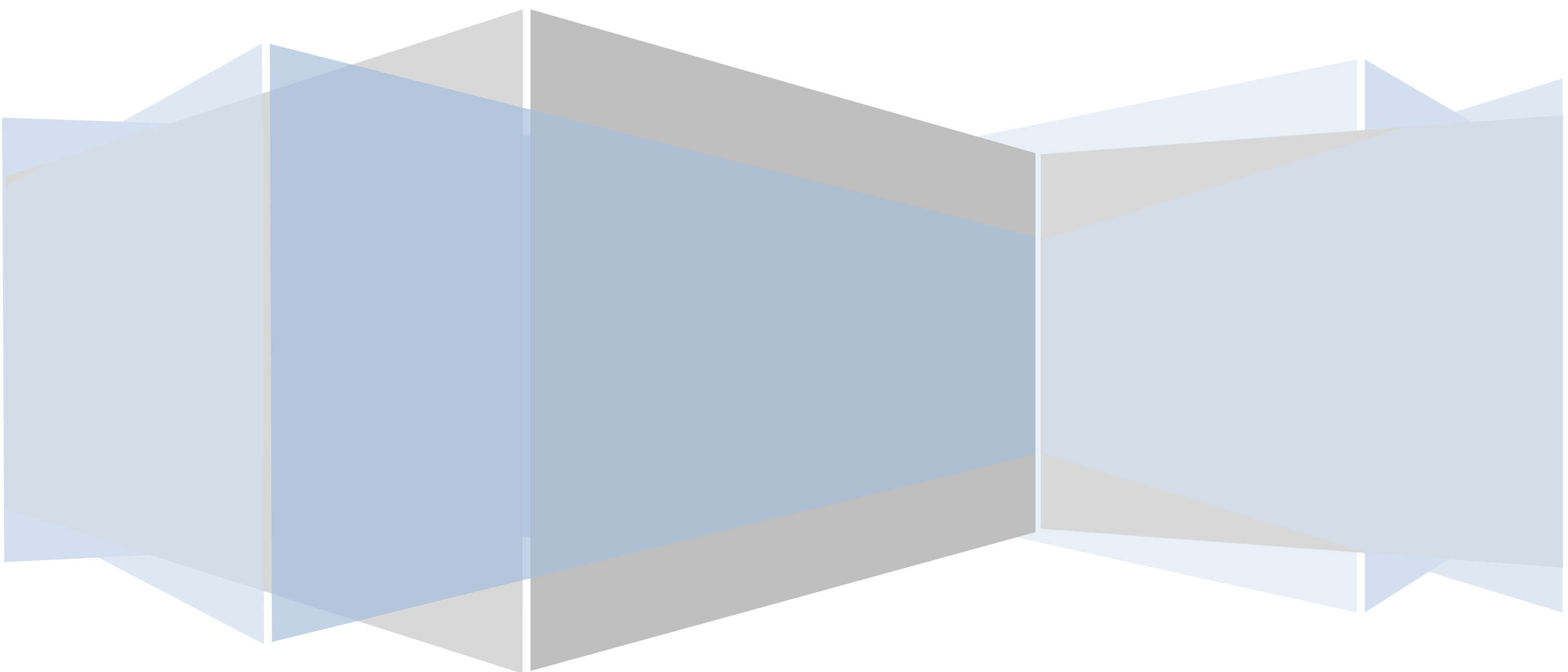
XML Gateway

Installation and usage

J System Solutions

http://www.javasystemsolutions.com

Version 3.5



Introduction....7
AR System, SQL and LDAP integrations....7
Videos....7
Installing the XML Gateway....8
Installing the XML Gateway and the Mid Tier together....8
CMDB support....8
Background and required knowledge....9
XPath....9
Getting started, quickly....10
Creating a new AR System user through an XML post....10
Performing a query against an AR System database....10
Modifying an AR System user through an XML post....10
Configuration files....11
Primary configuration file – xmlgateway.xml....12
AR System servers....13
HTTP create/modify bindings....13
HTTP query bindings....13
File fetchers....14
POP3 fetchers....14
Logging to a database....14
Creating, updating and deleting records....16
Location of templates....16
Performing your first transaction....16
Template structure....16
The root node....16
Transaction ID....17
Email on error....17
Form definitions....17
Target object (setting the form)....17
Set field....18
Setting a field to null....18
Field mapping....18
Setting a field by an XPath query....18

Setting a field to null....18
Importing attachments....18
Controlling the create/modify/delete functionality....20
Performing multiple operations with a single formDefinition....20
XML responses to create, modify and delete transactions....21
Additional operations....22
dateFormat....22
queryFieldFromXPath....22
setFieldFromQuery....23
onError....23
setMessage....24
setGatewayValue....24
secure....24
Using AR System groups....25
schema....25
namespaces....25
Saxon vs Xerces....25
responseBuilder....25
filter....26
CSV....26
JSON....26
Swift....26
filter-class....26
Impersonating a different AR System user....27
Checksum....27
Functions....27
Checksum....27
Encrypt....27
GUID....28
Querying the AR System....29
Location of templates....29
Performing your first transaction....29
Template structure....29
The root node....29
Search criteria....30
Specifying the XML response template....30

Specifying a result field....31
Additional operations....32
Email on error....32
Namespaces....32
Securing a template....33
Validating against a schema....33
Changing the HTTP content type....33
Query statistics generated by the gateway....33
Removing elements....34
AR System server info and statistics....34
Attachments....34
Diary fields....35
Retrieving field properties....36
Complex query example....37
Sorting results....37
Parameterised queries....37
Chaining query templates....38
Encouraging portable templates....39
XSL Transformations....39
Filters....39
CSV....39
JSON 40
Swift....40
Setting file download headers....40
filter-class....40
Template based filters....40
Globally....40
Configurations....40
Setting filter on query request....41
JSON based queries....41
The AR System impersonate feature....41
Securing templates through client authentication....42
HTTP URLs exposed by the gateway....43
Performing create/modify operations....43
Performing query operations....43

Authentication....44
Integrating with the gateway with other systems....45
Java Messaging Services....45
Integration technique for create/modify/delete....45
Integration technique for queries....46
Configuration files....46
Core configuration....46
Referring to templates from gateway configuration....48
Webservices exposed by the gateway....49
Locating the WSDLs....49
Important note....49
Services provided by the XML Gateway WSDL....50
Create and modify....50
Querying....50
Querying through HTTP Query bindings....50
Query and forward to URL....50
Query and forward to another XML Gateway....51
Query and forward to a Java Messaging Service....51
Query and forward to a message sending destination....52
Query ARS and forward to message sending framework....52
Query ARS and invoke a Plugin....53
Combined post and query....53
Combined query and post....53
Authenticate....54
Unauthenticate....54
Perform an XPath select....54
Message sending framework....55
Overview....55
Third party to AR System....55
AR System to third party....55
Key components of the message sending functionality....56
Destinations....56
HTTPDestination....56
SampleDestination....56
WebserviceDestination....56
Message handlers....57

DefaultMessageHandler....57
ARSMessageHandler....57
Error handlers....58
Triggers....58
Methods of operation....58
Invoking through a webservice....58
Using the scheduler....61
Spring configuration file....61
Core scheduler....61
Triggers....61
Jobs....62
Query and post to gateway....63
Query and post to a Java Messaging Service....63
Query and send to destination (such as HTTP)....64
Developing plugins....65
Bespoke response builders....65
Webservice plugins....65
Message sending destinations....66
Filters....66
Pre-production optimisation and system testing....67
AR System form caching....67
Session handling....67
Memory footprint....68
Increasing the heap memory....68
Increasing the PermGen size....68
Reviewing log files for performance issues....69
OutOfMemoryError....69
OutOfMemoryError: PermGen space....69
Logging in production....69
Performance logging....69

Introduction

The XML Gateway should run on any Java servlet engine however we recommend Apache Tomcat (from the Apache Foundation). The Tomcat download page is here: http://tomcat.apache.org/index.html

You require a Tomcat installation for your platform and we recommend the latest 6.0 server.

The XML Gateway requires a Java 1.5, however we recommend the latest 1.6.

If you are in any doubt what to download, please contact JSS Support. You will need to install the Java SDK before you proceed to install Tomcat and the XML Gateway.

We do not provide a guide on how to install Tomcat as there are hundreds of guides on the Internet, and BMC automatically install Tomcat with the Mid Tier.

AR System, SQL and LDAP integrations

The gateway provides integration features for SQL databases and LDAPs, however this manual currently focuses on the most common implementation – AR System integrations. You will find examples of SQL/LDAP integrations as the techniques are similar to AR System, with small variations in the template structure.

Videos

There are a range of videos on the JSS website that demonstrate how to install and use the gateway – these are a valuable source of information so please take a few minutes to review them.

Installing the XML Gateway

The XML Gateway is shipped as a WAR file. Tomcat's administration console provides a mechanism to install the war file, as do other servlet engines. With Tomcat, it's just as quick (if not quicker) to install the war file manually. The steps are written for Windows (we assume Unix users will have no problem following them) and are as follows:

1. Copy the XML Gateway WAR file to the [Tomcat Directory]\webapps.
2. Start Tomcat, which can be done through Windows Services.
3. Tomcat will find the new WAR file and unpack the contents into a new directory with the same name as the WAR file. Wait until this appears then test by opening your browser and type http://[servername]:8080/xmlgateway. If the installation was successful then you will see the XML Gateway welcome page, with a message requesting a license.
4. If you have a production license key, copy it to the [Tomcat Directory]/webapps/xmlgateway/WEB-INF/classes directory. If you are evaluating the product then the gateway will be limited to 100 transactions per hour.
5. Re-start Tomcat and go back to the welcome page, you can start to use the system.

Installing the XML Gateway and the Mid Tier together

If you wish to install the XML Gateway with another product running in the same servlet engine that uses AR System (such as the Mid Tier) then there may be an issue with the 'shared' use of the AR API. Older versions used native libraries, and while these were dropped in version 7.5 (for the AR API – they are still required for CMDB), there may be issues when mixing applications using the AR API. If you have any problems or concerns then contact JSS Support.

CMDB support

The XML Gateway has a number of CMDB features for creating and modifying CMDB classes. The CMDB API (at least, on version 7.5) still makes use of native libraries, so if you intend to use this functionality of the gateway then you must install the native libraries on your machine. They are not shipped with the gateway so contact JSS Support if this is a requirement.

Please do not confuse operations on CMDB classes with operations on CMDB data – out of the box, the XML Gateway can manipulate CMDB data as this is performed through the AR API.

Background and required knowledge

The gateway is designed to provide a more complete solution to integrating with the BMC AR System than the out of the box webservices. The out of the box webservice implementation has a place in integration, but they are by far a complete solution – for example, they don't handle attachments, provide no facility for handling errors, and if your client wishes to a messaging solution (such as Tibco of Weblogic), an entirely different technology is required.

The gateway provides a comprehensive XML messaging hub with a range of connectors, specifically designed around the AR System, as opposed to the majority of other products that provide AR System integration where the AR System was an 'after thought' (hence, the feature set is smaller and less reliable).

XPath

The gateway makes heavy use of XPath to manage data within XML. You can read more about it on this wiki page:

http://en.wikipedia.org/wiki/XPath

There are many tutorials on the Internet and we recommend you read the one on w3schools:

http://www.w3schools.com/Xpath

The gateway makes use of the Saxon XPath libraries and the documentation is worth reviewing. In particular, the following URL provides a list of functions you can use when writing create/modify templates:

http://www.saxonica.com/documentation/functions/intro.html

Getting started, quickly

This guide assumes you've configured a AR System instance with the alias myarserver (through the xmlgateway.xml file).

Creating a new AR System user through an XML post

We supply a sample template called newUser.xml and it can be found in the [tomcat]/webapps/xmlgateway/WEB-INF/classes/templates/createModify directory. If you open the template in a text editor, you will see that the comments at the top provide some sample XML to post into the gateway.

To post the XML into the gateway, go to the test post page; paste the sample XML from the template (at the top, in comments) into the top text field; enter newUser in the template field; select myarserver from the data source drop down and press the process XML button.

You will now be taken to the statistics page and should see that one new entry was created in the User form.

Performing a query against an AR System database

We supply a sample query template called sampleQuery and it is installed with the product in the [tomcat]/webapps/xmlgateway/WEB-INF/classes/templates/queries directory. This is a very simple query template and returns only a few fields, but also demonstrates some of the more advanced gateway features such as date formatting. To perform a query against a data source, an XML query must be passed to the gateway and a sample one is provided at the top of the template.

To run this example, go to the test query page; post the sample XML query request from the template (at the top, in comments) into the first text field and press the process XML button.

Modifying an AR System user through an XML post

Repeat the process you followed when creating the xmlgwtest user but before pressing process XML button, modify one of the text values in the sample XML you are about to post. We suggest you change the Full Name from XML Gateway Test to something else. Now press the process XML button.

You will now be taken to the statistics page and should see that one entry was updated.

Configuration files

Once you've successfully installed the gateway, you will also need to review the configuration files to setup your AR System/SQL server/Directory Services/etc.

The configuration files are all located in the [tomcat]/webapps/xmlgateway/WEB-INF/classes directory.

However, to make the XML Gateway upgrade path easier, all configuration files are loaded from the classpath and therefore they can be present in any directory present on the JVM classpath. For example, you may wish to add a directory /xmlgateway-config to the classpath and store your configuration files in this directory, or if you're using Tomcat, you may wish to use the tomcat/common/classes directory as your storage location.

It is of course entirely optional and new users can simply edit the existing files.

When you make changes to the configuration files you must also restart the gateway.

The following list provides an overview of the configuration files:

- xmlgateway.xml: The main XML Gateway configuration file which includes configuration of AR System server instances.
- databases-context.xml: All SQL databases are defined within this file.
- directories-context.xml: All directory services are defined within this file.
- templates/createModify: Directory containing user defined create/modify templates.
- templates/queries: Directory containing user defined query templates.
- schemas: Directory containing schemas used by the templates.
- xmlgateway.license: Your XML Gateway license file for those who are not evaluating/running the community edition.
- messageSending.xml: The message sending framework configuration file.
- scheduler-context.xml: The gateway's inbuilt Quartz scheduler is defined in this file. You can also configure scheduled jobs to run, such as performing a query and posting the results to another XML Gateway server.
- jms-core.xml: The JMS core configuration, which defines connections, destinations, queues, etc.
- jms-jobs.xml: JMS message listeners are defined in this file. These listeners bind to a queue and perform create/modify or query operations.

Primary configuration file – xmlgateway.xml

The following items are configured through the xmlgateway.xml. Do not forget that any changes to the file will not be reflected until you restart the Tomcat server.

logLevel: The level of logging that will be performed. This should be set to INFO for production environments, and DEBUG or TRACE if you wish to send log files back to JSS. Logging can be sent to a database as described below.

emailFromAddress: When the XML Gateway sends an e-mail to signal a processing error, the from address of that e-mail is configured here.

attachmentKeyExpirtyTime: The amount of time that keys generated by the attachment query system will remain valid. I.e. attachment URLs generated by the query system should be accessed within this period of time.

cacheRefreshInterval: The number of minutes that the internal AR System object cache will persist. It is rebuilt at these intervals.

cachedFormFieldSets: Retrieving AR System form definitions is very slow, hence this configures the number of forms to cache.

cachedAttachments: The number of AR System API Attachment objects that will be cached while performing a query. Queries against attachment fields result in the attachment being held in memory for the client to collect via a subsequent request.

responseBuilder: Bespoke response builders can be written for the gateway, and this tag is used to define a system wide response builder.

databaseLoggerSQLServer: XML Gateway logging can be written to a database table by passing an SQL server definition (discussed below) to this tag. This provides an easy mechanism of producing XML Gateway reports, given selecting from a table is much easier than parsing a file. The logs will still be sent to a file.

maximumQueryCount: A global maximum query count can be defined that will be used to limit any query performed, however, the figure can be changed by defining the value in a template.

XPathFactory: Used to configure the gateway to use a particular XpathFactory. By default, Saxon is used, however if you wish to use the standard Java Xerces implementation then this is documented in the configuration file.

TransformerFactory: Used to configure the gateway to use an alternative TransformerFactory. By default, Saxon is used, however if you wish to include Java extensions into XSL templates then you will need to set the standard Java Xalan implementation as per the example in the configuration file.

AR System servers

AR System connection pooling is performed using the ARAPI connection pooling functionality, and this can be configured in the arsys_api.xml file in the xmlgateway/WEB-INF/classes directory.

Inside the ars tag, a number of AR System server definitions can be defined as follows:

```
<server name="myarserver" host="192.168.0.54" user="Demo"
password="itsm" tcp="" rpc="" />
```

States that an AR System server called myarserver (**internal XML Gateway alias**) is located at the host 192.168.0.54, and that the login details (does not have to be admin) are Demo/password. No TCP or RPC ports are defined.

The definition can also include the following:

1. clientType: This allows the AR API client type to be set. By default, it is not set on the ARServerUser login call.

HTTP create/modify bindings

In order for third party clients to be able to post XML to the gateway without sending any parameters (i.e. stating the template or server required), you can bind an IP address to a template and server pair using the map tag. For example:

```
<map source="192.168.0.2" template="newUser" server="myarserver" />
```

States that connections to the /pushxml/createmodify.do servlet or webservice post function, from the IP 192.168.0.2, will result in the newUser template being used against the AR System alias myarserver.

HTTP query bindings

In order for third party clients to be able to query gateway without sending any parameters (i.e. stating the template or server required), you can bind an IP address to a template and server pair using the map tag. For example:

```
<map source="192.168.0.2" template="sampleQuery" server="myarserver"
key="2" />
```

States that connections to the /pushxml/query.do servlet or webservice query function, from the IP 192.168.0.2, will result in the sampleQuery template being used with the authorisation key 2. The query bindings must be used with an authentication key, allowing further information to be configured within the template.

File fetchers

The XML Gateway can retrieve create/modify XML requests from files (ending in .xml or .csv) by scanning a directory at a defined interval. To set up this functionality, define a fileFetcher tag as follows:

```
<fileFetcher directory="/tmp" targetServer="myarserver" interval="5"
template="newUser" />
```

Intervals are interpreted as minutes. No XML response is provided when processing files. You can optionally set a targetDirectory attribute where files will be moved when they have been successfully processed.

POP3 fetchers

The XML Gateway can retrieve create/modify XML requests from POP3 mail boxes. To set up this functionality, define a pop3Fetcher tag as follows:

```
<pop3Fetcher mailServer="localhost" user="user" password="password"
targetServer="myarserver" interval="5" template="newUser" />
```

Intervals are interpreted as minutes. No XML response is provided when processing XML sent via e-mail.

All POP fetchers will run as soon as the system starts, and will then run at their defined interval until the system shuts down. Please note that messages retrieved from the POP3 mailboxes are deleted upon retrieval, and therefore we recommend you send all XML messages to two mailboxes; one which the system uses, and one which forms your backup.

The POP fetchers also use a system of document queueing if the AR System server can not be contacted. When the fetcher runs at on it's predefined interval, it will check to see if there are documents in the queue and if so, process them before it processes any new documents from the mailbox. When documents are reprocessed, they are excluded from all statistics generated. This behaviour would be useful in a situation where the AR System server is rebooted for some reason, at a time when the system wishes to process new documents. In this event, they will be held in the queue indefinitely until the AR System server becomes available.

Please note: JavaMail sometimes has problems connecting to the host 'localhost', so please use 127.0.0.1 as the POP3 server instead.

Logging to a database

The XML Gateway can write log information to any SQL database by adding a table (detailed below), setting up an SQL server definition (see SQL datasources) and providing the name of that SQL server definition in the *databaseLoggerSQLServer* directive within the xmlgateway.xml file.

The SQL for creating the database table is as follows:

```
CREATE TABLE log (
	level integer NOT NULL,
	logger varchar(64) NOT NULL,
	message varchar(255) NOT NULL,
	lineNumber varchar(10) NOT NULL,
	sourceClass varchar(64) NOT NULL,
	sourceMethod varchar(32) NOT NULL,
	threadName varchar(64) NOT NULL,
	timeEntered datetime NOT NULL
);
```

If the connection to the database is broken while the gateway is running, it will attempt to reconnect, although some log messages are likely to have been lost.

Creating, updating and deleting records

This functionality is collectively referred to as 'create/modify' and is accessible by posting to the gateway.

Location of templates

The default templates are located in the following location:

xmlgateway/WEB-INF/classes/templates/createmodify

Templates are loaded from the classpath and can therefore be located anywhere on the classpath within the templates/createmodify directory structure.

Performing your first transaction

The gateway ships with a number of sample templates and the newUser.xml template will run with any AR System instance – it creates or updates an entry in the User form. To execute the template:

1. Go to the gateway console.
2. Click on 'Interact' and 'Post'.
3. Select the newUser template, click 'Fetch example' and press 'Post XML'.

The browser should now show the results of the post, which is an XML response. This will indicate whether the record in the User form was created or updated (the template was configured to look for an existing user), or if an error occurred. If a record was modified then the entry ID is returned.

Template structure

Start by opening the newUser.xml template in a text editor. The following is a break down of what's contained within the newUser template and hence when reviewing the examples, you should refer to the sample XML detailed in the section above.

Please note, both field IDs and names can be used when defining rules within the templates.

The root node

```
<createModify>
```

The root node of a create/modify template.

Transaction ID

This is an optional directive and mentioned here as it's used in the example.

```
<transactionID>/newuser/transactionID/text()</transactionID>
```

The location of the transaction ID. If you wish to pass a value in and out of the gateway – i.e. when you are dealing with many transactions – this is set through the transaction ID directive. You can see the transaction ID in the XML we posted above, and you will have seen it in the XML response.

Email on error

This is an optional directive and mentioned here as it's used in the example.

```
<emailOnError>xmlgwerror@javasystemsolutions.com</emailOnError>
```

An e-mail address to contact if an error occurs processing when the gateway processes a request with this template. Please note, to use e-mailing, you must setup a mail server in the applicationContext.xml configuration file.

Form definitions

```
<formDefinition returnFields="1,8">
```

The form definition specifies the start of a block of instructions to process the input XML and perform operations on an AR System form. You can have as many form definitions as you require and any form definition can use any XML from the input. This enables multiple forms to be updated through one transaction.

The returnFields attribute allows field values to be returned from an entry after a transaction. If you do not include returnFields, the default is the entry ID.

The `directive also accepts an options attribute which` is used to pass various configuration options, some of which are data source dependent. They are as follows:

- `removeDuplicates`: If a form definition generated duplicate entries then duplicates will be removed. If a form definition uses the *setFieldToUID* directive then these will be ignored when calculating duplicates.

Target object (setting the form)

```
<targetObject>User</targetObject>
```

The name of the form to be used by the form definition. **You can only define one of these within a form definition.**

Set field

```
<setField target="7">0</setField>
```

The setFIeld directive will set a field (by ID, but you can use names too) with a pre-defined value. When creating new entries in the AR System, some fields require default values and hence when designing a form definition, you may need to define them if your workflow does not do it for you.

The values are set regardless of whether an entry is being created or modified.

Setting a field to null

To set a field to null, use the setGatewayValue directive.

Field mapping

```
<fieldMapping target="109" defaultValue="0">
  <map from="Read" to="0" />
  <map from="Fixed" to="1" />
  <map from="Floating" to="2" />
</fieldMapping>
```

In the event you need to translate the input data, you can define a mapping. This is applied to the field defined in the fieldMapping element and the mappings are defined as child elements. If no mapping matches the input value then an optional default value can be defined, and if not defined then the input value is set on the field.

Setting a field by an XPath query

```
<setFieldFromXPath target="101"
expression="/newuser/loginName/text()" />
```

This directive is used to extract a value from the input XML and set it on a field using an XPath expression. This is one of the most common directives you will use to build your integration templates.

Setting a field to null

The AR System API will set a field to null when an empty string is passed, so if the expression evaluates to an empty String, this will be set in the field.

Importing attachments

```
<attachment target="536880912" type="url" />
<setFieldFromXPath target="536880912"
expression="/newuser/picture/text()" />
```

The attachment directive works alongside a setFieldFromXPath directive. The setFieldFromXPath directive provides the location for the value used by the attachment directive. The attachment can be set in three ways:

1. By passing a URL in the XML. This is the easiest way to get an attachment into the AR System and to do this, set the attachment type attribute to url.
2. By passing base64 encoded in the XML. For small attachments, you may wish to pass base64 encoded values in the XML. To do this, set the attachment type to base64.
3. By passing plain text in the xml. To do this, set the attachment type to plain.
4. By passing the attachment through a SOAP header as a MIME attachment through a webservice call to the gateway (this is an advanced option). The attachments passed in this fashion are matched against XML elements by the MIME content ID. The gateway can also accept filenames in content IDs using the syntax cID;filename (the XML must still contain just the cID).

The newUser.xml example includes an attachment example that is commented out. To enable it, uncomment the relevant section and post an XML sample that includes a picture element (a sample is included at the top of the template), i.e.:

```
<newuser>
  …
  <picture>http://www.javasystemsolutions.com/roland.jpeg</picture>
</newuser>
```

The attachment directive accepts a number of other attributes:

1. The *filename* attribute is used to specify the filename which will be used for the file created in the attachment. If no filename is present then the filename is taken from the URL or MIME attachment. If no filename can be determined then a temporary one will be assigned by the gateway.
2. The *filenameXPath* attribute is used to specify an XPath that will be evaluated to retrieve a filename from the input XML. If no filename is retrieved then the default mechanism (as specified in *filename* above) will be used.
3. The *compression* attribute is used to specify whether the attachment should be automatically decompressed. Currently, only gzip is supported. This allows a third party system to gzip attachments to reduce the amount of data passing over a network, before the attachment is decommpressed and pushed into the attachment field. If the filename ends in .gz then this will be stripped off.

Do not forget to specify the field ID of the attachment item, and not the attachment pool! To find the item field ID, view the form in the admin tool, double click on the item and go to the database tab.

Controlling the create/modify/delete functionality

```
<query>'101'="$101$"</query>
<updateStrategy>UPDATE_OR_CREATE</updateStrategy>
```

The query and updateStrategy directives are configured to control create, modify and delete operations – i.e. you can tell the gateway to look for an existing entry and update any matches or create a new one if there's no match, or only create, or only update, etc.

The query is a standard AR System query that can include x place holders that refer to fields set while processing the form definition. In the example above, a query will be executed for entries where field 101 is set to the value of field 101 as returned from the input XML. To recap, the following setFieldFromXPath was defined:

```
<setFieldFromXPath target="101"
expression="/newuser/loginName/text()" />
```

Therefore the value returned from the XPath expression /newuser/loginName/text() will be put in place of 101 before the query is executed.

The updateStrategy sets the operation to be performed. The possible values are as follows:

1. UPDATE_OR_CREATE: This is the default behaviour of a query. The query will be executed and entries returned will be updated, and if none are returned then one will be created.
2. UPDATE_ONLY: If the query returns entries then they will be updated, however if no entries are returned then no further action is taken.
3. CREATE_IF_NO_MATCH: A create will be performed if no matches were returned by the query. If results were returned then no action is taken.
4. CREATE_ONLY: A create will be performed (no query directive required).
5. DELETE: When using an AR System source, entries matched will be deleted. **Use with care!** You may wish to consider an alternative system of deleting entries involving a flag (set via an update) and an AR System escalation.

Performing multiple operations with a single formDefinition

The form definition within the newUser is only executed once. It will be more common to write a form definition that deals with many subsections of the input XML. This is achieved by setting the parentPath attribute to the form definition and this example refers to the newUsersAndGroups create/modify example template:

```
<formDefinition parentPath="/newusers/newuser">
```

This will cause the gateway to perform an XPath select on the input XML, and for each newuser element it finds, the form definition will be executed.

When doing this, the XPath directives within the form definition would be defined using relative expressions, such as:

```
<setFieldFromXPath target="101" expression="loginName/text()" />
```

This will result in the text under the loginName element for the current newuser element being set on field 101.

For example, consider the following XML (taken from the newUsersAndGroups template):

```
<newusers>
  <newuser>
    <loginName>xmlgwtest</loginName>
   ..
  </newuser>
  <newuser>
    <loginName>xmlgwtest2</loginName>
    ..
  </newuser>
</newusers>
```

The form definition will result in the XPath expression /newusers/newuser being executed, which will return two newuser elements. Each contains a set of child nodes including a login name, and hence with the directives within the form definition referring to relative elements (i.e. the login name in question), each time the form definition is executed (with a different newuser parent node), a different set of data will be extracted and used in the subsequent query/operation.

The newUsersAndGroups example provides a thorough example of how form definitions can be executed multiple times by showing how to create/update multiple users and groups through one template.

XML responses to create, modify and delete transactions

The XML gateway will return an XML document to provide information on entries created, modified or delete. After a successful transaction, the following is returned by the gateway:

```
<success>
  <object name="User" transactionID="ABC123">
    <change operation="created">CHG00123</change>
    <change operation="updated">CHG00234</change>
    <change operation="deleted">CHG00237</change>
    <error>Error description</error>
  </object>
</success>
```

The error elements are provided when the template is configured to 'resume on error'.

The following three responses will be returned by all operations to provide error messages:

- A server alias was provided that is not configured in the gateway:

```
<failure><serverNotFound>theGivenServer</serverNotFound></failu
re>
```

- Internal exception:

```
<failure><exception>Exception message from the
transaction</exception></failure>
```

- If no operations result from the processing of XML:

```
<failure><noTransactionsPerformed /></failure>
```

These responses are provided by the servlet, web services of Java Messaging Services. All failure messages will include resend="true" as an attribute of the failure node if the data source was unavailable (indicating the message can be resent later).

Additional operations

The newUser example only demonstrated a subset of the functionality available when writing create/modify templates.

dateFormat

```
<dateFormat>yyyyMMdd</dateFormat>
```

This allows you to override the standard date format for all date/datetime/timestamp fields. By default, the XML gateway will decode standard XML formats, such as:

- yyyy-MM-ddZ: Date.
- HH:mm:ssZ: Time.
- yyyy-MM-ddTHH:mm:ssZ: Date/time.

queryFieldFromXPath

This allows you to set field IDs from XPath expressions, but only for use in the query. i.e. if you want to use part of the incoming XML document in the query, but not set it on a target field when an entry is created or updated, use this directive as follows:

```
<queryFieldFromXPath target="123"
expression="/newuser/fullName/text()" />
```

Which would result in \$123\$ within the query being replaced with the result of the XPath expression /newuser/fullName/text().

setFieldFromQuery

While creating, modifying or deleting an entry, you may wish to look up data from another object in the data source. The *setFieldFromQuery* directive allows you to search for data using a query and optional parameters.

The example below performs the following operations:

1. Queries the User form.
2. Extracts the value under the loginName node and maps it to field 101 (within only the setFieldFromQuery).
3. Executes the query defined in the query tag, replacing x with values extracted using the param directive(s).
4. If an entry is returned from the query, field 8 is returned and set on the target field (103) within the current field set of the field definition.

```
<setFieldFromQuery target="103">
  <source>User</source>
  <param name="101" expression="loginName/text()" />
  <query>'101' = "$101$"</query>
  <return>8</return>
</setFieldFromQuery>
```

The expressions defined in the param directive adhere to the relative/absolute rules.

If more than one entry is returned from the query, the first entry is used to retrieve the return value.

Please note: This operation will run a query for every element processed by the parent form definition, so if the form definition matches 20 elements within the input XML, 20 queries will be run. This could have a significant impact on performance so use with care.

onError

```
<onError>RESUME</onError>
```

If your form definition has defined a parentPath, and the system finds a list of matches (and hence, will perform multiple create/modify/delete operations), this option states how the system should behave on an error. The valid values are HALT or RESUME, i.e. resume processing further entries, or halt processing matches for the current form definition and move on to the next one (if applicable).

If no onError directive is defined then HALT is used.

Please note that when RESUME is used then errors will be provided with the <success> XML response, and no <exception> response will be produced in the event of an error.

setMessage

```
<setMessage target="123" />
```

This directive allows you to set the entire message into a (text) field. If you are using a filter and wish to store the message before it was filtered (i.e. a raw Swift message) then set the attribute raw to true.

setGatewayValue

```
<setGatewayValue target="123" value="jmsid" />
```

This directive sets a field from the gateway, such as the IP of the connecting client or the JMS Message ID. If the value is not available or understood then the field remains unset. The optional restrictions attribute is used to control when the value should be set and can take two values: createOnly and updateOnly.

Values are as follows:

- remoteip: The IP address of the client (if available, i.e. not JMS).
- jmsid: The JMS Message ID.
- null: Set field to null.
- uid: Set field to unique identifier.

secure

```
<secure />
```

If you wish to force users to authenticate before they can make use of a template then the secure directive must be provided.

When templates have been secured, users will be required to authenticate using the /authentication/authenticate.do servlet, and an authentication will be required for each server a user wishes to access. The servlet works by providing the client with a session ID cookie, and that cookie must be presented on each subsequent request whether through HTTP or webservices.

When an AR System template is marked as secure, and authentication is required, subsequent operations will be executed as the user who has authenticated, and not using the credentials assigned in the relevant server configuration. However, this differs slightly to the behaviour of a directory service template which will still perform operations with the credentials defined in the server configuration.

You can see how the authentication system works by looking at the test page, where a login example is provided and functions correctly with the create/modify and query test harnesses.

This functionality is unsupported with Java Messaging Services.

Using AR System groups

The secure directive can be configured with one or more semi-colon separated AR System group names, ie.

```
<secure groups=”Incident Master;Incident Config” />
```

If an authenticated user is not in one of these groups, they will not be able to use the template.

schema

```
<schema>mySchema.xsd</schema>
```

You may wish to use an XML schema to validate incoming XML. Schema will be sourced from the `xmlgateway/WEB-INF/classes/schemas` directory. You must enter the full filename, including extension. If you do not use a schema then the incoming XML will be accepted if it valid XML.

namespaces

If the incoming XML document makes use of namespaces then they must be declared using the following syntax:

```
<namespaces>
  <namespace prefix="pr1" url="http://www.whatever.com/1.xsd" />
  <namespace prefix="pr2" url="http://www.whatever.com/2.xsd" />
</namespaces>
```

Saxon vs Xerces

We recommend you use the Saxon XPath implementation (configured in xmlgateway.xml and the default) because it's modern and provides a range of features above Xerces, such as a wider range of XPath functions.

However, while standards compliance is a positive point for Saxon and some users, others want the quickest solution to a problem. When using the Xerces parser, if the incoming XML document contains namespaces, they can be ignored by Xerces: Saxon doesn't offer this feature.

When Xerces is enabled, if no namespaces have been defined in the template, Xerces is instructed to ignore namespaces in the incoming XML document when executing XPath expressions.

If your XPath expressions contain namespaces then they will fail because the namespaces must be defined, but if you are defining namespaces in XPath expressions then adding namespace declarations is no additional effort.

responseBuilder

This feature allows one to write their own response builder in Java, by implementing a published interface. The compiled class must be in the classpath for the gateway, and to use this feature simply define the fully qualified classname in the responseBuilder directive. For example:

```
<responeBuilder>
com.mycompany.xmlgateway.CustomResponseBuilder
<responseBuilder>
```

filter

This feature enables one of the inbuilt filters that are used to translate an input stream into an XML Document - hence, allowing any data to be submitted to the gateway and converted into XML before being processed. The following standard filters exist within the gateway:

CSV

The CSV filter will translate a CSV document into an XML document with the following structure:

```
<root><row index="0">
  <col index="0">Data</col>
  <col index="1">Item</col>
</row><row index="1">
  <col index="0">Data</col>
  <col index="1">Item</col>
</row></root>
```

Each row and/or column can be uniquely identified by the index attribute.

JSON

The JSON filter will translate text formatted as a JSON string into an XML document that can then be posted to a template. This isn't a perfect process as it's difficult to convert pure JSON into XML without further information, such as inventing a notation for setting XML attributes instead of elements and text content. The real benefit of this filter is the output/query functionality, converting XML to JSON.

Swift

The Swift filter will translate standard Swift messages into an XML document with a fairly complex structure. To view the structure we recommend you go to the 'Test filters' page on the web interface, insert a raw Swift message and look at the output of the filter.

An AR System 'def' file has been provided along with a template (swiftMessage.xml) that will allow you to push Swift messages into the AR System.

filter-class

Bespoke filters can be written by implementing the filters interface. To use your own filter, simply provide the classname in this directive.

Impersonating a different AR System user

The AR API contains functionality to impersonate a user, available to connections using an account with administrator privileges. If you wish to use this functionality, you can point to a username in the input XML using the following syntax:

```
<impersonate>/path/to/user</impersonate>
```

Checksum

Create a checksum from a set of input values, allowing updates to be dropped if the data within the database is the same as the incoming data.

The checksum is generated using a gateway XPath function in conjunction with the checksum directive. A template called newUserChecksum.xml is included with the gateway and this example is taken from it:

```
<checksum field="checksum"
expression="jss:checksum(/newuser/checksumvalue1/text(),/newuser/che
cksumvalue2/text())" />
```

In this example, the field called checksum will be checked before an entry is updated, and if the existing value is identical to the new checksum value (generated from the values retrieved from the two XPaths), the update will not be performed.

You can specify any number of XPaths in the checksum function.

Functions

XPath provides a range of functions and Saxon implements many of them; a good resource can be found here:

http://www.w3schools.com/Xpath/xpath_functions.asp

However, the gateway provides a number of functions to provide more bespoke functionality. They are detailed below.

Checksum

Create a checksum from a set of input values. This is used in conjunction with the checksum directive described above.

Encrypt

Encrypt the value before setting on the input field. The value can be decrypted using the query option decrypt.

This is not a strong encryption routine and is only provided to mask values.

The following provides example usage:

```
<setFieldFromXPath target=”fieldName”
expression=”jss:encrypt(/path/to/node/text()” />
```

GUID

Generate and return a unique ID.

Querying the AR System

This functionality is accessible by posting to the gateway through the test interface, as well as through webservices and JMS.

There is a graphical query template designer available through the gateway portal and we recommend you use this to build query templates.

Location of templates

The default templates are located in the following location:

xmlgateway/WEB-INF/classes/templates/queries

Templates are loaded from the classpath and can therefore be located anywhere on the classpath within the templates/queries directory structure.

Performing your first transaction

The gateway ships with a number of sample templates and the sampleQuery.xml template will run with any AR System instance – it queries the User form. To execute the template:

1. Go to the gateway console.
2. Click on 'Interact' and 'Query'.
3. Select the sampleQuery template, press the fetch example button and then press 'Perform query'.

The browser should now show the query results presented as an XML document.

Template structure

Start by opening the sampleQuery.xml template in a text editor. The following is a break down of what's contained within the sampleQuery template and hence when reviewing the examples, you should refer to the sample XML detailed in the section above.

Please note, both field IDs and names can be used when defining rules within the templates.

The root node

```
<query type="ars">
```

The root node of a query template. The type ars declares the template as one that connects to AR System server datasources.

Search criteria

```
<configuration>
  <user key="userFormSearch">
    <searches><search>
      <server>myarserver</server>
      <form>User</form>
      <maxResults>2</maxResults>
    </search></searches>
  </user>
</configuration>
```

The configuration block holds a set of different configurations for the template, and each one contains a list of searches. Each configuration is referred to by the <user key=”xx”> element.

Please note, the <user key=”xx”> is poor terminology and will be reviewed in a future version of the gateway.

When a query request is processed, the configuration is retrieved and the searches executed in sequence until one matches – the results of this search are used to build the XML response. This functionality allows developers to build multiple types of searches into one template, for example, there may be two forms holding user data and the developer may wish to search both to provide the result set.

Each search can contain pre-defined values, and if not defined then they must be presented on the request. **The pre-defined value will always take precedence over a value submitted in the query request**. The following values can be defined:

1. server: The name of the AR System server.
2. form: The name of the form to query.
3. query: A query string to execute.
4. maxResults: The maximum number of results to return, set on the AR API call.
5. startRow: Return results from this row number.
6. endRow: Return results up to this row number.
7. sorts: The results can be sorted and this is described below.

Specifying the XML response template

```
<responseXPath rootExpression="/query/sample"
               resultExpression="/sample/result" />
```

This directive tells the gateway where to find the template XML response within the query template by providing two XPaths:

- rootExpression: This is relative to the root node and points to the node that will be used as the root of the XML response.
- resultExpression: This is relative to the root of the XML response. It will be recursively copied for each result.

The directive also accepts the transactionIDExpression attribute which is an XPath expression that selects a node for the transaction ID. This is optional and ties in with the transactionID given in an XML based query request.

Specifying a result field

```
<result field="7" expression="status" options="toText" />
```

This tells the gateway to place the value for field 7 under the Xpath specified in the expression. The expression is **relative to the resultExpression defined in the responseXPath**. The directive accepts the following attributes:

- `field`: Field or column to be retrieved from the current result set.
- `expression`: XPath expression of value to set in response, *relative to the result root, defined in the resultXPath directive*.
- `options`: The options attribute is used to pass various configuration options:
  - `attachmentName`: If the field is that of an attachment then this option will set the name of the attachment at the given expression.
  - `attachmentLength`: If the field is that of an attachment then this option will set the length of the attachment (in bytes) at the given expression.
  - base64: Base64 encode the value, particularly useful for binary values such as an Active Directory objectGUID.
  - `cdata`: This option will result in the output data being wrapped in a cdata element.
  - decrypt: Decrypt a value if it was encrypted when pushing it into the AR System. See the create/modify encrypt option.
  - epoch: This option can be used when outputing date/time fields. Instead of the date being formatted as a string, it will be output as the number of milliseconds since 1970 (the epoch value).
  - removeControlCodes: This option removes control codes that may be present in a value before it is inserted into the XML document. Control codes are not valid in the XML 1.0 specification yet some DOMs (such as Saxon) do not remove them, and the output can then break parsers (such as the one in Firefox or Chrome). There is a small performance hit to using option so only use it when necessary.

    - replace: The node referenced by the result expression will be removed and replaced with the text. This is useful when you want to use nodes to act as place holders for text.
    - `toText`: This option can be used on various field types in order to convert an internal AR System numeric value into the string equivalent. The following type conversions are supported:
        - Drop down menu fields, such as status, result in the enumerated value being converted to a string.
        - Group list fields, such as 104, are converted from semi-colon separated group Ids to semi-colon separated group names.
    - `trim`: White space will be removed from the field before it is output. The trim happens before splitting occurs.
- `dateFormat`: This attribute can be used on date, time and date/time fields in order to convert an internal numeric date into a human readable date. The attribute requires a valid Java date format.
- `timezone`: When using the dateFormat attribute, the timezone can be used to force the date to be rendered in a particular timezone. If not provided then the machine timezone is used. If provided but not understood then GMT is used.
- `numberFormat`: This attribute can be used on numerical types in order to format the output. The attribute requires a valid Java number format.
- `splitSize`: This accepts a numeric value and will result in a string being split into substrings of a length equal to the given value. The output node is then cloned for each substring, with each cloned node being placed directly after the original.

Additional operations

Email on error

```
<emailOnError>xmlgwerror@javasystemsolutions.com</emailOnError>
```

An e-mail address to contact if an error occurs processing when the gateway processes a request with this template. Please note, to use e-mailing, you must setup a mail server in the applicationContext.xml configuration file.

Namespaces

If the query response XML makes use of namespaces then they must be declared using the following syntax:

```
<namespaces>
  <namespace prefix="pr1" url="http://www.whatever.com/1.xsd" />
  <namespace prefix="pr2" url="http://www.whatever.com/2.xsd" />
</namespaces>
```

Securing a template

```
<secure />
```

Please refer to the create/modify documentation as the operation is identical in both processes.

Validating against a schema

```
<schema>mySchema.xsd</schema>
```

Please refer to the create/modify documentation as the operation is identical in both processes, however in query templates, this will validate the XML response against the schema.

Changing the HTTP content type

```
<contentType>text/html</contentType>
```

This directive allows you to change the content type (from text/xml) set on the HTTP response. You may wish to do this if you've written a template (and optionally an XSL translation) that creates an HTML page, and you wish the browser to render as HTML.

Query statistics generated by the gateway

```
<statistic type="startTime" expression="/sample/startTime" />
```

The gateway exposes various statistics during the query process and they can be accessed by using the statistic directive given two attributes:

- expression: XPath expression of value to set in response.
- type: The type of statistic, detailed in the following list:
  - startTime: The time at which the query process started.
  - endTime: The time at which the query process ended..
  - numberOfResults: The number of results generated by the query.
  - totalResultCount: The number of results generated by the query if startRow/endRow had not been provided.
  - query: The query executed against the data source.

The startTime and endTime statistics support the optional parameters dateFormat and timezone, which operate in exactly the same way as they do on result fields returning dates. I.e.

```
<statistic type="startTime" expression="/sample/startTime"
dateFormat="hh:mm dd/MM/yyyy zzz" />
```

Removing elements

```
<removeElement field="536870934" expression="diary" condition="null"
/>
```

The removeElement directive can be used to remove nodes from the response based on a field and condition. It accepts three parameters:

- field: Field to evaluate from current result.
- expression: XPath expression of the node to remove, relative to the result root.
- condition: The condition on which the node should be removed. The condition attribute currently accepts two values: 'null' and 'not null'

AR System server info and statistics

```
<arServerValue field="1" type="info"
expression="/sample/serverInfo/dbType" />
```

The AR API defines over a hundred server information items, each with a unique index. You can query these (on ARS query templates) by using the arServerValue

- field: API field/constant value.
- type: Either info or statistic.
- expression: XPath expression of value to set in response.

Attachments

```
<attachment field="536880912" expression="picture/link"
baseURL="http://localhost:8080/xmlgateway"
action="view" />
```

Retrieving AR System attachments can only be achieved through a client or by writing a program making use of the AR System API. However, the gateway includes support for attachment retrieval through the query interface. The attachment directive is used to provide an HTTP link to an attachment, or encode it within the output XML. The following attributes are used to configure the action:

- field: Attachment field.
- expression: XPath expression of value to set in response.
- action:
    - If you wish to encode the attachment into the XML, set this to base64.
    - If you wish to publish an HTTP URL, there are two options available to you because different HTTP headers must be sent to a browser to produce the required behaviour.

    - If you wish to make the browser either display an image in a separate window,, then set this attribute to view.
    - If you wish to make the browser open a 'save as' dialog box, set this attribute to download.
- baseURL: This is only relevant when action is set to download or view and allows you to set the full URL to the gateway application. Acceptable examples include /xmlgateway (relative) and http://myserver/xmlgateway (absolute).

When a URL is generated, the gateway generates an attachment 'key' that must be presented by the client to retrieve the attachment. The lifetime of this key is configured in the xmlgateway.xml configuration file.

Diary fields

```
<diary field="536870934" entries="all" expression="diary"
       options=”...”>
  <item expression="time" type="timestamp"
        dateFormat="hh:mm:dd/MM/yyyy zzz"
        timezone="America/Los_Angeles" />
  <item expression="user" type="user" />
  <item expression="text" type="text" />
</diary>
```

In AR System, diary fields contain a list of entries, each with a timestamp, user and block of text. The diary directive allows the extraction of individual components of a diary field. The diary directive accepts the following attributes:

- field: Field to be retrieved from the current result.
- entries: Which entries to output - first, last or all.
- expression: XPath expression of the diary node, relative to the result root. When multiple diary entries are output, this node will be cloned for each result.
- options: If 'reverse' is passed into the options attribute, the diary entries will be evaluated in reverse. This is useful in conjunction with 'all' being passed to the entries attribute. You may also pass the options available when specifying a result field (i.e. base64, removeControlCodes, etc).

The diary directive can contain multiple item directives that map XPath expressions to an individual diary component. The following attributes can be set on an item:

- type: The diary item value type - timestamp, user or text.
- expression: XPath expression of value to set in response, relative to the node identified by the expression attribute of the diary directive.

- dateFormat: This attribute can be used on timestamps in order to convert an internal numeric date into a human readable date. The attribute requires a valid Java date format.
- timezone: When using the dateFormat attribute, the timezone can be used to force the date to be rendered in a particular timezone. If not provided then the machine timezone is used. If provided but not understood then GMT is used.

In the event of multiple diary entries, the root diary node is repeated for each entry. An example diary output follows:

```
<diary>
  <time>23/6/2006 10:30<time>
  <user>Fred<user>
  <text>User has a new printer<text>
</diary><diary>
  <time>22/6/2006 9:42<time>
  <user>Bill<user>
  <text>Printer has been ordered<text>
</diary>
```

Retrieving field properties

```
<arsProperty field="109" property="AR_DPROP_ENUM_LABELS"
expression="/sample/properties/licenseTypeLabels" />
```

If you wish to retrieve field properties then this can be achieved using the arsProperty directive. The field and property are provided as attributes, and an absolute XPath expression that states where the value (or values) will be output. The property value can either be a reference to a Java variable name within the AR API, or a numeric value - a complete list of properties can be found in the AR API guides.

If a field has multiple values for a property (for different views), then they will all be output at the given XPath location through multiple property elements. Each property element will have an attribute (view) containing the view ID, and the values will be placed in value elements with the enumerated field ID set as the arid attribute.

This example is taken from the complexQuery.xml query template (detailed below) and was generated using the arsProperty directive listed above.

```
<licenseTypeLabels>
  <property view="20002">
    <value arid="0">Read</value>
    <value arid="1">Fixed</value>
    <value arid="2">Floating</value>
    <value arid="3">Restricted Read</value>
  </property>
</licenseTypeLabels>
```

Complex query example

The complexQuery.xml template provides a range of advanced querying functionality and can be found in the standard location for query templates. We recommend this is reviewed as it runs against the User form so forms a good basis for building your own advanced templates.

Sorting results

```
 <searches><search>
   <server>myarserver</server>
   <form>User</form>
     <maxResults>3</maxResults>
     <sorts>
       <sort>
       <field>Login Name</field>
       <order>desc</order>
     </sort><sort>
       <field>Create Date</field>
       <order>asc</order>
      </sort>
    </sorts>
</search></searches>
```

The results of a query can be queried using the AR System API functionality through the structure demonstrated above. You can specify as many columns as you wish and set the order to desc or asc.

Parameterised queries

To keep the query requests as simple as possible, parameters can be passed through the request and placed into the query executed, allowing the template author to hide the complete query from the process querying the gateway. The parameters can also be placed into the output XML.

The following directive illustrates how to pass a value with a parameter called author:

```
<query>
  <template>templateName</template>
  <configKey>userQuery</configKey>
  <searches><search><parameters>
    <parameter name="author" value="Dan B" />
    <parameter name="city" value="Texas" />
  </parameters><search><searches>
</query>
```

To use this parameter in the query (with a AR System data source) then the author parameter is used as follows (note the syntax :author within the query):

```
<configuration>
  <user key="userQuery"><searches><search>
    <server>myarserver</server>
    <form>User</form>
    <query>
      'Login Name' = ":author" and 'Description' = ":city"
    </query>
  <search><searches></user>
</configuration>
```

If you also wish to place the parameter in the output XML then use the following syntax:

```
<searchParameter name="author" expression="/query/theAuthor" />
```

The complexQuery template demonstrates this functionality.

Chaining query templates

You may wish to query multiple forms in one transaction, for example, when retrieving a list of incidents with associated tasks and worklogs (both of which are stored in separate forms to incidents in BMC ITSM). The following directive allows you to specify a call to another query template while processing entries:

```
<callTemplate template="incidents-worklogs" key="search"
              expression="worklogs" fields="Incident Number" />
```

This means, for each result processed, execute the query template called incidents-worklogs, use the configuration key search, pass the field Incident Number (from the current entry) as a parameter to the template, and place the results under the worklogs element.

In the incidents-worklogs template, a search is defined as follows:

```
<searches><search>
  <server>myarserver-appadmin</server>
  <form>HPD:WorkLog</form>
  <query>":Incident Number" = 'Incident Number'</query>
</search></searches>
```

You can see the Incident Number has been defined as a parameter to the search (using the syntax :Incident Number), and hence the query will return all worklog entries for the given incident.

An example of retrieving incidents with worklogs and tasks is provided through the incidents.xml template (which calls the incidents-worklogs.xml and incidents-tasks.xml templates). There is also a video on the website that demonstrates the functionality.

Encouraging portable templates

To avoid having to define the server in the template being called, the server used for the original query will be passed into the query. However, you can override this by defining the server in the child template.

XSL Transformations

XSL transformations allow you to build complex XML documents that are beyond the scope of the gateway, which is designed to simply extract data in a logical fashion.

If you wish to perform an XSL transformation on the XML output through the query, you can do this by using the xslt directive. Your xslt files should be stored in a directory called xslt in the classpath – i.e. WEB-INF/classes/xslt.

For example, if you had an xslt file called convertRSS.xslt, you'd use the directive as follows:

```
<xslt>convertRSS.xslt</xslt>
```

We provide an example RSS feed of ITSM incidents that's built up from a query template using an XSL transformation. The template is called incidentsRSS.xml.

There's many good XSLT tutorials on the web and one can be found here:

http://www.w3schools.com/xsl

Filters

This feature enables one of the inbuilt filters that are used to translate the query output from an XML document into another form - hence, allowing data to be transformed from XML into another format. Filters are applied on a per configuration key basis. This is the inverse of the filter configuration defined in the create/modify documentation so please refer to it if considering using filters.

The following standard filters exist within the gateway:

CSV

The CSV filter will translate a pre-defined XML format into CSV. The format required is as follows:

```
<root> <row index="0">
  <col index="0">Data</col>
  <col index="1">Item</col>
</row><row index="1">
  <col index="0">Data</col>
  <col index="1">Item</col>
</row></root>
```

Each row and/or column can be uniquely identified by the index attribute.

Please see the sampleQueryCSV template for a working example.

JSON

The JSON filter will translate an XML document into a JSON string. An example can be found in the sampleQueryJSON.xml template. Running this through a browser will cause a file download prompt because the filter element is set with a filename (query.json).

Swift

The Swift filter uses the Open Source WIFE project to generate XML output. Please refer to the WIFE XML documentation for details of the output format.

Setting file download headers

When defining a filter in a query template, the *filename* attribute can be used to set a filename that will be set in the HTTP headers. This causes a browser to open a 'save as' dialog, which is a useful addition to those building web interfaces around the gateway.

filter-class

Bespoke filters can be written by implementing the filters interface. To use your own filter, simply provide the classname in this directive and ensure the class is present on the Java classpath.

Template based filters

Query templates can be configured to run with a filter through a configuration or globally.

Globally

```
<query type="ars">
  <filter>csv</filter>
```

Configurations

```
<configuration>
  <user key="userQuery"><searches><search>
   <filter>csv</filter>
   <server>myarserver</server>
```

Setting filter on query request

If a template has not got a filter defined on a configuration or globally, the query request can contain a filter element to tell the XML Gateway that the data should be converted into a different output format, ie.

```
<query>
  <template>templateName</template>
  <configKey>configurationKey</configKey>
  <filter>csv</filter>
  …
</query>
```

JSON based queries

Whilst the XML Gateway is primarily an XML based platform, some third party applications would like to communicate using JSON. The most common request is to perform a query, through a JSON based request, and receive results in XML or JSON.

There is no exact science in converting an XML structure to JSON so the filters page within the web interface provides functionality to convert XML into JSON. This can forward the browser to the query form so it can be executed against the appropriate template.

If a third party application wishes to use JSON, the test query page can be used to determine how the HTTP call should be made.

The AR System impersonate feature

It is possible to use this feature via a query request, assuming the template has been configured to run with an AR System administrator user. To do so, use the following syntax in bold:

```
<query>
  <template>templateName</template>
  <configKey>configurationKey</configKey>
  <impersonate>user</impersonate>
  …
</query>
```

Securing templates through client authentication

The Gateway Authentication supports AR System servers and Directory Services..

The system allows templates to be restricted to AR System or Directory Service users, and ensures that operations are run as the authenticated user, and not the user defined in the XML Gateway configuration file. On the test page, you can see an example form that makes use of the /authentication/authenticate.do servlet. Templates are restricted by using the secure directive (detailed in the manual pages).

The sessions are maintained by the standard Java Servlet engine cookie, commonly called JSessionID. After authenticating with the servlet (or via webservice call to the authenticate function), the client must present this cookie on all subsequent requests to the gateway. Failure to do so will mean the gateway is unable to look up the session, and hence, the request will be denied because the template has been marked as secure.

When you access the servlet from outside of the XML Gateway test page, you will get an XML response providing the results of the authentication challenge. The response will either be:

```
<success sessionID="12345" />
```

for a successful authentication, or:

```
<exception>Message</exception>
```

If the username and/or password was invalid for the authentication source.

HTTP URLs exposed by the gateway

The gateway web interface demonstrates all the HTTP URLs exposed by the gateway. Whilst the web interface can be used to look at how the operations work, the majority are summarised below.

Performing create/modify operations

/pushxml/createmodify.do:

- template: Name of create/modify gateway template. If not passed then the HTTP bindings will be consulted for a template based on source IP address.
- server: The server in which operations will be performed. If not passed then the HTTP bindings will be consulted for a server based on source IP address.
- xml: The XML to process. If not passed then the XML will be read from the raw input stream, ie allowing a third party to post raw XML into the servlet.
- view: Optional Spring view to which the servlet will forward. The views should be a reference to a JSP files without the .jsp extension.

/pushxml/createmodifyupload.do:

- template: As above.
- server: As above.
- file: Multipart file containing the xml to process.
- view: Optional Spring view to which the servlet will forward. The views should be a reference to a JSP files without the .jsp extension.

Performing query operations

/pushxml/query.do:

- xml: The XML to process. If not passed then the XML will be read from the raw input stream, ie allowing a third party to post raw XML into the servlet.
- json: If provided, the JSON text will be converted to XML using the reverse of the process used by the form on the filters page to generate JSON from XML. This parameter allows third party clients to send JSON requests, perhaps when they are not capable of sending an XML request.
- fiilename: If supplied, the HTTP download headers will be set causing the browser to open a 'save as' dialog. This is useful for building web interfaces around the gateway.

/querybyparameter/arsystem.do:

- template: Name of query gateway template. If not passed then the HTTP bindings will be consulted for a template based on source IP address.
- configkey: Optional template configuration key.
- server (optional): The server to query.
- form (optional): The form to query.
- p-x: One or more parameters can be set on the query request by passing them as HTTP parameters with the key set to p-. ie if you wish to set the query search parameter loginname to dkellett, you would pass p-loginname=dkellett.
- fiilename: If supplied, the HTTP download headers will be set causing the browser to open a 'save as' dialog. This is useful for building web interfaces around the gateway.

Authentication

/authentication/authenticate.do:

- server: The server on which you wish to authenticate.
- user: The username.
- password: The password.
- auth: AR System authentication string.
- view: Optional Spring view to which the servlet will forward.

/authentication/unauthenticate.do:

- view: Optional Spring view to which the servlet will forward.

Integrating with the gateway with other systems

The gateway's template and XML management is separated from the connectors, and a number of connectors are supported out of the box. The gateway is also heavily dependent on the open source Spring Framework (http://www.springframework.org) and hence developers can re-use existing components to build integrations – the JMS connectivity is almost entirely based on Spring.

The connectors are listed below:

1. HTTP: Servlets and webservices are exposed. The servlets are used by the web test harness and hence it's easy for developers to undestand how they function. The webservices are described below and URLs to the WSDLs are provided in the web interface.
2. Java Messaging Services: The gateway provides a number of JMS tasks to perform operations.
3. File fetchers: Defined in the xmlgateway.xml file, these are tasks that run every X minutes, pulling xml/csv files from a directory and processing the data.

Java Messaging Services

JMS form the core messaging infrastructure in many large organisations, and essentially enable an enterprise to build fault tolerant messaging system allowing different systems to exchange information. More information on JMS can be found at the Sun JMS homepage. The XML Gateway can integrate with various J2EE servers providing JMS (version 1.1) functionality.

Before you can connect to a JMS, you must setup the Java Web Server running the gateway to communicate with the JMS (instructions to cover common cases are provided in the XML Gateway configuration files). This usually involves copying some vendor specific JMS jar files from the JMS server product into the Java Web Server running the XML Gateway.

Integration technique for create/modify/delete

The gateway binds to an inbound JMS queue (or topic, but does not remove messages!) and is configured to use a specific template and server. It may also bind to an outbound JMS queue (or topic) to which the XML Gateway response will be sent. When a message arrives, it is processed with the template and server configured, and if an outbound queue has been configured then the XML response will be published.

Given JMS is an asynchronous messaging system, an external 'transaction ID' is supported to allow an ID (of some kind) to be taken from the inbound XML response and set on the XML response. This allows third

party systems to reconcile a source message and a response that will arrive at some point in the future.

Integration technique for queries

The process for integrating a JMS for querying functions is almost exactly the same as it is for create/modify/delete. The only difference is the number of configuration items set up in the XML Gateway configuration file.

Configuration files

JMS configuration is not an easy procedure and we suggest that you talk to Java System Solutions support if the following information requires further clarification.

There are two Spring configuration files shipped with the gateway: jms-core-context.xml and jms-jobs-context.xml. They are not loaded by default so the first task is to enable them – we recommend you start with just the jms-core-context.xml as if the Spring file is not configured correctly, the gateway may not start when Tomcat is restarted and hence configuring one file at a time reduces the amount of potential debugging.

To load the JMS configuration, locate the file web-application-context.xml (in WEB-INF/classes) and uncomment the first resource so it looks like this:

```
  <import resource="jms-core-context.xml" />
  <!-- import resource="jms-jobs-context.xml" /-->
```

The jms-core-context.xml now needs configuring.

Core configuration

The Spring configuration file jms-core-context.xml defines the core JMS modules required for interaction with a messaging server.

Datasources

If you're running the gateway outside of an application server, you will need to configure the following bean:

```
  <bean id="jms.core.JNDITemplate"
class="org.springframework.jndi.JndiTemplate">
    <property name="environment">
      <props>
        <!-- Tibco configuration -->
        <prop key="java.naming.factory.initial">
          com.tibco.tibjms.naming.TibjmsInitialContextFactory
        </prop>
        <prop key="java.naming.provider.url">
          tibjmsnaming://localhost:7222
        </prop>
```

```
      </props>
    </property>
  </bean>
```

This configures the Java Naming Directory Interface (JNDI), which tells it how to connect to the messaging server. You will almost certainly need to deploy a vendor driver jar file to the gateway WEB-INF/lib directory. The properties are also vendor specific, so please consult the vendor's documentation.

If you are running the gateway within an application server, such as Weblogic, Websphere, JBoss, etc., then you may wish to not configure this bean and look up the JMS datasource directly from the JNDI (if you configured it through the application server).

Connection factories

A connection factory is required for each messaging server to which you want to connect. The following defines a connection factory:

```
  <bean id="jms.core.connectionFactory"
        class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiTemplate" ref="jms.core.JNDITemplate" />
    <property name="jndiName"
value="com.javasystemsolutions.connectionFactory" />
  </bean>
```

Please note, if you have not defined a datasource (because the application server provided one) then you do not need to specify the jndiTemplate property in bold.

Defining a queue

To define a queue, we define two beans – one which is merely a pointer to the queue, and another which provides management facilities on the queue (a Spring JmsTemplate) and is used heavily by the gateway. The name of the queue is highlighted in bold:

```
 <bean id="jms.core.destination.createModifyIn"
        class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiTemplate" ref="jms.core.JNDITemplate" />
    <property name="jndiName"
value="com.javasystemsolutions.createModify.InQueue" />
  </bean>
  <bean id="jms.core.template.createModifyIn"
class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory"
ref="jms.core.connectionFactory" />
    <property name="defaultDestination"
ref="jms.core.destination.createModifyIn" />
  </bean>
```

Please note, if you have not defined a datasource (because the application server provided one) then you do not need to specify the jndiTemplate property defined in the JndiObjectFactoryBean.

Referring to templates from gateway configuration

The JmsTemplates are referred to by various integration points, such as the 'query and forward to a Java Messaging Service' webservice or the 'query and send to JMS' scheduled job. When referring to a template, you use the ID of the bean – i.e. the text highlighted in bold:

```
  <bean id="jms.core.template.createModifyIn"
class="org.springframework.jms.core.JmsTemplate">
```

Webservices exposed by the gateway

The gateway now *also* incorporates Apache CXF, an actively maintained project that provides a SOAP stack and a range of other features (such as REST based services). While the gateway primarily makes use of the SOAP stack, other CXF functionality will be made available in a later release.

In a previous version of the gateway, AR System integration had to be performed with the Axis SOAP stack built into the gateway. This is being removed in favour of CXF. Please report any issues with being unable to migrate to CXF, which should simply involve re-creating the filter set fields with the CXF WSDL.

Locating the WSDLs

There are a number of WSDLs and they are exposed through both SOAP stacks. These are all linked through the portal page (under documentation).

- xmlgateway.wsdl: Provides a range of XML Gateway functionality such as querying for XML, posting XML and authenticating with the gateway. This is the only WSDL you'll likely to require when integrating with AR System.
- forwards.wsdl: Provides a range of data forwarding functionality (send a String to a JMS destination, URL, another gateway, file, etc.). This functionality is largely wrapped by the XMLGateway webservice but is useful standalone and hence exposed to the user.

Important note

WSDLs contain the location of the SOAP server and the WSDLs are usually produced with the hostname set to localhost. Therefore, your client code will have to change the location unless it's running on the local machine (which is very unlikely). If you are integrating with AR System then you will definitely have to obtain a WSDL with a modified wsdl:location, so AR System knows where to find the SOAP server.

To obtain a WSDL and modify the location, follow these steps:

- Go to the WSDL (using a link above).
- Save it to a local file.
- Open the WSDL in a text editor.
- Search for *wsdl:location* and change the hostname.

However, the location can be configured to your environment for **CXF**. The file `xmlgateway/WEB-INF/classes/cxf-server-context.xml` contains instructions on how to configure the location (search for 'address').

Services provided by the XML Gateway WSDL

To obtain the XML Gateway WSDL, use one of the links above.

Create and modify

The function returns the standard XML response string.

The post function allows you to post XML into the gateway and takes the following three parameters:

1. template: The name of the template to use.
2. server: The name of the AR System server to use.
3. xml: The XML document to process.

The postThroughBindings method also allows you to post XML into the gateway using the HTTP create/modify bindings to specify a template and server mapped to the source IP address. This works in the same way as the /pushxml/createmodify.do servlet. The method only takes one parameter:

1. xml: The XML document to process.

Both the postThroughBindings and post methods can read attachments from the SOAP headers, which can then be matched to the XML via the MIME content ID. To make use of this feature, see the attachment tag in the create/modify manual for more details.

The method returns the standard XML response string.

Querying

The *query* method will perform a query and return the XML query response. The method takes one parameters:

1. xml: XML based query.

Querying through HTTP Query bindings

The queryThroughBindings method allow a client to query the gateway using the HTTP Query bindings to specify a template and authorisation key mapped to the source IP address. This works in the same way as the /pushxml/query.do servlet and takes just one parameter:

1. xml: The XML query document.

Query and forward to URL

If you wish to perform a query and send the results directly to a listening service, you should use the *queryAndForwardToURL* method. The following two parameters are required:

1. xml: XML based query.

2. targetURL: The URL target to which results will be sent.

At present, HTTP and file URL targets are supported, and the method will write the results directly to the third party using an HTTP post, or directly to a file.

The function will return true or false indicating the success of the transaction.

Query and forward to another XML Gateway

XML Gateway instances can be used in clusters to replace AR System functionality such as DSO. You may perform a query and have the results posted directly to another XML Gateway server through the normal create/modify mechanism, and the method will return true if the transmission was successful.

The function to support this functionality is called *queryAndForwardToXMLGateway* and takes the following four parameters:

1. xml: XML based query.
2. targetURL: The target URL of the remote XML Gateway (such as: http://anotherserver:8080/xmlgateway/pushxml/createmodify.do).
3. targetTemplate: The create/modify template to be used on the remote XML Gateway.
4. targetServer: The target AR System server to be used on the remote XML Gateway.

While we appreciate it is a little long winded to put in such a long target URL, we feel this is the most generic way of providing this functionality. After all, you may have changed the application context path or servlet location! But in practice, we expect the URL example we have provided will be sufficient.

The function will return true or false indicating the success of the transaction.

Query and forward to a Java Messaging Service

If you wish to perform a query and send the results directly to a JMS, you should use the *queryAndForwardToJMS* method.

The first step is to configure JMS, which is covered in a separate section of this documentation.

While setting up JMS, you will define one or more destinations (queues or topics) and a Spring JmsTemplate for each destination. Each JmsTemplate is given an ID – i.e., from the example jms-core-context.xml file, the ID is in bold:

```
<bean id="jms.core.template.createModifyOut"
class="org.springframework.jms.core.JmsTemplate">
```

Typically, you will define a destination with a meaningful name and another third party process will subscribe and process the XML messages as they are published.

The webservice method is very easy to use and takes two parameters:

1. xml: XML based query.
2. The ID of a Spring JmsTemplate.

The function will return the JMSMessageID if the transaction was successful.

Query and forward to a message sending destination

The message sending framework contains Destinations that can be invoked with this webservice. The Destination must be declared in the Spring context and the bean ID passed into the webservice.

This is beneficial because you can easily define bespoke Destination objects within the Spring configuration and reference through workflow. Here is an example of the HTTPDestination being configured in Spring, the ID is in bold and it defines a custom connect/read timeout of 30s (which is a very high figure and used for illustrative purposes):

```
<bean id="webservice.http.mydestination"
  class="com.javasystemsolutions.xml.gateway.messagesending.HTTPDestination">
  <property name="url" value="http://localhost:8181/target" />
  <property name="timeout" value="30000" />
</bean>
```

The function to support this functionality is called *queryAndForwardToDestination* and takes the following four parameters:

1. xml: XML based query.
2. destinationSpringID: The ID of the Spring bean that defines a Destination.

The function will return true or false indicating the success of the transaction.

Query ARS and forward to message sending framework

The *queryARSAndSend* method runs an AR System query and sends the results to the message sending framework. The following six parameters are required:

1. template: The name of the template to use.
2. authKey: The authorisation key to use.
3. server: The name of the AR System server to use.
4. form: The AR System form to query.
5. query: The AR Syste, QBE query to executed.
6. target: Name of message sending target.
7. handler: Name of message sending handler.

8. schedule: Name of message sender scheduler.
9. attemptSyncSendFirst: Whether a synchronised send should be attempted. If set to false then a schedule should be provided. If true then an attempt will be made to send the message (and decode the response) synchronously, before attempting asynchronously if a schedule has been provided.

The function will return a value from the message sending framework.

Query ARS and invoke a Plugin

This functionality should no longer be used. The message sending framework, scheduler and queryAndForwardToDestination webservice provide a much better solution.

The *queryARSAndInvokePlugin* method will perform a query against the gateway and pass the results into a webservice plugin. A webservice plugin can be written by anyone with Java skills using an interface provided with the gateway, and allows higly bespoke functionality to be bolted onto the gateway with ease. The following seven parameters are required:

1. template: The name of the template to use.
2. authKey: The authorisation key to use.
3. server: The name of the AR System server to use.
4. form: The AR System form to query.
5. query: The AR Syste, QBE query to executed.
6. plugin: The fully qualified class name of the Plugin implementation.
7. param1-param10: A set of parameters that will be passed to the Plugin. An array would have been used however it appears the AR System can not consume a WS and pass attributes to an array!

Refer to the Developing plugins documentation for more information on using this powerful functionality.

Combined post and query

The *postAndQuery* function combines the post and query webservice functions into one function. It will process the input XML and then query the gateway, returning the XML response.

1. template: The name of the template to use.
2. server: The name of the AR System server to use.
3. postXML: The input XML document to process.
4. queryXML: The XML query based request.

Combined query and post

The *queryAndPost* function combines the query and post webservice functions into one function. It will perform the query and then post the response using the template and server provided.

1. queryXML: The XML query based request.
2. template: The name of the template to use.
3. server: The name of the AR System server to use.

Authenticate

The *authenticate* method allows a client to authenticate with the gateway so it may access secured templates.

1. serverAlias: XML Gateway server alias.
2. username: AR System user or Directory Service principal.
3. password: AR System password or Directory Service credentials.
4. authentication: AR System or Directory Service authentication.

The method returns a session ID if successful. The client must also retain the JSESSIONID cookie to maintain the session through subsequent operations.

Unauthenticate

The *unauthenticate* method clears a client's authentication. It takes no parameters.

Perform an XPath select

The *performXPath* method allows a client to perform an XPath select on a given XML string. The method was added for the benefit of the AR System that has no XML parsing abilities.

1. xml: XML string.
2. expression: XPath expression that should return one value.

The method returns the value returned by the expression.

Message sending framework

Overview

When passing data between two services, an element of ‘store and forward’ must exist in each direction. The primary reason for this functionality is to allow a message to be stored in the event of the remote service being unavailable. Given both sides of the integration have to implement this functionality, it is best practice for each party to handle the store and forward requirement as part of the message sending functionality.

The XML responses from the third party also need to be decoded by the gateway, and given every third party response will be different, a configurable system is required.

These two problems – store & forward, and third party message handling – are addressed using the XML Gateway message sending framework.

The framework is written using Spring configuration files, and includes a variety of interfaces, allowing developers to build integrations around the framework.

Third party to AR System

The XML Gateway will provide an XML response when processing a message passed to it by the third party, and this response can be used to determine whether the remote data source (such as AR System or a database) is currently unavailable. By using this information, the third party can make a decision to whether it wishes to store the message and resend it at a later date. It is up to the third party to decide a resend policy.

AR System to third party

The XML Gateway provides store and forward functionality to workflow through a number of simple webservice calls to the gateway. While there are a number of webservice calls that allow messages to be sent, many send XML and return the third party response to the client. AR System has no XML parsing capabilities, and hence an XML response is of no use to AR System workflow. Therefore, a system is provided that examines responses, provides store and forward functionality and can update the AR System once the transfer of a message has been completed.

Typically, the framework would be invoked through the XMLGateway sendMessage webservice method (described below).

Key components of the message sending functionality

There are four key components of the message sending architecture, all of which are defined in the Spring message-sending.xml configuration file: destinations, message handlers, error handlers and triggers.

Destinations

A destination defines a way in which a message will be sent. There are a number of destinations provided with the gateway and more can easily be provided by JSS or through implementing the Destination interface – api documentation is linked from the XML Gateway web interface.

The following is a list of destinations provided with the gateway.

HTTPDestination

This destination will perform an HTTP post to a given URL passing the XML with a parameter or as a raw post. The raw post facility allows you to create raw SOAP messages and post to a third party.

Example Spring configuration:

```
<bean id="ms.http.destination.example"
      class="com.javasystemsolutions.xml.gateway.messagesending.HTTP
Destination">
  <property name="url" value="http://localhost:8181" />
  <!-- If not provided then the XML is posted directly to the
destination without an HTTP parameter -->
  <!--property name="parameter" value="xml" /-->
</bean>
```

You can also set the connection timeout using the timeout property and passing a value in milliseconds. The default is 60s.

SampleDestination

This destination can be used as a part of a test rig. It takes one parameter, success, and this may be true or false, and the destination will either succeed or fail given the value of success.

Example Spring configuration:

```
<bean id="ms.test.destination.sample.success"
      class="com.javasystemsolutions.xml.gateway.messagesending.Samp
leDestination">
  <property name="success" value="true" />
</bean>
```

WebserviceDestination

This destination calls a method (by loading a WSDL) that is assumed to take a single string parameter and passes the XML to it. This is a simple destination in the event you want to deliver XML to a third party. You

could build a similar destination to perform a more complex integration (we are happy to provide source code).

```
<bean id="ms.test.destination.sample.success"
      class="com.javasystemsolutions.xml.gateway.messagesending.Webs
erviceDestination">
  <property name="wsdl" value="http://remote/service?wsdl" />
  <property name=”method" value="postxml" />
  <!-- These are optional -->
  <property name="username" value="myuser" />
  <property name="password" value="mypassword" />
</bean>
```

Message handlers

A response handler sends a message to a destination and deals with the response. While sending a message is the only method on the MessageHandler interface, the implementation would typically do something with the response. You can write your own implementations of a MessageHandler if you wish to customise the behaviour.

The following handlers are provided with the gateway:

DefaultMessageHandler

The DefaultMessageHandler sends a message and pushes the response through the gateway if a create/modify template has been defined. A sample syntax is defined as follows:

```
<bean id="ms.test.messageHandler.sample.success"
  class="com.javasystemsolutions.xml.gateway.messagesending.DefaultM
essageHandler">
  <property name="destination"
    ref="ms.test.destination.sample.success" />
  <!-- Optionally send XML response to create/modify -->
  <property name="createModifyTemplate"
    value="messagesending-update-from-test" />
  <property name="createModifyServer" value="myarserver" />
</bean>
```

ARSMessageHandler

This handler extends the DefaultMessageHandler to provide the following extra properties:

- xmlWriteBackField: The ID of a field that exists on the form used to generate the XML that is passing through the handler. Before the XML is sent to the destination, the XML is written to the entries used to generate the XML by updating this field.

Error handlers

The error handler provides a mechanism to invoke functionality on the entries returned from the original query should the message sending fail. Typically, this would be used to mark an entry as 'unsent', by setting a field (such as status) to a particular value.

An example error handler is defined as follows:

```
<bean id="ms.test.errorhandler"
class="com.javasystemsolutions.xml.gateway.messagesending.ARSErrorHa
ndler">
  <property name="server" value="myarserver" />
  <property name="form" value="User" />
  <property name="field" value="536780912" />
  <property name="value" value="SEND_FAILED" />
</bean>
```

The ErrorHandler interface allows you to build your own implementations if the default functionality does not surfice. The interface is defined in the api documentation linked from the XML Gateway web interface.

Triggers

Triggers are defined exactly the same way as they are when using the scheduler, so please refer to that part of the manual.

Methods of operation

The message sending framework operates synchronously and asynchronously to the gateway call. There are a number reasons for providing asynchronous functionality:

- Running synchronously means the webservice call could take a long time to complete if the third party is running slowly.
- If the third party is unavailable, a system of 'store and forward' is required to send a message at a later time. This can only practically happen in an asynchronous fashion.

These two methods of operation allows store and forward functionality to be implemented in and out of an AR System server thread. In, by attempting to send a message synchronously using the sendMessage webservice function (which returns true on success), or out, by not setting the synchronous flag and letting the gateway run the functionality within its own thread.

Invoking through a webservice

The sendMessage webservice provides a facility to send an XML message to a third party and decode an XML response. The workflow call takes a

number of well documented parameters, plus additional parameters that are specific to message sending:

The method requires the following paramerters:

1. Query request, used to generate an XML message.
2. Message hander, used to transport the message to a destination and deal with the XML response.
3. Error handler, used to deal with an error if one should occur.
4. Trigger, used when a message is sent asynchronously and added to the Quartz scheduler. By not passing the name of a schedule to the webservice call, no store and forward will be attempted.
5. A boolean flag (attemptSyncSendFirst) stating whether a synchronous send should be attempted first. This would take place within the AR System server thread.

There are three possible outcomes from the sendMessage method:

1. If attemptSyncSendFirst was set to true and the message was delivered, return true.
2. If attemptSyncSendFirst was set to true, the message was not delivered and a trigger was not provided, the gateway will return false and invoke the error handler (if provided).
3. If a trigger was provided and either attemptSyncSendFirst is false or it is true and the message was not delivered, and a trigger was defined, the gateway will schedule the message and attempt to deliver it at a later time.

In this case, the webservice call will return true.

If the scheduler is invoked as per (3) and the message is never delivered, the error handler will be invoked if provided to the sendMessage method. Typically, the error handler would be used to set a field on the entries returned from the original query in order to indicate that they were not sent.

Using the scheduler

The gateway makes use of a scheduler called Quartz. This is configured through the Spring framework in the file scheduler-context.xml. The gateway comes with jobs that can run through the scheduler in order to assist with integrating with a third party system. An approach where by the gateway queries AR System records and then pushes them out to a third party is a good way of reducing the load on AR System and not interrupting the workflow to perform the tasks of querying data and pushing it to a third party system.

Spring configuration file

The scheduler context file is broken down into the following components.

Core scheduler

The scheduler configuration starts with defining the scheduler itself and a set of jobs to run. The jobs are linked via triggers, and each job requires one trigger. Each trigger is referred to through the *<ref bean=”..” />* element.

```
<!-- Create scheduler -->
<bean id="global.schedulerFactoryBean"
      class="org.springframework.scheduling.quartz.SchedulerFactoryB
ean">
  <property name="quartzProperties">
    <map>
      <entry key="org.quartz.threadPool.threadCount" value="10" />
    </map>
  </property>

  <!-- Tasks to kick off -->
  <property name="triggers">
    <list>
      <ref bean="scheduler.triggers.queryAndPost.example" />
    </list>
  </property>
</bean>
```

Triggers

The trigger defines when the job will be executed – note that the bean id (scheduler.triggers.queryAndPost.example) is referred to in the list of triggers above.

The example below uses a trigger based on the Unix cron system – it is documented here:

http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/scheduling/support/CronSequenceGenerator.html

The jobDetail property references the job.

```
<bean id="scheduler.triggers.queryAndPost.example"
class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail"
ref="scheduler.jobs.queryAndPost.example" />
    <property name="cronExpression" value="0 20 1 * * ?" />
  </bean>
```

Jobs

The job defines the activity of work that will be performed by the trigger – note that the bean id (scheduler.jobs.queryAndPost.example) is referenced in the trigger above.

The following example job performs a query and post, which queries the AR System and pushes the results back through the gateway into the same (or another) AR System server. This is a great way to easily transfer data from one AR System server to another, by defining a query template that produces output that can be processed by a create/modify template.

The query request and post template/server are clearly identifiable, and the process also sets the status of all entries queried to 1 if the post is successful.

```
<bean id="scheduler.jobs.queryAndPost.example"
class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="applicationContextJobDataKey"
value="applicationContext" />
  <property name="jobClass"
value="com.javasystemsolutions.xml.gateway.jobs.QueryAndPost" />
  <property name="jobDataAsMap">
    <map>
      <entry>
      <key><value>query</value></key>
      <value><![CDATA[
        <query>
          <template>sampleQuery</template>
          <configKey>userFormSearch</configKey>
          <searches><search>
            <server>myarserver</server>
            <form>User</form>
            <query></query>
          </search></searches>
        </query>
```

```
      ]]></value>
      </entry>
      <entry key="postTemplate" value="newUser" />
      <entry key="postServer" value="myarserver" />
      <entry key="statusAfterQuery" value="1" />
    </map>
  </property>
</bean>
```

Instead of changing the entry status after successfully querying and actioning the results, a job can be configured to delete the entries queried by setting:

```
<entry key="deleteAfterQuery" value="true" />
```

The jobs ships with a number of jobs and if you require something different then we can build them for you (please contact JSS for more information).

Query and post to gateway

This job runs a query and pushes the results through a create/modify template. This is documented above and is a great way to transfer data between AR System servers.

Query and post to a Java Messaging Service

This job runs a query and pushes the results into a Java Messaging Service destination (i.e. a queue or topic). A (reduced for clarity) example follows:

```
<bean id="scheduler.jobs.queryAndSendToJMS.example"
class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="applicationContextJobDataKey"
value="applicationContext" />
  <property name="jobClass"
value="com.javasystemsolutions.xml.gateway.jobs.QueryAndPost" />
  <property name="jobDataAsMap">
    <map>
      <entry>
      <key><value>query</value></key>
      <value><![CDATA[
        <query>
         ...
        </query>
      ]]></value>
      </entry>
      <entry key="target" valueref="jms.core.template.target" />
      <entry key="statusAfterQuery" value="7" />
    </map>
```

```
  </property>
</bean>
```

The query is very similar to the query and post job previously detailed, however it is given a reference (note the attribute valueref) to a JMS template defined in the jobs-core Spring configuration file. To make this work, you would need to configure the JMS service in the jobs-core file, and if you do not have experience in this area and no access to a Java developer with some Spring/JMS experience, please contact JSS for assistance.

Query and send to destination (such as HTTP)

This job runs a query and sends the results to one of the destinations defined in the message sending framework. This functionality could be used to regularly deliver a SOAP message to a third party.

An example is detailed below (and reduced for clarity):

```
  <!-- Define a job detail bean and trigger for performing a query
and sending to an HTTP destination -->
  <bean id="scheduler.jobs.queryAndSendToDestination.destination"
        class="com.javasystemsolutions.xml.gateway.messagesending.HT
TPDestination">
    <property name="url" value="http://localhost:8181/target" />
  </bean>
  <bean id="scheduler.jobs.queryAndSendToDestination.example"
class="org.springframework.scheduling.quartz.JobDetailBean">
    <property name="applicationContextJobDataKey"
value="applicationContext" />
    <property name="jobClass"
value="com.javasystemsolutions.xml.gateway.jobs.QueryAndSendToDestin
ation" />
    <property name="jobDataAsMap">
      <map>
        <entry>
          <key><value>query</value></key>
          <value><![CDATA[
            <query>
              ...
            </query>
          ]]></value>
        </entry>
        <entry key="destination"
valueref="scheduler.jobs.queryAndSendToDestination.destination" />
      </map>
    </property>
  </bean>
```

Developing plugins

No integration product will do everything out of the box, and the XML Gateway has a number of Java interfaces that can be used to develop bespoke plugins to provide customised behaviour. The interfaces are published as javadocs (available through the gateway portal page) and described below.

To develop a plugin, you will need a resource with Java programming knowledge that includes webservices and web applications.

Bespoke response builders

When pushing data into the gateway (via any create/modify/delete call), an XML response is created and sent back to the client. A standard XML Gateway response builder is used to generate the documented responses, however bespoke response builders can be written by implementing the ResponseBuilder interface.

Implementations of this interface can be enabled on a system wide and template level. Given that consistent responses are required for gateway calls, this would usually be implemented on a system wide level because not all gateway calls will result in a template being loaded, and hence, if no template is loaded, no configured response builder within the template can be loaded.

A system wide response builder is defined within the XML Gateway Configuration file.

Template level response builders are discussed in hte create/modify manual page.

Webservice plugins

The webservice interface can be used by the AR System developer to make use of the gateway through workflow. However, there will be many instances where required functionality does not exist within the gateway and can not be implemented through workflow. A good example of this would be a system where an XML document must be sent to a third party, and that third party will respond with another XML document. Decisions then need to be made based on the data within the XML response provided by the third party, and finally, a value can be passed back to the workflow call.

A webservice called queryAndInvokePlugin exists to perform this functionality, where a query is performed against the AR System, and the results are passed to an implementation of the Plugin interface. The Plugin interface then does something with the XML produced from querying the AR System (such as, for example, making another WS call and examining the response), before the Plugin has to return a value to the AR System.

The classname of the Plugin implementation is passed through the workflow call to queryAndInvokePlugin, and a set of parameters can also be passed to the Plugin. This approach provides a very flexible plugin interface and allows plugins to be developed by anyone with Java skills and no AR System knowledge.

Sadly, we have only been able to allow a fixed number of parameters to be passed to the webservice call because of AR System's limited webservice support (it seems, on version 7.0, unable to consume a webservice and populate an array). We have therefore limited the number of parameters to ten however this can be increased by JSS if required.

Message sending destinations

The message sending framework provides the ability for implementations of the Destination interface to be passed into a target. This allows a third party to provide their own message sending destination given there will always be the requirement for different message sending techniques. For example, one may wish to create a set of Java stubs from a WSDL, populate these stubs with data from a given XML Document and send the stubs to a remote webservice.

Filters

Filters are used to convert text into XML before it is passed into the create/modify interface. Bespoke filters can be written by implementing the Filter interface. A filter simply translates an input stream into an XML Document.

Pre-production optimisation and system testing

Please read this section carefully: Your support contract relies on evidence that this section has been reviewed and actioned.

It is important to ensure you load test the integration built with the XML Gateway. This is true for any integration product and service being deployed to a corporate network. There are no 'magic settings' to ensure the XML Gateway (or any other message handling) integration performs optimally and there are configuration/cache settings to consider in both the gateway and the JVM.

AR System form caching

The gateway deliberately avoids exposing many cache settings in order to reduce the configuration complexity. However, the AR System API is not quick at retrieving AR System form information (such as the fields and the associated data, ie maximum field length, enumerated string values, etc) so a cache is maintained within the gateway. The number of forms held in the cache is configured in the xmlgateway.xml file – see the *cachedFormFieldSets* setting.

If you review the standard out log files with DEBUG logging enabled, you will be able to see whether a form is being loaded from the cache or AR System.

Action point: Consider your use of the gateway and how many forms are referenced within the templates. Set this value to the number of forms reference, but remember that the higher the figure, the greater memory footprint required by the gateway.

Session handling

When dealing with user sessions, ie when a user authenticates with the gateway, it is important to ensure the unauthenticate method is called because this closes associated connections to the AR System server (ie ARServerUser.logout()).

If building a web based application, do not assume users will click a logout link – add a listener to the browser close window event to call unauthenticate, ie.

```
window.onbeforeunload= function() {
  // call unauthenticate
}
```

It is also important to consider the session timeout (defined in the web.xml file). A low session timeout (ie 5 minutes) will ensure connections associated with a session are closed, and the session disposed, regardless of whether the unauthenticate method is called by the client. Note, when

a session is closed through timeout, it is equivalent to calling unauthenticate.

Memory footprint

This figure is derived from the answer to the number of forms that require caching. Again, there is no magic answer because if you are caching 100 small forms, the amount of memory required will be lower than 20 of the CMDB forms that are known to have many hundreds of fields.

Once you've established the number of forms to cache, write a set of unit tests that interact with the gateway and perform many transactions per hour. This test should simulate real life use of the gateway. You can monitor the gateway's memory usage by installing jconsole, a tool shipped with the JDK. We have produced a video walkthrough on installing this tool:

http://www.javasystemsolutions.com/jss/video/view/Mid Tier-JMX

This tool can be used to generate very helpful screenshots for presenting to JSS when reporting a production issue. It is possible to run a production system with the JMX port open so jconsole can be used to monitor a production service.

Increasing the heap memory

The JVM heap memory is set by passing flags to the JVM. This is very well documented online. The flags are -Xms (initial setting) and -Xmx (maximum setting) – the values passed are in megabytes.

It is good practice to set the initial heap size to 90% of the maximum, so to set a typical maximum of one gigabyte, the settings are as follows:

```
-Xmx920m -Xmx1024m
```

There is a very good reason to set the initial to 90% of the maximum, and this is optimise the performance of the JVM's Garbage Collector. The GC locks the JVM threads while memory is freed, so allowing 10% “headroom” provides a much smaller amount of memory to free than, say, 50% of the heap size.

When using Tomcat and other servlet engines, you wil often find the configuration tool allows a user friendly way to set the initial and maximum heap size. For example, it's in the Tomcat system tray configuration tool when installed on Windows.

Increasing the PermGen size

One of the less well known settings is the amount of memory reserved for PermGen (an area used for class definitions and other non-working data). This has a default value of 64Mb which is inadequate for large applications

loading many Java APIs. It is recommended that this is set to 128Mb for all installations. To do this, set the JVM flag as follows:

```
-XX:MaxPermSize=128m
```

Searching the Internet on this topic will reveal many other flags and discussions you can set to avoid this problem.

Reviewing log files for performance issues

The standard out log files provide a wealth of information for reviewing the performance. During your load testing, if the gateway fails, you are likely to find the following errors, often at the end of the log files. We encourage you to read the log files and search for phrases below.

OutOfMemoryError

This means you've not got enough heap size allocated.

OutOfMemoryError: PermGen space

This means you've not got enough PermGen space allocated.

Logging in production

When running the gateway in production, set the log level to INFO (in xmlgateway.xml) unless you are debugging an issue. The DEBUG or TRACE settings not only write huge log files, but affect system performance because writing to disc wastes valuable milliseconds that could be spent processing transactions.

Performance logging

When running the gateway with the log level set to TRACE or DEBUG, all create/update/delete/query and send message (through the message sending framework) operations are timed and logged, along with the free memory at that point in time.

If an operation takes longer than 5 seconds, a log entry at level WARN is written that includes the time taken to perform the transaction.