicated and error prone.
7. Click **Next**. This opens the Generate and Compile XSDs wizard page.

To generate an XSD, fill the fields and click **Generate and Compile Target XSDs**, otherwise, click **Next** to skip the generation. All schemas are imported from the WSDL file from the step above.

Operation selects the first item by default. The **Target XSD filename** field is changed according to the selection of this field.

Soap Header Preference selects **`Any headers are allowed`** by default.

Click **Generate and Compile Target XSDs** to generate and compile the XSD out of the current WSDL. XSDs are generated under `$HCISiteDir/formats/xml/{THREADNAME}/` using the names `{OPERATION_NAME}_input.xsd` and `{OPERATION_NAME}_output.xsd`. The Results area shows the output and any errors during XSD generation..

8 Click **Next**. This opens the Create Conduit wizard page.

Creating a new conduit

To create a new conduit:

1 Select **Create New Conduit** and click **Next**.

The ensuing wizard pages are for setting conduit details. See Conduit on page 75.

2 Specify a conduit name. See TLS Secured on the conduit on page 41.

If an existing conduit matches, then you are not required to create a conduit. Otherwise, a new conduit with TLS is created to ensure the HTTPS Consumer functions.

3 Specify **TLS Secured** on the conduit using an HTTP address.

The default value of **TLS Secured** depends on the URL Consumer type:

- This is cleared and disabled by default when the Consumer address is an HTTP URL.
- This is enabled and selected by default when the address is an HTTPS URL.

Bus

A Bus is a bucket whose children apply to everything else "on the Bus."

For example, a SOAP client does not have WS-Addressing checked or logging enabled. If the Bus does, then the client inherits the ws-addressing and logging features.

This permits numerous clients in one thread config, and turning logging on/off for all at one time.

Creating a sample server

This section covers creating a SOAP Server, that is, Provider, as they are the most complicated. REST Servers are similar.

1. Select the **`java/ws-server`** protocol and click **Properties**.
2. Click **New** and select **SoapProvider**. The help at the top explains the general flow.
3. For **WSDL URI**, specify the WSDL and any referenced XSD files within the working directory for the Java driver thread. Then, reference it with a relative path.

   As there is only one service in the WSDL and only one port within that service, they are auto-selected. In most cases, the WSDL has a generic URL which is not the URL to deploy your service. Use the address field to override it and set a path for the URL.

   This example shows an XDS.b Registry. If this service is deployed to another machine and the same WSDL path does not exist, then the service cannot find the WSDL. In this instance, it fails to start.

   For example, a WSDL `XDS.b_DocumentRegistry.wsdl` is in the ws_sample site's `WS/XDSb.Support.Materials.v10/wsdl` directory. Click **Browse** to locate it, and **Load WSDL** to parse it.
4. **Service Mode** is the same as that in the previous Client section.
5. **Published Endpoint URL** is an optional field for populating the address in the generated WSDL when a Client adds `?wsdl` to the service address.

   For example, a service resides on a machine within a company's firewall named `amazing42`. The outside world is routed into this machine using the DNS name `cloud.amazing.com`.

   If the service is at `http://amazing42/myservice`, then you can get the WSDL within the network by calling `http://amazing42/myservice?wsdl`. This generates a WSDL with the service address as expected as `http://amazing42/myservice`.

   This works for internal testing, but when customers run it at `http://cloud.amazing.com/myservice?wsdl`, a public service address other than `http://amazing42/myservice` must be returned. If not, then their clients fail without address overrides.

   In this case, put `http://cloud.amazing.com/myservice` in **Published Endpoint URL**. The returned WSDL now has the correct service address to calling users.

Logical view

The properties of this server are found on the right side panel where they can be specified or selected from a menu. Pausing on a property opens the tooltip that explains the property.

As with the SOAP Client, the general properties are grouped under the **General** tab.

For configuring the WS-Policy settings, see WS-Policy.

Creating an engine

Selecting **Engine** opens the **Add Jetty Engine** dialog box, where you specify a **Port**.

An Engine represents a Jetty engine instance, which is to say that it controls how Jetty listens on given IP address and port. It is used to configure such things as threading and TLS security configuration on a given port.

If you specify only the port and no host, then the settings control that port on all interfaces. Specifying an IP address in the host box restricts the settings to that port on that specific IP address. Port is always required.

Properties are then set to configure how Jetty handles a single port. For example, port 9000 can be a mutually authenticated TLS server port and run with 100 listening threads.

The **Host** property configures the network interface on which this port is listening. This is left blank to listen on all interfaces for the host.

Server IP addresses

When setting up a Provider, it has an endpoint URL on which it listens.

- If the URL's port matches a port in the list of server engines, then the listening IP address is determined by the engine's host parameters.
- If the URL's port does not match any ports in the list of server engines, then the URL's IP address is used.

The IP address for a server, or Provider, can restrict from where requests can come.

- If the IP address is 127.0.0.1, then only clients on the same host can connect to the Provider.
- If the IP address is 0.0.0.0, then requests can come from any network interface that is connected to the computer. This can also happen if you are using an engine where the host is not specified,
- If the IP address is an address for a specific network interface (for example, 192.168.1.5), then only connections coming into that network interface are accepted. For example, if the machine has two NICs and the other one is on 10.0.0.2, then only requests to 192.168.1.5 are accepted.

Creating a RAW server

Selecting **`RawHandler`** opens the **Add Raw Server** dialog box, where you specify a Path and Port.

A Raw server is different from a SOAP or REST Server.

SOAP/REST runs in CXF which runs in Jetty. Raw is designed to be as close to the HTTP stack as possible with the system. It runs on top of Jetty and outside CXF.

The configuration is also different from SOAP/REST.

A Raw server is a Jetty Handler object. It is a handler within an engine within the engine-factory. Further, it is a handler within a handler.

Click **New Raw Server** in the **Properties** dialog box of the sample thread that was created for the sample SOAP server. This opens the **Add Raw Server** dialog box:

- **Port** is the port on which to listen. For example, **`9000`**.
- **Path** is where all requests starting at that path are handled. For example, **`/raw`**.

This creates a new engine on the port on which to listen. Or, this reuses an engine if one is already configured. If you created an engine on port 9000 with the SOAP example, then it opens a message stating the engine already exists. This is not an error, only a notice.

Within the new engine a context handler is added to handle all requests to a given path or below. For example, `/raw` or `/raw/whatever/path/works` are also handled.

Inside that context handler goes the CAA-WS handler that takes all requests to that context and passes them to the system.

Placeholder REST server

CXF is made so that it only starts engines for ports which have a configured CXF server.

When a Raw server is created on a port that does not have another configured CXF server, a placeholder REST server is also created. This is created on the same port, so that CXF starts the port.

Select it to see that the address is `http://localhost:9000/dummyServerToStartPortForRawServer`.

This is a placeholder path not meant to be used. The port is 9000. That starts that 9000 engine. The handler within it is also started.

Switching to the physical view shows the added corresponding elements.

The context handler grabs all requests to the `/raw` context. Within that context is the handler to send all those requests to the system.

Note: Do not delete the placeholder REST endpoint unless there is another SOAP or REST server listening on that same port. This ensures that the engine gets started.

Logical client items and their fields

This table lists the logical client items.

Item	Field	Description
SOAP Client		
	Name	Dispatch name
	WSDL Location	URI to the WSDL
	Service Name	QName showing service name within the WSDL

Item	Field	Description
	Port Name	QName showing port name within the WSDL
	Service Mode	Combo box containing PAYLOAD, MESSAGE
	Address	Address override field
	WS-Addressing	Whether to enable WS-Addressing
	- Allow Duplicates	Boolean: Only enabled if WS-Addressing is enabled
	- Addressing Required	Boolean: Only enabled if WS-Addressing is enabled
	Logging	Enables CXF message logging. **Note:** The default size limit at which messages are truncated in the log is 102400.
	auth.spnego.useKerberosOid	Selecting this with `true` causes `SpnegoAuthSupplier` to use the OID (Object identifiers) for Spnego. Some servers require the OID for Kerberos.
	auth.spnego.requireCredDelegation	Selecting this with `true` causes the receiving service to implement the credential delegation.
	Cloverleaf TrxId Determination	This is a list that contains `SOAPACTION`, `VALUE`, `USERDATA`, and `NULL`.
	- Cloverleaf TrxId Value	String field that is enabled only if prior Determination is set to VALUE
	Cloverleaf Attachment Directory	Relative or absolute path to store inbound attachments
	Cloverleaf Copy DriverControl	Boolean to copy inbound driver control to the outbound response message. `false` is the default.

Item	Field	Description
	Cloverleaf No Exceptions	Boolean, where a `true` value declares that all messages should get a response message, even if it is an error message. `false` is the default and indicates that exceptions cause the message to go to the Cloverleaf error database. If `true`, then the runtime attempts to return a message to the calling Cloverleaf thread. In general, when something is incorrect there is an exception thrown and the message gets thrown into the error database. When this setting is `true`, CAA-WS attempts to return a message instead. In the case of an error, that message contains an `exception` key in the USERDATA response. That keyed list entry has a nested keyed list containing `message`. This is the error message. Optionally, there is a `cause` key if the exception has cause information. There is always a `stackTrace` key holding the full error stack trace. There is a `level` key with the values `send` or `top`. • `send` indicates it failed to send from a connection problem. If the user has selected **Copy driver control** or chosen to set the Trxid, then those things are performed on the response message. • `top` indicates there is a more serious programming error and it cannot copy driver control or set a Trxid on the response message. In this case, implementers must check for this key in Tcl code and handle it appropriately.
	Schema Validation Enabled	Boolean to validate inbound response messages against WSDL's schema. `false` is the default.
	MTOM Enabled	Boolean to enable MTOM support. `false` is the default
	Request Header Overrides	Provides a list of headers to send/override in outbound request
REST Client		
	Name	Dispatch name
	Address	Except for SOAP, this is required because there is no WSDL from which to get the address.

Item	Field	Description
	Logging	Enables CXF message logging. **Note:** The default size limit at which messages are truncated in the log is 102400.
	auth.spnego.useKerberosOid	Selecting this ("true") lets SpnegoAuthSupplier use the OID(Object identifiers) for Spnego. Some servers require the OID for Kerberos.
	auth.spnego.requireCredDelegation	Selecting this ("true") lets the receiving service implement the credential delegation.
	Cloverleaf TrxId Determination	Combo box - VALUE, USERDATA, NULL
	- Cloverleaf TrxId Value	String field that is enabled only if prior Determination is set to VALUE
	Cloverleaf Attachment Directory	Relative or absolute path to store inbound attachments
	Cloverleaf Copy DriverControl	Boolean to copy inbound driver control to the outbound response message. `false` is the default.

Item	Field	Description
	Cloverleaf No Exceptions	Boolean where a `true` value declares that all messages should get a response message, even if it is an error message. `false` is the default and indicates that exceptions cause the message to go to the Cloverleaf error database. If `true`, then the runtime attempts to return back a message to the enabled Cloverleaf thread. In general, when something is incorrect there is an exception thrown and the message gets thrown into the error database. When this setting is `true`, CAA-WS attempts to return a message instead. In the case of an error, that message contains an `exception` key in the USERDATA response. That keyed list entry has a nested keyed list containing `message`. This is the error message. Optionally, this is a `cause` key if the exception has cause information. It is always a `stackTrace` key holding the full error stack trace. There is a `level` key with the values `send` or `top`. • `send` indicates it failed to send from a connection problem. If the user has selected **Copy driver control** or chosen to set the Trxid, then those are performed on the response message. • `top` indicates there is a more serious programming error and it cannot copy driver control or set a Trxid on the response message. In this case, implementers must check for this key in Tcl code and handle it appropriately.
	Request Header Overrides	Provides a list of headers to send/override in outbound request
Raw Client		
	Name	Web Client name
	Address	Except for SOAP, this is required because there is no WSDL from which to get the address.
	Default Method	Combo box - HTTP Methods: GET (default), `POST`, `HEAD`, `PUT`, `DELETE`, `OPTIONS`, `TRACE`, and `CONNECT`.
	Cloverleaf TrxId Determination	Combo box - VALUE, USERDATA, NULL
	- Cloverleaf TrxId Value	String field that is enabled only if prior Determination is set to VALUE

Item	Field	Description
	Cloverleaf Copy DriverControl	Boolean to copy inbound driver control to the outbound response message. `false` is the default.
	Cloverleaf Attachment Directory	Relative or absolute path to store inbound attachments.
	Cloverleaf No Exceptions	Boolean, where a `true` value declares that all messages get a response message, even if it is an error message. `false` is the default. This indicates that exceptions cause the message to go to the Cloverleaf error database. If `true`, then the runtime attempts to always return a message to the enabled Cloverleaf thread. In general, when something is incorrect there is an exception thrown and the message gets thrown into the error database. When this setting is `true`, CAA-WS attempts to return a message instead. In the case of an error, the message contains an `exception` key in the USERDATA response. That keyed list entry has a nested keyed list containing `message`. This is the error message. It could optionally have a `cause` key, if the exception has cause information. It always has a `stackTrace` key that holds the full error stack trace. There is a `level` key with the values `send` or `top`. • `send` indicates it failed to send from a connection problem, and if the user has selected **Copy driver control** or chosen to set the Trxid. These are performed properly on the response message. • `top` indicates there is a more serious programming error and it cannot copy driver control or set a Trxid on the response message. In this case, implementers must check for this key in Tcl code and handle it appropriately.
	Request Header Overrides	Provides a list of headers to send/override in outbound request
HTTP Conduit		
	Name	Pattern with which to match outbound messages
	Client	Group of settings
	- Connection Timeout	Default 30000

Item	Field	Description
	- Receive Timeout	Default 60000
	- Auto Redirect	Default false - Follow server redirects
	- Max Retransmits	Default - 1 Number of redirects to follow
	- Allow Chunking	Default true
	- Chunking Threshold	Default 4096
	- Accept	HTTP Accept header
	- Accept Language	What language (for example, American English) the Client prefers for purpose of receiving responses
	- Accept Encoding	Specifies what content encodings the Client is prepared to handle
	- Content Type	HTTP ContentType header
	- Host	HTTP Host header
	- Connection	Combo box - Keep Alive, Close
	- Cache Control	Values and their meaning are documented on the CXF website at Client Cache Control Directives.
	- Cookie	Static cookie to be sent with all requests
	- Browser Type	HTTP User Agent header
	- Referer	HTTP Referer header
	- Decoupled Endpoint	Specifies the URL for the Client to listen on for asynchronous replies. Requires WS-Addressing. All calls to the server using this Client are performed asynchronously with this as the ReplyTo address.
	- HTTP Authentication	Group of fields to send HTTP Basic, Digest, or Negotiate Authentication
	-- username	The username with which to authenticate
	-- password	The password with which to authenticate
	-- type	Basic, Digest or Negotiate authentication type
	- Proxy	Group of fields to set outbound communication proxy
	-- server	Proxy server address
	-- port	Proxy server port

Item	Field	Description
	-- type	SOCKS or HTTP proxy type
	-- non proxy hosts	List of hosts with which to not use proxy
	-- Proxy HTTP Authentication	HTTP Basic, Digest or Negotiate Authentication for the proxy connection. Child fields are exactly the same as the above HTTP Authentication section. See that section for child fields.
	TLS Client Parameters	Group of settings for TLS support
	- Disable CN Check	Boolean. CN = Common Name.
	- Secure Socket Protocol	Combo box - SSL, TLS, TLSv1, TLSv1.1, TLSv1.2. To use TLSv1.1 or TLSv1.2, the process must be running within a Java 7 JVM. Cloverleaf 6.1 and above already run Java 7. For earlier releases, see the Cloverleaf Java Driver documentation for information on how to use your own JVM in a Java Driver thread.
	- SSL Cache Timeout	Milliseconds
	- Keystore Type	JKS, JCEKS, PKCS12
	- Keystore Password	Password for the entire keystore
	- Keystore Path	Absolute or relative path from the thread's working directory.
	- Keystore Key Password	Password for keys within the keystore.
	- Truststore Type	JKS, JCEKS, PKCS12
	- Truststore Password	Password for the entire truststore
	- Truststore Path	Absolute or relative path from the thread's working directory.
	- Cipher Suite	A comma-separated list of supported cipher suite names. This cannot be combined with the cipher suites filter.
	- Cipher Suites Filter Include	A comma-separated list of regular expressions matching cipher suite names to include. This cannot be used with the Cipher Suite list.
	- Cipher Suites Filter Exclude	A comma-separated list of regular expressions matching cipher suite names to exclude. This cannot be used with the Cipher Suite list.

Item	Field	Description
Bus This works on clients and servers.		
	WS-Addressing	Only affects SOAP Clients/Servers
	- Allow Duplicates	Boolean: Only enabled if WS-Addressing is enabled
	- Addressing Required	Boolean: Only enabled if WS-Addressing is enabled
	Message Logging	Enables CXF message logging for all components except Raw Servers. **Note:** The default size limit at which messages are truncated in the log is 102400.
	Message Validation Check Mode	Boolean to enable detailed checks of the outbound message. It is CPU intensive and should only be enabled in development to check for coding errors, as it slows production systems. This checks the outbound message content against the MESSAGE/PAYLOAD setting. It also checks for USERDATA keys looking for incorrect keys and SOAPAction overridden as a header. It tries to build a list of the probable mistakes being made in an outbound message and puts that list in the process log. By default, this is cleared.
	Enable GZIP Encoding	This enables RawHandler to compress the inbound and outbound messages.
	Asynchronous Message Delivery	This enables all message delivery options.
	-Core Pool Size	The core number of threads. The default is 8.
	Maximum Pool Size	The maximum allowed number of threads. The default is 64.
	Keep Alive Time	This is the amount of time that threads in excess of the core pool size can remain idle before being terminated. The default is 30 seconds.
	Shutdown Timeout	The maximum time to wait for the completed execution after a shutdown request. The default is 15 seconds.
	Cloverleaf No Exception Handler	The full class name that is used to customize the error handler.

Logical server items and their fields

This table lists the logical server items:

Item	Field	Description
SOAP Server		
	WSDL Location	URI to the WSDL.
	Service Name	QName showing service name within the WSDL.
	Port Name	QName showing port name within the WSDL.
	Service Mode	Combo box: PAYLOAD, MESSAGE.
	Address	Address override field.
	Published Address	Override generated address when someone requests WSDL with ?wsdl.
	WS-Addressing	Whether to enable WS-Addressing.
	- Allow Duplicate	Boolean: Only enabled if WS-Addressing is enabled.
	- Addressing Required	Boolean: Only enabled if WS-Addressing is enabled.
	Logging	Enables CXF message logging. **Note:** The default size limit at which messages are truncated in the log is 102400.
	Cloverleaf TrxId Determination	Combo box: SOAPACTION, PATH, NULL.
	Cloverleaf Timeout	Default is 30000.
	Cloverleaf Log Exceptions	Boolean: Should Cloverleaf log exceptions or not. is `false`.
	Cloverleaf Attachment Directory	Relative or absolute path to store inbound attachments.
	Cloverleaf Request Forward	When this is selected, the HTTP request headers from the inbound thread are removed to avoid interfering with and overriding the information set by the next outbound thread. This does not happen when their names are listed in "Cloverleaf Forward Headers". Otherwise, the HTTP request headers will interfere with the next outbound thread.

Item	Field	Description
	Cloverleaf Forward Headers	This assists in the selection of the HTTP request header name list from the inbound thread to be forwarded to the next outbound thread. This is a comma or space separated string. For example: "mytestheader1,mytestheader2" or "mytestheader1 mytestheader2". This is only enabled when "Cloverleaf Request Forward" is selected.
	Fault Stack Trace Enabled	Boolean: Should a fault response contain a stack trace or not. Default is `false`.
	Exception Message Cause Enabled	Boolean: Should a fault response contain a cause or not. Default False.
	Schema Validation Enabled	Boolean: Whether to validate inbound messages against the WSDL Schema. Default is `false`.
	MTOM Enabled	Boolean: WShether MTOM support is enabled. Default is `false`.
REST Server		
	Address	Except for SOAP, this is required because there is no WSDL from which to get the address.
	Logging	Boolean: Enables CXF message logging. **Note:** The default size limit at which messages are truncated in the log is 102400.
	Cloverleaf TrxId Determination	Combo box: PATH, NULL
	Cloverleaf Timeout	Default is 30000.
	Cloverleaf Log Exceptions	Boolean: Should Cloverleaf log exceptions or not. Default is `false`.
	Cloverleaf Attachment Directory	Relative or absolute path to store inbound attachments.
	Cloverleaf Request Forward	When this is selected, the HTTP request headers from the inbound thread are removed to avoid interfering with and overriding the information set by the next outbound thread. This does not happen when their names are listed in "Cloverleaf Forward Headers". Otherwise, the HTTP request headers will interfere with the next outbound thread.

Item	Field	Description
	Cloverleaf Forward Headers	This assists in the selection of the HTTP request header name list from the inbound thread to be forwarded to the next outbound thread. This is a comma or space separated string. For example: "mytestheader1,mytestheader2" or "mytestheader1 mytestheader2". This is only enabled when "Cloverleaf Request Forward" is selected.
Raw Server		
	Port	Port on which to listen.
	Context Path	Path of URL on which to listen. For example, "/raw".
	Cloverleaf TrxId Determination	Combo box: PATH, NULL.
	Cloverleaf Timeout	Default is 30000.
	Cloverleaf Request Forward	When this is selected, the HTTP request headers from the inbound thread are removed to avoid interfering with and overriding the information set by the next outbound thread. This does not happen when their names are listed in "Cloverleaf Forward Headers". Otherwise, the HTTP request headers will interfere with the next outbound thread.
	Cloverleaf Forward Headers	This assists in the selection of the HTTP request header name list from the inbound thread to be forwarded to the next outbound thread. This is a comma or space separated string. For example: "mytestheader1,mytestheader2" or "mytestheader1 mytestheader2". This is only enabled when "Cloverleaf Request Forward" is selected.
	Response Header Overrides	Provides a list of headers to send/override in outbound response.

Item	Field	Description
	Message Validation Check Mode	Boolean to enable detailed checks of the outbound message. It is CPU intensive and should only be enabled in development to check for coding errors, as it slows production systems. This checks the outbound message content against the MESSAGE/PAYLOAD setting. The USERDATA keys look for incorrect keys, a SOAPAction overridden as a header, and other common mistakes. It tries to build a list of the probable mistakes being made in an outbound message and put that list in the process+ log. Default value is `false`. **Note:** All other protocols inherit this from the Bus. A Raw Handler does not have a connection to the Bus. It has this setting separate.
	Enable GZIP Encoding	This enables RawHandler to compress the inbound and outbound messages. This option has an additional **Minimum GZIP Size** condition. Only message sizes greater than this value are compressed. Default value is 4096 bytes.
Engine		
	Host	IP of network interface on which to listen. Blank indicates "all."
	Port	Must be unique from other engines.
	Minimum Threads	Default is 5.
	Maximum Threads	Default is 15.
	TLS Server Parameters	Group of settings for TLS support.
	- Secure Socket Protocol	Combo box - SSL, TLS, TLSv1, TLSv1.1, TLSv1.2. **Note:** To use TLSv1.1 or TLSv1.2, the process must be running within a Java 7 JVM. Cloverleaf 6.1 and above already run Java 7. For earlier releases, see the Cloverleaf Java Driver documentation for information on how to use your own JVM in a Java Driver thread.
	- Keystore Type	JKS, JCEKS, PKCS12
	- Keystore Password	Password for the entire keystore.

Item	Field	Description
	- Keystore Path	Absolute or relative path from the thread's working directory.
	- Keystore Key Password	Password for keys within the keystore.
	- Truststore Type	JKS, JCEKS, PKCS12
	- Truststore Password	Password for the entire truststore.
	- Truststore Path	Absolute or relative path from the thread's working directory.
	- Cipher Suite	A comma-separated list of supported cipher suite names. This cannot be combined with the cipher suites filter.
	- Cipher Suites Filter Include	A comma-separated list of regular expressions matching cipher suite names to include. This cannot be used with the Cipher Suite list.
	- Cipher Suites Filter Exclude	A comma-separated list of regular expressions matching cipher suite names to exclude. This cannot be used with the Cipher Suite list.
	- Client Authentication Required	Boolean: Indicates if mutual authentication is required. This is also know as Client Authentication.
	- Client Authentication Wanted	Boolean: Indicates if mutual authentication is wanted. This is also know as Client Authentication.
Bus This works on clients and servers.		
	WS-Addressing	Only affects SOAP Clients/Servers
	- Allow Duplicates	Boolean - Only enabled if WS-Addressing is enabled
	- Addressing Required	Boolean - Only enabled if WS-Addressing is enabled
	Logging	Enables CXF message logging for all components except Raw Servers. **Note:** The default size limit at which messages are truncated in the log is 102400.

Item	Field	Description
	Message Validation Check Mode	Boolean to enable detailed checks of the outbound message. It is CPU intensive and should only be enabled in development to check for coding errors, as it slows production systems. This checks the outbound message content against the MESSAGE/PAYLOAD setting. The USERDATA keys look for incorrect keys, a SOAPAction overridden as a header, and other mistakes. It tries to build a list of the probable mistakes being made in an outbound message and put that list in the process log. Default value is `false`.

Message validation check mode

Mistakes can be made when composing a CAA-WS message in Cloverleaf. For example, building your `USERDATA` in Tcl with a bad key value or sending a SOAP Envelope to a `PAYLOAD` mode thread. A message validation check mode is available to run checks on your messages. This is configured for all protocol types on the Bus, except Raw servers.

If left blank, then the default is "false." Raw handlers have the same configuration option directly on their configuration screen.

You should be careful to only enable this during development. The validation checks can consume significant CPU resources. After the bugs are removed from the user's code, there is no necessity to run the checks in production.

These checks are performed:

- For a SOAP client, do a `MESSAGE`/`PAYLOAD` check by checking the first XML start element. Check its QName to see if it is in the SOAP namespace with the local name "Envelope."
  - If the first element is a SOAP Envelope and the client is in `PAYLOAD` mode, then it is an error.
  - If the first element is not a SOAP Envelope and the client is in `MESSAGE` mode, then it is an error.
- For a SOAP server, do a `MESSAGE`/`PAYLOAD` check the same as for the SOAP client.
- UserData override checks. These tables show all overrides.

  `USERDATA` is checked for any keys not on this list, or in the nested lists.

  The `REQUIRED CONTENT` column lists the checks on individual keys. Keys that are only valid in certain modes are flagged as an error in other modes.

Client overrides

This table lists the client overrides:

Field	Required content if present	Modes
`httpRequestHeaders`	This contains a map of all the HTTP request headers	ALL
– `*`	HTTP request headers can sometimes have arbitrary names. Others have well-known special meanings. For example, Content-Length, Content-Type, User-Agent, and Host. The header name is the key and the header value is the value in the map. Case does not matter. Special checks: • You should not attempt to override the SOAPAction header here. Check if that has been attempted. • Setting Content-Type fail when sending attachments, as CXF controls the Content-Type header. In this instance, check that is not the case.	
`httpRequestInfo`	This contains a map of fields permitting further overrides regarding the HTTP request.	ALL
– `method`	Must be one of: GET, POST, PUT, HEAD, DELETE, OPTIONS, TRACE, CONNECT.	
-- `requestURL`	Full URL, excluding the query string.	
-- `path`	The portion of the URL after the port. For `/raw/customer`, check that it leads with a slash and does not have a query string. If a `requestURL` is provided, then this field is ignored. In this case warn the user.	
-- `query`	The content here is arbitrary, although one confusion is leading with a question mark. Ensure this is not the case.	

Field	Required content if present	Modes
-- soapAction	Should be a URI, check that it can be parsed as such.	SOAP
-- oneWay	true or false	SOAP REST
-- tls	This contains a map.	
--- certAlias	Get KeyManager from HTTPConduit and verify this is a valid key.	
-- authorization	This contains a map.	
--- user	Can be any string.	
--- password	Can be any string.	
--- type	Basic, Digest, or Negotiate; case insensitive.	
attachments	Map of all attachments to be sent.	SOAP REST
-- * This is an identifier.	Each attachment is itself a nested map of its components where the key is the identifier for the attachment.	
--- xop	true or false	
--- headers	A further nested map containing the headers to override for the attachment.	
--- *	The header names are the keys in this map, with the header values being the values in the map. The same as with httpRequestHeaders. Content-Type header defaults to application/octet-stream if no value is provided.	
--- content	Base64 encode your attachment content to be sent here. This field or contentFile field is required. Check that it is valid base64 content. Ensure contentFile is not also present.	

Field	Required content if present	Modes
--- `contentFile`	File name for a file containing the attachment content. It is not necessary to make a special check if the file exists. This error is noted when it attempts to send the attachment. Ensure content is not also present.	
`wsa`	WS-Addressing information.	SOAP
-- `action`	Check that this can be parsed as URI.	
-- `replyTo`	Check that this can be parsed as URI.	
-- `to`	Check that this can be parsed as URI.	
-- `messageId`	Can be about anything.	
-- `from`	Check that this can be parsed as URI.	
-- `faultTo`	Check that this can be parsed as URI.	
`trxid`	Nothing to check here, so must let it proceed into CL and get an error there.	ALL
`wss`	WS-Security map.	SOAP
`*`	Technically, any value is permitted. At this point all values start with `ws-security`, except for `org.apache.cxf.ws.security.tokenstore.TokenStore`. If these cases are found, then the user is warned if the value found does not match.	
`dispatch`	If there are multiple dispatches configured in the client configuration, then assign a dispatch NAME to this field in USERDATA. This tells the client which dispatch to pick for the message. Additional information is available for `dispatch`. See REST.	SOAP REST

Field	Required content if present	Modes
`webClientFactory`	If there are multiple raw consumers configured in client configuration, then assign a consumer NAME to this field in USERDATA. This tells the client which one to pick for the message.	RAW

Server overrides

This table lists the server overrides:

Field	Required content if present	Modes
`httpResponseCode`	Integer	ALL
`httpResponseHeaders`	This contains a map of all the HTTP response headers.	ALL
-- `*`	HTTP response headers can sometimes have arbitrary names, but others have well-known special meanings. Special checks: Setting Content-Type here fails when sending attachments, as CXF controls the Content-Type header. In this instance, check that is not the case.	
`attachments`	Map of all attachments to be sent.	SOAP REST
-- `*` This is an identifier.	Each attachment is itself a nested map of its components. where the key is the identifier for the attachment. The value you specify here is used as the ID for the attachment and is set as the Content-ID header.	
--- `xop`	`true` or `false`	
--- `headers`	A further nested map containing the headers to override for the attachment.	

Field	Required content if present	Modes
--- *	The header names are the keys in this map, with the header values being the values in the map. This is the same as `httpRequestHeaders`. Content-Type header defaults to `application/octet-stream` if no value is provided here.	
--- `content`	Base64 encode your attachment content to be sent here. This field or `contentFile` field is required. Check that it is valid base64 content. Check that `contentFile` is not also present.	
--- `contentFile`	File name for a file containing the attachment content. It is not necessary to make a special check if the file exists. This error is already noted when it tries to send the attachment. Check content is not also present.	
`wsa`	WS-Addressing information.	SOAP
-- `action`	Check that this can be parsed as URI.	
-- `replyTo`	Check that this can be parsed as URI.	
-- `to`	Check that this can be parsed as URI.	
-- `messageId`	Can be about anything	
-- `from`	Check that this can be parsed as URI.	
-- `faultTo`	Check that this can be parsed as URI.	
-- `relatesTo`	It is rare that this would require to be overridden. Warn the user that overriding this is probably a mistake as the automated handling should put the correct value there.	
`wss`	WS-Security map	SOAP

Field	Required content if present	Modes
*	Technically any value is permitted, but at this point all values start with `ws-security`, except for `org.apache.cxf.ws.security.tokenstore.TokenStore`. If these cases are found, then the user is warned when the value found does not match.	

Web Services consumer wizard

For the Cloverleaf web services adapter, creating a web services consumer is the most common use case. The web services consumer wizard simplifies the configuration of WS consumer protocol threads. Tooltips assist in configuration.

Wizards assist in creating new objects, such as:

- WS-Client protocol thread:
  - SOAP Consumer
  - REST Consumer
  - Conduit
- WS-RawClient protocol thread:
  - RAW Consumer
  - Conduit
- WS-Server:
  - RestProvider
  - SoapProvider
  - RawHandler
  - Engine

Information storage

Extracted schemas are stored in `$HCISITEDIR/formats/xml/`*`package`*.

Installation

The Consumer wizard is installed with Cloverleaf Integration Services and enabled by a license key.

SOAP, REST and RAW basics

SOAP is used to send SOAP 1.1 or SOAP 1.2 messages. There are many possible options and overrides with SOAP messages.

With SOAP, there are two options for a provider to use:

- The `PAYLOAD` option means that the system only deals with the contents of the SOAP body. This is the payload.

- The `MESSAGE` option means that the system works with the entire SOAP envelope. This is the entire message.

REST is used for services that send/receive only XML data. In other contexts, a service is said to be RESTful, regardless of whether the content is XML.

The Raw feature permits a client to act as a basic HTTP client. It sends and receives any sort of data, including multi-part, by constructing the exact message to be sent and specifies any HTTP headers. Similarly, a raw server is an HTTP server with few expectations of content. It can receive any type of data, including multi-part, and send anything back.

User interface

In the Wizard's ws-client, ws-rawclient, ws-server configuration GUIs:

- Configuration objects are shown in a tree view, to reflect their internal relationship.
- The property panel shows all the configurations for a single object.
- Configurations are split into several sections and have their own tabs
- Objects can be created step-by-step with the wizard.

In the server thread, each SOAP/REST provider and raw handler must have a parent engine object. New providers and handlers are located under the corresponding engine node with the same port number. This does not change the XML structure; it only shows the relationships.

For example, selecting the **`java/ws-client`** protocol on the Network Configurator opens the **WS-Client** dialog box. Clicking **New** opens the WS-Client wizard.

Tooltips are available for parameters that might require further explanation, such as **Service** or **Port** in SOAP.

The **Type Selection** page is where you select which consumer to create. Select from **`SOAP Consumer`**, **`REST Consumer`**, or **`Conduit`**.

SOAP Consumer

The SOAP Consumer wizard walks you through selecting a WSDL by:

- Extracting and compiling schemas if necessary.
- Creating all properties to create a SOAP Consumer.
- Automatically adding a SOAP envelope if required.

The conduit is automatically created. Options include:

- Setting security parameters in a single dialog box.
- Setting client parameters, such as proxy, username, and password, on another screen.

REST Consumer

The RESTful/Raw wizard walks you through creating all properties to create a RESTful or RAW Consumer.

Note: The RESTful Client is not used, as XML validation is not necessary for the majority of use cases.

The conduit is automatically created. Options include:

- Setting security parameters in a single dialog box.
- Setting client parameters, such as proxy, username, and password, on another screen.

Building a SOAP Consumer

To create a SOAP Consumer, start by creating a new soap_1 thread.

Then, follow these steps:

1. On the Network Configurator, select `java/ws-client` as the protocol. The WS-Client wizard displays the **Type Selection** page.
2. Select **Create SOAP Consumer**, and click **Next** to walk through the wizard steps.

WS-Server

When a tree node is selected, the configuration panel displays its configuration. The tree view lists the nodes of the existing Engines, Providers and RawHandlers.

There can be multiple Engines, Providers and RawHandlers.

If some Providers/Handlers have the same port number in the address, then these nodes are displayed under the Engine node of the same port number.

Each SOAP/REST Provider and RawHandler must have a parent Engine object.

Editing

Click **New** to open the wizard for adding a new Engine, SoapProvider, RestProvider, or RawHandler.

When you select an Engine node and click **Duplicate**, the Engine node and its sub-nodes are copied as new nodes. They use the same configuration as the selected node. This is enabled only when nodes are selected, but not applicable to the Bus node.

You can drag and drop the SOAP/REST Provider and Raw Handler nodes to put them into other Engine nodes. Their port and address fields are updated.

Delete removes the currently selected node and its sub-nodes from the view.

Note: The key fields of duplicated nodes, such as the **Engine Port** and **Rest Provider** address, are automatically assigned new values to avoid conflicts. For example, adding `1` to the port number or appending `_new` to the address.

XSD WSDL tool

The XSD WSDL Tool is integrated in the IDE as a wizard. The wizard guides you in generating and compiling the XSD file, and generating the WSDL file from the XSD file.

The wizard can also generate the WS-Policy by consuming an existing WSDL, if it is provided.

User interface

The wizard contains several dynamic pages. The usage of "dynamic" indicates providing different arguments. This selects different pages as the starting finishing points for different purposes.

For example, to only add Policy to an existing WSDL file, specify the **WSDL URL/file-path** and specify `Add-WS-Policy` to start. This would only show the steps for Policy Generation.

The **CAA-WS Server and Client thread protocol** dialog box can launch this tool for WSDL generation, Policy generation, and others. The XML Package Manager has several enhancements to take advantage of this tool. In this way, CAA-WS users can finish the task of WSDL file management and XSD compilation.

The first wizard page lists all the available operations according to the user-selected file.

Information storage

The generated files are stored in the host server.

XSD and WSDL file are stored at the site level. You can specify a sub-folder under this path, such as `<%HCI ROOT%>\<%SITEDIR%>\formats\xml\`.

The generated file is based on template files, such as XSD template and WSDL template. When the host server receives the generate request from the IDE, it loads the template file and fills the placeholder in the template. Then, it generates the file.

Selecting a WSDL file as input file

When you select a WSDL file and launch the wizard, you can generate an XSD file according to the WSDL file. You can also add WS-Policy to the WSDL file.

When you launch the wizard, the GUI first gets the WSDL configuration from host server. After this, it completes the components with the WSDL configuration. Then, you can then specify parameters, and then submit the generate request to the host server.

When the host server receives the generate request, it loads the template files and assigns the variables with user-specified parameters. Then, it generates the file and returns the generate details to the client.

Selecting an XSD file as input file

When you select an XSD file and launch the wizard, you can generate another XSD file or generate a WSDL file. This is according to the XSD file.

When you launch the wizard, the GUI first gets the XSD definition from the host server. After this, the GUI completes the components. You can then select types that are defined in the XSD file as input and output elements. After this, you can submit the generate request to the host server.

When the host server receives the generate request, it loads the template files and assigns the variables with user-specified parameters. Then, it generates the file and returns the generate details to the client.

WS-Client and WS-Server nodes

The tree lists nodes for SOAPConsumer, RESTConsumer, and Conduit.

- There is no Engine node. All nodes are on the same level.
- There can be multiple Conduits and REST/SOAP Consumers.

The WS-RawClient tree lists nodes for RawConsumer and Conduit. There can be multiple Conduits and RawConsumers.

This table lists the WS-Client and WS-Server nodes:

Node	Description
WS-Server RestProvider	This node can have multiple providers. If the port number in the address has been changed, then the system checks if there are any Engines with the same port. If there is a match, then the provider is put under the corresponding Engine node.
WS-Server SoapProvider	This node is similar to Rest Provider for Enginws-server soap provider node port change logic. The **Policy Properties** tab is only available when WSDL has policy settings.
WS-Server RawHandler	This node's configuration items are: • **Context Path** • **Cloverleaf TrxID Determination** • **Cloverleaf Timeout** • **Message Validation Check Mode Enabled** • **Response Header** • **Overrides**

Node	Description
WS-Server Engine	**Note:** This is applicable only in WS-Server. Multiple engine nodes can be added to the tree after the port is set. Other providers that have the same port number in **Address** automatically go under it. Any changes to the engine host, port, and TLS setting update the address of other providers that are under it. If the port is not specified in **Address**, then the provider is automatically under the engine. Port 80 is listed for HTTP. Port 443 is listed for HTTPS. **Note:** Server providers do not work for default HTTPS ports. Providers with HTTPS addresses must set the port to include port 443.

WS-Client conduit configuration

Specify an expression to be matched against the various clients that exist in your configuration file.

See Conduit on page 40 for a description of conduit names.

The first page of the **WS-Client Creating Wizard** defines two modes of message delivery:

- Asynchronous
- Synchronous

To configure the WS-Client conduit:

1. Select `ws-client` and click **Properties**.
2. On the **WS Client** dialog box, click **Add** to start the wizard.
3. On the initial wizard page, select **Create Conduit** and click **Next**.
4. Specify the **Name** of the conduit. See Conduit on page 75.
5. If **TLS Secured** is selected, then the next page contains **Secure Socket Protocol**, **Key Store**, and **Trust Store** configuration items. A **Test** button displays when **TLS Secured** is selected.
6. Click **Next** to configure proxy settings.
7. Click **Finish** to complete the configuration.

Use one of these formats for the expression:

- A regular expression to match a URL.

  For example, `http://somehost:port/url*`. This indicates that the conduit can associate with the clients whose address is `http://somehost:port/url`, or a subset of that path.

- `<soap port>.http-conduit`

For example, `{urn:ihe:iti:xds-b:2007}DocumentRegistry_Port_Soap12.http-conduit`. This indicates the conduit associates with the SOAP clients whose port is `{urn:ihe:iti:xds-b:2007}DocumentRegistry_Port_Soap12`.

- `*WebClient.http-conduit`

  The conduit associates with all Raw clients.

- `*.http-conduit`

  The conduit associates with all clients.

Conduit

A conduit is used to set HTTP connection details. For example, a conduit can put TLS on the connection, add HTTP basic authentication, proxy settings, and time-out settings.

The conduit uses a single `name` attribute that specifies an expression to be matched against the various clients that exist in your `config` file. This attribute can have one of these formats:

- A regular expression to match a URL.

  For example: `http://somehost:port/url*`.

  This indicates the conduit can associate with the client whose address is `http://somehost:port/url`, or some subset of that path.

- `<soap port>.http-conduit`

  For example: `{urn:ihe:iti:xds-b:2007}DocumentRegistry_Port_Soap12.http-conduit`.

  This indicates the conduit associates with the SOAP clients whose port is `{urn:ihe:iti:xds-b:2007}DocumentRegistry_Port_Soap12`

- `*WebClient.http-conduit`

  The conduit associates with all Raw clients.

- `*.http-conduit`

  The conduit associates with all clients.

A REST client uses this order until it locates the matched conduit:

1. Locate a conduit whose name is a URL regular expression and matches the address of the REST client.
2. Locate a conduit whose name is `*.http-conduit`.

A SOAP client follows this order until it locates the matched conduit:

1. Locate a conduit whose name is a URL regular expression and matches the address of the SOAP client.
2. Locate a conduit whose name is a Port format and matches the port of the SOAP client.
3. Locate a conduit whose name is `*.http-conduit`.

A Raw client follows this order until it locates the matched conduit:

1. Locate a conduit whose name is a URL regular expression and matches the address of the Raw client.
2. Locate a conduit whose name is `WebClient.http-conduit`.

3 Locate a conduit whose name is `*.http-conduit`.

For access, if the current Consumer URL is HTTPS, then it must have a Conduit that is configured with TLS.

If an existing conduit matches, then it can be directly used. You are not required to create a conduit. Otherwise, a new conduit with TLS must be created to ensure the HTTPS Consumer functions properly.

By default, when creating a new conduit, **Name** is populated with the URL from the Consumer address, with an asterisk at the end. For example, `http://somehost:port/url*`.

WS-Client SOAP Consumer configuration

The first page of the **WS-Client Creating Wizard** defines the two modes of message delivery:

- Asynchronous
- Synchronous

To configure the WS-Client SOAP Consumer:

1 Select **`ws-client`** and click **Properties**.
2 On the **WS Client** dialog box, click **Add** to start the wizard.
3 On the initial wizard page, select **Create SOAP Consumer** and click **Next**.
4 On the next page, configure the **Service**, **Port**, **Service Mode**, and **Address**.
5 To generate an XSD, specify the necessary information and click **Generate and Compile Target XSDs**. Otherwise, click **Next** to skip the generation.
   - All schemas are imported from the WSDL file.
   - **Operation** selects the first item by default. The field **Target XSD filename** is updated according to the operation selection.
   - By default, **Soap Header Preference** selects **Any headers are allowed**.
   - Click **Generate and Compile Target XSDs** to generate and compile the XSD from the current WSDL.
   - XSDs are generated under `$HCISiteDir/formats/xml/{THREADNAME}/` using the names `{OPERATION_NAME}_input.xsd` and `{OPERATION_NAME}_output.xsd`.
   - The Results area shows the output and errors after generating the XSD.
   - Click **Next** to skip the **Generating and Compile Target XSDs** step.
6 The remainder of the pages are used for setting up the Conduits. You can specify whether to **Create New Conduit** or **Do Not Create New Conduit**.See Conduit on page 75. If the URL address of the current Consumer is HTTPS, then **Create New** is selected by default.

   A URL pattern. For example, `http://somehost:port/url*`.

   When a matched conduit is located, the name is displayed in the **Conduit Matched** read-only text field. **Do Not Create New Conduit** is selected by default. You can click **Test** to verify that the current consumer can TLS socket connect with the server.

   The HTTP sites must specify TLS on the conduit. See TLS Secured on the conduit on page 41.
7 The next TLS page is shown only if **TLS Secured** is selected on the previous page.
   - **Service Socket Protocol** is set to TLSv1.2 by default. It contains SSLv3, TLS, TLSv1, TLSv1.1 and TLSv1.2.

- **Common Name Check** is selected by default. This is used for checking whether the DNS name from the server address matches the name from the license.
- All other fields in the above page are empty by default.

8 The next page is used for setting up a Conduit's client settings: Proxy & HTTP Authentication.

- Proxy

  Service and **Port** must be specified if the service is required to be accessed through Proxy.

  Type is `HTTP` by default.
- Proxy HTTP Authentication

  Username and **Password** must be specified if the Proxy authentication is required.

  Authentication Type is `Basic` by default. See SPNEGO.
- HTTP Authentication

  Username and **Password** must be specified if the HTTP authentication is required.

  Authentication Type is `Basic` by default. See SPNEGO.

All other fields are empty by default.

WS-Client REST Consumer configuration

A REST client follows this order until it locates the matched conduit:

1 Locate a conduit whose name is a URL regular expression that matches the address of the REST client.
2 Locate a conduit whose name is `*.http-conduit`.

When finding a matched conduit, the name is displayed in the **Conduit Matched** read-only text field.

Do Not Create New Conduit is selected by default. A **Test** button displays to test if the current consumer can TLS socket connect with the server.

By default, if **Create New Conduit** is selected, then the **Name** is populated with the URL from the Consumer address with an asterisk at the end. For example, `http://somehost:port/url*`.

Note: The HTTPs must specify TLS on the Conduit. See TLS Secured on the conduit on page 41.

The **TLS Secured** default value depends on the **Consumer URL** type.

By default, this is cleared and disabled when the address of the consumer is an HTTP URL.

By default, this is enabled and selected when the address is an HTTPS URL.

WS-Client Creating Wizard

The first page of the WS-Client Creating Wizard defines the two modes of message delivery:

- Asynchronous
- Synchronous

To configure the WS-Client REST Consumer:

1. Select `ws-client` and click **Properties**.
2. On the **WS Client** dialog box, click **Add** to start the wizard.
3. For REST Consumer, only the **Address** is required.
4. The remainder of the pages are used for setting up Conduits. You can specify whether to **Create New Conduit** or **Do Not Create Conduit**. If the URL address of the current Consumer is HTTPS, then **Create New Conduit** is selected by default.

   Conduit Name expression to match the client can be:

   - A URL pattern. For example, `http://somehost:port/url*`.
   - Match only `webClients` (raw). For example `*WebClient.http-conduit`.
   - A specific **Soap Port** name.
   - All clients. For example, `*.http-conduit`.

   If **Create New Conduit** is selected, then **Name** is filled with `{consumer_address_host_ip}.http-conduit` by default.
5. The **TLS** page is shown only if the **TLS Secured** is selected on the previous page.
   - **Service Socket Protocol** is set to TLSv1.2 by default. It contains SSLv3, TLS, TLSv1, TLSv1.1 andTLSv1.2.
   - **Common Name Check** is selected by default. This is used for checking whether the DNS name fromthe server address matches the name from the license.
   - All other fields in the above page are empty by default.
6. The next page is used for setting up the conduit client settings (Proxy & HTTP Authentication).
   - Proxy

     Service and **Port** must be specified if the service is required to be accessed through Proxy.

     Type is HTTP by default.
   - Proxy HTTP Authentication

     Username and **Password** must be specified if the Proxy authentication is required.

     Authentication Type is Basic by default. See SPNEGO on page 80.
   - HTTP Authentication

     Username and **Password** must be specified if HTTP authentication is required.

     Authentication Type is Basic by default. See SPNEGO on page 80.

   All other fields are empty by default.

WS-RawClient wizard flow

Conduit configuration is similar to the WS-Client.

The first page of the **WS-RawClient Creating Wizard** defines the two modes of message delivery:

- Asynchronous
- Synchronous

The current message delivery mode is listed at the bottom of the page.

In the remainder of the wizard:

- The name field is specified with letters only, no spaces. This is optional if there is only one `WebClient Factory` in a configuration file. If there are multiple instances of `WebClient Factory`, then the name fields are required and must be specified in the invocation from Cloverleaf.
- **Address** is required.
- **Default Method** is selected from a list of: **`GET`**, **`POST`**, **`HEAD`**, **`PUT`**, **`DELETE`**, **`OPTION`**, **`TRACE`**, and **`CONNECT`**.
- When **Next** is clicked, you can select from **Create New Conduit** or **Do not Create New Conduit**. These are the same as that for WS-Client REST/SOAP Consume with exceptions.

  A Raw client follows this order until it looks up the matched conduit:

  1. Locate a conduit whose name is a URL regular expression and matches the address of the Raw client.
  2. Locate a conduit whose name is "*WebClient.http-conduit".
  3. Locate a conduit whose name is "*.http-conduit".

SPNEGO

SPNEGO is necessary when using Tcl scripts to communicate with the CIS 6.2 web API end-point to enable/disable some config options.

A SPNEGO **Authentication Type** option is available in the WS Client GUI.

SPNEGO provides authentication mode support in CIS, so that CIS can access web services over RET end point with a logged-on user context.

A **`Negotiate`** option is on the **Authentication Type** list in the CAA-WS WS-Client wizard. Additional options are **`Basic`** and **`Digest`**.

Note: Although the type is Negotiate, it is SPNEGO/kerberos in the background.

Properties for Negotiate include the JAAS krb5 login module configure file and the Kerberos configure file to assign to JVM system properties. The latter can usually be generated by Kerberos tool kits on different platforms.

WS-Client adapters can be used in REST mode to communicate with the internal web service for any data update or insert.

This communication is secured and needs authentication. Because AD (Active Directory) is used-based on user authentication, the WS adapter supports SPNEGO mode to work with AD.

Scheduler node

The WS-Client and WS-Raw Client protocols have a scheduler node. When the node content is populated, the thread functions in a time-event-driven mode.

The protocols function as inbound by querying services in cycling instead of the outbound mode. The outbound mode requires Cloverleaf outbound messages to trigger.

Under the node, you can set up a series of events. When any of the events become active, all consumers that are configured in the thread are triggered.

A thread can also trigger all of its clients for the first time immediately after it is started. This uses the same convention as all other CIS protocols that support scheduling mode.

The scheduler configuration directly loads and saves the configuration in the `SCHEDULE` key of the `NetConfig` file.

On the CAA-WS IDE properties GUI, only one scheduler node is on the left panel of the WS-Client/WS-Raw Client. This cannot be deleted or copied.

Selecting the scheduler node opens the Scheduler configuration panel.

On the event toolbar:

- Click **New** to add an empty row at the end of the event table after the existing events are validated.
- Click **Delete** to remove the selected event row.

Scheduler panel

Event Table entries consist of editable Description and Recurrence columns.

In the Description column, you can specify an event name to be sent to a log file if the event causes an error.

In the Recurrence column, you can specify a time for the event to occur. You can specify a `CRON` expression in the text field. Clicking the detail button open the **Time Properties** dialog, where you can configure expressions.

The **Time Properties** dialog box has these fields:

- Seconds
- Minutes
- Hours
- Days of Month
- Months
- Days of Week

A valid event must have description and recurrence values.

Information storage

The scheduler configuration is saved in the `SCHEDULE` key of the `NetConfig` file as:

```
{ SCHEDULE {
    { EVENT1 {
        { ARGS {} }
        { CRON {0 0 8 * * *} }
        { DESCR {8am every day} }
        { PROCS {} }
        { PROCSCONTROL {} }
    } }
} }
```

When **Scheduler** is selected on the **WS Client**, **Description** and **Recurrence** are populated with the schedule.

Additional configurations

To enable time-event-driven mode after saving scheduler events, add `TIMEMETHOD=doTimeEvent` configurations to *%HCISITEDIR/thread_name*`.ini`.

To disable time-event-driven mode after saving scheduler events, delete `TIMEMETHOD=doTimeEvent` events from *%HCISITEDIR/thread_name*`.ini`.

Java driver protocol notes

There is no new key in *%HCISITEDIR/thread_name*`.ini` because Java Driver has a timed mode..

In the **Java Driver Protocol Properties** dialog box, the time method `doTimeEvent` must be selected. When this is selected, the interval times or advanced scheduling can be set.

Conversely, when `doTimeEvent` is selected, the interval or advanced scheduling must be specified as “Time Method Options”.

These configurations add `TIMEMETHOD=doTimeEvent` in *%HCISITEDIR/thread_name*`.ini` and `QUERY_INTERVAL` or `SCHEDULE` settings in `NetConfig`.

Web Services security

The Security testing and validation tool provides a means to connect to a defined web service provider. It also tests the security configuration without having to run the engine.

This tool can be used for a connectivity test if the web service is configured as a Consumer. It can verify that the Consumer can successfully connect to the provider using the configured security settings.

User interface

The web services security user interface contains a testing dialog box for web services security settings. You can run a validation and connectivity test, and review the returned results.

For a web services Consumer, you can enable TLS and configure it in an HTTP Conduit. After this, the SOAP Consumer, RESTful Consumer, or RAW Consumer can use the TLS configuration. The Consumer must match the name field previously configured in the HTTP Conduit.

The Test Tool verifies that the TLS settings of the HTTP conduit are configured correctly. The test tool attempts to use the configured client to connect to the server side for testing connectivity. It tests the connection that is from the host server to a server side.

For a web services Provider, you can enable TLS and configure it in an engine (Jetty). After it is configured, the SOAP provider, RESTful provider or RAW provider can use the TLS configuration. The Certificate Manager generates, imports, or export keys.

Testing

A **Test** button is on the last page of Consumer wizard if the consumer has a matched conduit with the TLS setting. Clicking Test starts an RMI invocation for testing in the host server.

Note: There is no **Test** button for HTTP addresses.

After the test is started, the Test Tool in the host server connects to the server to verify the connection.

The web service provider provides the certificate as its own when doing the handshake. On the client side, the Test Tool adds this certificate to the trust store that is based on your selection.

A **Confirmation** dialog box opens to verify your action. It contains a list with check boxes, all checked by default, to let you determine which certificates to import. Clicking **OK** imports the selected certificates to the trust store.

The server certificates display as a chain to indicate the relationship. Double-clicking a certificate shows its detail information.

If a provider requires verification of client identity, and the client does not have the certificate,then the testing fails.

Then, the Test Tool informs you if any certificates are missing that are required by the provider. These are the certificates that you must have for exporting the client certificate from the keystore and have the server trust it.

Certificate manager

Certificate Manager can generate certificates into a JKS Keystore. The **File > Issue JKS certificates** option opens a wizard for generating JKS store and certificate.

When you specify user information, validation date, and the key and keystore passwords, Certificate Manager generates a keystore in `$HCIROOT/server/certs/store`.

The **Store** tab lists all keystores in `$HCIROOT/server/certs/store`. The certificates in keystore also display in this list. This contains `Name`, `Expiration date`, and `Issued by` columns.

You can import the client certificates into the key store by right-clicking a keystore to open a file chooser. In the file chooser, you can select a certificate file that you can import into this keystore.

Export the server certificates to a file by right-clicking a certificate to export.

Double-click a certificate to view its information.

Non-mode Cloverleaf disables other functions of Certificate Manager, except `Issue jks keystore` and the **Store** tab.

If Cloverleaf is in non-mode, then you can sign a certificate without a CA certificate (self-sign).

Web services security use case

A user has configured the security entries in a web services connection. The user employs the web services security user interface to verify the connection settings before saving the configuration.

The tool prompts a message that the trust store is incorrectly configured. It is missing a certificate required for the trust chain to function correctly. The user adds the certificate to the trust store and runs the test again, where it reports a success.

Usage scenario

This section describes the different flows that you can use to develop applications that run on CAA-WS.

For a description of the overall functionality of the CAA-WS, see:

- CAA-WS on page 10
- Architecture and flow on page 12

Intended users

For best results, suggested users for this component are:

- Normal users: These are users who follow samples and configuration guidelines and apply business logic in typical system application development methodology. In general, Tcl API in UPoCs is used to employ existing system functionality.
- Power users: These are users who customize processing at the web services protocol level or deal with advanced web services topics, such as WS Security. Power users are comfortable programming in both Tcl and Java, and have a deep understanding of how certain open source Java Web Services technology works. For example, CXF.

Note: All users must be familiar with system implementation methodologies.

Basic flow

When building an application using the bundled Providers (server) or Dispatches (client), the CAA-WS user's main tasks are contained in these steps:

1. Create a Cloverelaf site and add a CAA-WS **`java/ws-*`** protocol thread.
2. Gather the necessary artifacts to configure your service or client.

   These could be keystores/truststores for doing secure communications, WSDL files for SOAP services, URLs for services you must invoke, and so on.
3. Use the **Properties** dialog box to configure your thread with the artifacts that are collected in the previous step.
4. Create the Tcl UPoCs that have the business logic of processing (server) or generating (client) request usage content.

To create Tcl UPoCs in Cloverleaf to process messages using the API, see Application Programming Interface (API).

Alternate flow: Normal users

These procedures illustrate how you can build web services clients or servers using different protocols.

Web Service SOAP client: payload

This is the same as the "Basic flow" topic, except that it creates a web service SOAP client that uses Payload mode.

See Basic flow.

1. Create a Cloverelaf site and add a CAA-WS **`java/ws-*`** protocol thread.
2. Gather the necessary artifacts to configure your service or client. These could be keystores/truststores for doing secure communications, WSDL files for SOAP services, URLs for services you must invoke, and so on.
3. Use the **Properties** dialog box to configure your thread with the artifacts that were collected in the previous step.
4. Create a UPoC that creates the payload of the SOAP message, and another UPoC that processes the payload of the response SOAP message. These payloads do not include the SOAP Envelope or Body elements.

Web Service SOAP client: message

Same as Payload version, but creates a web service SOAP client that uses Message mode.

1. Create a Cloverleaf site and add a CAA-WS **`java/ws-*`** protocol thread.
2. Gather the necessary artifacts to configure your service or client. These could be keystores/truststores for doing secure communications, WSDL files for SOAP services, URLs for services you must invoke, and so on.
3. Configure a SOAP client with Message mode.
4. Create a UPoC that creates the entire SOAP message including the SOAP Envelope, and another UPoC that processes the response expecting a whole SOAP Envelope.

Web Service RESTful client

This is the same as the "Basic flow" topic, except that it creates a web service RESTful Client.

See Basic flow.

1. Select the **java/ws-client** protocol.
2. Configure a RESTful client.
3. Create a UPoC that creates the RESTful message content to be sent by the client. In CXF's view, REST means XML, so this UPoC would be constructing an XML message to send. Create another UPoC to process the XML response message.

Web Service Raw client

This is the same as the "Basic flow" topic, except that it creates a web service Raw HTTP client.

See Basic flow.

1. Select the **java/ws-rawclient** protocol.
2. Configure a raw client.
3. Create a UPoC that creates the HTTP content to be sent by the client. This could be a manually created SOAP message, XML, plain text, an HTML form post, or an image file. It is anything that an HTML client can send. Create another UPoC to process the response, if any.

Web Service SOAP server: payload

This is the same as "Basic flow" topic, except that it creates a web service SOAP server that uses Payload mode.

See Basic flow.

1. Select the **java/ws-server** protocol.
2. Create or locate the WSDL for the service to provide. This could be in the form of an HTTP URL, or you could copy the WSDL, including XML schema files, to the disk.
3. Configure a SOAP server, with Payload mode.
4. Create a UPoC that processes the payload of the incoming SOAP message, and creates a response payload. These payloads do not include the SOAP envelope or body elements.

Web Service SOAP server: message

This is the same as "Web Service SOAP server: payload," except that it creates a web service SOAP server that uses Message mode.

See Web Service SOAP server: payload.

1. Create a Cloverelaf site and add a CAA-WS **java/ws-*** protocol thread.

2. Gather the necessary artifacts to configure your service or client. These could be keystores/truststores for doing secure communications, WSDL files for SOAP services, URLs for services you must invoke, and so on.
3. Configure a SOAP server, with Message mode.
4. Create a UPoC that processes the incoming SOAP message including the SOAP envelope, and generates a response payload including the SOAP envelope.

Web Service RESTful server

This is the same as "Basic flow" topic, except that it creates a web service RESTful server.

See Basic flow.

1. Select the **`java/ws-server`** protocol.
2. Configure a RESTful server.
3. Create a UPoC that processes incoming RESTful requests. In CXF's view, REST means XML, so this UPoC receives an XML message and creates an XML message in response.

Web Service Raw server

This is the same as the "Basic Flow" topic, except that it creates a web service raw HTTP server.

See Basic flow.

1. Select the **`java/ws-server`** protocol.
2. Configure a raw server.
3. Create a UPoC that processes incoming HTTP requests. This could be a SOAP message, XML, plain text, an HTML form post, or an image file. The raw server can receive anything that an HTML Client can send. It sends back a response, depending on the nature of the service. As with the client, it could be almost anything.

CAA-WS sample sites

The sample sites are “best practice” tools that are self-contained units. These assist CAA-WS users in understanding how to employ various parts of the functionality. This section describes the details of these sites and how they are configured.

Review the sample’s configuration in the **NetConfig Properties** dialog box and the XML files that are created for them in the `javadriver` subdirectory. Then, review the configuration in the NetConfig file, and any relevant Tcl procs. This, combined with running the samples and observing the logs, provides a basis for starting projects, testing, and debugging.

Samples are divided into two system sites:

- The primary sample site is `ws_samples`. This contains basic examples of SOAP, REST, and Raw Clients and Servers.
- The secondary sample site is `ws_adv_samples`, which includes:
  - In-depth samples requiring longer study to see all the details involved.
  - Attachment processing.
  - WS-Security usage.
  - A pass-through Provider/client pair as an intermediary where a web services request can be intercepted, modified, and then passed on to the real Provider endpoint.
- FHIR (Fast Healthcare Interoperability Resources) uses JSON/XML RESTful for an integration/query/response to mobile.

ws_samples

The ws_samples site contains basic usage patterns that illustrate the main functionality of CAA-WS.

In this layout:

- Providers are threads accepting requests, and are on the left.
- Clients are threads sending requests, are on the right side.

By default, the sample clients connect to the sample Providers. The Providers can accept messages from any client and the clients can be configured to connect to any Provider.

REST

REST is used for services that send/receive only XML data. In other contexts, a service is said to be RESTful, regardless of whether the content is XML. CXF uses REST to mean XML data, and so it also applies to WS Adaptor.

A request can be a basic HTTP `GET` that does not send any XML request, or a POST and sent XML.

In this sample, HTTP `GET` is used to illustrate this concept. POST is the default, so if you send an XML message, it is POST-ed.

Provider:

In the `RestProvider` thread's **Inbound** tab, the `dumpMsg` proc is frequently used in the samples. This dumps the message and then continues the message.

The `bounceREST` proc does the work to create a response and OVER that message.

This block of code in that proc generates the reply content:

```
if {[regexp {customers$} $requestPath]} {
     set cont "<customers><customer id='123'/><customer id='234'/></customers>"
} elseif {[regexp {customer$} $requestPath]} {
    set cont "<customer id='123'><name>joe blow</name><address>joe's house</address></customer>"
} else {
     set cont "<unrecognizedVerbInPath/>"
}
```

This looks in the request path to see if it ends in `customers`. For example, if you called `http://localhost:9001/RESTTest/customers`, the request path is `/RESTTest/customers`.

- If so, then it returns a list of customers.
- If not, then it sees if it ends in `customer` (no "`s`") and returns a specific customer entry.

  In actual scenarios, you search the query string for the customer ID and return that.
- If that does not match, then it returns an `error` element.

The rest of the Tcl proc handles the details of working with the messages. The Provider can be invoked by the sample client or by a web browser that does HTTP `GET` operations when you specify a URL. In this way:

- `http://localhost:9001/RESTTest/customers` returns the list of customers.
- `http://localhost:9001/RESTTest/customer?id=123` returns the single customer entry.

The use of a query string is not considered part of the path. The path interpretation works as shown above in the Tcl because the path is `/RESTTest/customer`.

For the client, the `restClientFile` thread is a basic Fileset-Local protocol thread. This watches a directory for a file to show up and then sends the file to the `RestClient` thread.

The `updateRESTClientMessage` Tcl proc sets the `USERDATA` to set the outbound HTTP method to GET instead of the default `POST`, and continues the message.

The update looks similar to this:

```
# update the user data to make it a GET instead of the default POST
     set userData {{dispatch restCustomerClientDispatch} {httpRequestInfo {{method GET}}}}
```

This also shows how to set the name of the Dispatch in case you had multiple Dispatch entries in the configuration file. The REST sample has only one, so this is only for demonstration, and is not strictly necessary.

The `RestClient` dumps the message before it goes outbound to the WS adaptor. Then it dumps the message that is sent to the system as a reply.

SOAP

SOAP is used to send SOAP 1.1 or SOAP 1.2 messages. There are many possible options and overrides with SOAP messages.

With SOAP, there are these options for a Provider to use:

- The `PAYLOAD` option means that the system only deals with the contents of the SOAP Body. This is the payload.
- The `MESSAGE` option means that the system works with the entire SOAP Envelope. This is the whole message.

SOAP Provider (MESSAGE mode)

The `SOAPProvider_Registry` thread's inbound and reply messages are dumped first and last to see what is received from and sent to the system. The `bounceSOAPRegistry` Tcl proc does the work to process the request and generate a reply message.

The request is a query but for this sample implementation the query content is ignored. An empty response that says "`no documents were found`" is sent back as the reply.

The Tcl proc has many lines commented out that show various optional overrides for a SOAP reply.

The reply message is created similar to this:

```
set cont {<?xml version='1.0' encoding='UTF-8'?><soapenv:Envelope
xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope">
<soapenv:Header/><soapenv:Body><query:AdhocQueryResponse xmlns:query="urn:oasis:names:tc:ebxml-
regrep:xsd:query:3.0"
status="urn:oasis:names:tc:ebxml-regrep:ResponseStatusType:Success"><rim:RegistryObjectList
xmlns:rim="urn:oasis:names:tc:ebxml-regrep:xsd:rim:3.0" /></query:AdhocQueryRe
sponse></soapenv:Body></soapenv:Envelope>}
```

In this example, the whole SOAP Envelope is created as the message in this `MESSAGE` mode.

Note: This is configured to listen on `http://localhost:9003/xdsregistryb`. This is the same as the PAYLOAD mode configuration. See SOAP Provider (PAYLOAD mode). Therefore, only one of these two can be run at a time because they listen to the same endpoint.

Editing WS-Addressing

For MESSAGE mode, there are two choice for editing WS-Addressing headers in a message.

- Enable the addressing property in CXF configuration and override different sections in USERDATA.
- Disable addressing property and add them directly into SOAP headers.

Note: Do not add WS-Addressing headers directly in a message when the addressing property is enabled. Instead, override in USERDATA. Otherwise, CXF appends its default values to your headers.

SOAP Provider (PAYLOAD mode)

In the SOAPProviderPayload_Registry thread's **Inbound** tab, the bounceSOAPPayloadRegistry Tcl proc ignores the actual request message. It sends back a canned response. In this case, the canned response has actual documents listed.

The Tcl proc creates the response message similar to this:

```
set cont {<query:AdhocQueryResponse ...
```

Instead of the whole SOAP envelope, it has the contents that go inside the SOAP body.

If SOAP headers are not required, then other than items such as WS-Addressing PAYLOAD mode are used to deal with basic messages instead of MESSAGE mode.

It is best to use PAYLOAD mode unless you require access to the SOAP headers in the system.

Note: This is configured to listen on http://localhost:9003/xdsregistryb which is the same as for the MESSAGE mode configuration. Oonly one of these two can be run at a time because they listen to the same endpoint.

SOAP Client

The SOAP client is configured in PAYLOAD mode. The messages do not deal with a SOAP envelope or body.

The registryClientFile thread is a Fileset-Local protocol to get messages from the file system with which to test. It watches the request directory for files and sends the contents to the RegistryClient thread, putting the responses in the response directory. To try this, copy the registryQuery.xml file into the request directory to see it run.

The RegistryClient thread dumps the message before it sends to the WS adaptor and dumps the response message. There are no other Tcl procs required, as nothing is overridden from the defaults.

This way, the message alone is all that is required to send it. The response is sent back to the registryFile Client thread and written to the response folder.

SOAP clients should not require any Tcl procs to override defaults after you have the payload to send and have configured the XML configuration files.

Asynchronous SOAP Client

The asynchronous SOAP client has the same business logic as the SOAP Client, but functions in asynchronous mode.

To adapt the mode, `asyncRegiClientFile` uses newline as the message separator for the inbound files. With this, you can feed many messages by a file. Route replies to only the original source are turned off in the process configuration.

To check efficiency difference, use `registryQuery.50.xml` in the same manner as `registryQuery.xml`.

The `registryQuery.50.xml` file contains 50 messages.

Raw

With the Raw feature, a client acts as a basic HTTP client. It sends and receives any type of data, including multi-part, by constructing the exact message to be sent and specifying any HTTP headers.

Similarly, a Raw server is an HTTP server with few expectations of content. It can receive any type of data, including multi-part, and send anything back.

Provider (Handler)

In Raw mode, the Provider side is implemented as a Jetty handler object, not a CXF Provider. This affects the configuration and makes it different from the `SOAP`/`REST` configuration entries.

CXF is responsible for starting Jetty port instances. The configuration requires a workaround of creating a `REST` endpoint on the same port as the Raw handler. This gets CXF to start that port so the handler can take effect. The sample configuration files for the Raw server do these tasks and are a place to learn how it works.

`RawHandler` uses the `dumpMsg` proc to show what is on the inbound message and what is sent back as a reply. The `bounceRaw` proc is used to look at the request URL and create a response. This is the most important part of the proc:

```
# get the user data
set userData [msgmetaget $mh USERDATA]
# get the request info
set requestInfo [keylget userData httpRequestInfo]
#puts "request info: $requestInfo"
# get the request path
set requestPath [keylget requestInfo "path"]
#puts "path is: $requestPath"
# create reply message
set cont ""
set httpResponseHeaders {}
# read the request path and operate accordingly
if {[regexp {customers$} $requestPath]} {
     set cont "<customers><customer id='123'/><customer id='234'/></customers>"
     keylset httpResponseHeaders Content-Type {text/xml;charset=UTF-8}
} elseif {[regexp {customer$} $requestPath]} {
    set cont "<customer id='123'><name>joe blow</name><address>joe's house</address></customer>"
```

```
     keylset httpResponseHeaders Content-Type {text/xml;charset=UTF-8}
} else {
    set cont "<html><head><title>hello!</title></head><body>Hello from Cloverleaf!</body></html>"

     # default content type is text/html so no requirement to set it
}
```

This is similar to the REST provider section. See REST. The Raw handler implements the same functionality as the REST Provider, but does it in the Raw HTTP manner. For example, it specifically sets the response Content-Type header to text/XML, whereas in REST this is assumed. Additionally, if it does not find anything recognizable in the request path, it returns an HTML page instead.

This Raw handler is tested from your browser and from the Raw client. For examples, you can try these URLs:

- http://localhost:9005/raw/customers
- http://localhost:9005/raw/customer?id=123
- http://localhost:9005/raw/somethingelse

These URLs trigger different areas of functionality.

Raw Client

The rawClientFile is a Fileset-local protocol thread which watches the request directory for messages to send on to the RawClient thread. It then writes replies into the response directory. The updateRawClientMessage Tcl proc does this:

```
# update the user data to make it a GET instead of the default POST
set userData {{httpRequestInfo {{method GET}}}}
```

Instead of sending the message using the default POST, the message content is ignored and it sends an HTTP GET. There is also commented out code that shows how to use USERDATA to override some other items as well. This type of code is good to review to understand how items can be overridden.

The RawClient dumps the outbound message as it is leaving the system, and the reply message as it returns. Everything else is already configured in the configuration files. The default HTTP POST is overridden to GET by the Tcl proc in the rawClientFile thread.

You can change the URL in different ways. One way is to change it from the configuration file to get a different response from the Raw handler. Another way is to update the updateRawClientMessage Tcl proc to override the URL to get another response.

Asynchronous RAW Client

The asynchronous RAW Client has the same business logic as the RAW Client, but functions in asynchronous mode.

To adapt the mode, `asyncRawClientFile` uses newline as the message separator for the inbound files. With this,you can feed many messages by a file. Route replies to only the original source are turned off in the process configuration.

To check the efficiency difference, place `test.100.dat` (provided) in the request directory. This triggers the Client 100 times. Then, check the efficiency difference.

ws_more_samples

`ws_more_samples` demonstrates these real-life configurations and overrides scenarios:

- `web_form_*` threads show a client connection sending HTML form values including an HL7 message as an HTTP POST web form. There is an HTML web form included in the BOX's data directory. This can be used to post the same data to the server thread. This is a real-life example where a healthcare facility is sending VXU messages to an Immunization system over HTTPS using a web form.
- `path_trxid_client_vxu` and `path_trxid_client_mdm` are two clients sending raw HTTP data using URLs hosted by a server thread. The server thread parses the URL that is used for trxid routing. This is also a real-life scenario where the user is sending MDM messages to Cloverleaf over web services.
- `qosfilter_in_rawhandler` is a server with a Jetty built-in QOSfilter and `RequestLogHandler` enabled. This sample shows the possibilities of a raw handler to take advantage of Jetty-provided utilities in serving requests. Because the GUI support is not completed at the moment, users may see only limited information. Users who have tech backgrounds and are familiar with Spring XML configurations can read `application Context_qosfilter_in_raw_handler.xml` to understand how everything works. This is under the site's `javadriver` directory . Several things to highlight here are:
  - `com.infor.cloverleaf.gjdws.servlets.BypassServletHandler` and `com.infor.cloverleaf.gjd ws.servlets.BypassServlet` are two placeholders inside CAAWS. They ensure the defined filter chain is called by Jetty. `BypassServlet` does nothing in its `service()` function. The the only thing `By passServletHandler` does is to ensure requests flow to the raw handler after they have been handled by Jetty's `servletHandler`.
  - `BypassServletHandler` extends `org.eclipse.jetty.servlet.ServletHandler`. Users are still allowed to configure all bean properties on it including the context path. This decides how many paths the handler works.
  - `FilterHolder` is a class that Jetty uses to wrap filters inside its `servlet` context. This is capable of holding a filter bean directly or holding a filter class name and a map for the class object's initial parameters.
  - Most of Jetty-provided handlers are subclasses of `org.eclipse.jetty.server.handler.HandlerWrapper`. These have a property named `handler`. Users can assign to it a handler bean, and one-by-one these handlers form a handler stack. In the sample, it can be found that `RawHandler` inside `RequestLogHandler` inside `ContextHandler` forms a stack of this kind.

ws_adv_samples

This site demonstrates these more advanced complex configuration and override scenarios:

- Various WS-Security items in the `SignEnc` threads. For example, `UsernameToken`, signing, and encryption.
- How to handle attachments. `PassThrough` also does attachments, but the attachment threads only do attachments without the complexity of an intermediary.

Signing/Encryption

These samples demonstrate how to manually configure WS-Security settings on Client/server sides, using Username Tokens, signing, and encryption. CXF uses Apache's WSS4J project to handle the WS-Security. Understanding how WS-Security is configured is a combination of understanding how CXF calls it and what options WSS4J provides.

A basic way to set up WS-Security is to use the WS-Policy configuration settings.

WS-Security is a complex subject. You are expected to understand the WS-Security concepts underneath this to a reasonable level. This topic shows the configuration settings used and how those settings were determined.

The system configuration is similar to other samples. The Provider has a Tcl proc to send back a canned response. Both the Client and Provider have `dumpMsg` Tcl procs to show the message content during various points.

CXF has many WS-Security samples illustrating various concepts. This sample takes the `sign_enc` sample directory from CXF and includes it in the working directory for the threads.

In the CXF sample, the configuration is in Java code to set up the WSS4J classes into a CXF interceptor. This sample converts that Java code into a CXF XML configuration. This is completed by reading the CXF documentation and WSS4J documentation and converting the Java code to Spring XML.

The sample uses a basic “hello” type of WSDL to keep the focus on the WS-Security and not on the web service being protected.

Provider

In the CXF sample, `Server.java` has relevant Java code which configures how WSS4J handles the inbound message.

Basically the idea is to create a `WSS4JInInterceptor` class and set those properties there. This is accomplished in a Spring configuration file.

In the `applicationContext_SignEncProvider.xml` configuration file, this is mirrored with this XML.

This does the same thing as the Java code. It creates a `WSS4JInInterceptor` class by passing a properties map that exactly mirrors the map that was created in Java. It then adds that class to the list of `inInterceptors`.

The same paths and class names that work in Java also work in the Spring configuration file. This is because the path `sign_enc/build/classes` is included in the GJD thread's `classpath` .

The `outInterceptors` are configured similarly by converting the `outInterceptors` from the Java code to the same Spring configuration file.

Client

Client is the same as Provider. It uses a basic WSDL to create a Client configuration and converting the code from `client.java` into Spring configuration format. There is a Fileset-Local thread to handle sending the message content and writing the response. This is similar to the other samples.

Understanding the CXF/WSS4J Options

Using WS-Security is not difficult, but does require study to understand the interplay between CXF and WSS4J.

You can start at CXF's page on WS-Security at http://cxf.apache.org/docs/ws-security.html. This has a useful overview on encryption and signing followed by information on how WSS4J fits in with Interceptors. It also includes examples of how to do various things in CXF. Unfortunately, a large amount of it is in Java, not in configuration file format, but the conversion is not difficult.

All of the keys/values that are in the parameter map to the `WSS4JInInterceptor` are described in the WSS4J documentation at http://ws.apache.org/wss4j/config.html.

These two sources, plus the CXF samples on WS-Security, are the most helpful. CAA-WS does no specialization of WS-Security versus plain CXF. Generating these configuration files is completed by reading these pages and reading the CXF samples. It is best for users who must do more than this sample shows, to download the CXF 2.7.8 source code. Review the WS-Security samples for ideas.

Attachment

The attachment sample uses a basic WSDL to demonstrate both sending attachments and sending a one-way Web Service request. There is no response message other than `HTTP 202 Accepted`. This is the HTTP protocol method. No response message comes back to the system thread.

Provider

The Provider implements the basic WSDL.

The GJD thread forwards the untouched message to a file thread to write the attachment to a file. The file thread calls the `handleAttachmentProvider` Tcl proc.

The purpose of this code is to get the base64 data for the first attachment and decode that base64. It then changes the message content to be the content of the attachment. This way the file thread writes out the attachment contents to a file.

Client

The Client side invokes the WSDL implemented by the Provider. There is a Fileset-Local thread which monitors the request directory for a message to send. When it finds a message, it calls the Tcl proc `addRequestAttachment`. This adds an attachment to the message.

This reads the file `attachmentSource` in the process directory to determine the contents of the attachment. You can put the contents of various file types in there. For example, text files, images, and so on. Then, it tells

CAA-WS to send it by XOP, as MTOM is enabled in the configuration file. After this, it sets the content ID to the same XOP ID as that specified in the test message file `myattachment`.

This also tells CAA-WS that this is a one-way message. This cannot be completed in the configuration file because there could be multiple operations supported by the configuration. Because not all of them might be request/response or one-way, this flag must be set in `USERDATA`. In this way, CAA-WS sends the message properly using the one-way method.

FHIR

Fast Healthcare Interoperability Resources (FHIR) uses JSON/XML RESTful for an integration/query/response to mobile.

See http://www.hl7.org/implement/standards/fhir/.

The Cloverleaf FHIR BOX contains working examples and documentation for connecting to a FHIR server. It successfully exchanges patient demographics as defined in the HL7 FHIR Connectathon Track 1. Track 1 is for those new to FHIR.

See http://wiki.hl7.org/index.php?title=FHIR_Connectathon_7.

Track 1 includes information on:

- Registering a patient
- Updating a patient
- Retrieving patient history
- Searching patients by name

CAAWS sample site: FHIR_example.box

The sample site shows connections to the FHIR server with the XML and JSON format message.

These FHIR formats are supported:

- XML

  FHIR XML files are compiled and are included in the site. Any xlate files without JSON in the file name uses XML as the destination format.

  CIS provides pre-compiled FHIR XML files under the root. For any version earlier than CIS 6.2, you can use the files in the sample site.

- JSON

  CIS 6.2 provides JSON DSTU2 schema files. The sample site uses the root-level JSON schema for translation. Any xlate files with JSON in the file name use JSON as the destination format.

  Because JSON is supported from 6.2, the JSON format does not work in versions earlier than 6.2.

Note: The current BOX samples are with version 2. The BOX is as-is. Refer to the FHIR Bridge for more comprehensive FHIR enablement tools.

oauth2_sample

This sample BOX demonstrates several configurations that are possible with CAAWS where the adaptors access resources behind OAuth2 Authorization. This also demonstrates how OAuth2 Authorization is used to protect resources/providers that are set up by the adaptors.

The CAA-WS GUI has out-of-box OAuth2 support on the client-side. This is similar to Basic HTTP Authentication support. Some topics are not covered. Other topics are in the experimental stage.

To use this feature, advanced users should be familiar with the OAuth2 framework and what the CXF framework can do.

For additional information, go to:

- OAuth 2.0 Authorization Framework
  https://datatracker.ietf.org/doc/html/rfc6749
- CXF JAXRS OAuth2 overview
  https://cwiki.apache.org/confluence/display/CXF20DOC/JAX-RS+OAuth2
- CXF OAuth2 package source repository
  https://github.com/apache/cxf/tree/cfe9f430ee617552eb743140cb78cc5df4c4eb83/rt/rs/security/oauth-parent/oauth2/src/main/java/org/apache/cxf/rs/security/oauth2

oauth2_sample

This process uses a TCL solution for clients that require OAuth2 connections.

Connections to the authorization server and the resource server are configured in the WS client. TCL scripts generate access token requests based on the resource server response and then requests the authorization client to send it for the token.

After the token is fetched, it is saved locally for the resource requests.

oauthClient

This process uses the Cloverleaf built-in WS client solution for OAuth2 connections.

In the conduit configuration for the client, the OAuth2 type authorization is chosen. The ensuing grant type is switched to the client credentials grant type. In this way, only "client ID" and "secret" are required to identify the client.

An access token service URI is necessary for the client to locate the authorization server. The scope in this sample is set in the extra parameters. This can also be set in the grant.

With this information, the client, although different from the client in `oauth2_sample`, has only one defined consumer. This automatically acquires the access token when constructing the connection and refreshes the token when it is expired or about to expire.

oauthResource (Experimental)

This WS server is the counterpart of the oauthClient. Because all CAAWS OAuth2 server-side supports are still in the experimental stage, there is no front-end GUI.

This sample is for advanced users to read its application context XML. Additional steps can then be taken to use CXF JAXRS features inside CAAWS, especially for the OAuth2 aspects.

The server uses a `OAuthRequestFilter` CXF filter to protect its resource.

A CAAWS raw handler can take advantage of CXF JAXRS filters when there is a defined `<jaxrs:server/>` in the context. This context must serve the same path and use a `ClDefaultRsResource` resource bean.

The critical property of `OAuthRequestFilter` is the `tokenValidator`. The rest are mostly checkpoints under the OAuth2 framework.

In this sample, the validator is an `AccessTokenIntrospectionClient`. This comes with a common web client to the authorization server. When a client request with an OAuth2 token is sent to the server, the filter passes the token to the validator. Then, the validator, `AccessTokenIntrospectionClient`, forwards the token to the authorization server and makes an introspection invocation for validation. If the validation fails, then the request is denied.

`AccessTokenIntrospectionClient` is not the only choice from CXF. Other `AccessTokenValidator`s include `AccessTokenValidatorClient`, `HawkAccessTokenValidator`, `JwtAccessTokenValidator` and others.

To make use of these validators in the configuration, the beans related to `AccessTokenIntrospectionClient` in the sample are replaced with other validator beans.

oauthCodeResource (Experimental)

This is another experimental server. The first time this is accessed from a browser, the user is redirected to Google's OAuth2 service API to get an authorization code.

When the authorization server redirects the user back to `oauthCodeResource` with a code, the server exchanges the code for an access token. It then associates it with the original request and sends them together into Cloverleaf.

The majority of the work is accomplished by CXF `ClientCodeRequestFilter`. For details, go to:

https://cwiki.apache.org/confluence/display/CXF20DOC/JAX-RS+OAuth2#JAXRSOAuth2-OAuth2clientapplicationswithcode-grantfilters

For the source code, go to:

https://github.com/apache/cxf/blob/master/rt/rs/security/oauth-parent/oauth2/src/main/java/org/apache/cxf/rs/security/oauth2/client/ClientCodeRequestFilter.java

In the sample, `CAAClientCodeRequestFilter` is an extension that can handle any issues found during tests.

To use the filter:

1. Set up `<jaxrs:server/>` for a raw handler to use filters.
2. Insert the filter into the server together with `ClientTokenContextProvider`. This ensures the client token context is available to the Infor code.
3. Ensure the filter has all necessary information during the authorization code and the access code exchange. For example, `authorizationServiceUri`, `accessTokenServiceClient`, and so on.
4. The jetty engine must be session-enabled because the client token requires local session storage. For example,`<httpj:sessionSupport>true</httpj:sessionSupport>`.
5. Set up `HTTPAuthenticationHandler`, as the filter requires user principle.

samlSSOResource(Experimental)

This is an example in the `oauth2_sample` site. This is a SAML SSO (Single Sign On) server sample in addition to the OAuth2 sample. SAML and some OAuth2 types are comparable to some of the SSO parts and can be used in mixed mode. This site also contains SAML and OAuth2 types.

Key components in this sample are `org.apache.cxf.rs.security.saml.sso.SamlPostBindingFilter` and `org.apache.cxf.rs.security.saml.sso.RequestAssertionConsumerService`. Details about these components and related concepts are located at https://cwiki.apache.org/confluence/display/CXF20DOC/SAML+Web+SSO.

Similar `crypto` settings are found between this setting and the one inside `ws_adv_samples`.

The difference between SAML SSO and OAuth2 is when applying the `POST` style. SAML SSO relies on browser-based HTML form forward. This is one reason why OAuth2 is preferred in applications.

In the link above, a JSP file is set up as the view provider of `SamlPostBindingFilter`.

For CAAWS, in most instances a standard servlet context is not completely constructed. A SAML request form is built by `BoringSAMLRequestBodyWriter`.

A similar HTML response is generated in the body with SAML data. This is similar to the example JSP in CXF documents. This response is suitable in most SAML SSO cases. In special cases, you can develop your own view provider.

For files under the `java_uccs` site, you can index the directory as a resource directory on the `classpath` of the Java Driver protocol.

Most of these files are referenced in the SAML SSO sample's Spring application context, except for `mymeta1.xml`. This file contains the same metadata as that uploaded to `samltest.id`. In this way, the test authorization server trusts this sample.

HL7 FHIR requirements and tools

Most healthcare application vendors support the HL7 2.x standards. With "meaningful use" and other initiatives, HL7 v3.x is becoming more familiar to IT staff. Both of these standards provide much in the way of functionality for interoperability and application interfacing, although both have significant limitations.

HL7 FHIR is developed to overcome some of the limitations that are imposed by the previous HL7 standards. It can employ XML, JSON, and web service architecture.

Cloverleaf can bridge the gap in the healthcare information environment between HL7 v2.x, v3.x and the emerging HL7 FHIR standard. This helps to create efficient interoperability between the standards and applications supporting different standards.

This topic describes the methods used to create Cloverleaf interfaces to work with HL7 FHIR in XML format and JSON format. Starting in CIS 6.2, there is a “Cloverleaf FHIR examples BOX” with translation examples for HL7 v2.x to HL7 FHIR XML/JSON. This is located in the `CAA/ws/samples` directory of Cloverleaf Integration Services.

These interface examples are provided in the BOX:

- An “HTTP ReST Patient Resource Conditional Update/Create (HTTP PUT)” example utilizes a public FHIR server at UHN. The translation for this interface results in a single Patient Resource being updated or created on this public test FHIR server.
- An “HTTP ReST FHIR Transaction Bundle” contains a patient resource and an encounter resource. This can create or update a patient and encounter on a FHIR server, if a match of one or none is obtained. The translation for this interface uses the Include translation operation in Cloverleaf Integration Services.

Both of these examples read in a sample HL7 V2.3 Admit/Visit Notify (ADT^A01) message and translates this to HL7 FHIR resources in XML. These sample translations are developed primarily for the purpose of demonstrating functionality and are not for production use.

Requirements

- Cloverleaf 6.2 or later
- CAA-WS 2.0.1 or later
- Outbound HTTP access to the internet
- Knowledge of HL7 v2.x, XML, and web services

Open source tools

- soapUI: http://www.soapui.org/
- Oxygen XML: http://www.oxygenxml.com/
- XML Spy: Not open source

- FHIR standard: http://www.hl7.org/fhir

Notes

- Regarding the FHIR patient conditional create/update and the conditional create/update for patient and encounter in the FHIR bundle. These require a match of one or none on the FHIR server. These are based on identifiers in the sample data. A match of more than one results in an error reported by the FHIR server.
- The FHIR server used in the FHIR sample BOX is a public test server. Therefore, patient identifiers and encounter identifiers can exist in duplicate due to others testing against the same server. This could lead to errors being reported on more than one match.
- Similarly, on the public servers, data previously posted can be wiped from the server without notice or warning. Additional data can also be bulk-loaded overwriting previous tests.
- A solution to an identifier that return a multiple match error is to change the identifiers in the sample data file and try again.
- The FHIR BOX does not contain any updates for FHIR STU-3 examples. The current BOX examples are with version 2. This remains as-is. You should use the FHIR Bridge for more comprehensive FHIR enablement tools.

Deploying Cloverleaf FHIR examples BOX

This procedure describes how to deploy the FHIR examples BOX, including post-deployment manual steps.

1 Deploy the FHIR examples BOX.
   a Verify that the `FHIR_example.box` file exists in the Cloverleaf root directory at `$HCIROOT/CAA/ws/samples`.
   b To deploy the BOX, open the Cloverleaf client GUI and select the site where you deploy the BOX.
   c Open the BOX Manager from the Tools menu and select **BOX > Import**.
   d On the **Import BOX** dialog box, click **Browse** and double-click the `box` directory.
   e Select `FHIR_example.box`.
   f In the BOX Manager, right-click `FHIR_example` and select **Deploy**.
   g In the **BOX Summary & Resources** dialog box, click **Next**.
   h Click **Next**.
   i On the **Confirm Environment Configurations** dialog box, click **Finish**.
   If an information dialog box opens, then it is a reminder to compile the FHIR XML schema. Click **OK**.

2 Compile the FHIR XML format. This is `fhir-single.xsd`. Included in the BOX is the FHIR schema for DSTU2 (Draft Standard for Trial Use v2).

   The FHIR standard is still under development, so stable releases are called “Standard for Trial Use” followed by a version number. The Draft designation has been dropped for future releases starting with STU3.

   Note: The included translations might not work without modification under the new release. CIS 6.2 and later versions provide FHIR XML and JSON format for DSTU2.

   - When you deploy the BOX in CIS 6.2 and later versions, then the root-level format can be directly used.

- When you deploy the BOX in earlier versions, then you should compile the XML format. The JSON format cannot be used.

There are many FHIR XML `xsd` schema files when you unzip the file http://www.hl7.org/fhir/fhir-codegen-xsd.zip. In this guide, only `FHIR-single.xsd` is compiled. This contains all schema code necessary for all defined FHIR Resources. You can select which FHIR resource to use in translation configuration.

To compile the FHIR XML Schema `xsd` file:

a Open the XML Package Manager from the Tools menu and expand **xml** and then expand **FHIR_DSTU2_201510**.

b Highlight and right-click **fhir-single.xsd** and select **Compile**. This is the only schema that must be compiled.

c Click **OK**, and close XML Package Manager.

3 Run `mktclindex`:

a In the Script Editor, open the `XltStrFhirFormatDateTime.tcl` file.

b Save the file to recreate the `tclindex`.

Cloverleaf BOX contents

Threads in the NetConfig:

- XML
  - `ADT_Bundle_filein`
  - `ADT_Patient_filein`
  - `fhir_patient_fileout`
  - `fhir_bundle_fileout`
  - `FHIR_Server_response`
  - `FHIR_Server_out`
- JSON
  - `ADT_filein_a_JSON`
  - `ADT_filein_b_JSON`
  - `UHN_HAPI_JSON`
  - `file_out_post_JSON`
  - `fhir_bundle_fileout_JSON`

Tcl procs

- `XltStrFhirFormatDateTime.tcl`
  - `XltStrFhirFormatDate`
  - `XltStrFhirFormatDateTime`
- `TpsFhirSetServerUrl.tcl`
- `XltStrFhirGenerateUuid.tcl`

Translations

- FHIR_DSTU2_Patient(_JSON).xlt
- FHIR_DSTU2_Encounter(_JSON).xlt
- FHIR_DSTU2_Message_Bundle_JSON_Include.xlt
- FHIR_DSTU2_Transaction_Bundle_Include.xlt

Tables

- V2-FHIR-Encounter-SystemValues.tbl
- V2-FHIR-IdentifierTypes.tbl
- V2-FHIR-Patient-AddressUse.tbl
- V2-FHIR-Patient-Gender.tbl
- V2-FHIR-Patient-Language.tbl
- V2-FHIR-Patient-MaritalStatus.tbl
- V2-FHIR-Patient-SystemValues.tbl
- V2-FHIR-ServerURL.tbl

Test data file: `Test_HL7_Message.dat`

Formats:

- json/fhir_1.0.2
- xml/FHIR_DSTU2_201510

Running examples

These examples use a sample HL7 v2.3 ADT^A01 message, located in `testdata/Test_HL7_Message.dat`.

There are two sample interfaces provided in the BOX:

- A file protocol that reads in the HL7 v2.3 message. Then, it translates it to an XML FHIR Patient Resource. This Patient Resource is sent to the public FHIR server at UHN through a Cloverleaf web services client.
- A file protocol that reads in the HL7 v2.3 Admin/Visit Notification. Then, it translates it into an XML FHIR Transaction Bundle containing a Patient and an Encounter Resource. This FHIR Bundle is sent to the public FHIR server at UHN through a Cloverleaf web services client.

There are also three “file out” protocol threads for writing out the XML that is being created, sent, and received.

JSON threads do the same work as XML threads. Compared with XML threads, they use JSON FHIR format, and the translation files that are based on JSON format.

Sample HL7 message

An example ADT `ADT^A01` is included in the box.

Example inbound data file. This is read from `testdata/Test_HL7_Message.dat`, before translation:

```
MSH|^~\&|SCM|SCM|HL7REVISION||20100205101158||ADT^A01|18003310|D|2.3^MEVN|A01|20100128154025|||
registrar2test^Test^Registrar2|20100128154025^MPID||40000051^^^XA System ID|40000051^^^MRN||
walker^george^barry^^Mr||19531120|Male||White|1000 sandtown circle^^captial^NT^98211^^Home||
(521)7835632^^^^^521^7835632^^Home|(218)3219611^^^^^218^3219611^^Business|English|Married|Baptist|
70000209^^^Account|332214321|||Not Hispanic/Latino|atlanta^MPD1|||||||N^MPV1||Inpa
```

```
tient|INC^^^DMC|Routine|||
027979^Ho^Eleanor^M|||||||Trans-Foster Home|||059964^Xerogeanes^John|Private|70000209^^^
Account||||||||||||||||||||||||20100128153600^MPV2||^N|leg pain|||||||||||||||||||||||||||||||||
Ambulance^MGT1|1|40000057^^^XA System ID|walker^george^barry^^Mr||
1000 sandtown circle^^cap
tial^NT^98211|(521)7835632^^^^^521^7835632|(218)3219611^^^^^218^3219611|19531120|Male|I|Self|332214321^M
```

Running the FHIR patient create/update interface

1. Open the Network Monitor, and start the `ADT_Patient_filein` protocol thread.
2. Start the `fhir_patient_fileout`, `FHIR_Server_out`, and `FHIR_Server_response` protocol threads.

   At this point, a Patient FHIR resource goes out to UHN and create or update the patient resource, resulting in:

```
<Patient xmlns="http://hl7.org/fhir">
        <identifier>
                <use value="usual" />
                <type>
                        <coding>
                                <system value="http://hl7.org/fhir/v2/0203" />
                                <code value="MR" />
                        </coding>
                </type>
                <system value="urn:oid:1.2.26.146.555.217.0.1" />
                <value value="40000059" />
                <assigner>
                        <reference value="http://somefhirserver.com/DSTU2/Organization/23"
/>
                        <display value="SCM" />
                </assigner>
        </identifier>
        <identifier>
                <use value="usual" />
                <type>
                        <coding>
                                <system value="http://hl7.org/fhir/identifier-type" />
                               <code value="SB" />
                        </coding>
                </type>
                <system value="urn:oid:2.16.840.1.113883.4.1" />
                <value value="332214321" />
                <assigner>
                        <display value="U. S. Social Security Administration" />
                </assigner>
        </identifier>
        <name>
                <use value="usual" />
                <text value="Mr george barry walker " />
                <family value="walker" />
                <given value="george" />
                <given value="barry" />
        </name>
        <telecom>
                <system value="phone" />
                <value value="(218)3219611" />
                <use value="work" />
        </telecom>
        <gender value="male" />
        <birthDate value="1953-11-20" />
        <address>
                <use value="home" />
                <line value="1000 sandtown circle" />
                <city value="captial" />
                <state value="NT" />
                <postalCode value="98211" />
        </address>
```

```
        <maritalStatus>
                <coding>
                        <system value="http://hl7.org/fhir/v3/MaritalStatus" />
                        <code value="M" />
                </coding>
                <text value="Married" />
        </maritalStatus>
        <contact>
                <relationship>
                        <coding>
                                <system value="http://hl7.org/fhir/patient-contact-relation
ship" />
                                <code value="guarantor" />
                                <display value="Guarantor" />
                        </coding>
                        <text value="Guarantor" />
                </relationship>
                <name>
                        <use value="usual" />
                        <family value="walker" />
                        <given value="george" />
                        <given value="barry" />
                        <prefix value="Mr" />
                </name>
                <telecom>
                        <system value="phone" />
                        <value value="(521)7835632" />
                        <use value="home" />
                </telecom>
                <telecom>
                        <system value="phone" />
                        <value value="(218)3219611" />
                        <use value="work" />
                </telecom>
                <address>
                        <line value="1000 sandtown circle" />
                        <city value="captial" />
                        <state value="NT" />
                        <postalCode value="98211" />
                </address>
                <gender value="male" />
        </contact>
        <communication>
                <language>
                        <coding>
                                <system value="urn:ietf:bcp:47" />
                                <code value="en" />
                        </coding>
                        <text value="English" />
                </language>
        </communication>
</Patient>
```

The response from the FHIR server is viewable in the `fhirwsclient` engine log.

This is an example response:

```
Response-Code: 200
Encoding: UTF-8
Content-Type: application/xml+fhir;charset=UTF-8
Headers: {connection=[close], Content-Location=[http://fhirtest.uhn.ca/baseDstu2/Pa
tient/122411/_history/2], content-type=[application/xml+fhir;charset=UTF-8], Date=[Tue, 18
Oct 2016 23:27:11 GMT], ETag=[W/"2"], Last-Modified=[Tue, 18 Oct 2016 23:27:11 GMT], Loca
tion=[http://fhirtest.uhn.ca/baseDstu2/Patient/122411/_history/2], Server=[Apache-Coyote/1.1],
 transfer-encoding=[chunked], X-Powered-By=[HAPI FHIR 2.1-SNAPSHOT REST Server (FHIR Server;
 FHIR 1.0.2/DSTU2)]}
Payload: <OperationOutcome xmlns="http://hl7.org/fhir">
   <text>
      <status value="generated"/>
      <div xmlns="http://www.w3.org/1999/xhtml">
         <h1>Operation Outcome</h1>
         <table border="0">
```

```
            <tr>
               <td style="font-weight: bold;">information</td>
               <td>[]</td>
               <td>
                  <pre>Successfully created resource &amp;quot;Patient/122411/_history/2&amp;quot;
in 33ms</pre>
               </td>
            </tr>
         </table>
      </div>
   </text>
   <issue>
      <severity value="information"/>
      <code value="informational"/>
     <diagnostics value="Successfully created resource &amp;quot;Patient/122411/_history/2&amp;quot;
 in 33ms"/>
   </issue>
</OperationOutcome>
```

Running the FHIR transaction bundle interface

1. Open the Network Monitor and start the `ADT_Bundle_filein` protocol thread.
2. Start the `fhir_bundle_fileout`, `FHIR_Server_out`, and `FHIR_Server_response` protocol threads.

   The `fhir_bundle_fileout` protocol thread writes the resulting FHIR Transaction Bundle to `tmp/fhir_bundle_out.xml` in the site directory, after translation:

```
<Bundle xmlns="http://hl7.org/fhir">
    <id value="e023021b-4e18-4d59-59c5-74e24fd823d9" />
    <type value="transaction" />
    <entry>
        <fullUrl value="urn:uuid:322b9283-ab9e-4c51-4a7b-6060513ffc3c" />
        <resource>
            <Patient>
                <identifier>
                    <use value="usual" />
                    <type>
                        <coding>
                            <system value="http://hl7.org/fhir/v2/0203" />
                            <code value="MR" />
                        </coding>
                    </type>
                    <system value="urn:oid:1.2.26.146.555.217.0.1" />
                    <value value="40000051" />
                    <assigner>
                        <reference value="http://somefhirserver.com/DSTU2/Organization/23"
/>
                        <display value="SCM" />
                    </assigner>
                </identifier>
                <identifier>
                    <use value="usual" />
                    <type>
                        <coding>
                            <system value="http://hl7.org/fhir/identifier-type" />
                            <code value="SB" />
                        </coding>
                    </type>
                    <system value="urn:oid:2.16.840.1.113883.4.1" />
                    <value value="332214321" />
                    <assigner>
                        <display value="U. S. Social Security Administration" />
                    </assigner>
                </identifier>
                <name>
```

```
                    <use value="usual" />
                    <text value="Mr george barry walker " />
                    <family value="walker" />
                    <given value="george" />
                    <given value="barry" />
                </name>
                <telecom>
                    <system value="phone" />
                    <value value="(218)3219611" />
                    <use value="work" />
                </telecom>
                <gender value="male" />
                <birthDate value="1953-11-20" />
                <address>
                    <use value="home" />
                    <line value="1000 sandtown circle" />
                    <city value="captial" />
                    <state value="NT" />
                    <postalCode value="98211" />
                </address>
                <maritalStatus>
                    <coding>
                        <system value="http://hl7.org/fhir/v3/MaritalStatus" />
                        <code value="M" />
                    </coding>
                    <text value="Married" />
                </maritalStatus>
                <contact>
                    <relationship>
                        <coding>
                           <system value="http://hl7.org/fhir/patient-contact-relationship"
/>
                            <code value="guarantor" />
                            <display value="Guarantor" />
                        </coding>
                        <text value="Guarantor" />
                    </relationship>
                    <name>
                        <use value="usual" />
                        <family value="walker" />
                        <given value="george" />
                        <given value="barry" />
                        <prefix value="Mr" />
                    </name>
                    <telecom>
                        <system value="phone" />
                        <value value="(521)7835632" />
                        <use value="home" />
                    </telecom>
                    <telecom>
                        <system value="phone" />
                        <value value="(218)3219611" />
                        <use value="work" />
                    </telecom>
                    <address>
                        <line value="1000 sandtown circle" />
                        <city value="captial" />
                        <state value="NT" />
                        <postalCode value="98211" />
                    </address>
                    <gender value="male" />
                </contact>
                <communication>
                    <language>
                        <coding>
                            <system value="urn:ietf:bcp:47" />
                            <code value="en" />
                        </coding>
                        <text value="English" />
                    </language>
                </communication>
            </Patient>
        </resource>
        <request>
```

```
            <method value="PUT" />
            <url value="Patient?identifier=40000051" />
        </request>
    </entry>
    <entry>
        <fullUrl value="urn:uuid:ec4b0444-99f3-4ac0-5d99-d4eecf27d641" />
        <resource>
            <Encounter>
                <identifier>
                    <use value="usual" />
                    <system value="urn:oid:1.2.26.146.555.217.0.2" />
                    <value value="70000209" />
                    <assigner>
                        <reference value="http://somefhirserver.com/DSTU2/Organization/23"
/>
                    </assigner>
                </identifier>
                <status value="arrived" />
                <class value="inpatient" />
                <type>
                    <text value="Routine-Inpatient" />
                </type>
                <patient>
                    <reference value="Patient?identifier=40000051" />
                    <display value="Patient MRN=40000051" />
                </patient>
                <period>
                    <start value="2010-01-28T15:36:00-05:00" />
                </period>
                <reason>
                    <text value="leg pain" />
                </reason>
                <hospitalization>
                    <admitSource>
                        <text value="Trans-Foster Home" />
                    </admitSource>
                </hospitalization>
                <location>
                    <location>
                        <display value="INC" />
                    </location>
                </location>
            </Encounter>
        </resource>
        <request>
                <method value="PUT" />
                <url value="Encounter?identifier=70000209" />
        </request>
    </entry>
</Bundle>
```

Additionally, if the Cloverleaf server can access the internet, the FHIR Transaction Bundle should have updated the public FHIR server at UHN.

This should be observable in the `fhirwsclient` process log, or similar:

```
Response-Code: 200
Encoding: UTF-8
Content-Type: application/xml+fhir;charset=UTF-8
Headers: {connection=[close], Content-Location=[http://fhirtest.uhn.ca/baseDstu2
/Bundle/6ac30661-930b-4ada-a7f8-9930aee9b116], content-type=[application/xml+fhi
r;charset=UTF-8], Date=[Tue, 18 Oct 2016 23:48:00 GMT], Location=[http://fhirtes
t.uhn.ca/baseDstu2/Bundle/6ac30661-930b-4ada-a7f8-9930aee9b116], Server=[Apache-
Coyote/1.1], transfer-encoding=[chunked], X-Powered-By=[HAPI FHIR 2.1-SNAPSHOT
REST Server (FHIR Server; FHIR 1.0.2/DSTU2)]}
Payload: <Bundle xmlns="http://hl7.org/fhir">
   <id value="6ac30661-930b-4ada-a7f8-9930aee9b116"/>
   <type value="transaction-response"/>
   <link>
      <relation value="self"/>
      <url value="http://fhirtest.uhn.ca/baseDstu2"/>
```

```
   </link>
   <entry>
      <response>
         <status value="200 OK"/>
         <location value="Patient/122414/_history/2"/>
         <etag value="2"/>
         <lastModified value="2016-10-18T19:47:59.894-04:00"/>
      </response>
   </entry>
   <entry>
      <response>
         <status value="200 OK"/>
         <location value="Encounter/122403/_history/2"/>
         <etag value="2"/>
         <lastModified value="2016-10-18T19:47:59.913-04:00"/>
      </response>
   </entry>
</Bundle>
```

This log file sample shows that the Patient and Encounter already existed and were updated to "`history/2`".

Public FHIR test servers and this BOX

The public FHIR server used with this BOX is the one provided by UHN. Because this is a public reference server provided at the discretion on UHN, it could go down, break, or be enhanced at any time. A FHIR Resource created or otherwise present today could be deleted without notice.

For additional public servers, see:

http://wiki.hl7.org/index.php?title=Publicly_Available_FHIR_Servers_for_testing

The URL for the server used by this BOX is read from `Tables/V2-FHIR-ServerURL.tbl`. This allows for changes to test with a different end-point. Values that are used from this table are Bundle, Patient and Patient_Action. The Patient and Bundle values are for the URL end-point (FHIR server URL). The Patient_Action specifies the ReST method to use. For this sample, PUT and POST are supported for Patient.

For the Bundle, the action is coded in the translation and is not overridden by this table. The Transaction Bundle is a ReST PUT for Patient and Encounter. The value for the match to determine update or create for patient is the first identifier from the translation (XML result). The Encounter match value is the Encounter ID from PV1-19.

Note: The Bundle succeeds only if there is a single match, or no match. Multiple matches cause the action to be unsuccessful on the FHIR server and an HTTP error (404) is returned. The response from the server in the error condition should state the reason for the rejection.

Updating FHIR schemas

When the FHIR examples BOX was created, the newest version of FHIR had not been balloted. You can obtain and use the interim standard known as STU3.

Use these steps to perform an update to the FHIR Schemas:

1. Download the FHIR schema zip file from http://hl7.org/fhir/2016May/fhir-codegen-xsd.zip.
2. Create a directory under `formats/xml` for the new `xsd` files. For example, `Stu3_may2016`.

   When you unzip the file, it contains many `xsd` files. There is a separate `xsd` file for each defined resource. For example:

   - `account.xsd`
   - `allergyintolerance.xsd`
   - `appointmentresponse.xsd`
   - `appointment.xsd`
   - `auditevent.xsd`

   There are also specialty `xsd` files, such as:

   - `fhir-all.xsd`
   - `fhir-base.xsd`
   - `fhir-single.xsd`
   - `fhir-xhtml.xsd`
3. The primary file is `fhir-single.xsd`.

   In the `xsd` compile section of the BOX deployment, this `xsd` file contains almost everything that is required in one file. In this case `xml.xsd` and `fhir-xhtml.xsd` are also required. The rest can be deleted.
4. Use the same steps in compiling the FHIR XML format (`fhir-single.xsd`) to compile.

   Optionally, you can run this command from the directory that contains `fhir-single.xsd`:

   ```
   hcixmlcompile -f fhir-single.xsd -p stu3_May2016 -F pretty
   ```

   Note: If you are using the new standard in an existing translation configuration, then the translation must be reconfigured to use the new XML package. The paths to some elements have changed. There is also the addition or removal of some elements and even whole resources. It is important to thoroughly and carefully review the translation to account for required adjustments.

Cloverleaf 6.2 translation Include operation

One of the features in Cloverleaf Integration Services is the Translation Configurator's Include operation. This is helpful with HL7 FHIR.

In the BOX that is described in the examples, one of the translations creates a FHIR Transaction Bundle. This contains a Patient Resource and an Encounter Resource. See Running examples. The translation for the individual resources, Patient, and Encounter can be completed first and tested separately. Then, they can be included to build a FHIR Bundle. This is how this BOX was created.

Before this feature was added, Transaction Bundle mapping had to be recreated starting from scratch, remapping each element for Patient and Encounter. With the Include operation, the stand-alone Patient and Encounter translations can be "included" into the translation for the FHIR Bundle.

To use the Include operation in the context of an FHIR Transaction Bundle:

- A FHIR Transaction Bundle includes a few additional elements.
- We recommend that a few elements of the FHIR Patient Resource also be performed. This establishes the differences in the paths to the elements in a Bundle versus stand-alone.
- The path difference information is used in the Include operation.

Using the Include operation

1 In the Translation Configurator, select **File > Reconfigure** and for **Root** select **`nm1:Bundle`**.

These are sample mappings from a Bundle:

- Source: transaction

  Action: COPY

  Destination: `nm1:Bundle.0.nm1:type.&value`

- Source: `0(0).PID(0).#3(0).[0]`

  Action: COPY

  Destination: `nm1:Bundle.0.nm1:entry(0).1.nm1:resource. nm1:Patient.1.nm1:identifier(0).0.nm1:value.&value`

- Source: `=usual`

  Action: COPY

  Destination: `nm1:Bundle.0.nm1:entry(0).1.nm1:resource.nm1:Patient.1.nm1:identifier(0).0.nm1:use.&value`

These are the sample mappings, similar to above, from a Patient Resource only:

- Source: `0(0).PID(0).#3(0).[0]`

  Action: COPY

  Destination: `nm1:Patient.1.nm1:identifier(0).0.nm1:value.&value`

- Source: `=usual`

  Action: COPY

  Destination: `nm1:resource.nm1:Patient.1.nm1:identifier(0).0.nm1:use.&value`

Note: The element path differences are used in the next step.

2 Select **New Append**, and for **Action** select **`INCLUDE`**.

3 In the **XLT File** list, select `FHIR_DSTU2_Patient.xlt`.

4 In this example, after `FHIR_DSTU2_Patient.xlt` is selected for the **Source**, the same inbound message structure is used for the host translation as the included translation. The host translation is the Xlate. The included translation is the sub-Xlate. The path for the whole message is `0(0)`. This is the value for both **Group Prefix** in **Host Xlate** and **Group Prefix** in **Sub-Xlate** on the Source side.

5 For the Destination, the information from the previous sample Copy operations, step 1, provides the necessary information. The path to the destination element needs to be modified. Taking the Patient ID value element from above, notice the difference in paths:

```
nm1:Bundle.0.nm1:entry(0).1.nm1:resource.nm1:Patient.1.nm1:identifier(0).0.nm1:value.&
    value
```

```
nm1:Patient.1.nm1:identifier(0).0.nm1:value.&value
```

With this information:

- The Destination column “Group Prefix in Host Xlate” should be:

```
nm1:Bundle.0.nm1:entry(0).1.nm1:resource.nm1:Patient
```

- The “Group Prefix in Sub-Xlate” column should be:

```
nm1:Patient
```

Note the value in the entry(0) path element. The FHIR Resources contained within an FHIR Bundle are a repetition of the Entity XML element. The next FHIR Resource in the Bundle has paths containing entry(1), and so on. In the example provided, the Encounter Resource is added after Patient and has an entry value of 1.

This completes the necessary steps to include a sub-Xlate within a host-Xlate.

Note: You cannot change the included Xlate code in the host translation. Changes to the included translation must take place in the included translation file. For example, `FHIR_DSTU2_Patient.xlt`. Changes can also be as additional code within the host translation after the Include operation. In this case, the full path to the elements is required.

Creating an HTTP outbound web service client thread

Web service client protocol threads and web service server protocol threads should always be in their own process. One process cannot have more than one web service protocol thread.

1. Create a new Cloverleaf thread using the `java/ws-rawclient` protocol.
2. Click **Properties** to start the web services wizard.
3. For debugging, select **Message Logging Enabled**.

   Note: This should not be selected for production interfaces due to log file performance, size, and data security.
4. Click **New** and on the Type Selection wizard page select **Create Raw Consumer**.
5. Click **Next**.
   - For **Address**, specify, for example, `http://fhirtest.uhn.ca/baseDstu2/Patient`.
   - For **Default Method**, select **POST** from the menu.

   Note: The URL that is used here is overwritten in the provided Box through the `Tcl UPoC: TpsFhirSet ServerUrl`, which is in place as a translation post-proc.
6. Click **Next**. Because this example is not using TLS, a conduit is not required. Select **Do Not Create New Conduit** and click **Finish**.
7. On the WS Raw Client dialog box, specify a **Name** for the web service. In this example, specify UHN_HAPI.

8 Because this web service Client is FHIR XML, click **Add** to create two Request Header Overrides:

```
Header Name: Content-type   Header Value: application/xml+fhir
Header Name: Accept         Header Value: application/xml+fhir
```

9 Click **OK**, and then save the changes in Network Configurator.

CAA-WS Swagger

With CAA WS swagger support, you can get the target service API definition in real-time to assist in the Consumer configuration.

Raw Consumer configuration

On the **CAA WS-RawClient Creating Wizard** dialog box, there is a **Configure Raw Consumer From Swagger File** page after the **Conduit Configuration** page. The **Finish** button is always enabled on this page. To skip this step, click **Finish**.

On this page, you can load the swagger JSON file from any location. This can be a local file, external URL, or ION API gateway URL. The swagger file is only supported in JSON format.

The loaded API paths and corresponding request method (HTTP verb) are presented as a tree structure on the left side of the page. You can select the request method node to switch to the corresponding **Parameter List** on the right side.

The **Parameter List** panel lists the **Name**, **Value**, **Type**, **Required**, and **Location** for each parameter. These fields are defined from the swagger file `parameters` node. Only the **Value** column is editable.

For the HTTPS URL, you must ensure the conduit configuration has already been prepared before configuring the Swagger Raw Consumer. Otherwise, the loading process opens a warning message and the process stops.

The loading process downloads the swagger file through the conduit.

When **Finish** is clicked, the wizard generates an individual Raw Consumer for each selected HTTP verb node.

The selected request method and all parameters are displayed on the **General** tab.

The selected request method name is displayed in **Default Method**.

For a regular HTTP request, the parameter can appear in the `path`, `header`, `query`, `FormData`, `cookie`, and `body`. The parameter values are stored at the corresponding places on RawConsumer General panel according to the location type.

The `path` and `query` parameters are displayed in **Address**.

The `header`, `cookie`, `formData`, and `body` parameters are displayed in the corresponding sub-tabs.

The **Headers**, **Cookies**, and **Form Data** sub-tabs have **Name**, **Value** columns and **Add**/**Remove** buttons. The **Body** sub-tab is a text area.

Parameter locations

Parameter location	UI component of General panel	Stored location
`path`	**Address** field	This value is stored in the address attribute of the `bean` tag.
`query`	**Address** field	This value is stored in the address attribute of the `bean` tag.
`header`	**Headers** sub-tab	This value is stored in the `headers` property of the `bean` tag.
`cookie`	**Cookies** sub-tab	This value is stored in the `cookies` property of the `bean` tag. **Note:** This location is only supported on OpenAPI 3.0.
`FormData`	**Form Data** sub-tab	This is stored in the `formData` property of the `bean` tag. This is used to construct an HTTP message payload when working with the POST action. This conflicts with the Cloverleaf outbound messages role. To provide better flexibility, the `=CLMsgPayload` keyword is reserved for a form-data value. In the server, when the keyword is found the related form key is assigned with the engine message payload. Form data value is taken as URL encoded. If you requires a value that is the same as `=CLMsgPayload`, then enter `%3DCLMsgPayload`. By doing this, you can take advantage of the form data and the message payload. **Note:** This location is only supported on Swagger 2.0.
`body`	**Body** sub-tab	This is stored in the `body` property of the `bean` tag. **Note:** This location is only supported on Swagger 2.0.

OAuth2 client on the Conduit panel

To access the published web service, the WS client supports OAuth2 authentication.

With OAuth 2.0 (Open Authorization), a website or application can access resources that are hosted by other web apps on behalf of a user.

The CAA thread functions as a server service and a OAuth2 client.

The Conduit panel contains an **Authentication** tab. This is for defining all authentication-related contents.

The **Authentication Type** contains an OAuth2 options.

The OAuth 2.0 Authentication panel includes these configuration fields:

- **Grant Type**
- **OAuth2 Token URL**
- **Scope**
- **Client ID**
- **Client Secret**
- **Service Account Name**
- **Service Account Password**

Notes:

- **Grant Type** supports “Resource Owner” and “Client Credential”.
- **Resource Owner** supports **OAuth2 Token URL**, **Scope**, **Client ID**, **Client Secret**, **Service Account Name**, and **Service Account Password**.
- The **Client Secret** and **Service Account Password** values are encrypted.
- **Client Credential** supports **OAuth2 Token URL**, **Scope**, **Client ID**, and **Client Secret**.
- **OAuth2 Token URL** cannot match with the current **Conduit Name**. Otherwise, the current conduit is unable to obtain the access token.

XSD WSDL tool: Client

This topic describes how and when to use the XSD WSDL tool client version to access a web service from the system.

When accessing a web service from the system, use the WSDL that describes the web service that the system accesses to create an XSD. This XSD describes the messages you can send back and forth.

You can then compile the XSD in a system XML package. This is used in creating transformations between the SOAP messages. These messages are sent between the web service and the message format you use internally to process the request.

This tool reads the WSDL and generates XSD files for input and output messages to the service. Then, the generated XSD files can be copied to the server using the XML Package Manager to compile for the service.

Before running the tool, prerequisites for all runs are:

- Both the system and CAA-WS must be installed properly.
- Create a site and have an XML package directory in it.

To run the tool from the command prompt:

- Have a command window open to the `CAA\ws\tool\xsdWsdlToolClient` directory.
- Run `setroot` in the command window.
- In the command window, run `setsite MySite`, replacing `MySite` with the name of your site.

To run the tool from the GUI, start the Cloverleaf GUI and switch to the desired site.

After running the tool, complete step 3 of Usage scenario: Accessing a web service in the system.

Usage scenario: Accessing a web service in the system

You can use these steps to access a web service in the system using the XSD WSDL tool.

1. Create a site and make a CAA-WS SOAP client thread for each web service operation you must support.

   Alternatively, one client thread can be used to invoke multiple operations within the same or even multiple WSDLs. In this case, specify a dispatch name to distinguish the SOAP clients belonging to a single thread.

2. Create a system XML package to hold your files. This is accomplished from the GUI or a command prompt.

   Run the XSD WSDL tool client. This creates XSD files for the input and output messages from the web service. The files are copied to the xml package that was created in XML Package Manager or the command

line. It then compiles them using `hcixmlcompile` from the GUI or command prompt. For example, `xyz_input.xsd` and `xyz_output.xsd`.

3 Create a translation from the client thread using `xyz_input.xsd`.

When creating the translation, the **Choose File Formats** dialog box opens.

- For **Format**, select **`XML`**. Then select your **Package**.
- For **Xml**, select **`xyz_input`**.
- For **Root**, only **`SOAP-ENV:Envelope`** is listed.

The result of the translation can be sent to invoke the service.

4 The response can be translated back as a reply if the service sends a reply. To do this, use **`xyz_output`** when selecting the translation format to read the XML output of the web service.

Client: Setting up single runs

You can do subsequent runs of the tool on the same source XSD to overwrite the files that were made in a previous run. You can repeat these steps with different XSD source files for different web services.

After completing the prerequisites, you can set up single runs.

See XSD WSDL tool: Client.

Determine the URI to the WSDL to use:

- If it is on a website, then the URI would be similar to `http://example.com/some/directory/path/xyz.wsdl`.
- If it is on your local computer, then the URI is similar to `file:///c:/temp/xyz.wsdl`.
- If you have the WSDL in the same directory in which you launched the tool, then use a relative path URI, for example, `xyz.wsdl`.

Running XSD WSDL tool: Client version by command line

1 From the command prompt, run `setroot` and `setsite` in a command window.

2 Run the `xsdWsdlToolClientGUI.bat` batch file to start the tool. In Linux, use `./xsdWsdlToolClientGUI.sh`.

Running the XSD WSDL tool: Client by GUI

From the GUI, the tool can be launched from **Launch Bar > Configuration > WSDL2XSD** or you could get a prompt when creating a WS client thread.

1. When the GUI is first opened, all entries are blank and **WSDL URI** is selected. Specify the URI to the WSDL to use.
2. Before clicking **Load WSDL**, select a check box.

   Show fully qualified names: Select to have the names from the WSDL that have a namespace associated with them printed with the namespace prefixed before the name.

   Print WSDL contents after loading: Select to have to have the WSDL text contents printed after loading.
3. Click **Load WSDL**. The program attempts to read the WSDL from the location specified.

   If it cannot be found, then an error prints in Results.

   If it finds the WSDL and can parse it, then the **Service**menu populates with the list of services in the WSDL. If there is only one choice in a menu or list, then it is automatically selected.
4. After selecting the service, the **Port** list populates. After selecting the port, the **Binding** and **PortType** fields show the name of the Binding and PortType. These are associated with the chosen port from the WSDL.
5. Select the **Operation** to invoke. This populates the **Input Message** and **Output Message** with the names of the messages for this operation.
6. Select the **Target Folder** where the target `xsd` files are generated.

   This could be the XML package in your site in which to generate and compile the XSDs if you are at the server side.
7. Specify the **Target XSD Filename**. When this is finished, **Full Input Filename** and **Full Output Filename** show the full path names of the resulting XSD files that are generated.
8. Click **Generate Target XSDs**. This takes the selected settings, pulls the relevant information from the WSDL, and creates two XSD files: input and output versions.

   These files have SOAP envelopes around the content chosen. System transformations can generate the entire SOAP envelope to send to a web service and parse the response SOAP envelope.

   If the generation fails, then an exception message is shown giving a description of the problem. Otherwise, the result shows that the XSD generation was successful.
9. This step is optional. If you did the previous steps at the client side, then upload the generated `xsd` files to the server side's xml package. You can use any tool or copy local folders to the package in the Cloverleaf XML Package Manager GUI.

   These XSDs can then be compiled in the XML Package Manager or directly by running the `hcixmlcompile` command at the server side into OCM files. These are used by the system to permit translations in the given XML structure.

XSD WSDL tool: Server

This topic describes how and when to use the XSD WSDL tool when creating a web service in the system.

When creating a web service in the system, it is useful to compile an XSD in a system XML package. This is used to create transformations between the SOAP messages sent over the web service and the message format that processes the request.

It is also useful to provide clients with a WSDL that describe your web service. This tool assists with the creation of these files.

Before running the XSD WSDL tool:

- Both the system and CAA-WS must be installed properly.
- Create a site and have an XML package directory in it.
- Have a command window open to the `CAA\ws\tool\xsdWsdlTool` directory.
- Run `setroot` in the command window.

After running the tool, go to steps 3 and 4 in the Usage scenario: Creating a web service with the XSD WSDL tool.

Multiple web services

A separate WSDL is created for each service. To have one WSDL describe all services, you must manually merge them.

The XSD WSDL server tool does not support adding more than one operation to a WSDL.

Multiple XSDs

If you have an XSD with your intended input element and another XSD with the output element, you must merge them.

The XSD WSDL server tool does not support using multiple XSDs or reading included XSDs inside of the first one.

Usage scenario: Creating a web service with the XSD WSDL tool

These are the typical steps to create a web service in the system using this tool:

1. Create a site and make a CAA-WS server thread in the site for each web service operation to support.
2. Create a system XML package to hold your files, from the system GUI or from a command prompt.
   a. Copy an XSD file which describes the input and output elements for the operation into the package folder. The XSD must have both input and output elements in one file. XSD includes are not supported.
   b. Run the XSD WSDL tool to create and compile `*_soap.xsd` and generate a WSDL.

   For example, if your source XSD is named `xyz.xsd`, then the tool creates `xyz_soap.xsd` and `xyz.wsdl`. The `xyz_soap.xsd` is then compiled with the system's `hcixmlcompile` program.

   Note: It is only necessary to create and compile a `*_soap.xsd` file if you are using `MESSAGE` mode and doing translations with the whole SOAP Envelope. If you are using `PAYLOAD` mode, then this step can be skipped. You must do translations only using your original XSD that describes the payload structure.
3. Create a translation from the server thread using `xyz_soap.xsd` when you are using `MESSAGE` mode in these steps. If you are using `PAYLOAD` mode, then you create a translation using only your original XSD.

   When creating the translation, the **Choose File Formats** dialog box opens.
   - For **Format**, select **`XML`**. Then select your **Package**.
   - For **Xml**, select **`xyz_soap`**.
   - For **Root**, only **`SOAP-ENV:Envelope`** is listed.

   The result of the translation can be used to implement the service. The response can be translated back as a reply if the service requires it.
4. Now that the service is ready to use, the WSDL, if you created one, can be provided to users of the service. Because the WSDL references the source XSD you provide, you must copy the XSD to the same directory as the WSDL. You can also have it sent along with it.

   Distribute the WSDL by configuring the server thread to serve that WSDL. When this is finished, you can get the WSDL by appending `?wsdl` to the service address that is specified in the server thread configuration.

   For example, if you specified a service URL `https://my.greatsite.com/uploadHL7`, the WSDL is available at `https://my.greatsite.com/uploadHL7?wsdl`.

   Alternatively, WSDLs can be sent out-of-band. For example, emailed.

Server: Setting up single runs

You can do subsequent runs of the tool on the same source XSD to overwrite the files that were made in a previous run. You can also repeat these steps with different XSD source files for different web services.

After completing the prerequisites, you can set up single runs.

See The XSD WSDL tool: Server.

1. Obtain or create an XSD file which has the input and output elements for your web service. Copy this file to your site's XML package directory. For example, `%HCISITEDIR%\formats\xml\MyPackage`. In UNIX, this is `$HCISITEDIR/formats/xml/MyPackage`.
2. In the command window, run `setsite` *`MySite`* , replacing *MySite* with the name of your site.

Running the XSD WSDL tool: Server from command line

Ensure that you have already run `setroot` and `setsite` and copied the source XSD into your XML package.

1. Run `setroot` and `setsite` in a command window.
2. Run the `xsdWsdlToolGUI.bat` batch file to start the tool. In UNIX, this is `./xsdWsdlToolGUI.sh`.

Running the XSD WSDL tool: Server from GUI

Ensure you have already run `setroot` and `setsite` and copied the source XSD into your XML package. The GUI tool uses a basic Java Swing.

1. When you first open the GUI, all entries are blank. From the **XML Package** list, select the XML package directories in your site.

   The **XSD File** list is populated with the list of XSD files in your selected package. Selecting the XSD file populates the list boxes for **Input Element** and **Output Element**. It also populates some of the WSDL generation input boxes with defaults based on the XSD file name.
2. Select the input and output elements from their menus. This populates the WSDL Generation's **Input Message** and **Output Message** input boxes with defaults. These defaults are from the selected input and output element names.
3. Select the SOAP version to use. SOAP 1.1 is the most widely supported. SOAP 1.2 is now supported by most web service stacks. This selection is used to generate the correct SOAP Envelope namespace. If you generate a WSDL, then it is set as the WSDL Binding type, so clients know to send SOAP messages of the correct version.
4. Click **Generate and Compile** to generate the new XSD and compile it. The output prints in Results.

   The results show that the XSD was successfully written. If there are lines showing XML compiler's `stdout:` or XML compiler's `stderr:`, then that indicates there was a problem compiling the new XSD. If it is not necessary to generate a WSDL, then close the dialog box.
5. To generate a WSDL, continue by doing this:
   - If you alter the **Service Name** field, then the other fields whose defaults are based on it are automatically repopulated.
   - Update **Service URL** to the correct `host:port/path` for your service.
   - Click **Generate** to write the WSDL.

     The results are written to **Results**, indicating that it successfully wrote the WSDL.

Portecle keystore management tool (third-party)

CAA-WS takes advantage of multiple CXF features which depend extensively on the usage of the Public Key Infrastructure (PKI) technologies. For example, HTTPS and WS-Security.

Therefore, the management of X.509 digital certificates and private keys are an essential part of the operation of these features.

Java

CAA-WS is written in Java, so the management of PKI and certificates is heavily influenced by the mechanism that Java uses to manage them. This is centered around the concept of keystore and truststore.

These types of stores are of the same internal structure, the JKS format or Java KeyStore. The distinction between them is primarily a logical one.

This specifies that the keystore contains public keys, and their associated private keys. These keys are used to authenticate self to remote partners. The truststore contains only the public keys of remote partners that are to be trusted.

Java run time, on which the system and CAA-WS run, provides a command line tool keytool that is capable of managing the keystore. Management of the keystore is in a variety of ways. These include generating a PKI key pair and its CSR, or certificate signing request, importing/exporting certificates, and so on.

Portecle open source GUI

One of the features in Portecle that is absent from keytool is the ability to import key pairs from another keystore. This is especially true for one in a different format from JKS. Sometimes it is convenient to import key pairs in the pkcs12 format, which is used in Microsoft's and other popular security frameworks.

Portecle has a user-friendly GUI and is simpler to use than the command-line based keytool. You should use Portecle to manage the keystores required for the CAA-WS HTTPS and other PKI functionalities.

Because the Portecle site http://portecle.sourceforge.net/ has extensive information on how to use this tool, this topic focuses more on installation.

Portecle installation

Download Portecle from http://sourceforge.net/projects/portecle. Click the **Download** link to get the latest version. Then, unzip it to a directory, for example, `portecle-1.7`.

Alternatively, some operating systems such as Linux may already have an RPM package built for your system.

Read the `readme` file in the unzipped directory about prerequisites to complete before launching the tool and other installation information.

- If the machine that runs this tool already has JDK/JRE 1.6 installed, then the installation is already finished.
- You should update the JCE unlimited strength jurisdiction policy files in the JRE so that Portecle can handle more key sizes or algorithms as necessary.

  These policy files are downloaded from Java's website and overwrite the same named files under `lib/security` of the JRE installation. The overwritten files include `local_policy.jar` and `US_export_policy.jar`

Launching Portecle

The `readme` file that comes with Portecle has detailed instructions on launching.

After the correct JCE provider jar and unlimited strength policy files have been properly installed, the tool is launched using the `java -jar` command.

If `.jar` files are associated with Java on your operating system, then double-click `portecle.jar` to launch it.

WS-Policy

This enhancement relies on the WS-Policy framework already available within CXF. See http://www.w3.org/TR/ws-policy/.This framework permits relatively convenient engagement of these technologies:

- WS-SecurityPolicy 1.2 specifies security policy assertions for WS-Security, WS-Trust, and WS-SecureConversation.
- WS-Security 1.0 and 1.1 includes UserNameToken 1.0 and 1.1, X509 Certificate Token 1.0 and 1.1, and SAML Token 1.0, 1.1, and 2.0 support.
- WS-SecureConversation 1.3 builds on WS-Trust and WS-Security to establish a faster channel for multiple secure messages between two systems, the conversation. This is in comparison with using asymmetric X509 encryption on every message.
- WS-ReliableMessaging, 2005 version, a protocol that permits messages to be delivered reliably between distributed applications in the presence of software component, system, or network failures.
- WS-Addressing 1.0 conveys end-to-end message characteristics including references for source and destination endpoints and message identity.
- WS-Trust 1.3 WS-Security defines the basic mechanisms for providing secure messaging. WS-Trust uses these base mechanisms and defines additional primitives and extensions for security token exchange. These enable the issuance and dissemination of credentials within different trust domains.
- WS-Trust, SAML, and other technologies can be engaged by WS-Policy documents but are not supported in this release due to time constraints for dependency configurations. An advanced user can manually create the required policy document and create the necessary Java classes to support these technologies.

This topic gives a short introduction of how to setup a WS-Policy based WS-Security setup. It may be all that is required for those users with an understanding of WS-Security and related configuration technologies such as Java keystores/truststores. These topics explain more of the configuration details.

There are four major components involved with configuring WS-Policy in CXF:

- Policy files
- jaxws:client and jaxws:server configuration properties
- Encryption/signature property classes
- Callback classes

The steps are illustrated by adding UsernameToken processing to the ws_samples site's registry and registryClient example. Message body encryption is not used to keep the messages as basic as possible for analysis and explanation.

Overview of steps

1. Modify the WSDL to include the WS-Policy documents necessary. A copy of the WSDL is created for this.
2. Set the involved client and server threads to use that WSDL copy.

3 Provide a list of valid user names/passwords for the server, and provide one of those user names/passwords to the client.
4 Start the threads and test.
5 Change to an unknown user name and verify failure.

Modifying the WSDL

1 Open the **Properties** dialog box for the SOAPProvider_Registry thread and switch to the **Policy Generator** tab.
2 Select the **Use Username Token** check box. This is all that is required to keep the messages as basic as possible to demonstrate how it works.
3 Scroll to the bottom of this dialog box and click **Generate WSDL with Policy Save As**. This opens a dialog box to save the modified WSDL document. This is the same WSDL, but has WS-Policy docs included. Save it with a new name, so as not to overwrite the original.

Using the new WSDL

1 Select the **General** tab.
2 For **WSDL Location**, change the WSDL file name to match the one created.
3 Go to the `RegistryClient` thread and change the **WSDL Location**. This sets both the client and server threads to use the same WSDL and WS-Policy documents.

Providing valid usernames for server and select username for client

1 On the server thread, SOAPProvider_Registry, go to the **Policy Properties** tab.
2 In the General Properties section, select **Use Default Handler** and specify a **Name** and **Password**. The callback handler is used to look up passwords for a variety of WS-Security features. You can supply your own class if necessary. For now, the default works to demonstrate username token validation.
3 Save the changes.
4 On the Client thread, RegistryClient, go to the **Policy Properties** tab.
5 Specify this same user in the UsernameToken Properties **username** field. Leave the **password** field blank.
6 Do the same for the callback handler that was completed on the server. Select **Use Default Handler** and put in the same username/password pair. Then add another invalid user.
7 When the client sends its message, it uses the username specified under UsernameToken Properties and uses this list to look up the password. When an invalid test to verify functionality is made, change the username to “bob."

8 Save the changes.

Java driver bug

There is a Java Driver bug in Cloverleaf 6.0.1 on Windows which causes the Java temp directory to be incorrect. This causes the server threads to fail when receiving a message. This happens because CXF attempts to use the temp directory to store information to help prevent replay attacks. With an unusable temp directory, this fails.

To work around this issue, in the Network Configurator's **Process Configuration** dialog box, on the **Java Driver** tab, **User Defined Options** sub-tab, set:

- **Name**: `java.io.tmpdir`
- **Value**: `C:\Users\hciuser\AppData\Local\Temp`

This sets `java.io.tmpdir` to the `hciuser` temp directory. Ensure not to put a "`\`" on the end of the directory name. This is only necessary on Windows and only causes issues on server threads, preventing replay attacks.

Starting and testing threads

1 Start the registry and registryClient threads and send the sample message request. See CAA-WS sample sites.

2 View the log file for the registry process.

On the inbound message is this header in the SOAP Header section of the message log:

```
<wsse:UsernameToken xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-
wssecurity-secext-1.0.xsd"
xmlns:wsse11="http://docs.oasis-open.org/wss/oasis-wss-wssecurity-secext-1.1.xsd"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd"

wsu:Id="UsernameToken-937A5BD494264E539813838651676032"><wsse:Username>jesse</
wsse:Username><wsse11:Salt>AvHvscM5boqmq+YCCd7FaQ==</wsse11:Salt><wsse11:Itera
tion>1000</wsse11:Iteration></wsse:UsernameToken>
```

In this, there is a request username. Note that the password is not visible on the network. This header goes in the message to Cloverleaf if the server is set in `MESSAGE` mode. This is so Tcl or other code can access the **Username** value for the request. Then the code can ascertain that it has already been validated by the server. Otherwise, the message would not have made it to Cloverleaf.

Policy files

Policy files are XML descriptions of the policies that a service is using. These can range from basic assertions such as “use WS-Addressing,” to complex WS-SecureConversation security settings. CXF supports many ways to attach these policy files to a service.

They can be attached in the `jaxws:server` and `jaxws:client` configuration settings. This has a potentially severe drawback because only one policy can be specified and encryption/signing parts cannot vary between request and response messages. This could cause trouble with customers whose consultants have advised variation between request/response signing/encrypting.

Due to this essentially cosmetic issue, CAA-WS attaches policy files directly into the WSDL files that describe a service. This has these advantages:

- If the WSDL is supplied with WS-Policy already in place, then do not be concerned about policy file settings. Nevertheless, there could be non-standard policy assertions in place that are not supported by CXF. An example of CXF is Microsoft specific policy settings that are created by `.NET` services. There could also be assertions that require the user to write unimplemented class files. For example, various validators.
- Flexibility is increased by specifying different signing/encryption and other rules on the input/output messages of individual operations within a given binding.
- Service consumers can use the policy described in the WSDL to automatically configure their own client systems.

The CAA-WS GUI only supports configuring the WSDL files to contain policy elements and not attached by `jaxws:client` and`jaxws:server` configuration elements. A user that knows CXF can manually write the configuration files as desired.

Running fail test

To verify that messages that do not pass the security check do not make it to Cloverleaf, send another request with an invalid username. Use one that the server does not recognize.

1. Return to the **Properties** for the registry client and change the username to "bob." This should match the username that was previously added in anticipation of this test to the default callback handler configuration. This makes it a valid user from the client perspective so it can send the request, but the server should reject it.
2. Save the changes.
3. Restart the client process, resend the request message, and review the server log again.

   If you have left the default logging configuration for the registry process in place, then the inbound message and this log entry are:

   ```
   Nov 7, 2013 3:10:22 PM
       com.infor.cloverleaf.gjdws.utils.DefaultCallback handle FINER: password lookup
       for 'bob' found no password
   ```

   This is the server checking the username "bob" and not finding a password. Next in the log is the exception produced when the user fails to be authorized:

```
Nov 7, 2013 3:10:22 PM
    org.apache.cxf.ws.security.wss4j.WSS4JInInterceptor handleMessage WARNING:
    org.apache.ws.security.WSSecurityException: The security token could not be
    authenticated or authorized
```

At this point, the exception is sent back as a response to the query and no message is sent to Cloverleaf. This is the expected behavior. Unauthenticated messages do not go to Cloverleaf. Code is not required there to check whether a request was authorized against the policy.

jaxws:client and jaxws:server configuration properties

The `jaxws:client` and `jaxws:server` configuration elements can be used to configure these properties. These are configured automatically by the GUI tool.

The column on the right side is used to indicate if the given property is available in the release for configuration by the GUI. These properties are displayed by exact matching name, case included, in the **Policy Properties** tab. The names have the `ws-security` prefix stripped off for readability. All properties can be manually configured by editing the relevant application context XML file.

User properties

See http://cxf.apache.org/javadoc/latest/org/apache/cxf/ws/security/SecurityConstants.html.

Property name	Description	Supported
`ws-security.username`	This is the user's name. It is used differently by each of the WS-Security functions.	X
`ws-security.password`	This is the user's password when `ws-security.callback-handler` is not defined. This is only used when adding a password to a UsernameToken.	X
`ws-security.signature.username`	This is the user's name for signature. This is used as the alias name in the keystore to get the user's cert and private key for signature.	X
`ws-security.encryption.username`	This is the user's name for encryption. This is used as the alias name in the keystore to get the user's public key for encryption.	X

Callback class and crypto properties

See http://cxf.apache.org/javadoc/latest/org/apache/cxf/ws/security/SecurityConstants.html.

Class	Description	Supported
`ws-security.callback-handler`	`CallbackHandler`class used to obtain passwords.	X
`ws-security.saml-callback-handler`	SAML `CallbackHandler` class used to construct SAML Assertions.	
`ws-security.signature.properties`	Crypto property to use for signature, if `ws-security.signature.crypto` is not set instead.	
`ws-security.encryption.properties`	Crypto property to use for encryption, if `ws-security.encryption.crypto` is not set instead.	
`ws-security.signature.crypto`	Crypto object to be used for signature. If this is not defined, then `ws-security.signature.properties` is used instead.	X
`ws-security.encryption.crypto`	Crypto object to be used for encryption. If this is not defined then `ws-security.encryption.properties` is used instead.	X

Note: For symmetric bindings that specify a protection token, the `ws-security-encryption` properties are used.

Boolean WS-Security configuration tags

For example, the value should be "true" or "false."

Constant	Default	Definition	Supported
`ws-security.validate.token`	true	Whether to validate the password of a received `UsernameToken` or not.	X
`ws-security.enableRevocation`	false	Whether to enable Certificate Revocation List (CRL) checking or not when verifying trust in a certificate.	X

Constant	Default	Definition	Supported
`ws-security.username-token.always.encrypted`	true	Whether to always encrypt `UsernameTokenS` that are defined as a `SupportingToken`. This should not be set to false in a production environment, as it exposes the password, or the digest of the password, on the wire.	X
`ws-security.is-bsp-compliant`	true	Whether to ensure compliance with the Basic Security Profile (BSP) 1.1.	X
`ws-security.self-sign-saml-assertion`	false	Whether to self-sign a SAML Assertion. If this is set to true, then an enveloped signature is generated when the SAML Assertion is constructed.	
`ws-security.enable.nonce.cache`	varies	Whether to cache `UsernameToken` nonces.	X
`ws-security.enable.timestamp.cache`	varies	Whether to cache Timestamp Created Strings.	X

Non-boolean WS-Security configuration parameters

Constant	Definition	Supported
`ws-security.timestamp.timeToLive`	Time in seconds to append to the Creation value of an incoming Timestamp to determine whether to accept the Timestamp as valid. The default value is 300 seconds (5 minutes).	X
`ws-security.timestamp.futureTimeToLive`	Time in seconds in the future within which the Created time of an incoming Timestamp is valid. The default value is 60.	X

Constant	Definition	Supported
ws-security.saml-role-attributename	Attribute URI of the SAML AttributeStatement where the role information is stored. The default is http://schemas.xmlsoap.org/ws/2005/05/identity/claims/role.	
ws-security.kerberos.client	Reference to the KerberosClient class used to obtain a service ticket.	
ws-security.spnego.client.action	SpnegoClientAction implementation to use for SPNEGO. Permits the user to plug in a different implementation to obtain a service ticket.	
ws-security.kerberos.jaas.context	JAAS Context name to use for Kerberos. Currently only supported for SPNEGO.	
ws-security.kerberos.spn	Kerberos Service Provider Name (spn) to use. Currently only supported for SPNEGO.	
ws-security.nonce.cache.instance	Holds a reference to a ReplayCache instance that a is used to cache UsernameToken nonces. Default instance that is used is the EHCache ReplayCache.	
ws-security.timestamp.cache.instance	Holds a reference to a ReplayCache instance that is used to cache Timestamp Created Strings. Default instance that is used is the EHCacheReplayCache .	
ws-security.cache.config.file	Set this property to point to a configuration file for the underlying caching implementation. The default configuration file that is used is cxf-ehcache.xml in the cxf-rt-ws-security module.	
org.apache.cxf.ws.security.tokenstore.TokenStore	TokenStore instance to use to cache security tokens. By default, this uses the EHCacheTokenStore if EhCache is available. Otherwise it uses the MemoryTokenStore .	

Constant	Definition	Supported
`ws-security.subject.cert.constraints`	Comma separated string of regular expressions that are applied to the subject DN of the certificate. This is used for signature validation, after trust verification of the certificate chain that is associated with the certificate. These constraints are not used when the certificate is contained in the keystore (direct trust).	
`ws-security.role.classifier`	If one of the WSS4J Validators returns a JAAS Subject from Validation, then the `WSS4JInInterceptor` attempts to create a `SecurityContext` based on this Subject. If this value is not specified, then it tries to get roles using the `DefaultSecurityContext` in cxf-rt-core. Otherwise, it uses this value in combination with the `SUBJECT_ROLE_CLASSIFIER_TYPE` to get the roles from the Subject.	
`ws-security.role.classifier.type`	If one of the WSS4J Validators returns a JAAS Subject from Validation, then the `WSS4JInInterceptor` attempts to create a `SecurityContext` based on this Subject. Currently accepted values are `prefix` or `classname`. Must be used in conjunction with the `SUBJECT_ROLE_CLASSIFIER`. The default value is `prefix`.	

Unsupported entries are links to a java object classname which requires significant developer knowledge, or highly specialized configuration options. In both instances, in-depth knowledge is required. This level of user must determine the correct value to specify in the application context XML. These entries might be added in future revisions as more requirements are made.

Encryption/signature class files

A class `MerlinWrapper` is provided that permits the encryption/signature properties to be set into a Spring Bean in the configuration file. The GUI creates instances of this class and sets the parameters as specified by the user for keystore/truststore/and so on. Then, it sets the crypto properties to point at these class instances.

This removes the typical necessity for separate crypto property files. It also permits the entire WS-Policy set of properties to reside in the one XML configuration file generated by the GUI.

Callback classes

These are essentially classes that CXF calls when it requires more information. The value this field requires is the fully qualified classname that implements the callback interface.

The `ws-security.callback-handler` is used when CXF has a username or a certificate alias in hand and must know the password for it. This is the user's password in the case of username, or the key password in the case of certificate alias.

A sample class is supplied whose key/value pairs are supplied in the config file from the GUI, called the default callback handler. The source code is also supplied. Advanced users can follow the same interface and implement a database, or LDAP or XYZ, driven callback class.

This is the source code for the default `callback` class:

```
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;

import org.apache.ws.security.WSPasswordCallback;

public class DefaultCallback implements CallbackHandler {

 // hold the username to password data in this map
 private Map<String, String> passwords =
        new HashMap<String, String>();

 @Override
 public void handle(Callback[] callbacks) throws IOException,
   UnsupportedCallbackException {
  for (int i = 0; i < callbacks.length; i++) {
             WSPasswordCallback pc = (WSPasswordCallback)callbacks[i];

   String pass = passwords.get(pc.getIdentifier());
   if (pass != null) {
   pc.setPassword(pass);
   return;
   }
  }
 }

 public Map<String, String> getPasswords() {
  return passwords;
 }

 public void setPasswords(Map<String, String> passwords) {
  this.passwords = passwords;
 }

}
```

When the handle method runs, it does a lookup in the hashmap and stores the password in the Callback object. CXF then uses this password in different ways, depending on the context.

- If it is doing a UsernameToken operation, then it passes in a username to determine the password that goes along with it.
- If it is doing an X509 signing operation, then it looks for the password for the key alias.

User interface for configuration and policy generation

Policies are XML snippets to be embedded in the WSDL. A special tool within the **Policy Generation** tab on SOAP Clients and Servers is provided to generate these policy files. Because the WSDL can be at a remote location that is accessed by HTTP, there is no guarantee that the IDE can edit the WSDL.

For these reasons, the tool generates a copy of the WSDL document with the selected policy options. It is up to users to edit the original WSDL. Or, users can store this copy and point to that instead, so that the service uses these policy settings:

- Reliable Message Delivery
- User Username Token
- Use X509 certificates for signing/encryption
- Establish WS-SecureConversation
- All Input Operations, all options
- All Output Operations, all options

These options represent a relatively confined subset of possible WS-Policy configuration options. As the array of options is complex, this list provides the most commonly used configuration options. This tool is only a generator. It is not an editor because of the limitless configuration options that users could manually edit a WSDL to contain.

Options in the generator become disabled when incompatible options are selected. For example, when X509 signing/encryption is enabled, **Use Transport Security** is disabled. This is not to say that you cannot run X509 signing/encryption over HTTPS. Technically, you can run X509 if you manually configure WS-Security. It is only that WS-Policy deems it pointless. If they both were to be enabled simultaneously, then each can be used without the other by the policy compliance engine. This is because WS-Policy compliance permits automatic choosing between equivalent alternatives available in a policy.

Other options are child options, and so are disabled if their parent option is not selected. There are many options under X509 signing/encryption. If X509 signing/encryption is not selected, then those options are disabled.

WS-Addressing is automatically checked when another option is selected that requires WS-Addressing to be enabled. It is also disabled so it cannot be unchecked. If WS-Addressing is enabled on a policy, then it does not matter whether it is checked or not in the SOAP client or server options.

- If the policy requires it, then it is enabled.
- If the policy does not require it, then the SOAP Client/Server options take effect.

You can use the two buttons to generate the modified WSDL and display it in the box to be copied. You can also generate the modified WSDL and display it in the box. This opens a **Save As** dialog box for you to save the file to the file system.

Policy properties

The other configuration settings are `jaxws:server` and `jaxws:client`. These are separate from the WS-Policy snippets that are embedded in the WSDL. They are used by CXF to configure certificates, usernames, passwords, and so on, that are necessary for it to comply with those WS-Policies. They are all written to the same application context file to which the properties on the **General** tab are written.

All lowercase names that are shown in the UI match the tables in the User properties section. These are all prefixed by `ws-security`. For example, the username property that is shown is the `ws-security.username` property. You can refer to CXF documentation for more details if necessary. See jaxws:client and jaxws:server configuration properties.

The uppercase names that are shown in the UI are for the crypto properties that are not `ws-security.*` type properties. Instead, they are configuration elements for the CAA-WS provided cryptographic configuration class. For example, the keystore path and password are stored in the crypto class linked under the `ws-security.signature.crypto` property. Keystore and Truststore are the only ones currently similar to that.

There is no design time linkage to the policy files. This is impossible, as there could be multiple policies within a WSDL and the one to be enabled depends on the message being sent/received. The UI does not know which fields the user requires to fill out. You must know if your policy uses UsernameTokens and if so, fill out that section. The same applies for X509 certificates.

The General Properties section has items applicable to one or more of the previous sections. For example, the callback handler applies to UsernameToken and X509. It also has that are global to the policy. For example, the timestamp information.

The Callback classes section has more details on the callback handler in general and some specifics about the default callback handler. See Callback classes.

USERDATA overrides

Overrides are a necessary part of WS-Security. For example, in the case of UsernameToken, it would be advantageous to send a variety of usernames. The mechanism is not difficult to use. It is the same as all the other CAA-WS overrides, and uses the `USERDATA` field. To override any of the listed fields pass a key with that name into the nested keyed list under the new `wss` key. This includes those not supported by the GUI.

For example, to override the UsernameToken's username, a specific alias to sign with, and a specific alias to encrypt to:

```
{{wss {{ws-security.username james}
{ws-security.signature.username charles}
{ws-security.encryption.username rob}}}}
```

Different from other aspects of CAA-WS, inbound messages do not have a corresponding `wss` key with WS-Security related information. The issue is that the information is not readily available in CXF. Most applications, though, only require the security verified. This happens automatically when the WSDL contains WS-Policy statements. Applications requiring more than that can use `MESSAGE` mode to receive the entire SOAP message. Then, they can view the WS-Security SOAP headers to read information such as the UsernameToken's username, X509 certificate's CN, and so on.

Subsequent versions could change this and provide parsed information in a `wss` key. You can submit requests for this to Support.

CAA-WS logging

Logging is a developer's main method of fixing bugs. The key to good logging is to log only what you think you require, instead of an overwhelming flood of unnecessary information to sift through. This could lose the import logs within.

CAA-WS has these logging options:

- Inbound/outbound message logging
- Cloverleaf message dump
- CAA-WS internal logging

Inbound/Outbound message logging

With SOAP/REST clients and servers, and the Raw Client, CXF-based message logging can be enabled. The Raw Handler (server) does not have this feature. It runs at the Jetty level, and does not pass through CXF code. This type of logging shows inbound and outbound messages as the client sees them when it is sending.

SOAP/REST clients/servers have their own individual **Message Logging Enabled** check boxes on their configuration screens.

For everything except Raw Handler (server), you can turn all on/off on the Bus.

The tooltip reminds you about which type of configuration items the message logging affects.

Cloverleaf message dump

By calling Tcl to dump a message you can see exactly what Cloverleaf is getting from and sending to a CAA-WS thread.

All the sample threads use this. For outbound threads (clients), the **Outbound** tab has the **TPS Outbound Data** UPoC and **TPS Inbound Reply** UPoC options.

Both of these have the `dumpMsg` Tcl proc which dumps the message for the outbound message and its reply, respectively. For inbound threads (servers), the **Inbound** tab has the **TPS Inbound Data** and **TPS Outbound Reply** UPoCs.

CAA-WS internal logging

For troubleshooting, CAA-WS logs information as it processes messages. For most users, this is unnecessary. If you suspect a bug or some fault lies with CAA-WS or the underlying CXF or Jetty, then enable the Java-based logging for these software packages. This might give you hints about what is going on.

How to configure

This sort of logging falls under the Java Util Logging domain. You can search the internet for more information about what sort of logging configurations are possible under Java Util Logging. This includes specific examples of turning on logging to a separate file, learning about log levels, and so on.

For an example, the registry process in the ws_samples sample site has it enabled. This is configured in the **Process Configuration** dialog box.

For the process, you create a new **User Defined Option** with the name `java.util.logging.config.file` and set the value to the location of your `logging.properties` file. The Java Driver *$SITEPATH* variable points to the site directory. Under that, in the WS directory, is the sample `logging.properties` file.

By default at the bottom of this file are two relevant lines:

```
org.apache.cxf.level = INFO
com.infor.cloverleaf.gjdws.level = FINEST
```

These set CXF logging to a high level and GJDWS to the lowest level possible. Generic Java Driver web services (GJDWS) is the internal name for CAA-WS. By doing this, CXF logs major events and CAA-WS logs everything it can.

Output example

This is an example in the process log from CAA-WS:

```
Nov 7, 2013 2:58:50 PM com.infor.cloverleaf.gjdws.WSServer doStart
INFO: CAA-WS license verified
```

If the level was turned up higher, for example, `WARNING`, then this would not show in the log. An example from a library not specifically listed in `logging.properties` is:

```
Nov 7, 2013 2:58:50 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDef
initions
INFO: Loading XML bean definitions from URL [file:/C:/cloverleaf/cis6.0/integrator/ws_samples/
javadriver/registry/../applicationContext_SOAPProvider_Registry.xml]
```

This log is from the Spring Framework when the XML context file is loaded. `INFO` level logs from libraries are printed by default. This is also adjustable in the `logging.properties` file as the `ROOT` logger. To turn this off, set `org.springframework.level` to `WARNING`.

This example of a CXF log indicates on which URL the service is listening:

```
Nov 7, 2013 2:58:54 PM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be http://localhost:9003/xdsregistryb
```

These `INFO` level logs are useful in general and are not too verbose. None of them print on every message. This is a method of visually verifying the URL a server is listening on.

This is a lower level log example:

```
Nov 7, 2013 2:59:30 PM com.infor.cloverleaf.gjdws.utils.DefaultCallback handle
FINER: password lookup for 'jesse' found the stored password
```

This log entry happens when using WS-Security. It indicates that the username “jesse” was successfully used to locate a password. These lower level logs are not generally good in production. This is because they can happen on every message and clutter the log. They can be useful, though, in development to verify things such as the username/password lookup is working as expected, especially if security was failing.

Enable Jetty access log

All CAA-WS providers are built on Jetty. You can enable the Jetty access log onto the CAA-WS inbound/service side. This is configured by editing the thread's application context XML file.

The application context XML file is located in the site under the `javadriver` directory. The file name format is `applicationContext_`*`thread name`*`.xml`.

All provider threads listed in this file have at least one *`httpj:engine`* element, from which you can update handlers in the engine.

Example file content:

```
<httpj:engine-factory>
<httpj:engine port="8080">
<httpj:handlers>
......
<bean class="org.eclipse.jetty.server.handler.RequestLogHandler">
<property name="requestLog">
<bean class="org.eclipse.jetty.server.Slf4jRequestLog"/>
</property>
</bean>
......
<bean class="com.infor.cloverleaf.gjdws.handlers.CLWSJettyDefaultHandler"/>
```

Example output:

```
[java:java:INFO/1:thread_cmd:10/27/2020 11:18:36] org.eclipse.jetty.server.Slf4jRequestLog
write:INFO: 0:0:0:0:0:0:0:1 - -
[27/Oct/2020:17:18:06 +0000] "GET /raw/foo HTTP/1.1" 500 98
[java:java:INFO/1:thread_cmd:10/27/2020 11:18:36] org.eclipse.jetty.server.Slf4jRequestLog
write:INFO: 0:0:0:0:0:0:0:1 - -
[27/Oct/2020:17:18:36 +0000] "GET /favicon.ico HTTP/1.1" 200 1150
```

NCSA Style:

```
<httpj:engine-factory>
 <httpj:engine port="8080">
 <httpj:handlers>
 ......
 <bean class="org.eclipse.jetty.server.handler.RequestLogHandler">
 <property name="requestLog">
 <bean class="org.eclipse.jetty.server.NCSARequestLog"/>
 </property>
 </bean>
 ......
 <bean class="com.infor.cloverleaf.gjdws.handlers.CLWSJettyDefaultHandler"/>
```

Example output:

```
0:0:0:0:0:0:0:1 - [27/Oct/2020:17:48:20 +0000] "GET /raw/asdf HTTP/1.1" 500 98 "" "Mozilla/5.0
(Windows NT 10.0;
Win64; x64; rv:81.0) Gecko/20100101 Firefox/81.0"
```

GUI option

The Jetty access log is enabled with CAA-WS providers.

There is a **Log HTTP Requests** option located on the **WS Server** dialog box.

This supports NCSA and SLF4J at the Jetty/engine level.

This option enables users to log the HTTP requests in these different formats:

- blank (default)
- SLF4J
- NCSA
- The relevant `xml` is added to `applicationContext.xml`. If "blank" is selected, then the logging function is removed from `applicationContext.xml`.

Updating CAA-WS 1.x sites to 2.0 and later

For users with CAA-WS 1.x sites, this topic describes how to update your sites to 2.0 and later.

The `Update1xto2.jar` upgrade tool runs the update. This is located in `CAA/ws/tool`. To start it, open a command window and run these commands:

```
setroot
java -jar Update1xto2.jar
```

The tool searches your Cloverleaf for sites with CAA-WS 1.x threads, and prompts a list of sites to select which ones to update. When a site is selected, it identifies all the CAA-WS 1.x threads that it intends to update. Then, it prompts you whether to update these threads.

What gets updated

The upgrade tool runs this algorithm:

1. Loop over NetConfig JavaDriver protocol entries and determine process name and thread name.
2. Use process name to get the `pni` file, if present. Then, get `START_DIR` from the `pni` file.
3. Use thread name to get `ini` file, then get `START_DIR` from `ini` if there was no `PNI`. Fault if none present.
4. From `ini`, get `STARTARG`. For example, `STARTARG=WSApplicationContext_SOAPRegistryClient.xml`. Combine `START_DIR` with `START_ARG` to locate the path to the xml config file.
5. Move the config file to `$HCISITEDIR/javadriver` and rename to *`threadname`*`_applicationContext.xml`.
6. Update `ini` and set `STARTARG` to `../`*`threadname`*`_applicationContext.xml` . This is relative to `START_DIR` set in next step.
7. Update `pni` and set `START_DIR` to `javadriver/`*`processname`*. Create the `javadriver/`*`processname`* directory, if it does not exist.
8. Determine type from `CLASS` in `ini`. Use this to set the `SUBTYPE` key in NetConfig to the correct protocol: `java/ws-client`, `java/ws-server`, or `java/ws-rawclient`.
9. Advise user to update any relative WSDL file paths in their config entries so that they are relative to the new `START_DIR` HTTP. Otherwise, absolute paths are not affected.

Notes

As the above algorithm mentions in the last step, you must update any relative WSDL file paths. This is because in 2.0 and later, the default, and recommended, `START_DIR` setting is `javadriver/`*`processname`*. This might not be the `START_DIR` that the process had in 1.x. In this case, the relative path is different.

This affects only a small number of users. CAA-WS 1.0 provides an example CXF interceptor for writing an XML declaration at the start of a SOAP Envelope on outbound messages. This was created in 1.0 with the fully qualified name:

```
com.infor.cloverleaf.gjdws.utils.WriteXMLDeclarationInterceptor
```

This was updated in version 2.0 to:

```
com.infor.cloverleaf.gjdws.interceptors.WriteXMLDeclarationInterceptor
```

This created a package in which to group all provided sample interceptors, instead of leaving them in the internal utilities package. The update tool was not written to look for this. Users who have these must manually edit their XML config file and change `utils` to `interceptors` in the package name.

Migrating IHB threads to CAA-WS

This section provides the necessary guidelines for migrating from IHB threads to CAA-WS threads.

For configuration files and WS-Security, there is no requirement for a special configuration. These files do not use WS-Security.

Differences between IBMIME and CAA-WS messages

An `ibmime` message is a length-encoded message containing the main message being transported, along with any transport headers and attachments. Through the use of the `ibmime` API, these parts of a message are extracted and processed by Tcl code.

The CAA-WS message content is the main message being transported. This requires no API methods to access that content. This can be directly sent to an xlate process or processed directly in Tcl.

The transport headers and other metadata, including any attachments, are encoded using normal Tcl keyed lists in the message's `USERDATA` attribute.

To migrate `ibmime` code to CAA-WS code with respect to the message content, API calls into the `ibmime` object must be changed. This is because they must access the same data within the message content itself or the `USERDATA` content.

This is accomplished where the IHB thread uses the standard example procs. These take the message going through Cloverleaf and wrap or unwrap it with `ibmime`. Then, it passes the message body along. Examples of standard procs are `IBEnvelopeIn`, `IBClientEnvelopeOut`, and `IBServerEnvelopeOut`.

For these cases, you can remove those procs, because the CAA-WS message content is already the main message to be passed. There is no requirement for wrapping or unwrapping.

The process is more complicated in cases using other Tcl procs.

For example, a Tcl proc accesses attachments, attachment headers, transport headers, or transport statuses. The Tcl code must be converted to access those from `USERDATA`.

WSDL files

IHB does not require that you specify WSDL files when working with a client or server.

CAA-WS, though, requires WSDL files to start client dispatches or service endpoints. If a WSDL file is not present, then one must be created.

Server URLs

For IHB, to direct the message to the appropriate thread, IHB listens on URLs. For example, `http://localhost:20210/IB/servlet/runHXML?ibsite=mysite&ibthread=mythread`.

For CAA-WS, only one thread can listen to a given port at a time.

Two IHB threads cannot be replaced with CAA-WS threads and have those CAA-WS threads respond to the same URLs on which the IB threads are listening.

Such a thread can have multiple endpoints listening on that port and an endpoint can listen on any URL. It can listen for messages on `http://localhost:20210/IB/servlet/runHXML`, but it does not do any branching based on the query string `?ibsite=mysite&ibthread=mythread`.

Configuration files

Security configuration, for example, threading configuration, takes place in different configuration files in CAA-WS than in IHB.

WS-Security

For IHB, WS-Security configuration takes place in an IHB GUI.

To set up WS-Security, CAA-WS requires manual editing of an XML configuration file or WS-Policy file generation in the GUI matching WS-Security requirements.

CAA-WS adds support for `UsernameToken` processing, `SAML`, and other technologies.

Server thread example

This example converts the IHB example server wsdlserver3 to a CAA-WS thread.

Because of the differences between IHB and CAA-WS, some changes are required.

Message differences

In this example:

- The inbound thread is "LabIn".
- The inbound Tcl proc is `parseGenerateMsg`.
- The trxID UPoC is `getSOAPActionTrxId`.
- The outbound Tcl proc is `LabResultOut`.

The `parseGenerateMsg` proc contains this code:

```
set cont [msgget $mh]
 # parse the input ib Mime message
set ih1 [ibmimecreate $cont]
 # the 1st part (Part ID = 0) content is XML message
set data [ibpartcontentget $ih1 0]
 # set the new message content as the inbound Cloverleaf message
msgset $mh $data
msgmetaset $mh USERDATA [ibmimeheaderget $ih1 soapaction]
```

This extracts the message content from the ibmime package and sets that content as the content of the Cloverleaf message. Then, it sets `USERDATA` as the `SOAP` action.

The same thing can be performed in CAA-WS by removing this Tcl proc and setting the **Cloverleaf TrxID Determination** option to be **`SOAPACTION`**.

The `getSOAPActionTrxId` proc:

- Reads the `USERDATA` to get the `SOAP` action.
- Runs it through a table lookup, to get a shorter name for the trxID.
- Sets that lookup value as the trxID.

This happens with no Tcl coding by:

- Removing these procs.
- Changing the routing to use the literal `SOAP` action values.
- Setting the option in the CAA-WS configuration to use `SOAP` action as the TrxID.

The `LabResultOut` outbound Tcl proc is:

```
set cont [msgget $mh]
 # create an empty ib Mime object
set ih1 [ibmimecreate]
 # add ib Mime headers
ibmimeheaderadd $ih1 "IBStatus" "APP_RESP"
ibmimeheaderadd $ih1 "Content-Type" "multipart/mixed"
 # create the 1st part of this ib Mime
set phl [ibpartcreate $ih1]
 # add headers of the 1st part
ibpartheaderadd $ih1 $phl "Content-Type" "text/xml"
 # enveloping message into soap.
set soapenv $cont
 # add content of the 1st part.
ibpartcontentset $ih1 $phl $soapenv
msgset $mh [ibmimeencode $ih1]
```

This functions the same as the standard Tcl proc `IBServerEnvelopeOut`.

Because this wraps the message in ibmime, you can remove this Tcl proc and send the unmodified soap envelope back to the CAA-WS Provider code.

WSDL folder and server URLs

The sample included with IHB has a `wsdl_files` folder that has a `wsdlserver3` subdirectory containing the WSDL along with related `xsd` files.

Assign that WSDL to the CAA-WS Provider configuration.

Server URLs

For changing only one thread, change the URL to `http://localhost:8080/wsdlserver3`.

Configuration files and WS-Security

There is no requirement for special configuration here, as these files do not use WS-Security.

Making the changes

Required changes are only to the LabIn thread and consist of:

- Remove those three Tcl procs from it and change the routing to use the literal SOAP action values.
- Set up the CAA-WS configuration to use the WSDL and have it use the SOAP action as the trx ID.
- Convert it to a java/ws-server protocol thread.

Removing Tcl procedures and changing routing values

On the Network Configurator's **Thread** tab, you can:

- Remove the inbound data procedures and outbound reply proc.
- Change the Transaction ID determination format to the default FRL setting.

On the Network Configurator's **Route Messages** tab, you can change the routes to use the SOAP actions. These values are found in these lines of the WSDL on the tab:

- `soap:operation soapAction="http://www.infor.com/LabResult/GetLabResult"/`
- `soap:operation soapAction="http://www.myCompany.com/MyService/GetMyOutput"/`

Converting to CAA-WS server thread

1. Create a new `LabIn` folder under the site.
2. Copy the three WSDL and XSD files from the sample's `wsdl_files/wsdlserver3` folder into the `LabIn` folder. The folder contains a WSDL and two XSD files.
3. In the Cloverleaf IDE, change the `LabIn` thread's protocol to java/ws-server and click **Properties**. This opens a blank configuration screen.

   Note: If you are using Cloverleaf 6.0.1, then you must click **OK** on the blank configuration screen. Then, you click **Apply** for the thread and reopen the **Properties** dialog box. This is due to a bug that was fixed in Cloverleaf 6.1 and later versions.
4. Create a new SOAP server using the WSDL in the `LabIn` folder and set it to `MESSAGE` mode. This is because the rest of the site is expecting a full SOAP envelope.
5. Set the address field to any URL on which it should listen. If this is the only IB server, then you can replace IB without having the client change URLs. This is accomplished by setting the URL to be `http://localhost:20201/IB/servlet/runHXML`. The query string is ignored, but if this is your only thread that does not matter.
6. Click **OK**. This returns you to the main **Properties** dialog box.

   In this example, these changes have been made to the defaults:
   - The **WSDL Location** is changed to a relative path. Relative paths start at the working directory for the thread, which is the `javadriver/`*`processname`* folder. By doing this, the site can deploy on other servers without any requirement to update the absolute path.
   - Message logging is enabled. This is for debugging.
   - **Cloverleaf TrxId Determination** is set to `SOAPACTION`. This replaces the functionality of the Tcl proc that was removed.
7. Click **OK** to close the dialog box. Then apply the changes and save your NetConfig file.

Testing the changes

1. Save any open files, such as NetConfig, and start the `labresult` thread.
2. The process should show these lines, indicating this service was started:

```
Nov 5, 2013 11:29:30 AM org.apache.cxf.service.factory.ReflectionServiceFactoryBean buildSer
viceFromWSDL
INFO: Creating Service {http://www.quovadx.com/LabResult}LabResult from WSDL:
..\..\LabIn\LabResult3.wsdl
Nov 5, 2013 11:29:30 AM org.apache.cxf.endpoint.ServerImpl initDestination
INFO: Setting the server's publish address to be http://localhost:8080/wsdlserver3
```

3. With the service running, specify this in a browser to get a WSDL: `http://locahost:8080/wsdlserver3?wsdl`.
   - A WSDL file for this service displays.
   - There is also a log entry for the request in the Cloverleaf log.
4. Use a SOAP test client, for example, SOAP UI, to consume the WSDL to verify the service is working.

5 After changing the URL to `http://127.0.0.1:8080/wsdlserver3`, sending a message and getting a response is successful using the same client that worked for the wsdlserver3 site. This indicates the migration was successful. This is the best method for testing with a client that was already configured for the site you are migrating. This is equivalent to setting up a new SOAP UI Client using the WSDL by one of these methods:

- Directly loading it from the file system and setting the URL manually in the Client. This avoids any display of CXF changing the WSDL to work with itself.
- Have the SOAP UI generate a client by pointing at `http://locahost:8080/wsdlserver3?wsdl`. This automatically populates the service URL with the correct address.

CAA-Direct

CAA-Direct is an extension to the core system functionality.

With the Generic Java driver, the engine can associate many threads with the Java protocol. This provides a way for the engine to communicate by a public supported API with Java applications running in one or more JVMs.

This foundation provides a platform for you to build custom Java applications that extend the core system engine.

This is built using the Java Driver, where CAA-Direct provides:

- Support for the prevailing Direct client paradigms, including:
  - Sending secure emails by SMTPS to a HISP.
  - Retrieving secure emails from a HISP using POP3S or IMAPS.
- Customization points for:
  - Using Tcl UPoCs to process inbound emails and create outbound ones.
  - Passing the message body as a string to/from Tcl UPoCs.
  - Passing metadata as a keyed list to/from Tcl.
  - Developing a custom adapter for power users.
- GUI tools for help with deployment configuration for sending/retrieving emails from a HISP.
- Tutorials and sample sites.

Knowledge levels

CAA-Direct is essentially a Java mail application, and is usually configured using Spring Framework XML files. This is beyond the Generic Java driver configuration that is required in NetConfig. Some GUIs do not support a specialty Java mail configuration that you require. In these cases, you can configure a variety of Java mail behaviors from an XML file instead of writing Java code.

Users with a high level knowledge of Java mail can use the GUI to understand what beans are created. Then, the XML files can be modified with Java mail properties as required.

Users starting with Java mail can use the GUI until special Java mail properties customization is required that this tool does not support. When this happens, use the Java mail documentation to modify your XML configuration files. This is not required for users connecting to Direct conforming HISPs.

CAA-Direct architecture and flow

Similar to CAA-WS, CAA-Direct:

- Runs on a Java Driver foundation.
- Uses a Spring XML file for configuration.
- Uses USERDATA to provide inbound message metadata and outbound message overrides.
- Uses a well-known library to handle inbound/outbound message transmission. It uses Java Mail instead of CAA-WS's CXF.

This overlap assists CAA-WS users in adapting to CAA-Direct.

SMTP, POP3, and IMAP protocol handling are reliable with Java Mail.

Sample site

The sample site, containing TLS and non-TLS examples of POP3 and SMTP clients, is the starting point for learning how CAA-Direct works. You can then modify the samples to create your own applications. See CAA-WS sample sites.

Knowledge prerequisites

- Normal users follow samples and configuration guidelines. They can apply business logic in typical system application development methodologies using Tcl API in UPoCs to employ existing system functionality. These users should have a general understanding of the concepts behind POP3 and SMTP email protocols. They should also know hot to use Java keystores/truststores to establish secured TCP connections.
- Power users are those who must customize processing at the email protocol level. These users are comfortable programming in both Tcl and Java. They have a deep understanding of how certain open source Java email technology works. For example, the Mail API.

POP3/IMAP email retrieval usage

This shows the application as a POP3, or IMAP, client that interacts with an external email server/HISP.

The POP3/IMAP client can interact with an external email server/HISP.

An email retrieval request is processed using the Java Driver thread to route, based on trxID, to two outbound threads. The outbound threads are where the Tcl UPoCs apply the business logic.

This logic can also reside in an inbound UPoC within the Java Driver thread without involving additional outbound threads.

POP3/IMAP retrieves email instead of receiving it. It is not strictly a server, as messages are not sent to it from the outside world. It calls the configured email server periodically to request download of any new emails. It receives messages from the outside world and sends them into the system as a normal inbound thread.

Due to this inbound nature, you can think of it as a server thread. This type of thread is referred to as an "Email Retriever" thread.

CAA-Direct differs from CAA-WS in that there are no reply messages sent back to inbound system threads. For POP3/IMAP, an inbound message on the CAA-Direct thread is only sent to the outbound threads. There is no reply.

Source code is available for power users to modify the POP3/IMAP client if a special case arises. For example, a situation that cannot be addressed by configuration changes.

SMTP email sending usage

An SMTP client can send emails to an external mail server/HISP.

An inbound thread can create a system message to send by email to a HISP.

CAA-Direct differs from CAA-WS in that there are no reply messages sent back to inbound system threads. For SMTP, an outbound message on the CAA-Direct thread is sent to the mail server. After it receives an acknowledgment, it knows it has sent the email and is finished. The ack is not returned back to the inbound thread.

Source code is available for power users to modify the SMTP client if a special case arises. For example, a situation that cannot be addressed by configuration changes.

CAA-Direct Application Programming Interface (API)

This topic assumes you are familiar with the system Tcl UPoC development.

The Tcl user interface is intended for implementers using a Tcl UPoC to process messages coming into the system from CAA-Direct. Implementers can then create messages going outbound through CAA-Direct.

The Java mail API is for users who require custom behavior other than the CAA-Direct bundled POP3/SMTP clients. The source code for the CAA-Direct clients is part of the distribution as samples.

Note: If you develop and configure with Java Mail custom clients, then the Tcl User Interface might not apply. This is because that interface is based on the CAA-Direct bundled clients.

The Spring XML configuration interface is an XML configuration file that determines the email client behavior. You can edit the file using the GUI tool within NetConfig or manually edit the file.

CAA-Direct USERDATA for getting information and setting overrides

The USERDATA field in the system messages sent to and from CAA-Direct contains information that you can use to set overrides.

For emails, ignore this field if the default settings provide what is required.

More complex Direct processing, for example, attachments or custom email headers, requires one of these:

- Reading the USERDATA on an inbound message and implementing different handling logic in UPoC code, depending on the contents.
- Writing to the USERDATA on an outbound message to set specific email header fields, add attachments, and others.

See USERDATA format.

SMTP versus POP3 and IMAP

These different modes have commonalities. For example, headers are present as information on inbound messages from POP3, or IMAP, and can be used as overrides on outbound SMTP.

POP3/IMAP inbound information

This table lists the different fields, or keys, for inbound POP3/IMAP.

An asterisk (*) in the list is the wildcard that represents any other key names not listed in the same location of the map.

The "-" prefix represents a sub-layer. This indicates the key that follows belongs to a sub-level map under the nearest above key that contains one less dash in the prefix.

Field (key)	Description
headers	This contains a map of all the raw email headers.
-*	The email headers are named by the sender. These are items such as Return-Path, To/Subject, and other common values. They can also contain any arbitrary value sent by the sender. The header name is the key and the header value is the value in the map. Headers with multiple values can have the same key name repeated when you use a key suffix with :: and an incremental number.
-content-type	Content-Type is different from other email headers. • For single part emails, this is the email header. • For multi-part emails, this is the Content-Type taken from the part that is used as the Cloverleaf message. This is how Cloverleaf handles messages when they are forwarded to the outbound side. The global level Content-Type is controlled by other keys.
generalInfo	This contains a map of fields giving parsed information about the email message.
-to	This is the "To" address.
-cc	This is the "CC" address.
-from	This is the "From" address.
-replyTo	This is the "ReplyTo" address.
-subject	This is the "Email" subject.
-sentDate	This is present if a sent date was supplied.
payloadHeaders	A informative map that contains multi-part headers selected for the Cloverleaf message. This displays only when the email is multi-part.
attachments	This is a map of all attachments in the message.

Field (key)	Description
`-**`	Each attachment is a nested map of its components, where the key is the identifier for the attachment. If the attachment already has a designated `identifierID`, then that ID is used. For example, an inline-type attachment must have a Content-ID header. An attachment-type attachment must have an associated file name. If there is no existing ID, then one is automatically generated as an index number within the array of attachments. Attachments can exist within attachments in an email. This mechanism breaks them down into a flat array of attachments to avoid having to parse MIME structure in the system. The flatting order follows the DFS algorithm. The first part that is located without a disposition is taken as a Cloverleaf message.
`--disposition`	This is the multi-part type that is associated with the current attachment. This can be inline, attachment, or null.
`--headers`	This is a further nested map containing the headers for the attachment.
`---*`	The header names are the keys in this map, with the header values being the values in the map. This is similar to the top-level headers.
`--content-type`	This is the same value as `-content-type` listed under `headers` (above). When forwarded to the outbound side, this value is loaded by an SMTP protocol thread. **Note:** Some headers might require regeneration if the attachment's `headers` map does not function.
`--content`	This is a base64 encoded string containing the attachment's content. This is used if the `cloverleafAttachmentDirectory` field is missing or blank in the configuration XML. This is not populated if you specify an attachment directory in the GUI.

Field (key)	Description
`--contentFile`	`cloverleafAttachmentDirectory` is populated if you specify an attachment directory in the GUI. If this field exists in the configuration XML, then it is sent instead of the base64 encoded "content" field. This field contains the file name that holds the attachment content. This is relative to the `cloverleafAttachmentDirectory` that is specified in the configuration. The system application deletes this file after it has been used. This is useful for improving performance relative to base64 encoding in the case of large attachments.

SMTP outbound overrides

In the outbound message from the Client thread, `USERDATA` that is passed in overrides default behaviors and values.

An asterisk (`*`) in the list is the wildcard that represents any other key names not listed in the same location of the map.

The "`-`" prefix represents a sub-layer. This indicates the key that follows belongs to a sub-level map under the nearest above key that contains one less dash in the prefix.

Field (key)	Description
`headers`	This contains a map of all the raw email headers to override.
`-*`	The email headers are named by the sender. These are items such as Return-Path, To/Subject, and other common values. They can also contain any arbitrary value sent by the sender. The header name is the key and the header value is the value in the map. Headers with multiple values can have the same key name repeated when you use a key suffix with "`::`" and an incremental number.

Field (key)	Description
`-content-type`	This impacts SMTP outbound behaviors. The semantics follow RFC 7231, section 3.1.1.5: Content-Type. The "text/plain" and "text/html" types, and the absence of Content-Type, causes CAADIRECT to take the Cloverleaf message as text. Other explicit settings cause Cloverleaf to take messages as binary. When `charset` is appended, text types transfer the online content to the assigned UTF-8 encoding. This is a Cloverleaf internal encoding. Cloverleaf does not do binary encoding. Regardless of whether `charset` appears in Content-Type, the message processing is a user-defined procedure. This Content-Type is the Cloverleaf message type. It is not the outbound MIME message's root element type. As long as the Cloverleaf message is not the same as the root element, the MIME message is multi-part. The multi-part message's type is decided by the `multipartMode` key.
`multipartMode`	This is the method used to construct a MIME message. This is composed of a Cloverleaf message and its attachments. The allowed values include `mixed`, `related`, and `mixed/related` (default). For details, see MULTIPART_MODE_MIXED_RELATED.
`sender`	This contains the configuration for connecting to a specific server with a specific user name. Typically a sender is configured in the GUI, and thus to the underlying XML config file. • If there is only one sender, then it is not necessary to pass this, as that sender is used by default. • If there are no senders in the config, then you should pass a complete sender here. Parameters can be passed as necessary to override existing values. • When no senders are present, all parameters are required.
`-id`	For configurations with multiple senders, this uniquely identifies which one you require to use for this message.
`-host`	Overrides the default host in the sender, or sets one if there was no sender.
`-port`	Overrides the default port in the sender, or sets one if there was no sender.

Field (key)	Description
-username	Overrides the default user name in the sender, or sets one if there was no sender.
-password	Overrides the default password in the sender, or sets one if there was no sender.
-socketFactory	Overrides the specified SSL Socket Factory, or sets one if no sender or SSL Socket Factory was specified in the sender. The SSL Socket Factory is identified by the ID given to it on the GUI. This is useful for switching SSL configurations on a per-message basis.
helper	This contains the configuration for setting typical email parameters such as to/from/subject/and so on. Typically, a helper is configured in the GUI, and thus to the underlying XML config file. • If there is only one helper, then it is not necessary to pass this, as that helper is used by default. • If there are no helpers in the configuration, then you should pass a complete helper here. Some or all of the parameters can be passed as required when overriding the contents of an existing helper. • If there is no helper, then you must have a minimum of a To address.
-to	Normal email To address.
-cc	Normal email CC address.
-bcc	Normal email BCC address.
-from	Normal email From address.
-replyTo	This is the same as normal emails. It is the address to use when someone replies to you if the From address should not be used.
-subject	Normal email subject.
-priority	This is an integer priority value to indicate to the receiver what you feel is the priority on this message.
-validateAddresses	Boolean value to validate that addresses at least conform to the legal structure of an email address. That is, when it has an "@" sign, no illegal characters, and so on. This does not validate that the email address is a working in-box.
attachments	Map of all attachments to be sent.

Field (key)	Description
`-*`	Each attachment is itself a nested map of its components where the key is the identifier for the attachment. The specified value is used as the file name for an attachment or the Content-ID for an inline in the final outbound email message. Except for Content-Type, the other attachment headers or inline are not open for arbitrary editing due to the library implementation
`--disposition`	This value decides if the attachment element (above) is an attachment or an inline.
`--contentType`	This assigns the Content-Type of the element. The default is `application/octet-stream`. Alternatively, the key name can also be `content-type` (case-insensitive) as a shortcut for messages that are forwarded from an inbound POP3/IMAP protocol.
`--content`	Base64 encodes your attachment content to be sent here. This field or `contentFile` is required.
`--contentFile`	The file name for a file containing the attachment content. This is used for large attachments to improve performance versus the costs of base64 encoding. The file name is a relative path to the system process directory or an absolute path. This field or `content` field is required.

CAA-Direct IDE Properties GUI

The CAA-Direct protocols are configurable similar to other protocols. You select the protocol and use the dialog box that displays by clicking **Properties** to configure it. To keep the changes, click **Apply** and save the NetConfig. This is the same for CAA-Direct as for other protocols.

For the CAA-Direct protocols, versus other protocols, do not put two separate Java Driver threads in the same process. CAA-Direct is based on Java Driver. Sometimes, this can work with Java Driver, but with CAA-Direct creates issues. They must be in separate processes. They can share with other protocols, but not other CAA-Direct threads, or another Java Driver, such as CAA-WS.

Note: Do not change the Java Driver process working directory. When you create one of these product's threads in a given process, the working directory defaults to `$SITEPATH/javadriver/process name`. This should not be changed as it causes errors. As all file relative paths are computed automatically when you use the various **Browse** buttons to select files, this should not have a negative effect.
Note: Asterisks indicate required fields; all others are optional.

CAA-Direct is built upon the Java Driver. When CAA-Direct is installed, new protocols that are based upon Java Driver become available in your IDE.

In the Network Configurator, **Protocol** lists the CAA-Direct installed protocols.

- When **`java/direct-sender`** is selected, you can create SMTP configurations.
- When **`java/direct-retriever`** is selected, you can create POP3 or IMAP configurations.

These dialog boxes automatically configure Java Driver and create the XML configuration files that configure the CAA-Direct components.

Creating a sample sender

1. Select the**`java/direct-sender`** protocol and click **Properties**. This opens a blank configuration screen with buttons for adding sender configuration entries.
2. Click **New SMTP Sender** to create a new blank SMTP configuration entry.

Sender object's sample site "GreenMailServer" test server

This is an email server that runs in memory. When it gets shut down, all messages are destroyed.

SMTPS port is 3465. All ports are standard plus 3000. For example, SMTPS is 465+3000.

Sample user names/passwords are configurable. This example uses the default cloverleaf/gofish.

A “helper” is also present, including an SSL configuration.

A sender object’s purpose is to define all the configuration information necessary to make a connection to an SMTP server and send email to it.

- If only one is present in a configuration, then it is automatically used.
- If multiple are present, then USERDATA must be sent to identify which sender to use for the current message.

This configuration specifies the **Host**, **Port**, **Username**, and **Password** are required to connect to the SMTP server and have email sent to it.

Use SSL indicates whether to use SSL. If it is selected and there is no **SSL Config**, then the default Java settings are used to make the connection.

In this example, the SSL configuration is added later, so that you can return and specify that configuration.

In rare cases, the client is supposed to connect to an unencrypted SMTP port and then issue the STARTTLS command. This negotiates a secure connection for the ensuing conversation. This is chosen with **Use StartTLS**. It can also use **SSL Config** for configuring a truststore, and perhaps a keystore.

Sender's logical view

The properties of this Sender are shown on the right panel, where they can be specified or selected.

Pausing over a property opens a tooltip explaining the property.

This type of view is referred to as a logical view. This is because the underlying configuration can be made up of multiple components with various structures. The view displays them as one set of fields.

For a list of logical view objects and their descriptions, see Logical items and their fields.

Configuration entries, for example, the sender, can be deleted by right-clicking the tab and selecting **delete this tab**.

Additional sender configuration items

In addition to **New SMTP Sender**, **New Email Helper**, and **New SSL Socket Factory** are also available.

- **New Email Helper**

  An email helper is a configuration object which stores commonly used email fields. This can be used to define the To, From, cc, bcc, Subject, ReplyTo, and main body text of an email. This saves you from setting these fields in the USERDATA overrides.

  - If only one helper is present, then it is used by default when sending email.
  - If multiple helpers are present, then USERDATA must be sent to distinguish which helper to use.

- If no helpers are present, then `USERDATA` must be sent to define all the email fields to set. To, From, and Subject are a minimum. As To/From/Subject are typically used for an email, the application sends it with only a To address. This creates a From address for you. For example, if a From address is not specified, then it uses the logged in username. For Linux, this is `hci@yourmachine.com`. Windows would use `hciuser@yourmachine.com`.

- **New SSL Socket Factory**

  This defines the parameters necessary to create secure socket connections to a mail server. This is common to senders and retrievers.

  The minimum requirements for SSL Socket Factory are these entries:

  - **Id** is used to define a unique name within the thread for the SSL Socket Factory. This ID is referenced by the SMTP (POP3 or IMAP, in case of retriever) instances to specify their security connection. The list displays in the SSL Config boxes on the SMTP, POP3, and IMAP configuration tabs.
  - **Truststore File** is the path to the truststore file, relative to the process directory. An absolute path also works.
  - **Truststore Password** is the password to unlock the truststore.

Other fields are explained in Logical items and their fields.

For the sample sender, connecting to the test server requires the provided truststore. This is available in the direct_samples site.

To finish the sample client, return to the **SMTPS** tab and specify this SSL configuration entry. If **Use SSL** is not selected, then the tab is named **SMTP**. The sample client can now make a secure SSL connection to the local test server. This is part of the sample site. A message sent to this thread is sent as an email to the To address given in the helper configuration.

Creating a sample retriever

This is similar to a POP3S client, which is secure POP3. The procedure is the same for an IMAPS client, except that you click **New IMAP Retriever** to start instead. Asterisks indicate required fields, all others are optional.

1. Select the`java/direct-retriever` protocol and click **Properties**. This opens a dialog box with buttons to add retriever configuration entries.
2. Click **New POP3 Retriever** to create a new blank POP3 config entry.

Retriever object's sample site GreenMailServer test server

This is an email server that runs in memory. When it gets shut down, all messages are destroyed.

The POP3S port is 3995. All ports are standard plus 3000. For example, POP3S is 995+3000. IMAP is 3993.

Sample user names/passwords are configurable, but this example uses the default drsmith/doctor. This is distinct from the cloverleaf/gofish account that is used for SMTP.

An **SSL Config** is also present.

A retriever object's purpose is to define all the configuration information necessary to make a connection to a POP3 server. It also retrieves email for a specific user account. Email retrieval happens as often as is specified in the **Retrieve interval**field. If this is blank, then the default is 30 seconds. You can have multiple configurations. If this is the case, then they are all processed on every retrieve interval.

- This configuration specifies the **Host**, **Port**, **Username**, and **Password** for connecting to the POP3 server and retriever email for that account.
- The **Use SSL** check box determines whether to use SSL. If this is selected and there is no SSL Config, then the default Java settings are used to make the connection. In this example, the SSL configuration is added later, and then you can return and specify that configuration.
- Because you can have multiple configurations, it is useful to set the TrxId based on a field so that you can route messages appropriately. This sample uses the **From** email field as an example.
- If an attachment directory is specified, then email attachments are written into this directory instead of as base64 encoded data in the `USERDATA`.
- The **Cloverleaf Log Exceptions** setting is enabled. This is useful to know if some POP3 connections are failing and why. For example, if you have multiple connections or a server is regularly offline, then it is better to set this value to `false`. By doing this, the log is not filled with expected exceptions.

Retriever's logical view

The properties of this retriever are shown on the right panel, where they can be specified or selected.

- Pausing over a property opens a tooltip explaining the property.
- This type of view is referred to as a logical view. This is because the underlying configuration can be made up of multiple components with various structures. The view displays them as one set of fields.

Configuration entries, such as the retriever, can be deleted by right-clicking the tab and selecting **delete this tab**.

Additional retriever configuration items

Other retriever items are:

- **Retrieve Interval**

  This is how often the POP3, or IMAP, configurations are polled for new email messages. If cleared, then it defaults to every 30 seconds.

- **New SSL Socket Factory**

  This defines the parameters necessary to create secure socket connections to a mail server. This is common to senders and retrievers.

  At a minimum, the SSL Socket Factory requires these entries:

  - **Id** is used to define a unique name within the thread for the SSL Socket Factory. The ID is referenced by the POP3 or IMAP (or SMTP, in case of sender) instances to specify their security connection. The list shows up in **SSL Config** on the SMTP, POP3, and IMAP configuration tabs.

- **Truststore File** is the path to the truststore file, relative to the process directory. An absolute path also works.
- **Truststore Password** is the password to unlock the truststore.

Other fields are explained in Logical items and their fields.

For this sample, connecting to the test server requires the provided truststore. This is available in the `direct_samples` site.

To finish the sample client, return to the **POP3S** tab. Then, specify the **SSL Config**entry. The tab is named POP3 if the **Use SSL** check box is not selected.

The sample client can now make a secure SSL connection to the local test server. This is part of the sample site. Depending on the retrieve interval setting, emails for the drsmith user are retrieved. These emails are converted into system messages and sent into the engine.

Logical items and their fields

This table shows the sender SMTP items:

Item	Field	Description
Sender		
	Id	Uniquely identifies this sender. Shown on display under the tab name. Used in `USERDATA` sender parameter to select the sender to use if there is more than one present. Can be left blank if there is only one sender.
	Host	DNS name or IP address of server to which to connect.
	Port	Port on the server to which to connect.
	User name	The email account with which to authenticate.

Item	Field	Description
	Password Encoded	If selected, then the password you specify here must be the encoded version. To find the encoded version, start the process with an unencoded version. One of the first log entries that is printed out shows the encoded value. Replace your password here with that encoded value and check the box. This stores the encoded password in the XML config file, and not in the clear text password.
	Use SSL	If selected, then the connection made to the server is performed using SSL.
	Use StartTLS	If selected, then the connection to the server is initially made using unsecure SMTP. After the connection is made, it issues the `STARTTLS` command. This converts the connection to a secure connection. It errors out if this command is unavailable or does not succeed. This is a rare configuration possibility. Use this only if there is no normal SSL/TLS port with which to connect and if the SMTP port supports it.
	- **SSL Config**	The name of the SSL socket factory entry to use to make the SSL connection. The menu is populated with the list of SSL socket factory entries created. If left blank, then Java default SSL parameters are used.
	Mail Debug	If set to "normal debug," then this prints the email protocol conversation with the server. This is useful for debugging with a server that is not working properly. If set to "normal+auth debug," then the authentication portion of the conversation is also displayed. User name/password is hidden when set to normal debug.

Item	Field	Description
Helper		
	Id	Uniquely identifies this helper. Shown on display under the tab name. Used in USERDATA helper parameter to select the helper to use if there is more than one present. This can be left blank if there is only one helper.
	To	The normal email To address. This can have multiple values when separated by a comma.
	CC	The normal email CC address. This can have multiple values when separated by a comma.
	BCC	The normal email BCC address. This can have multiple values when separated by a comma.
	From	The normal email From address. Single valued.
	Reply To	The normal email Reply To address. Single valued.
	Subject	Text string for the email subject
	Text	The default email message body. This is used if the outbound message is blank. It is useful to say things such as "see attachment" when the important part of a message is always in an attachment.
	Priority	Integer priority value from 1 to 20 to indicate to the recipient the importance of this message.
	Validate Addresses	If selected, then the email addresses that are used in the message are checked to see if they conform to the email RFC format. For example, someone@somewhere.domain. It does not validate whether the address is a valid mail box, as this is impossible.

Item	Field	Description
SSL Socket Factory		
	Id	Required identifier. Must be unique within the thread configuration. Used to provide an identifier in the SSL Config menu in the SMTP config.
	Secure Socket Protocol	The preferred level of SSL/TLS to use when negotiating the secured socket connection. Some earlier email servers require SSLv3. Use this first if your SSL connection is failing with an obscure error message. **Note:** To use TLSv1.1 or TLSv1.2, the process must be running within a Java 7 JVM. Cloverleaf 6.1 and above already run Java 7.
	Key Password	Password for keys within the keystore.
	Keystore File	Relative path to the keystore from the process directory. This can also be an absolute path.
	Keystore Password	Password for the entire keystore.
	Keystore Type	Combo box containing **JKS**, **JCEKS**, **PKCS12**.
	Truststore File	Relative path to the truststore from the process directory. This can also be an absolute path.
	Truststore Password	Password for the truststore.
	Truststore Type	Combo box containing **JKS**, **JCEKS**, **PKCS12**.

This table shows retriever POP3/IMAP items:

Item	Field	Description
Retriever		
	Id	Uniquely identifies this retriever. Shown on display under the tab name. Optional field for GUI organization use only.

Item	Field	Description
	Host	DNS name or IP address of server to which to connect.
	Port	Port on the server to which to connect.
	User name	The email account with which to authenticate.
	Password Encoded	If selected, then the password you specify here must be the encoded version. To find the encoded version, start the process with an unencoded version. One of the first log entries that is printed out shows the encoded value. Replace your password here with that encoded value and check the box. This stores the encoded password in the XML config file, and not in the clear text password.
	Use SSL	If selected, then connection made to server is performed using SSL.
	- **SSL Config**	The name of the SSL socket factory entry to use to make the SSL connection. The menu is populated with the list of SSL socket factory entries created. If left blank, then Java default SSL parameters are used.
	Cloverleaf TrxId Determination	Combo box containing **NULL**, **TO**, **FROM**, **SUBJECT**, **VALUE**. **NULL** means do not set one. **VALUE** means use the value from the **Cloverleaf TrxId Value**field. The rest are the values from the email field of the same name.
	Cloverleaf TrxId Value	If **Cloverleaf TrxId Determination** is set to **VALUE**, then this value is used as the TrxId. This is useful if you have multiple POP3, or IMAP, accounts in the same configuration, so messages can be routed differently based on account.

Item	Field	Description
	Cloverleaf Attachment Directory	Relative or absolute path to store inbound attachments. If this is present, then attachments are stored in this directory as files. If left blank, then attachments are inside the `USERDATA` as base64 encoded content.
	Cloverleaf Log Exceptions	Should the system log exceptions or not. The default is "false." This is useful to know if some POP3 connections are failing, and why. If you have multiple connections or a server is regularly offline, then set this value to "false". By doing this, the log is not filled with expected exceptions.
	Mail Debug	If set to `normal debug`, then this prints the email protocol conversation with the server. This is useful for debugging with a server that is not working properly. If set to `normal+auth debug`, then the authentication portion of the conversation is also displayed. User name/password is hidden when set to normal debug.
SSL Socket Factory		
	Id	Required identifier. Must be unique within the thread configuration. Used to provide an identifier in the SSL Config menu in the POP3 or IMAP config.
	Secure Socket Protocol	The preferred level of SSL/TLS to use when negotiating the secured socket connection. Some earlier email servers require SSLv3. Use this first if your SSL connection is failing with an obscure error message. **Note:** To use TLSv1.1 or TLSv1.2, the process must be running within a Java 7 JVM.

Item	Field	Description
	Key Password	Password for keys within the key-store.
	Keystore File	Relative path to the keystore from the process directory. This can also be an absolute path.
	Keystore Password	Password for the entire keystore.
	Keystore Type	Combo box containing **JKS**, **JCEKS**, **PKCS12**.
	Truststore File	Relative path to the truststore from the process directory. This can also be an absolute path.
	Truststore Password	Password for the truststore.
	Truststore Type	Combo box containing **JKS**, **JCEKS**,**PKCS12**.

CAA-Direct usage scenario

This section describes the different flows that you can use to develop applications that run on CAA-Direct.

Basic flow

When building an application using CAA-Direct, the user's main tasks are:

1. Create a system site and add a CAA-Direct `java/direct-*` protocol thread.
2. Gather the necessary artifacts to configure your service or client. These could be keystores/truststores for doing secure communications, host/port for your HISP (email server), user name/password for email account to use, and so on.
3. Use the **Properties** dialog box to configure your thread with the artifacts that were collected in the previous step.
4. Create the Tcl UPoCs that have the business logic of processing inbound email from a POP3, or IMAP, client. You can also create Tcl UPoCs to generate outbound email content destined for an SMTP client.

Tcl UPoCs are created to process messages using the API. See CAA-Direct Application Programming Interface (API).

Alternate flows

These show how you can build Direct senders or retrievers.

- Simple Message Sender
- Message Sender with an attachment
- Simple message retriever
- Message retriever for an attachment

Simple Message Sender

This creates an SMTP client that sends the message you pass it.

1. Select the `java/direct-sender` protocol.
2. Locate the host/port information for your HISP. If the port is not mentioned, then leave it blank when creating the sender and use the default.
3. Configure a **Sender**, **Helper**, and **SSLSocket Factory**.

4 Create an inbound thread to send this message. It is not necessary to write any Tcl if the message can be sent directly as the email message body.

Message Sender with an attachment

This is the same as the "Simple Message Sender", except that it sends the message as an attachment.

These changes cause the message to be sent as an attachment instead of the email main body.

1 Select the **`java/direct-sender`** protocol. Add a default text message to the Helper, such as `see attachment`.
2 Locate the host/port information for your HISP. If the port is not mentioned, then leave it blank when creating the sender and use the default.
3 Configure a Sender, Helper, and SSLSocket Factory.
4 Create a UPoC that reads the outbound message, converts it to base64, and adds it into the `USERDATA` as an attachment. Have the UPoC also set the message content to blank so that the default message that is configured in the Helper is used.

Simple message retriever

This is the same as Basic flow, but creates a POP3 client for retrieving messages with no attachments.

See CAA-Direct usage scenario.

Creating IMAP is the same, except the retriever that is configured is IMAP, instead of POP3.

1 Select the **`java/direct-retriever`** protocol.
2 Locate the host/port information for your HISP. If the port is not mentioned, then leave it blank when creating the sender and use the default.
3 Configure a **Retriever** and an **SSL Socket Factory**.
4 Route the messages to another thread for processing. The inbound email message is the Cloverleaf message content. It is not necessary to write any Tcl procs.

Message retriever for an attachment

This expects the interesting part of the message to be in an attachment.

1 Select the **`java/direct-retriever`** protocol.
2 Locate the host/port information for your HISP. If the port is not mentioned, then leave it blank when creating the sender and use the default.
3 Configure a **Retriever** and an **SSL Socket Factory**.

4 Create a UPoC that processes the incoming message and looks in the USERDATA for the attachments key. In the case of receiving multiple attachments, locate the appropriate one and then process. If no attachment directory is set, then the attachment is base64 encoded. A procedure can be to base64 decode the attachment content. Then, set that value as the system message, overwriting the unimportant email message body such as `see attachment`.

CAA-Direct sample sites

Sample sites are “best practice” tools that are self-contained units to help you understand how to use various parts of the functionality. This section describes the details of these sites and how they are configured.

You can gain the most knowledge by reviewing this documentation alongside the sample’s configuration in the Network Configurator. Take note of these parts of the sample sites:

- XML files that are created for the sample sites in the `javadriver` subdirectory
- Surrounding configuration in the NetConfig file
- Relevant Tcl procs

This, combined with running the samples and observing the logs produced, provides a solid basis for starting real projects, testing, and debugging.

Samples are provided in a single sample site called direct_samples.

This sample site contains examples of SMTP, SMTPS, POP3, POP3S, and IMAPS threads.

In the sample site layout:

- Inbound threads are on the left.
- Outbound threads are on the right.
- The CAA-Direct threads all connect to the GreenMailServer test server to send and retrieve messages.

SMTP

SMTP is used to send emails. SMTPS is used to send emails securely. Use this when using CAA-Direct to connect to a Direct HISP. A request can be one or both of:

- A message sent in the email message body.
- The message can be an attachment or even multiple attachments.

In both the SMTP and SMTPS samples, there is an `attachmentSource` file which is used as the content of a single attachment. The message sent to the inbound file protocol thread is the email message body. The SMTP sample additionally shows how to use `USERDATA` to override the default To address for the email.

The `SMTPfiles` thread's **Inbound** tab has these parameters:

- **TPS Inbound Data**: **`addRequestAttachment`**
- **Trx ID Determination Format**: **`Fixed Record Layout (frl)`**
- **EDI Batch**: **`None`**

This is a Fileset-local protocol thread which watches a directory for a file to show up and then sends the file to the SMTP thread.

The addRequestAttachment Tcl proc is a proc that adds an attachment to the USERDATA, and continues the message. At the top of the file is a switch to configure the attachment to be stored as a file and read by CAA-Direct. It can also be passed directly in USERDATA as base64 encoded content.

Using base64 is acceptable for smaller attachments, but the CPU and memory overhead for larger files is not good for performance. It is recommended to use the file approach.

Using the file mechanism runs these relevant lines:

```
file copy attachmentSource attachmentSourceTemp
keylset myattachment contentFile attachmentSourceTemp
keylset attachments myattachment $myattachment
keylset userDataRequest attachments $attachments
msgmetaset $mh USERDATA $userDataRequest
```

This performs these tasks:

- Copies the attachmentSource test file to attachmentSourceTemp. This is because when CAA-Direct sends the email, it deletes the file used as the attachment. Because you require the sample to run repeatedly, first make a copy of the attachment source file.
- Creates a single attachment named myattachment with the contentFile key set to the string attach mentSourceTemp. When CAA-Direct sees this attachment, it reads attachmentSourceTemp to get the attachment bytes and then deletes that file.
- Adds this single attachment to a keyed list of attachments under the name myattachment. If you have multiple attachments, then add them into this attachments variable under different keys. The names make no difference, other than they are what is used in the attachment header values as the name of the attachment.
- Adds this keyed list of attachments to a keyed list to be sent as the entire USERDATA. If you have other USERDATA overrides, then those can be added to this userDataRequest variable. For example, changing the list of **To** email addresses.
- Sets this newly created keyed list variable as the message's actual USERDATA value.

The SMTP thread's protocol properties are:

- **Id**: **exampleMessageHelper**
- **From**: **cloverleaf@localhost.com**
- **Subject**: **test CAA-Direct message**
- **Text**: **This is the default message text, if Cloverleaf sends nothing.**

All other fields are blank.

The Helper does not have a default **To** address. This means the Tcl proc must set USERDATA to specify the **To** address. The **SMTP** tab has the configuration necessary to connect to the test server.

The **Outbound** tab's properties are:

- **TPS Outbound Data**: **updateOutboundMessage dumpMsg**
- **Retries**: **-1**
- **Interval**: **10**

The `updateOutboundMessage` proc is an example of how to use `USERDATA` to override the outbound email's **To** address. These are the relevant lines of Tcl:

```
set userData [msgmetaget $mh USERDATA]
keylset userData helper.to drsmith@localhost.com
msgmetaset $mh USERDATA $userData
```

This results in these tasks:

- Get the current value of `USERDATA` so it can be added to instead of overwritten.
- Create a `helper.to key` that sets the email **To** address to be `drsmith@localhost.com`. This shorthand creates a `helper` list with a `to` key under that.
- Set the message's `USERDATA` to be this new updated value.

The `dumpMsg` proc is frequently used in the samples. It dumps the message and then continues the message. This is useful for showing the message and `USERDATA` that are going to the CAA-Direct code. This can help you understand the details of how sending an email works.

The sample site sends a message when a text file is copied that must be sent as the email message body into the process's `request` directory. You can look at the log file to see the logs created.

SMTPS

The SMTPS example is the same as the SMTP example except for these points:

- It does not have a Tcl proc to update the To address. Instead, that is configured statically in the Helper config. The attachment handling is the same.
- It has SSL configured to send the message securely to the mail server.

Selecting **SSL (sslSocketFactory)**: has these parameters:

- **Id**: **`sslSocketFactory`**
- **Secure Socket Protocol**: This has SSL configured to **`SSLv3`**
- **Truststore File***:**`../../../javadriver/testdirectkeystore.jks`**
- **Truststore Password***: **`changeit`**
- **Truststore Type**: **`JKS`**

It has a truststore to authenticate the server, but no keystore because mutual authentication is not typically required. A user name/password is required to access the email account.

Selecting **SMTPS (exampleSSLSender)** has these parameters:

- **Id**: **`exampleSSLSender`**
- **Host***: **`localhost`**
- **Port**: **`3465`**
- **Username***: **`cloverleaf`**
- **Password***/**Confirm Password***:These must be specified.
- **Use SSL**: This is selected.
- **SSL Config**: **`sslSocketFactory`**

- **Mail Debug**: `disabled`

This is almost the same as the SMTP thread's SMTP tab. The exception is that the port is different, being set to the test server's SMTPS port. Another exception is that **Use SSL** is selected with **SSL Config** that is set to the name of the SSL Socket Factory from the previous tab. This is how an SMTP sender links to a specific SSL configuration.

Another difference from the SMTP sample is that the **Mail Debug** setting is disabled, indicating the conversation with the email server is not logged.

Selecting **Helper ()** has these parameters:

- **To**: `drsmith@localhost.com`
- **From**: `cloverleaf@localhost.com`
- **Subject**: `test message`
- **Text**: `This is the test text`

The **To** address is statically set to `drsmith@localhost.com`. It is not set in the Tcl proc, such as the SMTP sample. That Tcl proc is removed.

POP3

POP3 is used to retrieve emails.

POP3S is used to retrieve emails securely. Use this when using CAA-Direct to connect to a Direct HISP.

An email can be one or both of:

- A message sent in the email message body, which is turned into the system message body.
- The message can be an attachment or even multiple attachments, which are linked or encoded in `USER DATA` on the inbound system message.

In both the POP3 and POP3S samples, there is an attachment directory set which is used as the directory where attachments are stored. The links to these files are available in the `USERDATA`. If this blank, then attachments are base64 encoded in the `USERDATA`.

The retriever protocol connects to the test server and retrieves email for the `drsmith` user. This is the address to which the SMTP and SMTPS samples are sent. It specifies an attachments directory to store the attachments, and enables **Mail Debug** so you can see the conversation with the mail server about retrieving messages. Because there is no "Retrieve interval" specified, it defaults to 30 seconds.

There are no TPS procs on this inbound thread. It raw routes the inbound message to the outbound POP3files thread.

The `POP3files` thread **Outbound** tab has **Retries** configured as "-1" and **Interval** as "10."

It first dumps the message so that you can see the message coming in from CAA-Direct.

The next proc `showInboundAttachment` is a proc whose goal is to list the file names under which the attachments have been stored. This shows you how to iterate over a list of attachments and get information about them.

Having the file name is sufficient to load the contents of the file into Tcl. Various actions can be made upon the contents or they can be sent elsewhere.

The main portion of this proc is:

```
# get the USERDATA
set userData [msgmetaget $mh USERDATA]
# if there's any attachments, process them
set attachments {}
if [catch {set attachments [keylget userData attachments]}] {
 puts "no attachments"
} else {
 # attachments are set
 # get an ordered list of the keys
 set sortedKeys [lsort -dictionary [keylskeys attachments]]
 foreach {key} $sortedKeys {
  # get the attachment for this key
  set attachment [keylsget attachments $key]
  # get the contentFile in this attachment, but be careful the user didn't switch to base64
  if [catch {set contentFile [keylsget attachment contentFile]}] {
   puts "got attachment, but not contentFile, probably base64"
  } else {
   # found contentFile, print just the file as the user presumably knows the directory because
it's in the config
   set fileName [lindex [file split $contentFile] end]
   puts "for attachment $key, got file name: $fileName"
  }
 }
}
```

You should read through the code to understand how it achieves the prior described objectives.

Note: The use of various keyed list commands such as `keylskeys` are not part of standard TclX keyed list processing. These are special keyed list commands starting with `keyls` instead of `keyl`. The difference with these commands is that they do not treat "." characters in key names as special tree indicators. Instead, they are treated as normal characters.

Essentially, when using these `keyls` commands, key names can have dots in them. This is critical when processing attachments whose key names are from the Content-ID header in which the attachment came. These can have values such as `my.special.attachment`. With these commands, that string is a legal key name.

POP3S

The POP3S example is the same as the POP3 example except that it has SSL configured to retrieve the message securely from the mail server. These parameters are used for SSL:

- **Id**: **`test`**
- **Secure Socket Protocol**: **`SSLv3`**
- **Truststore File***: **`../../../javadriver/testdirectkeystore.jks`**
- **Truststore Password***: **`changeit`**

It is the same configuration for SSL as with the SMTPS version. It has a truststore to authenticate the server, but no keystore. This is because mutual authentication is not required as a user name/password is required to access the email account.

These parameters are used for **POP3S**:

- **Id**: `simple`
- **Host***: `localhost`
- **Port**: `3995`
- **Username***: `drsmith`
- **Password***/**Confirm Password***: These must be specified
- **Use SSL**: This is selected
- **SSL Config**: `test`
- **Cloverleaf Trxid Determination**: `FROM`
- **Cloverleaf Attachment Directory**: `attachments`
- **Cloverleaf Log Exceptions**: `true`
- **Mail Debug**: `disabled`

This is similar to the POP3 thread's **POP3** tab. The port is different, as it is set to the test server's POP3S port. The **Use SSL** check box is selected with **SSL Config** that is set to the name of the SSL Socket Factory from the previous tab.

This is how a POP3 retriever links to a specific SSL configuration. Another minor difference from the POP3 sample is that **Mail Debug** setting is disabled. This indicates the conversation with the email server is not logged.

Note: You should not run the POP3, POP3S, or IMAPS samples simultaneously. This is because they are all checking for email messages on the same account. It would be random which client received a given email.

IMAPS

The IMAPS sample is exactly the same as the POP3S sample, except using IMAPS instead.

CAA-Direct Portecle Keystore Management tool (third-party)

CAA-Direct takes advantage of Java Mail's SSL handling, which depends extensively on the usage of the Public Key Infrastructure (PKI) technologies. Therefore, the management of X.509 digital certificates and private keys are an essential part of the operation of these features.

CAA-Direct is written in Java. The management of PKI and certificates is influenced by the mechanism that Java uses to manage them, using the concept of keystore and truststore.

These two types of stores are of the same internal structure, the JKS format or Java KeyStore. The distinction between them is primarily a logical one. The keystore contains public keys and their associated private keys that are used to authenticate self to remote partners. The truststore contains only the public keys of remote partners that are to be trusted.

Java run time provides a command line tool `keytool` that is capable of managing the keystore in a variety of ways. This includes generating a PKI key pair and its CSR (Certificate Signing Request), importing/exporting certificates, and others.

Portecle open source GUI

The Portecle open source GUI tool provides more functionality than `keytool`.

One of the features in Portecle that is absent from `keytool` (Java SE 6.0) is the ability to import key pairs from another keystore. This is noticeable when one is in a different format from JKS. Sometimes, it is convenient to import key pairs in the pkcs12 format, which is used in the Microsoft and many other popular security frameworks.

Portecle has a user-friendly GUI and is simpler to use than the command line based `keytool`. You should use Portecle to manage the keystores required for the CAA-Direct SSL functionalities.

The Portecle site, http://portecle.sourceforge.net/, has additional information on how to use this tool.

Installation

Download Portecle from http://sourceforge.net/projects/portecle (click the **Download** link to get the latest version) and unzip it to a directory . For example, `portecle-1.7`.

Alternatively, some operating systems such as Linux may already have an RPM package built for your system.

The `readme` file in the unzipped directory provides important information about things that must be finished before launching the tool.

If the machine that runs this tool already has JDK/JRE 1.6 installed, then the installation is already finished.

It is advisable to update the JCE unlimited strength jurisdiction policy files in the JRE. When this is finished, Portecle can handle more key sizes or algorithms.

These policy files are downloaded from Java's website and overwrite the same named files of the JRE installation: `local_policy.jar` and `US_export_policy.jar`. These are under`lib/security` .

The `readme` file in the download package also provides detailed installation information.

Launching Portecle

The `readme` file that comes with Portecle has detailed instructions on launching.

After the correct JCE provider jar and unlimited strength policy files have been properly installed, the tool is launched using the java `-jar` command.

If `.jar` files are associated with Java on your operating system, then you can double-click `portecle.jar` to launch it.

CAA-Direct logging

Logging is a developer's first line of attack with bugs. The key to good logging is to log only what you think you require. This keeps you from having a flood of information to sift through and perhaps losing the import logs within.

To start with, do not enable any system engine logs when trying to debug CAA-Direct. This buries relevant CAA-Direct logs in a flood of non-essential information. The system engine logs are useful when debugging system engine issues. Because CAA-Direct is a Java application running on top of the engine, the normal engine output logs tend to be unnecessary.

CAA-Direct has these useful logging options:

- Mail server conversation logging
- System message dump
- CAA-Direct internal logging

Mail server conversation logging

On the **SMTP** and **POP3** configuration tabs, you can enable **Mail Debug** logging. This type of logging shows the conversation between the email client and the email server.

The **SMTP**/**POP3**/**IMAP** config tabs have their own individual menus.

The values are:

- **`disabled`**
- **`normal debug`**
- **`normal+auth debug`**

  `+auth` means that the authentication part of the conversation is not suppressed as it is with normal debug. It shows the user name/password in plain text.

This is an example of the type of logging this produces in the process log file for the sample POP3 thread. This is found in the inbound process log file:

```
DEBUG: JavaMail version 1.4.7
DEBUG: successfully loaded resource: /META-INF/javamail.default.providers
DEBUG: Tables of loaded providers
DEBUG: Providers Listed By Class Name: {com.sun.mail.smtp.SMTPSSLTrans
port=javax.mail.Provider[TRANSPORT,smtps,com.sun.mail.smtp.SMTPSSLTransport,Oracle],
com.sun.mail.smtp.SMTPTransport=javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTrans
port,Oracle], com.sun.mail.imap.IMAPSSL
```

```
Store=javax.mail.Provider[STORE,imaps,com.sun.mail.imap.IMAPSSLStore,Oracle],
com.sun.mail.pop3.POP3SSLStore=javax.mail.Provider[STORE,pop3s,com.sun.mail.pop3.POP3SSLStore,Or
acle], com.sun.mail.imap.IMAPStore=javax.mail.Provider[STORE,imap,com.sun.mail.imap.IMAPStore,Or
acle],
com.sun.mail.pop3.POP3Store=javax.mail.Provider[STORE,pop3,com.sun.mail.pop3.POP3Store,Oracle]}
DEBUG: Providers Listed By Protocol:
{imaps=javax.mail.Provider[STORE,imaps,com.sun.mail.imap.IMAPSSLStore,Oracle],
imap=javax.mail.Provider[STORE,imap,com.sun.mail.imap.IMAPStore,Oracle],
smtps=javax.mail.Provider[TRANSPORT,smtps,com.sun.mail.smtp.SMTPSSLTransport,Oracle],
pop3=javax.mail.Provider[STORE,pop3,com.sun.mail.pop3.POP3Store,Oracle],
pop3s=javax.mail.Provider[STORE,pop3s,com.sun.mail.pop3.POP3SSLStore,Oracle],
smtp=javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Oracle]}
DEBUG: successfully loaded resource: /META-INF/javamail.default.address.map
DEBUG: getProvider() returning javax.mail.Provider[STORE,pop3,com.sun.mail.pop3.POP3Store,Oracle]
DEBUG POP3: mail.pop3.rsetbeforequit: false
DEBUG POP3: mail.pop3.disabletop: false
DEBUG POP3: mail.pop3.forgettopheaders: false
DEBUG POP3: mail.pop3.cachewriteto: false
DEBUG POP3: mail.pop3.filecache.enable: false
DEBUG POP3: mail.pop3.keepmessagecontent: false
DEBUG POP3: mail.pop3.starttls.enable: false
DEBUG POP3: mail.pop3.starttls.required: false
DEBUG POP3: mail.pop3.apop.enable: false
DEBUG POP3: mail.pop3.disablecapa: false
DEBUG POP3: connecting to host "localhost", port 3110, isSSL false
+OK POP3 GreenMail Server ready
CAPA
-ERR Command not recognized
USER drsmith
+OK
PASS doctor
+OK
STAT
+OK 1 604
NOOP
+OK noop rimes with poop
TOP 1 0
+OK
Return-Path: cloverleaf@localhost.com
Received: from 127.0.0.1 (HELO USSPNJPANGBU01.infor.com); Mon Mar 10 14:41:56 PDT 2014
Date: Mon, 10 Mar 2014 14:41:56 -0700 (PDT)
From: cloverleaf@localhost.com
To: drsmith@localhost.com, drsmith@localhost.com
Message-ID: 15865423.2.1394487716825.JavaMail.jpangburn@USSPNJPANGBU01
Subject: test CAA-Direct message
MIME-Version: 1.0
Content-Type: multipart/mixed;
 boundary="----=_Part_0_15184882.1394487715184"
RETR 1
+OK
Return-Path: cloverleaf@localhost.com
Received: from 127.0.0.1 (HELO USSPNJPANGBU01.infor.com); Mon Mar 10 14:41:56 PDT 2014
Date: Mon, 10 Mar 2014 14:41:56 -0700 (PDT)
From: cloverleaf@localhost.com
To: drsmith@localhost.com, drsmith@localhost.com
Message-ID: 15865423.2.1394487716825.JavaMail.jpangburn@USSPNJPANGBU01
Subject: test CAA-Direct message
MIME-Version: 1.0
Content-Type: multipart/mixed;
 boundary="----=_Part_0_15184882.1394487715184"
------=_Part_0_15184882.1394487715184
Content-Type: multipart/related;
 boundary="----=_Part_1_20306499.1394487715278"
------=_Part_1_20306499.1394487715278
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
5632
------=_Part_1_20306499.1394487715278--
------=_Part_0_15184882.1394487715184
Content-Type: application/octet-stream; name=myattachment
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=myattachment
attachment content goes here in the attachmentSource file, binary can work too
------=_Part_0_15184882.1394487715184--
```

This log shows the full conversation with the mail server, including user name/password. This is a helpful if you are unsure of what you should be receiving.

Cloverleaf message dump

By calling Tcl to dump a message, you can see what is retrieved from and sent to a CAA-Direct thread.

All the sample threads use this. For outbound threads (SMTP), look on the **Outbound** tab for the `TPS Outbound Data` UPoC. These have the `dumpMsg` Tcl proc, which dumps the message to the process log. For POP3 or IMAP inbound threads, look on the **Outbound** tab for the `TPS Inbound Data` UPoC.

If you have the Mail Debug conversation logging enabled on the POP3 sample thread, then you can see the conversation with the mail server logged. After CAA-Direct processing, it is sent to the system where the Tcl proc `dumpMsg` runs. This results in this message dump being sent to the log:

```
[tcl :out :INFO/0:    POP3files:03/10/2014 14:43:48]
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--] sms_ob_data Dump:
[tcl :out :INFO/0:    POP3files:03/10/2014 14:43:48] msg: 0x02AED828
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgType           : DATA
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgClass          : ENGINE
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgState          : OB pre-SMS (10)
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgPriority       : 5120
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgRecoveryDbState: Log:update (3)
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgFlags          : 0x8002
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgMid            : [0.0.17055]
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgSrcMid         : [0.0.17054]
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgSrcMidGroup    : midNULL
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgHostId         : 3070311904
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgOrigSrcThread  : POP3
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgOrigDestThread : POP3files
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgSrcThread      : POP3
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgDestThread     : POP3files
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgXlateThread    :
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgSkipXlate      : 0
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgSepChars       :
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgNumRetries     : 0
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgGroupId        : 0
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgDriverControl  :
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgRecordFormat   :
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgRoutes         :
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgUserData       : {headers {{content-
type multipart/mixed\;\ \r\n\tboundary=\"----=_Part_0_15184882.1394487715184\"} {to drsmith@lo
calhost.com,\ drsmith@localhost.com} {subject test\ CAA-Direct\ message} {mime-version 1.0}
{message-id <15865423.2.1394487716825.JavaMail.jpangburn@USSPNJPANGBU01>} {received from\
127.0.0.1\ (HELO\ USSPNJPANGBU01.infor.com)\;\ Mon\ Mar\ 10\ 14:41:56\ PDT\ 2014} {from clover
leaf@localhost.com} {date Mon,\ 10\ Mar\ 2014\ 14:41:56\ -0700\ (PDT)} {return-path <cloverleaf@lo
calhost.com>}}} {generalInfo {{to drsmith@localhost.com,drsmith@localhost.com} {replyTo clover
leaf@localhost.com} {subject test\ CAA-Direct\ message} {sentDate Mon\ Mar\ 10\ 14:41:56\ PDT\
2014} {from cloverleaf@localhost.com} {cc {}}}} {attachments {{1 {{headers {{content-type appli
cation/octet-stream\;\ name=myattachment} {content-transfer-encoding 7bit} {content-disposition
 attachment\;\ filename=myattachment}}} {contentFile C:\\cloverleaf\\cis6.0\\integrator\\di
rect_samples\\javadriver\\inbound\\attachments\\CAAEmail2763823352417330571attachment}}} {0
{{headers {{content-type text/plain\;\ charset=us-ascii} {content-transfer-encoding 7bit}}}
{contentFile C:\\cloverleaf\\cis6.0\\integrator\\direct_samples\\javadriver\\inbound\\attach
ments\\CAAEmail3218073862942335862attachment}}}}}
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgStaticIsDirty  : 0
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgVariableIsDirty: 0
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgTimeStartIb    : 1394487828.830
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgTimeStartOb    : 1394487828.853
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgTimeCurQueStart: 0.000
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgTimeTotalQue   : 0.023
```

```
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgTimeRecovery   : 1394487828.830
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgEoConfig       : 0x00000000
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     msgData (BO)      : 0x02AED928
[tcl :out :INFO/0:    POP3files:--/--/---- --:--:--]     message           : '5632'
```

Note: Useful information includes all the metadata information available in `USERDATA` and the message itself. This includes attachment file information.

CAA-Direct internal logging

For last line troubleshooting, CAA-Direct logs some information as it processes messages. In most cases, this is completely unnecessary. If you suspect a bug or some fault lies with CAA-Direct or the underlying JavaMail API, then enable the Java-based logging for these software packages. Then, you can see if it shows any information about what is happening.

This sort of logging falls under the Java Util Logging domain. You can search the internet for information about what sort of logging configurations are possible under Java Util Logging. You can also find examples of turning on logging to a separate file, learning about log levels, and so on.

For an example, the inbound process in the `direct_samples` sample site has this enabled.

The **Process Configuration** dialog box uses this configuration on the **Java Driver** tab's **User Defined Options** tab:

- **Name**: **`java.util.logging.config.file`**

  For the process, you create a user-defined option with this name.

- **Value**: **`$SITEPATH/javadriver/logging.properties`**

  The value is set to the location of your `logging.properties` file. The Java Driver `$SITEPATH` variable points to the site directory. `javadriver` is the directory, and `logging.properties` is the sample file.

By default, at the end of this file is this relevant line:

```
com.infor.cloverleaf.direct_adapters.level = FINEST
```

This is an example in the process log from CAA-Direct:

```
Mar 10, 2014 2:43:46 PM com.infor.cloverleaf.direct_adapters.retriever.Retriever doTimeEvent
FINE: running doTimeEvent
Mar 10, 2014 2:43:46 PM com.infor.cloverleaf.direct_adapters.retriever.Retriever doTimeEvent
FINER: processing config 'pop3test'
Mar 10, 2014 2:43:46 PM com.infor.cloverleaf.direct_adapters.retriever.POP3RetrieverConfig get
Password
INFO: Encoded password is "c81=8a=>UT19eAon3c<MIAH:jbiXQI30", you can copy it to your config to
 protect it and set passwordEncoded to true.
Mar 10, 2014 2:43:46 PM com.infor.cloverleaf.direct_adapters.retriever.Retriever doTimeEvent
FINEST: password retrieved: true
```

These logs can occasionally be useful because they show when the `doTimeEvent` fires and then what happens afterward. The `doTimeEvent` is the event that goes every so often and triggers the adapter to check for new mail. Then it tells you which config entry it is processing. This is useful to separate other log entries if you have multiple POP3 or IMAP config entries.

By default, it runs at `INFO` level, so you would always see the notice about the encoded version of the password. With this, you can put that into the configuration tab instead of the plain text password. To get a lower level, such as `FINE`, `FINER`, `FINEST`, set it by the `logging.properties` file.

If the level was turned up higher, for example, `WARNING`, then this would not show in the log.

Be careful with settings lower than `INFO`. These lower level kind of logs are not good in production. This is because they may happen on every message and clutter the log. They are useful, though, in development to verify such things as from which POP3 server was it retrieving messages when an error happened.

CAA-Direct known issues

The inbound side of attachment's `contentFile` path is different from where it is saved.

On the inbound side, an attachment's `contentFile` path in `USERDATA` is different from where the file is saved.

For example, below is the content of a sample site's inbound `USERDATA`:

```
{contentFile c:\\cloverleaf\\cis6.1\\integrator\\direct_samples\\javadriver\\inbound\\attachments\\
CAAEmail136403010493537233 9attachment}
```

The file is stored in:

`C:\cloverleaf\cis`*`version`*`\integrator\direct_samples\exec\processes\inbound\attachments\ CAAE mail136403010493537233 9attachment`

This is due to a bug in the Cloverleaf Java Driver code.

Workaround

Extract the file name from the `contentFile` name/value pair in `USERDATA` and place it in the location where you configured it to store the files.

The sample Tcl for "POP3" in CAA-Direct sample sites shows how to extract the file name.

Log files and troubleshooting

The Bridge has its own log file system, configured at `conf/log4j.xml`. This is a standard `log4j` configuration file. By default, it has appenders that:

- Clients can connect with to get log information.
- Writes everything to standard out.
- Writes everything to a rolling file appender at the installation root folder in a file `bridge.log`. This is the first place to look for debugging at the Bridge level.
- Uses a custom appender that is aware of transactions to write to the bridge folder. It creates subfolders named by RHIO which contain log files specific to the transactions in that RHIO. This granularity can be helpful when debugging in a production environment with large standard out log files.

The log level for Bridge code is set to “debug” which means it outputs much debugging information for every transaction. This is useful for debugging in development but can generate massive log files if used on a busy production machine. The levels should be turned down to at least “info” for day-to-day production logging.

Additional webapps logging

These apps also use `log4j` logging which can be configured for each app. This is accomplished by going to `webapps/`*`the app`*`/WEB-INF/classes` and editing the `log4j.xml` file and restarting Tomcat:

- `InitiatingGateway`
- `QvdxADTListenerApp`
- `RegRepoCleanupApp`
- `RespondingGateway`

The Repository Viewer app uses `log4j`, and is located at `browser/Viewer/WEB-INF/classes`.

The Registry/Repository writes most information to standard out and is not configurable.

Standard out

On Windows, most of the standard out logging goes to `logs/catalina.*.log` where `*` is the current date when Tomcat was started.

On Linux, depending on how you start your script, the standard out can go to the console or the catalina log files. This is the first place to look for non-Bridge problems.

Frequently asked questions

Frequently asked questions include:

- What is the affinity domain universal ID?
- What fields are in the bridge submission for metadata map to the CodeType/@name in the registry's `codes.xml`?
- What are the required fields in the HL7 v2 ADT feeds to the registry?

What is the affinity domain universal ID?

This is the assigning authority for allowable patients. This value tells the Registry to only accept patients within this assigning authority. Outside patient submissions are ignored.

If the patient's universal ID is left blank on feed messages, then `UniversalId` is supplied and stored in the Registry.

The ID is always stored without the `NamespaceId` portion. Document submissions and queries with a `NamespaceId` are stripped.

```
<Parameter name="assigningAuthority.NamespaceId" value=""/>
<Parameter name="assigningAuthority.UniversalId" value="1.3.6.1.4.1.21367.2005.3.7"/>
<!-- UniversalIdType is always ISO -->
```

What fields are in the bridge submission for metadata map to the CodeType/@name in the registry's codes.xml?

- documentEntryClassCode → classCode
- documentEntryConfidentialityCodes → confidentialityCode
- documentEntryEventCodes → eventCodeList
- documentEntryFormatCode → formatCode
- documentEntryHealthcareFacilityCode → healthcareFacilityTypeCode
- documentEntryPracticeSettingCode → practiceSettingCode
- documentEntryTypeCode → typeCode
- submissionSetContentTypeCode → contentTypeCode

`codes.xml` has an extra `mimeType` entry that is used by the Repository for setting the file name extension. This happens when storing the document and setting the attachment content type to the right `mimeType` when retrieving the document. As you're only submitting CDA docs, leave this field as-is.

What are the required fields in the HL7 v2 ADT feeds to the registry?

Infor uses these required fields:

```
messageType from MSH-9-1
triggerEvent from MSH-9-2
patientId from PID-3
patientName from PID-5
 from the repeats we use the following cells:
   Cell lastName = currentPatientName.getElement("1");
   Cell firstName = currentPatientName.getElement("2");
   Cell middleName = currentPatientName.getElement("3");
   Cell suffix = currentPatientName.getElement("4");
   Cell prefix = currentPatientName.getElement("5");
   Cell degree = currentPatientName.getElement("6");
```

These are not required fields, but are stored in the database if you pass them. They can be used by administrators for manually auditing and error checking purposes :

```
patientAddress from PID-11
 from the repeats we use the following cells:
   Cell streetAddress = currentPatientAddress.getElement("1");
   Cell otherDesignation = currentPatientAddress.getElement("2");
   Cell city = currentPatientAddress.getElement("3");
   Cell stateOrProvince = currentPatientAddress.getElement("4");
   Cell zipcode = currentPatientAddress.getElement("5");
   Cell country = currentPatientAddress.getElement("6");
   Cell countryOrParish = currentPatientAddress.getElement("9");
patientRace from PID-10
 from repeats:
  Cell raceCode = currentPatientRace.getElement("1");
  Cell raceText = currentPatientRace.getElement("2");
birthday from PID-7
sex from PID-8
accountNumber from PID-18
bed from PV1-3-3
```

Index

A

Add Jetty Engine dialog box 47
Add Raw Server dialog box 47
affinity domain universal ID 191
Apache CXF configuration guide 34
applicationContext_SignEncProvider.xml 96
asynchronous mode clients 30
asyncRawClientFile 94
asyncRegiClientFile 93
authentication type 80

B

BODs 35, 38
bounceRaw 93
bounceSOAPRegistry 91

C

CAA-Direct
    logging options 184
caa-direct retriever/sender 34
caa-ws client/rawclient/server 35
callback classes 138
codes.xml 191
conduit configuration 78
configuration flow
    alternate 86
    basic 85
    Raw client 87
    raw HTTP server 88
    RESTful client 86
    RESTful server 88
    SOAP client message 86
    SOAP client payload 86
    SOAP server message 87
    SOAP server payload 87
configuration objects 70
contentFile 189
CXF
    configuration interface 15
CXF options 97

D

direct-retriever 34
direct-sender 34
dispatch name 90
dispatcher 15
dumpMsg 93, 96

E

engine
    creating 47

F

FHIR BOX 98
FHIR examples BOX 103
FHIR schemas 111
FHIR test servers 111
FHIR_example.box 98
fields
    client inbound 25
    override 15
    provider inbound 18

H

HL7 FHIR 102
HL7 message sample 105
HL7 v2 ADT 191

I

Include operation 112–113
ion retriever 36
ion sender 39

J

java/ion-retriever protocol 35
java/ion-sender protocol 38
jaxws:client 138
jaxws:server 138
JCE policy files 126
jetty 12, 142
JKS keystore 84

L

logging
    mail debug 184
    standard out 190
    webapps 190
logical items
    client 48
    server 57

M

MerlinWrapper 135

message validation 62
messages
    inbound 15, 17
    outbound 15, 17
    SOAP 91

N

new conduit
    creating 45
nodes
    ws-client/ws-server 73

O

outInterceptors 96
overrides
    client 63
    client outbound 20
    provider outbound 28
    server 66

P

Policy Generation tab 137
Portecle
    installing 182
    launching 182
Portecle open source 182
portecle.jar 126
provider inbound fields 17
public key infrastructure 125

R

raw client
    asynchronous 94
raw server
    creating 47
RAW server 69
RawClient 94
rawClientFile 94
RawHandler 93
RegistryClient 92
registryClientFile 92
registryFileClient 92
registryQuery.50.xml 93
requests
    basic 90
REST
    XML data 69
REST Consumer 70
REST endpoint 93
rest server
    dummy 48
retrieve interval 36

S

sample client
    creating 44
sample server
    creating 46
sample sites
    CAA-WS 89
security settings 83
security testing 83
Server.java 96
sign_enc sample 96
soap client
    asynchronous 93
SOAP client 86
SOAP Consumer
    creating 71
SOAP envelope 86
SOAP messages 69
SOAPProvider_Registry 91, 128
SOAPProvider_Registry thread 128
SOAPProviderPayload_Registry 92

T

Tcl keyed list 17
tcl overrides 30
Tcl user interface 15
test button 83

U

updateRawClientMessage 94
URL
    endpoint 47
Use Transport Security option 137
USERDATA
    overrides 17
users
    normal 85
    power 85

V

validation 83

W

web service server/client 12
web services
    multiple 122
webclient factory 78
wizard
    web services consumer 69
ws_more_samples 95
WS-addressing 92
ws-client 35

ws-rawclient 35
ws-samples site 89
WS-Security parameters
    non-boolean 133
WS-Security tags
    boolean 132
ws-security.callback-handler 136
ws-server 35, 71
WSDL file
    selecting as input file 72
WSDL location 128
WSDL2XSD 120
WSS4J options 97
WSS4JInInterceptor 96

X

xsd 76
XSD file
    selecting as input file 73
XSD WSDL files 72
XSDs
    multiple 122
xsdWsdlToolClientGUI.bat 120
xsdWsdlToolGUI.bat 124