

Cúram 8.2.2

XML Infrastructure Guide

Note

Before using this information and the product it supports, read the information in [Notices on page 53](#)

Edition

This edition applies to Cúram 8.2.2.

© Merative US L.P. 2012, 2026

Merative and the Merative Logo are trademarks of Merative US L.P. in the United States and other countries.

Contents

| | |
|--|------------|
| Note..... | iii |
| Edition..... | v |
| 1 Developing with the Cúram XML Infrastructure..... | 9 |
| 1.1 Representing Cúram data as XML..... | 9 |
| 1.2 Developing for XML..... | 9 |
| The XMLDocument class..... | 9 |
| The XML Print Stream..... | 14 |
| Sample Usage..... | 19 |
| Load balancing and failover..... | 24 |
| 1.3 The XML server..... | 24 |
| XML server architecture..... | 25 |
| Configuring the XML server..... | 25 |
| Starting the XML server..... | 44 |
| Shutting down the XML server..... | 45 |
| Statistics..... | 45 |
| 1.4 XML and XSL templates..... | 46 |
| Cúram DTD..... | 47 |
| Examples..... | 48 |
| Job types and template types..... | 49 |
| XSL template example..... | 51 |
| Generating templates from RTF documents..... | 52 |
| Globalization considerations..... | 52 |
| Notices..... | 53 |
| Privacy policy..... | 54 |
| Trademarks..... | 54 |

1 Developing with the Cúram XML Infrastructure

Learn how to develop applications that use XML with the Cúram Server Development Environment. The XML Server can be used to convert XML data into various formatted document types and then manipulate these documents for printing, e-mailing, and so on.

Related concepts

1.1 Representing Cúram data as XML

In Cúram, a specific set of tags represents data that is generated from the struct classes of an application at run time. These tags are contained in a supplied Document Type Definition.

The Cúram XML infrastructure is based on the Apache XML Project's suite of Java® XML libraries. The XML parser is Apache Xerces. Apache Xalan is used for XSL processing and Apache Formatting Objects Processor (FOP) is used for PDF rendering. The Apache RTFLib (JFOR) library is used for rendering documents in RTF format.

All XML is from struct classes that are defined in the application model. The Cúram XML definition uses tags to generically identify the parts of these model entities. So, the XML includes tags for structs, fields, values, types, lists, and so on. These tags are described in an Cúram specific *Document Type Definition* (DTD).

Developers require knowledge of the XML format for XSL template development, although there can be some cases in which developers might want to manipulate the XML directly.

Related reference

[Cúram DTD on page 47](#)

The following markup declaration is the DTD for Cúram XML. The DTD can be found in the `/lib` directory of the SDEJ. The comments within the declaration describe each element.

Related information

[Apache Software Foundation](#)

1.2 Developing for XML

The two most important classes you need when adding XML functionality to your applications are `curam.util.xml.impl.XMLDocument` and `curam.util.xml.impl.XMLPrintStream`. The classes can be used together to generate XML and print documents.

The XMLDocument class

Cúram XML data is generated according to the rules of a simple DTD. The `XMLDocument` class is used to hold the generated XML and wraps the data in the necessary root element. The `XMLDocument` class allows well-formed XML documents to be generated using struct classes or

lists of struct classes. Instances of `XMLDocument` can be created, saved, loaded, and written to arbitrary output streams.

This class is central to all XML operations that you can perform in Cúram. Its interface can be found in the `curam.util.xml.impl` package within the supplied SDEJ JavaDoc. In the rest of this section, you will learn how to use this interface to create XML documents from your application data.

The use of the `XMLDocument` class follows the following broad pattern:

1. Create a new instance of the `XMLDocument` class.
2. Open the XML document to create the root element and provide a context for the XML data that you want to create.
3. Add a struct class (or struct classes) to the open XML document to create the XML data.
4. Close the XML document to complete the root element.

XML documents

A number of operations can be performed on XML documents.

- A document can be created and stored in memory. This document can then be stored in the database, or written to a stream, or both.
- A document can be created and written to a stream directly to reduce storage requirements. This is particularly useful for very large documents that do not require an archived copy.
- A previously archived document can be retrieved from the database and written to a stream.

As streams are flexible, there are many things you can do with them.

- You can use a stream to save the XML data to a file.
- You can use the `XMLPrintStream` class to request that a document should be printed.
- You can use a stream to transfer information over a network via a secure socket connection.
- You can use a `java.io.BufferedOutputStream` to buffer all the XML data.
- You can create your own stream classes (or use any of the standard stream classes) to do just about anything you want with the XML data!

Encoding

You must ensure that the character encoding scheme used for your data is specified for the XML document.

All XML data are represented in plain-text. A small number of characters have a particular meaning to XML (“<”, “>”, “””, “'”, “&”) and if these occur in your data they are automatically converted to their corresponding XML character entities to avoid problems. However, if you use characters outside the normal US-ASCII range (characters 0-127), even plain-text becomes ambiguous. For example, in Western Europe, you might typically store your data using the ISO-8859-1 character set also known as “Latin 1”. In this character set, the character “ë” (e-umlaut) is character number 235. However if you sent this XML data to a person in Greece who would typically use the ISO-8859-7 (Greek) character set, the same character 235 would appear as the lower-case Greek letter lambda.

To avoid this problem, XML allows the character encoding used for a document to be stated in the XML processing instruction found at the top of all XML documents. Now, when you create your document you can explicitly state that you want to use ISO-8859-1 for your data because

that is the form in which it is stored in your database. When you send the file to Greece, the person there knows not to use the ISO-8859-7 character set to interpret the data but ISO-8859-1 instead. In general, this will be handled by their XML parsing software which will read the encoding information from the document.

By default, XML uses an encoding scheme known as UTF-8. This modified Unicode scheme creates a document that uses two bytes to represent characters greater than 127. However, you will need to set the encoding explicitly if the data stored in your database uses a different encoding scheme.

Cúram XML provides a range of constants for the common encoding schemes. The available schemes are shown in [Encoding on page 10](#) below.

Table 1: XML Character Encoding Constants

| Constant | Alternative Constant | Encoding Scheme |
|---------------------|----------------------|-----------------|
| kEncodeUTF8 | | UTF-8 |
| kEncodeISO10646UCS2 | | ISO-10646-UCS-2 |
| kEncodeISO10646UCS4 | | ISO-10646-UCS-4 |
| kEncodeISO88591 | kEncodeISOLATIN1 | ISO-8859-1 |
| kEncodeISO88592 | kEncodeISOLATIN2 | ISO-8859-2 |
| kEncodeISO88593 | kEncodeISOLATIN3 | ISO-8859-3 |
| kEncodeISO88594 | kEncodeISOLATIN4 | ISO-8859-4 |
| kEncodeISO88595 | kEncodeISOCYRILLIC | ISO-8859-5 |
| kEncodeISO88596 | kEncodeISOARABIC | ISO-8859-6 |
| kEncodeISO88597 | kEncodeISOGREEK | ISO-8859-7 |
| kEncodeISO88598 | kEncodeISOHEBREW | ISO-8859-8 |
| kEncodeISO88599 | kEncodeISOLATIN5 | ISO-8859-9 |
| kEncodeISO885910 | kEncodeISOLATIN6 | ISO-8859-10 |
| kEncodeISO885913 | kEncodeISOLATIN7 | ISO-8859-13 |
| kEncodeISO885914 | kEncodeISOLATIN8 | ISO-8859-14 |
| kEncodeISO885915 | kEncodeISOLATIN9 | ISO-8859-15 |
| kEncodeISO2022JP | | ISO-2022-JP |
| kEncodeSHIFTJIS | | Shift_JIS |
| kEncodeEUCJP | | EUC-JP |

The relevant constant should be specified when constructing a new `XMLDocument` in order to set the encoding scheme as appropriate for the XML document. This encoding will be used for

the XML document declaration as well as for the XML document itself. If loading an XML document from the database, the encoding of that document should match the encoding used to construct the `XMLDocument` class. If you supply no value, no encoding scheme will be specified in the XML and XML parsers will thus assume UTF-8 according to the XML standard. If the encoding scheme you wish to use is not among those listed, you may supply a string containing the encoding value you wish to use.

All of the encoding constants are within the `XMLEncodingConstants` interface. To use, for example, the Latin 1 set, you would use `XMLEncodingConstants.kEncodeISOLATIN1` or `XMLEncodingConstants.kEncodeISO88591`.

Creating an XMLDocument

As XML data is created, it is written to a stream. By default, an instance of the `XMLDocument` class maintains an internal stream that holds the XML data.

By allowing the document to store the data in this stream, you can then save the document to the database or write it to another stream, if you require. If you do not want to save the document, you can specify an alternative stream where the XML data can be written as it is created. This can help to reduce memory overhead if the data stream is large. For example, data for a large report may not need to be stored in the database. This data can be generated and processed on-the-fly without any intermediate storage. The following code is the `XMLDocument` Constructor:

```
XMLDocument(String encoding);
XMLDocument(OutputStream stream, String encoding);
```

Both constructors take a parameter to set the character encoding. You can set the encoding value using one of the encoding constants or an encoding string of your own choosing.

The first constructor is used when you want the XML document to use its internal string buffer to store the XML data. This allows you to save the document to the database later or to write to another stream once it is complete. If you intend to load an XML document from the database, you should also use this constructor. In that event, the encoding string is irrelevant.

The second constructor allows you to specify an output stream that the document should be written to as it is created. This precludes the possibility of storing the document in the database once it is complete. However, for large documents that do not need to be stored but rather printed, saved to a file, or transferred over a network, this is a more efficient method than the first. For streams such as file and print streams that are required to be explicitly opened, it is important that the stream passed to this constructor is already open as the document will expect to be able to write to it immediately.

Opening an XMLDocument object

Once you have instantiated an `XMLDocument` object, you need to open it in one of two ways.

```
open(String generatedBy, String generatedDate, String version,
     String comment);
openForList(String generatedBy, String generatedDate,
            String version, String comment);
```

If you want to write the details of a single struct class to the XML document, you must open the document with the `open()` method. If you want to write the details of several different struct classes of the same type to the document, you must open the document with the `openForList()`

method. This latter method allows you to create a document that contains a list of struct classes where each one is added in turn. All the struct classes must be of the same type. The former method allows you to add only a single struct class to the document before closing it. This single struct class can, however, contain fields that are lists of struct classes.

Both of the open methods take several parameters that can be used to set meta-data for the document. You can include the name of the entity that generated the document, the date and time on which it was generated, the version of the document, and any other comments you wish to associate with the document. Each parameter is a string and you can use any length of data formatted in any way you wish. You must, however, respect the requirement of XML that certain characters be converted to character entities. If your strings contain any of the following characters: “”, “'”, “<”, “>”, or “&”, you must convert them to their character entity values. This can be done by calling the `XMLDocument.escape()` method. The method takes a string parameter and returns a new string with the character entity conversions done for you.

Once opened, you can begin adding struct classes to your XML document.

Adding data to an XMLDocument object

The `add()` method of the `XMLDocument` class can be used to produce XML data from an instance of a struct class.

```
add(Object value);
addFromXML(String xmlFragment);
```

For documents opened with the `open()` method, you may only issue a single call to `add()` before closing your document. For documents opened with `openForList()`, you may use several calls to `add()` but should ensure that you only add instances of the same struct class type.

`addFromXML()` is a convenience method allowing an XML fragment to be directly added to the document, rather than using the struct class. It is the responsibility of the caller to ensure this fragment respects the DTD.

Closing an XMLDocument object

Once you have finished adding data to an XML document, you need to close it.

```
close();
```

The `close` method of the `XMLDocument` class takes no parameters. Calling the `close` method will not close the output stream you specified as a parameter to the XML document. You must close this stream separately.

Once closed, a document will write all remaining XML information to the stream to complete a well formed XML document. If the document object is using an internal string stream buffer, you may save the document to the database or write it to another stream.

Saving an XMLDocument object

Once closed, any XML document you created to write to the default internal string stream buffer can be saved to the database.

```
save(String name, XSLTemplateInstanceKey templateKey);
```

Saving to the database is useful if you want to print information yet keep a record of what was printed. As information in the database may change, it will not always be possible to simply print out the same form, letter, etc., and expect it to contain the same data as before. Using the XML document archive, however, you are guaranteed that the data will be identical as it represents a snapshot of the values at a particular point in time.

Each document can be saved along with the details of an associated template. This allows any print job, for example, to be rerun in the future with the same data and the same version of the template. The `save` method takes two input parameters and has one return value. The input parameters allow you to specify a name for this saved document. This can be any string-type information that you want. The maximum length is 100 characters. The second parameter is the template instance (version of a template) that you want to associate with this document.

The return value is the key value of the new archived document record that will be created to hold the XML data. This key value can be stored elsewhere to keep track of what documents are available. For example, if you print a letter to send to a client, you could associate this key with a diary entry recording the sending of the letter. The letter could then be reprinted at any time in the future by accessing the key stored with the diary entry.

Loading an XMLDocument object

To load an XML document from the document archive, you should first create a default `XMLDocument` object.

```
load(XMLArchiveDocumentID key);
```

The `load` method takes one parameter which is the key to the archive document. The details returned include the template information that you saved with the document such as its version and locale, and the XML representation of the data in the document.

Once loaded, the XML document object can be treated like any other document object that was created, opened, had data added and was closed.

The XML Print Stream

The `XMLPrintStream` class is a type of output stream that allows jobs to be submitted to the XML Server for processing. Used in combination with the `XMLDocument` class and XSL templates, it allows XML data to be formatted and printed. The `XMLPrintStream` can be configured on a per-server or per-job basis for maximum flexibility. The `XMLPrintStream` class includes features for previewing documents that are generated by the server.

For developers, the interface to the XML server that is included in the SDEJ is the `XMLPrintStream` class. Using this class, you can to send print job requests to the Cúram XML Server.

Related concepts

[The XML server on page 24](#)

The product XML server is a Java application that processes jobs submitted by a client to produce a formatted document. Each job requires an XML document generated by a Cúram server application and an XSL template. XSL template is applied to the XML to render it to PDF, RTF, HTML, or plain text.

The *XMLPrintStream* class

The public interface to the `XMLPrintStream` class can be found in the `curam.util.xml.impl` package within the SDEJ JavaDoc.

In use the following basic pattern will be followed:

1. Create a new instance of the `XMLPrintStream` class.
2. Set the various printing options.
3. Open the secure connection to the XML server.
4. Write to the print stream object. (This is done by an `XMLDocument` object).
5. Close the print stream object to initiate the print job.

The following subsections will look at these steps in detail, but first there are steps you can take to configure default values for your print streams.

Default configuration for *XMLPrintStream*

When you submit a print job, you can use the `XMLPrintStream` class to set a number of options. The options include the printer name, the paper tray name, the server host name, and the server port number.

Each of the options can be set in your project's properties. The values that are required are shown in the following table. All values are entered as strings and are not converted to any other data type. You must convert any special characters with a meaning in XML to character entities.

Table 2: The application prx settings for *XMLPrintStream*

| Variable Name | Description |
|---|---|
| <code>curam.xmlserver.printer</code> | The name of the default printer to use for jobs that are submitted by this application. On Microsoft® Windows, this property might be, for example, <code>\\myhost\printer1</code> , or <code>lpt1:</code> . |
| <code>curam.xmlserver.tray</code> | The name of the paper tray to use for jobs that are submitted by this application. |
| <code>curam.xmlserver.host</code> | The host on which the XML Print Server resides. To use multiple XML Servers, you can specify this property as a '/' separated list of host names. |
| <code>curam.xmlserver.port</code> | The port on which the XML Print Server is listening. To use multiple XML Servers, you can specify this property as a '/' separated list of host names. |
| <code>curam.xmlserver.fileencoding</code> | The default encoding that is used for the encoding of files that are provided to the XMLServer. You can override this value for individual instances of <code>XMLPrintStream</code> by using the <code>setEncoding</code> method. The default value for this property is UTF-8. |
| <code>curam.xmlserver.serializeLocaleNeutral</code> | Specify that XML Server data is serialized in a locale-neutral way instead of being based on the locale properties on the server. |

When your application submits a print job, these values are used as the defaults for the job. You can use the individual setter methods to override these defaults.

Creating an XMLPrintStream object

An `XMLPrintStream` object can be instantiated by providing the name of the host on which the XML Server resides and the port on which the XML Server is listening. However, as documented in the Java documentation, these properties are not used and it is recommended to use the empty constructor.

```
XMLPrintStream(String host, int port)
XMLPrintStream(final XMLServerEndPoint[] endpoints)
XMLPrintStream()
```

Configuring an XMLPrintStream object

Once instantiated, an `XMLPrintStream` object can be configured

```
setPrinterName(String name);
setPaperTray(String tray);
setUserName(String user);
setEmailAddress(String email);
setEncoding(String encoding);
setJobType(String job);
```

In [Default configuration for XMLPrintStream on page 15](#) the default configuration was covered. You can override the printer name and paper tray values using the `setPrinterName` and `setPaperTray` methods respectively. In addition, you can also set a user name and an e-mail address for the print job. The user name might be that of the user who initiated the print job, or any other user name you prefer to use. The e-mail address, similarly, can be any e-mail address you want to associate with the job.

The encoding can also be set here. This encoding is used within the `XMLServer` for such purposes as printing documents in the specific encoding. If the encoding is not explicitly set through the `setEncoding` method, then the value will be taken from the `curam.xmlserver.fileencoding` configuration property. If this property is not set, then the default encoding of UTF-8 will be used.

Note: It is important to set the encoding correctly when using `XMLDocument` and `XMLPrintStream` classes together. For example, if you create an `XMLDocument` class with an encoding of UTF-8 and you create the `XMLPrintStream` class setting the encoding to be US-ASCII, there may be some issues with the document being printed. As US-ASCII contains a smaller character code set than UTF-8, some characters may not be supported and therefore when printing the document, the resulting document may contain unrecognizable characters. Therefore, if you wish to have the UTF-8 document printed correctly, you should set the encoding of the `XMLPrintStream` instance to use UTF-8 encoding. Please see [Encoding on page 10](#) for further information on encoding.

All the parameters are strings and you must respect the requirement of XML that certain characters must be replaced with character entities. You can use the `XMLDocument.escape(String value)` method for this conversion.

Overriding the default values allows you, for example, to print a document to a printer nearest the current user, rather than to a default printer.

By default, the XML Server will combine your XML data with an XSL template and attempt to render the resulting document as a PDF document. The XML is transformed based on the

template locale and for Right-to-Left languages. These are the supported languages, which are specified by locale code:

Table 3: Right-to-Left Supported Languages and Locale Codes

| Language | Locale Code |
|---------------|-------------|
| Arabic | ar |
| Farsi | fa |
| Hebrew | he |
| Hebrew | iw |
| Yiddish | ji |
| Yiddish | yi |
| Pashto/Pushto | ps |
| Urdu | ur |

Due to the limitations of FOP, you must have a supporting Right-to-Left implementation in the XML Server configuration (e.g., see [RenderX configuration on page 32](#)). For this rendering step to work, the combination of the XML data and XSL template should produce a document marked up using XSL Formatting Objects. As an alternative to PDF output, you can specify RTF, HTML or plain text output using the `setJobType()` method. This method can be used to specify any of the supported output formats using the appropriate constant as shown in [Configuring an XMLPrintStream object on page 16](#). All the constants are within the `XMLPrintStreamConstants` class and should be prefixed with `XMLPrintStreamConstants` in your code unless you have implemented this class as an interface.

Table 4: XMLPrintStream Job Types

| Job Type | Description |
|---------------------------|--|
| <code>kJobTypePDF</code> | This is the default job type. The XML data will be combined with the XSL template and the resulting document will be rendered as a PDF document. The template should be developed to produce a document marked up with XSL Formatting Objects. Temporary files will be given the extension “.pdf”. |
| <code>kJobTypeRTF</code> | The XML data will be combined with the XSL template and the resulting document will be rendered as an RTF document. The template should be developed to produce a document marked up with XSL Formatting Objects. Temporary files will be given the extension “.rtf”. |
| <code>kJobTypeHtml</code> | The XML data will be combined with the XSL template and the resulting document is assumed to be HTML. Appropriate indentation will be applied automatically. The <code><xml></code> declaration at the top of the file will be omitted. The template should be developed to produce a document marked up with HTML. Temporary files will be given the extension “.html”. |

| Job Type | Description |
|--------------|--|
| kJobTypeText | The XML data will be combined with the XSL template and the resulting document is assumed to be plain text. The <code><xml></code> declaration at the top of the file will be omitted. Temporary files will be given the extension “.txt”. |

In addition to the predefined job types it is possible to define a custom job type. If a custom job type is to be used the `setJobType()` method should be passed a string matching the new job type, where the job type is defined in the XML Server configuration file. For more information on defining and implementing custom job types consult [Custom configuration on page 32](#).

Opening an XMLPrintStream object

Opening an `XMLPrintStream` object, establishes a secure connection with the chosen XML Server, sends the job configuration information, and the XSL template. Once open, the XML data can be written to the connection. In general, you will let an `XMLDocument` object write the data to the stream. All XML documents must be accompanied with an XSL template to allow the data to be formatted.

```
open(XSLTemplateInstanceKey key);
open(String xslTemplate);
open(XSLTemplateInstanceKey key,
     String host,
     int port);
open(String xslTemplate,
     String host,
     int port);
open(XSLTemplateInstanceKey key,
     XMLServerEndPoint[] endpoints)
open(String xslTemplate,
     XMLServerEndPoint[] endpoints)
```

There are a number of `open()` methods. The main difference between these is that you can specify a key to an XSL template in the database or provide the XSL template document directly in a string. Also, you can provide the secure connection information for the XML Server (host and port) or alternatively leave these values to be picked up from the `curam.xmlserver.host` and `curam.xmlserver.port` properties.

Once opened, you should immediately begin writing data to the secure connection. A long delay causes a time-out to occur and the connection will be lost.

Closing an XMLPrintStream object

Closing an `XMLPrintStream` object causes the print job to be started. Before closing the object, a well-formed XML document must have been written to it. The `close` method takes no parameters.

```
close();
```

Print preview

You can preview your document before printing it.

```
setPreviewStream(OutputStream preview);
```

The XML Server takes an XML document and an XSL template and processes the two to produce another document which could be in PDF, RTF, HTML, or plain text format. Normally, the XML Server will run a further command to print, or otherwise process, the document. However, you can instead direct the XML Server to return the document to your application server rather than process it further. This allows you to preview the document before printing it or just store the document in the database for later retrieval.

To preview a document, you must specify a preview stream when configuring the print stream object. After the XML Server has generated the PDF it will return it to the print stream object which will in turn write it to the stream specified as a parameter to the `setPreviewStream` method. This stream could be a simple string stream buffer or a file stream, whatever is required. If no stream is specified, the XML Server will assume that a preview is not required.

Once the print stream object is closed, the preview stream will contain the document and the application server can manipulate it in any way required. For example, it could be returned to the client application and displayed in an appropriate viewer of some kind.

Note: If a preview stream has been specified, the XML Server will not print anything, nor will it create a temporary file containing the document.

Sample Usage

This section presents some samples of the way the `XMLDocument` and `XMLPrintStream` objects can be used together.

The samples included cover the following scenarios:

- Saving XML data to a file.
- Printing a simple XML document.
- Saving and loading XML documents using the archive.
- Previewing an XML print job's output.
- Building a document from a list.

Along with the code samples are suggestions of how they can be further developed and used.

All the methods are developed as methods of process stereotyped classes in the application model.

Saving XML Data to a file

This sample demonstrates how XML data can be created and written to a stream, in this case a file stream. The function assumes that a file name and an instance of a struct class are passed as parameters.

This method demonstrates how to save XML data to a file using `FileWriter`.

```
import curam.util.xml.impl.XMLDocument;
import curam.util.xml.impl.XMLEncodingConstants;
import java.io.FileWriter;

public class XMLSample {

    void saveToFile1(String fname, MyStruct myStruct) {
        FileWriter myFile = new FileWriter(fname);

        XMLDocument myDoc =
            new XMLDocument(XMLEncodingConstants.kEncodeISOLATIN1);

        myDoc.open(A User, 31-Dec-2002, 1.0, Sample 1);
        myDoc.add(myStruct);
        myDoc.close();

        myFile.write(myDoc.toString());
        myFile.close();
    }
}
```

Printing an XML document

This sample shows how the struct class used in the previous sample can be written to an `XMLPrintStream` object to print the data. It is assumed that a template instance key is supplied to the function and that the default configuration values will be used.

```
import curam.util.xml.impl.XMLDocument;
import curam.util.xml.impl.XMLEncodingConstants;
import curam.util.xml.impl.XMLPrintStream;
import
    curam.util.administration.struct.XSLTemplateInstanceKey;

public class XMLSample {

    void printDoc1(XSLTemplateInstanceKey tempKey,
        MyStruct myStruct) {

        XMLPrintStream myPrintStream = new XMLPrintStream();
        myPrintStream.open(tempKey, MyPC, 1234);
        myPrintStream.setEncoding(
            XMLEncodingConstants.kEncodeISOLATIN1);
        XMLDocument myDoc =
            new XMLDocument(myPrintStream.getStream(),
                XMLEncodingConstants.kEncodeISOLATIN1);

        myDoc.open("A User", "31-Dec-1999", "1.0", "Sample 1");
        myDoc.add(myStruct);
        myDoc.close();

        myPrintStream.close();
    }
}
```

Saving and loading XML documents

In this sample, two functions are presented. The first sample, which is based on the previous sample, saves a document to the archive. The second sample retrieves the document and prints it again. The direct streaming method cannot be used to create the document if it is to be saved.

```
import curam.util.administration.struct.XSLTemplateInstanceKey;
import curam.util.xml.impl.XMLDocument;
import curam.util.xml.impl.XMLEncodingConstants;
import curam.util.xml.impl.XMLPrintStream;
import curam.util.xml.struct.XMLArchiveDocumentID;
import curam.util.xml.struct.XMLArchiveDocDetails;

public class XMLSample {

    /**
     * Creates an XMLDocument and saves it to the database.
     */
    XMLArchiveDocumentID saveDoc(
        XSLTemplateInstanceKey tempKey, MyStruct myStruct) {

        XMLDocument myDoc = new XMLDocument(
            XMLEncodingConstants.kEncodeISOLATIN1);

        myDoc.open("A User", "31-Dec-1999", "1.0", "Sample 1");
        myDoc.add(myStruct);
        myDoc.close();

        // Save the document to the database.
        final XMLArchiveDocumentID docKey =
            myDoc.save("Sample Saved Document 1", tempKey);
        return docKey;
    }

    /**
     * Loads an XMLDocument from the database and prints it.
     */
    void loadDoc(XMLArchiveDocumentID docKey) {

        // First load the archived data for the document and get
        // its template details and data content.
        final XMLDocument docForLoading = new XMLDocument(
            XMLEncodingConstants.kEncodeISOLATIN1);
        final XMLArchiveDocDetails docDetails =
            docForLoading.load(docKey);

        final XSLTemplateInstanceKey tempKey =
            new XSLTemplateInstanceKey();
        tempKey.templateID = docDetails.templateID;
        tempKey.templateVersion = docDetails.templateVersion;
        tempKey.locale = docDetails.locale;

        final String xmlContent = docDetails.document;

        docForLoading.close();

        // Now use this information to reconstruct a new
        // XMLDocument and print it.
        final XMLPrintStream myPrintStream =
            new XMLPrintStream();
        myPrintStream.open(tempKey, MyPC, 1234);
        myPrintStream.setEncoding(
            XMLEncodingConstants.kEncodeISOLATIN1);
        XMLDocument docForPrinting = new XMLDocument(
            myPrintStream.getStream(),
            XMLEncodingConstants.kEncodeISOLATIN1);
        docForPrinting.addFromXML(xmlContent);
        myPrintStream.close();
    }
}
```

Previewing an XML print job

This sample demonstrates how you can process an XML print job and receive a preview of the data that would have been printed for that XML document and XSL template.

```
import curam.util.administration.struct.XSLTemplateInstanceKey;
import curam.util.exception.AppException;
import curam.util.exception.DatabaseException;
import curam.util.exception.InformationalException;
import curam.util.internal.xml.impl.XMLPrintStreamConstants;
import curam.util.type.Blob;
import curam.util.xml.impl.XMLDocument;
import curam.util.xml.impl.XMLEncodingConstants;
import curam.util.xml.impl.XMLPrintStream;
import java.io.ByteArrayOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class XMLServerTest {

    MyResult previewJob(
        final XSLTemplateInstanceKey tempKey,
        final MyStruct myStruct)
        throws DatabaseException, AppException,
        InformationalException, IOException {

        final XMLPrintStream myPrintStream =
            new XMLPrintStream();
        final ByteArrayOutputStream previewBuffer =
            new ByteArrayOutputStream();
        myPrintStream.setPreviewStream(previewBuffer);

        // Explicitly specify that a PDF document be created:
        myPrintStream.setJobType(
            XMLPrintStreamConstants.kJobTypePDF);

        myPrintStream.open(tempKey, MyPC, 1234);
        final XMLDocument myDoc =
            new XMLDocument(
                myPrintStream.getStream(),
                XMLEncodingConstants.kEncodeISOLATIN1);
        myDoc.open("A User", "31-Dec-1999", "1.0", "Sample 1");
        myDoc.add(myStruct);
        myDoc.close();
        myPrintStream.close();

        // Now that we have created the PDF document the
        // following code illustrates three things that
        // can be done with it.

        // (1) Save the document to disk.
        final FileOutputStream previewFile =
            new FileOutputStream("/preview.pdf");
        previewBuffer.writeTo(previewFile);
        previewFile.close();

        // This class contains both a String and
        // a Blob for demonstration purposes.
        final MyResult result = new MyResult();

        // (2) Store the PDF preview in a String:
        result.previewDocString = previewBuffer.toString();

        // (3) Store the PDF document in a Blob:
        result.previewDocBlob =
            new curam.util.type.Blob(previewBuffer.toByteArray());

        return result;
    }
}
```

Having received the PDF preview of the data, this sample illustrates three ways in which the preview can be used:

1. Save it to disk.
2. Store it in a String variable.
3. Store it in a Blob. This is recommended if the document is to be stored on the database.

This example used an `java.io.ByteArrayOutputStream` as a buffer to hold the generated PDF document because this class was most suited to the three examples above. However any sub-class of `java.io.OutputStream` can be used, depending on your needs. For example, a `java.io.FileOutputStream` could be used if you wish to write the data to a file.

Building a document from a list

In the following samples, the use of list documents is demonstrated. Once an XML document built from a list has been closed, it may be manipulated in the same manner as any other XML document.

The first sample shows how a vector of struct classes can be added to an XML document.

```
import curam.util.xml.impl.XMLEncodingConstants;
import curam.util.xml.impl.XMLDocument;

public class XMLSample {
    void listDoc1(MyStructList myStructList) {
        XMLDocument myDoc =
            new XMLDocument(XMLEncodingConstants.kEncodeISOLATIN1);

        myDoc.openForList("A User",
                        "31-Dec-1999",
                        "1.0",
                        "Sample 1");

        myDoc.add(myStructList);
        myDoc.close();

        // The document may now be manipulated as before.
    }
}
```

In the second sample below, the list of struct classes is iterated over and only those elements whose *value* field is greater than 100 are added to the document. You can, of course, apply any condition you like to this basic pattern. In Cúram, the list of a type called `MyStruct` is called `MyStructList`, and the *dtls* field of the list is a `java.util.Vector` of the basic struct class type, this is assumed below.

```
import curam.util.xml.impl.XMLEncodingConstants;
import curam.util.xml.impl.XMLDocument;

public class XMLSample {
    void listDoc2(MyStructList myStructList) {
        XMLDocument myDoc = new XMLDocument(
            XMLEncodingConstants.kEncodeISOLATIN1);

        myDoc.openForList("A User",
                        "31-Dec-1999",
                        "1.0",
                        "Sample 1");

        for (int i = 0; i < myStructList.dtls.size(); i++) {
            if (myStructList.dtls.item(i).value > 100) {
                myDoc.add(myStructList.dtls.item(i));
            }
        }
        myDoc.close();

        // The document may now be manipulated as before.
    }
}
```

Load balancing and failover

XMLPrintStream supports load balancing and failover. Load balancing increases the capacity of the XML server by sharing the load among a number of replicated XML servers by making them appear as one large virtual server.

Using *XMLPrintStream* and *XMLServerEndPoint* classes to load balance

Load balancing and failover are configured in the *XMLPrintStream*, and *XMLServerEndPoint* classes. An instance of the *XMLServerEndPoint* class contains the endpoint details such as server name, port number, and a weight, which dictates the percentage of requests that are directed to this server. The `open()` method of the *XMLPrintStream* class can optionally take a list of *XMLServerEndPoints* as parameter. The secure connection is made to one of these endpoints based on the weight that is attached to it and its availability.

Using *curam.xmlserver.host* and *curam.xmlserver.port* properties to load balance

Load balancing and failover can also be configured by using the *curam.xmlserver.host* and *curam.xmlserver.port* properties. The *curam.xmlserver.host* property specifies the names of the servers that host the XML server as a '/' separated list of host names. For example:

```
curam.xmlserver.host="server1/server2/server3"
```

The *curam.xmlserver.port* property specifies the ports the XML server is running on as a '/' separated list of entries in the following format: `port[#weight]`, where the part in square brackets is optional and weight is a number in the range 0 - 1. The weight dictates the percentage of requests that are directed to the particular server and port. For example:

```
curam.xmlserver.port="1801#0.6/1802#0.2/1803#0.3"
```

There is a one-to-one mapping between the servers and ports specified. For example, server1 is running the XML server on port 1801 and server3 is running the XML server on port 1803.

1.3 The XML server

The product XML server is a Java application that processes jobs submitted by a client to produce a formatted document. Each job requires an XML document generated by a Cúram server application and an XSL template. XSL template is applied to the XML to render it to PDF, RTF, HTML, or plain text.

TheCúram server application and the XML Server can be hosted on different machines. Multiple XML servers can be run on the same host by specifying different port numbers for each server. Each server can perform a different operation, but can only perform one operation.

You can configure the server, for example, you can define default values for a printer name, printer tray, e-mail address, and user name. The XML Server was primarily designed to support printing of XML documents, however, you can also configure it to perform any required operation on the output document including printing, e-mailing or displaying by specifying a command that should be run against the document data. You can also customize the server, for example you can define implementations that overwrite the default job types, or to define new

job types. To improve performance you can use the template cache for templates that are used regularly. In addition, there are debugging features available to assist you in solving problems with templates or XML data.

(which are described in more detail in [1.4 Cúram XML and XSL templates on page 46](#))

XML server architecture

An application can read application data from a database by using `curam.util.xml.impl.XMLDocument` and `curam.util.xml.impl.XMLPrintStream` can transmit XML data to the XML server. The XML server processes the data and renders a document in any of a number of formats. This document is then submitted to the system to allow arbitrary commands to be run on the document so that it can be printed, e-mailed, transferred, or stored. in any system-specific way.

The following diagram shows how the XML server fits into the architecture of a Cúram application:

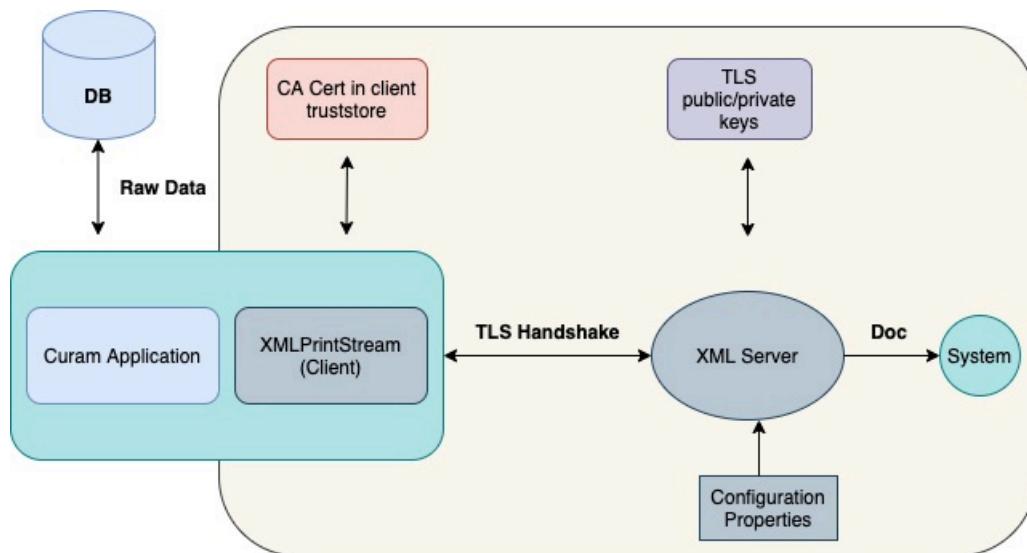


Figure 1: XML Processing Architecture

The connection from `XMLPrintStream` to the XML server is over a secure TCP/IP connection over TLS (Transport Layer Security) socket by allowing the XML server to be located remotely. For more information, see [Securing the XML server on page 34](#). The XML server is configured, at startup, to run a command on its host to process the document.

The XML server is fully threaded, allowing it to process multiple jobs simultaneously.

Configuring the XML server

The XML Server has a number of configuration options that specify how it works. All the options are set in a configuration file that is written in XML notation. This file is picked up when the XML Server is started and as such the configuration cannot be changed without stopping and starting the server.

The following operation areas of the server can be configured:

- Network
- Default values
- Server command
- Template cache
- Debugging
- Apache Log4j 2 logging
- RenderX configuration
- Custom

Configuration options

The XML Server has a number of configuration options used to specify how it should work. All the options are set in a configuration file written using XML notation.

. This file is picked up when the XML Server is started and as such the configuration cannot be changed without stopping and starting the server. There are a number of areas of the operation of the server that can be configured:

. All the configuration options are enclosed in an XML root element `<XML_SERVER_CONFIG>`. As with all XML documents, you must ensure that the characters, `<`, `>`, and `&` used in the values of your options in the configuration file are replaced with their respective character entities: `'`, `"`, `<`, `>`, and `&`.

Table 5: Configuration Options

| Option | Category | Description |
|---------------------------------------|----------------|--|
| <code><SERVER_PORT></code> | Network | The TCP/IP port number that the XML Server uses to listen for secure client connections. |
| <code><SO_TIMEOUT></code> | Network | A positive integer specifying the timeout (in milliseconds) on socket operations. If zero value is specified then it is interpreted as an infinite timeout. If this option is not specified a default value of 60000 milliseconds is used. |
| <code><DEFAULT_PRINTER></code> | Default Values | The name of the default printer. The format used should be that required by the server command. |
| <code><DEFAULT_TRAY></code> | Default Values | The name of the default printer tray. The format used should be that required by the server command. |
| <code><DEFAULT_USERNAME></code> | Default Values | The name of the default user. The format used should be that required by the server command. |
| <code><DEFAULT_EMAIL></code> | Default Values | The default e-mail address. The format used should be that required by the server command. |
| <code><SERVER_COMMAND></code> | Server Command | The command string to use to process the document. If the command string is empty, no processing will be attempted. |

| Option | Category | Description |
|-----------------------|-----------------------|--|
| <USE_PIPE> | Server Command | Indicate that the output document from the XML Server should be piped to the standard input of the server command when it is executed. One of USE_PIPE or USE_TMP_FILE is required to be true. |
| <USE_TMP_FILE> | Server Command | Indicate that the output document from the XML Server should be written to a temporary file before the server command is executed. One of USE_PIPE or USE_TMP_FILE is required to be true. |
| <USE_STDOUT_SINK> | Server Command | Start a thread to read and discard any data written to standard output by the server command. |
| <USE_STDERR_SINK> | Server Command | Start a thread to read and discard any data written to standard error by the server command. |
| <TMP_DIRECTORY> | Server Command | Specifies the directory into which temporary files containing the document data should be written. Required only if USE_TMP_FILE is true. |
| <TMP_FILE_ROOT> | Server Command | Specifies the root part of the file name to use to create the temporary file. A sequence number and the appropriate extension will be appended to create the full file name. Required only if USE_TMP_FILE is true. |
| <FOP_CONFIG_FILE> | Server Command | The name and location of a FOP configuration file. This can be used to add additional fonts for use when processing PDF files. Consult the Apache FOP documentation for more information. |
| <RENDERX_CONFIG_FILE> | RenderX Configuration | The name and location of a RenderX configuration file. This is required to initiate the RenderX rendering engine. RenderX can be used as an alternative to Apache FOP. Consult the RenderX documentation for more information. |
| <RENDERX_LOGGING> | RenderX Configuration | Specifies how RenderX's internal logging should be configured. Consult the RenderX documentation for more information. |
| <USE_TEMPLATE_CACHE> | Template Cache | Indicates that the template cache should be used to avoid having to read templates each time a job is submitted. |
| <TEMPLATE_CACHE_DIR> | Template Cache | The name of the directory in which to store the cached template files. Required only if USE_TEMPLATE_CACHE is true. |

| Option | Category | Description |
|--------------------------|-------------------|---|
| <CLEAR_TEMPLATE_CACHE> | Template Cache | When the server is started, this option will force all files in the template cache directory to be deleted. |
| <TRACE_TRAFFIC> | Debug | A debug option to echo all data received by the server to the servers standard output. |
| <STATISTICS_FOLDER> | Debug | This option will output statistics for the XML Server in the folder specified by the option. |
| <THREAD_POOL_SIZE> | Sizing | The amount of threads in the pool. |
| <THREAD_POOL_QUEUE_SIZE> | Sizing | This can be tuned if needed so that requests are held inside the XMLServer rather than out in the TCP backlog queue. The process memory space required for an accepted TCP/IP secure connection should be taken into consideration when setting this configuration parameter. |
| <JOBS> | Custom | The parent element of <JOB> children elements which specify a job type for the XML Server. |
| <JOB> | Custom | Specifies a job type for the XML Server. Multiple <JOB> elements can be defined, each detailing a new job type and the implementing class. |

Network configuration

Set the TCP/IP port number and timeout value on all XML servers.

TCP/IP port number

The TCP/IP port number on which the XML server listens for secure connections. Clients of the XML Server connect to the host on which the server is running and must specify the port that is used for secure communications. The <SERVER_PORT> element specifies the port number. The number must be an available port on the system, this means a port number between about 1000 and 32767. If the server is started with a port that is already in use, this is reported and you can select a different port.

Timeout value

Specify a timeout value for network socket operations to ensure that the job threads are not blocked indefinitely, while reading template files across the network and in the event of any network problems. The <SO_TIMEOUT> element specifies the timeout value (in milliseconds). This option allows a network socket operation to block for the time specified. If the timeout expires, a *java.net.SocketTimeoutException* is raised, although the socket is still valid. A timeout value of zero is interpreted as an infinite timeout. The default value is 60000 (one minute).

Default value configuration

There are a number of default values that can be specified for the server. These are the default printer name, the default paper tray, the default e-mail address, and the default user name.

They are specified using the elements `<DEFAULT_PRINTER>`, `<DEFAULT_TRAY>`, `<DEFAULT_EMAIL>`, `<DEFAULT_USERNAME>` respectively. The values can be anything you wish.

If a job submitted to the XML Server via an instance of the `XMLPrintStream` class includes these values, the defaults will be overridden for that job.

Print command configuration

Once a job is processed by the XML server and provided the client did not request a preview, the server runs its server command.

Note: The server command cannot be set per invocation. If multiple commands are required, multiple XML servers must be used.

The server command is a command that is sent to the system to manipulate the output document. Usually it involves printing or emailing the document, but there are no restrictions on what the command can do other than the ones that are imposed by your system. No built in server commands are provided. The command is free-form and is specified by using the `<SERVER_COMMAND>` element.

The server command uses token substitution to pass parameters to the system. The tokens consist of a % character followed by a letter (it is case-insensitive). Tokens that appear in the server command string are substituted with the relevant value of the token just before the server command is run for each job. The table lists the tokens.

Table 6: XML Server Command Tokens

| Token | Meaning |
|-------|---|
| %p | The name of the printer is set to either the default printer ID attribute on the Users table or the default printer name that is specified in the XML server configuration. The default printer ID represents the user who is trying to print the document. |
| %t | The name of the paper tray as specified in the job configuration received from the client, or the default paper tray as specified in the server configuration. |
| %u | The name of the connecting user, or the default username specified in the server configuration. This might be the application username that the user logged in as. |
| %e | The email address of the connecting user or the default email address that is specified in the server configuration. You can use this if you want to email the result of the XML job back to the user. For example, you might configure a server to email PDF to a user as well as print the PostScript output. You might use this token to configure two servers where one supplies emailed copies and the other generates hardcopies. |
| %f | The name of the file where the document was saved. This is a generated temporary file name and does not include that path to the file. The file extension will depend on the specified job type and will default to .pdf. |

| Token | Meaning |
|-------|--|
| %d | The directory where the temporary file is located. You can use a trailing directory separator character and then specify %d%f or you can leave out the character and use, for example, %d/%f. The XML server does not insert one for you. Care must be taken to use the correct separator character for your system. |
| %% | If you want to use a % character in a command but not as a token, use %% instead. The first % will be removed before invoking the command. |

For example, if the server command is specified as:

```
mail -s 'Your Print Job' %e
```

The %e token is replaced with the email address that is specified for the job (or the default email address if none was supplied).

For more complex server commands, it might be necessary to wrap the actual commands in a batch or script file. This batch file is then run by a server command such as:

```
<SomeLocation>/MyBatch.bat 'Your Print Job' %e
```

The server command tokens are not available in the batch file but are only replaced in the server command that is specified in the server configuration file and must be passed into the batch program as normal parameters.

The main consideration when writing a server command is to identify whether you want the output document of the XML server to be piped to your command or stored in a temporary file for your command to process. This can be chosen by setting one of the mutually exclusive <USE_TMP_FILE> or <USE_PIPE> elements in your configuration.

If you opt to use a temporary file, the document data is written to the temporary file and then the server command is run. The XML server does not delete the temporary file for you. Your server command must do that if that is what you want. The temporary file is named by using the value of the TMP_FILE_ROOT element with a sequence number and the appropriate extension is appended according to the job type. For example, if the value was temp, and the job type was XMLPrintStreamConstants.kJobTypePDF, the first file that is generated by the XML server is *temp0.pdf*, the next file is *temp1.pdf*, and so on. This is useful if you start several XML servers that all share the temporary directory to avoid servers over-writing each other's temporary file. The file is created in the directory that is specified by the <TMP_DIRECTORY> element in the configuration. This element must contain an absolute path or a path relative to the directory in which the XML server was started. The directory name and the generated file name are made available to your command by using the %d and %f tokens.

If you opt to use a pipe, your command is run and the XML server writes document data to the standard input of the command. No temporary file is created. However, there is an issue that can occur and must be resolved when using pipes. If the command writes buffered data to a standard error or standard output that is not read by any process, the command might block when the buffer is full. As no process ever reads from the streams, the command will remain blocked or hang indefinitely. There are two methods that you can use to avoid this. First, you can redirect all unused output from your command to a device that can read the output and ensure that the process does not block. Second, you can have the XML server do this for you by using the <USE_STDOUT_SINK> and <USE_STDERR_SINK> elements. While the former method is

recommended where possible, using the XML server sinks can help in situations or on systems where it is not possible. Both elements cause the creation of threads in the XML server that read and discard data output by the server command.

For more information about how to write server commands, see [Sample configuration files on page 38](#).

Template cache configuration

Each job submitted to the XML Server requires an XSL template to be applied to an XML document. Both the template and the document must be supplied by the client. As it is likely that a template may be used more than once, the server can be instructed to store copies of the templates in local files rather than request that the client send a new copy of a template each time it is used.

The cache is enabled using the element `<USE_TEMPLATE_CACHE>`. The templates are then stored in the directory specified using the `TEMPLATE_CACHE_DIR` element. Only templates that are supplied to the `XMLPrintStream` with a template ID and template version number will be cached.

The files in the template cache are not deleted when the XML Server is shut down. They will be reused the next time the server is started. If this behavior is not desired, the `<CLEAR_TEMPLATE_CACHE>` element will ensure that all files in the template cache directory are deleted on server start up.

Debug configuration

You can enable tracing on all network traffic that is received by the server to debug issues.

If the server complains that your XSL template or XML document contain errors, you can view what the server sees by tracing all network traffic that is received by the server. Use the element `<TRACE_TRAFFIC>` to enable the debugging feature. The output is written to the server's standard output.

For secure server communications, lines in the template that start with a period (".") have an extra period character that is inserted. Because input fields that contain a period on a line by itself (that is, "." surrounded by "\n" or "\r") cause the XML server, when the data is processed, to throw an error. The XML server uses this character sequence to mark the end of client transmission, however in the context of data that is entered from a web client this is undesirable behavior. The end of the client transmission is marked by a line that contains only a single period. You can ignore these extra periods.

Apache Log4j 2 logging

You can improve logging performance by using Apache Log4j 2.

Logging with Apache Log4j 2 improves the performance of logging. Logging can be configured by using the `log4j2.properties` file in the XML Server directory. For more information on how to configure Apache Log4j 2, see [Apache Log4j 2](#).

RenderX configuration

The XML Server provides support for RenderX as an alternative to the Apache FOP document rendering engine.

It must be installed on the server on which XML Server is running before it can be used within the XML Server.

<RENDERX_CONFIG_FILE> is used to locate the configuration file that is required by RenderX engine to start.

<RENDERX_LOGGING> is used to configure internal logging for RenderX. The following options are available:

- default - the RenderX DEFAULT_LOGGER is used to log information.
- null - the RenderX NULL_LOGGER is used to log information.
- File Path - the RenderX DEFAULT_LOGGER is used, but the logging stream is redirected to the file specified.

The default value for this property is default. Further information on DEFAULT_LOGGER and NULL_LOGGER can be found on the RenderX website.

Related information

[RenderX website](#)

Custom configuration

The XML Server provides support for defining custom rendering implementations, which allows the use of third party rendering tools. A custom rendering implementation can be added in the form of a new job type; alternatively the default implementation can be replaced.

By default, the XML Server provides four <JOB> definitions catering for processing four types of documents: HTML, RTF, TEXT, PDF. The following list contains the default rendering implementations for each document type:

- HTML - `curam.util.xmlserver.HTMLDocumentGenerator`
- RTF - `curam.util.xmlserver.RTFDocumentGenerator`
- TEXT - `curam.util.xmlserver.TEXTDocumentGenerator`
- PDF - `curam.util.xmlserver.PDFDocumentGenerator`

The default document formatting solution uses Apache Formatting Objects Processor (FOP) to define processing for the document types HTML, PDF, RTF, TEXT. This default implementation can be replaced with a custom implementation by implementing the `curam.util.xmlserver.DocumentGenerator` interface.

Due to FOP's limited capabilities on processing Right-To-Left (RTL) documents, a second pdf rendering tool can be used to specifically handle RTL documents. This can be done using the *direction* attribute when defining a <JOB>. This attribute is optional, and only applicable for pdf job type. The possible values it may contain are: `rtl` and `ltr`. The default value is `ltr`.

Custom job type

A new job type is specified using a `<JOB>` element which must be created with the `<JOBS>` element.

The new job type should be specified using the `type` attribute. This attribute is not case sensitive, and may not contain spaces. Attribute `class` should be used to specify the fully qualified name of the class implementing the `curam.util.xmlserver.DocumentGenerator` interface.

For example:

```
<JOB type="CUSTOM_JOB_TYPE" class="custom.JobImpl" />
```

The configuration file supports the definition of any number of `<JOB>` elements.

The `curam.util.xmlserver.DocumentGenerator` interface requires the following two methods to be implemented.

```
/**
 * This method should be implemented to generate the document
 * for the custom job type. The method is provided with the
 * xml template and xml data to be merged to create the
 * document. The document result should be sent to the
 * output stream provided.
 *
 * @param xslTemplate The XSL template transformer.
 * @param xmlDataStream The input stream from which to read
 * the XML data.
 * @param docOutput The output stream for the generated
 * document.
 *
 * @throws XMLJobException Generic exception to be thrown on
 * error. Exception handling should be handled within the
 * implemented method.
 */
void generateDocument(final Transformer xslTemplate,
    final InputStreamReader xmlDataStream,
    final OutputStream docOutput)
    throws XMLJobException;

/**
 * This method should return a String containing the file
 * extension for the file to be generated. For example if
 * generating a HTML file the method should return the
 * String ".html".
 *
 * @return The extension of the file to be generated.
 */
String getFileExtension();
```

Font configuration

By default, the XML Server uses FOP (Formatting Objects Processor) for rendering documents in various formats. FOP supports a default set of fonts, including Helvetica, Times and Courier, and it is possible using a FOP configuration file to include support for additional fonts, for example a simplified Chinese font.

The `<FOP_CONFIG_FILE>` configuration option allows you to specify the name and location of a FOP configuration file. The path specified for the configuration file can be absolute, for example, `c:/directory/fop-config-file.xml`, or relative to the `xmlserver` directory, for example, `(./fop-config-file.xml`. Any references to files within the FOP configuration file can also be absolute or relative to the `xmlserver` directory.

The following is a sample FOP configuration file:

```
<fop>
  <renderers>
    <renderer mime="application/pdf">
      <font>
        <font metrics-url=".\\chinese\\pmingliu.xml" kerning="yes"
          embed-url=".\\chinese\\mingliu.ttc">
          <font-triplet name="PMingLiu" style="normal"
            weight="normal"/>
        </font>
      </font>
    </render>
  </renderers>
</fop>
```

The example FOP configuration file references a font metrics file called *pmingliu.xml*, and an embed file called *mingliu.ttc*. The embed file is the true type collection font file. True type collection font files can be found on a Windows machine in the installed fonts directory, for example *c:/Windows/Fonts*. Apache provides utilities to generate the necessary font metrics file from a true type collection font file and also from other formats. The Apache FOP documentation should be consulted for more information on font configuration.

Securing the XML server

Secure an XML server that is remotely located from the client by setting up certificates and encrypting data. The XML server processes jobs submitted by a client to produce a formatted document in either PDF, RTF, HTML, or plain text.

About this task

Use TLS v1.2 to secure communication between the XML server and client by configuring the server certificate and encrypting data in motion.

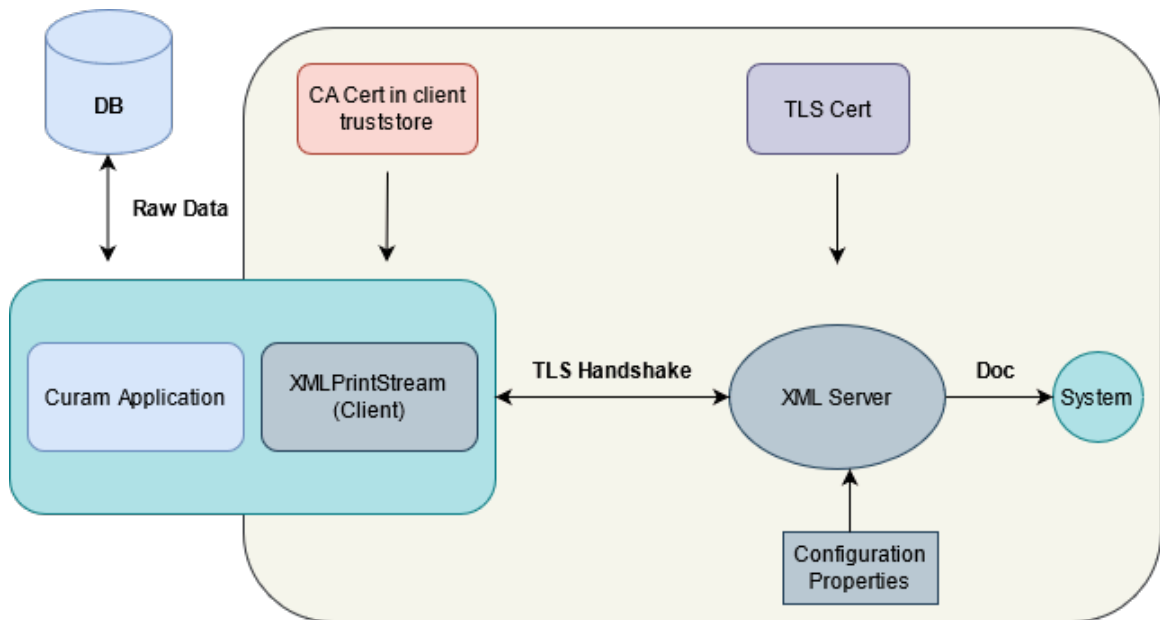


Figure 2: XML Processing Architecture

Procedure

1. Building a script with *TLSKeystore.properties*.

The XML server uses the newly added *TLSKeystore.properties* file to build a script to specify the keystore and certificate properties that enable TLS communication between the XML server and client.

2. The *TLSKeystore.properties* file properties are as follows:

- `keystore.filename` - The name of the keystore file. The default is `keystore.p12`, but you can specify your own file name.
- `keystore.location` - The location of the keystore file. `keystore.location` must be the full path to the keystore. For example, `/Home/Server/KeystoreFolder/`. The default is the current directory.
- `keystore.type` - Specifies the type of keystore used. 'PKCS12' is the default and preferred, but you can also use 'JKS'.
- `keystore.alias` - The alias that is used in keystore generation. The default is 'xmlserverks'.
- `keystore.password` - The password for the keystore. This is mandatory.
- `key.algorithm` - Specifies the algorithm to be used to generate the keystore. The default is 'RSA'.
- `key.size` - Key size in bits. The default is 2048.

- `key.validity` - The number of days that the keystore is valid for. The default is 1825, which is 5 years, excluding leap years.

If the certificate exists, modify the `certificate.filename` property to be the existing certificate's file name. This existing certificate must be located in the path that is specified in the `keystore.location` property, if specified. The existing certificate must be in Base64 encoded format. If certificate exists, the remaining properties are not needed:

- `certificate.filename` - The name of the certificate to be used by the XML Server. The default is `server.cer`.
- `common.name` - The default is 'XML Server'.
- `san.name` - The Subject Alternative Name (SAN) for the application server connects to the XML server. It may check to make sure that the hostname or IP address from the XML Server's certificate SAN matches the hostname or IP address used when establishing the connection. The default is `san.name=dns:xmlserver,dns:localhost,ip:127.0.0.1`.
- `organisation.unit` - The default is an empty string.
- `organisation` - The default is an empty string.
- `locality` - The default is an empty string.
- `state` - The default is an empty string.
- `country` - The default is an empty string.

3. Generating a keystore:

- You can specify your own keystore and certificate by changing the relevant properties in `TLSKeystore.properties`. The keystore and certificate location must be the same and must be specified in the `keystore.location` property.
- If you do not have a pregenerated keystore, the XML Server build script generates one for use in TLS communication. When the XML Server is started, the build script calls the newly added targets that check whether a keystore exists. If it does not, the build script uses the Java™ keytool and the properties that are specified in `TLSKeystore.properties` to generate a new keystore and self-signed certificate.
- The build scripts then attempt to add the certificate authority certificate to the truststore. That is, the public key certificate of newly generated self-signed certificate or CA certificate of the customer's certificate to the truststore of `JAVA_HOME`. For example, `$JAVA_HOME/lib/security/cacerts`.
- The targets that create a new keystore and certificate are also called during the server configuration stages of IBM® WebSphere® Application Server and Oracle WebLogic Server. For both WebSphere® Application Server and WebLogic Server the configuration scripts call the build targets to generate a new keystore and certificate unless a different keystore is specified in `TLSKeystore.properties`.
- In WebSphere® Application Server, the default key and truststores are stored in the node directory of the configuration repository. Ensure that the XML Server certificate is added to the truststore to allow communication between the XML server and the application server. For example, the default trust.p12 stores are created with the `AppSrv01` profile name, the `myhostNode01Cell` name, and the `myhostNode01` node name. The truststore is located in the following location: `/opt/IBM/WebSphere/AppServer/`

```
profiles/AppSrv01/config/cells/myhostNode01Cell/nodes/
myhostNode01/trust.p12
```

- In Weblogic, the XML Server certificate loaded into `$JAVA_HOME/jre/lib/security/cacerts` will allow communication between the XML server and the application server.

Note: Troubleshooting tips:

- Make sure that the certificate truststore of `JAVA_HOME`, for example: `$JAVA_HOME/lib/security/cacerts` has write permission.
- If the XML Server fails to start due to corrupted generated default keystore `keystore.p12`, delete the `keystore.p12` and `server.cer` files from the XML Server folder and restart the XML Server.
- When using the `JAVA_HOME` default certificate truststore, for example: `$JAVA_HOME/lib/security/cacerts`, the XML Server assumes that the default password of the truststore is “changeit”. If the password is different from this, search for the string “changeit” in `xmlserver.xml` file and replace all the instances with the relevant password before restarting the XML Server.

Overriding the default port

The Cúram XML Server application comes and runs with a default configuration file which is generated each time the application is started.

To override the default port the `-Dxmlserver.port` option can be specified, overriding the Ant script. For example:

```
ant -file xmlserver.xml -Dxmlserver.port=1805
```

Overriding the default configuration

The XML server application has a default configuration file which is generated each time the application starts.

To override this default version, take a copy of the `xmlserverconfig.xml` and place in a custom location. The `xmlserverconfig.xml` is created from the `xmlserverconfig.xml.template` file the first time the XML Server is run. This file contains all the configuration elements for the XML Server.

To start the server using this custom configuration use the following Ant command:

```
ant -f xmlserver.xml -Dxmlserver.config.file=C:\Custom
\xmlserverconfig.xml
```

Overriding the Java thread stack size

If large XML or XSL files are passed to the XML server for processing, a stack overflow exception might occur. To prevent stack overflow exceptions from occurring, you can increase the Java thread stack size to enable the XML server to process larger XML or XSL files.

About this task

The Java thread stack size is determined by the value of the `java.thread.stack.size` property. The `java.thread.stack.size` property is specified in the `xmlserver.xml` Ant script that starts the

XML Server. The default value of the property, which is `-Xss4m`, sets the Java thread stack size to 4 Mb.

Procedure

To override the default value of the `java.thread.stack.size` property, when you start the XML server, use the `-Djava.thread.stack.size` build command syntax.

The command in the following example sets the value of the `java.thread.stack.size` property to `-Xss8m`, and therefore sets the Java thread stack size to 8 Mb:

```
ant -Djava.thread.stack.size="-Xss8m" -f xmlserver.xml
```

Disabling configuration file schema validation

To disable validation, you can specify the `novalidation` option as an extra argument to the Ant script invocation.

For example:

```
ant -file xmlserver.xml -Dadditional.args=-novalidation
```

Sample configuration files

In this section a number of samples are presented to illustrate ways the XML Server can be configured

The configurations are dependent on the platform or operating system used, and include the following:

- Printing a document (Windows);
- Displaying a document for testing purposes (Windows);
- Printing a document (UNIX and IBM® z/OS®).

Where path names are specified (e.g. to commands) your customizations may need to be changed if you base your configurations on any of these samples.

The server command (and all other options) should be entered on a single line in the configuration file. In this document they may display with line wrapping for formatting purposes, however,, in your implementation they will need to be specified on a single line to be valid.

Printing a document on Windows™

On Microsoft™ Windows™ servers, the server command (specified in the `<SERVER_COMMAND>` element) is not executed in a command shell unless explicitly invoked via the Windows™ command interpreter (`cmd.exe`) and this is necessary in order to use such facilities as pipes and redirection. The configuration described here is representative for Windows™ platforms.

Depending on the file type, your printing requirements, and the target printer there are a number of possible options and configurations for printing on Windows™. For instance, your particular version of Adobe™ Reader may allow for direct printing or your printer may support direct PDF printing.

A convenient way to implement print functionality is to write a batch file for the Windows™ command interpreter to invoke and perform any necessary operations and to get the server to execute this batch file. A sample batch file is shown in [Printing a document on](#)

[Windows™ on page 38](#) below. Let us assume that the batch file is saved as `c:\xmlsrv\xmlserverprint.bat`¹. The server command can pass parameters to the batch file through the command line and the batch file accesses these as %1 for the first parameter, %2 for the second, etc. These parameters are provided to the batch file via the server command tokens specified in the batch file invocation in the server configuration file and replaced when it is invoked. (See [Print command configuration on page 29](#) and [Printing a document on Windows™ on page 38](#) for more information on command tokens.)

While Windows™ applications sometimes allow the use of either forward-slash (/) or back-slash (\) characters interchangeably as a path separator, the Windows™ command interpreter only allows the \ character. Care must be taken to ensure that all paths that may be visible to the command interpreter use back-slash characters (\) as separators. As path information will not be available in the context of your batch file, commands must have fully specified paths. The interpreters built-in commands do not require a path.

The following example illustrates the use of the sample `SimplePrintService` class, which is implemented using the Java™ Print Service API. You could utilize this API for your own custom solution; for instance, to utilize specific printer features in your environment. To print a PDF file using this sample class would require the printer to have direct PDF print support.

The following sample is a sample batch file for printing a document in Microsoft Windows:

```
@ECHO OFF

echo ----- ^
  >> XMLServer.log
REM log output
echo File:      %1      ^
  >> XMLServer.log
echo Print Server: %2      ^
  >> XMLServer.log

REM Call the system print command
echo Starting Print      ^
  >> XMLServer.log
echo %JAVA_HOME%\bin\java      ^
  -cp xmlserver.jar;xmlservercommon.jar      ^
  curam.util.xmlserver.SimplePrintService      ^
  %2 "%1" >> XMLServer.log 2>&1      ^
%JAVA_HOME%\bin\java      ^
  -cp xmlserver.jar;xmlservercommon.jar      ^
  curam.util.xmlserver.SimplePrintService      ^
  %2 "%1" >> XMLServer.log 2>&1      ^
echo Printing Completed      ^
  >> XMLServer.log
echo ----- ^
  >> XMLServer.log
```

Instead of the sample Java™ program above any appropriate processing could be specified or additional processing prior to printing or cleanup after printing can also be implemented as needed. If you use any command that may send output to the console, make sure that you add null redirection. This output needs to be redirected to the null device or it will cause the command to block and the batch file will hang. Therefore, redirection must be added to the command pointing to the null device; e.g.: `> nul:`, which avoids the problem of blocking the XML Server.

¹ Note that you should choose a target destination for setting up your XML Server and its customizations to avoid being overwritten by subsequent service pack updates.

Note: Setting the `<USE_STDOUT_SINK>` and `<USE_STDERR_SINK>` elements in the configuration will not work on Windows™.

The following is a sample configuration file used to launch the batch file . Note how the printer name and the details of the temporary file are passed to the batch file using the command tokens.

```
<XML_SERVER_CONFIG>
  <SERVER_PORT>6789</SERVER_PORT>
  <SERVER_COMMAND>
    c:\Windows\System32\CMD.EXE
    /C c:\xmlsrv\xmlserverprint.bat %d%f %p
  </SERVER_COMMAND>
  <USE_TMP_FILE>true</USE_TMP_FILE>
  <TMP_FILE_ROOT>temp</TMP_FILE_ROOT>
  <TMP_DIRECTORY>c:\xmlsrv\tmp</TMP_DIRECTORY>
  <DEFAULT_PRINTER>\\MyPC\ps1</DEFAULT_PRINTER>
  ...
</XML_SERVER_CONFIG>
```

The command interpreter (`cmd.exe`) uses the `/C` option to specify a batch file to execute. The batch file is passed two parameters. The first parameter is the name of the temporary PDF file created by concatenating the expanded `%d` token for the temporary directory name, a back-slash separator, and the expanded `%f` token for the name of the temporary PDF file. The second parameter is the expanded `%p` token for the name of the printer. The configuration file also includes a default printer name. But this may be overridden by the client. See [Print command configuration on page 29](#) for a more detailed description of these tokens.

Displaying a document for testing on Windows

When testing a new XSL template against XML data, it is useful to see the PDF output without printing it each time. If the code you are writing does not use the preview facilities of the `XMLPrintStream` class, you will need to look at the PDF output of the XML Server manually.

A simple solution is to run an XML Server on your development machine and configure it to open Adobe Reader to display the PDF data each time you submit a job. This will save you from running to a printer or manually opening PDF files. The configuration is shown in the following sample:

```
<XML_SERVER_CONFIG>
  <SERVER_PORT>6789</SERVER_PORT>
  <SERVER_COMMAND>c:/PROGRA~1/Adobe/AcrobatReader/AcroRd32.exe
  %d/%f</SERVER_COMMAND>
  <USE_TMP_FILE>true</USE_TMP_FILE>
  <TMP_FILE_ROOT>temp</TMP_FILE_ROOT>
  <TMP_DIRECTORY>c:/xmlsrv/tmp</TMP_DIRECTORY>
  ...
</XML_SERVER_CONFIG>
```

You cannot include space characters in the path to the server command as Java will interpret these as the end of the command file name and there is no way of escaping them. To avoid the problem, the above configuration file shows how the DOS short name of the directory containing the space character is used: `PROGRA~1` instead of `Program Files`. As the command was not passed to a command interpreter, the choice of `/` or `\` as a path separator character is arbitrary.

Installing RenderX for Right-To-Left PDF document processing on Windows

As Right-To-Left (RTL) writing languages are not supported in Apache FOP, the XML Server also provides the functionality to use alternative rendering tools.

RenderX is a third party document rendering engines that supports RTL writing languages. If RenderX is installed, and the XML Server is configured to use RenderX, the XML Server will automatically use RenderX to generate all RTL PDF documents. In order to use the default RenderX implementation in Cúram the following steps should be completed:

- Install RenderX according to the RenderX installation guide.
- Set a system environment variable `RENDERX_HOME` to point to the directory in which you have installed RenderX.
- Customize `xmlserver_config.xml` to use `curam.util.xmlserver.RenderXDocumentGenerator` to process Right To Left PDF documents. The following sample is a example of how to set up RenderX for RTL Document processing

```
<XML_SERVER_CONFIG>
...
<RENDERX_CONFIG_FILE>C:/RENDERX/xep.xml</RENDERX_CONFIG_FILE>
<RENDERX_LOGGING>off</RENDERX_LOGGING>
...
<JOBS>
...
  <JOB type="pdf" direction="RTL"
    class="curam.util.xmlserver.RenderXDocumentGenerator"/>
</JOBS>
</XML_SERVER_CONFIG>
```

The customizations in this example assume RenderX is installed to `c:/RenderX` directory

Note: In order to use a relative path with a default installation of RenderX, the images should be stored relative to the RenderX location. For example, if the `RENDERX_HOME` is `C:\projects\RenderX\`, and the images are stored in `C:\projects\RenderX\images`, then the relative path to an image would be `"/images/curam/curam.jpg"` which is the equivalent of `C:\projects\RenderX\images\curam\curam.jpg`.

Printing a document on UNIX and z/OS

Printing a document on UNIX and z/OS can be done similarly to Windows in that an invoked shell script can execute commands or other necessary processing. That is, you write a shell script that is invoked by the XML Server as per your configuration and the shell script performs the processing specific to the platform. For example, see [Printing a document on UNIX and z/OS on page 41](#) below. Let us assume that the shell script is saved as `/usr/local/xmlsrv/xmlserver.sh`². The server command can pass arguments to the shell script, which are accessed in a typical way: `$1` for the first parameter, `$2` for the second, etc. These arguments are provided to the shell script via the server command tokens specified in the script invocation in the server configuration file and replaced when the script is invoked. (See [Print command configuration on page 29](#) and [Printing a document on UNIX and z/OS on page 41](#) for more information on command tokens.)

² Note that you should choose a target destination for setting up your XML Server and its customizations to avoid being overwritten by subsequent service pack updates.

In general, printing capabilities vary widely by OS distribution, version, installed software, physical printer capabilities, etc. Review your local environment for requirements and how to best implement printing support. For instance, a z/OS implementation might use the IBM® InfoPrint® Server³.

The following example illustrates how printing might be done on various UNIX platforms. For instance, as on z/OS, if the software and printer hardware supports it direct printing via the the system print command (**lp** or **lpr**) may be possible. On IBM® AIX® you would require third-party software to convert the input PDF to PostScript for printing. For ease of monitoring the script contains **echo** commands to provide progress during its execution and appends the output to a file named *XMLServer.log*.

Note: On the z/OS platform you will have to convert the encoding of the *xmlserverprint.sh* script from ASCII to EBCDIC. For example:

```
tr -d '\15\32' < xmlserverprint.sh > xmlserverprint.sh-ASCII
iconv -t IBM-1047 -f ISO8859-1 xmlserverprint.sh-ASCII \
> xmlserverprint.sh
chmod a+rx xmlserverprint.sh
```

³ The installation and configuration of the InfoPrint Server is beyond the scope of this document.

The following is a sample shell script for printing a document on UNIX and z/OS systems:

```
#!/bin/sh

# Sample UNIX script for XMLServer printing.

echo ----- \
  >> XMLServer.log
# log output
echo File: $1 >> XMLServer.log
echo Print Server: $2 >> XMLServer.log
Platform=`/bin/uname`
echo Platform: $Platform >> XMLServer.log

# The following illustrates some possible print solutions
# for various platforms:

case $Platform in
  # z/OS:
  OS/390)
    # On OS/390 (z/OS) use of the lop command as
    # illustrated would be dependent on the InfoPrint
    # Server installation and configuration, related
    # software, and a printer with direct PDF support
    # and sufficient memory.
    echo Starting print... >> XMLServer.log
    lp -d $2 $1
    echo Printing Completed >> XMLServer.log
    ;;
  AIX)
    # AIX has no native print support for PDF files,
    # so you would need to implement functionality such as
    # pdf2ps to convert the generated PDF file to
    # PostScript for printing with lpr; e.g.:
    # see the IBM Redbook SG24-6018-00
    # pdf2ps $1 $1.ps
    # lpr -P $2 $1.ps
    echo $Platform printing implementation is TBD. \
    >> XMLServer.log
    ;;
  # Other platforms:
  *)
    # Your local print functionality to be implemented here ...
    echo $Platform printing implementation is TBD. \
    >> XMLServer.log
    ;;
esac

echo ----- \
  >> XMLServer.log
```

The configuration file used to launch this shell script is shown in [Printing a document on UNIX and z/OS on page 41](#) below. Note how the printer name (%p) and the details of the temporary file (%d and %f) are passed to the shell script using the command tokens. These are interpreted by the shell as two arguments inside the script: 1) The temporary directory and file name are concatenated with a forward-slash separator; and 2) name of the printer, which may be overridden by the client. See [Print command configuration on page 29](#) for a more detailed description of these tokens.

The following sample is an example configuration for printing a document on UNIX and z/OS systems:

```
<XML_SERVER_CONFIG>
...
<SERVER_COMMAND>
./xmlserverprint.sh %d/%f %p
</SERVER_COMMAND>
<USE_TMP_FILE>true</USE_TMP_FILE>
<TMP_DIRECTORY>./tmp</TMP_DIRECTORY>
<TMP_FILE_ROOT>doc</TMP_FILE_ROOT>
<DEFAULT_PRINTER>printer1</DEFAULT_PRINTER>
...
</XML_SERVER_CONFIG>
```

Starting the XML server

The XML server application is delivered as a separate component in Cúram. The XML server is started from the XML server installation directory using Apache Ant.

The following is an example of the ant command to start the server:

```
ant -file xmlserver.xml
```

You can also start the server with the additional options that are described in [Disabling configuration file schema validation on page 38](#) and in [Statistics on page 45](#).

A default *xmlserver_config.xml* is provided when you install the product which contains the default configuration file for the server. You can apply changes to this file as required.

When the server starts, it displays the configuration information it has read from the configuration file and displays the status of each job it receives.

Note: In addition to running as a command line application, you can also run the XML server in the background as a Windows service as described in [Running the XML server as a Windows service or UNIX daemon on page 44](#).

Running the XML server as a Windows service or UNIX daemon

For a production environment, it can be more effective, for purposes of ensuring availability at restart, avoiding accidental shutdowns via an open shell prompt, and so on, to run the XML server as a Windows service or UNIX daemon.

To run a program as a Windows service requires specific Windows infrastructure, that is, batch files and programs cannot be run this way out-of-the-box. However, there are third-party tools available to enable this function, for example, the Java Service Wrapper from Tanuki Software.

With Tanuki Java Service Wrapper, after installation, you can integrate the XML server using the `WrapperStartStopApp` class by setting `wrapper.java.mainclass` to `org.tanukisoftware.wrapper.WrapperStartStopApp`, and you must do the following:

- Set the class path to include the necessary Ant libraries
- Pass the Ant home into the environment
- Ensure adequate memory, for example, 768 MB
- Pass in the necessary parameters to start the XML Server Ant script.

Specifically, for the Java Service Wrapper the properties would look like:

```
wrapper.java.classpath.<n>=<ANT_HOME>/lib/ant.jar
wrapper.java.classpath.<n>=<ANT_HOME>/lib/ant-launcher.jar
wrapper.java.additional.<n>=-Dant.home=<ANT_HOME>
wrapper.java.maxmemory=768
wrapper.app.parameter.1=org.apache.tools.ant.launch.Launcher
wrapper.app.parameter.2=2
wrapper.app.parameter.3=-f
wrapper.app.parameter.4=<CURAMSDEJ>/xmlserver/xmlserver.xml
wrapper.app.parameter.5=org.apache.tools.ant.launch.Launcher
wrapper.app.parameter.6=true
wrapper.app.parameter.7=3
wrapper.app.parameter.8=-f
wrapper.app.parameter.9=<CURAMSDEJ>/xmlserver/xmlserver.xml
wrapper.app.parameter.10=stop
```

Note: The values in angle brackets must be substituted with the appropriate values for your installation. See the Java Service Wrapper documentation for more details on installation, configuration, and running.

Running the XML Server as a UNIX daemon is something that can typically be done with shell scripting and system facilities, for example, **cron**, but, UNIX compatible versions of Java Service Wrapper are available.

Related information

[Tanuki Software](#)

Shutting down the XML server

You can use the **XMLServerShutdown** command to shutdown the XML server.

In an environment where few jobs are printed or you can be sure the XMLServer is idle, you can safely shut down the XML Server with a simple Control-C key combination without causing any problems. However, the recommended and safer method is to use the **XMLServerShutdown** command. This will shut down any XML Server in an orderly fashion: the server will refuse any new jobs and allow all outstanding jobs to complete before exiting. This is done through the following Ant command:

```
ant -file xmlserver.xml stop
```

The server will be switched into shut down mode and all outstanding jobs will be completed before the server exits and the **XMLServerShutdown** command informs you that the server has been shut down. Depending on the number of jobs being processed, this may take some time to complete.

Statistics

Statistics for the XML server are available in the statistics folder that you specified in the configuration. If Cúram is deployed on Kubernetes, you can send the XML server statistics to a metrics tool for monitoring.

When you shut down the XML server, various statistics data for the XML server is collected in the statistics folder that is specified in *xmlserverconfig.xml*.

The statistics log includes the following columns:

- Success - Whether the job was successful (true, false).
- Job preview type - The job preview type (PDF, HTML, TEXT, RTF).
- Elapsed secure connection - the time elapsed (in milliseconds) since processing of a connection started until the connection was closed.
- Elapsed job - The time (in milliseconds) it takes to run the job.
- Elapsed job preview send - The time (in milliseconds) it takes to send the preview data to the client.
- Job preview data length - The length of the preview data (in bytes) that is sent to the client.
- Timestamp - The timestamp (Java timestamp value) when the secure connection entered the system.
- Template ID - The ID for the template that is being processed.
- Template version - The version number of the template that is being processed.
- Template locale - The locale of the template that is being processed.

Sending statistics to a metrics tool

In Kubernetes environments, XML server statistics can be made available as they occur by specifying `-forcestatswrite` as an additional argument when you start the XML server.

For example:

```
ant -f xmlserver.xml -Dadditional.args="-forcestatswrite"
```

For more information about how to make the XML server statistics available, see [Monitoring XML servers](#) in the *Cúram Kubernetes Runbook*.

1.4 Cúram XML and XSL templates

In this chapter, you will learn about the Cúram XML format used for all XML documents generated by your application server. You will need to know this format if you wish to write XSL templates for formatting and printing the XML documents.

Every XML document generated by the XML infrastructure uses a fixed format regardless of the struct classes being converted. This makes the development of XSL templates easier, as the format of the XML does not change. The following sections present that format and show what Cúram XML documents look like. This will help you when you are developing XSL templates.

Cúram DTD

The following markup declaration is the DTD for Cúram XML. The DTD can be found in the *lib* directory of the SDEJ. The comments within the declaration describe each element.

```
<!--A DOCUMENT element has an optional META element
      followed by a mandatory DATA element.-->
<!ELEMENT DOCUMENT (META?, DATA)>

<!--A META element has a number of optional elements that
      it can contain in no particular order.-->
<!ELEMENT META (GENERATED_DATE | GENERATED_BY |
                VERSION | COMMENT)*>

<!--A DATA element contains a single mandatory STRUCT_LIST
      or STRUCT element.-->
<!ELEMENT DATA ((STRUCT_LIST | STRUCT))>

<!--A STRUCT_LIST element has one or more STRUCT
      elements.-->
<!ELEMENT STRUCT_LIST (STRUCT+)>

<!--A STRUCT element has an optional SNAME element and one
      or more FIELD elements.-->
<!ELEMENT STRUCT (SNAME?, FIELD+)>

<!--A FIELD element has an FNAME and either a TYPE
      element and a VALUE element, or a STRUCT_LIST element,
      or a STRUCT element (in that order).-->
<!ELEMENT FIELD (FNAME, ((TYPE, VALUE) | STRUCT_LIST | STRUCT))>

<!--All these elements contain parsed character data only
      and do not contain sub-elements. Use ISO-8601 when
      formatting date values.-->
<!ELEMENT GENERATED_DATE (#PCDATA)>
<!ELEMENT GENERATED_BY   (#PCDATA)>
<!ELEMENT VERSION         (#PCDATA)>
<!ELEMENT COMMENT         (#PCDATA)>
<!ELEMENT SNAME           (#PCDATA)>
<!ELEMENT FNAME           (#PCDATA)>
<!ELEMENT VALUE           (#PCDATA)>
<!ELEMENT TYPE            (#PCDATA)>

<!--A TYPE element can have a SIZE attribute. If not
      supplied, the attribute will not be set by default
      and will have a null value. This is normally used
      for SVR_STRING types.-->
<!ATTLIST TYPE SIZE CDATA #IMPLIED>
```

Examples

Examples of simple XML documents generated for a struct, and for a list of structs.

The following example shows a simple XML document generated for a struct that contains two fields. Note that the field types will always be the basic types and not the domain definitions derived from those basic types.

```
<DOCUMENT>
  <META>
    <GENERATED_BY>My Server</GENERATED_BY>
  </META>
  <DATA>
    <STRUCT>
      <SNAME>DPTicketDtls</SNAME>
      <FIELD>
        <FNAME>ticketID</FNAME>
        <TYPE>SVR_INT64</TYPE>
        <VALUE>12796</VALUE>
      </FIELD>
      <FIELD>
        <FNAME>subject</FNAME>
        <TYPE SIZE="100">SVR_STRING</TYPE>
        <VALUE>This is the subject.</VALUE>
      </FIELD>
    </STRUCT>
  </DATA>
</DOCUMENT>
```

In the next example, the format of an XML document describing a list of structs is presented. Note that the <STRUCT> elements are the same as previously, but multiple <STRUCT> elements are contained within a <STRUCT_LIST> element.

```
<DOCUMENT>
  <META>
    <GENERATED_BY>My Server</GENERATED_BY>
  </META>
  <DATA>
    <STRUCT_LIST>
      <STRUCT>
        <SNAME>DPTicketDtls</SNAME>
        <FIELD>
          <FNAME>ticketID</FNAME>
          <TYPE>SVR_INT64</TYPE>
          <VALUE>12796</VALUE>
        </FIELD>
        <FIELD>
          <FNAME>subject</FNAME>
          <TYPE SIZE="100">SVR_STRING</TYPE>
          <VALUE>This is the subject.</VALUE>
        </FIELD>
      </STRUCT>
      <STRUCT>
        <SNAME>DPTicketDtls</SNAME>
        <FIELD>
          <FNAME>ticketID</FNAME>
          <TYPE>SVR_INT64</TYPE>
          <VALUE>35667</VALUE>
        </FIELD>
        <FIELD>
          <FNAME>subject</FNAME>
          <TYPE SIZE="100">SVR_STRING</TYPE>
          <VALUE>This is another subject.</VALUE>
        </FIELD>
      </STRUCT>
    </STRUCT_LIST>
  </DATA>
</DOCUMENT>
```

If a field of a struct is itself a struct, then instead of a `<TYPE>` and `<VALUE>` element, the `<FIELD>` element will contain a whole `<STRUCT>` element. Fields can also contain `<STRUCT_LIST>` elements in the same manner.

Job types and template types

Different job types can be specified when you use the `XMLPrintStream` class to communicate securely with the XML Server. These job types require different types of templates to succeed. While all the templates use XSL for formatting, the following two parts of that standard are used in specific situations.

- *XSL Transformations (XSLT)*

XSLT is a standard that defines a language for transforming XML documents in other XML documents. Elements of the XSLT language allow data from one XML document to be combined with static elements of a template (or stylesheet).

- *XSL Formatting Objects (XSL-FO)*

XSL-FO defines a set of elements for describing the physical layout of a document: paper size, fonts, spacing, image locations, and so on. The layout model that is used is based on the model that used for PDF documents. A formatting objects processor can convert data that is marked up with formatting objects into other representations such as PDF or RTF.

The following subsections outline how these standards can be used to develop templates for each of the supported job types.

XSL and XSL-FO are extensive standards and it is beyond the scope of this document to describe them in detail.

Related concepts

[Developing for XML on page 9](#)

The two most important classes you need when adding XML functionality to your applications are `curam.util.xml.impl.XMLDocument` and `curam.util.xml.impl.XMLPrintStream`. The classes can be used together to generate XML and print documents.

Templates for PDF documents

Generating PDF documents is a two stage process.

It is easiest to describe the process in reverse order.

PDF documents are generated from documents marked up with XSL-FO in a process called rendering. The document contains the data that will appear in the document (text, figures, and so on) and the XSL-FO mark-up that is needed to define how this data will be laid out (margins, paper-size, fonts, line-spacing, location of paragraphs, and so on) This rendering stage is handled by the Apache FOP library.

To prepare an XSL-FO document for rendering, the raw data is supplied in an XML document and a template uses XSLT to combine the raw data with the XSL-FO mark-up and the other static elements of the document. In essence, the XSLT inserts the raw data into the template to create the XSL-FO document. This transformation stage is handled by the Apache Xalan library.

Thus, templates for rendering documents as PDF are largely XSL-FO documents with elements of XSLT used to insert values from the XML document at the appropriate point. An example of such a template is given in the next section.

Templates for RTF documents

RTF templates are identical to PDF templates. The same template can be used to produce output in either format.

The template is mostly XSL-FO with XSLT used to insert values from the XML document in the appropriate locations.

The JFOR library is used to render RTF documents from XSL-FO documents, however, not all XSL-FO elements are supported. Unless you need to edit the documents in a word processor after they have been generated, you should use the better supported PDF generator.

Templates for HTML documents

Templates for HTML documents consist of HTML mark-up and XSLT elements that insert values from the XML document in the appropriate locations to create an HTML document.

Templates for HTML documents are simpler than the templates for PDF or RTF. XSL-FO mark-up is not used as the HTML mark-up is used to define the formatting. As such, there is no rendering step when you generate HTML documents. The templates consists of HTML mark-up and XSLT elements that insert values from the XML document in the appropriate locations to create an HTML document.

As XSLT converts only one XML document into another, the output includes some XML elements. These elements are automatically removed for this job type so that the output is a pure HTML document. The HTML is automatically indented during the processing.

Templates for plain text documents

Templates for plain text documents contain no XSL-FO mark-up and there is no rendering step.

The templates comprise plain text with embedded XSLT elements to insert values from the XML document in the appropriate locations.

Again, XML elements in the output document are stripped. As XML and XSL generally do not preserve white-space, use of the `<text>` element around white-space that is to be preserved is advised (for example, line breaks, indentation, etc.).

XSL template example

The following example shows the basic method of identifying and extracting data from an XML document containing a single struct.

```
<?xml version="1.0" standalone="yes"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fo="http://www.w3.org/1999/XSL/Format"
  version="1.0">

  <xsl:template match="DOCUMENT">
    <xsl:apply-templates select="DATA"/>
  </xsl:template>

  <xsl:template match="DATA">
    <xsl:apply-templates select="STRUCT[SNAME='DPTicketDtls']"/>
  </xsl:template>

  <xsl:template match="STRUCT">
    <fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
      <fo:layout-master-set>
        <fo:simple-page-master page-master-name="only"
          page-height="297mm" page-width="210mm"
          margin-top="30mm" margin-bottom="30mm"
          margin-left="30mm" margin-right="30mm">
          <fo:region-body/>
        </fo:simple-page-master>
      </fo:layout-master-set>

      <fo:page-sequence>
        <fo:sequence-specification>
          <fo:sequence-specifier-single
            page-master-reference="only"/>
        </fo:sequence-specification>

        <fo:flow>
          <fo:block font-size="12pt" font-family="serif"
            line-height="20mm">
            Ticket ID: <xsl:apply-templates
              select="FIELD[FNAME='ticketID']"/>
          </fo:block>

          <fo:block font-size="12pt" font-family="serif"
            line-height="20mm">
            Subject: <xsl:apply-templates
              select="FIELD[FNAME='subject']"/>
          </fo:block>
        </fo:flow>
      </fo:page-sequence>
    </fo:root>
  </xsl:template>

  <xsl:template match="FIELD">
    <xsl:value-of select="VALUE"/>
  </xsl:template>

</xsl:stylesheet>
```

The output is formatted for A4 paper (210x297mm) with 30mm margins and should appear like this, if the earlier sample XML document is used:

```
Ticket ID: 12796

Subject: This is the subject.
```

Generating templates from RTF documents

While templates cannot be generated directly from RTF documents, software is available to convert an RTF document created by a word processor into the corresponding XSL-FO document.

Once the XSL-FO document has been generated, you can insert the appropriate XSLT mark-up to convert it into a usable template.

Globalization considerations

Data that is transmitted to the XML server for printing can be sensitive to locale differences.

Structs are transmitted to the XML Server for printing by calling method

```
curam.util.xml.impl.XMLDocument .add(your-struct).
```

Structs are serialized into an XML representation that is then transformed via XSLT into a human-readable document. By default the following data types are serialized by calling their `toString()` method:

- `curam.util.type.Date`
- `curam.util.type.DateTime`
- `curam.util.type.Money`

The `toString()` method of `Date` and `DateTime` returns a string dependent on the value of property `curam.environment.default.dateformat` and the `toString()` method of `Money` returns a value dependent on the value of property `curam.environment.default.locale`.

For example, if the property `curam.environment.default.locale` is set to `en_GB`, a `Money` amount would be formatted as 12,345.67 whereas for `es_ES` it would be formatted to 12.345,67, that is the commas and dots are reversed. This formatting prevents the XSLT from de-serializing the data in a locale-neutral way. So if the server locale was set to English, then the XSL template for a Spanish letter must parse an English formatted numeric string instead of a numeric value.

Locale-related problems can be avoided in the following two ways:

- Use string fields to transfer all data to the XML Server, and ensure that these string fields are correctly formatted for the appropriate locale on the server beforehand.
- Transfer fields to the XML Server in a locale-neutral way by setting the property `curam.xmlserver.serializeLocaleNeutral` to true. For *Date* the format is `yyyyMMdd` and for *DateTime* the format is `yyyyMMddTHH:mm:ss`. For the field *Money* it is the same as for floating point decimals.

Notices

Permissions for the use of these publications are granted subject to the following terms and conditions.

Applicability

These terms and conditions are in addition to any terms of use for the Merative website.

Personal use

You may reproduce these publications for your personal, noncommercial use provided that all proprietary notices are preserved. You may not distribute, display or make derivative work of these publications, or any portion thereof, without the express consent of Merative

Commercial use

You may reproduce, distribute and display these publications solely within your enterprise provided that all proprietary notices are preserved. You may not make derivative works of these publications, or reproduce, distribute or display these publications or any portion thereof outside your enterprise, without the express consent of Merative.

Rights

Except as expressly granted in this permission, no other permissions, licenses or rights are granted, either express or implied, to the publications or any information, data, software or other intellectual property contained therein.

Merative reserves the right to withdraw the permissions granted herein whenever, in its discretion, the use of the publications is detrimental to its interest or, as determined by Merative, the above instructions are not being properly followed.

You may not download, export or re-export this information except in full compliance with all applicable laws and regulations, including all United States export laws and regulations.

MERATIVE MAKES NO GUARANTEE ABOUT THE CONTENT OF THESE PUBLICATIONS. THE PUBLICATIONS ARE PROVIDED "AS-IS" AND WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE.

Merative or its licensors may have patents or pending patent applications covering subject matter described in this document. The furnishing of this documentation does not grant you any license to these patents.

Information concerning non-Merative products was obtained from the suppliers of those products, their published announcements or other publicly available sources. Merative has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-Merative products. Questions on the capabilities of non-Merative products should be addressed to the suppliers of those products.

Any references in this information to non-Merative websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those

websites are not part of the materials for this Merative product and use of those websites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

The licensed program described in this document and all licensed material available for it are provided by Merative under terms of the Merative Client Agreement.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to Merative, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. Merative, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. Merative shall not be liable for any damages arising out of your use of the sample programs.

Privacy policy

The Merative privacy policy is available at <https://www.merative.com/privacy>.

Trademarks

Merative™ and the Merative™ logo are trademarks of Merative US L.P. in the United States and other countries.

IBM®, the IBM® logo, and ibm.com® are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide.

Adobe™, the Adobe™ logo, PostScript™, and the PostScript™ logo are either registered trademarks or trademarks of Adobe™ Systems Incorporated in the United States, and/or other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft™, Windows™, and the Windows™ logo are trademarks of Microsoft™ Corporation in the United States, other countries, or both.

UNIX™ is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.