Oracle® Insurance IStream

IStream Publisher Interface Reference Guide

Release 4.2

E14879-01

January 2009

ORACLE
INSURANCE

Copyright

Copyright © 2009, Oracle and/or its affiliates. All rights reserved.

Primary Authors: Andrew Brooke and Ken Weinberg

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS

Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are “commercial computer software” or “commercial technical data” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

CONTENTS

Chapter 1 — Overview 9

Document Conventions 10

IStream Publisher 11

Queues and Requests 11

IStream Publisher Documentation 13

Contacting Skywire Software for Help 14

Contact Information 14

Support Checklist 14

Chapter 2 — Simple Services 15

Parameters and XML Schema 16

Distributor.xsd 16

Validating Requests 17

Referencing Files 18

Delivering CLG Files through InfoSources 19

JMS Message Header and Properties 20

Request Metadata 21

Detailed Response Parameters 22

Content Service 23

Generate Calligo Document Service Overview 23

Using Referenced and Embedded XML Data 24

Generate IStream Document XML Sample 26

Rendering Services 28

Rendering a Microsoft Word Document to HTML 28

Rendering a Microsoft Word Document to PCL 29

Rendering a Microsoft Word Document to PDF 30

Rendering a Microsoft Word Document to PostScript 32

Rendering Service XML Sample 32

Rendering a Microsoft Word Document to TIFF 32

Rendering a Microsoft Word Document to TXT/RTF 32
Rendering a PDF Document to PCL 33
Rendering a PDF Document to PS 33
Rendering a PDF Document to TIFF 34
Rendering a TIFF Image to PCL 34
Rendering a TIFF Image to PDF 34
Rendering a TIFF Image to Postscript 35
Rendering an IStream Document to Microsoft Word 35

Delivery Service 36
Delivering Content to a Repository 36
Delivering Content to a Printer 37
Delivering Content to an E-mail Server 37
Delivering Content to a Fax Server 39
Delivery Service Request XML Sample 40

Utility Services 41
Run Word Macro 41
Concatenating PCL Streams 41
Concatenating PDF Files 42
Concatenating PS Streams 43
Encrypting PDF Documents 44
Deleting Files and Folders 44

Aggregate Request 47
Aggregate Request Processing 47
Aggregate Request Limitations 47

The Transform Service 49
Sample Request. 49
General Considerations 50
Postscript File with PJL Commands. 50

Chapter 3 — Distribution Service 51

The Distribution Service 52

The Distribution Request 53
The Distribution Request Structure 55
IStream Publisher Distribution Request Failure Policy. 56
Troubleshooting the Distribution Request 57

The Distribution Package 59
The Distribution Item 59

Recipients 62
Recipient 62
Recipient Package 64

Delivery Channels 67
Operating Modes 67

Event Handlers 70
Events 70
Distribution Request Metadata 74
Concatenating PCL Streams 75
Concatenating PS Streams 77

Calling the Transform Service 79

A Distribution Request Example 80

Chapter 4 — Tracking and Monitoring Requests 83

Request Messages 84
Unique Request IDs 84
Live Request Message Status 84

The Request Log Table 85
Request Table 85
The Status Table 86
The ErrorInfo Table 87
The StatusOrder Table 87

Resubmitting a Failed or Canceled Request 88
Distribution Requests 88
Mapping 88

Error Log Levels 89

Chapter 5 — Creating and Using Cover Pages 91

Delivering Cover Pages to Fax and Printer 92

Chapter 6 — SDK – The IStream Publisher Client API 95

The IStream Publisher Client API 96

Client API Interfaces 97

Distributor Factory 97

Session 99

The ResponseListener and ResponseExceptionListener Interfaces 100

Services 100

IStream Publisher Client Exceptions 103

Configuring the IStream Publisher Client API 104

Configuration Files 104

Configuration Implementation 106

Notification of Request Completion 108

Chapter 7 — SDK – Repository API 109

The Repository API 110

The API Architecture 111

Categories of Functionality. 112

Reference Language 113

Uniform Resource Identifiers 113

Query 115

The Connection Interface 116

Connection Factory 116

Creating a Connection 117

The Repository Interface. 119

Repository Objects 121

Object Metadata. 122

Versions 123

Renditions 124

Identifiers **125**
URLStreamHandlerFactory 125
URLStreamHandler 126
URLConnection 127
Content. 128

Adding a New Repository Adapter **129**
Java Code 130
Service Request Example 132

Chapter 8 — SDK – System Extensibility. 133

Creating and Adding a Simple Service **134**

Extending the Distribution Service **137**
Event Handlers. 137
Event Handlers in the Distribution Request 138
Distribution Request with Event Handler Example. 145
Distribution State DAO 151

Customizing a Request Log Message **153**
Customizing the Request Log Table 153
Adding Custom Fields 153
The Request Log Table 153

Chapter 9 — SDK – Web Service Interface 155

The Web Services Interface **156**
About Web Service Applications 156
IStream Publisher WSI Benefits. 156

IStream Publisher WSI Architecture **157**
General Information 157
Overview of WSI Architecture 157
Web Services Interface Methods 158
Flows of IStream Publisher WSI Calls 159
WSI WSDL. 164

Configuring the IStream Publisher WSI in the Console **165**
WSI Client Examples 166

Troubleshooting the IStream Publisher WSI **167**
IStream Publisher WSI Log files 167

Appendix A — Reference Material – Samples **169**

Sample Deliver-to-Email Request **170**

Sample Aggregate Request **171**
Sample Aggregate Request 171

Header Page Template Example **173**

Interactive, Batch, and Embedded XML Data **174**
Interactive Mode 174
Batch 174
Embedded Data 175

Appendix B — Glossary **177**

Appendix C — SDK - Encrypted Credentials **185**

Passing Credentials Securely **186**
The Java Cryptography Extension 186

Encrypted Credentials **187**

Encrypted Data **188**
Encryption Method 188
key-info Parameter 188
Cipher-Data Parameter 188

Security Keys **189**

Example of a Credentials Set **190**

Index **191**

Chapter 1

Overview

Welcome to the IStream Publisher Interface Reference Guide.

This guide helps developers and system integrators integrate IStream Publisher into their systems.

This chapter describes:

- *Document Conventions* on page 10
- *IStream Publisher* on page 11
- *IStream Publisher Documentation* on page 13
- *Contacting Skywire Software for Help* on page 14

Note: For information about IStream Publisher's architecture, see *IStream Publisher Architecture* on page 15 in the *IStream Publisher Administrator's Reference Guide*.

Document Conventions

Tips, Notes, Important Notes and Warnings

Tip: A **Tip** provides a better way to use the software.

Note: A **Note** contains special information and reminders.

Important: An **Important** note contains significant information about the use and understanding of the software.

Warning: A **Warning** contains critical information that if ignored, may cause errors or result in the loss of information.

Other Document Conventions

- Microsoft Window names, buttons, tabs and other screen elements are in bold, for example: Click **Next**.
- paths, URLs and code samples are in the Courier font, for example: `C:\Windows`
- values that you need to enter or specify are indicated in the italicized Courier font, for example, *`server name`*

IStream Publisher

IStream Publisher is an enterprise document automation software solution that complements core business systems for product development, sales and marketing, administration and customer service.

IStream Publisher provides a set of integrated services that have been specifically engineered to automate document-intensive business processes. It is used to satisfy event-driven requests, such as new policy fulfillment requests, which trigger a range of activities. These activities can include:

- automatically retrieving documents from multiple sources such as file systems, the web, or IStream Document Manager
- assembling personalized, complex documents and document packages such as policies, letters, contracts and booklets
- rendering them in multiple file formats including: DOC (Microsoft Word), HTML, PDF, PCL, PS (Postscript), RTF, TIFF and TXT; AFP and XML files can be rendered using the Transform service
- delivering them to multiple recipients through their preferred channels (print and mail, fax, email, the web)
- saving them to a file system, FTP or IStream Document Manager

IStream Publisher is available in two editions: IStream Publisher and IStream Publisher *Express*.

IStream Publisher *Expres*s provides a subset of the core Publisher functionality and provides the ability to easily add advanced capabilities as business needs dictate.

IStream Publisher is a J2EE application that uses JMS technology and an XML request-based interface to automate the entire document issuance process from content creation to delivery.IStream Publisher is a J2EE application that uses JMS technology and an XML request-based interface to automate the entire document issuance process from content creation to delivery.

Queues and Requests

You access all Publisher services through queues using either:

- **IBM WebSphere MQ**: IBM's WebSphere messaging platform
- **OpenJMS**: an open-source messaging platform

The Service Requests are delivered to the various components using JMS text messages (with the body in XML format).

This guide assumes that you have a basic understanding of:

- **IBM WebSphere MQ or OpenJMS**, including the basic messaging functions, and how to set up and manage queues
- the **JMS Message Service**, including architecture and messaging
- **XML**, including an understanding of its structure and styles

Note: Throughout the guide, there are references to **Calligo,** which is the previous name for IStream Document Manager. *Calligo documents* are now called *IStream documents*.

IStream Publisher Documentation

IStream Publisher includes the following documents and online help files. If you need a copy of any of these documents, please contact your system or product administrator.

- The *IStream Publisher Release Notes* include general product information, product enhancements and new features, supported platforms and third-party software, assorted considerations, and known issues and limitations.
- The *IStream Publisher Administrator's Reference Guide* helps system administrators configure, control, and manage operations and requests.
- The *IStream Publisher Interface Reference Guide* allows you to integrate IStream Publisher within your own systems. It includes the Software Developers Kit (SDK), which allows you to extend IStream Publisher, control its operation, and automate requests.
- The *IStream Publisher Schema* is a set of HTML files that describe the structure of Publisher's services and requests.
- The *IStream Publisher Error Messages* contains a list of error messages and their causes.

Contacting Skywire Software for Help

Customer Support hours are 8:00 A.M. to 8:00 P.M. (Eastern Time), Monday through Friday. Outside of these hours, send us a detailed e-mail message and you will be contacted during regular business hours. Please provide detailed information, as described in the *Support Checklist*.

Contact Information

Mail:	Customer Support Skywire Software 19 Allstate Parkway, Suite 400 Markham, Ontario, L3R 5A4
Phone:	1-905-513-7466
Fax:	1-905-513-1684
Email:	directsupport@skywiresoftware.com
Web:	www.skywiresoftware.com

Support Checklist

When contacting Skywire Software Customer Support, please provide the following information:

- Your name, company name, e-mail address, and phone number
- Version numbers of all your Skywire Software products
- Name and version of the network software
- Windows version, including any installed Service Packs
- Microsoft .NET Framework version
- DMS version, including any installed Service Packs (if applicable)
- Microsoft Word version (if applicable)
- Database vendor and version (if applicable)
- Error messages and the circumstances of their occurrence
- A full description of the problem:
  - What happened? What were the sequence of events that preceded the problem?
  - In which screen or window did the problem occur?
  - Was the problem the result of pressing a key?
  - Did the screen freeze? What functions of the software are affected?
  - How many people are affected?

Chapter 2

Simple Services

This chapter describes the IStream Publisher functional requests that are used to process IStream Publisher services. It provides the information necessary for a developer or system administrator to use IStream Publisher in a relatively simple way to manage documents.

Aggregate Request on page 47 and *Distribution Service* on page 51 explain how to produce the same final result, with less effort on the part of the user.

This chapter describes:

- *Parameters and XML Schema* on page 16
- *Referencing Files* on page 18
- *JMS Message Header and Properties* on page 20
- *Detailed Response Parameters* on page 22
- *Content Service* on page 23
- *Rendering Services* on page 28
- *Delivery Service* on page 36
- *Utility Services* on page 41
- *Aggregate Request* on page 47
- *The Transform Service* on page 49

Parameters and XML Schema

XML Schema is used to construct and validate XML requests for IStream Publisher.

The main parameters are listed for all services. For related parameters, see the schema files, which are located in:

`[`*`IStream Publisher install folder`*`]\dtd\`

The Functional Request schema describes IStream Publisher's Service Requests and responses.

Important: The functional requests are always validated against the internal copy of the XML schema. Any changes made to the schema located in the above folder does not affect the validation of the functional requests.

The syntax for the administrative commands is defined in the DTDs. The syntax for system requests is defined in the XML schema.

The following sections describe each DTD and their XML request elements. (You can use an XML editor to access these main elements using the referenced DTDs and schemas.)

Please note:

- For complete details of all requests and their parameters, refer to the *IStream Publisher Schema*.
- *Calligo documents* are now called *IStream documents*.

Distributor.xsd

This schema describes all service requests and responses including how to generate, render and deliver documents using a single or aggregate request.

You can validate requests against this schema to create Simple, Aggregate and Distribution Requests.

Simple Service XML Request Elements

Content

```
<generate-calligo-document></generate-calligo-document>
```

Render

```
<render-CLG-to-Word></render-CLG-to-Word>
<render-PDF-to-PCL></render-PDF-to-PCL>
<render-PDF-to-PS></render-PDF-to-PS>
<render-PDF-to-TIFF></render-PDF-to-TIFF>
<render-TIFF-to-PCL></render-TIFF-to-PCL>
<render-TIFF-to-PS></render-TIFF-to-PS>
<render-TIFF-to-PDF></render-TIFF-to-PDF>
<render-Word-to-HTML></render-Word-to-HTML>
<render-Word-to-PCL></render-Word-to-PCL>
```

```
<render-Word-to-PDF></render-Word-to-PDF>
<render-Word-to-PS></render-Word-to-PS>
<render-Word-to-TIFF></render-Word-to-TIFF>
<render-Word-to-TXT></render-Word-to-TXT>
<transform-request></transform-request>
```

Delivery

```
<deliver-to-email></deliver-to-email>
<deliver-to-fax></deliver-to-fax>
<deliver-to-printer> </deliver-to-printer>
<deliver-to-repository></deliver-to-repository>
```

Utility

```
<concatenate-pcl-files></concatenate-pcl-files>
<concatenate-pdf></concatenate-pdf>
<concatenate-ps-files></concatenate-ps-files>
<delete-file></delete-file>
<encrypt-pdf></encrypt-pdf>
<run-Word-macro></run-Word-macro>
```

Aggregate Service XML Request Element

```
<request-aggregate></request-aggregate>
```

Distribution Service

```
<distribution-request></distribution-request>
```

Validating Requests

All functional requests are always validated against XML schema.

Validating requests helps minimize user errors. The XML Parser will provide errors back to the client reducing the chance of failed requests.

Referencing Files

Using IStream Publisher, you reference files using a variety of different protocols. Note the following information when referencing files in IStream Publisher:

1. When references to files are passed as arguments to a service invocation, they are expressed in the form of a UISR, an FTP or a File URL.

- a UISR is used to access a source file if the source file is a model document (.cms) or generated document (.clg)
- an FTP URL is used to access a source file if the file is on an FTP server
- a File URL is used to access a source file in a File System

2. The following methods can be used to refer to files:

- **FTP URL** – the file must reside on the FTP server (Windows, Unix)
- **File URL** – the file must reside on the file system (Windows, Unix)
- **UISR** – the file must reside on a Windows File System or in an IStream DMS
- **URL** – the file must reside in a IStream DMS. The syntax for the URL is:

  `calligo://`*`user name:password@database name;server name:port/path`*

3. If you are accessing databases for content generation, you can use the following:

- ODBC InfoSources: see *InfoSource* on page 179
- XML InfoSources
- a UNC path to the XML file containing the data for generation
- a URL to the XML file containing the data for generation

4. When using the Repository API to access file(s) at specified FTP or File URLs, a Repository Adapter for the specific repository must be available on the Worker Machine where the service runs.

5. Some services that reference files using UISRs use InfoSources. The InfoSources referenced by the UISRs passed with the service invocation must exist and can be configured as either local or remote.

6. Examples of file reference syntax that is supported by all aspects of IStream Publisher:

- Local drives:

  `file:///C:/rest_of_file_reference`

  Make sure that all the slashes are forward slashes. Also remember to include the third slash, as in "///". **Do not** use:

  - `file:C:\`
  - `file://C:/`

Important: Mapped drives are not supported by IStream Publisher. Please use UNC paths instead.

- Network UNC shares:

  `file://server/share/`

  Make sure that all the slashes are forward slashes. Also remember to include the second slash, as in “//”. **Do not** use:

  - `file://\\server\share`
  - `file:///server/share`

- FTP:

  `ftp://server/dir/`

Note: Please refer to your IStream Document Manager documentation for more information about UISRs.

The Repository Adapters available are the FTP Server Repository Adapter, the File System Repository Adapter and the DMS Repository Adapter.

7. Element credentials may be defined in multiple places. Credentials also may be defined in the URL itself. If credentials are defined *both* in the URL and in the credentials element, then credentials defined in the element take precedence (unless empty strings are defined there, in which case non-empty credentials from the URL if any exist will be taken). These points are described in more detail in *Using Referenced and Embedded XML Data* on page 24.

Delivering CLG Files through InfoSources

The following table shows how InfoSources are used to deliver specified source content (CLG’s) to a File System or DMS.

InfoSource Types	Example	Description

FileSystem
(local)
	`UISR="`*`[FileSystem InfoSource]`*`:document.clg"`
	`FileSystem_INFOSOURCE =`
	`C:\PublisherFS\`*`[Destination folder]`*`\`
		For delivering the CLG to the specified location through the FileSystem InfoSource

CalligoDMS
(remote)
-or-
IStreamDM
(remote)
See description for use.
	`UISR="DMS_INFOSOURCE:document.clg"`
	`DMS_INFOSOURCE =`
	`[server]``://`
	`Admin:livelink@calligo;`*`[server]`*`:2099/test/`
	`[Destination folder]``/`
		For delivering the CLG to the specified location through the IStream DMS InfoSource
		Note:
		• **CalligoDMS** is the type used in Calligo 5.x.
		• **IStreamDM** is the type used in IStream Document Manager 6.x.

JMS Message Header and Properties

JMS header and property fields can be used to route messages or to carry user-specific metadata.

The following table represents the fields in the request header that have prescribed names. They are copied as request metadata into the request status update message, along with other custom metadata:

Parameter	Description
JMSType	The type of request – Request.JMSType
JMSMessageID	A string ID that uniquely identifies the message in the systems (set automatically by the Queue Provider).
JMSCorrelationID	Used only for response messages. Contains the ID of the request to which the response message is linked.
JMSReplyTo	A destination-object (JMS) indicating the queue where the response to the message should be submitted. If null, no response is required. The queue specified in the **JMSReplyTo** field must reside on the same IBM WebSphere MQ Queue Manager or OpenJMS Server as the queue used to submit the request.
JMSPriority	Specifies the priority of the message (a number from 0 to 9 with 0 being the lowest priority and 9 the highest). When not specified, the default value assumed is 4.
InternalRequest-ID	This value is assigned after successful submission.
RequestID	This value is assigned by the Requestor (client).
AggregateID	This ID is provided by the client and inherited by all subrequests.
ParentID	This value is assigned by the Requestor when a request is resubmitted.
OriginalRequest-ID	This ID is provided by IStream Publisher for resubmitted requests.
RequestType	The Request Type of the original Request.
LogLevel	The level of logging that is performed by the system, ranging from 0 to 6. The default is “4”.

Parameter	Description
DeferralTime	**fixed time – Dmm/dd/yyyy** *or* **Thh:mm**, where: • **D** – mandatory letter for fixed date • **mm/dd/yyyy** – date format; • **T** – mandatory letter for fixed time; • **hh:mm** – time format, where hh is in 24 hours format; • **D/05** – a request should be processed on 5th day of current month, year • **D1//2003**- a request should be processed on January 2003. • **D01/02/2002** – a request should be processed on 2nd January, 2002; • **D/15/** – a request should be processed on 15th day of current month, year • **D//2003**- a request should be processed in 2003 year. **timer** – +**Dmm/dd** *or* **Thh:mm**, where: • **D** – mandatory letter for date parameters in timer • **mm/dd**- number of months, days; • **T** – mandatory letter for time parameters; • **hh:mm** – time format, where hh is in 24 hours format; • +**D/05** – a request should be processed in 5 calendar days • +**D01/**- a request should be processed in 1 month. • +**D01/02** – a request should be processed in 1 month and 2 calendar days.
ExpirationTime	**fixed time – Dmm/dd/yyyy** *or* **Thh:mm**, where: • **D** – mandatory letter for fixed date • **mm/dd/yyyy** – date format; • **T** – mandatory letter for fixed time; • **hh:mm** – time format, where hh is in 24 hours format; **timer** – **date/time** – +**Dmm/dd** *or* **Thh:mm**, where: • **D** – mandatory letter for date parameters in timer • **mm/dd**- number of months, days; • **T** – mandatory letter for time parameters; • **hh:mm** – time format, where hh is in 24 hours format;

Request Metadata

The request metadata representing the information varies from deployment to deployment, and consists of request header fields. IStream Publisher components store the information about requests in the Request Log database.

The Request Log configuration maps Custom Request metadata fields to the Request Log table fields. If a mapping does not exist for one of the custom metadata fields, that field is not stored in the Request Log table.

The request metadata can be used when performing request searches. It can also be referred to in the selector parameter of the `FindRequest` administrative command.

Detailed Response Parameters

For simple requests, a response is returned when the request is run. A successful completion response simply echoes the destination URL. A failure response will contain one or more of the following items:

- an error ID
- an English text message explaining the reason for the failure
- an XML fragment with extended error information

Content Service

Documents can be generated using IStream Publisher with the Generate IStream Document service, a subset of the Content service. The Generate IStream Document service can produce both (.clg) and Microsoft Word documents (.doc).

The Content Service provides access to content generated by the Assembly Service.

Important: You can use InfoSources to access the content only if:

- The InfoSource exists and is configured on the Worker machine running the service.
- The source or target file is an IStream document (.clg), model document (.cms), or model section (.cds).

Generate Calligo Document Service Overview

Use the *Generate Calligo Document Service* to generate an IStream or a Microsoft Word document based on a model document.

Note: *Calligo* is the former name for *IStream Document Manager*. Therefore, *Calligo documents* are now called *IStream documents*.

IStream InfoSources are specific to the IStream Assembly Engine and are used to reference generated IStream documents or model documents. InfoSources are COM (see page 178) objects and are, therefore, platform dependent (WI32).

A *Generate Calligo Document Service* request can produce both an IStream document (.clg) and/or a Microsoft Word document (.doc). The document that is produced is determined by the destination parameters in the *Generate Calligo Document Requests* parameters table.

Important: At least one destination parameter must be specified, otherwise the document will not be produced.

IStream XML InfoSource

When generating documents using the IStream XML InfoSource (formerly called the *Calligo Extreme XML Infosource*), data for generation can be:

- passed by key data: one or more local or UNC paths to the XML data files containing the generation data can be passed as key data
- embedded in the request.
- referenced by a URL to a XML data file

The `key-data` tag represents definitions for key data elements that the model document expects. The parameters are used to let the document generation service know what information to put into the key data and in what format.

In case the Service Request fails, the body of the original request as well as the error details can be found in the Request Log.

A Generation Log produced by the Assembly Engine is provided whether the request completes successfully or not.

Important: The Generate IStream document service uses the Repository API to store the Word rendition of the generated document.

Using Referenced and Embedded XML Data

Note: The elements <xml-data> and <xml-data-def> are available in E-Delivery 2.1 (E-Delivery is the former product name of IStream Publisher), and are supported in this version for backward compatibility. New applications should use the new <generation-data> element.

Sample XML Fragments

Here are some XML fragments followed by an explanation of some of the tags used.

These XML fragments are two examples of generation data embedded in the generate IStream document Requests:

Example 1

```
<generation-data name="xml_file_name">
<job-data>&lt;?xml version=&quot;1.0&quot;?&gt;
   … embedded XML data fragment …
</job-data>
</generation-data>
```

Example 2

```
<generation-data name="xml_file_name">
<source url="ftp://host1/data/QPolicy.xml"/>
</generation-data>
```

As you can see, in example 1 the data is embedded and in example 2 it is referenced. The tags `<source>` and `<job-data>` are mutually exclusive.

Using Referenced XML Data

When using referenced data, you must use case sensitive XML data. Your XML InfoSource (used in your model document) must be marked as **case sensitive** in the *InfoSource Administrator*, and all data tags in the XML file and model must be the same, consistent case.

Note: All SQL query statements must be uppercase.

When using Batch Referenced data, you must ensure that the Query statement in the model document uses the following format:

```
QUERY "FILE=" + xml_file_name, "XMLInfoSource"
```

or

```
QUERY "FILE=" + xml_file_name + ";JobID=" + JobID ,
"XMLInfoSource"
```

where `xml_file_name` is the value specified in attribute `"name"` in the `generation-data` element.

Important: Do not use Batch data where the query is only the ID.

Code Samples

The following examples use Interactively Referenced, Embedded, and Batch Referenced XML.

Batch XML as the URL Reference

The JobID is the ID in the XML file.

```
<generate-calligo-document>
  <calligo-source UISR="modelIS:doc.cms" docType="cms">
    <credentials user="test" password="password"/>
  </calligo-source>

  <destination url="ftp://server/folder/document.doc"/>

  <key-data name="JobID" value="12345678" type="string"/>

  <generation-data name="keydataname">
    <source url="ftp://server/folder/xmldatafile.xml"/>
  </generation-data>
</generate-calligo-document>
```

Embedded XML

```
<generate-calligo-document>
  <calligo-source UISR="modelIS:doc.cms" docType="cms">
    <credentials user="test" password="password"/>
  </calligo-source>

  <destination url="ftp://server/folder/document.doc"/>
    <generation-data name="keydataname">

      <job-data>&lt;?xml version=&quot;1.0&quot;?&gt;
       &lt;interactive&gt;
       &lt;PolicyNumber type=&quot;double&quot;&gt;12345678
       &lt;/PolicyNumber&gt;
       &lt;EffectiveDate type=&quot;date&quot;&gt;01/01
       /2002
       &lt;/EffectiveDate&gt;
       &lt;CompanyCode type=&quot;string&quot;&gt;ABCT01
       &lt;/CompanyCode&gt;
       &lt;DataArray&gt;
       &lt;Id type=&quot;array&quot;&gt;
       &lt;row type=&quot;string&quot;&gt;111&lt;/row&gt;
       &lt;row type=&quot;string&quot;&gt;222&lt;/row&gt;
       &lt;row type=&quot;string&quot;&gt;333&lt;/row&gt;
```

```
        &lt;row type=&quot;string&quot;&gt;444&lt;/row&gt;
        &lt;row type=&quot;string&quot;&gt;555&lt;/row&gt;
        &lt;row type=&quot;string&quot;&gt;666&lt;/row&gt;
        &lt;/Id&gt;
        &lt;/DataArray&gt;
        &lt;/interactive&gt;
    </plain-data>
  </generation-data>
</generate-calligo-document>
```

Embedded XML with Plain Data

```
<generate-calligo-document>
  <calligo-source UISR="modelIS:doc.cms" docType="cms">
    <credentials user="test" password="password"/>
  </calligo-source>

  <destination url="ftp://server/folder/document.doc"/>

  <generation-data name="keydataname">
    <plain-data>
      <interactive>
        <PolicyNumber type="double">12345678</PolicyNumber>
        <EffectiveDate type="date">01/01/2002
         </EffectiveDate>
        <CompanyCode type="string">ABCT01</CompanyCode>
          <DataArray>
            <Id type="array">
              <row type="string">111</row>
              <row type="string">222</row>
              <row type="string">333</row>
              <row type="string">444</row>
              <row type="string">555</row>
              <row type="string">666</row>
            </Id>
          </DataArray>
        </interactive>
      </plain-data>
    </generation-data>
</generate-calligo-document>
```

Generate IStream Document XML Sample

The following code is an example of a Content Service Request that generates an IStream document.

```
<?xml version="1.0" encoding="UTF-8"?>
  <generate-calligo-document>
    <calligo-source UISR="ABC:Test_Letter.CMS"/>
    <destination url="ftp://abcserve/Test_Letter.clg"/>
    <calligo-destination
      UISR="Demo_Dest:contentCMS2CLG/Test_Letter.clg"/>
```

```
    <key-data
      name="$Policy"
       value="test_letter"
      type="string" />
</generate-calligo-document>
```

Rendering Services

After content has been extracted, the next step is to transform the content from its current format into a different format.

The Rendering Service is a group of services that transform content from its current format into a different format. Each service invocation renders from one source format into one destination format (for example, from Microsoft Word to PDF). Typically, a render request specifies the location (URL) of the source content and the location (URL) where the result of the render operation should be placed.

In addition to the source and destination, other arguments are specific to each particular rendering service. For example, rendering to PCL will include parameters specific to a print job such as duplex mode or page range. It is therefore important to have this type of information before you create your Rendering Service request.

Important: If a Page Range parameter is wrong, referring to pages that do not exist, an error is produced and the rendering request fails.

Updating a Table of Contents in Word

To update a table of contents in a Word document

Before rendering, printing or distributing the document, use the following standard Word macros in the simple rendering request or in the Distribution Item of the Distribution Request:

```
<word-options updateToc="true"/>
```

Rendering a Microsoft Word Document to HTML

This service transforms a Microsoft Word document into Hyper Text Markup Language (HTML, version 4.0) format page.

It uses Microsoft Word's application automation, which makes it Win32-platform dependent.

It also uses the Repository API to access all the files specified as parameters to the request.

Related Files

When a document is rendered into an HTML page, all graphics and objects are saved in GIF (.gif), JPEG (.jpg) or PNG (.png) format so that they can be viewed in a Web browser. These graphics and objects include:

- pictures
- AutoShapes
- WordArt
- text boxes

- callouts
- Equation Editor objects
- Organization Chart objects
- Graph objects

Graphics Files

When you render Word-to-HTML and the destination is FS (File System) or FTP, an HTML file and a folder are created. This folder contains all the related graphics.

Graphics might include bullets, backgrounds, and horizontal lines for each document. This folder (or subfolder) is always given the name of the associated HTML page, followed by the word `files`. For example, if the name of the HTML page is `letter.htm`, the graphics for that Web page are in a folder called `letter_files` or `letter.files`. The subfolder also contains a file called `filelist.xml` where all the graphics are listed.

If you move an HTML page to another location, all the related graphics must be moved, otherwise the hyperlinks might not work, and the graphics might not appear on the HTML page.

Important: When you render Word-to-HTML and the destination is DMS, only the HTML file is saved, and not the folder and its images.

Rendering from Word to HTML has some document layout limitations. Because Word provides formatting options that most Web browsers do not support, some text and graphics may look different when you view them on a Web page.

Note: Please note the following information:

- graphics with certain kinds of text wrapping will change position when you save your document as a Web page
- cross-referencing in a Word document cannot be translated to HTML
- hyperlinks inside an embedded OLE object (such as Microsoft Excel) cannot be converted to HTML

The render-Word-to-HTML request renders a Microsoft Word document into an HTML format.

Rendering a Microsoft Word Document to PCL

The render-Word-to-PCL request produces a PCL (Printer Control Language) representation of a Microsoft Word document. When the PCL stream is sent to a printer, it produces a hard copy of the document.

Keep in mind when rendering Microsoft Word documents to PCL that because this rendering service uses Microsoft Word’s application automation, which is only available on the Win32 platform, this service is limited to that platform.

Rendering a Microsoft Word Document to PDF

This service produces a Portable Document Format (PDF) document from a Microsoft Word document. The PDF files can be viewed on multiple platforms using the appropriate reader for that platform.

For this service, you can select to use the Amyuni PDF printer driver or Microsoft Word 2007 as the rendering application.

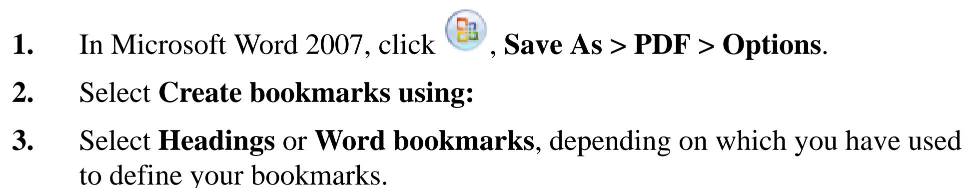
Using Microsoft Word 2007 as the Rendering Application

If you select Microsoft Word 2007 as the rendering application, note that:

- Microsoft Word 2007 must be installed
- you need to download and install the **Microsoft Save As PDF** add-in for Microsoft Office 2007
- the PDF will always include hyperlinks
- page ranges are not supported

To enable bookmarks or embedded fonts in the PDF, complete the following procedures in Microsoft Word 2007

Method: Enable bookmarks in the PDF



1. In Microsoft Word 2007, click , **Save As > PDF > Options**.
2. Select **Create bookmarks using:**
3. Select **Headings** or **Word bookmarks**, depending on which you have used to define your bookmarks.

Note: Only standard Microsoft Word heading styles (*Heading1*, *Heading 2*, *Heading 3*, and so on) from CMS files that have been converted to Word 2007 and configured in IStream Author as PDF bookmark styles will appear as corresponding bookmarks in the resulting PDF.

Non-heading Word styles, for example, *List* or *Note*, that have been configured in Author as PDF bookmark styles will *not* appear as PDF bookmarks. However, if you use the Amyuni PDF printer as the rendering application, then *all* of the selected styles will appear as PDF bookmarks.

Method: Embedding fonts in the PDF

---

1. In Microsoft Word 2007, click , **Save As > PDF > Options**.
2. Select **ISO 19005-1 compliant (PDF/A)**.

   The PDF will be created using the PDF/A standard. This standard ensures that all the fonts will be embedded in the PDF.

Note: The PDF/A standard has certain limitations. For more information, see **www.pdfa.org**.

Rendering a Microsoft Word Document to PostScript

This service renders a Word document to a PS stream which, when sent to the printer, creates a printed copy of the document. It uses Word's application automation and PostScript drivers, which makes it Win32-platform dependent.

Rendering Service XML Sample

```
<?xml version="1.0" encoding="UTF-8"?>
<render-Word-to-PCL>
  <source url="ftp://abcserve/demo/source/doc/ABC_LTC.doc"/>
  <destination
    url="ftp://abcserve/demo/destination/renderDOC2PCL/
    LTC.prn"/>
  <output-name>HPLJ8000</output-name>
  <printer-configuration
    copies="1"
    pageRange="1-3"
    duplex="none"
    collate="off"/>
</render-Word-to-PCL>
```

Rendering a Microsoft Word Document to TIFF

The render-Word-to-TIFF request renders a Microsoft Word document into a TIFF image.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a TIFF (Tagged Image File Format) rendering of a Microsoft Word document.

Note the following information when rendering Microsoft Word to TIFF:

- because this service uses Microsoft Word's application automation, it is Win32-platform dependent
- this service uses the Amyuni PDF Converter printer driver for rendering
- this service supports the CCITT group 4 compression, and multi-page image TIFF features

Rendering a Microsoft Word Document to TXT/RTF

The render-Word-to-TXT request renders a Microsoft Word document into a TXT or RTF file.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a TXT/RTF rendering of a Microsoft Word document.

Note the following information about this service:

- this service uses Microsoft Word's automation feature making it Win32-platform dependent
- you need to specify `text/plain` or `application/rtf` as the content type in order to perform a corresponding TXT or RTF rendering of the Microsoft Word document

Rendering a PDF Document to PCL

The render-PDF-to-PCL request renders a PDF document into a PCL stream.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a PCL (Printer Control Language) representation of a PDF (Portable Document Format) electronic document.

The Render PDF to PCL Service is Win32-platform dependent.

This request effectively creates a stream, which when sent to a printer will produce a hardcopy of the PDF document. Because rendering depends on the actual printer that is used to produce the printout, the request must include parameters specific to the printer (such as the number of copies and pages).

Note: This service supports only unprotected PDF documents. Please ensure that the documents specified in the source are not protected.

Rendering a PDF Document to PS

The render-PDF-to-PS request renders a PDF document into a PS stream.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a PS (Postscript) representation of a PDF (Portable Document Format) electronic document. The Render PDF to PS Service is Win32-platform dependent.

Note: This service supports only unprotected PDF documents. Please ensure that the documents specified in the source are not protected.

Rendering a PDF Document to TIFF

The render-PDF-to-TIFF request renders a PDF document to TIFF.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a TIFF (Tagged Image File Format) rendering of a PDF (Portable Document Format) electronic document. It supports Group-3 and Group-4 TIFF compression formats.

Note: This service supports only unprotected PDF documents. Please ensure that the documents specified in the source are not protected.

This service is Win32-platform dependent.

Rendering a TIFF Image to PCL

The render-TIFF-to-PCL request renders a TIFF image into a PCL stream which when sent to a printer produces a hard copy of the image.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a PCL (Printer Control Language) representation of a TIFF (Tagged Image File Format) image.

It uses a third party imaging product called Snowbound™. Since Snowbound is a pure Java implementation, this service is platform independent and is supported on any Java platform.

Rendering a TIFF Image to PDF

The render-TIFF-to-PDF request renders a TIFF image into a PDF document.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a PDF (Portable Document Format) rendering of a TIFF file.

Rendering a TIFF Image to Postscript

The render-TIFF-to-PS request renders a TIFF image into a PS stream which, when sent to a printer, produces a hard copy of the image.

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service produces a PS (PostScript) representation of a TIFF (Tagged Image File Format) image. It supports Group-3 and Group-4 TIFF compression formats.

Rendering an IStream Document to Microsoft Word

IStream documents are compound documents that contain a Microsoft Word document among other things such as persistent variables and customizations. The render-CLG-to-Word service extracts the Microsoft Word document from an IStream document.

Because the service uses the Assembly Engine, which is only available on the Win32 platform, the service itself is limited to this platform.

Delivery Service

Once the content has been extracted and rendered (if applicable), the final step in the distribution process is to deliver the content to the recipients. The following methods are available to deliver content:

- Repository: includes DMS, file system and FTP repositories
- Printer
- E-mail
- Fax

A request for delivery will typically specify the location of the content to be delivered. This content can be a specific file in any of the supported formats. See *Content Service* on page 23. The other parameters are specific to the actual channel used for delivery.

Delivering Content to a Repository

The Delivery Service uses the Repository API to deliver the source content documents (DOC, HTML, PDF, PCL, PS, RTF, TIFF, and TXT files) to the destination Repository Adapters.

Examples Using the Destination Element

```
<destination url="file:///C:/PublisherFS/Destination/letter.doc"/>

<destination url="calligo://user_name:password@Livelink;SERVERNAME:2099/
TEST/Destination/letter.doc"/>
```

Note: The above URLs are used to deliver specified source content to the destination. You cannot use these URLs to produce a CLG file.

Other adapters may be provided in the future, or custom adapters can be built based on the Repository API specification.

The Repository API will allow a delivery service to deliver content to different types of repositories as long as a Repository Adapter exists for that repository.

If the destination repository supports versions, and the source documents already exist, they are added as new versions. If an existing document cannot be reserved, so that the new version can be added, the operation will fail. All reserve operations are performed using the credentials supplied as parameters.

The deliver-to-repository request stores a document into a repository.

Delivering Content to a Printer

The deliver-to-printer request delivers an individual file to a printer. Note the following information when implementing this service:

- The printer must be a logical device (print server or spooler). This service only ensures that the spooler has accepted the content.
- Currently, the service uses native Windows APIs to submit print jobs to the print spooler and is therefore limited to the Win32 platform.
- The content delivered can be any file depending on the printer's capabilities.
- You can specify the name of the file to be sent to the printer.

Note: The request may fail because of a "physical" issue. Examples of physical issues include if the printer is unavailable, if it is offline, if the user lacks proper security permissions to use the printer, and others.

Delivering Content to an E-mail Server

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

The deliver-to-email request delivers content to an SMTP e-mail server as multi-part MIME messages. Note the following information when implementing this service:

- The content consists of one message body and zero or more attachments. Both the content making up the body of the message and the attachments can be either ASCII text or binary. The message body may also be embedded and may also be HTML.

Please note that:

- Binary content is base64 encoded
- ASCII text is 7bit encoded
- The service only delivers the e-mail (multi-part MIME message) to the SMTP e-mail server. It does not wait for the e-mail to be delivered to the actual recipient(s) or to get delivery confirmation.
- This service is a pure Java implementation that uses the JavaMail API. Therefore, it is platform independent and can be used on any platform that supports Java.
- The Worker machine on which the service is deployed must be configured with the name of SMTP e-mail server.

The following items are attributes of the deliver-to-email request

- Subject – The subject line of the e-mail.
- Priority – The priority of the message (High, Normal, or Low). This field is optional and if not specified, Normal priority is assumed.

Note: To see some sample “deliver-to-email” code, please see *Sample Deliver-to-Email Request* on page 170.

Delivering Content to a Fax Server

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

The deliver-to-fax request delivers TIFF (raster) content to the RightFax server from an SMTP e-mail server. The RightFax server automatically picks up e-mail messages addressed to it from the SMTP server.

Note the following information when implementing this service:

- The content can consist of one or many attachments.
- The service only delivers the e-mail (multipart MIME) message to the SMTP e-mail server. It does not wait for the e-mail to be delivered either to RightFax, or to the actual recipient(s).
- The service gets delivery confirmation from the SMTP e-mail server.
- The Worker machine on which the service is deployed must be configured with the name of the SMTP e-mail server.
- The delivery service invocation includes the URLs of the contents to be delivered and their MIME type.
- Embedded codes are special faxing instructions that can be passed through a request and inserted by the Service Manager directly into a fax-bound e-mail message body.
- Embedded codes can be used to specify:
  - A time to send the fax,
  - A time to delete the fax from the **FaxUtil** mailbox after it has been successfully sent, and
  - Other additional instructions.
- IStream Publisher supports embedded codes that are also supported by RightFax. The embedded codes must be specified in the Service Manager's configuration file.

Note: IStream Publisher allows you to deliver any file formats supported by RightFax. However, IStream Publisher does not guarantee the target layout if rendering has been done by RightFax.

Delivery Service Request XML Sample

The following code is an example of a Deliver to e-mail request:

```
<?xml version="1.0" encoding="UTF-8"?>
<deliver-to-email
 subject="rd0026_0034"
priority="normal">
<body-source>
```

Body Source Information

```
 <source
 url="ftp://anonymous:user@abcserve/source/e-mailmessage/e-
mail.txt"/>
</body-source>
<sender
     name="1st submitter"
     emailAddress="submitter1@abc.com" />
 <receiver
     name="1st Receiver"
     emailAddress="receiver1@abc.com
     type="to" />
  <receiver
     name="1st CC"
     emailAddress="CC1@abc.com
     type="cc" />
  <receiver
     name="1st BCC"
     emailAddress="bcc1@abc.com
     type="bcc" />
  <attachment>
```

Attachment Information

```
  <source
url="ftp://anonymous:user@abcserve/source/docs/sdoc.doc" />
  </attachment>
 </deliver-to-email>
```

Utility Services

IStream Publisher provides several Simple Services. Typically these services run after the rest of a request has successfully completed. These services are:

- *Run Word Macro* on page 41
- *Concatenating PCL Streams* on page 41
- *Concatenating PDF Files* on page 42
- *Concatenating PS Streams* on page 43
- *Encrypting PDF Documents* on page 44
- *Deleting Files and Folders* on page 44

Run Word Macro

The `run-Word-macro` service runs a macro on a Word document or CLG file and saves the resulting file.

The `run-Word-macro-response` contains the URL of the resulting file, plus the return status of the request and errorDetails if the request failed.

As indicated in this table, the **source** and **destination** parameters are for Word files only, and the **clg-source** parameter is for CLG files only. If you use a clg-source, the macro is applied to the Word document inside the CLG and then saved back to the same CLG location.

Concatenating PCL Streams

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

Some documents need to be printed as a single job to avoid pages from other documents being intermixed. This service provides the ability to concatenate the PCL streams for these documents in the correct order.

Alternatively, the service concatenates PCL streams to some specified number, N, of approximately equal-size PCL stream segments for delivery to a print server with load balancing across multiple (N) printers. The order of the concatenated documents remains unchanged.

The Number of streams is specified in the request. The default number of streams is "1".

Streams Header Page

Each subgroup must have an appropriate header page defined so that, after delivery to a print server with load balancing across multiple (N) printers, a person can reassemble the documents back into the correct sequence for delivery.

The Header Page can be composed using a Header Page Template. The template contains some special placeholders.

The service uses this template and substitutes the placeholders with actual values provided with the request. The place where content must be substituted, is marked by "{<name>}" – where <name> is the name of the value to be used.

If a name provided in the request does not match a placeholder in the template, the Header Page will not include the information.

Template URL

The URL to the template should be specified in a config file for the service. If there is no template associated with a Header Page, then a Header Page as plain text will be composed "on the fly" and every field will be printed in a separate row, left-aligned.

The template can be in plain text or in PCL format. You can create fancy Header Page templates with placeholders using Microsoft Word and Render to PCL format. See *Header Page Template Example* on page 173.

Optional Parameters

The following parameters can optionally be provided for a Header Page:

- paper size
- paper orientation
- tray/paper source

Each header page will be concatenated, with the appropriate PCL segment on top.

Duplexing Options

In the concatenation process, the duplexing of the pages can either be continuous (meaning that the next concatenated stream can start either on an even or an odd page) or, it can break at odd-numbered pages. If the Print Instruction parameter for duplexing is not specified in the request, the service will assume a break at odd-numbered pages.

Concatenating PDF Files

Some documents need to be printed as a single job to avoid pages from other documents being mixed in with them. This service provides the ability to concatenate the PDF documents in the correct order. This service is supported on any Java platform.

Please note:

- This service supports only unprotected PDF documents. Please ensure that the documents specified in the source are not protected.
- A standard concatenation event handler requires the `absolute-path` element within the `preference-repository` specified and cannot include a destination element.

Concatenating PS Streams

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

PS Concatenation Methods

IStream Publisher provides PS Concatenation in two ways, as a Utility Service and as an Event Handler in a Distribution Request. In both cases, two or more source PS streams are concatenated in a certain order into one or more destination PS streams. The actual printing of the destination PS streams can then be sent as one or more print jobs to one or multiple printers.

Each destination PS stream is printed as a single print job to one printer. PS Concatenation is useful in eliminating the possibility of other documents being mixed in with a set of documents intended to be printed as one package. It is also useful for printing a large package of documents with load balancing across multiple printers.

Note: PS concatenation in the event handler may not be continuous depending on which rendering printer or physical printer is used. You will need to confirm the behavior with your selected printer driver and printer.

PS Concatenation Variations

IStream Publisher supports these variations of PS Concatenation:

- where two or more source PS streams are concatenated into one destination PS stream
- where two or more source PS streams are concatenated into a given number of destination PS streams with a designation on which source PS stream should go to which destination PS stream
- where two or more source PS streams are concatenated into a given number of destination PS streams of approximately equal size to achieve load balancing across multiple printers

Header Pages and Templates

When concatenating to produce destination PS streams of approximately equal size, a Header Page can optionally be printed at the beginning of each destination PS stream to assist in reassembling the streams printed across multiple printers back into the correct sequence. A template may be used for the generation of the Header Page. The template must contain placeholders, corresponding to field entries specified in the request, where actual values of information relating to that PS stream can be substituted. The template must be in PS format.

The URL to the Header Page Template must be specified in the configuration properties for PS Concatenation. If no template is specified, the Header Page will be composed as plain text on the fly, with each of the field entries specified in the request printed on a separate row aligned to the left.

Encrypting PDF Documents

Note: You must have the appropriate Publisher license to run this service. Please check your software license or contact Customer Support to determine if you are licensed to run this service.

This service protects and encrypts PDF documents according to specified parameters. This service is supported on any Java platform.

Note: This service supports only unprotected PDF documents. Please ensure that the documents specified in the source are not protected.

Specifying Encryption Flags

If an `encrypt-pdf` request does not specify encryption flags explicitly, IStream Publisher assigns the highest level of security to the document's properties, and generates and assigns a unique encrypted master password to the document automatically so that its security level cannot be changed.

If the document's security settings need to be modified, the request must explicitly assign a master password to the document in the request which can then be used to change the security settings.

Deleting Files and Folders

You can use the Delete Files simple service to delete files and folders.

Source Parameter

The `source` parameter is the URL of the files or folders that you want to delete.

When you specify a `source`, IStream Publisher tries to delete the folder and its contents. In File System and IStream DM type-repositories, the contents of the folder are deleted regardless of whether the folder is empty. In an FTP repository, a folder that is not empty cannot be deleted.

Example: Deleting a folder from an IStream DM Repository

In this example, the **`documents`** folder and its contents are deleted:

```
<delete-file>
  <source url="calligo://admin:password@livelink;
   dmsserver:2099/documents"/>
</delete-file>
```

Example: Deleting a folder from an FTP Repository

In this example, the **`documents`** folder and its contents are *not* deleted if `documents` folder contains files or folders:

```
<delete-file>
    <source url="ftp://ftpserver/documents"/>
</delete-file>
```

Folder Sources

When you specify a `<folder-source>`, the contents of the folder will be deleted recursively, but not the folder itself.

Example

In the following example, everything under the **`documents`** folder will be deleted, but the **`documents`** folder itself will *not* be deleted.

```
<delete-file>
   <folder-source>
      <source url="calligo://Admin:password@livelink;
       dmsserver:2099/documents"/>
   </folder-source>
</delete-file>
```

Wildcards

The following wildcard characters are supported for deleting files and folders:

- the question mark (?), which represents any single character
- the asterisk (*) which represents one or more characters

Combining <source> and <folder-source>

You can use `<source>` and `<folder-source>` in any combination in the `<delete-file>` request.

The following example is valid:

Example: Source & Folder Source

```
<delete-file>
   <source url="calligo://Admin:password@livelink;dmsserver:2099/
    Test Docs"/>
      <folder-source>
         <source url="calligo://Admin:password@livelink;dmsserver:2099/
          Test Docs"/>
      </folder-source>
   <source url="file://somefolder/tempdocs"/>
      <folder-source>
         <source url="ftp://ftpserver/lastyear/worddocs"/>
      </folder-source>
</delete-file>
```

Credentials

You can optionally provide credentials if the source requires authentication. They can be specified either as a separate parameter or encoded in the URL.

The Service Response is the return URL of any successfully deleted files. There are two possible values: completed-successful or completed-failure.

A failure response will contain any or all of the following items:

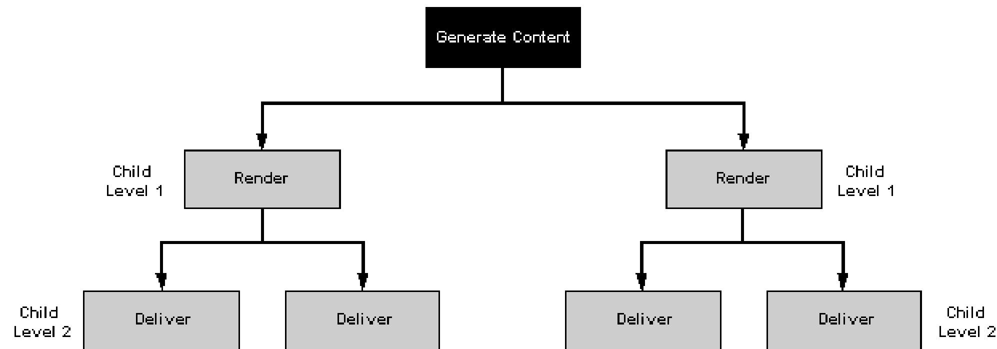
- an error ID
- an English text message explaining the reason for the failure
- an XML fragment with extended error information

Aggregate Request

If you want Simple Requests to be executed in a specific order, such as rendering before delivering, then you must submit an Aggregate Request. Otherwise, when you send multiple simple requests such as generate, render and deliver, there is no guarantee that they will be executed in the correct order.

Aggregate Request Processing

The processing order for an Aggregate Service Request is described as a dependency tree in which subrequests are processed only after the main “top” request has completed successfully.



Dependencies of the Aggregate Service Request

The figure above shows the dependencies of an Aggregate Service Request. The processing order is as follows:

1. The Render request (Level 1) is only processed if the Generate Content request (Main) is processed successfully.
2. The Deliver request (Level 2) is only processed if the Render request (Level 1) is processed successfully.

Aggregate Request Limitations

In order to maintain forward compatibility with your IStream Publisher Client there are some limitations for the kinds of requests that can be aggregated

1. You can specify only one document per request in the root.
2. The dependencies tree may have only one root node. It can, however, have one or more levels. Sub-requests at each level can be of different types (Generate, Render, Deliver, or Utility).
3. An Aggregate Request cannot handle situations where parallel executions must come back together in a synchronized way to perform a common action or request.
4. An Aggregate Request does not:
   - Support order processing of sub-requests belonging to the same level of the dependencies tree.

- Provide a cleanup of intermediate temporary files created as a result of processing Aggregate Request. Note that Utility Services are designed to provide cleanup and to delete temporary files.
- Provide consolidation of sub-request responses.

An Aggregate Request is considered complete as soon as the root process is complete and the sub-processes have been submitted.

The Transform Service

The Transform service is a simple service that uses a command interface to allow Transform Suite applications to be called from a request.(The Transform service is installed separately from IStream Publisher.)

For details of the Transform service parameters, see the IStream Publisher schema.

For details of the parameters that you can use to customize the rendering, see the assorted Transform Suite documentation.

Sample Request

Simple Request Example

The following code is an example of a simple request that calls the Transform service:

```
<transform-request templateName="PS-TO-AFP">

   <source url="file:///c:/Sourcefiles/Test.ps"
ContentType="application/postscript">
   </source>

   <destination url="file:///DestinationFiles/Test.afp"
ContentType="application/afp" copyMetadata="true">
   </destination>

   <parameters>
      <parameter name="papersize">letter</parameter>
      <parameter name="pagecount">5</parameter>
      <parameter name="pagestart">3</parameter>

   </parameters>

</transform-request>
```

Distribution Request Example

The following code is an example of a segment of distribution request that calls the Transform service:

```
<recipient-packages>
   <recipient-package id="RepId">
      <recipient-item refID="item2">
      <render-param>
         <transform templateName="PS-TO-PS">
            <parameters>
               <parameter name="configurationName">C:\Publisher
Transform Config Files\TransformPS\psdemo_Publisher.ini</parameter>
               <parameter name="papersize">legal</parameter>
               <parameter name="pagecount">5</parameter>
            </parameters>
```

```
         </transform>
      </render-param>

      </recipient-item>
   <delivery-preference refID="R-PID"/>
</recipient-package>
```

General Considerations

To invoke Transform through an IStream Publisher request:

- Transform 3.0 must be installed and working
- the value assigned to the `TransformHome` system variable must be the Transform installation folder: by default, this folder is:

  ```
  C:\Program Files\Whitehill Technologies\Whitehill
  Transform 3.0
  ```

A distribution item is not checked if the Transform service is invoked in a distribution request. As a result, some incorrect file types may be passed to the Transform service. You therefore need to ensure the source files are the correct type for the transform service being invoked from IStream Publisher.

Postscript File with PJL Commands

If you are rendering Postscript files that contain PJL (Printer Job Language) commands, you will need edit your Transform `system.ini` file:

1. Locate the Transform `system.ini` file. By default, this file is located in

  ```
  C:\Program Files\Whitehill Technologies\Whitehill
  Transform 3.0\config\
  ```

2. Add the following line to this file:

  ```
  PJLSUPPORT=IGNORE
  ```

Chapter 3

Distribution Service

This chapter describes:

- *The Distribution Service* on page 52
- *The Distribution Request* on page 53
- *The Distribution Package* on page 59
- *Recipients* on page 62
- *Delivery Channels* on page 67
- *Event Handlers* on page 70
- *Calling the Transform Service* on page 79
- *A Distribution Request Example* on page 80

Note: The main parameters are listed for all services. For related parameters, please refer to the schema.

The Distribution Service

The Distribution Service is a composite service, designed to provide user-friendly document distribution service. This service improves the document distribution process by allowing requests to be more oriented to the business user.

The business user does not need to know as much about all the steps of the distribution process including the intermediate content that gets created. Instead, the user can focus on the business requirement. The model that the Distribution Service uses is that of “a Distribution Package delivered to multiple recipients through various Delivery Channels”, in one request.

Example

A marketing document needs to be delivered by fax to a group of customers. A different version of that document needs to be sent by e-mail to 2,000 managers and a third version needs to be saved on the Web site for use by sales agents. All of these requirements can now be handled by one request using the IStream Publisher Distribution Service.

For an overview of the Distribution Service functionality, see *The Distribution Request Lifecycle* on page 28 of the *Administrator’s Reference Guide*.

The Distribution Request

The Distribution Service is invoked through a Distribution Request. The request has a declarative structure. The Distribution Request describes the documents to be distributed, their recipients, the delivery channels to use and the Event Handlers to apply.

Warning: The distribution request should not contain more than 1,000 delivery items, including items within folder distributions. This is because large Distribution Requests may take a long time to be processed. This can trigger an IBM WebSphere MQ server roll-back on the request itself. This will not cause the request to fail, but may result in the request being processed by another Service Manager and cause duplicate delivery items to be created and delivered. Single delivery of items using Distribution Requests may not work. Single delivery can be guaranteed using Simple or Aggregate Requests with the `preventDuplication` JMSHeader value set to true.

XML-Based

A Distribution Request is a self-contained XML file. All required information for the request processing is specified as various nested elements. The top-level elements are mainly collections of other elements. The top-level elements are described in the table below, while the other sections are drill-downs of each individual collection and their elements.

At various levels, the request also embeds custom information (called metadata) provided by the submitter for use by the Event Handlers. Metadata is custom, both in content and structure.

In the diagram on page page 55, notice the main components of a Distribution Request, which are also the main topics of the explanations that follow:

- Distribution Package
- Recipients
- Delivery Channels
- Event Handlers

ContentType

The **ContentType** parameter specifies the MIME type. (See *Multipurpose Internet Mail Extensions (MIME)* on page 180.) There are four places where ContentType can be specified:

- as a Distribution Item parameter
- as a Recipient Item parameter
- as a Delivery Channel parameter
- as a Delivery Preference parameter

The MIME types are:

- Adobe PDF – application/pdf
- HTML – text/html
- Microsoft Word – application/msword
- PCL – application/vnd.hp-PCL
- Postscript – application/postscript
- RTF – application/rtf
- TIFF – image/tiff
- TXT – text/plain

The following MIME types are also available for the Transform service only:

- AFP – application/afp
- XML – application/xml

Example

Here is a practical example of where those settings might be used:

> A Microsoft Word document needs to be saved in a repository in TIFF format. The Distribution Item ContentType is `application/msword`. However, the Recipient Item specifies a ContentType of `image/tiff`. The Distribution Service will understand that a Render service from Word to TIFF is required.

Additional Details about ContentType

The render-param parameter defines the Recipient's additional preferences for each type of delivery channel:

- the default driver name for each type of rendering
- the default number of copies for any rendering type
- duplex or collate – for pcl or ps rendering types

The default ContentType associated with an output channel is a configurable parameter. However, the ContentType and render-param parameters can be specified explicitly in a Distribution Request:

For each Delivery Channel:

The `ContentType` specified at the Delivery Channel level of a Distribution Request takes precedence over the parameters specified in the configuration settings.

For each Delivery Preference of a Recipient:

This ContentType value will be used as the default value for all Recipient Items in a Recipient Package associated with the Delivery Preference. For example, if a Recipient would like to receive e-mail attachments in Microsoft Word format,

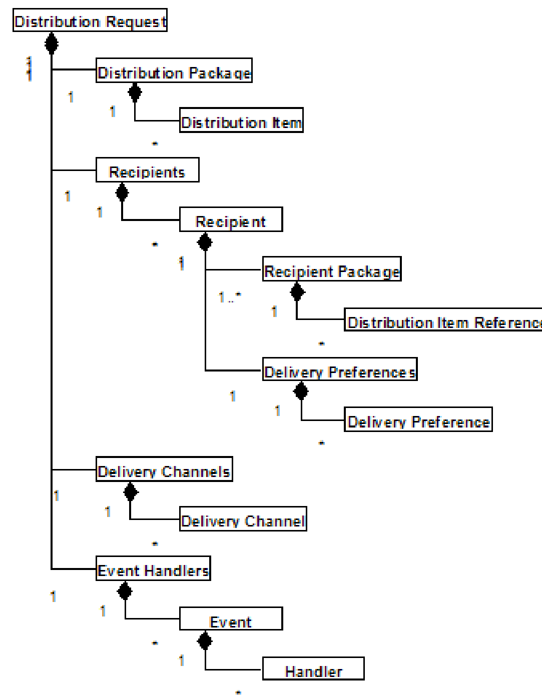
then it would be necessary to specify the ContentType for "preference-email" as "**application/msword**". If the Recipient prefers saving document in the DMS in PDF format the ContentType of a "preference-repository" should be specified as "**application/pdf**". The ContentType and render-param specified at the Delivery Preference level take precedence over the parameters specified at the Delivery Channel level.

For each Recipient Item in the Recipient Package:

This ContentType value will be used for the Recipient Item in the Recipient Package. The ContentType and render-param specified at the Recipient Item level take precedence over those parameters specified at the Delivery Preference level.

Note: With regard to the Content Type parameter when rendering from Word to HTML, the Distribution Request functions somewhat differently than the Simple Request, as described below. In a nutshell, the graphics sub-folder for a Distribution Request is named after the Distribution Item source, whereas for a Simple Request, the graphics sub-folder's name refers to the destination (final) file name.

The Distribution Request Structure



The following table describes the components of the Distribution Request.

Component	Description
distribution-package	A collection of Distribution Items that make up the subject of the Document Distribution process.
delivery-channels	A collection of Delivery Channel elements that are to be used to deliver the content created following the processing of the Distribution Package.

Component	Description
recipients	A collection of recipient elements that make up the full list of recipients to which documents in the Distribution Package are distributed.
event-handlers	A collection of special processing operations that are to be invoked as a result of events that occur during the processing of the Distribution Request. Event handlers are grouped per the events to which they are associated.
failure-policy	A policy for error treatment in the processing of a Distribution Request. The two options of the Failure Policy are: • **failFast** - the Distribution service will attempt to finish the execution of a Distribution Request as soon as possible. (See more details about this Failure Policy below). • **perseverance** – the Distribution service continues the processing of a Distribution Request to the best of its ability. (See more details about this Failure Policy below)

IStream Publisher Distribution Request Failure Policy

There are two different Failure Policies available that can be specified by the user: *Fail Fast* and *Perseverance*.

Fail Fast Failure Policy

Using the Fail Fast failure policy, all further processing of the entire Distribution Request will end at the first failure of a task, even if there are non-dependent tasks that could be processed. This is the default failure policy for a Distribution Service Request.

Example

A Distribution Request is required to generate Policy Pages for two recipients: an Agent and an Insured (customer). It must send the Policy Pages both to the Agent and to the Insured to print for further mailing. A letter of notification must be sent to the Agent by e-mail.

This request should be sent using the FailFast failure policy, since there is no reason to send a notification through e-mail if either the printing or the generation fails.

Perseverance Failure Policy

Using the Perseverance failure policy, any non-dependent tasks will still get processed, even if a failure occurs, while tasks that have assigned dependencies will end.

One reason you might want to use perseverance is when you are sending out e-mails or faxes to many recipients and you do not want the entire Distribution Request to stop just because one of them fails.

Example

A Distribution Request is required to generate a personalized letter of notification for each Policy Owner (multiple recipients). It must then send it to be printed for further mailing. If the generation of a document for one recipient fails, the business requirement is to send the notification to as many clients as possible.

This request should be submitted using the Perseverance failure policy. In this case, even if the distribution for one recipient fails, other nodes (generate-render-deliver) for the rest of the recipients should still be completed.

Troubleshooting the Distribution Request

All Distribution Requests are initially processed by the Distribution Service.

The Distribution Service validates an XML request and immediately returns failure if it is incorrect.

Example:

```
<distribution-request-response status="failure">
    <errorDetails messageKey="DR.01"
      text="DR.01: Failed to parse DistributionRequest: …
SAXParseException
…
```

This error message means that parsing of the XML request failed and the user should verify and fix the XML data before sending a new request.

Examples of other error messages include:

- **DE.06: DistributionItem not found: (item id)**. – means that the attribute refID in the element recipient-item has a reference to a distribution-item with an id which does not exist.
- **DE.07: Delivery preference refers to non-existing channel: refID=channel id** – means that an element preference-repository, preference-printer, preference-email or preference-fax has an attribute refID which is pointing to a non-existing delivery channel id.
- **RP.01: Failed to resolve delivery preference: preference id** – means the attribute refID of the element delivery-preference has a reference to a preference-* id which does not exist.

Errors such as these can be avoided if the user strictly follows the Distribution Service specification provided in the *IStream Publisher Interface Reference Guide*.

Distribution Service

The Distribution Service itself does not perform any actual operation apart from sending requests to simple services to perform specific operations and collect responses.

The Distribution Service always waits for the completion of all simple requests that are required for processing of the Distribution Request. However, if a required simple services is not available (due to system failures or incorrect configuration) then the Distribution Service will be trapped in a wait state.

Administrators can detect problems by monitoring messages waiting in the Service Queues or by using the System Query State operation in the Console.

Loopback Service

When a simple request created by the Distribution Service is completed, the Loopback Service (a special service used only for processing responses of simple requests created by the Distribution Service) receives a response and notifies the Distribution Service about completion of the simple request. When the simple request is completed successfully, the Distribution Service then sends simple requests, or returns a response to the requestor (if required in the request JMS message) if there are no more tasks required for completion of the Distribution Request.

All simple service failures are recorded by the Distribution Service, returned to the requestor, and logged to the Request Log after completion of the Distribution Request processing.

Error Messages

Error messages returned by various simple services can vary. The Distribution Service wraps simple service error responses into its own response message.

A user might find that a failure is based on a combined Distribution Service Response or by simulating Distribution Request processing using simple requests.

For example, a Distribution Item containing the element `<calligo-item>` requires an invocation of Content Service, which can return error messages if there is a problem during document generation.

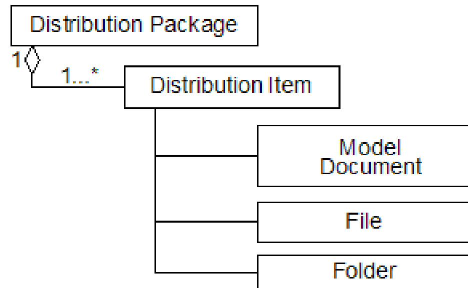
Typically, these sort of errors are self-sufficient and are quickly detected. But some errors can be caused by an implicit invocation of some services.

For example, if a clg-source is to be delivered as a fax, it is converted to a Microsoft Word format using the render-CLG-to-Word service. The Word document is then rendered to a TIFF format by the render-Word-to-TIFF service and sent to the fax server by the deliver-to-fax service. Any simple service can fail (for example, a Worker that does not have a Fax Printer Driver, but has started the render-Word-to-TIFF service) and cause the failure of the whole distribution request.

To avoid difficulties with troubleshooting the Distribution Requests, a client can set the JMS message property LogLevel to 6 in the request message, and check the request status in the Request Log (by using the Console *Request Log Request Info* operation or executing direct queries to the Request Log table).

The Distribution Package

The Distribution Package is the full set of interrelated items (such as documents, images, folders and files), that are to be delivered. (These items are called Distribution Items.) The Distribution Package is the object of the distribution process. A business process usually determines the contents of this package. The Distribution Package is an unordered set.



The Distribution Package Structure

The following table describes the parameters of the Distribution Package.

Parameter	Description
distribution-package metadata	Information specific to all Distribution Items in the Distribution Package. This also constitutes the Distribution Request global metadata. The metadata specified in this block is available to all Event Handler invocations, regardless of the event they are handling.
distribution-item	Documents or files that make up the Distribution Package.

The Distribution Item

Any file can be a Distribution Item: a Microsoft Word document, an image in TIFF format, a PCL stream, an XML data file, and so on. The system handles the following items in special ways, but like all items they can appear in any number and combination in the Distribution Package. These “special” Distribution Item types are described in more detail in the following sections.

The IStream Document Item <calligo-item>

Calligo documents are now called *IStream documents*, however the term *Calligo* is still used in the tags. Therefore, `<calligo-item>` refers to an IStream document item.

If a Distribution Request requires the generation of an IStream document, then either the Model document (.cms) or the IStream document (.clg) should be specified as the source for the Distribution Item. For more information about generating a document, please see *Generate Calligo Document Service Overview* on page 23.

If the model document (.cms) or generated document is the Distribution Item itself, then it should be included in the package as a generic file.

Keep in mind that there are two types of source documents: model documents (using an extension of .cms) and generated documents (using an extension of .clg).

Repository File

The files referenced by the Distribution Items are not physically contained in the request. Only references are included, each specified as a URL. RAPI (Repository Application Programming Interface) is used to access the referenced files and therefore they must be located in a RAPI compliant repository.

Repository Folder

The system iterates over the contents of the folder and adds all its files to the Distribution Package. The repository must be RAPI compliant, (that is, provide a RAPI adapter).

The Distribution Package can have one or many Repository Folders as Distribution Items.

IStream Publisher processes only the current Folder and does not support subfolders, where a subfolder is a subdirectory in a core Folder with a subset of distributable documents/files.

A Folder breaks up into a list of distributable items based on wildcards specified by the user.

Wildcards

Wildcards are supported for extracting files from the Folder. A wildcard is a character that represents one or more characters. Two commonly-used wildcard characters are:

- The question mark (?), which represents any single character, and
- The asterisk (*) which can be used to represent any character or group of characters that matches that position in the target set of filenames.

Distribution Item Description and Syntax

A Distribution Item is one of the documents or files that make up the Distribution Package. A Distribution Item can be either a file or a folder that contains any number of files.

Deleting after Distribution

By default, a Distribution Item will not be deleted after distribution. However, there is a flag – the “Delete After Delivery” flag (see below) – that can be set to automatically delete files after distribution, saving time and reducing expensive hard disk overhead. The “Delete After Delivery” flag is an option for the Distribution Item.

There is also the option of using a Wildcard “filter” to control very precisely the specific files that do get deleted if the “Delete After Delivery” flag is set to True.

Example

There are a number of distributable documents in PCL format in a Repository. The goal is to concatenate all distributable documents that start with “F” to one PCL stream. The Distribution Item is a Folder, the “Delete After Delivery” flag is set to True, and the filter specifies “F*” meaning all files starting with “F”. In this scenario, all files from the Folder that match the filter (that is, start with “F”) will be deleted after the distribution, assuming that it takes place successfully. The Folder itself and subfolders will not be deleted. Only the files that have been distributed by IStream Publisher, and that start with “F”, are deleted.

Note: Only File and Folder types of Distribution Item documents can be deleted. The “Delete After Delivery” flag does not apply to IStream source documents.

ContentType

The **ContentType** parameter is never used to filter files even if the folder is defined in the Distribution Item. (For filtering files, the Distribution Request “filter” element exists.) The ContentType parameter functions together with the other pieces of the Distribution Request to define what rendering will be applied to the file before delivery.

The following tables describe the Distribution Item parameters.

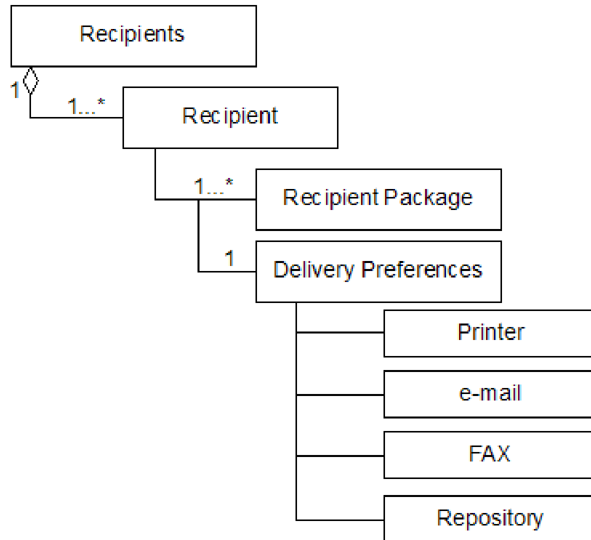
Distribution Item Parameters

Please refer to the schema.

Recipients

A recipients element represents metadata valid for all recipients as well as the collection of Recipient Package, Recipient Metadata and Delivery Preferences for each recipient.

This diagram gives an overview of the structure of the Recipients element:



The following table describes the parameters of the Recipients element.

Parameter	Description
recipients-metadata	This is information specific to all recipients in the Distribution Request. The metadata specified in this block is available for all Recipient Packages or Delivery Packages when the appropriate Event Handler is invoked.
recipient	A person or an organization to which the items in the Distribution Package are addressed. See below.

Recipient

A recipient is either a person or an organization to which the items in the Distribution Package are addressed. A recipient element contains Recipient Packages, recipient metadata and Delivery Preferences for the recipient. The list of recipients can optionally be ordered by assigning a delivery order to each recipient. Recipients for which a delivery order is not specified, are processed after all of those for which a delivery order is specified.

Note: If the delivery order is specified, the ordering is enforced only in the case of delivery through the same delivery channel and this channel operates in Synchronized mode.

Recipient Parameters Example

Here's a simple example of how the recipient parameters can be used in order to save time and streamline your business processes:

An agent prefers to receive Recipient Package #1:

- In TIFF format by FAX
- By e-mail.

At the same time, this agent prefers to receive Recipient Package #2:

- In PCL format by printer.

In multiple places in the Distribution Request, the `refID` attribute is used. The Distribution Request will not pass syntax validation if any `refID` references an element with an ID that does not exist somewhere in the request. However, request validation cannot check the semantics of such references. In other words, be aware that an ID can exist (and pass the syntax check) but nevertheless not work in the context.

If you want to be able to validate the `refID` references, you would need to use an external XML editor application.

For details of the Recipient element, please refer to the schema.

RefID Example

Here is an example of a refID along with the element to which it refers:

```
<preference-xxx refID={refers to the corresponding delivery
channel ID} ...>
```

Note: It is required that the preference-printer must be pointed to the printer delivery channel, the preference-email to the e-mail delivery channel, and so on.

Delivery-Preference Considerations

Delivery-preferences provided in the Recipient Package define what channels are used to deliver any given Recipient Package. This way the same Recipient Package may be delivered simultaneously through multiple channels.

Sometimes a service has finished the execution of a request, but one of the following failures occurs:

- The Service Manager fails
- A service fails
- The database connection is lost before the Service Manager has acknowledged the transaction

In this case the request is rolled back to a Service Queue and marked by setting the JMSRedelivered header field of the rolled back request to "true". After this happens, the next available Service Manager will execute the request.

In some failure cases the above process may lead to duplication of the request. Depending on the business situation, this may or may not be critical. If a request is

"print a bill", it might be critical. On the other hand, if it is the generation or rendering of a document, it may not be critical.

To prevent duplication, you should specify the 'preventDuplication' attribute of a <delivery-preference> as 'true'. If the 'preventDuplication" is true, the Service Manager will check the status of the previous attempt. If the previous attempt completed successfully, then the Service Manager just acknowledges the transaction without executing the request again.

Important: Setting the 'preventDuplication' flag set to 'true' will definitely impact performance, because of the extra checking operation. Use the flag only for critical cases or if performance is not an issue.

Logical vs. Physical Delivery Units

Note that the Recipient Package is considered as a logical delivery unit. For some channels the physical delivery unit coincides with the logical one, for example all recipient items are included in a single e-mail as attachments. (A similar situation can occur with a fax.) At the same time, for other channels the logical package is split into multiple physical units. Examples of this would be delivery to repository and to printer.

In other words for the e-mail and fax delivery, one simple Service Request is generated per Recipient Package while for the repository and printer one Simple Service Request is generated per Recipient Item.

Recipient Package

In the Recipient Package, you specify the recipients that will be receiving the distribution package and the delivery order for each recipient. You can have one or more recipients in each Distribution Request. A Recipient Package, therefore, is a subset of the Distribution Package to be delivered to a particular recipient. It contains a list of references to items in the Distribution Package that make up a Recipient Package.

ContentType Issue

The Content Type (ContentType) of a Recipient Item is required by the Distribution Service to determine the type of rendering for the preferred Delivery Channel. The ContentType parameter includes the MIME type of the content and the setting for an appropriate rendering driver.

If ContentType is not specified for a deliverable item (recipient-item) in a Recipient Package and in the delivery preference or the delivery channels in the Distribution Request, then the rendering format to pcl (for printer) and to TIFF (for fax) is derived from the configuration setting. No rendering is made for e-mail and repository channels when ContentType is omitted. When ContentType is specified explicitly it defines to which mime type the original distribution item is rendered (and is sent to delivery channel). An explicitly specified ContentType overwrites the type associated with a given delivery channel by default.

As a default, the Distribution service considers the document formats associated with each type of channel based on delivery preferences:

- PCL format for delivery to Printer
- TIFF format for delivery to FAX
- MIME type of an original Distribution Item in the Distribution Package for delivery to a Repository or by e-mail.

If a delivery preference is a FAX or e-mail, all items in the Recipient Package are sent as attachments in one FAX or e-mail.

The items in a Recipient Package may be ordered using the optional seqNumber attribute.

Note: For fax and e-mail, we can map many items to one Simple Request. However, for repository and printer, it is a one-to-one relationship: one Recipient Item maps to one request. Also, it is possible for one Recipient Item to produce many Simple Requests.

For details of the Recipient Package and the Recipient Item Reference parameters, please refer to the schema.

The ContentType Parameter

The Recipient Item has an optional attribute ContentType (described above), which takes precedence over the default rendering for a given channel.

The engine does not apply any rendering if ContentType is omitted in the configuration settings for the associated delivery channel.

Rendering Services

If there is no rendering service when rendering is required (all rendering services are defined in the config file DistributionRequestStatic.xml), then the whole Distribution Request fails. If the ContentType attribute is defined in the Recipient Item, then the engine looks for a rendering service to support rendition to that type. This ContentType determines the document type being sent to the delivery channel. For instance if the ContentType is 'application/msword' and the original document is 'application/msword' as well then no rendering is done for recipient item. The Word document is sent to the printer 'as is' (and as a likely result some garbage will be printed).

render-param subelement

Render-param defines additional configuration settings for rendering. It does not affect in any way what rendering is chosen (see ContentType above). Those settings are taken into account only if they are relevant to the chosen rendering. For example, if the rendering is TIFF to PCL while render-param defines some fax settings, then they are simply ignored (they would have to be printer settings in order to have any effect).

The rationale behind such behavior may be illustrated with the case when multiple delivery preferences are defined for a particular recipient item. In this case, the recipient item is delivered through multiple delivery channels with possibly

different renderings for each channel. The rendering settings defined in render-param that are irrelevant in each case, are ignored.

Element <part name="xxx">

This element (under recipient-item) was added to allow the specification of mail attachment names. By default, the attachment name is the same as the file name provided in the distribution item. However in the case of a `<calligo-item>` and generation, this name may be rather cumbersome and should be replaced in the e-mail with something more readable. It is not important for the case of delivery to fax and printer where those intermediate names are never exposed to the client. For the delivery to repository element, 'destination' serves the same purpose.

Delivery to Repository

In the case of delivery to repository, there are multiple ways to specify the destination. If the destination is defined per recipient-item then that value is used. The destination is treated as a folder if it ends with a trailing slash and the destination file name is defined by the distribution item (with possibly an extension replaced if rendering took place). Otherwise, the destination is considered as a file path and the generation/rendering result (if any took place) is eventually renamed (copied) to that file. Another way to define the destination (likely a destination folder) is in the absolute-path element of the preference-repository. The treatment is the same.

If the destination is not supplied, then the absolute path defined in the preference-repository will be used in the same way as is describe above.

Note: There are two cases with regard to the filename:

- The full path, including the file name, is specified, or
- The path is specified without the filename, and with a trailing slash – implying that this is a folder. In this case, the file name specified in the source, will be added to the path.

Note: The extension of the file name will be adjusted according to the ContentType of the resulting file.

Calligo-destination

This item is processed only if the corresponding Distribution Item contains a `<calligo-item>`. In this case, generation and delivery to repository may be done in one step using a single Service Request.

Delivery Channels

A Delivery Channel is the method by which a Distributed item is delivered. It represents a device that can transmit information in a printed form (print or fax) or in an electronic form (through e-mail or to a repository). Channels can be configured for different content formats, transmission protocols, and so on.

Specific information is associated with each type of channel in order for IStream Publisher to perform the physical communication with the channel.

IStream Publisher supports the following Delivery Channels:

- Printer
- E-mail
- Fax
- Repository

Operating Modes

Delivery channels can function in two operating modes depending on the moment when the actual delivery of an item occurs in relation to the other items in the delivery package: *Synchronized Delivery* and *Instant Delivery.*

Synchronized Delivery

In the *Synchronized Delivery* mode, the items in the Delivery Package are not delivered until they become available and a *delivery package ready* event is raised. At this point, all items are then delivered one by one.

The delivery order in Synchronized mode is affected by:

- the `deliveryOrder` attribute of all recipients
- the `deliveryOrder` attribute of all Recipient Packages
- the `SeqNumber` attribute of all Recipient Items

You can use the `syncId` Attribute for synchronization between multiple channels. The same value of this attribute in different delivery channel definitions means that delivery to all channels begins only when all the items for this channels are ready for delivery.

In synchronized delivery, if one transmission fails, then further requests will not be sent. See *Fail Fast Failure Policy* on page 56.

Instant Delivery

In the *Instant Delivery* operating mode, the items in the Delivery Package are delivered as soon as:

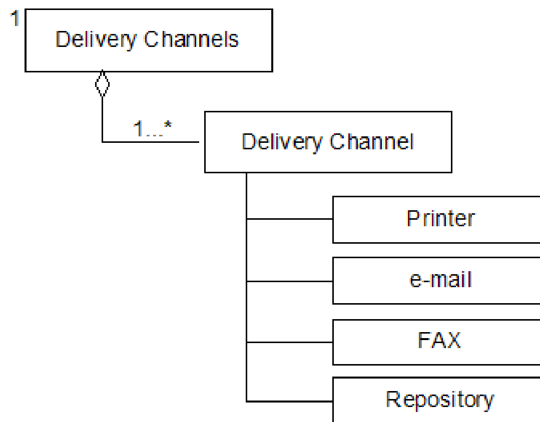
- all the items of a specific request are ready (generated/rendered), *and*
- all the Event Handlers associated with that item are complete

With e-mail, fax or repository, you would not use synchronized delivery because the order does not matter. When delivery in order is important there are several elements of the Distribution Request to check:

1. The Delivery Channel in use should be in 'Synchronized' mode. This means that the delivery does not start before all items are ready. All generations, renderings and proper Event Handlers must be complete. It also means that all Delivery Requests are sent one by one.
2. Recipients and Recipient Items must be ordered using their deliveryOrder attribute (integer) and the Recipient Item must be ordered using its seqNumber attribute (integer). Elements with lower numbers are served first.

Note: In the case of a Recipient Item that supports a Simple Request with many items, for example e-mail, the seqNumber controls the order within the e-mail, that is, the order of the attachments.

The following diagram describes the structure of Delivery Channels and the table describes the Delivery Channel parameters:



For details of the Delivery Channels, please refer to the schema.

Delivery Channel Settings

There are multiple places where Delivery Channel-related information is specified in IStream Publisher. Each setting ultimately references the appropriate Delivery Channel parameter.

The “Delivery Channels” item lists all output channels that can be used in the Distribution Request. In the Delivery Preferences for the recipient, there is a Reference ID pointing to the “Delivery Channel” ID.

Recipient-Specific Information

Additionally, there will be recipient-specific information related to the output channel. For example, the Fax Number in the Delivery Preference would be different for each recipient.

As a further example, a printer that is defined using the ID parameter of the Printer parameter, above, can be referenced with the (optional) preference-printer parameter of the recipient.

Similarly, an SMTP server can be identified as a Delivery Channel and referenced using the recipient preference-fax parameter.

Event Handlers

You can use Event Handlers to extend the Distribution Request processing.

During the processing of the Distribution Request, various events are activated to indicate where the processing is and to allow the request processing to be extended by invoking an Event Handler associated with the particular event.

Events occur every time the processing of a Distribution Request reaches a point where a specific action should be taken or where customization of the distribution process itself is possible.

When invoking events, the system must allow them access to a particular scope of metadata. The system informs the Event Handler with regard to what metadata scope it is allowed to access by passing it a metadata scope identifier.

Event Handlers can be critical or non-critical.

Events

The events raised by the Distribution service are:

Distribution Package Ready

Raised when all the items that make up the Distribution Package are available. At this point all document generation operations have completed and all folders have been iterated for content. The metadata scope for this event is that of the Distribution Package element.

Recipient Package Ready

Raised when all the items that make up a recipient's package are available. At this point all render operations have completed. The metadata scope for this event is that of the Recipient Package element.

Delivery Package Ready

Raised when all the items that are to be delivered through a Delivery Channel operating in synchronized delivery mode are available. At this point the Distribution Package and all the Recipient Packages are ready. The metadata scope for this event is that of the distribution channel element.

Delivery Item Ready

Raised when an item for delivery through a channel operating in instant delivery mode has become available. At this point the item is ready for delivery but the rest of the items in either distribution, recipient or delivery packages are not necessarily ready. The metadata scope for this event is that of the distribution channel element.

Package Delivered

Raised when a delivery package has been delivered. Applies only to Delivery Channels operating in synchronized delivery mode. The metadata scope for this event is that of the distribution channel element.

Item Delivered

Raised when a delivery item has been delivered. Applies only to Delivery Channels operating in instant delivery mode. The metadata scope for this event is that of the distribution channel element.

Distribution Complete

Raised when the processing of the Distribution Request has completed. At this point all the nodes in the task graph have been processed and the request state and temporary files have been removed. The metadata scope for this event is that of the Distribution Request (global scope).

Events and Services

As listed in the following table, an event either supports (●), partially supports (❍), or does not support (no dot) specific services. (For example, the recipient-package-ready event can perform all the services.)

Events / Services	concatenate-pdf	concatenate-ps	concatenate-pcl	delete-files	count-pages	generate-calligo-document	render-TIFF-to-PDF	render-Word-to-TIFF	render-Word-to-PCL	render-Word-to-PDF	render-Word-to-PS
distribution-package-ready				●		❍	●	●	●	●	●
distribution-complete				●		❍	●	●	●	●	●
recipient-package-ready	●	●	●	●	●	●	●	●	●	●	●
delivery-package-ready				●		❍	●	●	●	●	●
delivery-item-ready				●		❍	●	●	●	●	●
package-delivered				●		❍	●	●	●	●	●
item-delivered				●		❍	●	●	●	●	●

Note: You cannot use the **encrypt-pdf** services in *any* event.

Sequence Number

Event Handlers can be invoked and executed either concurrently or sequentially. The default behavior is for the Event Handlers to run concurrently, meaning that there is no predetermined order and none of the individual processes will wait for any others to complete. If both sequenced and concurrent Event Handlers are specified, the sequenced ones are executed first, in order, after which all non-sequenced requests are executed concurrently.

The event-handler “seqNumber” parameter is the specific setting that defines the order of execution of Event Handlers.

Multiple Event Handlers

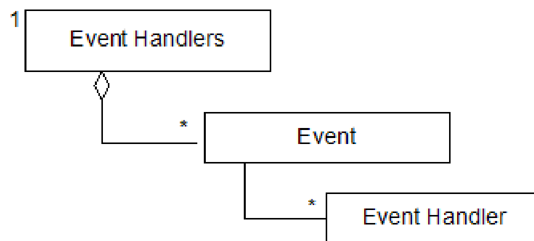
If multiple Event Handlers relate to the same event, the seqNumber parameter could be used to sequence the execution of these Event Handlers if it is important in the context. If it is not important, and if seqNumber is not specified, then processing may happen simultaneously.

Event Handlers can alter the contents of the Delivery Packages, for example, they can concatenate multiple Deliverable Items into one PCL stream. Event Handlers are not allowed to alter the Distribution Package or the Recipient Packages.

Critical or Non-Critical

Event Handlers may be labeled as critical or non-critical. Non-critical Event Handlers don't cause the whole distribution request to fail. Instead, in the case of failure, they are reported in the Distribution Request response (which in the case of failure has the status 'success-with-info'). Note that for some Event Handlers the Event Handler proxy may be defined in CommonSM.xml – those Event Handlers are always treated as critical. The Event Handler proxy (if defined) transforms the generic Event Handler request into a simple Service Request (it happens on the Distribution service side). This simple request is sent as usual to the simple service and the simple service response is again handed to the proxy for post-processing.

The diagram below describes the structure of the Event Handlers elements:



The following table describes the event parameters:

Event	Description
event	See “Event Handlers, Event Parameters” below.
distribution-package-ready	The metadata scope for this event is that of the Distribution Package element.
recipient-package-ready	The metadata scope for this event is that of the Recipient Package element. The Recipient Package Ready event has one Parameter: recipientpackageRefId – A reference to the Recipient Package ID. The event is raised when all the items that make up the specified Recipient Package are available. At this point all render operations have completed.
delivery-package-ready	The metadata scope for this event is that of the distribution channel element. The Delivery Package Ready event has one Parameter: • deliveryChannelRefID – A reference to a Delivery Channel. The event is raised when all the items that are to be delivered through the specified Delivery Channel operating in synchronized delivery mode are available.
delivery-item-ready	The metadata scope for this event is that of the distribution channel element. The Delivery Item Ready event has the following parameters: • deliveryChannelRefID – A reference to a Delivery Channel. • deliveryItemID (optional) - A reference to a Distribution Item. This parameter will be filled in by IStream Publisher. (If a user sets it in their request, it is ignored.) The event is raised when the item that is to be delivered through the specified Delivery Channel operating in Instant delivery mode is available.
package-delivered	The metadata scope for this event is that of the distribution channel element. The Package Delivered event has one Parameter: • deliveryChannelRefID – A reference to a Delivery Channel. The event is raised when all the items that are to be delivered through the specified Delivery Channel operating in synchronized delivery mode are available.
item-delivered	The metadata scope for this event is that of the distribution channel element. The Item Delivered event has the following Parameters: • deliveryChannelRefID – A reference to a Delivery Channel. • deliveryItemID (optional) - A reference to a Distribution Item This parameter will be filled in by IStream Publisher. (If a user sets it in their request, it is ignored.) The event is raised when all the items that are to be delivered through the specified Delivery Channel are available. This event applies only to delivery channels operating in Instant delivery mode.

Event	Description
distribution-complete	The metadata scope for this event is that of the distribution request (global scope).
event-handler	An Event Handler is invoked when the appropriate event happens. The Event Handler element has the following Parameters: • serviceType – name of the specific Event handler that should be invoked. • seqNumber – The order in which multiple Event Handlers are executed synchronously if there is more than one Event Handler associated with the particular event • event-handler-metadata – metadata specific to the Event Handler • critical – “true” or “false”

event-handler-response

This parameter provides a successful completion response by simply returning a successful completion status.

A failure response will contain:

- An error ID, and/or
- An English text message explaining the reason for the failure, and/or
- An XML fragment with extended error information.

The Response message for a particular Event Handler can be extended depending on the business requirements.

Distribution Request Metadata

Event Handlers

Event handlers are given access to request metadata and distribution, recipient and delivery packages by means of Data Access Objects and Value Objects. This DAO observes the scoping rules and make the metadata available to Event Handlers as XML fragments. Each element of the Distribution Request has its own scope, enclosed within the scope of its main, higher level element.

When an event occurs within the scope of a certain element, the metadata for that element, together with the metadata from all of its ancestor elements, is made available to the Event Handler. When the system invokes the Event Handlers, it passes them a scope identifier, which they can use with the DAO to access the metadata. This is the interface to the persisted state information and no Event Handler should access it directly.

The Event Handler can unmarshal the XML fragment into Value Objects.

Metadata Elements

Metadata elements may include arbitrary user data. However in order to allow co-existence of data belonging to multiple customers, the general agreement is that each customer defines its own sub-element in metadata. In other words, the

definition of the metadata element itself is not changed and PCDATA is only expected under the metadata element itself.

Concatenating PCL Streams

The `Concatenate PCL` Event Handler is invoked when the Recipient Package Ready event is raised and all items to be delivered through a delivery channel operating in synchronized delivery mode are available. At this point, the Distribution Package and all the Recipient Packages are ready.

Note: A standard concatenation event handler requires the `absolute-path` element within the `preference-repository` specified and cannot include a destination element.

Multiple Streams into One Job

Some documents need to be printed as a single job to avoid pages from other printer jobs being intermixed. The Event Handler provides the ability to concatenate the PCL streams for these documents in the correct order.

One Job into Multiple Streams

Alternatively, the Event Handler concatenates Distribution Items in a Recipient Package and/or through multiple Recipient Packages to some specified number, N, of approximately equal-size PCL stream segments. The order of the concatenated documents remains unchanged.

The number of streams is specified in the Distribution Request. The default number of streams is "1".

Header Page

Each subgroup must have an appropriate header page defined so that, after delivery to a print server with load balancing across multiple (N) printers, a person can reassemble the documents back into the correct sequence for delivery.

The Header Page can be composed using a Header Page Template. The template contains some special placeholders.

Event Handler

The Event Handler uses the template and substitutes these placeholders with actual values provided with the request. The place where content must be substituted, is marked by “{<name>}” – where <name> is the name of the value to be used.

If a name provided in the request does not match a placeholder in the template, the Header Page will not include the information.

Template URL

The URL to the template should be specified in a config file for the Event Handler. If there is no template associated with a Header Page, then a Header Page as plain text will be composed “on the fly” and every field will be printed in a separate row, left-aligned.

The template can be in plain text or in PCL format. You can create fancy Header Page templates with placeholders using Microsoft Word and Render to PCL format. See *Header Page Template Example* on page 173.

Optional Parameters

The following parameters can optionally be provided for a Header Page:

- paper size
- paper orientation
- tray/paper source

Each header page must be concatenated, with the appropriate PCL segment on top.

Duplexing Options

In the concatenation process, the duplexing of the pages can either be continuous (meaning that the next concatenated stream can start either on an even or an odd page) or, it can break at odd-numbered pages. If the Print Instruction parameter for duplexing is not specified in the Distribution Request, the Event Handler will assume a break at odd-numbered pages.

Concatenating PS Streams

For an overview, see *Concatenating PS Streams* on page 43.

The information below describes the parameters for the PS Concatenation Event Handler within a Distribution Request.

Parameter	Description
page-header	A separate page defined so that a person can reassemble documents printed across multiple printers back into the correct sequence. The Header Page composed on the fly for each of (N) PCL segments using a template with appropriate delivery order on each page. Each header page must be concatenated with the appropriate PCL segment on a top. See “Page Header Parameters”, below.
paperSize	Paper Size for a Page Header, for example, Legal, Letter
paperOrientation	Portrait or Landscape
paperSource	Defines the printer tray ID for the Header Page. By default and in case of wrong tray ID provided in the request, the service would use the default tray for the Header Page.
field	Specify fields to be printed on the Header Page. All fields are character type fields. Specify field names and value of the field. The field names are: • job-name – Standard name passed as a parameter on the request, Length – 10 • job-number – request ID number – format 99999999 (keep leading zeroes). Length – 8 • job-submit-date – job submit date – format YYYY-MM-DD • job-submit-time – job submit time – format HH:MM:SS (24 hour clock) • message – The message. Length – 256 • seq-number – sequence number for identification purposes • distribution-instructions – General message text defining job distribution information. Length – 512
numberPSsegments	The number of destination PS streams into which the source PS streams will be concatenated. The default number of PS segments is "1".

The information below describes the optional parameters included in the recipient-package-metadata within a Distribution Request when concatenating source PS streams for different recipients.

Command/Element	Description
PSsegmentID	The specific destination PS stream where the source PS streams for the particular recipient will be concatenated.

The information below describes the optional parameters included in the recipient-item-metadata within a Distribution Request when concatenating source PS streams for a particular recipient.

Command/Element	Description
PSsegmentID	The specific destination PS stream where the particular source PS stream will be concatenated. If a PSsegmentID is specified here for a particular recipient's item and a PSsegmentID is also specified for that recipient in the ps-concatenate-recipient-package-metadata, and the two do not match, the PSsegmentID specified here takes precedence.

Calling the Transform Service

You can call the Transform simple service from a distribution request, however you need to configure the request in a specific way: see *The Transform Service* on page 49.

A Distribution Request Example

The following is an example of a Distribution Request. It renders three files: Test1.doc, Test2.doc, Test1.tif to pcl format, concatenates them into a single file and then sends the result to a printer.

```
JMSType distribution-service
JMSDestination  ForTesting.submit
RequestID   123456

<?xml version="1.0"?>

<distribution-request>
  <distribution-package>
    <distribution-item id="D-ITEM-1">
    <file>
      <item-source url="ftp://user:password@ftpserver/
source/Test1.doc" ContentType="application/msword"/>
    </file>
    </distribution-item>
    <distribution-item id="D-ITEM-2">
    <file>
      <item-source url="ftp://user:password@ftpserver/
source/Test2.doc" ContentType="application/msword"/>
    </file>
    </distribution-item>
    <distribution-item id="D-ITEM-3">
    <file>
      <item-source url="ftp://user:password@ftpserver/
source/Test1.tif" ContentType="image/tiff"/>
    </file>
    </distribution-item>
  </distribution-package>

  <delivery-channels>
    <printer id="PRINTER-1" printerName="\\PDCMAR-
01\5_ES_HPLJ5si" operatingMode="Synchronized"
outputName="HP"/>
  </delivery-channels>

  <recipients>
    <recipient id="RCP-1" deliveryOrder="2">
    <delivery-preferences>
      <preference-printer id="PREF-PRINTER-1"
refID="PRINTER-1"/>
    </delivery-preferences>

    <recipient-packages>

      <recipient-package id="RCP-PKG-2" deliveryOrder="2">
```

```
        <recipient-package-metadata>
        <pcl-concatenate-recipient-package-metadata
PCLsegmentID="10"/>
        </recipient-package-metadata>

        <recipient-item refID="D-ITEM-1" seqNumber="1">
        <recipient-item-metadata>
          <pcl-concatenate-recipient-item-metadata
duplex="continuous"/>
        </recipient-item-metadata>
        <render-param outputName="HP">
          <pcl>
            <printer-configuration pageRange="4,1,1"
duplex="flipLongEdge" collate="off"/>
          </pcl>
        </render-param>
        </recipient-item>

        <recipient-item refID="D-ITEM-2" seqNumber="2">
        <recipient-item-metadata>
          <pcl-concatenate-recipient-item-metadata
duplex="continuous"/>
        </recipient-item-metadata>
        <render-param outputName="HP">
          <pcl>
            <printer-configuration pageRange="4,1,1"
duplex="flipLongEdge" collate="off"/>
          </pcl>
        </render-param>
        </recipient-item>

        <recipient-item refID="D-ITEM-3" seqNumber="3">
        <recipient-item-metadata>
          <pcl-concatenate-recipient-item-metadata
duplex="continuous"/>
        </recipient-item-metadata>
        <render-param outputName="HP">
          <pcl>
            <printer-configuration pageRange="4,1,1"
duplex="flipLongEdge"/>
          </pcl>
        </render-param>
        </recipient-item>

        <delivery-preference refID="PREF-PRINTER-1"/>

      </recipient-package>
    </recipient-packages>
    </recipient>
  </recipients>
```

```
  <event-handlers>
    <event>
    <recipient-package-ready recipientPackageRefID="RCP-PKG-
2"/>
    <event-handler serviceType="concatenate-pcl">
      <event-handler-metadata>
        <concatenate-pcl numberPCLsegments="1">
        <page-header paperSize="Letter"
paperOrientation="Portrait" paperSource="Tray 1">
          <field name="job-name" value="RCP-PKG-2"/>
        </page-header>
        </concatenate-pcl>
      </event-handler-metadata>
    </event-handler>
    </event>
  </event-handlers>
</distribution-request>
```

Chapter 4

Tracking and Monitoring Requests

This chapter explains the Request Log facility, which is used to trace requests and monitor their progress through their lifecycle. It is also used to log error information that results from a failure to process a request and to hold the actual content of a failed or canceled request so that it can be resubmitted later.

This chapter describes:

- *Request Messages* on page 84
- *The Request Log Table* on page 85
- *Resubmitting a Failed or Canceled Request* on page 88
- *Error Log Levels* on page 89

Note: The main parameters are listed for all services. For related parameters, please refer to the schema.

Request Messages

The system will trace all Request Messages that are waiting to be Processed, Executed or Completed.

Unique Request IDs

To trace requests the Request Log must be capable to uniquely identify them. Each request bears a unique ID assigned by IStream Publisher.

Unique Request ID

The Unique Request ID is stored under the name of InternalRequestID as Number (10) in the request table, and is used in the Status and ErrorInfo tables as a foreign key. The InternalRequestID as provided with the request at submission is kept in the request table. Since all system components need to refer to the request by the same unique ID, this ID must be assigned as soon as the request enters the system.

Live Request Message Status

Apart from being used to trace the history of past requests processed by the system, the Request Log is used to keep track of the current state of "live" requests. The status information that the system components log contains, includes the name of the component where the request resides, the time stamp when its status changed and its current status. The possible values for status are:

- **Pending** – the request awaits to be executed by a component.
- **Paused** – the request has been put on hold and the component will not continue its execution until it is explicitly directed to do so (resume the request).
- **Processing** – a component is currently processing the request.
- **Completed-Success** – processing has completed and the result was successful.
- **Completed-Success-With-Info** – Processing has completed because of the failure of a non-critical Event Handler in the Distribution Request processing.
- **Completed-Failure** – processing has completed because of a failure.
- **Completed-Canceled** – processing has completed due to an administrative command (cancel).
- **Resubmitted** – the request has been resubmitted for the processing.
- **Deferred** – the request awaits execution within a specified interval between Deferral and Expiration Time.

The Request Log Table

The system stores the request metadata and status information into a relational database table called the Request Log.

The Request Log table consists of four tables:

- Request
- Status
- ErrorInfo
- StatusOrder

Each of these tables is described in the following sections.

Request Table

This is the main table and contains one record for each request in the system. When a component submits request information to the Request Log, the metadata contained in this information is stored in the request table.

Since the latest status information about a request resides in the request table, it reduces the need to access the Status table for status information, other than looking at the history of a request's processing.

The `Request` table contains the following fields:

Field Name	Value	Data type	Size
InternalID	the primary key	NUMBER	4
RequestID	the Request ID	VARCHAR	64
OriginalRequestID	the request ID of the original failed or canceled request that had been resubmitted	VARCHAR	64
AggregateID	the aggregate Request ID	VARCHAR	64
ParentID	the parent (main) Request ID if part of an aggregate request	VARCHAR	64
ReqDocument	the Distribution Request name that refers to the type of business, for example “NEW BUSINESS”	VARCHAR	64
Requestor	a business user or component, for example, “DRM” or “John Doe”	VARCHAR	128
JMSType	the original request JMS type	VARCHAR	64
Priority	the priority of the request: a value between 0-9	NUMBER	4
TimeStamp	the final time that the request was updated by the system	DATE	8

Field Name	Value	Data type	Size
DeferralTime	the time when the request is scheduled to be processed	VARCHAR	64
ExpirationTime	the time the request must not be executed after	VARCHAR	64
SubmittedTimeStamp	the time the request was originally submitted	DATETIME	8
SubmittedComponentName	the component that the request was submitted to	VARCHAR	128
PendingTimeStamp	the time the request was moved to the service queue	DATETIME	8
PendingComponentName	the component that moved the request to the service queue	VARCHAR	128
ProcessingTimeStamp	the time the request began processing	DATETIME	8
ProcessingComponentName	the component that processed the request	VARCHAR	128
ProcessingAttemptCount	the number of times that the system tried to process the request	NUMBER	4
CompletedTimeStamp	the time the request completed processing	DATETIME	8
Component	the name of the component that produced the Request Log message	VARCHAR	128
StatusCode	the foreign key of the StatusOrder table	NUMBER	4

Note: Any number of additional fields can be added to the request table to permit an independent record of a Custom Service Request JMS Message header property.

The Status Table

This Table contains records only for requests that have the following status:

- **Deferred:** the request awaits execution within a specified interval between Deferral and Expiration Time
- **Resubmitted:** the request has been resubmitted for processing
- **Paused**: the request has been paused by the system and is not processed until it is resumed

The `Status` table contains the following fields:

Field Name	Value	Data Type	Size
InternalID	the foreign key of the main table (request)	NUMBER	4
Timestamp	the time the status of the request changed	DATETIME	8

Field Name	Value	Data Type	Size
Component	the name of the component that produced the Request Log message	VARCHAR	128
StatusCode	the Foreign key of the StatusOrder table	NUMBER	4

The ErrorInfo Table

This table contains zero or one record per request. If the request is completed in error, the affected component must log the error information, including the:

- error code
- textual description
- any available extended error information (the Java exception stack in XML format)

Additionally, if the request completes with a status of completed-failure or completed-canceled, the request is saved as it was submitted so that it can be re-submitted at a later point in time.

The `ErrorInfo` table contains the following fields:

Field Name	Value	Data Type	Size
InternalID	the foreign key of the main table (request)	NUMBER	4
ErrorCode	a unique ID code for the error	VARCHAR2	128
ErrorMessage	a message describing the error	VARCHAR2	1500
ErrorInfo	additional error information: consists of an XML representation of the Java exceptions produced by the faulty component.	CLOB	16
OriginalRequest	the entire content of the request	CLOB	16

The StatusOrder Table

The `StatusOrder` table enumerates request status values and also the order in the request processing.

Field Name	Value	Data Type	Size
StatusCode	a unique ID code for the status	NUMBER	4
RequestStatus	a description of the status	VARCHAR	32

Resubmitting a Failed or Canceled Request

The `ResubmitRequest` Admin Command can resubmit a request if it has been completed with failure or if it has been canceled.

The `ResubmitRequest` Admin command uses the specified `OriginalRequestID` or `Selector` to query the Request Log table and uses the logged information regarding the given request to compose a new resubmitted request.

Distribution Requests

A Distribution Request can be resubmitted as a whole request. With the Error Log Level equal to 6, all subrequests of the Distribution Request will be logged into the Request Log table. If a subrequest fails, the body of the subrequest will be logged into the Request Log table, but the subrequest cannot be resubmitted separately from the Distribution Request.

The body of the failed subrequest can be used for debugging purposes only.

Note: If multiple requests have the same RequestID, only those requests that failed will be resubmitted. For example, if two requests with RequestID "ABCD," where one of them failed and one succeeded, only the failed one will be resubmitted. If both requests failed, both will get resubmitted.

Important: A client is responsible for the uniqueness of the RequestID.

Mapping

The mapping between the JMS header fields and the Request Log Data Base fields is configurable:

- The Resubmitted Request is considered a new request with an OriginalRequestID correlated to the RequestID of the original failed request.
- The InternalRequestID, as well as AggregateID if applicable, is assigned by the IStream Publisher component that resubmits a request. IStream Publisher uses UUIDs (Universal Unique Identifier).
- IStream Publisher assigns a unique InternalRequestID when the newly composed request is submitted to a Service Queue.
- JMSReplyTo defines a Response Queue for the response messages. It is a configurable parameter.
- The name of the Queue Set.

Error Log Levels

You can configure the `LogLevel` parameter of the JMSType message to control the amount of logging that takes place. Because the amount of logging occurring affects performance, this setting should be carefully monitored.

You can add the following JMS header property to all functional requests:

```
LogLevel N
```

where N is one of:

- 0 – No logging. No RequestLog messages are sent to the RequestLog.
- 1,2,3,4 – log `pending`, `processing` and `completed` for Distribution Requests but nothing for generated Simple Requests. These values are required for backward compatibility.
- 5,6 – log all messages

Important: A `LogLevel` setting of “4” is the default for Distribution Requests, and “6” for all other types.

Chapter 5

Creating and Using Cover Pages

This chapter describes how users can create and use custom cover pages before distribution to fax or printer. This feature is implemented using event handlers and the Count Pages Service.

Delivering Cover Pages to Fax and Printer

Event handler definitions must be added to the Distribution Request in this format:

```
<distribution-request>
   … usual distribution request parameters …
  <event-handlers>
    <event>
        <recipient-package-ready
recipientPackageRefID="package id"/>
        <event-handler serviceType="count-pages"
seqNumber="1"/>
        <event-handler serviceType="generate-calligo-
document" seqNumber="2">
            <event-handler-metadata>
                <generate-calligo-document>
                    <calligo-source
UISR="is_name:CoverPage.cms" docType="cms"/>
                    <key-data name="numberOfPages"
type="numeric"
                        value="{numberOfPages}"/>
                </generate-calligo-document>
            </event-handler-metadata>
        </event-handler>
        <event-handler serviceType="render-Word-to-PCL"
seqNumber="3"/>
        <event-handler serviceType="concatenate-pcl"
seqNumber="4"/>
    </event>
  </event-handlers>
</distribution-request>
```

Based on the preceding event handler definitions added to the Distribution Request, the cover page creation follows these steps:

1. The Distribution Service processes the Distribution Request as usual, until the preparation of all parts (files) of the Distribution Package (`<recipient-package-ready RecipientPackageRefID="package id">`) is complete.
2. The Distribution Service invokes the first event handler associated with the `"count-pages"` Simple Service.

Note: This event handler is optional and required only if you want the generated cover page to include and display the total number of pages in all documents in the package. If this step is not used, go to step 6.

3. The first event handler finds all the files in the Recipient Package and sends its URLs to the Count Pages Service.

4. The Count Pages Service counts the pages in all of the source files and sends a response to the first event handler.
5. The first event handler replaces the `{numberOfPages}` placeholder in the parameters of the next `<event-handler>` with an actual value, extracted from the received response.
6. The Distribution Service invokes the second event handler, which prepares a request to the Content Service (`generate-calligo-document`) based on parameters provided in `<event-handler-metadata>`, specifying the destination as a DOC file in a temporary location (that is, the same place where the Distribution Service stores all intermediate files).
7. The Content Service generates a new document (`Cover Page`) based on the parameters received from the event handlers.
8. The second event handler receives a response from the Content Service and adds a new source file recipient package.
9. The Distribution Service invokes the third event handler, which creates a request to the simple Rendering Service to convert the generated Cover Page to a format required for a specific destination. In this example, PCL printer is used as the destination and is calling the render-Word-to-PCL service. For PostScript printers, render-Word-to-PS service should be used. For faxes, render-Word-to-TIFF service is used.

Note: This event handler is optional if the fax server can send Word documents without rendering. If this step is not used, go to step 13. Also, this event handler specifies the destination as a temporary file (that is, in the same location as source, with the file extension changed).

10. The Rendering Service converts the generated Cover Page to the requested format and sends a response to the third event handler.
11. The third event handler removes the generated Cover Page in Microsoft Word format from the Recipient Package and adds a rendered Cover Page to the Recipient Package.
12. The last event handler is optional and required only for delivery to a printer, if the client requires having all parts of the package printed on the same printer as a single job. The type of event handler could be `"concatenate-pcl"` for PCL-compatible printers and `"concatenate-ps"` for PostScript printers.
13. The Distribution Service resumes processing of the prepared package and sends it to the required destination.

More parameters can be added in the `<generate-calligo-document>` element if the client needs to provide additional data for cover page generation. If the generated Cover Page does not include the number of pages, the client may remove the first event handler and the `<key-data …/>` element from the generation request.

Important: You must have `seqNumber` attributes in all `<event-handler …>` elements, otherwise the Distribution Service will try to execute all event handlers at the same time, resulting in a failure.

Chapter 6

SDK – The IStream Publisher Client API

The IStream Publisher SDK allows you to extend IStream Publisher, control its operation, and automate requests.

This guide helps system administrators use the IStream Publisher SDK and its associated components.

This chapter describes:

- *The IStream Publisher Client API* on page 96
- *Client API Interfaces* on page 97
- *IStream Publisher Client Exceptions* on page 103

This chapter includes description on how to use the Client API, IStream Publisher client exceptions, and IStream Publisher client support classes.

The IStream Publisher Client API

The IStream Publisher Client API enables Java-based client applications to interact with and use the functionality of IStream Publisher. Semantically, and in terms of functionality offered, this API is no different than the XML interface for invoking requests.

Specific implementations will be provided by the different versions of the IStream Publisher Client. The differences between various IStream Publisher implementations consist of:

- the method used by the client to communicate with IStream Publisher
- the representation of the requests and responses in the invocation of services

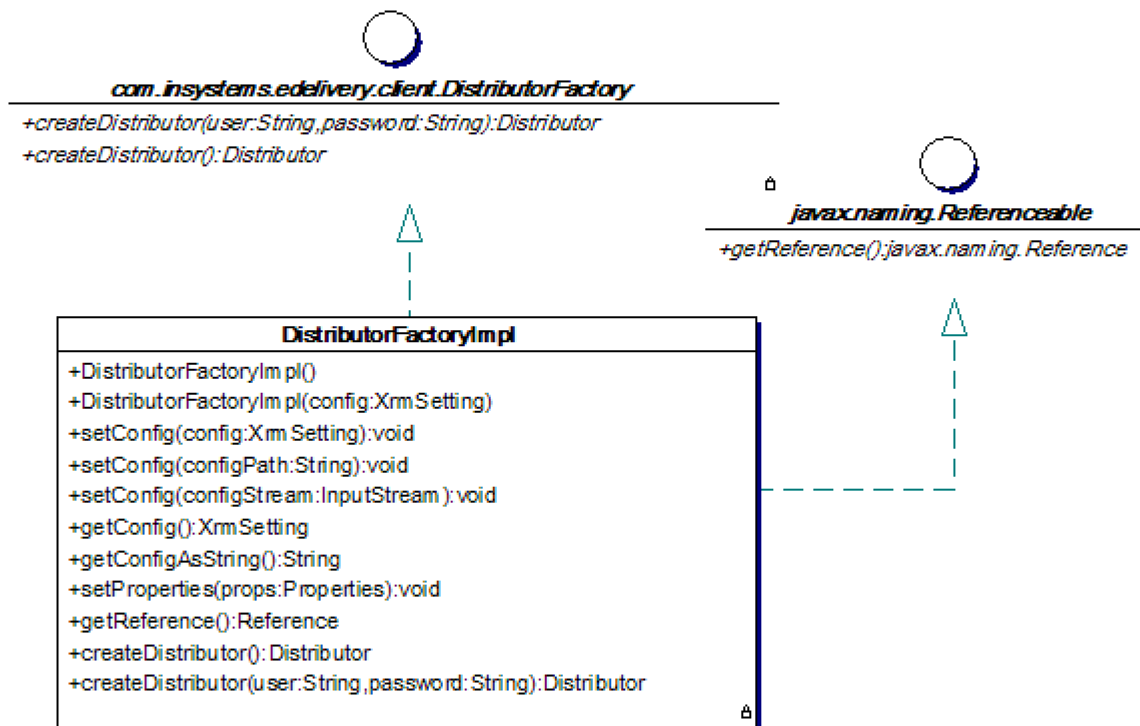
Initially, the default implementation will use both XML messages and serialized Java objects over message queues. The `Service` interface has two distinct versions of the `run` method, one that deals with requests and responses in the form of XML strings and the other that uses Java objects.

Client API Interfaces

The IStream Publisher Client API consists of the interfaces with which the client application must interact in order to access IStream Publisher functionality. The interfaces are contained in a package named `com.insystems.edelivery.client`. Implementations of the client API are contained in sub-packages that are named so as to reflect the characteristics of the implementation. In general, client applications need not be aware of the implementations.

Distributor Factory

To access the IStream Publisher functionality an application must first obtain an instance of a class that implements the `Distributor` interface. It obtains this from a factory class that exposes the `DistributorFactory` interface.



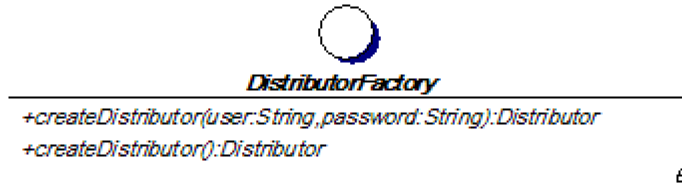
Creating an Instance of DistributorFactory

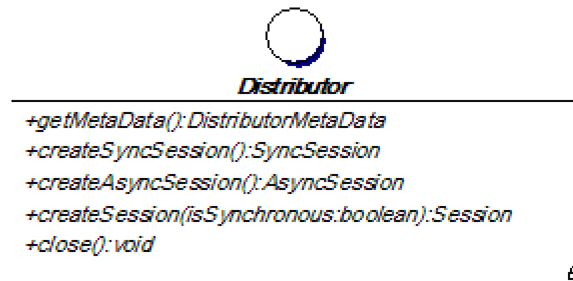
You can create an instance of DistributorFactory in different ways. You can create an instance directly when the client application knows the name of the class that implements the `DistributorFactory` interface.

When the client creates the instance directly it typically passes some configuration information in the form of an instance of the `XrmSetting` class. With the default implementation of the IStream Publisher client, this configuration information is

passed as a parameter of the constructor of the `DistributorFactoryImpl` class.

The DistributorFactory Interface





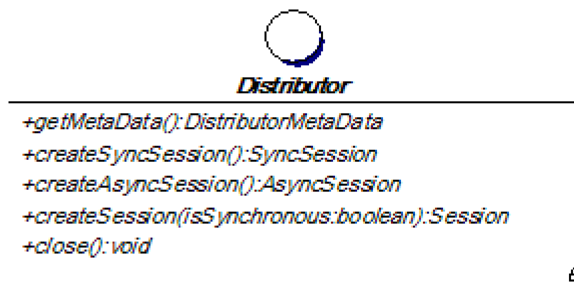
The client uses on of the two versions of the createDistributor methods to obtain an instance of the class that implements the Distributor interface.

The credentials that can be passed with one of the create methods are meant to authenticate the client application or the end user that this application impersonates, to the IStream Publisher system. Credentials that are used by the IStream Publisher client to establish connections for communication purposes should be passed as configuration information to the Distributor Factory class.

Distributor

The distributor interface is the main contract between the client application and the IStream Publisher client. The class implementing the Distributor interface is responsible for—

- allocating the resources necessary for the communication with IStream Publisher.
- initiating and managing the interaction sessions.
- freeing the allocated resources when the client wants to end the interaction.

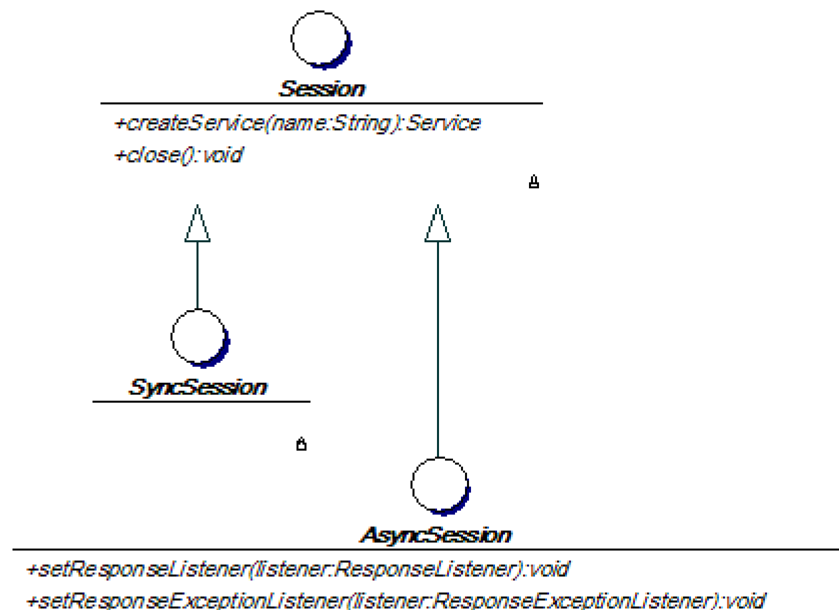


Session

A session represents a single-threaded interaction between the client application and the IStream Publisher Client. The client application creates a distributor session via the createSession method of the Distributor interface.

When creating a session, the client application must specify whether it is synchronous or asynchronous. Depending on the type of session requested, the distributor class returns either a session instance that implements the SyncSession interface for synchronous sessions or the AsyncSession interface for asynchronous sessions. The service instances created by the session will inherit the corresponding synchronous or asynchronous behavior of its higher level session.

The Session Interface



Through the Session interface the session class instance can be

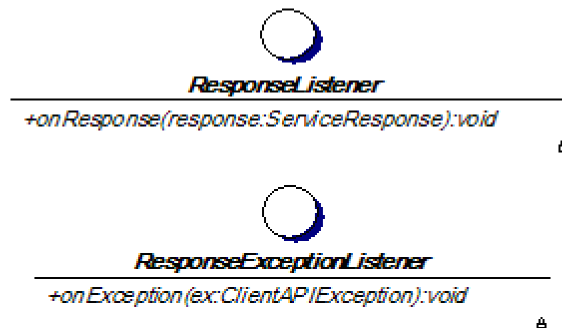
- used to create instances of services.

- closed so that any resources allocated for the communication session can be freed.
- set up with a response listener, which the session will call when a response arrives asynchronously. This is only available for the AsyncSession interface only.
- set up with an exception listener, which the session will can when an exception occurs while receiving a response asynchronously. Also this is only available for the AsyncSession interface.

To create a service from a session, the client application must know in advance the name of the service it wants to use.

With asynchronous sessions, the client application can register a response listener to receive the response object when in becomes available. Also it can register an exception listener to be notified of any exception that occurs during the asynchronous receiving of a response.

The ResponseListener and ResponseExceptionListener Interfaces



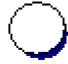
The client application is responsible for implementing the ResponseListener and ResponseExceptionListener interfaces and process the response or exception object passed back by the session.

Services

To work with a service, invoke it and receive a response, the client application must first create an instance of a class that implements the Service interface, via a call to the createService method of the Session interface. When invoking this method the client application must provide the name of the service it wants to create.

Through the Service interface, the service implementation class can be used to

- create instances of request classes that implement the ServiceRequest interface (this feature is deprecated as of IStream Publisher 3.2)
- run a service



Service

+getServiceMetaData():ServiceMetaData
+createRequest():ServiceRequest
+run(request:String):String
+run(request:ServiceRequest):ServiceResponse

The run method behaves differently depending on what type of session has been used to create the service instance. If a `SyncSession` has been used then the run method only returns when the processing of the service completes or it throws either a `ServiceException` or other runtime exception.

If an `AsyncSession` has been used to create the service instance, the run method returns immediately and the client application will receive a response via the `ResponseListener` interface of the listener object it has registered with the session. If no listener object has been registered, the IStream Publisher Client will assume that the client application is not interested in the response and it will not produce one.

Service Invocation Sequence

The following code fragment represents the sequence of calls the client application must perform in a typical IStream Publisher service invocation.

```
Distributor distributor = null;
Session session = null;
try
{
    DistributorFactory factory = ...
    distributor = factory.createDistributor();
    session = distributor.createSyncSession();
    Service service = session.createService(...);
    String request = "<!-- any IStream Publisher XML request
-->";
    String response = service.run(request);
}
catch(Exception ex)
{
    ...
}
finally
{
    if (null != session)
    {
        session.close();
    }
    if (null != distributor)
    {
        distributor.close();
```

```
        }
    }
```

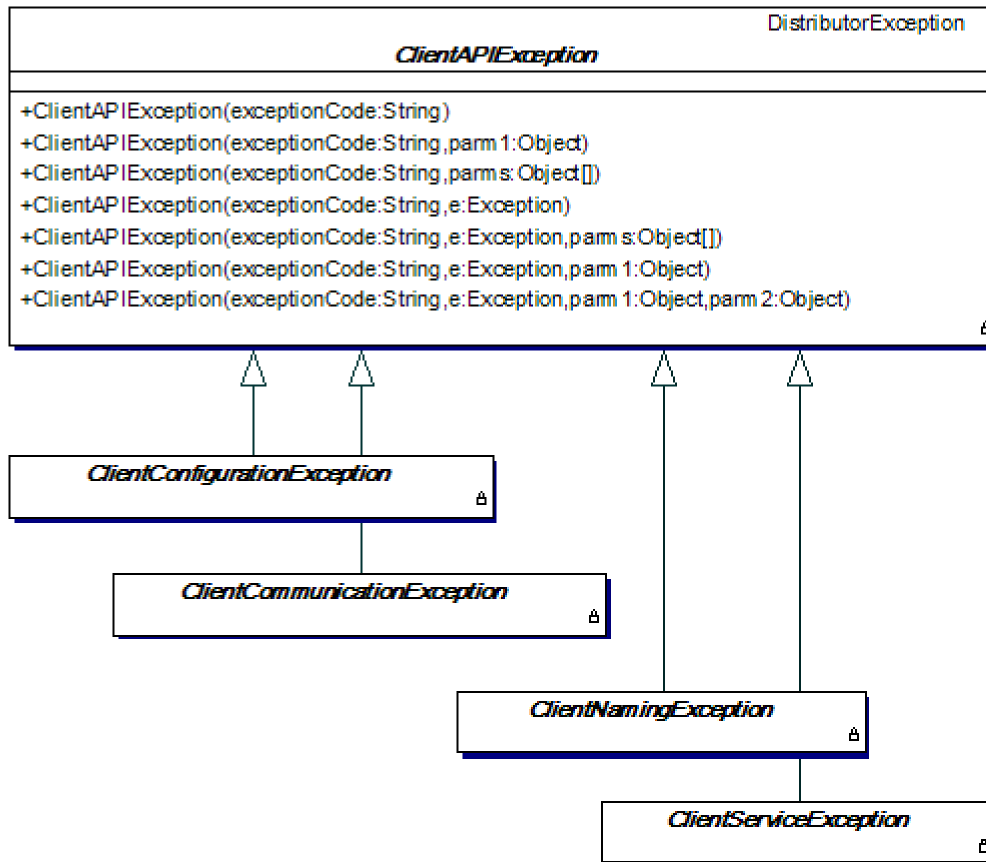
IStream Publisher Client Exceptions

There are four categories of exceptions produced by the IStream Publisher Client API:

- **Configuration exceptions** – thrown when invalid configuration was provided. The invalid configuration could consist of parsing errors, missing mandatory properties or other inconsistencies.
- **Communication exceptions** – thrown when communication problems are encountered between the client and the server. The exact types of problems are implementation dependent and are specific to the transport used to propagate the request from the client to the server.
- **Naming exceptions** – thrown when a JNDI naming exception occurs.
- **Service exceptions** – thrown by services when exceptional situations occur in the processing of the requests. These are also specific to the implementation of each service.

The `ClientAPIException` class is the base class for all the exceptions thrown by the IStream Publisher Client API. This can be used as a catchall when no special treatment is required for each individual type of exception.

The exception objects contain a textual description of the problem and possibly they can wrap another linked exception.



Configuring the IStream Publisher Client API

The IStream Publisher Client API should be properly configured before it is used. These groups of parameters are available for configuration:

- **adaptor** – used to define requests, and optionally reply queue settings, which are used to pass IStream Publisher requests as JMS messages
- **security settings** – used to encrypt a transmitted user name and password

Configuration Files

The following configuration files are provided with the IStream Publisher Client API:

- Client.xml
- ClientJMSQueues.xml
- ClientSecurity.xml

These files are stored in the `config` directory of the IStream Publisher Client API SDK installation. Any XML configuration files used in the IStream Publisher Client API must contain parameters with values that are defined either explicitly

(for example, `mq_port="1414"`) or through macros. The format for a macro is similar to `${macro_name}`.

A macro is resolved when configuration files are loaded, based on the system property value for the macro. If the system property is not specified, but the default value is specified for the macro, then the default value is used. The format for a macro definition with a default value is similar to `${mq_port|1414}`.

Client.xml

The Client.xml file provides links to other configuration files that contain real parameter values. This file provides the following group parameters:

adaptor

```
<adaptor xlink:href="ClienJMSQueues.xml"/>
```

ClientJMSQueues.xml

This file contains an adaptor group of parameters that are used as the setting for Queue Manager, which is used for transmitting messages from the client to IStream Publisher and back. Most of the parameters are defined through macros and therefore should be resolved by the system properties.

The following table list the parameters: most of these parameters are for IBM WebSphere MQ only:

Parameter Name	Description	Default Value	Notes
mq_name	Name of MQ manager		**For IBM WebSphere MQ only.** Should be consistent with mq_name in the IStream Publisher configuration.
mq_domain	Domain name of MQ manager		**For IBM WebSphere MQ only.** Should be consistent with mq_domain in the IStream Publisher configuration.
mq_port	Port number of MQ manager	1414	**For IBM WebSphere MQ only.** Should be consistent with mq_port in the IStream Publisher configuration.
mq_requestqueue	Name of queue to send requests		**For IBM WebSphere MQ and OpenJMS.** Should be consistent with submitting queue of IStream Publisher
mq_responsequeue	The reply queue name.		**For IBM WebSphere MQ and OpenJMS.** If the response queue is undefined, then a temporary queue is created.
mq_user	User name for MQ manager		**For IBM WebSphere MQ only.**
mq_password	Password for MQ manager user		**For IBM WebSphere MQ only.**

ClientSecurity.xml

Some client requests can contain confidential information such as a user name and password. To secure that information, the IStream Publisher Client API provides the ability to encrypt it. Configuration for the security setting is provided in the ClientSecurity.xml file, which is referred to by a link from Client.xml.

```
<security-settings xlink:href="ClientSecurity.xml"/>
```

Note: Before using this feature, make sure to remove comments in the Client.xml file to make `<security-settings>` available.

The ClientSecurity.xml file contains a list of encryption keys for different encryption algorithms.

Important: The IStream Publisher Client API security setting should be consistent with IStream Publisher security settings. The following is an example of security settings:

```
<key-data algorithm="DES" keyName="testKeyDES">
   ... encrypted key data is placed here ...
</key-data>
```

Logging

The IStream Publisher Client API provides the ability to customize logging output. The logging utility uses the log4j tool. Complete documentation for `log4j` is at `http://logging.apache.org`.

Default Configuration Files

The IStream Publisher Client API provides default configuration files. Those files can be used in some simple cases, since they limit the number of configurable parameters.

Configuration Implementation

To configure the IStream Publisher Client API, you edit the configuration files (see *Configuration Files* on page 104), if needed, and provide configuration parameters through system properties.

Implementing the configuration in code involves passing data from a configuration file to the `DistributorFactoryImpl` class. This class has a group of `setConfig...()` methods, which are used to process configuration data.

The following examples are the most useful of these:

1. Using the configuration file path.

You can use a fully qualified path to the configuration file. Since Client.xml is considered the main configuration file, provide a path to that file...

```
DistributorFactoryImpl distFact = new
DistributorFactoryImpl();
```

```
distFact.setConfig("<ed_client_path>\\config\\Client.xm
l");
```

...where `<ed_client_path>` represents the path where the IStream Publisher Client API is located.

2. Using a package name.

   You can use a package name (in effect, the name of directory where all configuration files are located). Make sure that the path of the directory containing the configuration files is included in the CLASSPATH of the application. For example, if the IStream Publisher Client API is located in `C:\EDClientAPI` and all the configuration files are in a `config` subdirectory then you must write the following code:

```
DistributorFactoryImpl distFact = new
DistributorFactoryImpl();

distFact.setConfigPackageName("/config");
```

Note: ensure that a forward slash character (/) is placed before the directory name.

When you run the application you must specify the classpath:

```
java -classpath C:\EDClientAPI <app_name>
```

3. Using the default configuration files.

   In some simple situations you can use the default configuration files (see *Default Configuration Files* on page 106) by writing the following code:

```
DistributorFactoryImpl distFact = new
DistributorFactoryImpl();

distFact.setDefaultConfig();
```

To complete configuration of the IStream Publisher Client API you must pass the system properties to resolve the parameter-macros.

1. Passing parameters in the command line.

   Use the following form:

```
java <app_name> -Dparam1=value1 -Dparam2=value2 ...
```

2. Setting system properties in Client.xml configuration file.

   Use the `<system-property>` element. Refer to *Client.xml* on page 105 for details.

3. Passing parameters through a custom properties file.

   It is acceptable to provide parameters in a custom properties file. However, you must ensure those parameters are moved into the system properties before configuring the `DistributorFactoryImpl` class.

   The following is a sample of code used to do this:

```
//load custom properties

Properties customProperties = new Properties();

...

//move custom properties into system properties
```

```
DistributorFactoryImpl.setConfig( customProperties );
//configure DistributorFactoryImpl
DistributorFactoryImpl distFact =
new DistributorFactoryImpl();
distFact.setConfigPackageName("/config");
```

Notification of Request Completion

The IStream Publisher Client API supports an asynchronous response processing mechanism for clients running in Application Servers. The client application must use the Distributor AsyncSession (see *Session* on page 99) and provide `Reply Queue` settings in the Distributor configuration.

Sample Distributor Configuration

```
<adaptor>
  <property key="RequestQueueFactory.class"
    value="com.ibm.mq.jms.MQQueueConnectionFactory"/>
  <property key="RequestQueueFactory.init.setQueueManager"
    value="QM_SomeQueueManager"/>
  <property key="RequestQueueFactory.init.setHostName"
    value="someHost"/>
  <property key="RequestQueueFactory.init.setPort"
    value="1414"/>
  <property key="RequestQueueFactory.init.setTransportType"
    value="@com.ibm.mq.jms.JMSC.MQJMS_TP_CLIENT_MQ_TCPIP"/>
  <property key="RequestQueue.name"
    value="SUBMISSION.QUEUE"/>
  <property key="ReplyQueue.name"
    value="CLIENT.REPLY.QUEUE"/>
</adaptor>
```

In the preceding example, the queue `CLIENT.REPLY.QUEUE` should be created and used only for the purpose of notifying the IStream Publisher Client about a Distribution Request completion.

IStream Publisher Client will send requests to the Submission Queue with the `JMSReplyTo` property set to the actual Reply Queue name.

Chapter 7

SDK – Repository API

This chapter describes:

- *The Repository API* on page 110
- *The API Architecture* on page 111
- *Reference Language* on page 113
- *The Connection Interface* on page 116
- *The Repository Interface* on page 119
- *Repository Objects* on page 121
- *Identifiers* on page 125
- *Adding a New Repository Adapter* on page 129

The Repository API

The Repository API (Application Programming Interface) is a set of Java interfaces and classes that abstract the operations that can be performed with a Repository. The API is intended to be generic and offer the functionality otherwise provided by most of the existing repositories. It is designed in such a way that it decouples the client from the actual Repository being used. It also permits the client to interchange similar repositories with similar schema, with minimal or no impact on the code.

The API Architecture

The API consists largely of interfaces and defines the contract between the Repository and the client accessing its functionality. There are a few abstract classes provided for convenience only.

An implementation of the Repository API interfaces is called a Repository Adapter. An adapter must completely implement all interfaces but this does not mean that it has to support functionality that its underlying Repository does not support.

Determining Supported Functionality

When the native Repository is limited in functionality, the API provides methods that a client can call to determine the level of functionality that the Repository supports. For example, the client may need to know whether the Repository has support for versions or for renditions of objects. Those methods can only be used to determine the functionality supported by the underlying Repository, not by the adapter. When the adapter doesn't support a feature, it responds with an exception - specifically, `java.lang.UnsupportedOperationException`.

Decoupling the Client

The client application is decoupled from the Repository by the Repository adapter implementation. All object instances are created by factories and factory methods. The only class that the client must instantiate directly is the class implementing the `ConnectionFactory` interface. Even this class can be obtained from a directory or naming service. As a result, the client and the implementation are completely independent of each other.

Accessing Repository Objects

In addition to the proprietary API, the Repository adapter extends Java's URLStreamHandler and URLConnection to allow client applications to access a Repository URI (Universal Resource Identifier) through Java's URL class. The client application uses the URL class to access Repository objects through the `getContent()` method.

Categories of Functionality

The functionality of the API can be divided into categories as follows:

- **Connection** – provides the means to authenticate, configure, and connect to the Repository.
- **Repository** – exposes Repository metadata and functionality. For example, it can create objects, find objects, and so on
- **Repository Object** – accesses the contents of the Repository with their versions and renditions.

Each of these categories is described in greater detail, starting with *The Connection Interface* on page 116. However, before we discuss these categories of functionality we must clarify some terms relating to the URI (Uniform Resource Identifier) specification.

Reference Language

The Repository API defines a reference language through which objects in repositories can be named and identified. This is accomplished using a common syntax, independent of the underlying Repository. The API also defines a way to map common existing URLs so that they can be expressed in the same reference language. As a result, the contents of the repositories to which they refer can be accessed through the Repository API.

The reference language is based on the URI specification.

Uniform Resource Identifiers

A Uniform Resource Identifier (URI) is a string identifier that allows for a resource to be named and located using its name or some other attributes by following a set of syntactic conventions. For a more detailed description of URIs see http://www.ietf.org/rfc/rfc2396.txt

The URI syntax is dependent upon a scheme that defines a namespace for the URI. The URI is created based on that scheme. In general, an absolute URI has the following structure:

```
absolute-URI= scheme ":" scheme-specific-part
```

As shown above, an absolute URI contains:

- The name of the scheme being used (scheme)
- Followed by a colon (":")
- Followed by a string (the scheme-specific-part) whose interpretation depends on the scheme.

Scheme Name

The name of the scheme is used to determine the Repository adapter that will be used. The URI class implements the specific methods of resolving the mapping of the scheme name to a Connection Factory class. Assisted by the connection class, the URI can parse the URI string into its components. (See *Examples of URIs* on page 115.)

The scheme-specific part for a typical hierarchical scheme consists of the following components:

```
scheme-specific-part = "//" context [absolute-path] ["?"
  query]
```

The double slash at the beginning indicates a hierarchical structure.

Context

The top hierarchical element constitutes the context (or naming authority) that governs the rest of the name. The context component consists of the user information and the Repository.

```
context = [ [ user [ ":" password ] "@" ] Repository
```

The **user information** is optional and provides for syntactic compatibility with existing URLs. Because of the security risks involved, its use should be discouraged.

The **Repository** part of the name consists of the name and other attributes that the Repository adapter can use to locate and connect to the Repository on a network. In its shortest form it should contain at least a Repository name and a host name in the following form:

```
repository = [repository-name ["!" properties] ";"] host
[":" port]
```

Important: The host name cannot start with a number.

The **repository-name** is the first optional component. It is normally used for existing schemes that do not permit a Repository name and for cases when there is only one Repository possible. In these cases, the host or the port number can be used to differentiate between instances.

```
repository-name= name
name= alpha | alpha ( alpha-num | "-" )* alpha-num
```

The **properties** component of the repository name is also optional. It represents a list of named values specific to each particular type of Repository. Typically, a client stores the properties as configuration parameters with the class that implements the `ConnectionFactory` interface. This class must be capable of retrieving the parameters and configuring the connection instance that it creates.

```
properties= property | properties "," property
property= name "=" value
value= ( alpha-num | unreserved | escaped )*
```

The **host** is a domain name of a network host or its IP address. IP addresses in normal use today appear as a set of four decimal digit groups separated by ".".

```
host= host-name | IP-address
host-name= ( domain-label "." )* top-label [ "." ]
domain-label= alpha-num | alpha-num ( alpha-num | "-" )*
  alpha-num
top-label= name
IP-address= 1 digit* "." 1 digit* "." 1 digit* "." 1 digit*
port= digit*
```

The **port** is a network port number. Most schemes that use a port designate a default value. A port number can optionally be supplied following the host and separated by a colon (":"). If the port is omitted the scheme default value is assumed.

Path

The **path** component contains data specific to the context, or the scheme if no context is present. It identifies the resource within the scope of that scheme or context. Paths can be either absolute or relative. A relative path can be relative to the root of the Repository or relative to another object in the Repository.

```
path = absolute-path | relative-path
absolute-path= "/" [ relative-path ]
relative-path= path-step | relative-path "/" path-step
path-step= [ relationship "::" role "@" ]  (   abbrev-
attribute | attributes )
relationship= name
abbrev-attribute= name
attributes= attribute | attributes "," attribute
attribute= name "=" value
```

A **path-step** is equivalent to navigating between two object instances in the Repository object model. The navigation happens over a specific relationship to a role. In the Repository object model, when there is only one relationship between the two objects, the name of the relationship and the role can be omitted.

An abbreviated attribute is one for which only the value of the attribute is specified but not the name. Paths can contain one abbreviated attribute (the first one) in cases when the name of the object is the identifying attribute.

Query

The **query** part of the URI represents a set of named values separated by "&". These values do not determine the location of the resource identified by the URI but are usually interpreted as arguments to a functional invocation referring to the URI.

```
query = arguments
arguments= argument | arguments "&" argument
argument= name "=" value
```

Queries can be used to refer to a specific version and/or rendition of the object.

```
version=2&type=text.plain
```

The query above refers to the second version of the object and to its plain text rendition.

Examples of URIs

Here are a few examples of URIs formed using the reference language defined by the Repository API:

- `ftp://guest:password@company.com/policies/renewal.doc`
- `http://www.company.com/index.html?user=guest&password=test`
- `file://user/setup/Adobe/Acrobat/viewer.exe`
- `file:///C:/infosourceFS/ds/source/filename.doc`

The Connection Interface

The `Connection` interface is the abstraction of the connection that a client must establish with the Repository before getting access to its content and functionality. (See the “Categories of Functionality” listed on 112). The connection:

- Maintains the configuration parameters required to access the Repository. Examples of these parameters are the protocol, the host name, the Repository name, and so on.
- Authenticates the user (the client) to the Repository.
- Manages the resources required to communicate with (connect to) the Repository. Examples of these resources are sockets, database connections, and so on. The connection also manages the lifetime of the connection itself (opening and closing).
- Registers listeners to receive notifications about changes to objects in the Repository.

Connection Factory

A client uses a class factory to instantiate a connection. (See *Creating an Instance of DistributorFactory* on page 97.) The Repository adapter must provide a class factory for the connection class that implements the `ConnectionFactory` interface. The Connection Factory class must also implement the `javax.naming.Referenceable` and `java.io.Serializable` interfaces so that it can be stored in a directory service.

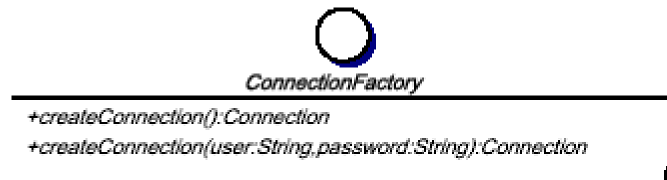
Required Properties

There are two mechanisms that a client must use, **in this order**, to locate and instantiate a Connection Factory:

1. The environment property `com.insystems.repository.scheme.connection.factory.class` must be set with the name of the class that implements the `ConnectionFactory` interface. The client uses this class name to create an instance of the Connection Factory class. The class factory itself uses a properties file to store its initialization parameters and reads them at creation time. The client can alter these parameters or place another properties file in the class path. In that case, the new properties file will get loaded before the default one.
2. If the name of the Connection Factory property is not present, the client should look for the environment property `com.insystems.repository.scheme.connection.factory.name`, which must be set with the JNDI name of the Connection Factory class. The client uses this name to retrieve an instance of the Connection Factory class from the directory service. In this case, the initialization parameters must be stored in the directory service together with the factory.

If neither of these properties is set, then **the operation will fail** and no Connection Factory class is instantiated.

The scheme in the two environment properties above is the name of the scheme (ftp, xrmrep, and so on) used by the reference language to identify an object in a Repository.



Important: The class extending `java.net.URLStreamHandler` is also a class factory for the class implementing the `Connection` interface. **Recommendation**: The class extending `URLStreamHandler` should also implement `ConnectionFactory`.

Creating a Connection

A client creates a connection in order to get access to the class that implements the `Repository` interface, and by doing so to have access to the Repository.

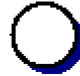
Because the client instantiates the connection objects through a Connection Factory, the factory has control of the way in which the connection instances are created. For example, the Connection Factory class can provide instance pooling to reduce the time required to connect to the Repository and thus reduce the overall time required to connect.

Multithreading

A client should be allowed to open as many connections with the Repository as its programming model requires. To fit a programming model where the client reuses a single connection instance on multiple threads, the connection objects must support concurrent use.

Opening the Connection

A connection object when instantiated represents an inactive connection to the Repository. The client must make an explicit call to open the connection. This allows the client to alter the properties that control the way in which the connection with the Repository is established (`setProperties()`).



Connection

+getConnectionProperties():Hashtable
+setConnectionProperties(properties:Hashtable):void
+open():void
+close():void
+getRepository():Repository
+getRepository(schemaName:String,schemaVer:String):Repository
+getEventListener(url:URL):EventListener
+addEventListener(listener:EventListener,url:URL):void
+removeEventListener(listener:EventListener,url:URL):void

The `getRepository()` method can be invoked only on an open connection. If the connection hasn't been opened yet or if it has been closed, the method throws an exception, namely the `InvalidConnectionStateException` exception.

The Repository Interface

The `Repository` interface abstracts the operations that can be performed against the Repository itself. It is the interface through which:

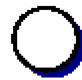
- the Repository objects can be manipulated (created and destroyed)
- the client can access the root object(s) of the Repository
- the client can query the Repository for objects based on object metadata
- the client can check the permissions to perform various Repository operations

Through this interface a client can get the naming context (URL) of the Repository. It can also retrieve the connection from which the Repository instance was obtained.

All the functionality of the `Repository` interface is available when an opened connection exists to the Repository. If the connection corresponding to the Repository object is closed, its methods will throw an `InvalidConnectionStateException` exception.

A client obtains an instance of the class that implements the `Repository` interface by calling the `getRepository()` method on the connection object.

Objects implementing the `Repository` interface are not required to be thread safe. Consequently, a client with a multithreaded programming model should obtain separate instances of the Repository for use on different threads.



Repository

+getConnection():Connection
+getContext():URL
+getSchema():Schema
+createObjectMetadata():ObjectMetadata
+createObjectMetadata(locale:Locale):ObjectMetadata
+getObject(url:URL):RepositoryObject
+createObject(typeId:String,url:URL):RepositoryObject
+createObject(typeId:String,url:URL,metadata:ObjectMetadata):RepositoryObject
+destroyObject(url:URL):void
+getReferenced():Iterator
+getReferenced(relationship:Relationship):Iterator
+getReferenced(relationshipId:String):Iterator
+query(criteria:ObjectMetadata):Iterator
+checkPermission(permision:Permission):void

The `Repository` Interface

When designing the Repository API it was assumed that the implementation for the `Repository` interface would be lightweight enough so that multiple instances

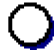
could be created without too much impact on scalability. In contrast, the class implementing the `Connection` interface - because it can reserve external resources - might have a greater impact on scalability. This is the reason for the requirement that the `connection` class supports **multithreading**, in order to conserve resources.

Repository Objects

Repository objects represent the entities stored in the Repository. The Repository object is a wrapper - it is not the data itself. This wrapper adds Repository-specific functionality to the data objects that the client deposits in the Repository. The actual data is treated as content of the Repository objects. Content type is differentiated based on the MIME type of the data. For a list of MIME types, please see http://www.iana.org/assignments/media-types/

The following features of Repository objects are considered common to most of the repositories and are exposed through the `RepositoryObject` interface:

- **Provide a reference language** that uniquely identifies and references objects in the Repository. Objects can be referenced either relative to each other or absolutely in the context of the Repository.
- **Access the actual content** of the object stored in the Repository.
- **Access metadata information** for objects – Metadata consists of name and type information along with any custom properties that an object might define.
- **Maintain versions of an object -** The latest version is the default for the object.
- **Maintain different renditions** for a version of an object. When the object does not have a visual representation of the rendition, it can be considered just a different format of representation for the same information.
- **Reserve an object for the purpose of changing it** and unreserve it so that your changes can be visible to others.
- **Navigate the relationships** the object is involved in with other objects in the Repository. Navigation is possible to both objects that reference and/or are referred by the current object.
- **Alter the relationships in which the object is involved,** with other objects in the Repository.
- Remove the object in the Repository.
- **Discover the features that the Repository supports.** Versions, renditions and effective dates are examples of supported features.
- C**heck a client's permissions** to perform various operations on a particular Repository object.



RepositoryObject

+getURL():URL
+getId():String
+getInputStream():InputStream
+getOutputStream():OutputStream
+getType():Type
+getType(locale:Locale):Type
+getRelationships():Iterator
+getRelationships(locale:Locale):Iterator
+getMetadata():ObjectMetadata
+getMetadata(locale:Locale):ObjectMetadata
+getDefaultContentType():String
+getVersion(versionNo:int):Version
+getVersions():Iterator
+getVersionsCount():int
+addVersion(contentType:String):Version
+addVersion(contentType:String,effective:Date):Version
+removeVersion(versionNo:int):void
+getRendition(contentType:String):Rendition
+getRenditions():Iterator
+getRenditionsCount():int
+getRendition(contentType:String,versionNo:int):Rendition
+getRenditions(versionNo:int):Iterator
+getRenditionsCount(versionNo:int):int
+addRendition(contentType:String):Rendition
+addRendition(contentType:String,versionNo:int):Rendition
+addDefaultRendition(contentType:String):Rendition
+removeRendition(contentType:String):void
+getReferencers():Iterator
+getReferencers(relationshipId:String):Iterator
+addReferencer(relationshipId:String,referencer:URL):void
+removeReferencer(rel:Relationship,referencer:URL):void
+getReferenced():Iterator
+getReferenced(relationshipId:String):Iterator
+addReferenced(relationshipId:String,referenced:URL):void
+removeReferenced(relationshipId:String,referenced:URL):void
+reserve():void
+unreserve():void
+destroy():void
+supportsReserve():boolean
+supportsVersions():boolean
+supportsRenditions():boolean
+supportsEffectiveDate():boolean
+checkPermission(perm:Permission):void

Repository Objects

Note: To access the actual content of the Repository object the client must obtain either the **InputStream** to read the object or the **OutputStream** to write it.

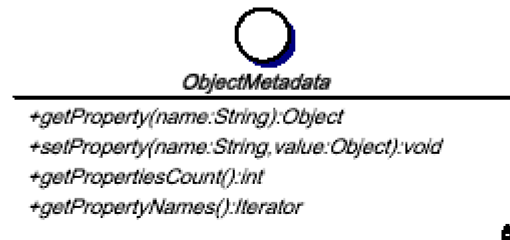
The functionality that the `RepositoryObject` interface exposes regarding metadata, versions, renditions and schema is presented in more detail in the following sections.

Object Metadata

There are two categories of metadata that are available for an object: schema metadata and custom metadata. Schema information for an object is available through specialized methods on the `RepositoryObject` interface. Two examples of these methods are `getId()` and `getType()`.

Object custom metadata is exposed through the `ObjectMetadata` interface. This interface can be accessed using the `getMetadata()` method of the `RepositoryObject` interface.

The object metadata can be localized. Custom properties can be described in either the default locale or a user-specified locale. These properties can be specified as an argument with the `getMetadata()` method of the `RepositoryObject` interface. When a client requests the metadata for a locale that is not available, the `java.util.MissingResourceException` is raised.



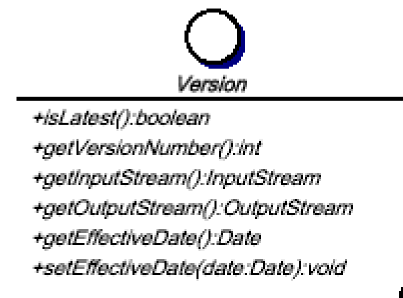
The `ObjectMetadata` Interface

The `ObjectMetadata` interface exposes object custom properties in a generic way, as named properties. If the properties are locale-sensitive, the object metadata must be obtained using the locale-aware `getMetadata()` method of the `RepositoryObject` interface.

Versions

One important feature of a Repository is the ability to retain different versions of an object. Methods of the `RepositoryObject` interface allow access to an object's versions. A client can:

- Obtain the number of versions of an object.
- Enumerate all the versions.
- Access a specific version based on its number.
- Add a new version. It is also possible to specify an effective date.
- Remove a specific version based on its number.



The `Version` Interface

Clients manipulate object versions through the `Version` interface. The object's content for a particular version can be obtained through the input and output streams exposed by this interface.

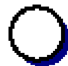
The default version - The content of the default version is the one accessed through the `RepositoryObject` interface. The default version is always the latest version added to an object and cannot explicitly be set to a different version.

Renditions

It is assumed that objects stored in the Repository can potentially have different representations. This is more true for objects that can be rendered visually - for example, documents and images. Object renditions are differentiated based on their content type (their MIME type). To support working with different renditions of an object, the `RepositoryObject` interface allows the user to:

- Obtain the number of different renditions of an object.
- Enumerate all the renditions.
- Access a specific rendition based on its content type.
- Add a new rendition, and potentially specify an effective date.
- Remove a specific rendition based on its content type.

Clients manipulate renditions through the `Rendition` interface. The content for the rendition is accessible through the input and output streams exposed by this interface.



Rendition

+isDefault():boolean
+setDefault():void
+getContentType():String
+getInputStream():InputStream
+getOutputStream():OutputStream

The `Rendition` Interface

The default rendition - The content for the default rendition is the one accessed through the `RepositoryObject` interface. The default rendition can be set either by calling the `addDefaultRendition()` method of the `RepositoryObject` interface or by explicitly invoking the `setDefault()` method on the `Rendition` interface.

Identifiers

The URI identifiers used by the Repository API are implemented by the `java.net.URL` class. In effect it is not the URL class itself that implements the identifiers but rather the classes implemented by the adapter and to which the URL class delegates. The following interfaces must be implemented or the classes must be extended by the Repository adapter in order to support the URL's class functionality:

- The `URLStreamHandlerFactory` interface must be implemented to control the way in which an instance of a URL stream handler class is created.
- The `URLStreamHandler` class must be extended to handle the parsing of the URL string.
- The `URLConnection` class must be extended to handle the connection to the Repository.

URLStreamHandlerFactory

The `URLStreamHandlerFactory` is an interface that the URL class uses to create instances of a URL stream protocol handler that extends the class `URLStreamHandler`.

The Stream Handler factory class can be installed once per JVM (Java Virtual Machine) by invoking the URL's class `setURLStreamHandlerFactory()` method. The Stream Handler factory should be used only if the other methods for instantiating a URL stream protocol handler are not appropriate.

The methods referred to above, that the URL class uses to determine what URL stream protocol handler to instantiate are:

Previous URLStreamHandlerFactory

If the application has previously set up an instance of `URLStreamHandlerFactory` as the stream handler factory, then the `createURLStreamHandler` method of that instance is called with the protocol string as an argument to create the stream protocol handler.

No Previous URLStreamHandlerFactory

If no `URLStreamHandlerFactory` has yet been set up, or if the factory's `createURLStreamHandler` method returns null, then the constructor finds the value of the system property `java.protocol.handler.pkgs`. If the value of that system property is not null, it is interpreted as a list of packages separated by a vertical slash character ('|').

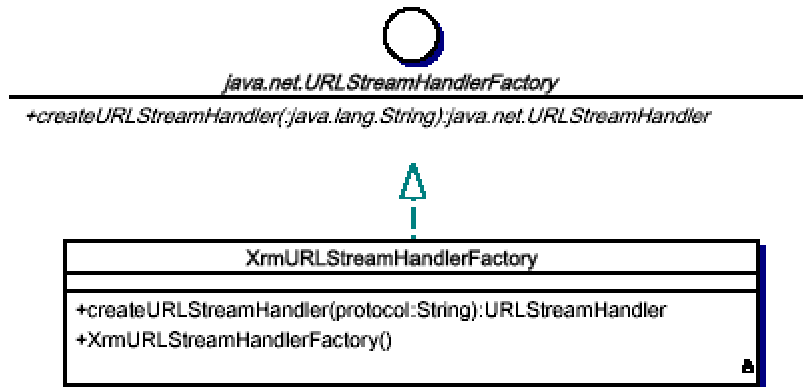
The constructor tries to load the class named `<package>.<protocol>.Handler` where `<package>` is replaced by the name of the package and `<protocol>` is replaced by the name of the protocol. If this class does not exist, or if the class exists but it is not a subclass of `URLStreamHandler`, then the next package in the list is tried.

No Protocol Handler

If the previous step fails to find a protocol handler, then the constructor tries to load the class named `sun.net.www.protocol.<protocol>.Handler`. If this class does not exist, or if the class exists but it is not a subclass of `URLStreamHandler`, then a `MalformedURLException` exception is thrown.

Illustrated below is an example of a class that implements the `URLStreamHandlerFactory`.

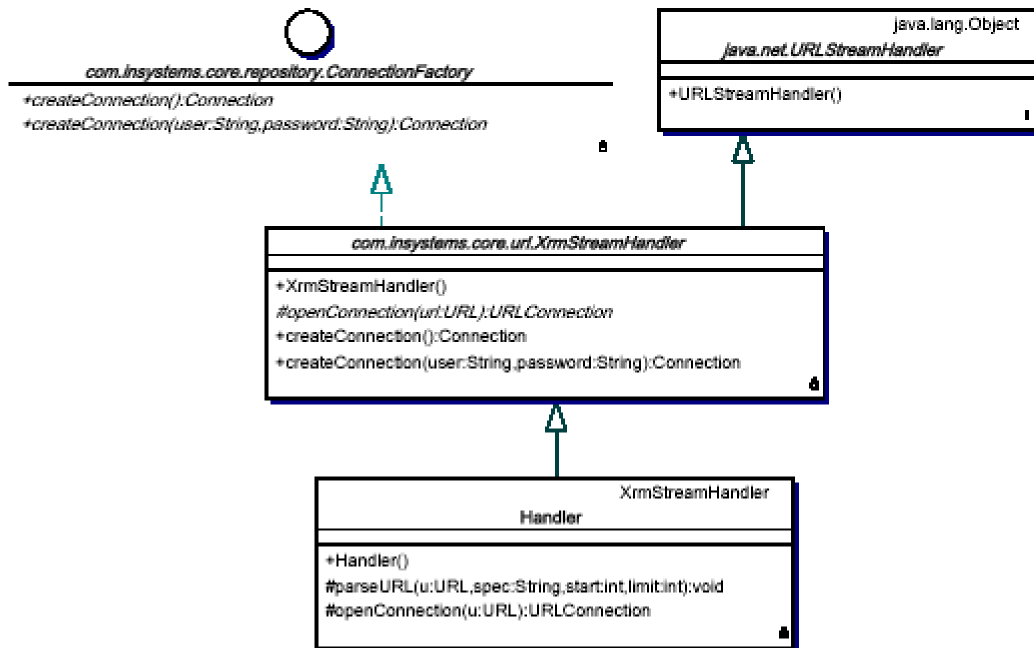
Note: Because there can be only one class per JVM, the factory is expected to exceed the scope of a single adapter. This is the rationale for the name `XRMURLStreamHandlerFactory`.



The `URLStreamHandlerFactory` Class

URLStreamHandler

The abstract class `URLStreamHandler` is the common superclass for all stream protocol handlers. Below is an example of a `URLStreamHandler` class for an IStream repository:



The `URLStreamHandler` Class

The `URLStreamHandler` class has two main responsibilities:

- **Parse the URL string** into its components and set the private fields of the URL class.
- **Create a connection t**o the Repository.

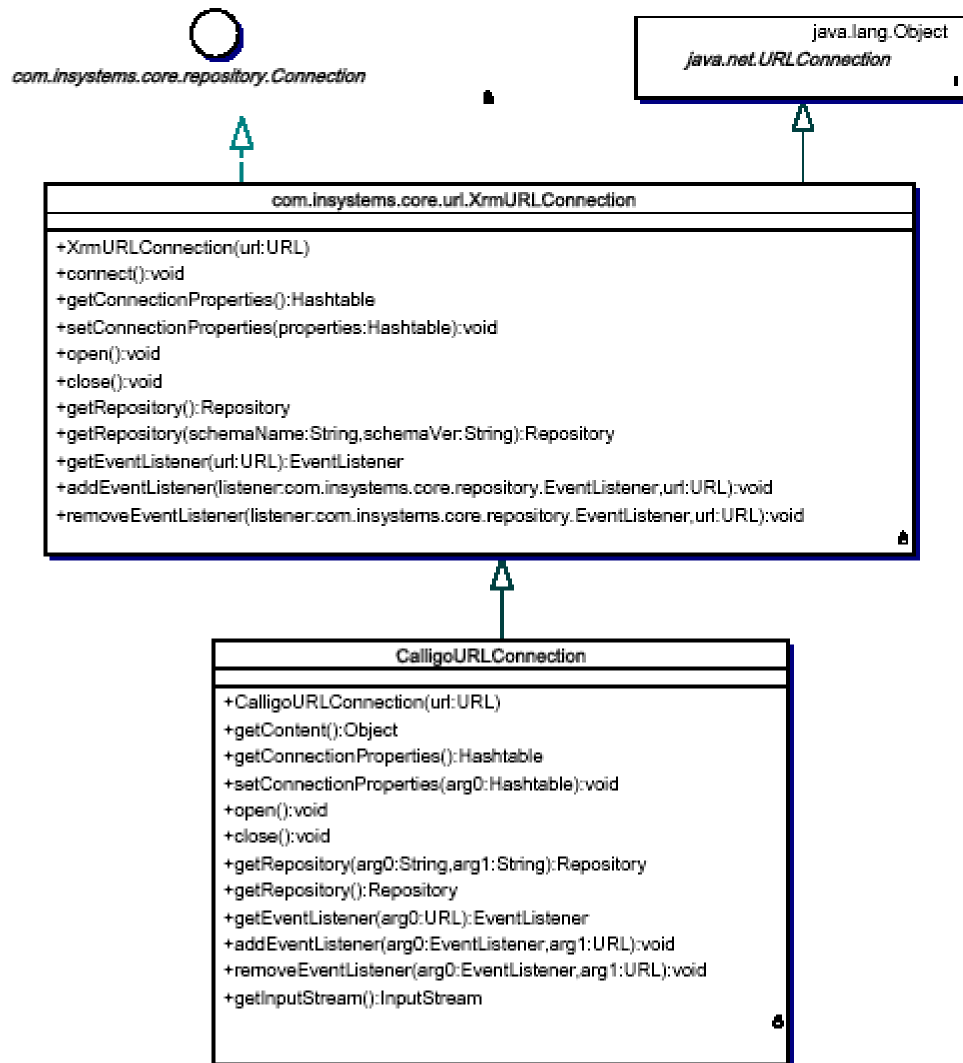
The class extending the `URLStreamHandler` is used to create instances of the class implementing the `Connection` interface. It also extends the `URLConnection` abstract class, therefore it must implement the `ConnectionFactory` interface.

URLConnection

The abstract class `URLConnection` is the superclass of all classes that represent a communications link between the application and a URL. Instances of this class can be used both to read from and to write to the resource referenced by the URL.

Below is an example of a `URLConnection` class for a repository:

A URLConnection Class



Content

To read the content of a Repository object, a client must:

- Create an instance of the URL class using the URI reference of the Repository object.
- Call the `getContent()` method of the URL class. The object returned by this method must implement the `RepositoryObject` interface.
- Obtain the `RepositoryObject` interface and call its `getInputStream()`.
- Read the object from the input stream.

Adding a New Repository Adapter

This section shows how to create a new Repository implementation and add it to IStream Publisher. “Test Repository” is used through as an example.

IStream Publisher uses various Repository API implementations to access objects stored in different locations. A standard installation includes implementations for the local file system, FTP and IStream DMS. In addition, a standard implementation supports any custom Repository implementations, which create interfaces as defined in the Repository API. All existing services (generation, rendering, delivery) will be able to use the new Repository as soon as its implementation is created and added to the system configuration.

IStream Publisher uses the protocol provided in the URL to select a specific Repository API. You can select any protocol name (excluding standard names such as file, FTP, http, and so on.) and use this name in all requests referring to objects in your Repository.

In general, in order to add a new Repository, the following steps are required:

1. Implement Java code specific to the new Repository. This code should expose at least one public class, which is implementing the `ConnectionFactory` interface and has a public constructor without arguments.
2. Create a `com/insystems/repository/url/protocol.properties` file in the JAR file, implementing the new adapter.
3. Add the following line to the file you've created in the previous step:

   ```
   protocol-name = implementation class name
   ```

   For example:

   ```
   myrep=com.acme.repository.MyRepHandler
   ```

4. Add your JAR file implementing the new adapter to the Worker class path.
5. Prepare a new request message according to the source or destination URLs referred to by objects in your Repository.

To illustrate the steps above with examples, specific elements of the IStream Publisher infrastructure related to the tasks described above will be presented and an example of a new Test Repository implementation will be considered in detail. The Test Repository allows you to save and retrieve data to the local file system.

Java Code

The `com.insystems.test.repository` java package consists of the following classes:

- `TestConnectionFactory` – Connection Factory class
  (Creates an instance of `com.insystems.core.repository.Connection`)
- `TestConnection` – connection class
  (Creates an instance of `com.insystems.core.repository.Repository`)
- `TestRepository` – repository class
  (Creates instances of `com.insystems.core.Repository.RepositoryObject`)
- `TestRepositoryObject` – Repository object implementation
  (Creates java.io.InputStream and java.io.OutputStream for specific object in Repository)

Note: The `TestConnectionFactory` class should have a public constructor without parameters.

Code Example

```
package com.insystems.test.repository;
import com.insystems.core.repository.ConnectionFactory;
import com.insystems.core.repository.Connection;
import com.insystems.core.repository.Repository;
import com.insystems.core.repository.RepositoryObject;
import java.io.InputStream;
import java.io.OutputStream;

public class TestConnectionFactory implements
ConnectionFactory
{
    public TestConnectionFactory()
    {
    }

    public Connection createConnection()
    {
        return new TestConnection();
    }

    public Connection createConnection(String user, String
password)
    {
        return new TestConnection();
    }
}
```

```
class TestConnection implements Connection
{
    public TestConnection()
    {
    }
    public Repository getRepository()
    {
        return new TestRepository();
    }
    public Repository getRepository(String schemaName,
String schemaVer)
    {
        return new TestRepository();
    }
    // Other methods defined in Connection interface are not
used.
}

class TestRepository implements Repository
{
    public TestRepository()
    {
    }
    public RepositoryObject getObject(java.net.URL url)
    {
        return new
TestRepositoryObject(url.getFile().substring(1));
    }
    public RepositoryObject createObject(String typeId,
java.net.URL url)
    {
        return new
TestRepositoryObject(url.getFile().substring(1));
    }
    // Other methods defined in Repository interface are not
used.
}

class TestRepositoryObject implements RepositoryObject
{
    java.io.File file;
    public TestRepositoryObject(String filename)
    {
        file = new java.io.File(filename);
    }
    public InputStream getInputStream()
    {
        try
        {
```

```
            return new java.io.FileInputStream(file);
        }
        catch (java.io.FileNotFoundException ex)
        {
            return null;
        }
    }
    public OutputStream getOutputStream()
    {
        try
        {
            return new java.io.FileOutputStream(file);
        }
        catch (java.io.FileNotFoundException ex)
        {
            return null;
        }
    }
    // Other methods defined in RepositoryObject interface
       can be implemented later.
```

Service Request Example

The following codes is an example of a Service Request with the new repository:

```
<?xml version="1.0" encoding="UTF-8"?>
<render-Word-to-PCL>
  <source url="test:///C:/TEMP/test.doc"/>
  <destination url="test:///C:/TEMP/test.pcl"/>
  <output-name>HPLJIII</output-name>
</render-Word-to-PCL>
```

Chapter 8

SDK – System Extensibility

This chapter describes how to create and add a Simple Service using XML message middleware, including the update of your configuration files. It then describes how to extend the Distribution Service by creating Event Handlers to customize your Distribution Request behavior. There is a discussion of Data Access Objects, and finally a procedure for modifying a Request Log message.

This chapter describes:

- *Creating and Adding a Simple Service* on page 134
- *Extending the Distribution Service* on page 137
- *Customizing a Request Log Message* on page 153

Creating and Adding a Simple Service

The list of possible customer services is open-ended. Currently-implemented services include:

- Document rendering in various formats (Word, PDF, HTML, TIFF and PCL).
- Document delivery through multiple channels (e-mail, fax and print).
- Delivery to multiple repositories, for example, FTP server, file system and DMS repositories.

New services may easily be added to your system. However, the relative ease of adding a particular new service depends on how similar this new service is to any already existing services that could be used as a prototype.

Generally, in order to add a new service the following steps are required:

- Update the system configuration, making it aware of the new service.
- Implement Java code specific to the new service.
- Update the system-configure command with the new service.
- Prepare a new request message according to the new message format.

To illustrate each of the above steps, specific elements of the IStream Publisher infrastructure related to the tasks described above will be presented and an example of a new ‘delete-files’ service will be presented in detail.

Note: The ‘delete-files’ service allows the user to delete files and folders provided in a request.

Simple Use Case

Consider the simplest case where a message contains only XML formatted text with a Simple Request (not aggregate):

- When a message from the Message Queue reaches IStream Publisher, it is picked up by one of the Service Managers servicing the given type of request.
- That listener in turn, based on the type of JMS message, looks the type of message up in the configuration file’s `ServiceFactory` class and instantiates it. To be precise, `ServiceFactory` is an interface and an implementation of it is instantiated.
- This `ServiceFactory` creates a service object (the object implementing the Service interface).
- The `ServiceRequest` class is then instantiated. Once again, to be precise, `ServiceRequest` is an abstract class and it is actually one of the derived classes that is instantiated.
- The method `run` of the service is then fed with the Service Request and the result of the method `run` is `ServiceResponse`.

- This `ServiceResponse` is packaged to reply to the message and sent back.

In summary, the service developer must implement service-specific `ServiceFactory, Service, ServiceRequest` and `ServiceResponse` classes. Two of those classes - the `ServiceFactory` and `ServiceRequest` classes - must be defined explicitly in the configuration. See details below.

In the following paragraphs, as the main code pieces are introduced, we will explain the required configuration changes.

Custom Service Deployment

1. Copy the jar file with custom service implementation to the Worker's *[IStream Publisher install]*`\etc\` folder. If you have multiple Workers, copy this file to all systems.
2. Log on to the Admin Console and click **Configuration**.
3. Add new `Service[Custom0]` entity under …`\Settings\Services`.
4. Select the check box near the `maxError` attribute when creating this entity, otherwise the entity may not be create properly.
5. Create a new `Property[factory]` entity under this Service. The **Attribute Name** must be the same as the **JMS Type** attribute of the request message. The **Attribute Value** should contain the Service Factory class name.
6. Create a new `Property[request]` entity. The **Attribute Name** should contain the name of the root element used in XML requests send to the custom service. The **Attribute Value** contains the class name, which extends the `ServiceRequest`.
7. Create a new `Property[response]` entity. The **Attribute Name** should contain the name of the root element used in XML responses received from the custom service. The **Attribute Value** contains the class name, which extends the `ServiceResponse`.
8. Restart the Worker service on all workers.

Sample Structure

For example, assume the following structure for the `delete-files` request:

```
<?xml version="1.0"?>
<delete-files>
  <file>
    <item-source url="ftp://.../Test.doc"
      ContentType="application/msword">
    </item-source>
  </file>
</delete-files>
```

Java Code

A typical implementation of a new Service consists of the following classes:

- Service Factory class,
- Service Class,
- Service Request, and
- Service Response.

All of these classes are usually grouped as a single service package. In our example case of the ‘delete-files’ service, the following files are found in the `com.insystems.distributor.services.cleanup` package:

- `CleanupServiceFactory` - the Service Factory class
- `CleanupService` - the service itself
- `CleanupServiceRequest` - the Service Request
- `CleanupServiceResponse` - the Service Response
- `CleanupFile`, `CleanupFolder`, `CleanupFilter`, and `ItemSource are` all used to map to the proper XML elements.

The `CleanupServiceFactory` service factory implements the `ServiceFactory` interface and has a public default constructor. The Service Factory creates a new instance of the service.

The `CleanupService` implements the `Service` interface and this is where the actual processing happens. It receives `CleanupServiceRequest` in its `run` method and returns `CleanupServiceResponse` when the service is complete.

Extending the Distribution Service

Event Handlers

The Event Handlers structure is a powerful tool to customize Distribution Request behavior. The idea is to provide some hooks for customer services during a Distribution Request execution.

Event Handlers may be called:

- From the Distribution Request when all document generation is complete but before any rendering takes place.
- When the Recipient Package is ready (all Recipient Items have been rendered).
- Immediately before and immediately after delivery.
- When a whole Distribution Request is complete.

Operating Mode

Event Handlers called before and after delivery are processed differently depending on the Delivery Channel operating mode. (For information about the operating modes, see the *Distribution Service* on page 51.) If the operating mode is Synchronized then the Event Handler may be called once when all Recipient Packages/Items for the channel are ready for delivery. After that, another Event Handler may be called when all items have been delivered through the channel.

On the other hand, if the channel acts in Instant mode, then the Event Handler may be called right before and right after any single delivery takes place.

Distribution Request API

Event Handlers may interact with the Distribution Request as they progress through the Distribution Request API which allows the user to fetch some information about the request based on the current scope available to the Event Handler.

In other words, an Event Handler associated for example with a particular Recipient Package may retrieve all details about this package’s Recipient Items based on the Recipient Package ID (and Request ID). Note that most of the information from the Distribution Request available to the Event Handler is in fact read-only. The only part of the Distribution Request that may be updated through Event Handlers is the Delivery Item.

Updating the Delivery Item

A Delivery Item is simply a URL, with some additional information like credentials, and it may be added, removed or updated by an Event Handler. Only the Delivery Items existing in the request just before the Delivery Request is issued are subject to delivery. For example, the concatenation service may need to

remove all items being concatenated and replace them with the concatenation result.

Using the Event Handler Response

Another way an Event Handler may affect Distribution Request execution is through the Event Handler response. If a failure response is returned, then the Distribution Request either stops issuing Simple Requests subsequent to the failed Event Handler or it stops issuing Simple Requests completely. (Which of these two actions is used will depend on the Distribution Request failure policy). The only exception to this behavior is when the Event Handler itself is marked as non-critical.

Event Handlers in the Distribution Request

In this section we will describe the classes that a Distribution Request Event Handler uses to access the state that the Distribution service stores during the processing of a Distribution Request. Through the classes described here the Event Handler can access all the state information in read-only mode. The only permitted read-write access is to the contents of the delivery packages.

Delivery Packages and Delivery Channels

The Event Handler is allowed to add and/or remove items from the Delivery Packages. These items are files that the Distribution service will deliver through the particular Delivery Channel to which the Delivery Package belongs. In fact for all purposes with these classes the Delivery Packages are identical to the Delivery Channels to which they belong. They are also referred to as Delivery Channels.

There are two categories of classes:

- Entities, and
- Data Access Objects.

Entities

Entities represent the elements of the Distribution Request. The metadata associated with the various elements is still in its raw form as it was passed in with the Distribution Requests, as an XML document. This is because the Distribution service does not know the meaning of the metadata and cannot interpret it. Metadata therefore is offered as an XML document and it is the Event Handler's responsibility to interpret it.

Data Access Objects

The Data Access Objects, although public, are not meant to be used by the client directly. The entity classes use them as they decouple the entities from the knowledge of how the data access is implemented.

All the classes, both entities and Data Access Objects, are available in the main package: `com.insystems.distributor.client.services.distribution.`

Entities

Entities are contained in a sub-package of the main package named `entities`. Apart from the entities there are some other supporting classes, for example primary key classes.

Entities represent the elements present in the Distribution Request. Some of the entities **aggregate** other entities. To reduce the overhead of having all the elements of the Distribution Request in memory at the same time, the collections of aggregated entities are not cached in memory by the entity that aggregates them. Instead, as a collection of entities is iterated the data is read from the database and objects are instantiated, but they are not held by their higher level entity.

Important: Every call to a get method that returns a collection of subentities will incur database access and should be considered expensive.

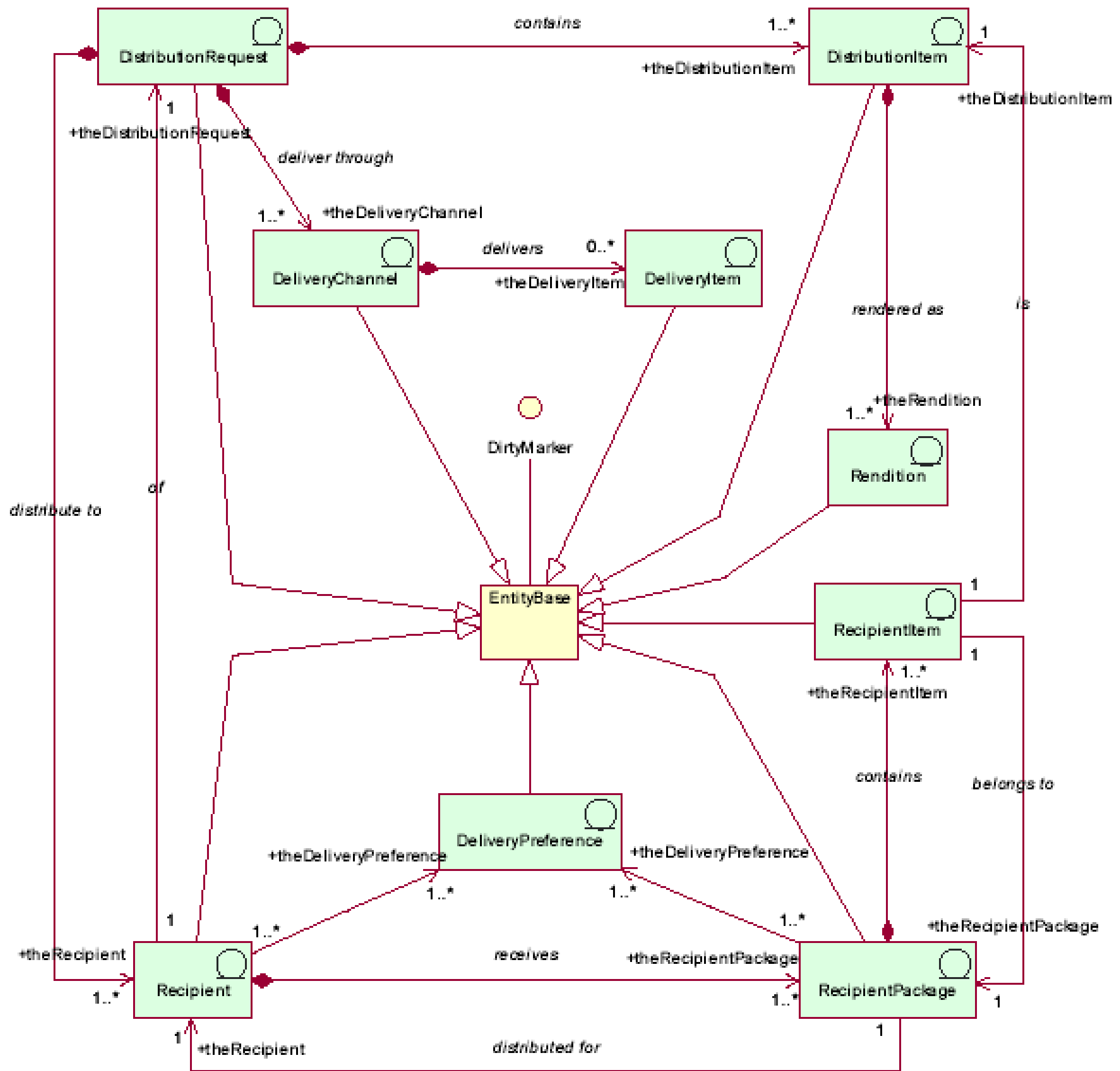
root entity

The root entity is the DistributionRequest and from it all other entities can be reached. All that is necessary to have in order to load a DistributionRequest entity is the Request ID of that request. This is the id that has been passed to the Event Handler in its invocation. It is a string whose structure is defined by the client that submitted the request in the first place.

Numeric IDs

Other ids, which are also strings, are present in the Distribution Request. However, to make the database more efficient, the Distribution service replaces the string ids with numbers that it ensures are unique. These number ids are part of the primary keys of the entities but the original ids can still be used to access the entities.

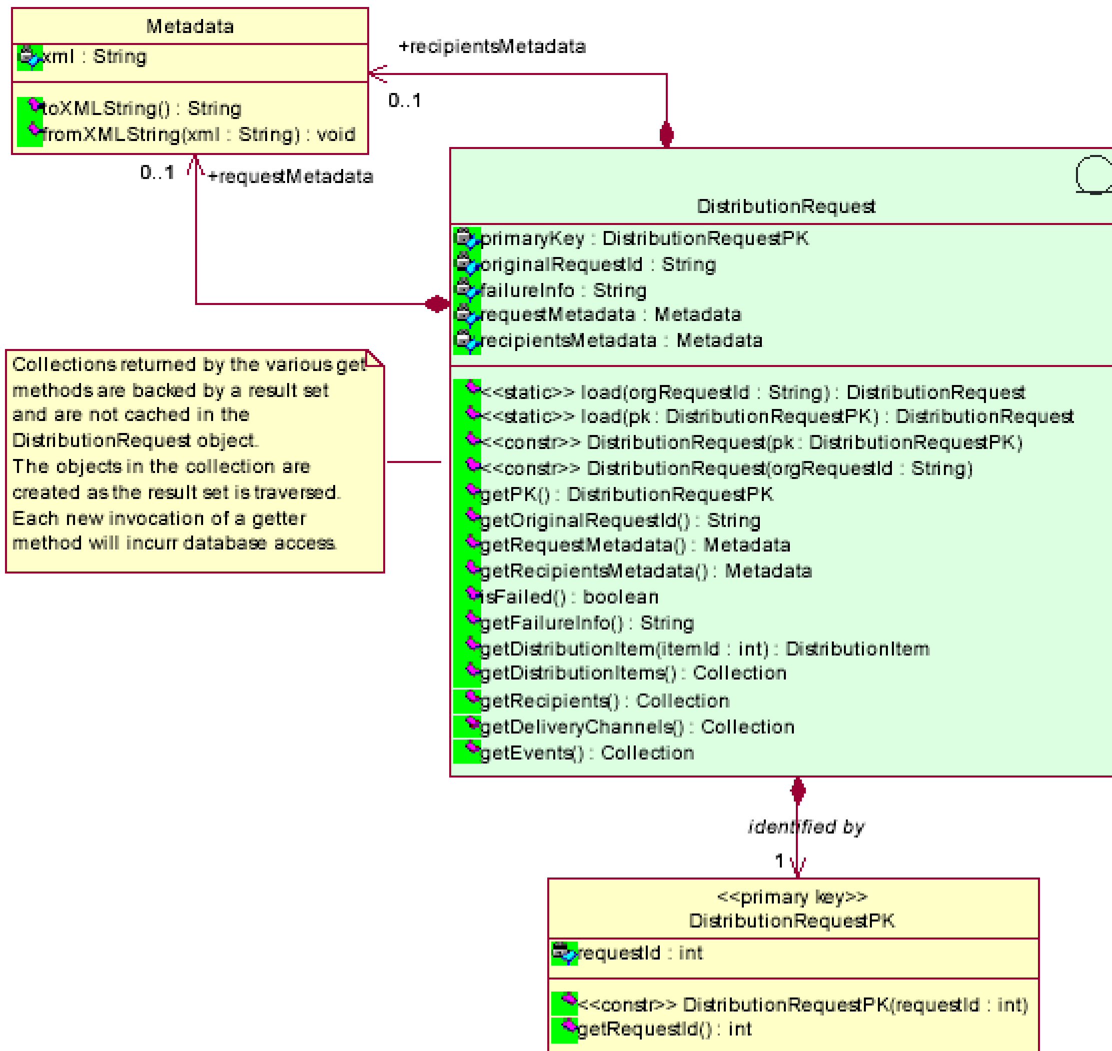
IStream Publisher Object Relationships



Distribution Request

The DistributionRequest entity is the starting point in accessing any other entity. To load an instance of a Distribution Request, the Event Handler uses the static method `load()`. This method accepts the `originalRequestID` of the Distribution Request whose data it wants to access.

Distribution Request Entity



Once loaded, the primary key of the Distribution Request can be obtained and used to further drill down on other contained entities.

Accessing the Metadata

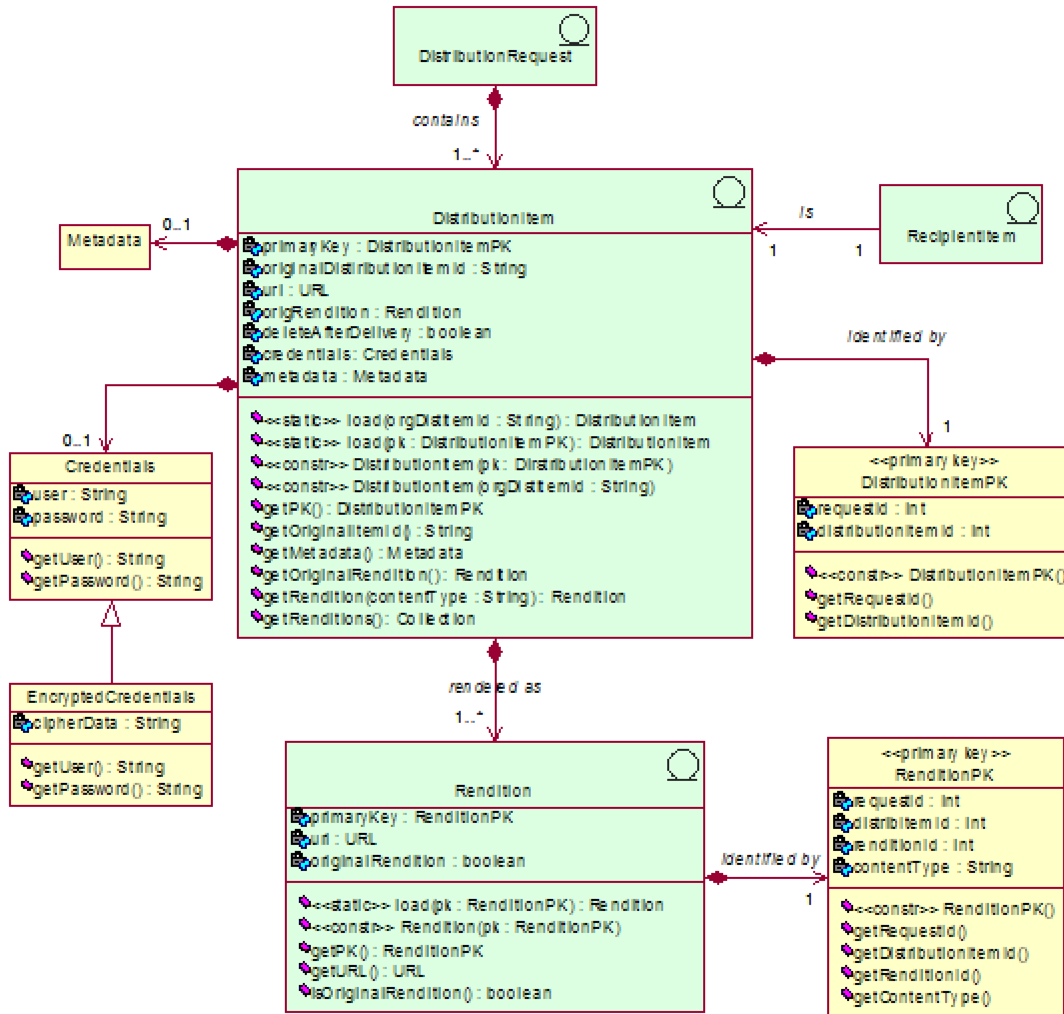
The metadata associated with the Distribution Request itself as well as that associated with the recipients can be obtained as an instance of the `Metadata` class. From this class, the metadata can be accessed as an XML string with the `toXMLString()` method.

Distribution Package

The contents of the Distribution Package are accessible through the DistributionItem and Rendition entities. The Event Handler does not directly load any entities below the Distribution Request. Instead it uses the appropriate `get` method of the higher level entity to access a collection of subentities.

In the case of Distribution Items, the Event Handler calls the `getDistributionItems()` method to obtain a collection of DistributionItem entities.

The Distribution Item



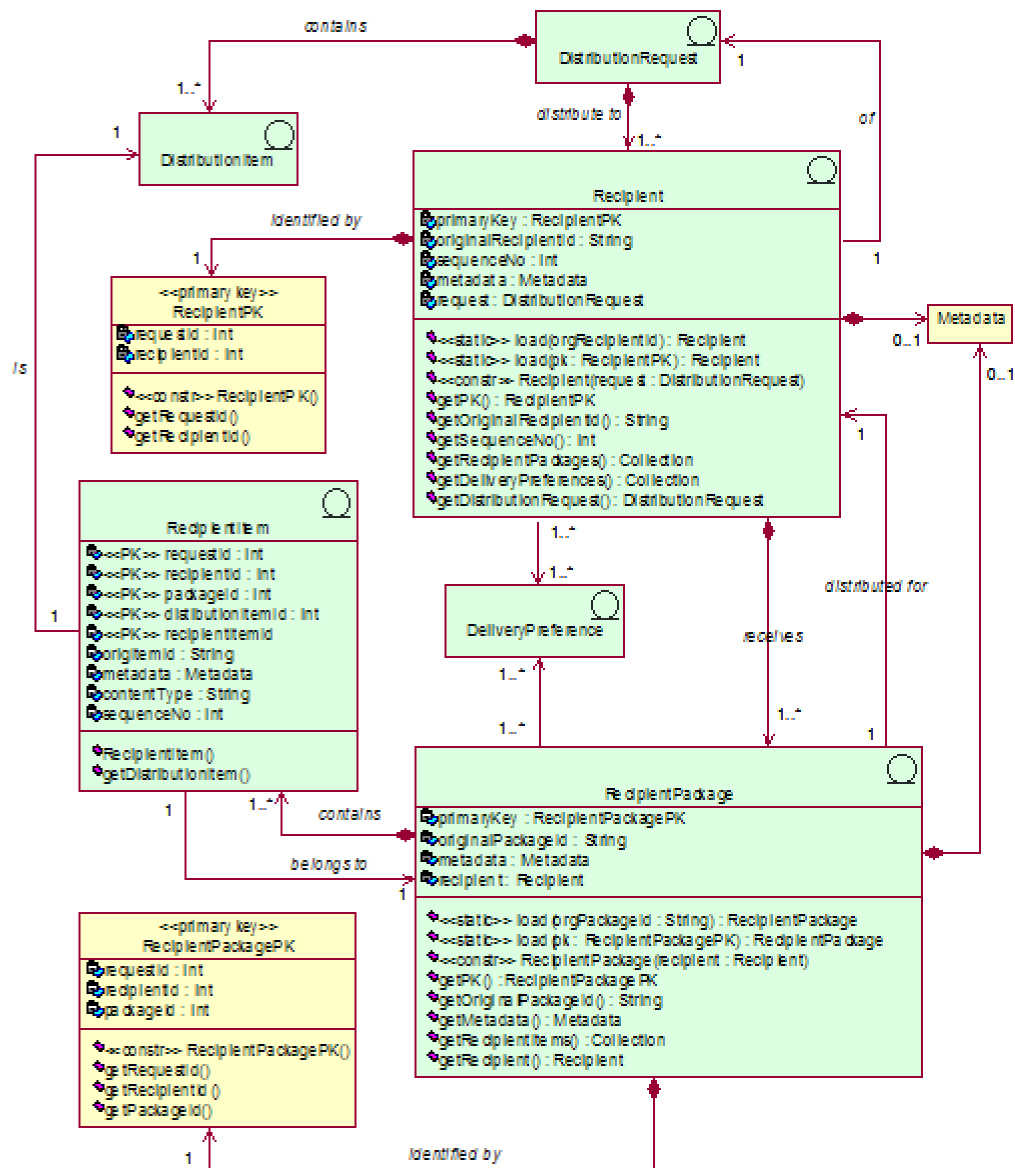
When the Distribution Item requires credentials, those credentials are accessible via a `Credentials` class or its subclass, EncryptedCredentials.

Each Distribution Item is associated with a particular set of renditions. One of these is the original rendition, meaning the rendition of the item as specified in the Distribution Package. Both the original rendition and other renditions can be obtained by specifying their content type (MIME type).

Recipients

The Recipient entities refer to the `Recipient`, `RecipientPackage` and `RecipientItem` classes.

Recipient Relationships

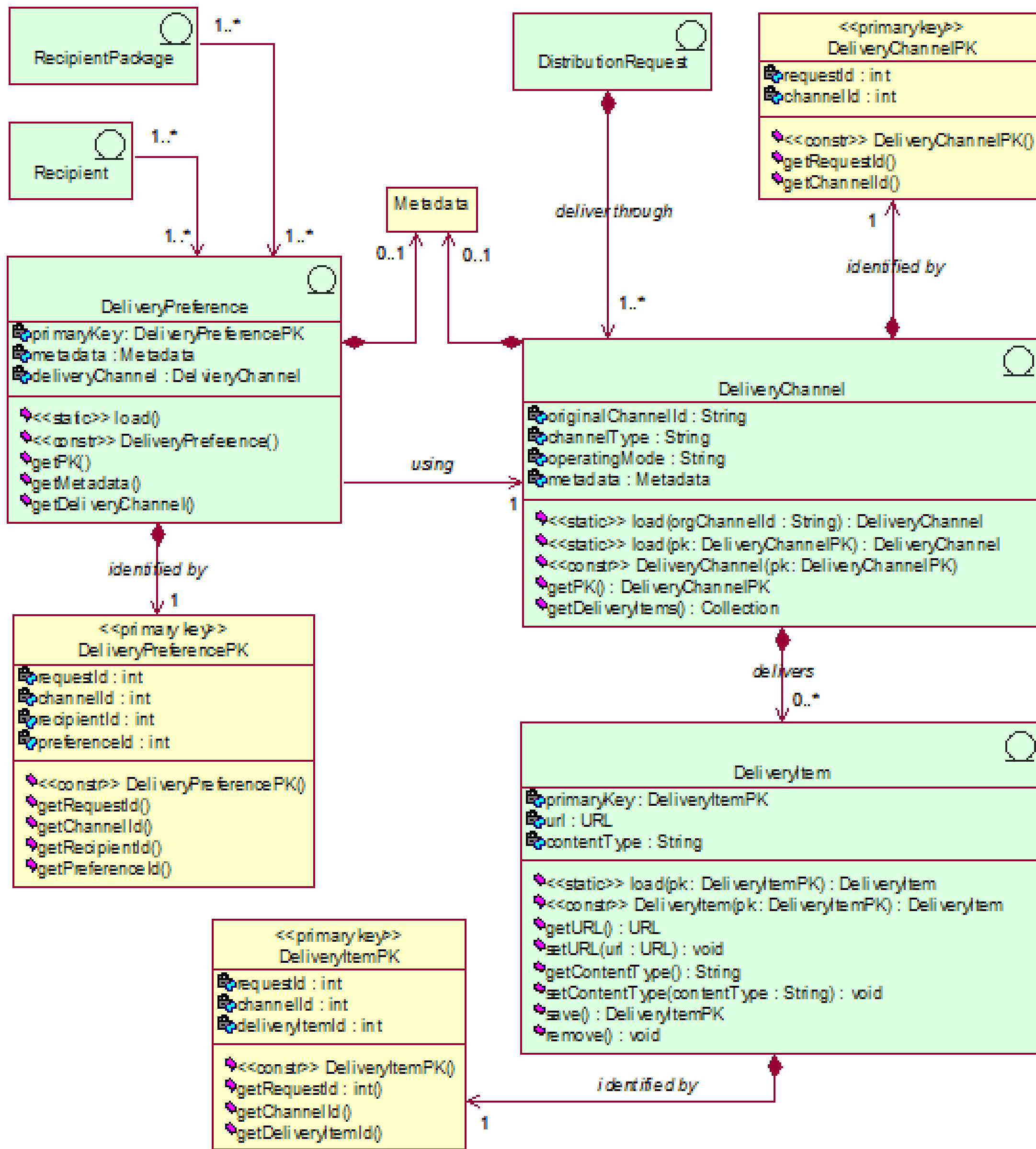


Each Recipient entity is associated with a set of RecipientPackage entities. Each RecipientPackage entity in its turn contains a set of RecipientItem entities. Each of the Recipient Items corresponds to one DistributionItem in the Distribution Package.

Each Recipient Package uses one or more Delivery Preference entities out of the set of Delivery Preferences that belong to the Recipient that owns that particular RecipientPackage.

The following diagram illustrates the Delivery Package entities.

Delivery Package



Delivery Package

The Delivery Package entities refer to the DeliveryPreference, DeliveryChannel and DeliveryItem entities.

Each DeliveryPreference uses one DeliveryChannel only. The DeliveryChannel defines the method that will be used for delivery. Each DeliveryChannel is associated with a set of DeliveryItem entities that will be delivered through that channel.

Updating the Delivery Items

The Event Handler can alter the set of Delivery Items to be distributed through a Delivery Channel. It can add new items or remove some or all of the existing items. When adding items, the Event Handler is also responsible for making sure that the content that the URL contained in the item is valid and points to an existing file. This file will remain available even after the Event Handler finishes execution.

Cleanup

When the Distribution service has completed the delivery, it will delete not only the Delivery Items but the content that they point to as well. The Event Handler should make copies of the content for the items that it adds to a Delivery Channel if it wants to retain that content after the delivery is complete.

Adding a Delivery Item

To add a Delivery Item, the Event Handler creates a new instance of a DeliveryItem entity. Before doing that, it must create an instance of the `DeliveryItemPK` class and populate it with values. With the primary key instance it can then create the instance of the DeliveryItem. To commit the change to the database, the Event Handler must call the `save()` method of the new DeliveryItem instance. If changes are made after that, the `save()` method must be invoked again to commit those changes. Any calls to `save()` other than the initial one will only update the Delivery Item and not insert a new record in the database.

Removing a Delivery Item

To remove an existing Delivery Item the Event Handler must call the `remove()` method on that particular instance of DeliveryItem.

Distribution Request with Event Handler Example

Event Handlers are described in the last section of Distribution Request.

The following is an example of a typical Event Handler structure:

```
<distribution-request>
  ...
  <event-handlers>
    <event>
      <one-of-predefined-events [optional data to define
        handler scope]/>
      <event-handler serviceType="handler(service) name">
        <event-handler-metadata>
          ... handler specific content ...
        </event-handler-metadata>
      </event-handler>
    </event>
  </event-handlers>
</distribution-request>
```

In the example above:

- `one-of-predefined-events` is a pre-defined type of event specified in the schema, which specifies which Event Handler to activate. Possible event types are:
    - distribution-package-ready
    - recipient-package-ready
    - delivery-package-ready
    - package-delivered
    - delivery-item-ready
    - item-delivered
    - distribution-complete).
- *optional data to define handler scope* is required when the Event Handler may be triggered more than once for a particular event type, for example recipient-package-ready. This functionality requires that a recipientPackageRefID value be specified.
- *handler(service) name* - This is the Event Handler name, or service name, as it is registered with the Service Manager. The service with this name handles Event Handler requests submitted from the Distribution Service.
- ... *handler specific content* ... This is the actual content of the information that is copied 'as is' into the Event Handler request.

Example

An example of an actual Event Handler in a Distribution Request follows:

```
<distribution-request>
  ...
  <event-handlers>
    <event>
      <recipient-package-ready
        recipientPackageRefID="RCP-PKG-1"/>
      <event-handler serviceType="concatenate-pcl">
        <event-handler-metadata>
          <concatenate-pcl numberPCLsegments="2">
            <page-header paperSize="Letter"
              paperOrientation="Portrait"
              paperSource="Tray 1">
              <field name="job-name" value="Job Name"/>
            </page-header>
          </concatenate-pcl>
        </event-handler-metadata>
      </event-handler>
    </event>
  </event-handlers>
</distribution-request>
```

When the Recipient Package referenced in the Event Handler above is ready (all items generated and/or rendered), then the following request is sent out:

```
<event-handler-request taskID="123"
  distributionRequestID="12345">
  <event>
    <recipient-package-ready
      recipientPackageRefID="RCP-PKG-1"/>
    <event-handler serviceType="concatenate-pcl"
      seqNumber=""
      critical="true">
      <event-handler-metadata>
        <concatenate-pcl numberPCLsegments="2">
          <page-header paperSize="Letter"
            paperOrientation="Portrait"
            paperSource="Tray 1">
              <field name="job-name" value="Job Name"/>
          </page-header>
        </concatenate-pcl>
      </event-handler-metadata>
    </event-handler>
  </event>
</event-handler-request>
```

Note: The request above is not a realistic 'concatenate-pcl' request. A real service uses the Event Handler proxy described below.

The above request becomes a payload of a JMS Message with the JMSType value set to "concatenate-pcl".

Event Handler IDs

There are several IDs that appear in the Event Handler request above:

- "**distributionRequestID**" and "**recipientPackageRefID**" are rather obvious and allow the user to define the Event Handler scope when additional data from the Distribution Request database is required.
- "**taskID**" is slightly more complex. According to the Distribution Service architecture, all tasks required to accomplish the Distribution Request constitute nodes in a single Task Graph. The execution of each task or node is subject to the successful completion of all preceding nodes in the Task Graph. When Event Handlers interact with Delivery Items, they need to associate a new or updated Delivery Item(s) with the delivery tasks responsible for the delivery of those items.
  - **taskID** in the example above allows that:
    - this **taskID** belongs to the delivery task
    - that is in charge of the delivery of Recipient Items from the given Recipient Package
    - through the given Delivery Channel.

- **Dynamically fetching Delivery Items** - The delivery task cannot know in advance what items should be delivered, therefore it contains only a template of the Delivery Request. Right before the actual delivery, this task fetches from the Distribution Request database all Delivery Items associated with this task, based on the **taskID** value.
- **Creating the actual Delivery Request** - The delivery task then transforms its Delivery Request template into an actual Delivery Request, or into multiple Delivery Requests. This is why when the Event Handler updates the Delivery Items, each new Delivery Item should have this **taskID** field properly set. Otherwise, new Delivery Items will be ignored.
- **In summary: taskID** is not an Event Handler task ID in the Task Graph but it is instead an ID of the delivery task that follows this Event Handler task.

This matter can become somewhat complicated when the Recipient Package is delivered via multiple channels. In this case, multiple delivery tasks follow a single Event Handler task. Then the **taskID** in fact contains the IDs of all delivery tasks that follow, for example: **taskID="123456"**.

Note: The Event Handler request presented above is issued by the Distribution Service. This request processing is a responsibility of the Event Handler implementation (as a service), which is the subject of the next section.

Event Handler Implementation

There are two ways to implement Event Handlers in IStream Publisher. One way is to provide an Event Handler implementation that:

- Accepts Event Handler requests from the Distribution Service,
- Interacts with the Distribution Request DAO API (reads data, updates Delivery Items), and
- Eventually returns an Event Handler response to the Distribution Service.

Disadvantages

The disadvantages to this approach is that such an Event Handler:

- Should be configured to get access to the Distribution Request data. This involves database connection configuration, firewall issues, and so on.
- Must have additional logic added in order to parse the Event Handler requests and to issue Event Handler responses. In addition to service that this Event Handler provides (concatenation) it must know how to interact through DAO with the Distribution Request, how to update Delivery Items and so on.
- In summary, such an Event Handler becomes much more complex than any other simple service.

Keeping it Simple

In order to keep things simple, an alternate approach to Event Handler implementation is provided. This Event Handler implementation is further broken up into two parts:

- The Event Handler proxy, and
- The actual Event Handler, or Event Handler implementation.

The Event Handler proxy is configured with the Distribution Service and runs in the same JVM (Java Virtual Machine) as the Distribution Service itself. Every time when an Event Handler request is formed, it goes first to the proxy where it is transformed into a regular Simple Request and sent for processing. In this case the actual Event Handler is no different from any Simple Service. It is called exactly the same way either from the Distribution Request or directly from the client.

These two approaches to implement the Event Handler are different and will be described separately, as Event Handlers without and with a proxy.

Event Handlers Without a Proxy

An Event Handler request received by an Event Handler without a proxy was already discussed in *Event Handlers in the Distribution Request* on page 138. To recap, here is some sample code:

```
<event-handler-request taskID="123"
  distributionRequestID="12345">
  <event>
    <recipient-package-ready
      recipientPackageRefID="RCP-PKG-1"/>
    <event-handler serviceType="concatenate-pcl"
      seqNumber=""
      critical="true">
      <event-handler-metadata>
        ... actual request content ...
      </event-handler-metadata>
    </event-handler>
  </event>
</event-handler-request>
```

The Service Manager does not parse this request. For this reason, the Event Handler must request the Service Manager to pass this request to it as a string value. To do this, the Event Handler must implement not only the `Service` interface (`com.insystems.distributor.Service`) but also the `ExtendedService` interface (`com.insystems.distributor.ExtendedService)`. Additional details of the `ExtendedService` interface may be found in the documentation generated by the Javadoc tool.

Using the Service Interface

In this case (no proxy), the method `String run(String)` from the `Service` interface is called, and the Service Manager never parses incoming requests on its own.

How this request is then parsed and what sort of actions the Event Handler performs is up to the implementation. The main contract between the Service Manager and this service is that the `run` method should return an Event Handler response marshalled into a string, something like:

```
<event-handler-response status="success"/>
```

or

```
<event-handler-response status="failure"/>
```

As was mentioned above, the main interaction of the Event Handler with the Distribution Request occurs not through this response but when the Event Handler updates the Delivery Items in the Distribution Database. These interactions are channelled through the Distribution Request DAO API, which is discussed in *Distribution State DAO* on page 151.

In all other aspects, the Event Handler implementation resembles other Simple Services. The developer must therefore follow the same steps in coding as well as updating configuration files, as were mentioned in *Creating and Adding a Simple Service* on page 134.

Event Handler with Proxy

Each Event Handler proxy implements the `EventHandlerProxy` interface (`com.insystems.distributor.EventHandlerProxy`). All details of this interface may be found in the Javadoc-generated documentation.

The Event Handler proxy:

- retrieves some request pieces from the event-handler-metadata section of the event-handler element
- supplements it with other data found in the relevant scope of the Distribution Request
- forms a Simple Request to be processed by the second part of Event Handler, the Event Handler implementation

This Event Handler implementation, like any other Simple Service, is running “somewhere else” on the network and returns a response when processing is complete.

Note: This is a regular Simple Service Response and not an Event Handler response. The Event Handler implementation does not access the Distribution Request data in any way. This is because in general, this implementation is called as a regular Simple Service, not from the Distribution Request.

Sometimes a particular Event Handler functionality assumes dealing with Delivery Items in the Distribution Request database. In this case, it is the Event Handler’s responsibility to update the Delivery Items in the Distribution Request Database, based on the response from the Event Handler implementation. This response should contain enough of the information required by the proxy to update the Delivery Items.

Concatenating PCL Files

This “Event Handler with Proxy” approach was implemented to support the concatenation of PCL files in the Distribution Request through Event Handlers.

In this approach:

- First, the Concatenate PCL proxy is called which forms a 'concatenate-pcl' Simple Request.
- The actual file concatenation occurs later in the Simple Service 'concatenate-pcl'.
- The request result is included in the service response. This concatenation result is added as a new Delivery Item while all items being concatenated are removed. Consequently, only the concatenation result is actually delivered through Delivery Channels.

Implementation

The Event Handler implementation is added to IStream Publisher in exactly the same way as all other Simple Services are added.

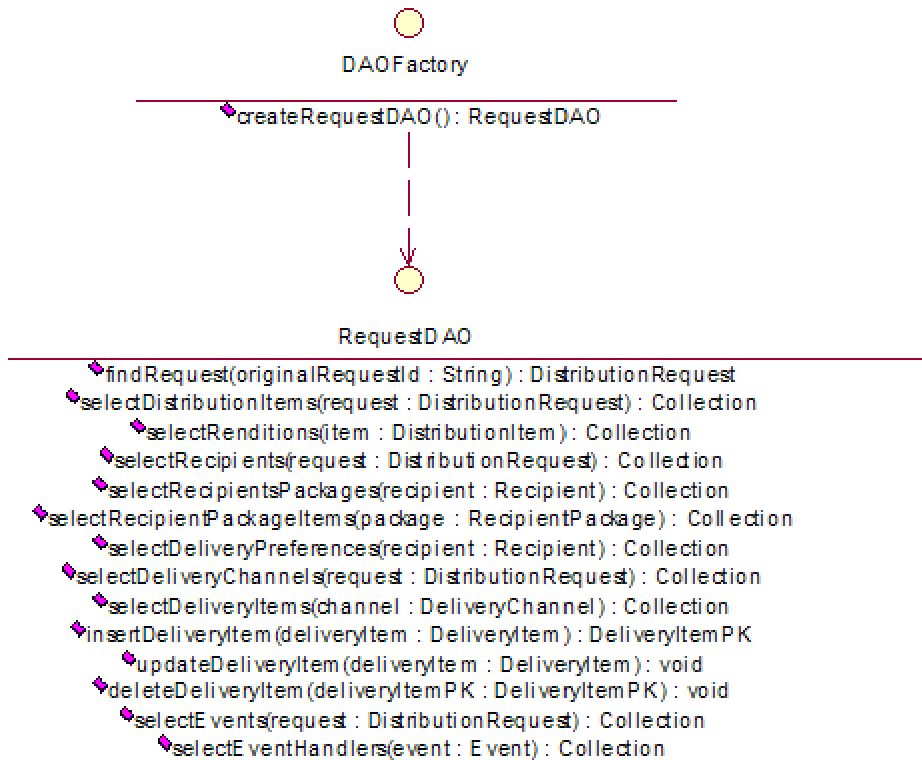
The Event Handler proxy to be called by the Distribution Service should be added to the Distribution Request configuration. After deploying a new custom service, using Admin Console, create new entity `Property[proxy]` with attribute Value set to custom proxy class name, implementing the `EventHandlerProxy` interface for this service.

Distribution State DAO

Data Access Objects

The specifics of data access are separated and made accessible in what is called a Data Access Object. In IStream Publisher, there is only one Data Access Object, the `RequestDAO`. Instances of this DAO can be created using a DAOFactory as depicted below:

The DAOFactory



Both `DAOFactory` and the `RequestDAO` are interfaces only. The implementation depends on the actual database being used. For example, in an Oracle deployment, the classes `OracleDAOFactory` and `OracleRequestDAO` will be used to implement these interfaces.

The `RequestDAO` provides the primitives required for database access but shields the entities that use them from the details of the implementation.

Customizing a Request Log Message

Customizing the Request Log Table

All JMS header properties that are found in the initial request message are converted to individual sub-elements of the Request Log message. By default, the system persists only the JMS header fields of a request, as specified in *JMS Message Header and Properties* on page 20.

The system can be configured to store other header properties (sub-elements of the Request Log message) as custom fields of the persistency layer. If it is configured that way, than those header properties can be used in subsequent query statements aimed at the system.

Adding Custom Fields

There are two steps involved in enabling the system to persist additional custom fields:

- add the appropriate database fields to the `Request` table
- provide a mapping between the new database fields and the corresponding custom JMS header fields

Modifying the Request Log Configuration

To provide the mapping between custom JMS header fields and new database fields, use the Admin Console's **Configuration** function to add a new `JMSProperty` entity under `Domain\RequestLog`. The entity name should match the JMS header property. The attribute `DBColumn` value should match the database column name in the `Request` table.

The Request Log Table

The system stores the request metadata and status information in a relational database called the Request Log.

The Request Log table consists of four tables:

- Request
- Status
- ErrorInfo
- StatusOrder

Each of these tables is described *The Request Log Table* on page 85.

Chapter 9

SDK – Web Service Interface

This chapter describes:

- *The Web Services Interface* on page 156
- *IStream Publisher WSI Architecture* on page 157
- *Configuring the IStream Publisher WSI in the Console* on page 165
- *Troubleshooting the IStream Publisher WSI* on page 167

The Web Services Interface

The Web Services Interface (WSI) is a new feature of IStream Publisher. It allows IStream Publisher functions to be invoked from various applications and platforms, and supports customers who are adopting Web services into their infrastructure.

About Web Service Applications

Web Service is an application that can be accessed on the Web or an intranet through a URL. It is accessed by clients using an XML-based Simple Object Access Protocol (SOAP) that is sent over HTTP or HTTPS. Clients access a Web service application through its interface using a Web Services Definition Language (WSDL) file.

IStream Publisher WSI Benefits

The IStream Publisher WSI is an alternative way to submit IStream Publisher requests; it is platform-independent and co-exists with current Java and JMS APIs.

The IStream Publisher WSI has the following benefits:

- **Interoperability in a heterogeneous environment** – The greatest strength of Web Services is their ability to enable inter-operability in a heterogeneous environment.
- **Easy integration with various front- and back-end systems** – The IStream Publisher WSI provides a standard way to access the services required by multi-tier applications and also provides standard supports for a variety of clients. It therefore gives IStream Publisher the flexibility to easily integrate with various front- and back-end systems.
- **Support of many client types** – Clients can be written in any language and deployed on any Web Service-enabled platform (Java, C++, C#, VB.NET, and son). You can select the configuration that best meets your application requirements.

IStream Publisher WSI Architecture

The IStream Publisher WSI provides various methods for invoking services synchronously and asynchronously, as well as cancelling asynchronous requests.

General Information

XML is the only supported format for all requests and responses. The IStream Publisher WSI accepts all currently supported XML requests. The existing request structure is supported for all requests to ensure compatibility with previous versions and minimize changes to existing components.

Because the IStream Publisher WSI supports clients running on non-Java platforms that may not have the same error-handling mechanisms, all Java exceptions are converted into meaningful service-specific exceptions and are returned as XML responses to clients.

Installation and Deployment

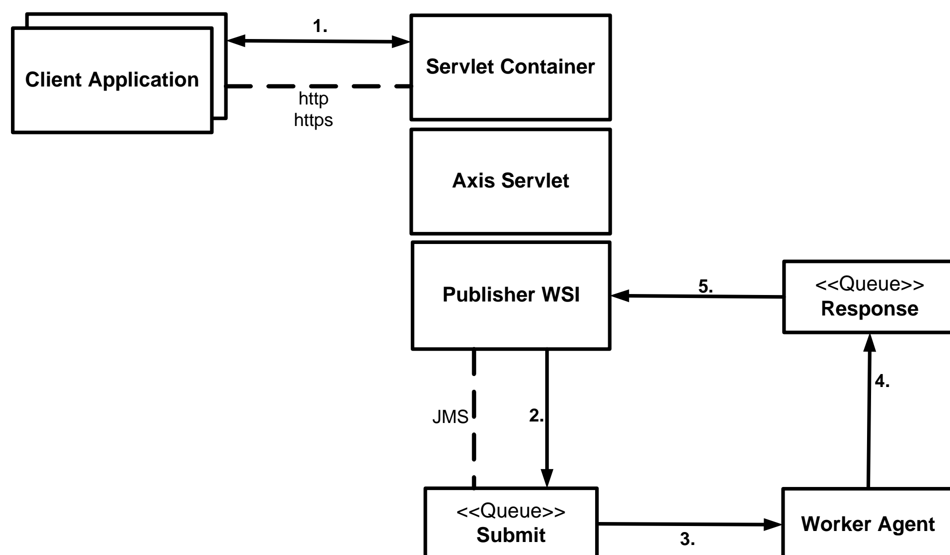
The Web Services Server can be installed on a separate machine, or on the same machine with other IStream Publisher components.

Server components can be deployed on Tomcat or other Servlet Container

Overview of WSI Architecture

The IStream Publisher WSI uses Apache Axis, a SOAP engine that plugs into Servlet engines, and can be deployed in many types of Servlet Containers, including Tomcat, WebSphere and WebLogic. Typically, it is deployed with the IStream Publisher Console.

The following diagram illustrates how Web Services is deployed and how it interacts with the IStream Publisher Core components.



WSI Workflow

A typical workflow scenario is:

1. The client application submits an XML request to the IStream Publisher WSI server.
2. The IStream Publisher WSI, running on Servlet Containers as a Web Service, receives the client XML request and analyzes it. If the request had been handled before and has an XML response in the database, the IStream Publisher WSI returns the response to the client directly, otherwise it submits the request to Submission Queue.
3. IStream Publisher fetches the XML request from the Submission Queue.
4. The Worker fulfills the request, generates an XML response and sends it back to response queue.
5. The IStream Publisher Client API monitors the response queue and receives an XML response back.
6. The IStream Publisher WSI forwards the response back to the client.

Web Services Interface Methods

The IStream Publisher WSI exposes one main interface, `Processor`, with the following functional interface methods:

Synchronous Invocation

```
String execute(String xmlRequest, long timeout);
// return XML response
```

Asynchronous Invocation

```
String submit(String xmlRequest, String deferralTime);
// return request Id

String getResponse(String requestId);
// return XML response

String cancel(String requestId);
// return cancellation result
```

Response Handler Interface

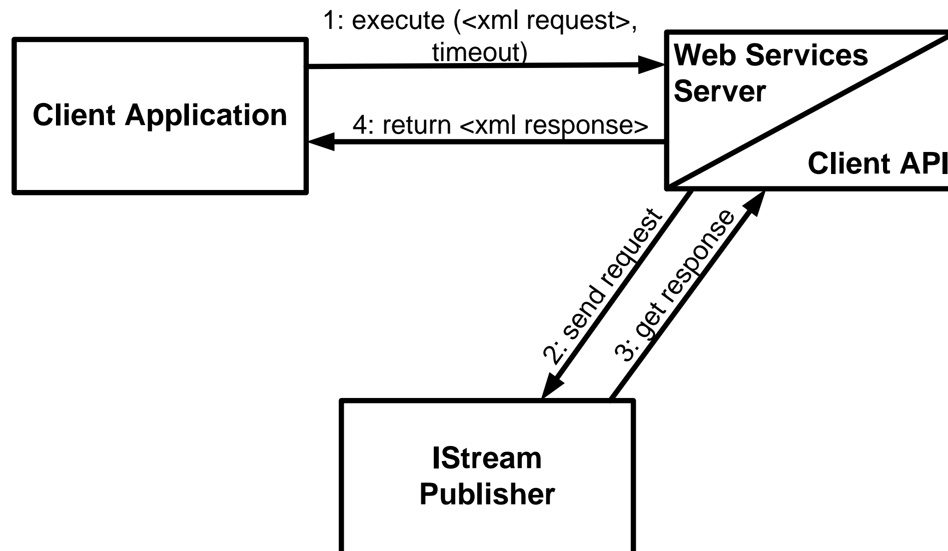
(This interface is implemented by your company.)

```
void processResponse(String requestId, String xmlResponse)
```

Flows of IStream Publisher WSI Calls

Synchronous Invocation

Normal Flow – Synchronous Call

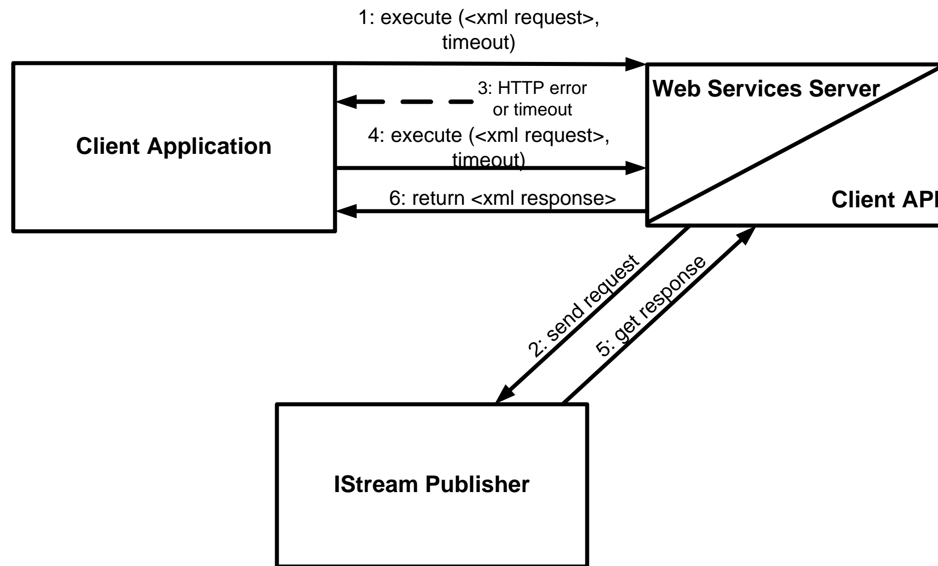


1. The client calls `execute` method with an XML request string and timeout. The timeout is specified in milliseconds and must be greater than 0.
2. The Web Services Server calls the IStream Publisher Client API and sends the request to the IStream Publisher Core.
3. The IStream Publisher Core processes the request, receives a response, then sends it back to the IStream Publisher WSI.
4. The IStream Publisher WSI wraps up the XML response and returns it to the client application.

Notes

- The IStream Publisher WSI generates an error and returns it to the client if a response was not received within the specified timeout period.
- The IStream Publisher WSI keeps requests and responses in the database for the specified time. During this period, when the client recalls the `execute` method with the same XML request, the IStream Publisher WSI simply returns the response from the database. Once the specified time expires, the data in the database is cleared. The client recalls the `execute` method with the same XML request again. The IStream Publisher WSI submits this XML request to the IStream Publisher Core, processes it and receives a new response.

Retry Flow - Synchronous Call with Retry



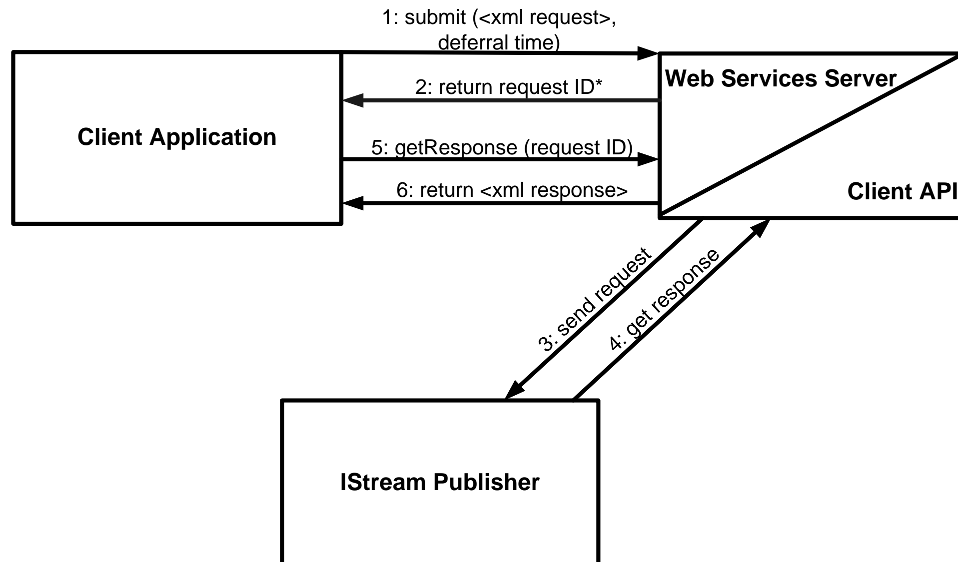
1. The client calls the `execute` method with an XML request string and timeout. The timeout is specified in milliseconds and must be greater than 0.
2. The Web Services Server calls the IStream Publisher Client API and sends a request to the IStream Publisher Core.

   (Note that the call to the IStream Publisher WSI can fail at any time because of system or application issues.)
3. The client recalls the `execute` method with the same XML request again.
4. The IStream Publisher Core processes the request and receives a response, then sends the response back to the IStream Publisher WSI.
5. The IStream Publisher WSI wraps up the XML response and returns it to the client application.

Note: The IStream Publisher WSI prevents multiple submissions of the same request by persisting the message digest.

Asynchronous Invocation

Normal Flow – Asynchronous Call



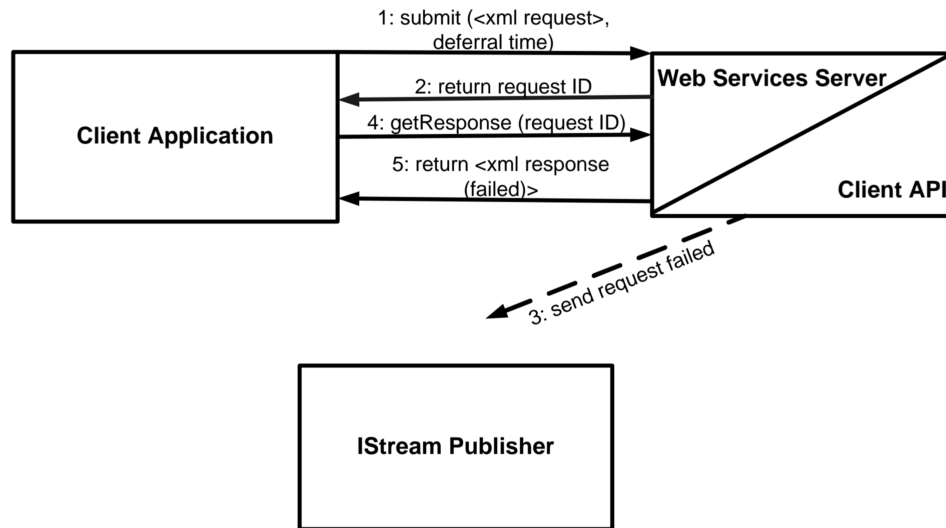
* This return request ID is not the request ID from the request table. To specify the request ID from the request table, use `name=RequestID` in the `jms-properties` element.

1. The client calls the `submit` method with an XML request string and deferral time. The deferral time parameter is optional and should be specified in the supported format.
2. The Web Services Server generates a message digest and returns it to the client.
3. The Web Services Server calls the IStream Publisher Client API and sends a request to the IStream Publisher Core.
4. The IStream Publisher Core processes the request and generates a response, then sends the response back to the IStream Publisher WSI. The IStream Publisher WSI receives the response and saves it to the database.
5. The client asynchronously calls the `getResponse` method, specifying the digest returned by the WSI as an argument.
6. The IStream Publisher WSI obtains the associated XML response from the database, wraps it up, and returns it to the client application.

Note:

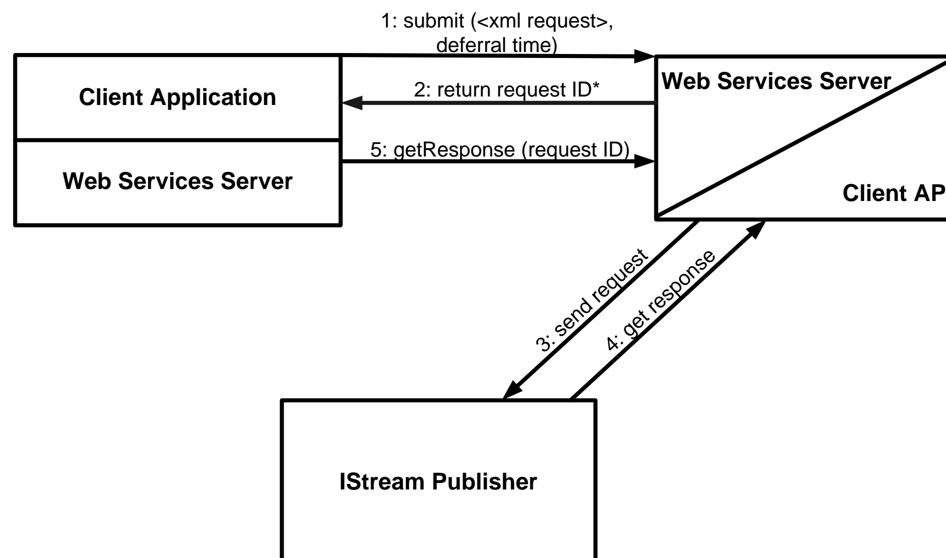
- Each message digest generated by the IStream Publisher WSI is unique and persistent for each XML request.
- If the request response is not available when the client calls the `getResponse` method, then this method returns an appropriate XML message instead of response.

Failure Flow – Asynchronous Call with Failure



1. The client calls the `submit` method with an XML request string and deferral time. The deferral time parameter is optional and should be specified in the supported format.
2. The Web Services Server generates a message digest and returns it to the client.
3. The call to the IStream Publisher Client API fails.
4. The client asynchronously calls the `getResponse` method using the digest returned from WSI as an argument.
5. The IStream Publisher WSI returns an appropriate XML failure message.

Call Back Flow – Asynchronous Call with Call-Back

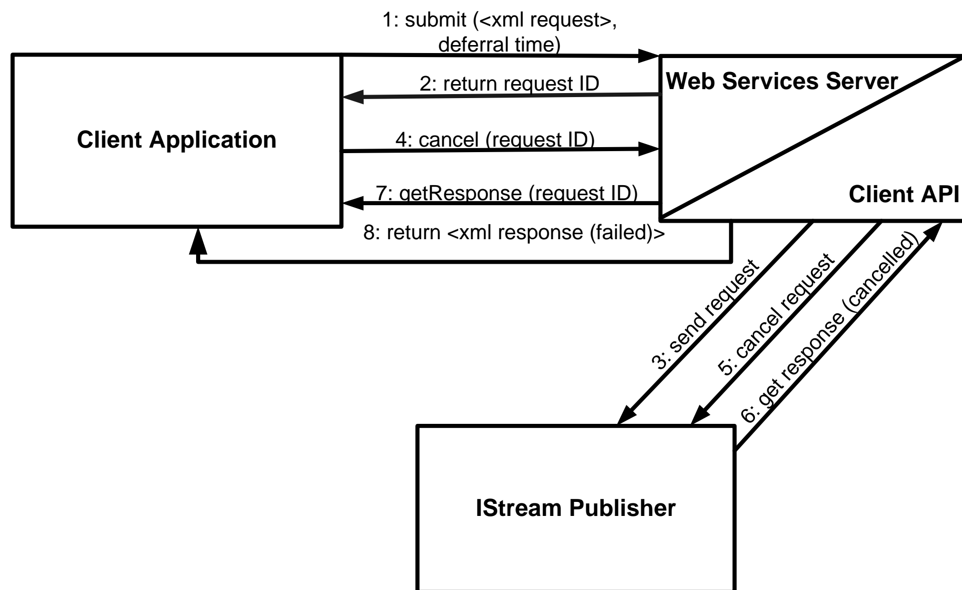


1. The client calls the `submit` method with an XML request string and deferral time. The deferral time parameter is optional and should be specified in the supported format.
2. The Web Services Server generates a message digest and returns it to the client.
3. The Web Services Server calls the IStream Publisher Client API and sends a request to the IStream Publisher Core.
4. The IStream Publisher Core processes the request and generates a response, then sends it back to the IStream Publisher WSI. The IStream Publisher WSI receives the response and saves it to a database.
5. The IStream Publisher WSI calls the `processResponse` method of the `ResponseHandler` web service endpoint. The message digest and XML response are passed as arguments of the `processResponse` method.

Notes

- The client application itself is a Web Services Server which implements the `processResponse` interface.
- The URL to a client's Response Handler Web Service is an optional item of the IStream Publisher WSI configuration, and is in the IStream Publisher Console Web Service subfolder.
- The IStream Publisher WSI supports a push back of the XML response to a single client Web Service URL. If multiple clients request a callback, the client's web server may be designed to propagate this response to other clients.

Cancel Flow – Asynchronous Call with Cancellation



1. The client calls the `submit` method with an XML request string and deferral time. The deferral time parameter is optional and should be specified in the supported format.
2. The Web Services Server generates a message digest and returns it to the client.
3. The Web Services Server calls the IStream Publisher Client API and sends a request to the IStream Publisher Core.
4. The client asynchronously calls the `cancel` method using a digest returned from the IStream Publisher WSI as an argument.
5. The Web Services Server calls the IStream Publisher client API and sends a `cancel` request to the IStream Publisher Core.
6. The IStream Publisher Core cancels the request, generates a `cancel succeed` response, then sends this response back to the IStream Publisher WSI. The IStream Publisher WSI received the response and saves to a database.
7. The client asynchronously calls the `getResponse` method using a digest returned from IStream Publisher WSI as an argument.
8. The IStream Publisher WSI obtains the associated `cancel succeed` XML response from the database, wraps it up, and returns it to the client application.

Notes

- When the IStream Publisher Core processes the `cancel` request, if the original `submit` XML request has been completed, a `cancel failed` XML response will be generated and returned to the client.
- Call back response processing is also supported with the `cancel` method.

WSI WSDL

WSDL describes the point of contact for a service provider. This point of contact is also called the service endpoint. It provides a formal definition of the endpoint interface and establishes the physical location of the service.

You can retrieve the IStream Publisher WSDL using the following URL on the server where WSI is deployed:

```
http://wsihost:8080/wsi/services/Processor?wsdl
```

Configuring the IStream Publisher WSI in the Console

To configure the IStream Publisher WSI

1. When the IStream Publisher WSI starts for the first time the following messages may appear in the log:

```
2006-02-10 11:08:06,338 [ERROR]
(RDHelperClassicWithFastFetch.java:491) - Could not fetch
values

java.sql.SQLException: [Microsoft][SQLServer 2000 Driver for
JDBC][SQLServer]Invalid object name 'VALUATION'
2006-02-10 11:08:06,369 [ERROR]

(ConfigurationHelper.java:494) - Could not configure
WebServices for domain  (Context[null])

com.insystems.edelivery.client.wsi.ConfigurationException:
WSI.CFG.09: WebServices configuration not found for context
"default"
```

These errors occur if the Console and Domain databases have not yet been created by a Console. Once these databases are created, restart the IStream Publisher WSI Web application to reconnect to these databases.

2. Create a Web Service entity under the `Domain` folder by clicking the **Select an entity to add** drop-down list, then selecting **Web Service**.

3. Click the **Web Service** item. All the attributes appear on the right side. If you have multiple WSI instances, click the **Select an entity to add** drop-down list and select a context.

4. Enter the attributes.

Attribute	Example	Description
`simpleSubmissionQueue`	submit	the Submission Queue for requests
`replyQueue`	response	the reply queue for web service requests
`replyTTL`	1440	the number of minutes to keep XML responses in the database for client requests
`responseHandlerURL`		the URL of the client web service endpoint that implements the `ResponseHander` interface, for example `http://server/wsiclient/services/ResponseHandler`

5. Save the configuration and restart the IStream Publisher WSI Web application.

WSI Client Examples

IStream Publisher WSI client examples are in `wsi-sdk.zip` on the IStream Publisher installation package.

Note: If you are implementing Response Handler Interface on the .NET platform, ensure the implementation class has `RoutingStyle` set to `RequestElement` and that the main method uses RPC formatting. See the following C# .NET example:

```
[SoapDocumentService(RoutingStyle=SoapServiceRoutingStyle.
RequestElement)]
public class ResponseProcessorService : System.Web.Services.WebService
{
    ...
    [WebMethod]
    [SoapRpcMethodAttribute("http://wsi.client.edelivery.XYZ.com",
       RequestNamespace="http://wsi.client.edelivery.XYZ.com",
       ResponseNamespace="http://wsi.client.edelivery.XYZ.com")]
   public void processResponse(string digest, string xmlResponse)
   {
      ...
   }
   ...
}
```

Troubleshooting the IStream Publisher WSI

- Always start the IBM MQ Series or OpenJMS Agent before starting the IStream Publisher Web Service.
- When starting the Web Service, check the log file to ensure that the `AsyncResponseReceiver` listener has started correctly. Look for errors such as `Listener cannot start`.

  On WAS, for example, the log file is located in:

  `C:\Program Files\WebSphere\AppServer\logs\server1\`
  `SystemOut.log`
- Ensure that the IStream Publisher WSI and Communicator (formerly called Correspondence) are not using the same response queue, otherwise the response may be processed by the wrong application.
- Ensure that the IStream Publisher Web Service JDBC provider is set up correctly, and test that the connection is successful.
- The Web Service is designed to prevent the execution of the same request more than once. However, if you *want* to execute the same request more than once, simply to add a space to the XML request so that the system can generate a unique digest.

IStream Publisher WSI Log files

WSI can write system messages to the following log files.

wsi.log

This log file captures events of IStream Publisher WSI activities.

Default location: `C:\Whitehill\`*`[IStream Publisher install folder]`*`\logs\wsi.log`

There are five debug levels: `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`. If you change the log level in

`[IStream Publisher install folder]``\tomcat\webapps\wsi\`
`WEB-INF\classes\log4j.properties`

to `DEBUG`, then all debug information will be recorded in `wsi.log`.

Deployed Application Server JVM Log Files

The location of these log files depends on the specific application server.

For example, for WAS, the default location for these files is:

`[WebSphere install folder]``\AppServer\logs\server1\`

The files are `SystemOut.log` and `System Err.log`.

Appendix A

Reference Material – Samples

This appendix contains code samples that are referenced from other sections of this document, and includes describes:

- *Sample Deliver-to-Email Request* on page 170
- *Sample Aggregate Request* on page 171
- *Header Page Template Example* on page 173
- *Interactive, Batch, and Embedded XML Data* on page 174

Sample Deliver-to-Email Request

```
<deliver-to-email subject="Test Subject" priority="normal" >
   <body-source>
   <source url="ftp://{ftp_domain}/{path} emailbody.txt"
ContentType="text/plain">
   <credentials user="{ftp_user}" password="${ftp_pwd}"/>
   </source>
 </body-source>
 <sender
   name="1st Submitter"
   emailAddress="{SenderEmail}" />
 <receiver
   name="1st Receiver"
   emailAddress="{ReceiverEmail}"
   type="to" />
 <attachment>
   <source
   url="ftp://{ftp_user}:{ftp_pwd}@{ftp_domain}/path
document.doc"
      ContentType="application/msword" />
 </attachment>
</deliver-to-email>
```

Sample Aggregate Request

This request will generate to DOC, render DOC to PDF, render PDF to PCL, and render DOC to HTML.

Sample Aggregate Request

```
<request-aggregate>
   <request>

      <generate-calligo-document>
         <calligo-sourceUISR="ModelDoc_Infosource:'ModelDocument.CMS'"
          docType="cms"/>

         <destination
         url="ftp://{ftp_user}:{ftp_password}@{ftp_domain}/{path}/
         document.doc"/>

         <key-data name="policy" value="12345" type="string"/>

      </generate-calligo-document>
     </request>

<dependent-aggregate type="render-Word-to-PDF">
   <request>
      <render-Word-to-PDF>
         <source
         url="ftp://{ftp_user}:{ftp_password}@{ftp_domain}/{path}/
         document.doc"/>

         <destination
         url="ftp://{ftp_user}:{ftp_password}@{ftp_domain}/{path}/
document.pdf"/>

         <output-name>PDF Compatible Printer Driver</output-name>
         <macro name="UpdateFields" type="word"/>
      </render-Word-to-PDF>
     </request>

<dependent-aggregate type="render-PDF-to-PCL">
   <request>
      <render-PDF-to-PCL>
         <source url="ftp://{ftp_user}:{ftp_password}@{ftp_domain}/
         {path}/document.pdf"/>

         <destination url="ftp://{ftp_user}:{ftp_password}@{ftp_domain}/
         {path}/document.pcl"/>

         <output-name>HPLJIII</output-name>
      </render-PDF-to-PCL>
   </request>

<service-queue>
```

```
   <queue name="{ServiceQueue}" type="Name"/>
</service-queue>

<reply-to-queue>

   <queue name="{ResponseQueue}" type="Name"/>
</reply-to-queue>
</dependent-aggregate>

<service-queue>
   <queue name="{ServiceQueue}" type="Name"/>
</service-queue>

<reply-to-queue>
   <queue name="{ResponseQueue}" type="Name"/>
</reply-to-queue>
</dependent-aggregate>

<dependent-aggregate type="render-Word-to-HTML">
   <request>
      <render-Word-to-HTML>
          <source url="ftp://{ftp_user}:{ftp_password}@{ftp_domain}/
         {path}/document.doc"/>

   <destination url="ftp://{ftp_user}:{ftp_password}@{ftp_domain}/
   {path}/document.htm"/>

   <macro name="UpdateFields" type="word"/>
      </render-Word-to-HTML>
   </request>

<service-queue>
   <queue name="{ServiceQueue}" type="Name"/>
</service-queue>

<reply-to-queue>
   <queue name="{ResponseQueue}" type="Name"/>

</reply-to-queue>
</dependent-aggregate>
</request-aggregate>
```

Header Page Template Example

{sequence number}

{job-name}

Job Number:	{job-number}
Job Submit Date:	{job-submit-date}
Job Submit Time:	{job-submit-time}

Distribution Instructions:

{distribution-instructions}

{message}

Interactive, Batch, and Embedded XML Data

Interactive Mode

In the example below, the XML file in the referenced data is in Interactive mode, meaning: one XML file per document generation.

```
<generate-calligo-document>
   <calligo-source
UISR="ModelDocuments:LTC_Rate_Increase_Letter.CMS"
docType="cms"/>

   <destination url="ftp://host/destination/
   LTC_Rate_Increase_Letter.doc">
      <credentials user="ftp_user" password="ftp_password"/>
   </destination>

   <generation-data name="infosource_location">
      <source url="file://domain/share/source/2.xml"
      deleteAfterExecution="success"/>
   </generation-data>

</generate-calligo-document>
```

Here, the name attribute in the `xml-data-def` element is the name of the location of the XML file which references key data.

Example: `QUERY "FILE=" + infosource_location, "XMLInfoSource")`

Batch

Batch data has one XML file for many different document generations, each of which is separated by a unique JOB ID. Using Batch referenced data would result in the `<xml-data>` section being different than the previous example, as follows:

```
<generate-calligo-document>
   <calligo-source
UISR="ModelDocuments:LTC_Rate_Increase_Letter.CMS"
docType="cms"/>

   <destination url="ftp://host/destination/
   LTC_Rate_Increase_Letter.doc">
      <credentials user="ftp_user" password="ftp_password"/>
   </destination>

   <key-data name="$jobIDkey" value="2" />
   <generation-data name="infosource_location">
      <source url="file://domain/share/source/
      2Batch.xml"/>
   </generation-data>
</generate-calligo-document>
```

Here, the name attribute in the `xml-data-def` element is the name of the location of XML file which references key data. The name attribute in the `xml-data` element is the name of the key data to be replaced with the value specified in the attribute `jobID`.

Example:
```
QUERY "FILE=" + infosource_location + ";jobid=" + $jobIDkey,
"XMLInfoSource")
```

Embedded Data

Embedding data encloses existing data “as-is” for generation in the XML file. As with referenced data, the name element is used to specify the key data that will be replaced with the embedded data. All of the data for generation must be embedded in the request.

```
<generate-calligo-document>
   <calligo-source
   UISR="ModelDocuments:LTC_Rate_Increase_Letter.CMS"
   docType="cms"/>

   <destination url="ftp://host/destination/
   LTC_Rate_Increase_Letter.doc">
      <credentials user="ftp_user" password="ftp_password"/>
   </destination>

   <generation-data name="infosource_location">
      <job-data>
      &lt;?xml version=&quot;1.0&quot;?&gt;
      &lt;interactive&gt;
      &lt;PolicyNumber
      type=&quot;string&quot;&gt;0764344444&lt;/
      PolicyNumber&gt;
      &lt;NewPremium
      type=&quot;double&quot;&gt;404.68&lt;/
      NewPremium&gt;
      &lt;EffectiveDate type=&quot;string&quot;&gt;12/
      15/2001&lt;/EffectiveDate&gt;
      &lt;CompanyCode
      type=&quot;string&quot;&gt;01&lt;/
      CompanyCode&gt;
      &lt;SystemId type=&quot;string&quot;&gt;23&lt;/
      SystemId&gt;&lt;Entity&gt;
      &lt;TaxId type=&quot;array&quot;&gt;
      &lt;row type=&quot;string&quot;/&gt;
      &lt;row type=&quot;string&quot;/&gt;
      &lt;row type=&quot;string&quot;/&gt;
      &lt;row type=&quot;string&quot;/&gt;
      &lt;row type=&quot;string&quot;&gt;
      555555555&lt;/row&gt;
      &lt;row type=&quot;string&quot;
      &gt;555555556&lt;/row&gt;
```

```
        &lt;/TaxId&gt;
        &lt;TaxIdType type=&quot;array&quot;&gt;
        &lt;row type=&quot;string&quot;/&gt;
        &lt;row type=&quot;string&quot;/&gt;
        &lt;row type=&quot;string&quot;/&gt;
        &lt;row type=&quot;string&quot;/&gt;
        &lt;row type=&quot;string&quot;&gt;
        undefined&lt;/row&gt;
        &lt;row type=&quot;string&quot;&gt;
        undefined&lt;/row&gt;
        &lt;/TaxIdType&gt;
        &lt;/Entity&gt;
        &lt;/interactive&gt;
        </job-data>

   </generation-data>
</generate-calligo-document>
```

Here, the name attribute in the `xml-data-def` element is the name of the location of XML file which references key data.

Example: `QUERY "FILE=" + infosource_location, "XMLInfoSource")`

Appendix B

Glossary

A – C – D – E – I – J – M – O – P – Q – R – S – T – U – W

A

Administrative Command

A request to control and administer IStream Publisher components such as start, stop, or queryState.

Admin Queue

A Queue used to store administrative commands.

Agent

IStream Publisher uses agents to control and manage the functioning of the system.

Aggregate Request

A type of Composite Request where its component Simple Requests are specified in the body of the composite in a tree-like structure. The hierarchy they form represents their processing dependencies. Each request spawned from an aggregate request or Composite Request has a main request ID and an aggregate or composite request ID in addition to its own request ID. When tracing an aggregate or composite request there are multiple responses, one for each simple request.

Aggregate Service

A type of Service used when one or more Simple Services are required in a particular order.

Application Programming Interface (API)

A set of routines, protocols, and tools for building software applications. A good API makes it easier to develop a program by providing all the relevant building blocks.

C

Calligo Document

The previous name for an IStream Document.

Composite Request

A Request containing more than one requests.

Completion Queue

The Queue into which the Simple Service places a message to inform the Distribution Service that the processing of the Task Graph is now complete.

Component

A separate, functioning area of the IStream Publisher system,

Console

A front-end application used to send administrative commands to IStream Publisher.

Console Database

The database from which the IStream Publisher components directly retrieve their configuration.

Content Service

Provides access to content generated by the IStream Document Assembly Service.

Coordinated Universal Time (UTC)

A time scale, based on the second (SI), as defined and recommended by the CCIR, and maintained by the Bureau International des Poids et Mesures (BIPM).

Component Object Model (COM)

A model for binary code developed by Microsoft. The Component Object Model (COM) enables programmers to develop objects that can be accessed by any COM-compliant application. Both OLE and ActiveX are based on COM.

D

Delivery Channel

Represents a physical device that can transmit information in printed form (print, fax) or electronic form (e-mail, repository).

Delivery Package

The package of Recipient Items after rendering and before they are about to be delivered to a Delivery Channel.

Delivery Preference

A method of the recipient that controls the default settings for how a Request will be delivered for that recipient.

Delivery Service

All functionality related to delivering the content to the recipients. Once content has been extracted and rendered (if applicable), the final step in the distribution process is to deliver the content to the recipients. The following methods are available to deliver content: Repository, Printer, E-mail and Fax.

Distribution Event

An event that occurs every time the processing of a Distribution Request reaches a predetermined point where specific action should be taken or where customization of the distribution process itself is possible.

Distribution Item

One of the documents or files that make up the Distribution Package.

Distribution Package

The full set of interrelated Distribution Item documents that make up the subject of the Document Distribution process.

Distribution Request

A request to distribute a package of interrelated documents to a group of recipients through various Delivery Channels. The Distribution Request is a Composite Request.

Distribution Service

A composite service that provides document distribution functionality invoked through a Distribution Request. The process of document distribution is defined as the selective delivery of a package of interrelated documents to multiple recipients, through various Delivery Channels.

Distribution Service Queue

The Queue that services all Distribution Requests.

Document Distribution

Consists of the distribution of a package of business interrelated documents to a list of recipients, in various formats and through different Delivery Channels.

Document Management System (DMS)

A multi-dimensional or hyperlinked organization of documents.

Domain Database

The database to which Request Log Events are logged.

E

Event Handler

The Event Handler is invoked and executed following the occurrence of the event to which it is associated.

I

InfoSource

IStream InfoSources are specific to the IStream Assembly Engine and are used to reference generated IStream documents or model documents.

Instant Delivery

An operating mode for Delivery Channels. In this mode, the actual delivery for every item in the Delivery Package can proceed as soon as the item becomes available and a *delivery item ready* event is raised.

IStream Document

A document generated from a component in IStream Document Manager. Formerly called a *Calligo document*.

J

Java Message Service (JMS)

An application program interface (API) from Sun Microsystems that supports the formal communication known as messaging between computers in a network. Sun's JMS provides a common interface to standard messaging protocols and also to special messaging services in support of Java programs.

M

Multipurpose Internet Mail Extensions (MIME)

MIME is a specification for formatting non-ASCII messages so that they can be sent over the Internet. Many e-mail clients now support MIME, which enables them to send and receive graphics, audio, and video files via the Internet mail system. In addition, MIME supports messages in character sets other than ASCII.

O

Open Database Connectivity (ODBC)

Open DataBase Connectivity is a standard database access method developed by Microsoft Corporation. The goal of ODBC is to make it possible to access any data from any application, regardless of which database management system (DBMS) is handling the data. ODBC manages this by inserting a middle layer, called a database driver, between an application and the DBMS.

The purpose of this layer is to translate the application's data queries into commands that the DBMS understands. For this to work, both the application and the DBMS must be ODBC-compliant – that is, the application must be capable of issuing ODBC commands and the DBMS must be capable of responding to them.

P

Public Admin Queue

The Queue from which an Admin system using JMS messages submits administrative requests to IStream Publisher. IStream Publisher Console does not require this queue since it communicates to the system using RMI.

Q

Queue

The area where messages (or requests) are sent to.

Queue Set

A logical set of Submission Service, Simple Service and Distribution Service Queues.

Queue Set Reference

A reference to a Queue Set within a Worker. It contains the number of listeners and the name of the Queue Set.

R

Recipient

A person or an organization to which the items in the Distribution Package are addressed. Each recipient declares preferences for the delivery methods and formats to be used. Other business process specific information can be attached as metadata, for example, the name, title, or contact.

Recipients are grouped together into a collection also called *recipients*. This provides a place to attach metadata that is common to all recipients.

The list of recipients can be ordered by assigning a delivery order to each recipient. Recipients where a delivery order is not specified are processed after all of those for which a delivery order is specified.

Recipient Item

Recipient Items as defined in the Recipient Package are sent through all Delivery Channels, as defined by the `delivery-preferences` setting.

For example, if a Delivery Preference is a `preference-fax` or a `preference-email`, it is sent as a single delivery. That is, one e-mail or fax request is sent per recipient package, and all recipient items are sent as attachments in it.

For `preference-printer` or `preference-repository`, there will be multiple delivery requests – one per recipient item. An example of multiple delivery requests per recipient item is a standard print request.

Recipient Package

A subset of the Distribution Package that contains only the items addressed to a particular recipient.

Remote Method Invocation (RMI)

The action of invoking a method of a remote interface on a remote object. A method invocation on a remote object has the same syntax as a method invocation on a local object.

Rendering Request

A request for rendering from one particular format to another.

Rendering Service

Once content has been generated, the next step is to transform the content from its current format into a different format. The formats include: Microsoft Word, HTML, PCL, PDF, and TIFF.

Repository API

The API for accessing the repository.

Request
A specific set of instructions to perform a task within the IStream Publisher system. See Aggregate Request, Composite Request, Distribution Request, Rendering Request, Service Request, and Simple Request.

Request Log
A facility used to log information about the state of all Requests currently in the system. It is made up of the `Request`, `ErrorInfo`, `Status`, and `StatusOrder` tables.

Request Log Event
An action that occurs in the IStream Publisher system when processing a request. Request log events are stored in the Domain Database.

Request Metadata
Custom information contained in the Distribution Request, for the use of the Event Handlers. The metadata associated with the Distribution Request itself as well as that associated with the recipients can be obtained as an instance of the Metadata class. From this, the metadata can be accessed as an XML string using the `toXMLString()` method.

Response
A message that IStream Publisher components send to Service Response Queues as the result of the execution of functional or administrative commands.

Response Queue
A Queue where the service will place messages in response to Service Request messages. The queue is designated by the submitter of the Service Request.

S

Service
A generic name for both Simple and Composite Services. See also Composite Request and Simple Service.

Service Manager
The component that retrieve requests from a Service Queue and selects the appropriate service to process them.

Service Manager Listener
One of the many threads on which a Service Manager can process requests from a Service Queue.

Service Queue
A Queue in the JMS Server that is used to store Service Requests.

Service Request
A Simple Request or Composite Request.

Simple Mail Transfer Protocol (SMTP)
A protocol for sending e-mail messages between servers.

Simple Request

A request to execute a single document-related operation, such as generating a document, rendering from one particular format to another, delivering through a particular channel.

Simple Service

A part of the system functionality that performs a well-defined document distribution function. The service can be invoked through a Service Request.

Submission Service

The Queue to which messages are submitted.

Sub-Request

A sub-tree of an Aggregate Request. It can be either a nested tree of requests, or a just a single Simple Request.

Synchronized Delivery

An operating mode for Delivery Channels. In this mode, all items in a Delivery Package must first be accumulated and available and a “delivery package ready” event must be raised before the actual delivery can begin.

T

Task

An operation that must be performed as part of the processing of a Distribution Request. The operation consists of the invocation of a Simple Service.

Task Graph

A graph, with nodes that represent tasks, events and Event Handlers, and with edges that represent transitions and dependencies between the nodes. Task graphs represent general computation jobs which have been decomposed into modules called tasks that are executed according to some precedence constraints, such as a distribution of the overall completion time.

Test Console

An application that allows you to create, save and manually submit requests to IStream Publisher, as well as monitor the Queues. In doing so, you are verifying that your requests and IStream Publisher are working properly.

Transform Service

A service that allows Transform applications to be called from a request using a command interface.

U

Universal Information Service Resource (UISR)

A method for specifying the location of IStream documents.

Universal Naming Convention or Uniform Naming Convention (UNC)

A PC format for specifying the location of resources on a local-area network (LAN). UNC uses the following format:

```
\\server-name\shared-resource-pathname
```

Uniform Resource Locator (URL)

The global address of documents and other resources on the Internet. The first part of the address indicates what protocol to use, and the second part specifies the IP address or the domain name where the resource is located.

Utility Service

A set of standard services provided with IStream Publisher, including *Run Word macro*, *Concatenate PCL streams* and *Delete files*.

W

Worker

The IStream Publisher component responsible for processing requests.

Worker Machine

A computer on which Service Managers and services run. Workers are the system's basic unit of scalability.

Appendix C

SDK - Encrypted Credentials

This chapter describes:

- *Passing Credentials Securely* on page 186
- *Encrypted Credentials* on page 187
- *Encrypted Data* on page 188
- *Security Keys* on page 189
- *Example of a Credentials Set* on page 190

Passing Credentials Securely

To allow passing credentials (user name and password) in a secure manner in the XML request, IStream Publisher supports passing of the security sensitive information in encrypted form.

The only form of encryption that IStream Publisher will initially support will be symmetric key encryption. This means that both the client submitting the message that contains encrypted credentials and IStream Publisher will have to share the same key. The client uses it to encrypt the credentials and IStream Publisher uses it to decrypt them. The management of the key will be addressed by the IStream Publisher installation and deployment specification.

The Java Cryptography Extension

IStream Publisher uses the Java Cryptography Extension for credentials encryption/decryption. More specifically it supports the Sun JCE provider version 1.2 for JCE 1.2.1. This provider implements the following symmetric key algorithms: DES, Triple DES, Blowfish and PBE. These algorithms are referenced in the XML element encryption-method as DES, DESede, Blowfish and PBEWithMD5AndDES respectively.

Encrypted Credentials

Encrypted credentials in the Distribution Request are contained within an element named <encrypted-credentials>. The structure of this element is presented below:

```
<!ELEMENT encrypted-credentials (encrypted-data)>
```

The <encrypted-credentials> element contains only an <encrypted-data> element. This represents a block of cipher data and will be used in the future more generally for any kind of encrypted information.

Encrypted Data

The <encrypted-data> element contains a block of encrypted information.

```
<!ELEMENT encrypted-data (encryption-method, key-info?,
cipher-data)>
<!ATTLIST encrypted-data
         id ID #IMPLIED>
```

Encryption Method

The encryption method defines the parameters use for the encryption of the data.

```
<!ELEMENT encryption-method EMPTY>
<!ATTLIST encryption-method
         algorithm CDATA #REQUIRED >
```

The algorithm attribute contains not only the name of the encryption algorithm used but also the feedback mode and the padding scheme. The feedback modes and padding schemes allowed are dependent on the encryption algorithm used.

The structure of the string in the algorithm attribute is like this: "*algorithm/mode/padding*" or just "*algorithm*" case in which default values are used for mode and padding.

key-info Parameter

The <key-info> element identifies the key that has been used for encryption.

```
<!ELEMENT key-info EMPTY>
<!ATTLIST key-info
         keyName CDATA #REQUIRED>
```

IStream Publisher is configured possibly with many symmetric keys at deployment time. Keys are generated at deployment and multiple keys can be generated for the use of multiple, different clients. Each key is assigned a name and the client must refer to the key that it used to encrypt the credentials by its name.

Cipher-Data Parameter

This element contains the actual encrypted information (cipher data).

```
<!ELEMENT cipher-data (cipher-value)>
<!ELEMENT cipher-value (#PCDATA)>
```

Cipher data is base64-encoded sequence contained within the <cipher-value> element.

Security Keys

Security keys can be used in IStream Publisher when you want to use encrypted credentials as parameters for source and destination files.

The keys can be generated by a client application and added to a static IStream Publisher configuration at any time.

Every key has three main parts:

1. algorithm name (used in `<encryption-method algorithm="value"/>`, can be DES, DESede, PBEWithMD5AndDES or Blowfish).
2. key name (used in `<key-info keyName="value"/>`, can be any alphanumeric value)
3. key data (actual key, which will be used for encryption by you and decryption by IStream Publisher itself).

For a IStream Publisher installation using IStream Publisher Console, the location of the Security Keys in the Console database is

`Domain/Settings/SecuritySettings/Algorithm[`**`algorithm name`**`]/KeyData[`**`key name`**`]/Value=`**`key data`**

where

- **algorithm name** is one supported by IStream Publisher (DES, DESede, Blowfish, or PBEWithMD5AndDES)
- **key name** is any name used in the element, <key-info> in <encrypted-credentials>
- **key data** is the actual key used for encryption in base 64 encoding.

Example of a Credentials Set

Here is an example of a set of credentials passed in a Distribution Request along with one of the items of the Distribution Package:

```
<distribution-request>
         <encrypted-key id="X1"
carriedKeyName="Credentials">
            <encryption-method algorithm="DSA"/>
            <key-info keyName="E-Delivery Public"/>
         </encrypted-key>

<!-- the rest of the request ommited for brevity -->

<source URL="ftp://host/location/filename.tiff"
contentType="image/tiff"
         <encrypted-credentials>
            <encrypted-data>
               <encryption-method algorithm="DES/CFB/
NoPadding"/>
               <key-info keyName="session key"/>
            </encrypted-data>
            <cipher-data>
               <cipher-value>AJD3242D53EW34JTWK</cipher-
value>
            </cipher-data>
         </encrypted-credentials>
</source>

</distribution-request>
```

INDEX

A

aggregate
    request
        limitations, 47
        processing, 47
    service XML request element, 17
architecture, 111
asynchronous
    call
        call back, 162
        cancellation, 163
        failure, 162
        flow, 161
    invocation
        diagram and process, 161
        web service interface, 158

B

batch, 174

C

call back flow, asynchronous, 162
Calligo
    item, 59
    references to, 12
Calligo Extreme XML InfoSource, *see* IStream XML InfoSource,
cancel flow, asynchronous, 163
cancelled requests, 88
cipher-data parameter, 188
client
    API, 104
        configuration, 104
        interfaces, 97
    exceptions, 103
client.xml, 105
clientjmsqueues.xml, 105
clientsecurity.xml, 106
concatenating
    PCL files, 151
    PCL streams, 41, 75
    PS streams, 43, 77
configuration
    files, 104
        default, 106
    implementation, 106
connection
    creating, 117
    factory, 116
    interface, 116
    opening, 117
contacting Skywire Software for help, 14
content service, 23
contenttype
    details, 54
    distribution request, 53
    issue, 64
    parameter, 61, 65
cover pages, 91
credentials
    deleting files, 45
    set example, 190
critical event handlers, 72
custom fields, adding, 153
custom service deployment, 135

D

DAOfactory, 152
data access objects, 138, 151
decoupling the client, 111
default configuration files, 106
deleting
    distribution items, 60
    files, 44
delivering
    CLG files through InfoSources, 19
    content to a repository, 36
    cover pages to a fax and printer, 92
delivery
    channels, 67
        and delivery packages, 138
        settings, 68
    items
        ready, 70
        removing, 145
    items, adding, 145
    package ready, 70
    packages, 138, 144
    preference considerations, 63
    service, 36
    to repository, 66
    units, logical vs. physical, 64
deployed application server JVM log files, 167
deployment, 157
detailed request parameters, 22
distribution
    complete, 71
    item, 142

description and syntax, 60
overview, 59
parameters, 61
package, 141
overview, 59
ready, 70
request, 88, 140
API, 137
completion, 108
completion, notification of, 108
entity, 141
event handler example, 145
failure policy, 56
metadata, 74
overview, 53
sample, 80
structure, 55
troubleshooting, 57
XML-based, 53
service, 52, 57
cleanup, 145
extending, 137
state DAO, 151
distributor
factory, 97
creating an instance, 97
interface, 98
interface, 98
distributor.xsd, 16
document conventions, 10
documentation
Publisher, 13
duplexing options, 42, 76

E

element, 66
embedded
data, 175
XML, 25
with plain data, 26
encrypted
credentials, 186, 187
data, 188
encryption
flags, specifying, 44
method, 188
entities, 138, 139
error
log levels, 89
messages, 58
errorinfo table, 87
event handlers
concatenate PCL streams, 75
critical, 72
distribution request
metadata, 74
processing, 70
distribution request, in, 138
IDs, 147
implementation, 148
multiple, 72
overview, 137
response, 138
with proxy, 150
without a proxy, 149
events, 70
supported services, 71

F

fail fast failure policy, 56
failed requests, 88
failure flow asynchronous call, 162
files, referencing, 18
flows of WSI calls, 159
folder sources, 45
functionality
categories, 112
determining supported, 111

G

generate Calligo document
XML sample, 26
generate Calligo document service, 23
generate IStream document service *see* generate Calligo document service
glossary, 177
graphics, 29
guide overview, 9

H

header page, 75
template example, 173
help, contacting Skywire Software, 14
high and low parts, 84

I

identifiers, 125
implementation, 151
installation, 157
instant delivery, 67
interactive
batch
embedded XML data, 174
mode, 174
IStream
XML InfoSource, 23
IStream document item, 59

item delivered, 71

J

java
  code, 130, 135
    example, 130
  cryptography extension, 186
JMS message header and properties, 20
job, single, into multiple streams, 75
JVM log files, 167

K

key-info parameter, 188

L

live request message status, 84
logging, 106
logical vs. physical delivery units, 64
loopback service, 58
low and high parts, 84

M

mapping, 88
metadata
  accessing, 141
  elements, 74
MIME types, 54
monitoring requests, 83
multiple
  event handlers, 72
  streams into one job, 75
multithreading, 117

N

non-critical, 72
normal flow
  asynchronous call, 161
  synchronous call, 159
notification of distribution request completion, 108

O

object
  metadata, 122
  relationships, 140
opening the connection, 117
operating mode, 67, 137
optional parameters, 42, 76
overview
  Publisher, 11
  WSI architecture, 157

P

package delivered, 71
parameters, 16
path, 115
perseverance failure policy, 56
physical vs. delivery units, 64
Publisher
  overview, 11

Q

query, 115
queues, 11

R

recipient
  description, 62
  element, 62
  entities, 142
  package, 64
    ready, 70
  parameters example, 63
  relationships, 143
reference language, 113
  context, 114
referenced
  XML data, 24
    code samples, 25
    embedded, 24
referencing files, 18
RefID example, 63
rendering
  IStream doc to Word, 35
  PDF to PCL, 33
  PDF to PS, 33
  PDF to TIFF, 34
  service XML sample, 32
  services, 28, 65
  TIFF to
    PCL, 34
    PDF, 34
    PS, 35
  Word to
    HTML, 28
    PCL, 29
    PDF, 30
    PS, 32
    TIFF, 32
    TXT, 32
render-param subelement, 65
renditions, 124
repository
  adapter, adding, 129
  API overview, 110
  file, 60
  folder, 60

interface, 119
objects, 121
accessing, 111
content, 128
request
log
agent configuration, modifying, 153
database, 85
database table, 153
message, customizing, 153
table, customizing, 153
messages, 84
metadata, 21
monitoring, 83
table, 85
tracking, 83
validating, 17
requests
queues and, 11
required properties, 116
response
exceptionlistener interface, 100
handler interface, 158
listener interface, 100
parameters, detailed, 22
resubmitting failed or cancelled requests, 88
retry flow - synchronous call, 160
run Word macro, 41

S

sample, 169
aggregate request, 171
deliver-to-email, 170
distributor configuration, 108
structure, 135
XML fragments, 24
schema name, 113
SDK overview, 96
security keys, 189
sequence number, 72
services, 100
and events, 71
interface, 149
invocation sequence, 101
request example, 132
session, 99
interface, 99
simple service, 15
creating and adding, 134
XML request elements, 16
simple use case, 134
Skywire Software, contacting for help, 14
status table, 86
statusorder table, 87
streams, 75
header page, 41
structure, sample, 135
support checklist, 14
synchronized delivery, 67
synchronous
call, 159
call with retry, 160
invocation, 158, 159

T

technical support, 14
template URL, 42, 76
tracking request, 83
Transform service
details of, 49
MIME types, 54
using in a distribution request, 79
troubleshooting
distribution request, 57
WSI, 167

U

uniform resource identifiers, 113
unique request ID, 84
updating delivery items, 137, 145
URI examples, 115
URL
URLconnection, 127
URLstreamhandler, 126
URLstreamhandlerfactory, 125
utility services, 41

V

versions, 123

W

web services
applications, 156
interface, *see* WSI
wildcards, 45, 60
Word
table of contents, updating, 28
to HTML related files, 28
WSI
architecture, 157
benefits, 156
client examples, 166
configuring, 165
configuring in the console, 165
flow of calls, 159
general information, 157
log files, 167

methods, 158
overview, 156
troubleshooting, 167
workflow, 158
wsdl, 164
WSI.log, 167

X

XML InfoSource, 23
XML schema, 16