TIBCO iWay® Service Manager

Programmer's Guide

Version 8.0 and Higher
March 2021
DN3502121.0321

Contents

Chapter 1

Programming Your Applications in iWay Service Manager

iWay Service Manager (iSM) is a fully featured message analysis and execution system. In iWay Service Manager, applications are defined by the configuration of actions to be taken in response to any specific message that arrives. As is the case with any generalized system, the set of supplied actions may not exactly match the needs of your specific application. iWay Service Manager is designed to be easy to modify using custom programming objects to meet specific application requirements. This chapter describes the schematic architecture of the engine, and provides a context for understanding and using the development services that iWay Service Manager supports.

In this chapter:

Overall Programming Guidelines

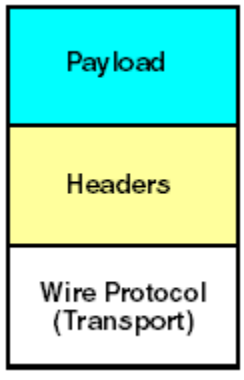
When coding any iWay Service Manager exit component, you must adhere to the following guidelines:

- All code must be reentrant. The code can run on multiple threads. Avoid use of monitor locks unless they are required.

- Avoid over tracing. Only issue traces that are required to understand and debug a problem. Use only the provided trace levels to issue trace or log messages. Properly categorize the trace levels (for example, DEBUG, DEEP, PARSE, and so on). For more information on the supported trace levels, see *Tracing* on page 147. Never use System.out tracing, since the server cannot capture or control these traces. For a RuntimeException, include the stack trace, otherwise do not include the stack trace. Such traces are cumbersome and add to trace volume.
- Only use classes and methods that are documented in this manual or in the released Javadoc. iWay Software does not guarantee other interfaces.
- Always use only the documented metadata functions in the component to pass metadata to the system. Never use your own configuration files, which cannot be managed. The value in a configuration element can, of course, be stored in an external file using iFL functions such as _property() or _file() to retrieve the information.
- Always query the server for information related to the environment, such as the current context (for example, iwayhome, the configuration name, and so on). This information is available through direct calls to the manager class and is also stored in special registers. Never rely on system environment values for such information, since different systems and application servers may maintain that information in an idiosyncratic fashion.
- Never use System.exit() regardless of the problem you encounter. Handle errors carefully. If you catch an exception, you must recover from the error and not continue processing bad documents. An exception caught by the engine is handled automatically by terminating the transaction.
- If you need to issue an exception, use the XDException class or a subclass of XDException. This allows the server to recognize logical errors. Never simply issue Exception(). If you trace an exception, try to avoid tracing the stack trace except for a runtime exception. The actual trace should be sufficient to track down the logical (non-runtime exception) issue.

Overview of the Adapter Process

iWay Service Manager is designed for hosting adapters and for enabling messages received from one adapter to be sent to another adapter. In general, the messages arriving at the server have three main parts as illustrated below:

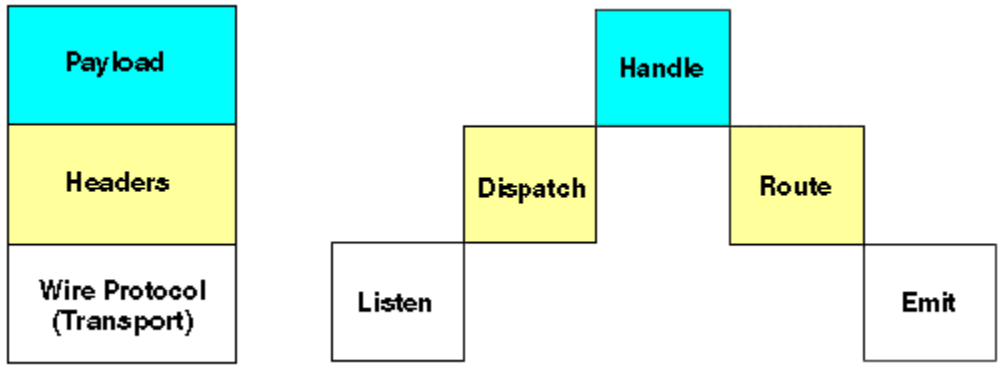


Payload. The actual content of the message to be processed.

Message headers. In HTTP, these are the HTTP headers. They may also include encoding such as multipart MIME headers. In SOAP, these are the SOAP headers and, in SOAP with attachments, the MIME headers. In MQ Series, these values are in the MQMD header.

Transport. For instance, in HTTP, this is TCP. In SOAP, this can be HTTP, SMTP, or FTP.

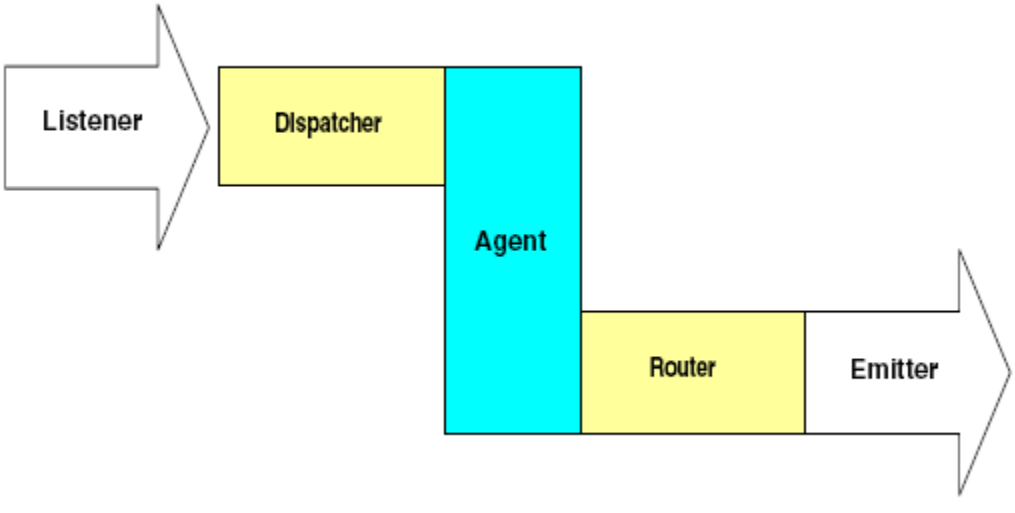
An adapter has to handle all three layers of the message and iWay Service Manager is set up to make it easy to configure each of these individually. The way these layers map onto functions in iWay Service Manager is shown in the diagram below.



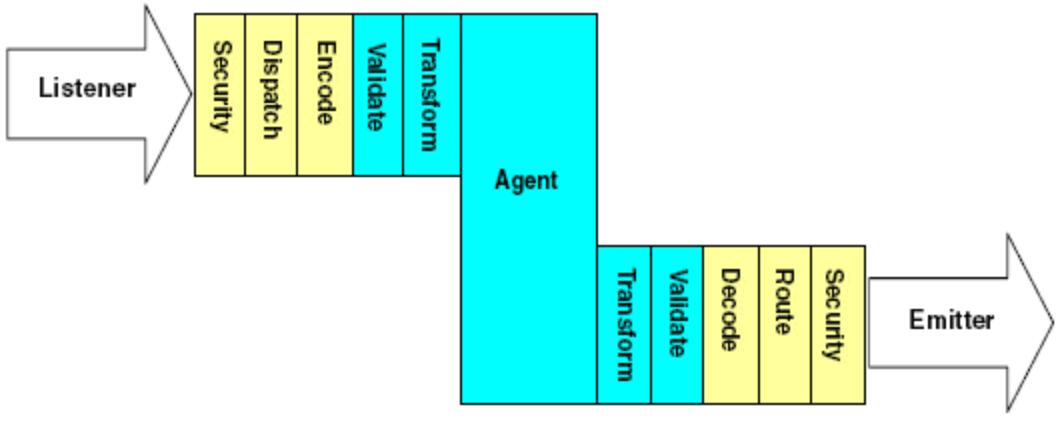
The transport is handled by a listener (for incoming messages where iWay Service Manager is acting as a server) and an emitter (for outgoing messages where iWay Service Manager is acting as a client). This is the level where components (listeners and emitters) are written. In most cases, the transport components delivered with the product are sufficient; there is no need to create new listeners or emitters to set up an adapter.

The headers are handled by dispatching and routing. This is the middleware level where most of the added value of iWay Service Manager lies and where the iWay preparser exits are executed. Although a complete set of exits is provided with iWay, it may be necessary to create a new exit, for security or for formatting a message into XML. Although a non-XML message can be routed through the system and acted upon by later exits, iWay Service Manager operates most effectively with XML documents. If your incoming message is not natively in XML and if one of the standard iWay-provided preparsers does not suit your needs, you will want to write a preparser exit.

Finally, the message is handled by the application level. This is where iWay business agents are executed and the level where integration applications are written. iWay business agents are exposed as business services in the iSM console. Although a large number of business agents are provided with the product, it may be necessary to write a custom business agent for specific programming needs. In many cases, it may be enough to assemble the business agents provided into a process flow. For more information, see the *iWay Designer User's Guide*.



In implementation, this step is more conducive to application developers. There are various exits that can be used in dispatching and routing. If these are added, the diagram looks like:



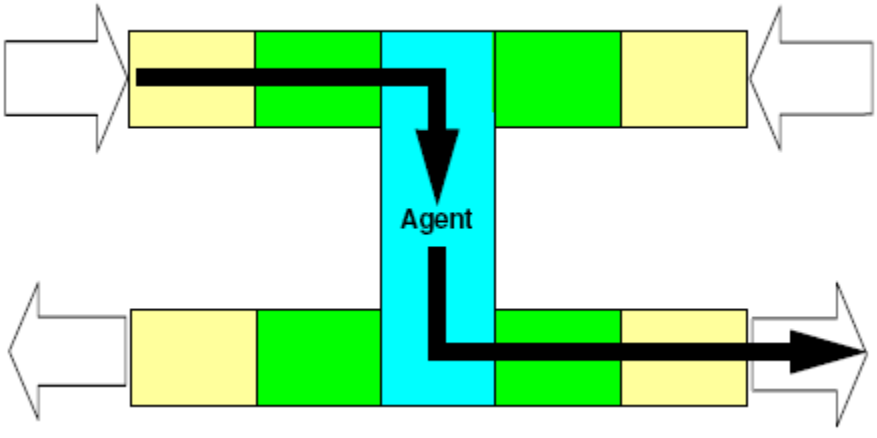
This diagram illustrates the flow of a document as seen by a single adapter.

The first step after receiving the message is always to verify the security of the message – that is, to verify that the sender is authorized, to check that the message has not been changed, and to decrypt any part of the message that has been encrypted. iWay has security exits for many common security algorithms including W3C Digital Signature, ebXML, and PGP, among others. It is also possible from the exit to check credentials in an external security manager such as Netegrity, Active Directory, or an LDAP server.

As all incoming messages need to be changed to XML before their payloads can be handled, there is a dispatching step that is called for non-XML messages that encodes their contents as XML. This is called a preparser since it occurs before the message is parsed into the iWay document tree. Similarly, there is a step called preemitter that can encode an XML document into a non-XML format. iWay Service Manager supports many non-XML formats for preparsing and preemitting thereby eliminating the need to code a specific exit.

Finally, there is an exit to validate the contents of the payload. This validation can use iWay Service Manager rules described in *iWay Service Manager Rules System* on page 171 or it can use an XML validator such as Apache XERCES. It is also possible to do both.

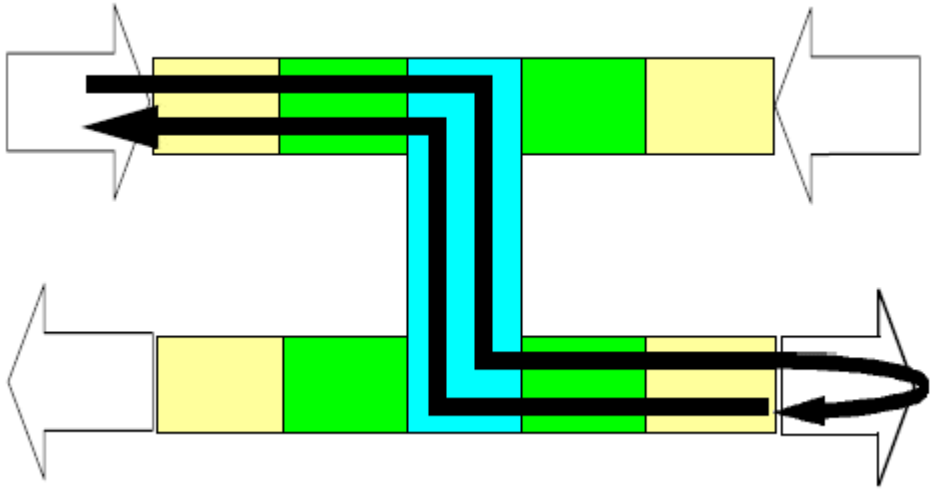
Because an integration application generally needs two adapters, a typical use of iWay Service Manager consists of two such adapters, one for handling the incoming message, for example, from SAP, and one for handling the outgoing message, for example, to PeopleSoft. This is illustrated below.



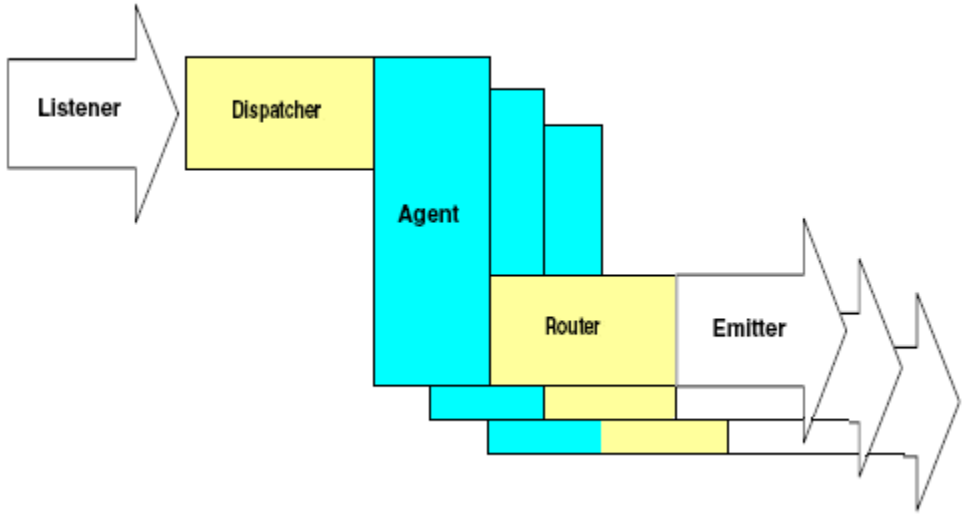
The arrow on the diagram shows the one-way message. This is a one-way adapter as the message can only come in on the left and go out on the right. This configuration would support two scenarios:

- A one-way message. If the message is one-way, the listener does not wait for the business agent to complete before completing the listen operation. For an asynchronous protocol, such as MQ Series, this is done without sending an acknowledgment. For a synchronous protocol, such as an HTTP POST, this is done by sending back the HTTP 200. Note that one-way messages are always asynchronous at the application level even if they use a synchronous protocol at the transport level.

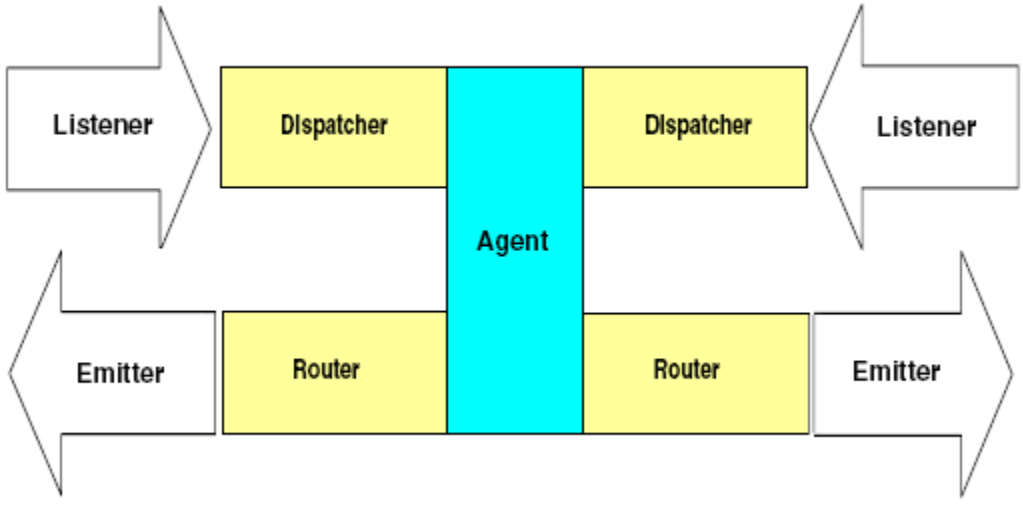
- A request reply where the reply is received synchronously at the emitter and the listener is holding the request until the reply document is made available. This is a synchronous interaction as shown in the following diagram:



Although the previous diagram shows a single business agent, business agents can be chained or be part of a flow. In this case, there can be multiple outputs for a single input either to the same adapter or to different adapters. This is illustrated below.



Adapters can also be bi-directional as shown below.



This is a bi-directional adapter as there is a listener and emitter on both sides. Messages can come in on the right or the left and be emitted on the left or the right, respectively.

To accomplish its mission, iWay Service Manager executes a defined flow as the message travels from input to output. This flow is standard for all messages, but the execution steps selected and the processing at those steps varies by message type. The steps of the flow take the document from input, conversion to XML, parsing, transformations, rules validations, business processing, and then back through transformations, conversion from XML to the native transmission format, and then transmission through a port itself. Each of these steps is performed by an exit, a routine invoked to perform the specific action at that step.

In some respects, iWay Service Manager can be thought of as simply a platform for the execution of exits. No document processing is performed by the server itself. Instead, the server assembles the appropriate exits to accomplish the business purpose of the document.

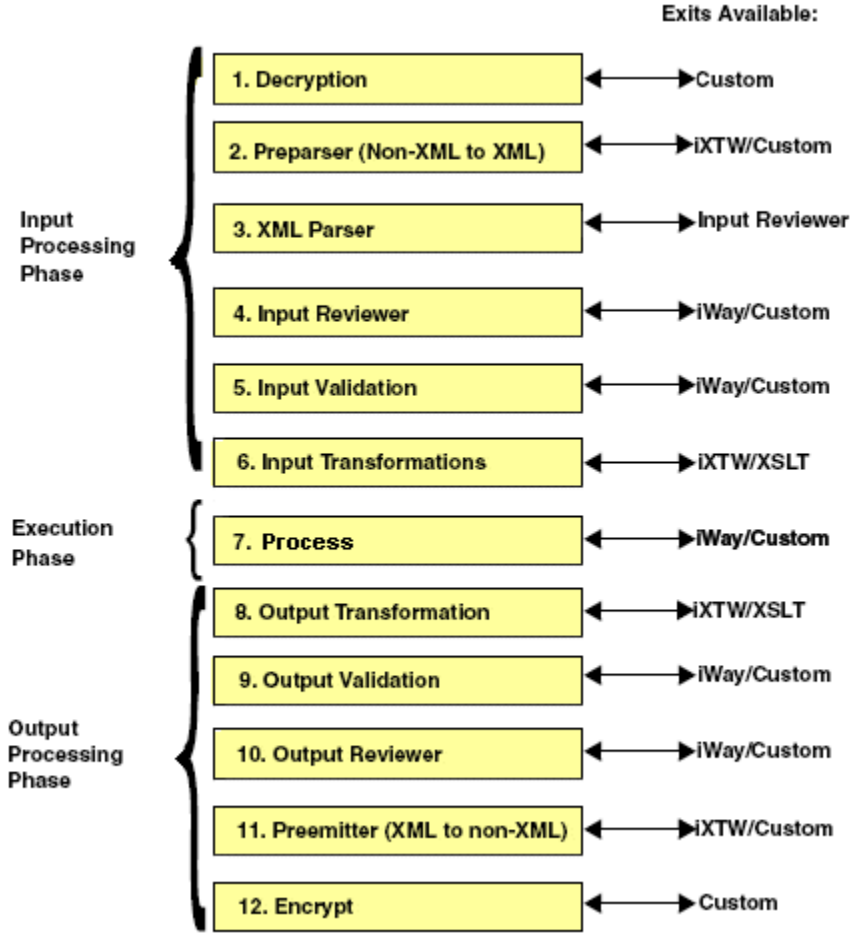
Application programmers can assemble flows by using pre-built exits. These exits can be supplemented by application-specific code that can be intermixed with and scheduled along with exits offered by iWay.

The iWay Service Manager array of protocol handlers can also be supplemented by new protocols tailored to specific application situations. These protocol handlers, called listeners and emitters, service the input channels and output ports that represent the iWay Service Manager interface to the outside world.

Document Life Cycle

When a document is received, it is normally put through the optional input transformation and validation of steps 1–6. It is then ready for execution by an agent in step 7. This is typically where an adapter is used for data access. Alternatively, a custom agent may be involved. Next, the response document is prepared using the optional output transformation and validation of steps 7–12.

The life cycle of a document, from input to output, is shown below. On the right side are the optional programming exits that may be configured to call iWay-supplied or custom-written modules.



Reference: Steps of Document Transformation

The following are descriptions of each document transformation step:

1. **Decryption.** Applies a decryption algorithm (this must be supplied in a custom module).
2. **Preparse.** (Required to transform non-XML to XML). You can choose from several supplied preparsers or use a custom preparser.

   Note: Unless the listener allows the documents, XML is required from here on; otherwise, an XML parser error occurs.
3. **XML Parser.** For XML messages, this parser converts the incoming stream to an internal tree for further execution. Flat (non-XML) messages bypass this step and enter the process flow (or agent stack, see step 7) in non-XML (flat message or stream) form.
4. **Input Reviewer.** Analyzes the document. This stage is usually used to extract and handle headers.
5. **Input Validation.** Applies validation using the rules validation engine. Rules are supplied when adapters are installed or may be custom written to input documents.
6. **Input Transformations.** Applies Process Transformer or XSLT transformations to input documents received by a listener.
7. **Process Flows/Agents.** Process flows are logical constructions consisting of execution agents and logical flow controls. Agents (flow nodes) are the components that execute the business logic needed by your application. You can choose from many supplied iWay agents that implement logical operations or can access iWay data adapters and JDBC data sources. You may also choose to develop custom-written agents that can access third-party applications and systems or implement application-specific functionality.
8. **Output transformations.** Applies Process Transformer or XSLT transformations to all outbound response XML documents.
9. **Output Validation.** Applies validation using the rules validation engine. Rules are supplied for adapters, or may be custom written to output documents.

   Note: Non-XML is allowed from here on.
10. **Output Reviewer.** Analyzes the document. This stage is usually used to add headers.
11. **Preemit.** (Required to transform XML to non-XML) You can choose from several supplied preemitters or use a custom Preemitter.
12. **Encrypt.** Applies an encryption algorithm (this must be supplied in a custom module).

General Processing Exits

In addition to document life cycle exits, iWay Service Manager supports exits that tailor general processing. You can develop such exits to meet your specific needs. Among the exits supported are:

Transaction Log Exits. Used in place of (or in addition to) the general transaction log recorder, this exit records the event information to a specific destination or provides record selection. An example of such an exit is writing the transaction information to a JDBC data base.

Trading Partner Agreement Drivers. Allow the TPA() function to obtain its information from a specific data format and location.

Developing For Your Own Needs

iWay Service Manager provides for a wide range of business exits for user-customization. Exits exist in document execution, parsing, output, validation, transformation, and security.

Exits are written in Java, and can take advantage of programming services offered by the core system during their execution. Services such as document parsing, document modification, tracing, and use of an internal object store are all made available as appropriate to the task of the exit. Much of the standard functionality of the system is provided through iWay-written code run at the exit points, which is simply replaced or supplemented by user-written code for special tasks.

There are several standard exit points:

Document execution exits handle the actual payload. These are often called business agents. They are at the application and business level. Business agents execute following all input transforms, and represent the turn-around point at which an input document becomes an output document. Business agents typically encompass the business logic associated with the document. Business agents can be stacked, such that the output of one business agent is passed into the next business agent, or multiple business agents can be executed in parallel.

Preparsers process the input before it becomes a processable document. Typically, preparsers are used to change non-XML into XML, for example an input HIPAA document in EDI format that is to be processed through the system. Preparsers can be stacked, so that the output of one preparser can be passed to the next preparser. Input to the first preparser is a byte stream, and output is a string. Subsequent preparsers in the chain accept strings and emit strings. Following the last preparser, the engine parses the string as an XML document.

Reviewers act immediately following the XML parse, and immediately before the preemitter exit stage. Reviewers act in the same manner as business agents: an input document becomes an output document and chaining is supported. Document execution exits are intended to process payloads, while reviewers are intended for handling headers.

Preemitters process the output document just before emit. Typically, preemitters turn the XML into a non-XML form, such as EDI. Preemitters can be stacked. The first preemitter accepts as input a document, and emits a byte stream. Subsequent preemitters accept the byte stream output from the first preemitter, and emit a byte stream.

Encryptors accept byte streams and emit byte streams. Encryptors are the first and last exits through which the document passes.

Channel initializers are exits that are invoked when the protocol listener is first started. These exits can perform any special processing required such as initializing an external system, and then return control to cause the listener to either continue startup or to terminate. Use of these exits is rare, and you will probably never need to develop such an exit.

Validation exits receive control through the rules system, by which nodes of the incoming and outgoing document are checked for validity. The rules system is discussed in *iWay Service Manager Rules System* on page 171.

Protocol Components

In addition to business processing exits, iWay Service Manager allows user code to implement protocols that receive and send documents. An example of such a user-written protocol is the bi-synchronous communications support implemented through hardware supplied by Protogate, Inc. The Protogate listener and emitter components integrate with the server and appear in all respects identical to the built-in protocol handlers. For more information on protocol components, see *Writing Protocol Components* on page 95.

Encoding

Messages consist of a sequence of characters. A character itself is an abstract notion; a character is the assignment of a group of bits to a glyph, or some displayable instance of a character. Encoding refers to the sequences of bits assigned to represent related characters.

Since there are eight bits in a byte and a seemingly unlimited number of characters that these bits might represent, the same sequence of bits is often assigned to multiple characters. The bits themselves refer not to the character as unique among all characters, but rather to a specific character within a limited group of characters. An example of this is letters in a local alphabet such as Hebrew, English, or French.

iWay Application System Adapter for PeopleSoft must be able to recognize and react to specific characters; therefore, it is important to identify for any message exactly to which sequence or group of characters the bits of the message belong. Only with this information can iWay Service Manager properly understand and treat the message.

Because XML documents can originate from sources using many languages, the encoding of the specific document is, as standard, included in the document. iWay Service Manager recognizes this encoding assertion, and properly respects it for analysis and handling of the XML message. Any XML message without a specific encoding declaration is given a default encoding by examining the first few characters of the XML message itself. The usual default assignments are ASCII or EBCDIC. The responsibility of declaring the correct encoding lies with the originator of the document. Encodings for XML documents are expressed by names assigned by the Internet Assigned Naming Authority (IANA). Because messages are processed using the Java language, the appropriate Java encoding must be used to convert the sequence of bits into usable information. To this end, iWay Service Manager converts the IANA names to their appropriate Java encoding equivalents.

Non-XML documents do not carry their encoding in any manner that iWay Service Manager can recognize. Such documents need to be processed into XML by preparser exits. In this case, iWay Service Manager needs to know what encoding to apply to the message. The listener configuration must specify the encoding to be used. There is a one-to-one mapping from IANA names to Java names, however, there is no mapping in the other direction. In addition, Java names can vary by platform and locale. Therefore, listener encoding configurations are defined directly as Java encoding names by selecting one of the offered values in the configuration entry for the listener, or by entering the appropriate Java encoding name directly.

The default is the platform system encoding used to read and write characters into the native file system.

Java Properties
Listed below are the java system runtime properties that are in effect for the base configuration of this server.

Java System Runtime Properties	
awt.toolkit	sun.awt.windows.WToolkit
file.encoding	Cp1252
file.encoding.pkg	sun.io
file.separator	\
iway.build.date	SOCRATES Wed 09/30/2009 05:25 AM EDT
iway.build.info	ga.22561
iway.config	base
iway.home	C:/Program Files/iWay60/
iway.startup.time	1265899112843
iway.version	6.0.1

iWay Service Manager applies the listener encoding as a default whenever it cannot determine the encoding in any other more appropriate manner.

The ISO-8859-1 encoding does not interpolate characters; it assigns characters on a byte for byte basis from the message to the iWay Service Manager representation of an encoded message. Therefore, this is a good encoding to use for messages that are in binary and do not directly represent a sequence of glyphs, for example, a JPG message.

Specifying the wrong encoding for the message is a common source of problems. Usually this is evidenced by the inability of iWay Service Manager to parse the message, resulting in an error. For example, it is a common, erroneous, practice to assign the encoding UTF-8 to every message under the mistaken assumption that this is the "cover all cases" encoding. In reality, UTF-8 is a variable bit sequence that is very specific; some characters (the ASCII lower 127) map correctly. However, other characters above 127 consist of bit patterns that may or may not be valid (but rarely correct) UTF-8 encodings. This results in parsing errors.

The specified encodings must be available for use by iWay Service Manager. Encodings are provided to Java in the I18N.jar file. You must obtain the appropriate I18N.jar for your locale and platform, and load it into the iWay lib directory. This jar file can be obtained from www.javasoft.com.

Flat Documents

Although the engine is optimized for handling XML documents, including non-XML that passes through preparsers to create XML, it is possible to pass non-XML through the engine stages. Non-XML are referred to as flat documents, and flatness can be set by the listener or explicitly by an exit. Flat documents do not pass through the preparser and reviewer exits, but are passed through business exits.

Flat documents store the message as a byte array, a String, or an attachment array, depending upon the message itself.

Incoming messages can be established as flat documents by setting the appropriate listener parameter, so that all documents arriving on that listener are treated as flat. An exit can also store flat information in the document, in which case the document is marked as flat. Another exit can return the document to XML state by storing an XML tree.

A common use for flat documents is protocol conversion adapters, in which a message is retrieved on one protocol and emitted on another – if no transformation or processing is required, a performance benefit can often be obtained through the elimination of the XML conversion and parsing.

Protocol emitters can emit messages from both XML and flat documents.

Routing

Routing is accomplished in the system by looking for a routing control element. A routing control element in the dictionary is a <document>name</document> section, in which the name of the element (its value) is by default the root tag of the document to which it applies. This can be overridden by an explicit condition instruction, which always takes the form of a child as follows:

```
<if>
     <cond>conditional expression</cond>
</if>
```

If the conditional expression is omitted, the default is _isroot(document name).

A search follows the routing algorithm, testing the condition for truth. The first routing element that evaluates to 'true' is selected.

There are two routing algorithms in use, called precedence 1 and precedence 2. In general, we use precedence 1.

The order of search for precedence 1 (the default) is:

1. Look in the current listener definition for an appropriate <document> element. Take the item we need from the document if found.
2. Look in global documents for an appropriate <document> element. Take the item we need from the document if found. In iSM, v1 global documents are not definable. The run time router considers any that have been stored by the iSM console or a package to be valid routes.
3. Look in the current listener for what we need. This is the default route.
4. If all fails, ignore the optional operation or set an error if it is a required operation (agent or process flow).

Once a route is located, it is the source of the operation. That is, routes do not inherit. So if a route is selected and it does not have the desired component element, the run time does not look elsewhere.

This search takes place several times during the processing, so that as an incoming message's root element name changes, alternate routes might be selected. We refer to this as dynamic routing.

If the selected <document> carries the type='fixed' attribute, once selected it remains the route for the remainder of the handling of the current message. This adds performance by avoiding the frequent route evaluation but limits the flexibility. On the other hand, most documents do not change routing so fixed routing can be a good choice.

The actual task elements, for example, the process flow to run, are taken from the selected routing location. Each can contain a condition (<if > tag) which if evaluated to true causes the element to be selected. If no such conditional element is present, the default is true and the element is selected.

As an optimization, the transform nodes (for example, in_xmlg, in_xslt) can be aggregated under the optional in_transforms element. This is also true of out_xxx transformations. It is not necessary to aggregate, and if the aggregation element is omitted it is created automatically when the document route is loaded. Thus, old routes can work with no change.

Some best practices advice is:

1. Define routes in the order you want them examined in each section. The search continues until it gets a true condition. Therefore, unless you want a specific dynamic order, defining them in order of probability is best.
2. Make the tests as simple as possible. For example, _isroot('x') executes slightly faster than **_root()='x'** or **xpath(/*/name())**, although the difference is very slight. If you want to use an OR such as _root('x') or _root('y'), specify the most likely first. This avoids having to evaluate the second term. Similarly, if AND is used, specify the least likely term first to avoid unneeded evaluation of the subsequent term.
3. Use comments liberally.

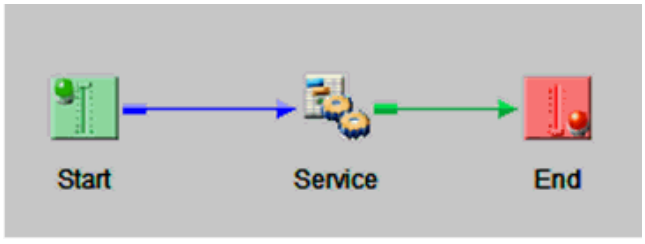
Programming Terminology

Throughout this documentation, many iWay Service Manager-specific objects have a name that begins with the letters XD, such as XDDocument. This has been done to make it easier for the programmer to differentiate between iWay Service Manager objects and other objects in a program. There is no requirement that processing exits such as business agents or preparsers use this convention, or exist in any specific package. On the other hand, some objects that are loaded by design pattern name, such as protocol emitters, must have names that conform to their design pattern and must be filed in a specific package. Where such rules apply, they are clearly stated in this manual.

Process Flows

A process flow consists of service nodes (agents) and the edges that connect them. Edges have names (for example, success or fail_connect) that control their characteristics and use. Nodes provide services within the process flow, and although they have names, the names are not used for any purpose other than tracing the execution of the process flow.

The following image is an example of a simple process flow that contains three nodes (Start and End nodes, and one that performs a service).



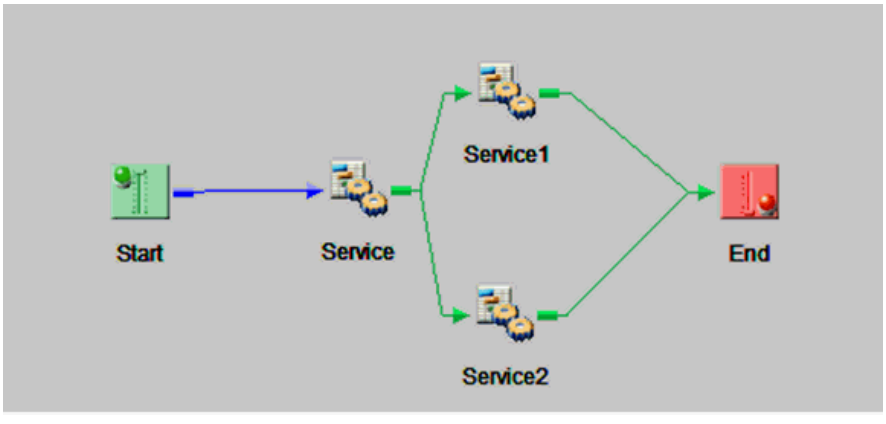
The arrows that connect the nodes are the edges. In the example, the edges (relations) are the OnComplete (from Start to Service) and OnSuccess (from Service to End) edges.

Process flows are managed by the Process Flow Interpreter, which handles all aspects of process flow execution.

As the process flow begins, the XDDocument reaching it passes to the first node of the process flow. From there it passes from node to node, through the edges. Each node examines, possibly modifies and re-emits a document back into the process flow.

If the process flow is running in Local transaction mode, on completion of the process flow execution the transaction handler determines how the process flow ended (reached the end or failed during run) and calls commit() or rollback() as appropriate.

In cases in which more than one edge should be followed, the process flow is multithreaded. In the example below, both Service1 and Service2 receive the output copies of the Service node. The documents are distinct, and can be modified by either edge without affecting the document on the other edge. Programmers are reminded that documents occupy memory, and application developers are cautioned that memory may be limited.



Documents in Error

Some nodes may mark the document in error. Usually a status document detailing an error is also marked. A document that exits the process flow in an error state will be sent to the errorTo address(es) if available. Otherwise, the document is sent to the replyTo addresses. The process flow provides a node (XDDocAgent) that can set or reset the error status as required.

Emitting Information in a Process Flow

Process flows offer two methods of emitting information that differ in their use:

- emitters
- emit agents

An emitter adds an address to the current document that is passing through a process flow. When the document is finally emitted from the end nodes of the process flow, it eventually reaches the emit stage, as described above. At that time, the document is sent to the list of addresses associated with the document. No emit occurs as the document passes into the emit nodes. Rather, it is emitted only when the channel reaches the output stages.

An emit agent causes the current document to be emitted when the node is executed. Thus the process flow can send a message, and possibly act on the reply. Emit agents do not associate an address with the current document, but rather executes the emit operation at the time called. An emit agent does not change its behavior based on the error state of the document.

Some emit agents can operate selectively (by configuration) within or outside of a transaction boundary, For example, an MQ emit agent can be configured in this way. Doing so permits an error message to be placed on a queue (in this example) that will be committed immediately and will not roll back if the process flow is in error.

Understanding How Process Flows Handles Edges

As the service node ends and passes control back to the Process Flow Interpreter, it returns the name of one or more edges that are to receive the output document. The edge is returned as a String from the agent, using commas to delimit multiple edge names if necessary. For example, an HTTP node that cannot connect to a server might return:

```
fail_501,fail_5xx,fail_connect
```

The interpreter first examines the edges, and determines which edges are actually connected. It compares this with the list of possible edges. The document flows down all edges with names that match an offered name, under the following rules:

1. The String return value execute method of an agent determines the process flow edge or edges that will be followed. You can return more than one edge, using comma as a delimiter.
2. If the edge name starts with fail_, edges are tested until a fail_ edge is found wired. Thus the most exact edge is the one followed. In the example, if fail_501 is wired, it will receive the document. If not, then fail_5xx, and then fail_connect. A document follows only one fail edge.
3. If the edge name starts with fail_ and no specific edges are wired, then the process flow looks for a general fail edges (selected as failures from the relations wizard in iWay Designer).
4. If an OnCompletion edge is offered, any edge name from the node is considered a match, in addition to any specific wired edges.
5. If no edge is found to be followed, an error is thrown. Note that a node that throws an exception cannot return an edge name, and so an OnError edge is assumed to have been returned. An OnError edge will not follow an OnComplete edge. If you write agents (described later in this manual) rigorously, avoid throwing exceptions-use fail edges.
6. If an OnError edge is wired, the node/flow is not considered to be in error, as the application process flow has elected to handle the error.

7. If an OnError edge is not wired, the interpreter searches upward in the process flow looking for a catch node. If one is found, the error message document is passed down that edge, and the process flow is not considered in error (again, the application has elected to handle the error).
8. If the search reaches the Start node, then the process flow is considered in error. In such a case, the nodes of the transactional process flow receives a rollback() call. Otherwise, a commit() call will be received when the process flow ends.

Additional rules:

1. While you can use the relations wizard in iWay Designer to create edges of any name, never create one starting with a dollar character ('$'). These are reserved for the system.
2. A successful node should always return **Success**.
3. iWay suggests that you use an OnCompletion edge only in cases in which a node cannot fail. An example is a Move or Copy node, the edge coming from the Start node, or the edge followed by an iterator on completion of the iteration.
4. Be sure that the getOPEdges() method of a node returns a String[] with all edge names that the agent could return. The only exception to this is the return of one or more of a long list of possible returns, such as XOPEN SQL codes. The user can, using wizards in iWay Designer, enter a specific name for an edge not offered in the metadata of the node.

Chapter 2

Writing Business Components

Business components (also called exits) perform the actual application processing of messages in iWay Service Manager (iSM). Business component use is coordinated through configured channels and process flows in iSM. While iWay provides a wide variety of such components, from time to time a specific application requires some business service not available through the usual iWay libraries. In such cases, application developers can write their own components to satisfy their specific requirements.

This chapter describes how to write iSM business components. Exits include the development of nodes of a process flow called a service in iWay Integration Tools (iIT). All exits, regardless of whether they are to be used as agents (services), preparsers, or preemitters extend a common class called *XDExitBase*, which manages metadata and connects the exit component to iSM.

In this chapter:

Getting Started With Business Components

Business components generally transform a message in some manner, and then pass on the message to the next component along the channel or bus. Messages are enclosed in containers called XDDocuments, which contain both the message and the state of the message. Business components can inspect, operate on, and change both the message and the state by working with the passed in XDDocument. The XDDocument is described in associated Javadoc.

The exit components are written in Java, and must meet the standards for programming discussed in this manual. Generally, the exit must perform properly in the server. It must never issue a System.exit() or throw exceptions, but should be reusable as a minimum, and reentrant where possible. Additionally, the exit must be stateless and not take actions that will tie up the server in long operations.

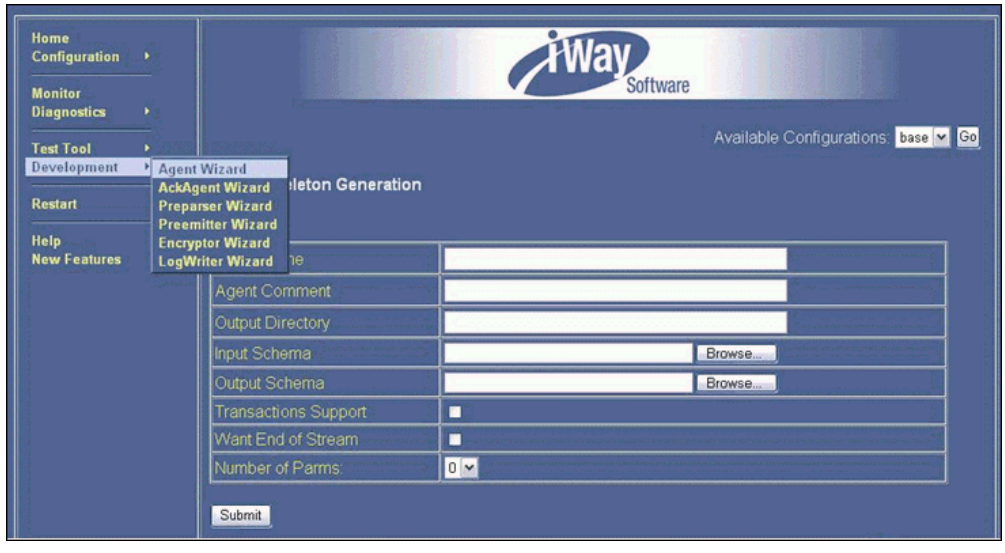
For some components such as business agents, iIT offers a wizard to generate the *boiler plate* portions, including metadata, required for the component. Other exits, such as the preparser or preemitter, can be generated using a wizard in the *blue console* which is deprecated in iSM release 7, but still performs some useful services. In future releases, these exits will be generated by the iIT wizard. Where supported, the iIT version should always be used in preference to deprecated wizards.

All exit types offer the same metadata APIs to iSM, and differ only by the class that they extend.

This section discusses message handling exits.

Exit Type	Use	Extends
Agent (iIT Service)	Process flow service node.	XDAgent
Preparser	Immediately following message acquisition, prepares to parse if XML form is wanted.	XDPreParser

Exit Type	Use	Extends
Preemitter	Formats the actual outgoing message, based on destination type.	XDPreemitter



The Wizard prepares a skeleton component based upon the information given in the forms of the wizard. The skeleton contains appropriate metadata functions and basic init() and execute() functions. The component programmer adds logical code for the purposes of the agent, but once generated, the agent acts as a full copy business agent, ready for compiling and testing.

Once generated and compiled, the agent must be deployed to the system using the configuration tools of the console. This is handled by the iIT wizard where possible.

You can then test your new business agent by passing a document to it. It is recommended that you deploy and test the generated agent skeleton before you make any coding changes, and then run incremental tests during the development cycle. A good way to pass a test document to the business agent is using a file listener. Copy a test document into the listener input directory and retrieve the result from the destination directory.

A few metadata methods are different between agents, preparsers, and preemitters based on their purpose, but generally the programming is very similar.

Programming Exit Components for iWay Service Manager

iWay Service Manager must be able to run on any platform. Normally, this is a simple Java concern. However, some platforms such as IBM Open Edition under MVS, run in a different code page (1049) than the standard ASCII code page. If an exit or business agent must read or write a byte stream, it must also use the appropriate encoding version of the methods used. This issue applies to all exits and business agents, but is especially true in encryption exits, which must deal with raw bytes unassociated with code pages.

An instance of the exit is created as needed, and cached. A new instance is not created unless there is no available instance in the cache. You cannot predict which instance will be assigned to your channel at any time. Therefore, your exit must be serially reusable. Special register facilities exist for passing information between executions, but these facilities should be avoided if possible.

The Init() and term() methods in all exits run synchronized. However, the execute() and transform() methods can run in multiple threads. They must be reentrant as a general rule. Any global data can be initialized only in the init() routine, and unless your code locks the variables, it cannot be changed in the execute() or transform() routines at the risk of memory read failures. Such failures can be difficult to debug, but are usually evidenced by the reentrant method receiving unexpected data from the global area as other threads change the values. Do not depend upon the init() routine being called only once. The engine may elect to roll out your exit during periods of peak use and reinstate it later.

For more information on the standards that apply when coding any iWay Service Manager exit, see *Programming Standards, General Rules of the Road* on page 97.

Common Methods

All exits types provide a common set of control and metadata interrogation methods. Normally, these methods are placed into your exit code when the iIT or other wizard prepares your skeleton Java source code. Usually, you only modify the init() and execute() methods. The others are described here for completeness.

All exits descend from XDExitBase, providing them with common services such as parameter analysis, tracing, and so on. XDExitBase also provides *empty* versions of all required exit methods such as term(), eliminating the need for every exit to provide an implementation if the method is not needed for that exit().

Because all exits provide these methods, they are not further discussed in the sections dealing with individual exits.

For more information, see *Using the Eclipse-based iWay Component Generator Wizard*.

Configuration Parameters

Parameters are defined as properties in the XDPropertyInfo() method calls. Each property is a [static] class giving the internal property name, screen prompt, detailed description, type, and so on. The properties are grouped into PropertyGroup[] which are then passed to the server through an ExitInfo() definition in the constructor of the exit. These methods are described in the Javadoc.

The following syntax is an example of a single parameter:

```
public void validateExitParms(Map<String,String> parms, IXLogger logger)
```

A functioning exit will validate input parameters during design time. At that time, parameters are passed to a validateExitParms() method as a Map<String,String>, but iFL cannot be evaluated. You can, however, validate iFL syntax if it is used. The iFL compiler provides methods to check for syntax issues without actually compiling the function, as shown in the following syntax:

```
public void validateExitParms(Map<String,String> parms, IXLogger logger)
throws ConfigurationException
{
    String fileName = (String) parms.get(filenameParm.getName());
    if(!isPresent(fileName))
    {
        throw new ConfigurationException("File name is required");
    }
    int r = FunctionContainer.isValidExpression(fileName);
    if(r == FunctionContainer.SYNTAXERR)
    {
        throw new ConfigurationException("Invalid iFL in
"+filenameParm.getLongName());
    }
    else if(r == FunctionContainer.CONSTANT)
    {
        // other validation here?
    }
}
```

Other Metadata Methods

In addition to runtime parameters, exits support a metadata interface to permit their setup using the console.

The Wizard tools provided to construct exit frameworks automatically generate these metadata methods to describe the component. Usually, there is no reason to modify this code. The descriptions of the most common metadata methods are provided in the following list.

- **public String getDesc()**

  Returns a short string describing the purpose of the exit. For example, the description of XDCopyAgent is: *Copies document from in to out*. The description appears as help text when the exit is selected during configuration.

- **public String getLabel()**

  The label is a short name that appears in the exit selection list. For example, *copy documents*.

- **public String getCategories()**

  Exits are sorted in category groups to assist process flow developers in finding the correct exit. Any exit can be in one or more categories. Exits that do not implement this method are placed in the misc category.

  An exit is placed into multiple categories by returning a comma-delimited list, for example, *queue,emit* for the MQ emit agent. During the compile phase, a set of categories are available to the programmer, each starting with the prefix GROUP_. Your exit should always use an available group as opposed to creating your own, although there is no real constraint on the string you return. For example, in Java:

  ```
  return GROUP_QUEUE+GROUP_EMIT;
  ```

- **public String getOPEdges()**

  The execute() method returns the list of edges to be followed in a process flow. Implement this method to return the list of possible edges that the execute() method can return. iWay Designer uses this information to manage the edges that can be described in the flow.

  Return null if you cannot predict the set of values returned from execute().

  The default method (if you do not implement your own) returns an array signifying that *success* is the only possible result. By default, *error* is returned when your execute() method throws an exception, and is always permitted by the Designer program. If execute() returns an unpredicted result (one for which there is no edge defined), the process flow attempts to go down the default edge. If no such edge is defined, error is used. If no error is defined on the node, the flow throws an exception.

```
String[] getOPEdges()
{
String[] rts = {"asset","liability"};
return rts;
}
```

A set of mnemonics are defined for common edge names. You should always use a common name when returning an edge name if possible. The mnemonics begin with EX_. Some common examples are shown in the following table. The Javadoc lists the available common mnemonics.

Mnemonic	Description
EX_CANCELLED	The exit was cancelled by an external request.
EX_DUPLICATE	Attempting to perform an operation detected a duplicate. This may not be a failure.
EX_FAIL_CONNECT	The exit cannot connect to an external resource.
EX_FAIL_DUPLICATE	Attempting to perform an operation detected a duplicate. This is a failure.
EX_FAIL_NOTFOUND	A resource was not found.
EX_FAIL_OPERATION	The exit could not complete due to some non-specific cause.
EX_FAIL_PARSE	The iFL cannot be parsed - syntax error.
EX_FAIL_PARTNER	The exit received a failure from a [connection] partner.
EX_FAIL_SECURITY	The exit could not proceed due to a security violation.
EX_FAIL_UNREACHABLE	The exit attempted to reach an external host. The host was identifiable but not reachable. It is usually a network problem.
EX_NOTFOUND	A resource was not found, but this is not a failure. An example might be to handle a situation in which a key is not in a database.

Mnemonic	Description
EX_SUCCESS	Some resource was not found, but this is not a failure. An example might be to handle a situation in which a key is not in a data base.

- **public boolean getCanCancel()**

  The server provides for two methods of signaling a desire to cancel:

  - The cancel() method exposed by an agent will be called by the Process Flow Interpreter. This will signal that the agent should cancel its current operation. The agent can elect to return XDAgent.EX_CANCELLED to cause the flow to pass down the cancelled edge, or to take other action to terminate the agent.
  - Test the Java-standard interrupt isInterrupted() method of the system on the current thread. The server supports the interruption framework, allowing software that does not or cannot accept the cancel() method to be interrupted. For example, a current wait will be terminated with an InterruptedException. The application code should test the interrupted flag on the current thread and if it is set, to terminate the operation gracefully.

The simplest method to test this is to use the following Java call:

```
Thread.currentThread()isInterrupted()
```

You can then escape the current adapter if it is *true*. Do not reset the interrupted flag, as other threads may need to interrogate it. Proper use of this facility is discussed in relevant Java documentation.

If the exit implements the cancel() method, it returns *true*, else returns *false*. This method will be called immediately prior to the server calling cancel(). There may be phases of your exit that can be safely cancelled, and others that cannot. You can control whether the cancel() will be called by returning the appropriate result. The default is false.

The thread isInterrupted() call is not masked, but the exit can elect to test it at appropriate times.

The Executable Portion of an Exit

Business agents are written in standard Java language, and can use any available Java libraries or services.

public void init(String[ ] parms) throws XDException

Parameters are passed to the exit as name:=value pairs in a Map<String,String>. The init() method prepares the map. As it is initially prepared, any iFL (for example, _sreg, _property, and so on) in the value has not yet been evaluated. Doing so can be performed at the appropriate points in the execution of the exit. The wizards insert standard iFL evaluation code at the appropriate points.

Once prepared, parameters are available by name in the Map. The Exit Wizards automatically provide an init() method that handles the parameters in this way. The code in the generated init() method calls upon services of the XDExitBase class from which all exits descend to create the Map. The parms parameter of the method is deprecated and should not be used.

In an init() method, the application should accept and save configuration parameters that relate to the component instance itself, and not to the document passing through it. Because there is no document when the exit is loaded, the iFL evaluation cannot take a value from a document such as by using an _xpath() function. The iFL evaluator will throw an exception should a document be required by a specific configuration of the user.

Programmers generally cannot depend on the order in which parameters (especially user parameters) are stored. Consequently, one parameter should not depend upon the value of another parameter. Special code can be used if such is required, but this is not part of a normal exit.

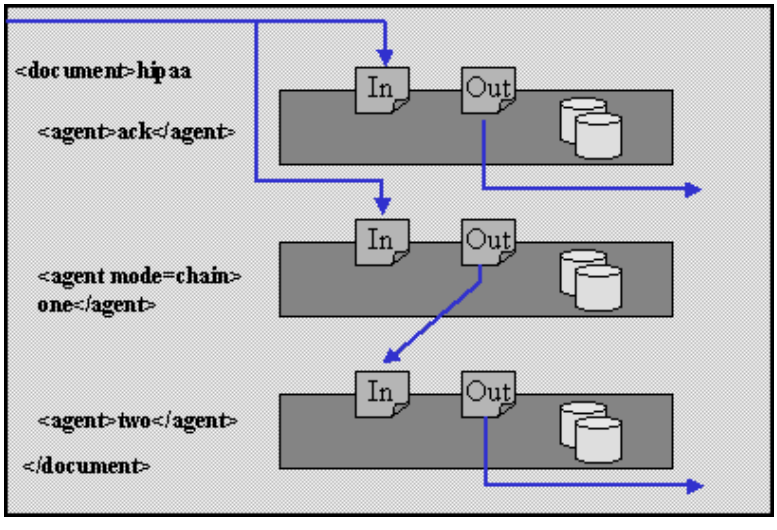
Exits may be loaded multiple times as they are paged in and out. Developers should not depend upon the init() method being called once. The init() method will, however, be called in a synchronized block so that initialization of any specific exit instance will not conflict with another.

public void term() throws XDException

This method is called as the engine closes down all uses of the exit. It is always called as the engine stops, and may be called at other times if the engine elects to page out the exit. If term() is called in such a case, the init() method is called before the exit is used again.

public String execute(XDDocument indoc, XDDocument outdoc)

The message handling portion of the exit is performed in the execute() method. The exit accepts its input in the indoc and is responsible for loading the result in the outdoc. The outdoc of your exit will be the input to the next exit, regardless of the type or where the exit runs in the channel.



The exit reports the result of its operation by returning a string. The value in the string instructs the process flow interpreter as to which edge is to be followed to the next exit. A successful exit should always return EX_SUCCESS (*success*) which follows the success edge. Standard EX_ mnemonics represent common causes of failure, and should always be used where applicable. Failures should begin with the characters *fail_* (for example, *fail_connect*), as the interpreter will follow a general wired fail edge if a specific failure edge cannot be located.

Note that the results of an exit that does not return EX_SUCCESS need not represent an error, but can instead simply denote a processing path. For example, an agent that checks for the presence of a record in a data base might return *notfound* to route the flow down a specific path. If a record is expected to exist and it is not found, this may be a failure, and the exit should return fail_notfound.

Although only agent exits (services in process flows) use wired edge names at this time, your exit should follow the rules of returning results, as future versions of iSM will utilize these returns.

The return statement can also return edge names other than those in the metadata list. The SQL Service object uses this capability to return XOPEN SQL codes. In some cases, the SQL result might not be interpretable by the SQL Service. For example, an SQL GRANT command might result in an XOPEN code of 01007, meaning *privilege not granted*. The SQL Service returns the following syntax:

```
return "01007,fail_operation"
```

This instructs the flow to follow the failure edge or the specific edge if known. The designer allows relations (edges) to be wired for the specific codes returned by getOPEdges as well as user codes, in this case, 01007. Naturally, the process flow designer needs to be aware of the possible (situational) returns and design accordingly.

Note: When making returns with lists of edges, always return the list in the order of the most specific to the least specific. For example, in an http emitter:

```
return("fail_501,fail_5xx,fail_server");
```

The flow continues to look for a wired edge, so if your node is wired to take fail_501 as specific and fail_5xx for the general case of all other 500-level errors, then you want the 501 edge to be taken and not the 5xx.

The parent object from which your exit descends, provides a set of standard edge names to make process services appear more consistent. It is recommended to use standard edge names for all agents to simplify process designs and documentation. Note that these edge names may not always denote a failure. For example, a service to check if a key is already in a database should return duplicate, which may be a perfectly reasonable and desired result. In other cases, such as an SQL update, this might be an error.

Edge Name	Description
closed	Resource being tested is closed.
duplicate	An operation denoted a duplicate situation.
fail_connect	Unable to connect to a resource.
fail_operation	Some other activity failed, not covered by another return.
fail_parse	Parsing operation has failed.
fail_partner	External resource issued a failure. For example, an SAP system signaled a failure to update a database.

Edge Name	Description
fail_security	Security failure. Possibly an encryption or decryption operation failed or a password was incorrect.
fail_timeout	Operation timed out.
notfound	An operation denoted that a resource was not found, such as a key in a database.
open	Resource being tested is open.
reissued	Resource that had been previously handled has been provided again. For example, a token extracted from a security system.
replace	Resource was replaced in a database or other collection.
success	Service completed successfully.

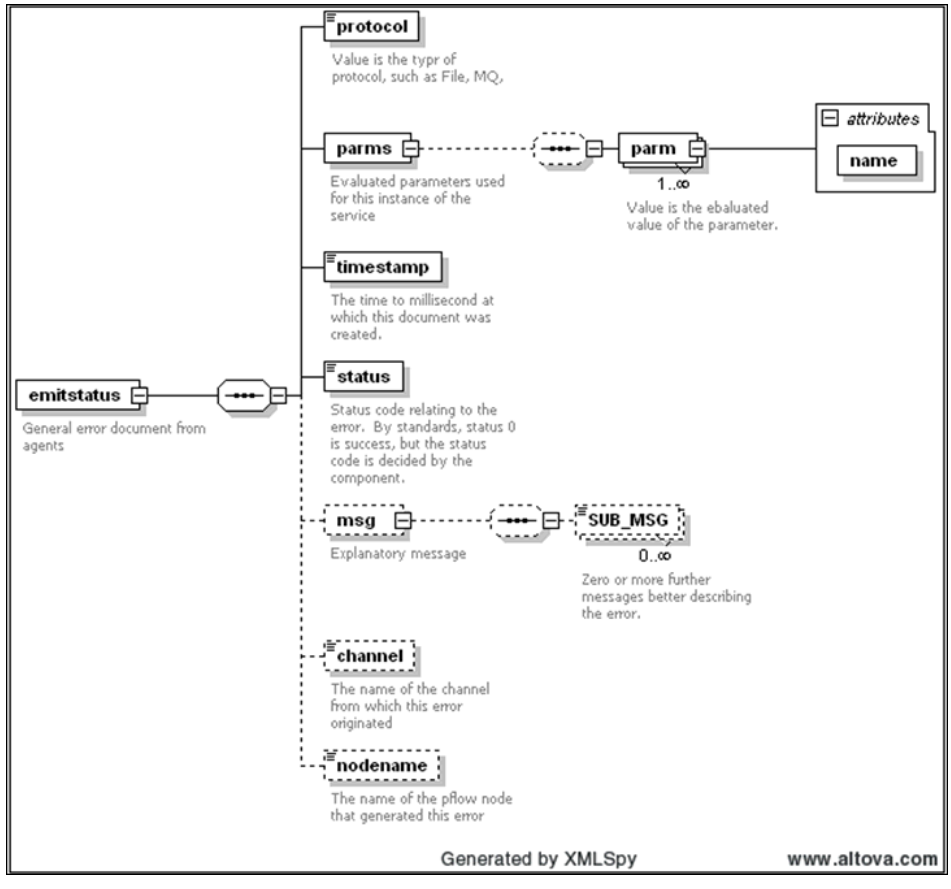
There is a distinction between errors and failures. The OnError edge is used when either:

- An exception has been thrown by the service agent.
- No returned code matches a wired edge.

This behavior makes the error edge the last resort. An exit cannot return errors, and signals problems it finds with failures. If the failures are not wired as described above, then the process flow will follow the error edge.

Some agents choose to emit a status document to describe their operation. A standard status document is constructed by the MakeStdStatusTree, which returns an XDNode of a tree that can be stored as the root of the output document of the exit. The exit can populate the tree accordingly, provided that downstream components can work with the changed document.

The following image shows the standard call.



The following syntax is an XML schema that represents the document from the standard call.

```
<?xml version="1.0" encoding="UTF-8" ?>
-    <!--    edited with XMLSpy v2009 (http://www.altova.com) by Richard
Beck (Information Builders) -->
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="emitstatus">
    <xs:annotation>
<xs:documentation>General error document from agents that emit a status
document</xs:documentation>
</xs:annotation>
    <xs:complexType>
    <xs:sequence>
    <xs:element name="protocol">
    <xs:annotation>
<xs:documentation>Value is the type of protocol, such as File, MQ,
etc.</xs:documentation>
</xs:annotation>
</xs:element>
    <xs:element name="parms">
    <xs:annotation>
<xs:documentation>Evaluated parameters used for this instance of the
service</xs:documentation>
</xs:annotation>
    <xs:complexType>
    <xs:sequence minOccurs="0">
    <xs:element name="parm" maxOccurs="unbounded">
    <xs:annotation>
<xs:documentation>Value is the evaluated value of the
parameter.</xs:documentation>
</xs:annotation>
    <xs:complexType>
<xs:attribute name="name" use="required" />
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
    <xs:element name="timestamp">
    <xs:annotation>
<xs:documentation>The time to millisecond at which this document was
created.</xs:documentation>
</xs:annotation>
</xs:element>
    <xs:element name="status" type="xs:integer">
    <xs:annotation>
```

```
<xs:documentation>Status code relating to the error. By standards, status
0 is success, but the status code is decided by the
component.</xs:documentation>
</xs:annotation>
</xs:element>
    <xs:element name="msg" minOccurs="0">
    <xs:annotation>
<xs:documentation>Explanatory message</xs:documentation>
</xs:annotation>
    <xs:complexType>
    <xs:sequence minOccurs="0">
    <xs:element name="SUB_MSG" minOccurs="0" maxOccurs="unbounded">
    <xs:annotation>
<xs:documentation>Zero or more further messages better describing the
error.</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
    <xs:element name="channel" minOccurs="0">
    <xs:annotation>
<xs:documentation>The name of the channel from which this error
originated</xs:documentation>
</xs:annotation>
</xs:element>
    <xs:element name="nodename" minOccurs="0">
    <xs:annotation>
<xs:documentation>The name of the pflow node that generated this
 error</xs:documentation>
</xs:annotation>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

Dealing With iFL In the Execute Method

Once the init() method has loaded the parameter map (called parmMap as a standard), the execute method needs to retrieve actual values of the parameters during the execute() method. This occurs because the parameter value may depend upon information in the incoming XDDocument. For example, if the applicatoion user configured a filename to be something such as _xpath(/root/input/filename), then the value will vary per message.

Standard code in the execute() can evaluate the iFL each time that the method is entered.

```
Map<String, String> tmap = (Map<String, String>) parmMap.clone();
       try
    {
        XDUtil.evaluateWithException(tmap, docIn, getSRM(), logger);
    }
    catch (Exception e)
    {
XDDocument errorDoc = new XDErrorDocument(worker,
docIn,
XDErrorDocument.PIPE_AGENT,  getNodeName(), null, e);
        errorDoc.moveTo(docOut);
        return EX_FAIL_PARSE;  // set for the proper edge
    }
```

The parmMap is cloned so that the original map with the iFL values is not changed. Instead, the temporary map serves as the source of the values to the method.

```
String filename = FileNameParm.getString(tMap);
```

The above syntax returns the value. The methods for dealing with values for the actual XDPropertyInfo describing the file name are available in the Javadoc for iSM.

public void cancel()

Called on an external thread, this instructs the exit to attempt to cancel its operation. Functioning exits try to implement cancel(). For example, the provided SQL agent attempts to send the cancel to the data base, and if it is in retrieval phase (reading messages from the data base), it will close the result set and return on the EX_CANCELLED edge.

public void onError(XDNode rulenode, XDNode errornode, int thePosition, int thePiece, String theSegID) throws XDException

This method exists only in acknowledgment business agents. It is called by the rule engine to post errors that can be handled when the actual execute() method is called later. The input consists of the rule node (the node of the rule that detected the error, the node on which the error was detected, the child of the node in error, the offset component into that child, and the identifier of the EDI segment in error). Some of these fields may not be meaningful in non-EDI situations.

Acknowledgment business agents are expected to mark the document tree with the error, using the getAssociatedVector() and setAssociatedVector() calls to post the error objects to the appropriate node. Later, in the execute() method, a tree walk can locate the errors for construction of the acknowledgment.

Acknowledgment business agents can be scheduled for input rules violations and for output document rules violations. The execute() method for the input case generates an output document that is immediately emitted. The business agent scheduled for the output case receives the document and can modify it, but is not expected to emit its own document. It can redirect the output by using set/addReplyTo() and set/addErrorTo() calls to change destinations. It can also set special registers that can be used in routing logic later in the cycle, such as in selection of a reviewer or preemitter.

Although the exit is not restricted in the edge names that it uses, it is a best practice to use standard edge names. A list of these can be found under the execute() method description, and iSM offers standard mnemonics for common edge names.

Example: Source Code for the XDMoveAgent

For simplicity, metadata calls are not shown.

```
import com.ibi.edaqm.*;
/**
*    This is the plug-in agent used to move the document from input to
output.
*    This is the simplest possible agent. It can be called to service any
 document type.
*
*    This class extends the XDAgent class, which provides several services,
 such as tracing.
*
*/public final class XDMoveAgent extends XDAgent
/**
*    Agents have a null constructor. It must be public as it is reached from
 the edaqm package.
*/
public XDMoveAgent()
{
            super();
}
/**
*    Execution of the agent begins at the execute method. The input
document and
*    the output document are passed in. This particular execute() method
simply
*    copies the message tree from the input document to the output document.
*    This agent is, in effect, a NOP. Upon completion of this agent, the
output
 document
*    is passed to the next stage.
*
*    @param docIn Input document.
*    @param docOut Output document to create.
*    @exception XDException (unused in this implementation)
*/
public String execute(XDDocument docIn, XDDocument docOut) throws
XDException
{
docIn.moveTo(docOut); //service method in XDExitBase return "success";
}
}
```

Partial Flow Agent Composition

In some use cases or scenarios, you may require two or more process flow services (agents or nodes) to appear as a single node. Once defined, this can simplify process flow development when the composed unit is frequently used. Improved process flow performance is also a benefit. An example for such a composed component might be a single component that:

- Accepts a user ID and password, resetting the security principal (often called impersonation).
- Performs some action at the new security level.

- Resets the security level to the original level.
- Appears as one service node on the palette.

Composition incorporates existing services, generating configuration groups in iWay Integration Tools (iIT) for each incorporated service after discarding duplicates. You must compile the composed service in Java just as you would your own agent. The included services will not be brought into your new service, but will reside in the existing *iwcore* or other .jar file on the classpath.

As each included service executes, it passes the current document to the next incorporated service (or out of the composed service) in the *Success* edge. If an incorporated service returns an edge code other than *Success*, then the service terminates on that edge. Since the included services operate on a *stack* of included agents, this is referred to as a Stack Agent. The example that is described in this section will stack two agents (simplifying a common task of formatting a document for mailing and then sending it). The use of the tree map causes the agents to execute in the defined order. The properties of the tree map are described in the following table:

Key	Value
Class name of the agent	Prefix for the agent's configuration groups to differentiate the combined configuration parameters.

The remaining required methods give your composite agent a label, description, and category.

There is no requirement for the composed Stack Agent to only include iWay agents. The agent must be able to execute independently, even if its purpose is only to be part of a Stack Agent.

```
public final class FormatAndMail extends XDStackAgentBase
{
  private static Map<String, String> stackMap = new TreeMap<>(); //
  class,GroupnamePrefix
  static
  {
      stackMap.put("com.ibi.agents.XDTransformAgent", "TransformIn");
      stackMap.put("com.ibi.agents.XDEmailEmitAgent",   "Mail");
  }
  public StackTestAgent2()
  {
      ExitInfo ei = makeExitInfo(stackMap);
      setExitInfo(ei);
  }
// Because we inherit from XDStackAgentBase, we must override visibility
in order to show up
  public int getVisibility()
  {
      return VISIBILITY_VISIBLE;
  }
  public String getCategories()   // which categories should the new
stacked
  agent appear in?
  {
      return GROUP_MAIL;
  }
  public String getDesc()
  {
      return "Email Error Message According to Company Standards.";
  }
  public String getLabel()
  {
      return "Send Error";
  }
}
```

Status Documents

This section lists and describes different splitter exits used in iWay Service Manager (iSM).

Splitter Aware Exits

Splitters are preparsers that break up large input messages into smaller messages that are processed piece by piece. Each piece flows through the work flow independently, as if it were a complete document. At the end of the stream, a special End of Stream (EOS) message is sent, enabling the subsequent exits in the flow to take any end-of-stream actions such as emitting aggregation documents, issuing commits, and so on.

Generally, exits downstream from a splitting preparser need not be concerned with the splitting of the message or with the end of stream signal. The exit processes the split portion exactly as it would a complete message. The exit can expose the `boolean wantEOS()` method to signal to the engine that the exit is to be called on EOS. False is the default value, so that if this method is not implemented by the exit, the exit is not invoked on EOS. To receive EOS signals, implement:

```
boolean wantEOS()
{
  return true;
}
```

Note: The Development Wizards set this as required.

In an execution method, such as a process of execute method in a business agent, the method can call:

```
int getStreamState()
```

to determine the state of the stream. Possible return values are:

Return Value	Description
STREAM_NONE	Exit is not operating in a streaming/splitting mode. This is the normal mode of a workflow.
STREAM_ACTIVE	Exit was invoked for a split portion of a streamed document.
STREAM_EOS	Exit was invoked to handle the EOS signal. No other data is available. In addition, special register EOS is defined with the value 1.

Splitting is performed by preparsers. A splitting preparser must be the first user-configured preparser. Pre-built splitting preparsers are provided for XML, non-XML, and EDI messages.

Example: Stream Handling Agent

This example shows two methods from a stream handling agent and illustrates the handling of EOS and non-EOS tasks. The agent emits an accumulated document when all parts have arrived, but avoids any output at non-EOS. The execute method adds each arriving part as a child of a root, and at EOS emits the new tree based on root.

```
List reps;     // place to save replyTo list
List errs;     // place to save errorTo list
XDNode accumroot=null;  // build accumulation document here
public String execute(XDDocument docIn, XDDocument docOut)
       throws XDException
{
   int streamState = getStreamState();  // ask what state we are in
   XDNode passRoot;
   if (streamState != XDAgent.STREAM_EOS)
   {
       if (accumroot == null)  // is this new document (new stream)?
       {
           accumroot=new XDNode("root"); // make root for accum document
           reps = docIn.getReplyTo();  // save the replyTo list
           errs = docIn.getErrorTo();  // save the errorTo list
       }
       passRoot = XDNode.cloneTree(docIn.getRoot()); // add to accum
       docOut.setRoot(null);                        // we want NO doc out
       docOut.setReplyTo(null);                // clean out any replyto's
       docOut.addReplyTo(XD.PROTOCOL_NULL, null);    // avoid ANY emit
       accumroot.setLastChild(passRoot);          // add to accumulation
   }
   else   // this is EOS, so emit accumulated document
   {
       docOut.setRoot(accumroot);    // set last copy to output
       docOut.setReplyTo(reps);      // restore replyTo
       docOut.setErrorTo(errs);      // restore errorTo
       accumroot=null;               // for next pass
   }
   return "success";
}
 protected boolean wantEOS()  // added by development wizard
 {
     return true; // tell engine we want to receive the EOS signal
 }
```

Business Agents Running in Transactions

The engine manages transactions, invoking code registered by the business agent when the transaction edge is reached. To join a transaction, the business agent must register a class meeting the XDTx interface, supporting the prepare(), commit(), and rollback() methods. Dealing with the transaction completion is the responsibility of the business agent.

Transactions can be local or global. A local transaction is a business agent group process flow running for a single message. If the local transaction state has been set (a flag in the listener) a worker-specific transaction is begun. The transaction ends when a business agent throws an exception or the business agent group ends successfully.

A global transaction is one begun on the XDMTLclMaster() by an outside caller. The beginTx() call associates the local master with the transaction. The outside caller later issues the commit(), rollback(), or prepare() methods as appropriate to each instance of the XDMTLclMaster in the transaction. Establishing global transactions is performed by the server, and cannot be directly configured within the server.

The business agent calls the getTID() method to obtain a string of the transaction identifier assigned by the engine. If the getTID() call returns null, there is no transaction available for this business agent to join.

If a transaction is available, the business agent uses the storeTx() method to register its transaction handler. This is the equivalent of a transaction join.

```
Object storeTx(String tid, Object o);
```

The storeTx() method returns the transaction handler class to be used by the transaction manager. Usually, this is the class that you attempted to store. If the transaction manager finds a duplicate class, it merges the transaction classes and returns the class of the appropriate type. In determining the class duplication, it uses the object hashCode method. We recommend that writers provide their own hashCode method to control this duplication. You must work with the returned class reference rather than the reference to your constructed class. Once you have the reference, you can set values into the transaction handler class. Your code must take into account that the classes may be merged.

If your class can handle XA two-phase commit, you should provide a full implementation of the prepare() method. Otherwise, return XDTx.SUCCESS. You report whether you can handle two-phase commit with the canPrepare() method return.

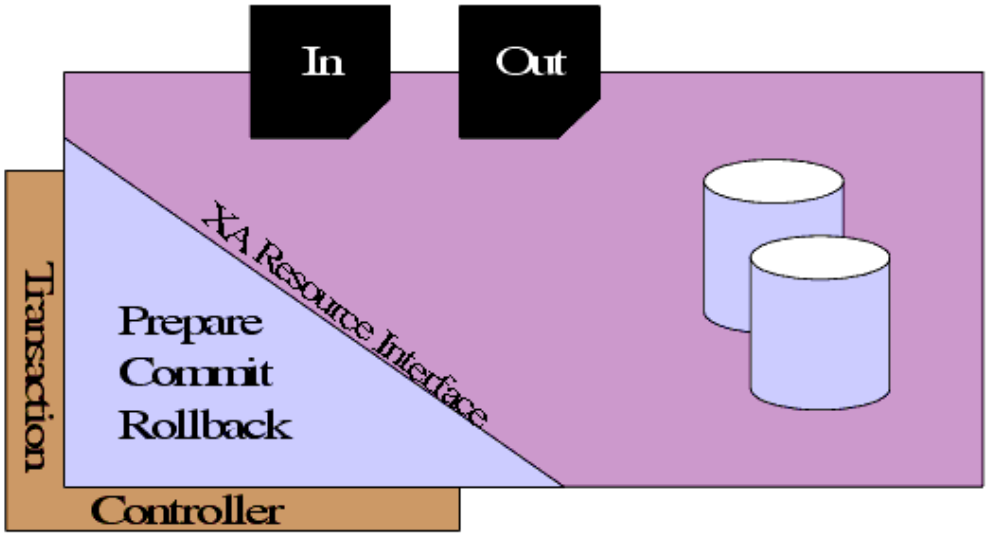
Your business agent delegates to the transaction handler the actual execution of the commit and rollback code, which is invoked after your business agent has completed its process() method.

Methods in the XDTx Interface Class

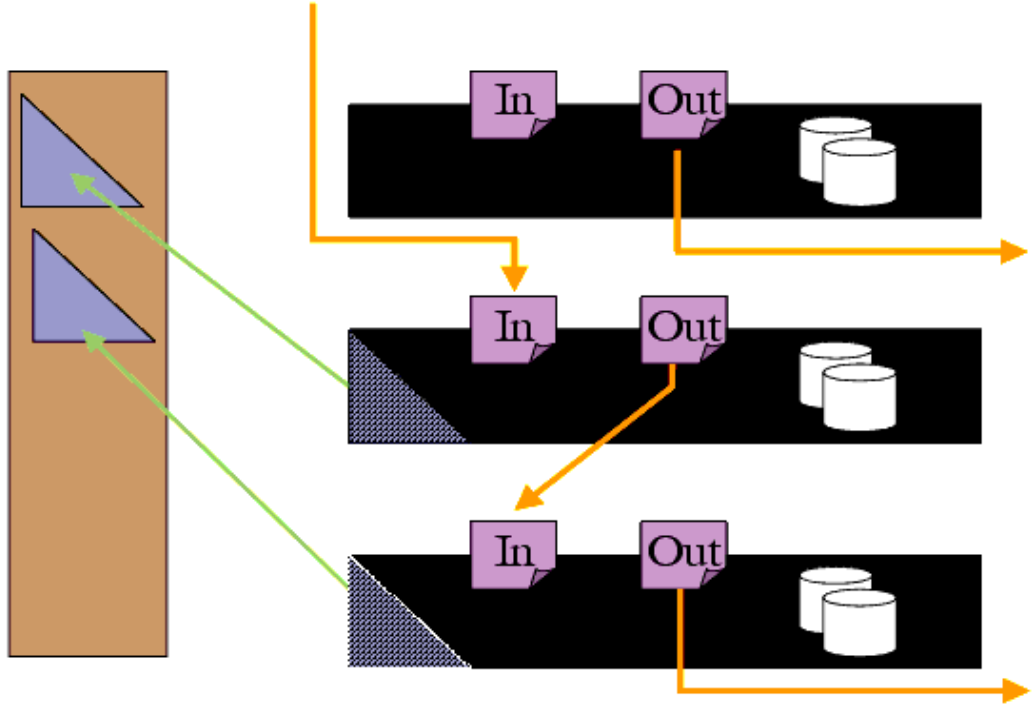
Method	Returns	Description
canPrepare	boolean	Specifies whether the interface can support two-phase commit.
commit	int	Commits the transaction.
prepare	int	Prepares to commit.
rollback	int	Rolls back the transaction.

The business agent code is prepared normally. A section of the code is isolated into an object which is used by the business agent to house references and code for the transaction completion activity.

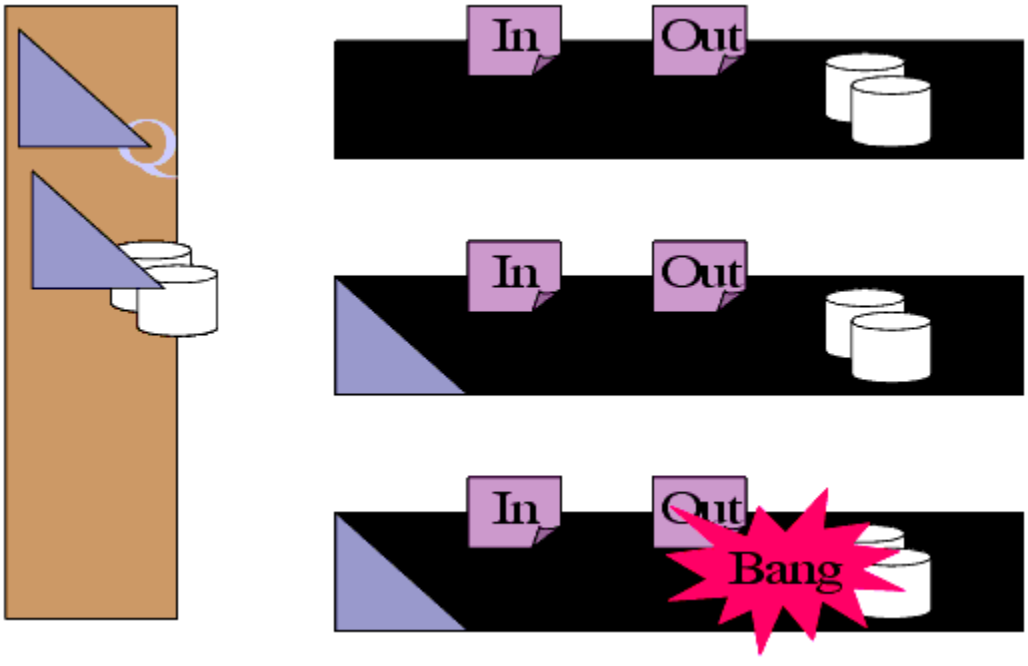
<agent>method</agent>



As the business agent operates, it passes a reference to its transaction object back to the server. It uses the storeTX() method to do this. On completion of the business agent stack or process flow, the server's transaction system holds references to the stored transaction objects.



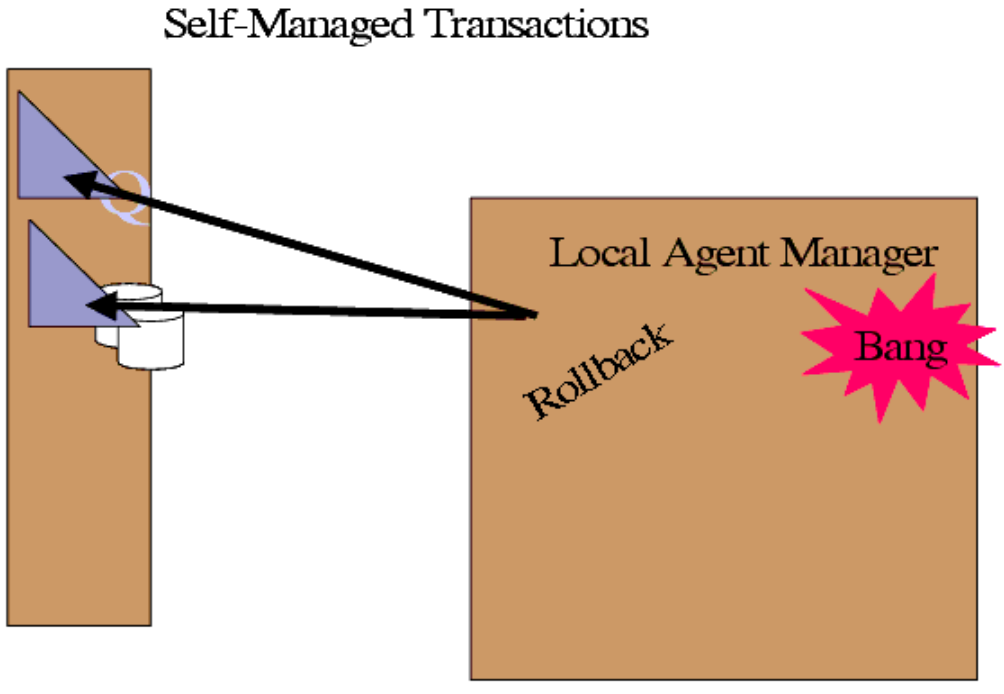
Assume that one of the business agents encounters an unrecoverable condition. This might be a program bug that is trapped by the server, or it might be a logical condition that causes the business agent to throw an exception.



The local transaction manager intercepts the exception and steps though the list of stored transaction objects, sending a rollback() call to each. It is the responsibility of the stored transaction object to affect the rollback(). If no exception is encountered, the local transaction manager instead takes the following actions:

1. For each transaction object that returns true from the canPrepare() method, a prepare() is issued.
2. If all prepare()s return successfully, the transaction manager issues commit() calls to the transaction object. Commit sends commit() to any outstanding emitters.

3. If any prepare() call returns that the prepare was unsuccessful, the transaction manager sends rollback() calls to all of the stored transaction objects. It also sends rollbacks to any outstanding (uncommitted) emitters.



Sample Transaction Handler

This example is taken from the XDJDBCAgent, which registers an inner class to handle transactions if it finds a Transaction ID (TID). All JDBC commit and rollback functions are handled by this class. The JDBC agent does not close the connections until the transaction is complete. The handler closes all JDBC connections regardless of which instance of the JDBC agent opened the connection.

```
class TransControl implements XDTx
{
    ArrayList conList;
    int len;
    Connection con;
    TransControl()
    {
        conList = new ArrayList();
    }
    void addConnection(Connection c)  // add a connection to this transaction
    {
        if (!conList.contains(c))
        {
            conList.add(c);
        }
    }
    public int hashCode()  // all JDBC connections are on a single object
    {
        return "jdbc".hashCode();
    }
    /**
      *Commit the transaction
      *
      */
    public int commit()
    {
        int rc = XDTx.SUCCESS;
        len = conList.size();
        for (int i=0;i<len;i++)
        {
            con=(Connection)conList.get(i);
```

```
            if (con != null)
            {
                try
                {
                    if (!con.isClosed())  // make certain we are still active
                    {
                        con.commit();
                        con.close();
                    }
                }
                catch (SQLException e)
                {
                    rc = XDTx.FAIL;
                }
            }
        }
        return rc;
    }
    /**
      * Roll back the transaction
      *
      * @return SUCCESS, FAIL, or HEURISTIC
      */
    public int rollback()
    {
        int rc = XDTx.SUCCESS;
        len = conList.size();
        for (int i=0;i<len;i++)
        {
            con=(Connection)conList.get(i);
            if (con != null)
            {
                try
                {
                    if (!con.isClosed())
                    {
                        con.rollback();
                        con.close();
                    }
                }
                catch (SQLException e)
                {
                    rc = XDTx.FAIL;
                }
            }
        }
        return rc;
    }
    /**
      * Business agent does not support 2 phase commit, so always return SUCCESS
      */
    public int prepare()
    {
        return XDTx.SUCCESS;
    }
   public boolean canPrepare(){return false;}
}
```

Calling Code for Registering With the Transaction Handler

```
String tid = getTID();  // get the transaction identifier TransControl
tctl=null;
if (tid != null)  // if in a transaction, set up for control
{
    trace(XDState.DEBUG,"Agent in transaction "+tid);
    tctl = (TransControl) storeTx(tid,new TransControl());
}
```

Notice that the transaction handler returns the transaction object, which may be the one you are attempting to register, or it may be the one already registered. You should always use the object returned from the storeTx() method. The scope of the object is the transaction in progress.

Your program may need to check at various points whether a transaction is in use. For example, if no transaction is in use, you close the connections and commit the work. If a transaction is in use, leave this to the transaction controller to call your registered transaction control object to do the commit for you.

Preparsers

A preparser is a Java class designed to convert an incoming non-XML source into an XML document. This preparsed document then passes through the standard transform services to reach the designated processing business agent. Preparsers are not invoked when the *Accepts non-XML (flat) only* property in listeners is set to true.

The engine calls the preparser with the appropriate data form, if possible. Otherwise, an XDException is thrown. Stream and byte[] format can only be specified for the first preparser in a sequence.

The preparser is called on the worker thread, and one instance exists for each worker operating under the listener.

The preparser must support the appropriate transformations.

Preparsers execute for all documents (flat and XML) as well as streaming listeners. The XDDocument object passed to your exit contains the input and APIs are available to determine the type. For example if the input type in reports true to isBytes(), then you would convert the input from bytes to the desired output format.

On completion of the stream, the preparser is responsible for emitting a "default" dummy document to satisfy the preparser specification (do not emit null) and to call

```
setStreamProcessed(true);
```

to signal to the engine that work with this stream is complete.

Note: <BATCH>EOT</BATCH> is emitted as a dummy, which is the format generated automatically by the Preparser Wizard.

Stream preparsers interact with the transaction control system slightly differently than do single entity preparsers. All documents on the stream are treated as a single transaction, and commit is not called until the setStreamProcessed(true) is called by the preparser. The signal to business agents that the stream of related documents is complete is the call to commit() in the business agent XDTx class. For example, an acknowledgment business agent might need to accumulate the acknowledgment until an entire EDI document has been handled. Its commit() method or its rollback() method would be the signal to assemble the final acknowledgment (EDI 997) and emit it to the proper destination.

public String transform()

All preparsers have at least one transform() method as their main process method. You should return a string suitable for either the standard XML parser, or (if you are depending upon chained preparsers) a string suitable for input to the next preparser. For example:

```
public String transform(byte[] b) throws Exception
public XDDocument transform(String s) throws Exception
public XDDocument transform(InputStream is) throws Exception
```

While String and byte[] formats operate on single entities (a complete input from a source) the stream format preparser is different. It receives a stream, which it can read as needed. If the stream is not complete (EOS) when the preparser emits a document, the preparser is called again upon completion of the processing for the emitted document. This provides the ability to split the input stream into several documents. For example, a file source might contain several documents, each of which is processed separately.

On completion of the stream, the preparser is responsible for emitting a "default" dummy document to satisfy the preparser specification (do not emit null) and to call

```
setStreamProcessed(true);
```

to signal to the engine that work with this stream is complete.

Note: <BATCH>EOT</BATCH> is emitted as a dummy, which is the format generated automatically by the Preparser Wizard.

Stream preparsers interact with the transaction control system slightly differently than do single entity preparsers. All documents on the stream are treated as a single transaction, and commit is not called until the setStreamProcessed(true) is called by the preparser. The signal to business agents that the stream of related documents is complete is the call to commit() in the business agent XDTx class. For example, an Acknowledgement business agent might need to accumulate the acknowledgement until an entire EDI document has been handled. Its commit() method or its rollback() method would be the signal to assemble the final acknowledgement (EDI 997) and emit it to the proper destination.

public int inputForm()

Returns whether the preparser expects:

Mnemonic	Input Format
XDPreParser.BOTH	Byte array and String are both accepted.
XDPreParser.BYTE	Byte array.
XDPreParser.STREAM	InputStream. Used by splitting preparsers only. Can apply only to the first preparser in a stack.
XDPreParser.STRING	String.

public int resultForm()

Returns whether the remainder of the message handling is to treat this document as flat or XML. A flat document is not parsed, but is sent to the business agents in the form returned from the preparser.

Mnemonic	Input Format
XDPreParser.XML_OUTPUT	Treats this message as an XML document to be parsed. This is the default value.
XDPreParser.FLAT_OUTPUT	Prevents parsing of the document and treats the message as flat (non-XML).

Mnemonic	Input Format
XDPreParser.DOCUMENT	Provides an alternate call to transform the message. The result of this call is an XDDocument; this is an XML tree. Preparsers that can automatically construct a tree can avoid the subsequent parse.

The resultForm() value is only checked for the last preparser to be executed. Values returned from preparsers earlier in the chain are ignored.

The return should always be "success" unless the preparser needs to report an error.

public XDDocument transformToDoc()

The following transformToDoc methods can only be called on the last preparser to execute in the preparser chain. These are analogous to the methods that return Strings, however, these return XDDocuments.

```
public XDDocument transformToDoc(byte[] b) throws Exception
public XDDocument transformToDoc(String s) throws Exception
public XDDocument transformToDoc(InputStream is) throws Exception
```

public String execute(XDDocument indoc, XDDocument outdoc) throws XDException

The execute() method is available to replace the transform() method. The preparser will need to examine the input document and set the output document appropriately for passing on through the channel.

The return should always be "success" unless the preparser throws an exception.

Writing a Splitting Preparser

Splitting preparsers enable a message to be divided into sections for processing. A splitting preparser must be the first configured preparser in the message flow. Such a preparser must implement the getInputForm() method, returning XDPreParser.STREAM.

The preparser reads the stream, dividing the message into portions for processing. At the end of stream, it emits an end of stream document <batch>EOT</batch> and signals to the worker that the end of stream has been reached. Your agent or reviewer can optionally elect to "see" the end of stream signal.

This example is taken from the flat document preparser. It uses a method isDelim() to determine whether the character read from the stream is a delimiter. Several delimiters in a row are permitted; for example, if the delimiter were \n, then several blank lines would be treated as a single delimiter.

```
StringBuffer sb = new StringBuffer(100);
BufferedInputStream bis = null;
boolean delimChar = false;
int lastChar = -1;
public String transform(InputStream is) throws Exception
{
    if (bis == null)   // initial state
    {
        bis = new BufferedInputStream(is);
    }
     String s;
    sb.setLength(0);      // clear out for next message
    int v = -1;
    for (;;)
    {
        if (!delimChar)   // need to read from stream
        {
            v = bis.read();
        }
        else              // filled lastChar trying to swallow delimiters
        {
            v = lastChar;
            delimChar = false;
        }
        if (v == -1)
        {
            break;
        }
```

```
            else
            {
                char c = (char)v;
                if (isDelim(c))     // delimiter?
                {
                    for (;;)        // swallow any other delimiters
                    {
                        v = bis.read();
                        char cc = (char)v;
                        if (!isDelim(cc))    // oops, read a valid char
                        {
                            lastChar = v;    // save it and signal
                            delimChar = true;
                            break;
                        }
                    }
                    break;
                }
                else       // append to output
                {
                    sb.append(c);
                }
            }
        }
        s = sb.toString();       // create output
        if (s.length() == 0)     // end of processing?
        {
            s = "<batch>EOT</batch>";           // eos signal
            worker.setStreamProcessed(true);    // mark end of stream reached
            bis = null;                         // reset
            delimChar = false;                  // reset
        }
        return s;
    }
```

Note the call to setStreamProcessed() on completion of the stream. This is required to instruct the flow to take end of stream actions on the EOT message.

Reviewers

An inbound reviewer is the first exit to receive the document after parse. An outbound reviewer is the last exit to receive the document prior to the actual emit operation. These exits are intended for envelope handling, but can be used for any desired purpose.

Reviewers are similar to business agents, in that they receive as input the incoming document, and are responsible for loading the outbound document with the appropriate information after the review. For example, an inbound reviewer might handle the WS-SECURITY header, operating on the payload of the document as directed by the fields in the header.

If more than one reviewer is defined for a message they are chained, such that the output of each reviewer is the input to the next. Several reviewers, one following the other, might each be responsible for handling one type of envelope header.

The reviewer receives control at its execute() method, which is as described for business agents. The method must fill in the outbound document, and should return "success" or throw an exception.

This example handles a bit of the WS-ADDRESSING envelope header. The document looks like this:

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing">
    <S:Header>
       <wsa:ReplyTo>
         <wsa:Address>http://business456.com/client1</wsa:Address>
       </wsa:ReplyTo>
       <wsa:To>http://fabrikam123.com/Purchasing</wsa:To>
       <wsa:Action>AgentToDO?patm1=fred&amp;parm2=tom</wsa:Action>
    </S:Header>
    <S:Body>
       <some xml message that the agent or flow can handle>
    </S:Body>
</S:Envelope>
```

The code to locate the Action and push the agent for execution might look like this:

```
public String execute(XDDocument docIn, XDDocument docOut)
        throws XDException
{
   copyTree(docIn, docOut); // make sure we have output
   XDNode root = docOut.getRoot();
   Iterator iter = root.getAttributeNames();
   String addrPrefix="";
   while (iter.hasNext())  // find my namespace prefix
   {
       String attrName = (String)iter.next();
       String attVal = root.getAttribute(attrName);
       if (attVal.equalsIgnoreCase(
           "http://schemas.xmlsoap.org/ws/2004/03/addressing"))
       {
           int ax = attrName.indexOf(':');
           addrPrefix= attrName.substring(ax+1);
           addrPrefix+=":";
           break;
       }
   }
   XDNode saNode = root.findByName(addrPrefix+"Action");
   if (saNode == null)
   {
       debug("Did not locate action");
       return "success";
   }
   String agname = saNode.getValue();
   String parmStr = null;
   int qmix = agname.indexOf('?');
   if (qmix == -1)
   {
   }
   else
   {
       parmStr = agname.substring(qmix+1);
       agname = agname.substring(0,qmix);
   }
```

```
    XDDictionary dict = getDictionaryDocument();
    XDNode dictroot = dict.getRoot();
    XDNode sysNode = dictroot.findChild("system");
    XDNode defNode = sysNode.findChild("define");
    XDNode agNode = defNode.findChild("agent");
    XDNode exNode = agNode.findByValue(agname);
    if (exNode == null)   // not found
    {
        worker.pushAgentName(agname);
    }
    else  // found it in defined agents
    {
        XDNode loadNode = XDNode.cloneTree(exNode); // make a parm
        // now set values
        if (parmStr != null)
        {
            String key;
            String val;
            StringTokenizer pst = new StringTokenizer(parmStr,"&");
            while (pst.hasMoreTokens())
            {
                String tok = pst.nextToken();
                int ix = tok.indexOf('=');
                key = tok.substring(0,ix);
                val = tok.substring(ix+1);
                XDNode pn = loadNode.findByAttributeValue("name",key);
                if (pn == null)
                {
                    pn=new XDNode("parm");
                    pn.setAttribute("name",key);
                    loadNode.setLastChild(pn);
                }
                pn.setValue(val);
            }
        }
        worker.pushAgentName(agname, loadNode);
    }
    saNode.snipNode();  // remove this from the envelope
    return "success";
}
```

Preemitters

A preemitter is a Java class designed to convert from an outgoing XML document to a non-XML format. A standard preemitter is provided with the system capable of calling an iWay transform to accomplish this conversion. This "emit-ready" document then passes through the encryption services and is emitted based on the protocol. Preemitters can be associated with listeners, documents, or directly with XDDocuments within a business agent.

An example of a preemitter is a class that accepts an XML document and converts it to EDI.

Preemitters can be chained, so that the output of the first preemitter is passed to the second preemitter. This facility is useful in cases in which the outgoing document must be processed in its native format before final emit processing. The first preemitter accepts the document as an XDDocument (the internal form of a document) and must convert it to a byte stream. Subsequent preemitters accept byte streams and emit byte streams.

The preemitter may implement init() and term() calls, and must implement an input method.

public byte[ ] transform(XDDocument d) throws XDException

This is responsible for transforming the incoming document into an output byte stream. It is the first preemitter in the chain.

public byte[ ] transform(byte[ ] b) throws XDException

This is responsible for subsequent transforming of the outbound byte stream into another processed byte stream. This method is called for preemitters following the first preemitter, which uses the transform (XDDocument) signature to convert from document form to flat form for transmission.

Example, replace all 'X' characters with a new line.

```
public byte[] transform(byte[] b) throws XDException
{
    // for each character message, replace X with new line
    for (int j=0;j<b.length;j++)
    {
        if (b[i]=='X')
        {
            b[i]='\n';
        }
    }
    return b;
}
```

Building the Exit

To build an exit, you only need to include the iwcore.jar and iwutil.jar files in the compile classpath. There is no requirement for the exit to exist in any specific package, although it should not be in the default package. You should register any exits in your extension jar. The ilT wizard will generate a .jar file for the exit types it supports. You should place your exit .jar file in the etc/manager/extension directory. A register method is called based on the class denoted in the manifest of the .jar file.

Using the addExit(String *fullname*, String shortname) method, the exit is automatically defined for configuration. In this call, *fullname* is the full package name of the exit (for example, com.yourco.agents.myexit). The *shortname* is the alias by which the exit is accessed in the configuration.

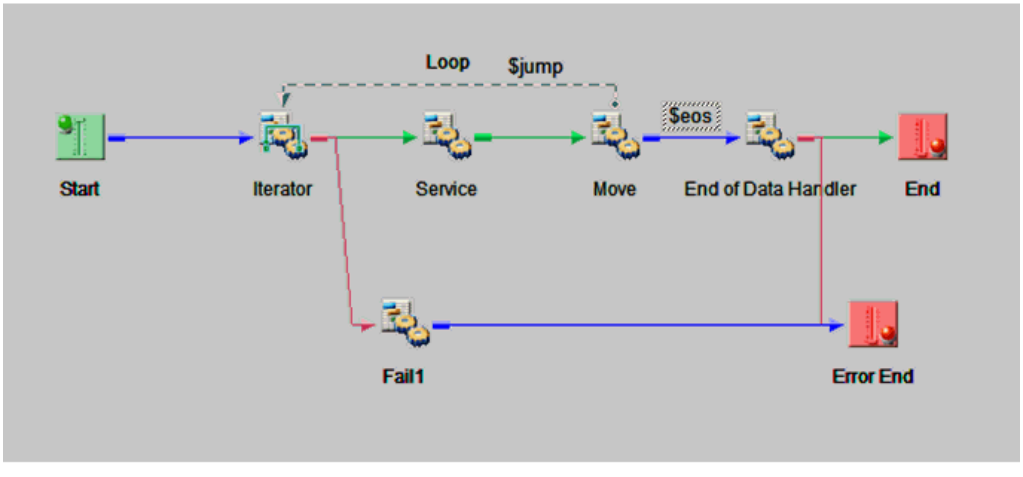
You can use the ANT build script included in *Standard Business Exits* on page 187 to build exits. As is the case with protocol elements (see *Installing Components* on page 100), if you place your exit .jar file in the etc/manager/extension directory with the literal name beginning with the letters “iw”, a register method is called.

Writing Iterators

Iterators are services (agents) that loop. Each iteration results in an XDDocument that is emitted to the next node in a process flow, with the exception of the final loop step, which passes control to the end of data node. An example of an iterator might be one that sends the incoming document into the loop a specified number of times (a count iterator). Other iterators split incoming payloads on line breaks or by using XPATH into the XML payload.

An iterator must declare itself to the Process Flow Interpreter, which is accomplished by including the isIterator() method in the service code (as shown in the sample iterator syntax at the end of this topic).

An iterator takes action depending on the edge on which it receives control. The following image shows a sample iterator in a process flow, which is used for demonstration purposes.



The iterator accepts documents on one of the following edges:

- The original incoming edge (as shown in the above image as arriving from the Start node).

- A special edge named `$jump`.

The iterator can test the incoming edge by using the following call:

```
public String getEdgeName()
```

This returns the name of the edge on which the agent received control. If the edge name is not $jump, then it is the initial entry to the iterator and the first output document must be created. If the edge name is $jump, then this is a subsequent entry and a subsequent document is to be created. In either case, the result should be returned in the output document as it would be for any normal agent.

The iterator should return a success result or a failure edge name depending upon the result of the iteration. Ideally, on original (non-$jump edge) it should attempt to determine whether the iteration can be completed, so as to avoid returning failures while it is looping. On the last entry, if there are no further data to be emitted, the iterator then returns a status or other document as configured, and returns the $eos edge. This causes the Process Flow Interpreter to pass control to the node following the origin of the $jump edges. In the example shown, this is the node named End of Data Handler.

When the process flow ends, it is returned to the system. At this time, the following method is called:

```
public void reset()
```

Your iterator should ensure that all resources used in the iteration are reset. For example, if your iterator is using a database, it releases the resources in this call if they have not been released earlier. Be aware that an error in the loop, such as the Service node in this example, may prevent your iterator from reaching the final $jump to return $eos.

Be aware that no XML tree or resource should be referenced by the iterator and the output document. Your iterator may need to copy information to prevent crossed-references. For example, an XML tree is changed in the loop but the original iterator retains a reference to it.

The following syntax is an example of a simple count iterator:

```
private int counter=0;  // we keep counter here
private XDDocument originalDoc;
  public String execute(XDDocument docIn, XDDocument docOut)
        throws  XDException
    {
        String edgeName = getEdgeName();
        if (!edgeName.equals("$jump")) // if not on "loop" edge, this is
start of iteration
        {
            docIn.moveTo(originalDoc);   // save for iterations
            counter = 0;
        }
        if (counter > countWanted)  // have we reached the count of
iterations? If so, return a status document
        {
             XDNode statusNode=new XDNode("status");
             statusNode.setAttribute("count",""+counter);
             docOut.setRoot(statusNode);
             originalDoc=null;   // help the GC
             counter=0;
             return "$eos";   // tell engine that end of set has been
reached
        }
        else
        {
            counter++;
            originalDoc.copyTo(docOut);   // iterating document
            return "success";
        }
    }
Public boolean isIterator()
{
      return true;
}
```

Assume that the limit has been set into the countWanted field. This will loop and return a copy for the number of times that are required. Note that copies are returned, not simply references to the original. A more realistic iterator might process the incoming document on the entry for iteration, perhaps simply reloading the payload (the XML or other content), which allows other changes to the document to be made in the iteration loop. In addition, this example returns a trivial status document to the $eos point.

Chapter 3

Writing Management Components

Management components provide stateful services to the server as a whole. Like business components discussed in *Writing Business Components* on page 31, management components are exits that can be installed and configured. Like business components, they can be developed as exits for specific purposes. This section discusses how these exits work and how to develop such exits for your purposes.

Developers of management components are expected to be fully familiar with the concepts discussed in *Writing Business Components* on page 31.

In this chapter:

Getting Started With Management Components

System management component exits provide overall services such as activity logging. Unlike business components such as agents or preparsers, management component exits can provide stateful services.

As standard exits, these exits receive their metadata in the normal manner, and also have available such services as tracing. The services are provided at the server level rather than at the message level; for example, traces are issued with origin as the manager rather than the message executor. The writing of exits in general and the use of common methods is discussed in *Writing Business Components* on page 31.

The exits are initialized via the init() method when the server starts, and are terminated via the term() method as the server shuts down. Although the server makes every effort to call the term() method, certain system failures can prevent this and so exits should not depend upon receiving a term() for correct operation.

Unlike business exits, management exits can run on any thread and must be fully reentrant. Developers are cautioned to avoid non-synchronized object variables and expectation of sequence. Likewise, thread-specific data cannot be used. Developers are guaranteed that the init() and term() calls are controlled by the server and are not subject to reentrancy issues.

Users are cautioned that exits should be reentrant. Although some synchronized blocks may be necessary, their use should be reduced. Synchronization can lock the server and have a disproportionate effect on system performance.

Any component exit needs to be included in an extension and registered with the system. Use the addExit() method, discussed in *Installing Components* on page 100. However, the standard reference is done.

The name of the jar file containing the extension that you build should begin with the letter 'I'. This is a simple filter used do avoid attempting to investigate non-extension jars, and thus improve loading time.

Developers are encouraged to make management components "self-healing." Failure should not prevent the system from running and the exit should attempt to recover from errors. For example, an exit that loses access to a data base might hold its data in memory or on a disk until the data base is recovered during a subsequent entry to the component.

Activity Log Writer

An activity log writer receives control at strategic points in a message cycle:

- When a message is acquired.
- When a message is emitted.
- When an error occurs.
- When a component such as an agent or process flow is called.

The driver can maintain a log or perform another activity as desired.

The log writer extends *com.ibi.edaqm.XDLogBase*. The calls pass in the worker, internal signature information, and the message being worked on. For example, the start method receives the original document, while the emit entry receives the message just prior to it being emitted.

Most calls pass timing information. The *now* field is the current time (obtained by System.getCurrentTimeMillis()) when the call is made. Synchronizations or simply dispatching rules may prevent the call from reaching the driver at that time, however, this is the actual time of the event. The *cpu* and *user* fields provide CPU and user time as defined for the iWay measurement extension. These fields are set to 0, unless the iWay measurement extension is installed and operational.

Log entries provide the current context in the form of access to the special register manager. Using the manager, the context can be iterated and thus examined and stored.

startEntry

Called for any type of message except streaming input type.

```
public int startEntry(XDWorker worker,
         byte[] who,
         String protocol,
         XDOrgData orgdata,
         ISpecRegManager srm,
         long now, long cpu, long user)
```

endEntry

Called when the message operation ends, regardless of whether there were intermediate emits.

```
public int endEntry(XDWorker worker,
       byte[] who,
       String status,
       String msg,
       ISpecRegManager srm,
       long now, long cpu, long user)
```

msgEntry

Logs a message to the transaction log. Messages are those emitted as BIZERROR messages. In the event of an error, error information is usually logged as standard ending information.

```
public int msgEntry(XDWorker worker,
       byte[] who,
       int level,
       String message,
       ISpecRegManager srm,
       long now, long cpu, long user)
```

emitEntry

An emit operation is being performed to send a message to a recipient.

```
public int emitEntry(XDWorker worker,
        byte[] who,
        String protocol,
        String destination,
        byte[] data,
        ISpecRegManager srm,
        long now, long cpu, long user)
```

eventEntry

Events include the start and end of a component such as an agent or the parser, or other significant events. The call passes the event code, the "direction", and any appropriate text. For example, a pair of event entries surrounding the calling of an agent will contain the name of the agent.

- Operation begin/end
  - Agents
  - Process flows
  - Transformations
  - Rules operations
- Listener/channel begin/end
- Security events
  - Signature operations
  - S/SMIME operations

A complete list of event codes is given in the Javadoc for the LogBase class.

```
public int eventEntry(XDWorker worker,
         byte[] who,
         int eventCode,
         int action,    // 0=begin, 1=end, 2=fail
         String message,
         ISpecRegManager srm,
         long now, long cpu, long user)
```

Activity Driver

Activity Log events are sent to configured drivers. Each driver is responsible for filtering the events of interest and writing the event information to appropriate targets. iWay provides several drivers that can be used individually or in combination:

- **XDSQLWriter.** Writes the required events to a relational data base. This data base can be queried for inquiry purposes using standard relational data base tools. The SQL objects in server-supported process flows can be used for this purpose.
- **XDTraceLogger.** Writes the method calls to a sequential file. This tool is useful for debugging purposes, as it shows each input value.
- **XDTranLog.** Writes the activity log information to a sequential file in the format of the prior Adapter Manager servers. Programs designed to work with a log will find identical input files. The prior activity log support is deprecated and users that wish to produce “old style” logs are encouraged to use this driver.

XDSQLWriter Activity Driver

XDSQLWriter Activity Driver inserts records into the activity database formatted as described in this section. The records are designed for fast writing rather than for ease of later analysis. A set of inquiry service agents suitable for use in process flows is available to assist in analysis of the log. Users are cautioned that iWay does not absolutely guarantee the layout of the record from release to release, and this should be checked against the actual schema.

Fields in the Database	Description
recordkey	Unique record identifier.
recordtype	Type of this record - the event being recorded.
signature	Encoding of the listener name and protocol.
protocol	Name of the protocol.
address	Address to which an emit is to be issued. The format depends on the protocol.
tstamp	Time of record.
correlid	Correlation ID if present.

Fields in the Database	Description
tid	Transaction ID assigned to this message.
msg	Message appropriate to this record type. For example, an input message contains the original message received, if possible. Streaming input does not contain a record.
context	Serialized special registers that were in the context at the time the record was written.
text	Message text for business errors (rules system violations).
status	Status code recorded. Success if 0; other status codes are based on the type of record being recorded.
subtype	Event code for event records, for example, parsing, agent calls, and returns.
partner_to	If the TPA has recognized a to party, the value is put here.
partner_from	If the TPA has recognized a from party, the value is put here.
encoding	Encoding of the listener that obtained the document.
mac	Hash encoded by a symmetric algorithm. Not used in this version.
version	Driver version (1.0 in 5.5.SM).

Note:

- Partner fields contain the information extracted from the TPA via the TPN() function. For example, an EDI record has a field in the ISA that encodes partner information. This is passed into TPA to get the partner's name.
- Mac is an encoded field of a hash value. It is intended to ensure that the record has not been modified by an unauthorized party. Once implemented, it cannot be null.
- Time stamps are in compressed RFC 1138 form such as 20050509130705021Z.
- The context is a string of token=value[;token=value]* elements. This consists of all simple value special registers in the context at the time the record was written, with the exception registers marked as secure or hidden.

Record Type Codes	Description
101	Message start.
131	Entry to event (see subtype codes).
132	Normal exit from event.
133	Failed exit from event.
151	Ancillary message (usually rules violation).
181	Emit.
191	Message end.

Record Subtype Codes	Description
1	Preparser
2	Parser
3	In reviewer
5	In validation
6	In transform
7	Agent or flow
8	Out transform
9	Out validation
11	Preemitter

Trading Partner Agreement

A Trading Partner Agreement (TPA) driver enables information from an external data store to be accessed during message handling. Such data stores often hold information relating to the processing of a message based on characteristics of the message; most commonly, the trading partner. For this reason, iWay considers these data stores to house trading partner information, although they can hold any information organized in an arbitrary fashion.

Information from the data store is accessed using the _tpa() function, which provides information from the data store at any point in the flow in which the _tpa() function is referenced in the configuration. This is similar to the xpath() and sreg() functions. But, because the format of the data store housing this trading partner information can be expected to vary, Service Manager routes trading partner information requests through drivers, referred to as TPA drivers. Each driver is charged with handling some aspect of a TPA request, and any single request must be resolved by a single driver.

Access Components

Each request provides the following information:

A TPA Class ID. This is an arbitrary token that distinguishes drivers. There is no restriction that drivers support unique class IDs, only that they recognize the ID. An example of a class ID might be "XML" for a driver that works with XML files, or "AUTO" for a driver that works with automotive requests.

Attribute wanted. This is the name of a single attribute to be located in the data store.

Default. The default value is returned, if the attribute wanted is not located.

Search context. One or more tokens used to direct the search within the driver. For example, if the driver works with a Java properties object, with keys such as

```
Toyota.wheels=4
```

the context would be "Toyota" and the configured request might be:

```
_tpa("CARS","wheels","4",xpath(/sales/car/model))
```

Similarly, if the TPA takes the form of a relational data base, the context variables might resolve to values used in select statements to locate the desired data.

Driver Resolution

When a _tpa() request is made, Service Manager evaluates the class ID and context to determine whether it knows which of the configured drivers is to receive the request. If it does not already "recognize" the class/context, it inquires of each driver in configuration order whether it can service the request for attributes based on that class ID and context. The first driver to respond that can service the request will receive this request and all subsequent requests for the combination of class ID and context.

Driver Internals

Drivers, like all other iSM exits, extend XDExitBase, which provides the driver with parameter resolution services, tracing, and so on. The driver itself extends XDTPADriver, which provides driver-specific exit services. If the driver exists in an extension, the XDTPADriver class is responsible for loading the driver, as is customary for exits. The driver should (by convention) be in the com.ibi.exits package.

Each driver must expose, in addition to the standard init() and term() methods, two "business" methods:

```
public int handles(String tpaid, String[] context) throws Exception;
```

and

```
public String access(String tpaid, String[] context,
String attribute, String deflt) throws Exception;
```

The handles() method returns whether the driver can service requests for the combination of tpaid(class) and the context variable values. It can select to handle all requests for the specified tpaid or only requests for the tpaid in combination with the context access pattern values. The handles() method returns:

Mnemonic	Value	Description
HANDLES_NO	0	Cannot handle this tpaid/context pattern.
HANDLES_YES	1	Can handle this tpaid/context pattern.
HANDLES_ALL	2	Can handle any call for this tpaid, regardless of the context values.

The access() method is called by iSM to obtain the attribute value based on the context values, which the driver has already reported that it can support.

For example, consider a driver that obtains TPA values from an XML file. The file looks like this:

```
<tpa>
  <group1>
    <attr1>value1</attr1>
    <attr2>value2</attr2>
  </group1>
  <group2>
    <attr1>value1</attr1>
    <attr2>value2</attr2>
  </group2>
</tpa>
```

The init() routine accepts two parameters: the class ID that this driver represents (it might have been the root element name as well) and the path to the XML file. The init() routine creates a list of the class IDs it supports and then parses the file and loads a second list with the names of the first level children (group1, group2). This example assumes that the object context map is constructed containing class IDs that this driver represents.

```
public int handles(String tpaid, String[] context) throws Exception
{
  if (tpaidList != null && !tpaidList.contains(tpaid))
  {
      return HANDLES_NO;
  }
  if (contextMap == null)
  {
      return HANDLES_ALL;   // supports all contexts
  }
  if (contextMap.containsKey(context[0]))  // this specific context
  {
      return HANDLES_YES;
  }
  return HANDLES_NO; // context required and not in my context
}
```

After the driver has reported that it handles the specific request, it will be called upon to service the request. It might do this by taking the context and finding that subtree, and within that subtree locating the desired attribute:

```
public String access(String tpaid, String[] context, String attribute,
 String deflt) throws Exception
{
  XDNode contextRoot = root.findChild(context[0]);
  XDNode attrNode = contextRoot.findChild(attribute);
  if (attrNode == null)
  {
    return deflt;
  }
  else
  {
    return attrNode.getValue();
  }
}
```

As above, no error handling is provided in this example.

Assuming that the driver handles "XML" as the class ID, then the call in the configurator might have been:

```
_tpa("XML",attr2,"default",sreg("toparty"))
```

If the toparty special register holds the name of the to party in the TPA, which in our case is either "group1" or "group2", the external call to the TPA function will look like:

```
_tpa("tpa","attr1","def1",sreg("tpa.toparty"))
```

Correlation Manager

Correlation Manager maintains records of anticipated activities occurring in the system. Correlation actions take the correlation from OPEN to CLOSED state, and allow history to be recorded. Agents are provided to implement Correlation Manager interactions within process flows, however, it is possible to use this API to accomplish this same purpose within your own exits.

Correlation Manager in Programs

A static controller, XDCorrelExitControl, offers an API for adding, updating, and inquiring on records. It passes method calls to the stateful services handler which executes the requests. The handler executes the requests by calling each correlation driver to offer it the opportunity to deal with the specific request. Drivers manage one or more namespaces (qualifiers), which can offer each driver the opportunity to provide specialized services for specific message types. Drivers cannot overlap namespaces; the first driver to report that it handles the namespace qualifier will receive all messages for that qualifier.

Static methods provide access to the stateful services.

Although your application can issue any of the calls documented in the interface, usually you will be opening and closing correlations. The provided methods for this capability are:

```
public static int addCorrelationEntry(
    String correlID,
    String namespace,
    String tid,
    String msg,
    String exp,
    int dupeStrategy,
    String correlSetID,
    String userDef,
    String comment,
    boolean isSet);
```

The following table lists and describes the related parameters.

Parameter	Description
correlID	Correlation identifier. Should be unique within namespace.
namespace	Namespace qualifier.
tid	Transaction ID. Can be retrieved from sreg(tid) through a special register lookup.
msg	Reserved for future use. Pass null for now.
exp	Expiration string as a duration. For example, one hour can be 1h.
dupeStrategy	How to handle duplicates. Choices are to ignore a duplicate or to return an error.
correlSetID	Identifier of this correlation set. It is the responsibility of the application to ensure that the child correlations are in the same namespace as the correlation set.
userDef	User field for any purpose.
comment	Comment for use in displays.
isSet	Indicates a correlation set record or a correlation interaction. Correlation set records are used for long running transactions, and need not be created for a simple correlation interaction. The XDBaseCorrelDriver will automatically change this parameter to true when a child interaction is added to this information.

The return code depends upon the dupeStrategy selected.

```
public static int updateCorrelationEntry(
     int msgType,
     String correlID,
     String namespace,
     String tid,
     String msg,   // Reserved for future use
     boolean satisfy,
     String comment);
```

The update function adds a history entry to the correlation and, optionally, closes it. If you are working with a correlation set entry for a long running transaction, refer to the correlation entry by its correlation ID within its namespace. Closing such a master correlation set record (which was created by setting the isSet parameter of the add function to true) closes all interactions under the set.

The noteworthy parameters are described in the following table.

Parameter	Description
correlID	Correlation identifier. Should be unique within namespace.
namespace	Namespace qualifier.
tid	Transaction ID of the event that causes the history/change. It is not that of the original transaction that caused the add to be performed.
msg	Reserved for future use. Pass null for now.
satisfy	True if the correlation is to be closed. This closes all interactions in the set for correlation set entries.

Inquiring About Correlation

Correlation Manager offers the ability to inquire on the state of correlations. By using these inquiries, your program can obtain lists of correlations in given states and can drill down to details about the correlations.

As with the open and update calls, you make the inquiry calls through the XDCorrelExitControl static object.

Each of the calls returns a status code and an XDNode tree of information. The tree is rooted by the correl_mgr element that describes the query arguments. The query result set appears as children of the correl_mgr element. The schema for the tree is shown in the *iWay Trading Manager User's Guide*.

The getCorrelStatus method returns a single correlation record.

```
public static int getCorrelStatus(List result, String correlid,
   String namespace, boolean history);
```

Parameter	Description
result	XDNode tree will be returned as the first element in this List.
correlid	Correlation identifier. Should be unique within namespace.
namespace	Namespace qualifier.
history	If true, all history records are included as child nodes. If false, only the status record is provided.

The status of a correlation set can be returned directly from the correlation set identifier. To obtain all information pertaining to the correlation set, you can follow the chain of returned information available through this call.

```
public static int getCorrelSetStatus(List result,
   String correlSetID, String namespace, boolean detail);
```

Parameter	Description
result	XDNode tree will be returned as the first element in this List.
correlSetID	Identifier of this correlation set.
namespace	Namespace qualifier.
detail	Whether to include the status of child correlations in the report.

You can obtain information on any interaction within a date range or in a given state.

```
public int getCorrelActivity(
     List result,
     String namespace,
     boolean history,
     int timetype,
     int state,
     String sdate,
     String edate,
     int maxrows)
```

Parameter	Description
result	XDNode tree will be returned as the first element in this List.
namespace	Namespace of interest. Use * or null for all namespaces.
history	Whether the history events are included in the report.
timetype	Meanings of the dates given in the sdate and edate parameters.
state	OPEN, CLOSED, ALL, or PROBLEM. Only interactions in the given state are returned.
sdate	Low date of the range in compressed RFC 1138 form.
edate	High date of the range in compressed RFC 1138 form.
maxrows	Number of rows to return. Use 0 for all.

Developing Correlation Manager Drivers

Any drivers added to the stateful service bus must implement the complete API. The driver must extend com.ibi.edaqm.XDCorrelDriver and must implement com.ibi.common.IXDCorrelDriver.

The driver must implement all of the calls in the interface, as described in this manual and the User's Guide.

Writing iWay Function Language (IFL)

The server provides a wide range of functions, such as xpath(), sreg(), tpa(), ldap(), file() and so on. It is possible to supply a new function in an extension that will be compiled by the function compiler and executed just as an iWay function.

Functions are registered by the addFunction(name,className) function of the extension's register() function, in the same manner as listeners, emitters, console pages, log exits, and so on are registered.

Functions extend com.ibi.funcs.FunctionNode. Your function must offer a null constructor that calls a parental constructor with the "root" name of your function. This becomes the name of the function representation in the Abstract Syntax Tree created during the compile step. In the example, the name of the function is REVERSE, and only the reversal service is offered. If you call addFunction() more than once, adding multiple functions for the same class, then the class name is assigned at construction, and the setFuncName() function can be used to distinguish entries. An example of this is a STRING class that might support substr, length, startswith, and so on.

The name of your function should be clearly distinguishable, and it is suggested that it start with a non-alphabetic character. This assists in avoiding confusion when the compiler is passed over other functional languages for which there may be overlap. The most common example is confusion created by function names that mirror SQL scalar functions. The initial underscore that iWay uses helps to alleviate this issue.

Your function returns a string value when the getXValue method is called. When this is called, the parameters may have been resolved by an upward walk, so that the parameters are themselves strings. In other cases, you must evaluate them. You do this by calling the getXValue() function, which causes a walk down the tree.

For example, if your function is named REVERSE and you want to reverse the characters in a special register named sr1, you configure the call

```
_reverse(sreg(sr1))
```

passing one parameter to the reversal function. When the getXValue method of the reversal function is called, it can get the parameter and reverse it.

Parameters are children of the node in the AST. The node itself extends XDNode, enabling you to use the standard tree calls to step through your parameters.

Functions designed to participate in Boolean operations should return strings of "true" or "false". Any value not exactly "true" is considered to be false, but iWay strongly recommends that the two values be returned as specified.

Each function has, in addition to the getXValue() method needed to obtain the function result, an optional optimize() method, receiving the manager and the logger, is called during construction of the abstract syntax tree. A common use of optimize() is to load the parameters, preventing the need for a treewalk during actual execution. This is shown in the example below. Although the example function has one parameter, and because the parameter is required for compilation, use of the array is simplified. Functions with multiple parameters, some of which may be optional, can interrogate the array in the getXValue() method to determine the actual parameters passed in at runtime.

Functions should return metadata describing their purpose and parameters. This is done using the getDesc() function. The getDesc() returns a tree of information. The following is an example of the _inflate() function tree.

```
<func  name='_inflate' class='encoding'>
  <desc>Inflate a compressed value</desc>
  <parm name='value' type='string' req='yes'>The string to inflate</parm>
  <parm name='type' type='keyword'  req='yes'>Type of input
       <kw name='base64' >The compressed data is in base 64 form</kw>
       <kw name='string' default='yes'>The decompressed data is a UNICODE
string</kw>
  </parm>
</func>
```

The example function is:

```
package com.ibi.funcs;
import com.ibi.edaqm.*;
import java.io.StringReader;
import java.util.*;
/**
* Sample function extension. This is called with
* one parameter, to reverse its value
*/
public class REVERSE extends FunctionNode
{
       // the constructor must call super() with the name of the AST node
       public REVERSE() // for when I make default nodes
       {
            super("REVERSE");
       }
        // called by the interpreter with the name of the function for which
       // this instance is created. Useful if multiple functions are
       // implemented.
       public void setFuncName(String funcName)
       {
       }
       // returns the name of the function for error handling
       public String getTypeName()
       {
            return "REVERSE";
       }
       public XDNode getDesc()
       {
            String desc = "<func name='_reverse' class='string'>" +
                          "<desc>Reverse a string</desc>" +
                          "<parm name='value' type='string' req='yes'>" +
                          "The string to reverse</parm>" +
                          "</func>";
              try (StringReader sr = new StringReader(desc); XDParser parser
            = XDParserFactory.borrow())
            {
                 parser.parseIt(sr);
                 return parser.getResult();
```

```
            }
            catch (Exception e)
            {
                return null;
            }
        }
        // how many parameters are needed by this instance
        public int getMinParms() // minimum parms is one
        {
            return 1;
        }
        // what is the maximum number of parameters I need
        public int getMaxParms() // maximum parms is one
        {
            return 1;
        }
        // this will be an array of parameters
        private FunctionNode[] parms;
        // Optimize is called during ASTgeneration
        public void optimize(XDManager manager, com.ibi.common.IXLogger
        logger) throws XDException
        {
            super.optimize(manager, logger);
            parms = getParms(); // get the array of parameters
        }
        // return the value -- parms are operands in order
        public String getXValue(XDDocument doc,
          com.ibi.common.ISpecRegManager srm, com.ibi.common.IXLogger logger)
        throws XDException
        {
            // get the first parameter
            String value = parms[0].getXValue(doc, srm, logger);
            if (value.length() == 0)
            {
                throw new XDFunctionException("_reverse", 1, "length to
                reverse cannot be zero");
            }

            StringBuffer sb = new StringBuffer(value);
            sb = sb.reverse();
            return sb.toString();
        }
}
```

Once written, you must register the function with the system. Do this using the addFunction() registration call. For example,

```
package com.ibi.example;

 public class Register implements com.ibi.common.IComponentRegister
 {
   public Register()
   {
   }

   public void register (com.ibi.common.IComponentManager cm)
   {
      cm.addFunction("_REVERSE","com.ibi.funcs.REVERSE");
      cm.addFunction("_reverse","com.ibi.funcs.REVERSE");
      System.out.println("**** registering _REVERSE  ****");
   }

   public void unregister (com.ibi.common.IComponentManager cm)
   {
   }
 }
```

The manifest as described in *Installing Components* on page 100 points to the registration method, which uses addFunction() to install a function. The name of the extension jar containing the new function must, like all extensions, begin with the letters “iw”. The name of the function must be defined with an underscore character (_).

Service Providers

Providers supply runtime services that intermediate between server component code and a common capability or service. Providers are configured once and referred to by name when configuring the component. For example, there might be several PKI keystores referenced by individual components such as SSL, digital signature, and so on. Each keystore would be accessed through a provider that is configured accordingly. By referencing the provider name in the component, the component gains access to the provider's services and configuration. Using providers enables the configuration to be centralized and simplifies use of the supplied service.

Among the provider types supported by the server are data access, keystore, SSL, namespace maps and LDAP.

Providers are instances of classes that offer a type-specific interface appropriate to the service being provisioned. With the exception of the data access provider, which offers the standard SQL DataSource interface, the interfaces are iWay proprietary and subject to change in future releases.

Accessing Providers

Access to a specific provider requires knowledge of the provider type and the name of the specific provider instance desired. Access to the desired provider follows the JNDI method of locating named objects.

Note: It is possible to access providers that are offered externally to the iWay server, although at this time only Data Sources offer standard usage patterns.

Accessing JNDI first requires the construction of a context factory. The default context factory is the iWay server, however, the formal statement of this is always a good idea:

```
Hashtable ht = new Hashtable(2);
ht.put ("java.naming.factory.initial",
"com.ibi.jndi.XDInitialContectFactory");
```

Once the Hashtable has been constructed, you can create your JNDI context, for example:

```
Javax.naming.Context ctx = new InitialContext(ht);
```

Now that the context is available, it can be used to locate the required provider. The access name consists of the type and the configured name as a string <type>/<name>. For example, a JDBC data source (JDBC provider) named "correltable" would be accessed as:

```
DataSource ds = (DataSource)ctk.lookup("jdbc/correltable")
```

The following table lists the object type names and their associated class instances:

Name	Class is	Explosed Methods
jdbc	DataSource	javax.sql.DataSource
ldap	XDLdapProvider	iWay Specific
dircertstore	XDDirCertstoreProvider	iWay Specific
keystore	XDKeystoreProvider	iWay Specific
sslcontext	XDSSLContextProvider	iWay Specific
nsmap	XDNsmapProvider	iWay Specific

Providers extend XDProviderbase, which offers some common methods. In turn, XDProviderBase extends XDExitBase. In this manner, providers set their configuration metadata and are configured in the same manner as other exits.

For more detailed information on the individual providers, see the Javadoc that is provided with the product.

Chapter 4

Writing Protocol Components

iWay components include listeners or emitters. Listeners are channels through which documents reach the iWay Service Manager processing flow. Each listener handles a specified protocol such as FTP or HTTP. Emitters are channels through which documents leave the process flow on a specified protocol.

Components differ from exits, which are application-oriented and intended to be easily written by end users. Components are usually more complicated, provide no application logic, and are thus document-agnostic. The exit interface is a published API, and is consistent across iWay Service Manager releases. Exits run in the iWay Service Manager container. Components, on the other hand, are release specific, and may need to be modified as the needs of the iWay Service Manager container change. Components run outside of the iWay Service Manager container.

In this chapter:

Protocol Components

Protocol components implement channels through which iWay Service Manager gets messages and emits messages. The protocol components are:

- **Listeners** get messages to be processed. They obtain their configuration information from the properties stored when the server was configured. For synchronous, correlated responses, they provide the associated emitter with the properties needed to emit back to the source of the message. Listeners set special registers to record information about the message, such as header values, input source, and so on.
- **Emitters** pass messages out of the server. They obtain their configuration either from the console by accessing the listener properties for synchronous responses, or the replyto properties for non-correlated messages. In addition, a business agent can set emitter properties explicitly by loading a hash map with the properties to be used for a specific message. Some emitters consult special registers to control their activity or to further format the output. An example is HDR-level special registers to set header values.

This chapter discusses technical details of writing and packaging components, and is intended for a select audience of those well-versed in Java and familiar with the architecture and implementation of iWay Service Manager. Those preparing components are encouraged to follow the design patterns in a current component. Some good ones to look at are:

- **XDTickMaster** represents a listener that receives control from the XDManager class, which maintains state for each protocol. Some masters are simply created by the manager, while others are run in threads. For more information, see *XDTickMaster* on page 109.
- **XDTickWorker** represents a listener that is controlled by the XDTickMaster. There is one XDxxxWorker for each thread of execution or document that can be handled in parallel. Each XDxxxWorker extends XDWorker, which manages the processing sequence. For more information, see *XDTickWorker* on page 111.

All emitters follow the same pattern. The example shown in this manual of the Line Handler follows the first pattern. The pattern you select must be appropriate to the protocol.

This document is not intended to be a complete description of writing components. Such work should be done in conjunction with the iWay Service Manager development group. It does, however, provide an overview of the work that must be done and how the pieces fit together.

Programming Standards, General Rules of the Road

Components are critical parts of the server, and must adhere strictly to the programming standards listed in *Programming Your Applications in iWay Service Manager* on page 11. Components are expected to operate with no human intervention. In addition, the following rules apply:

- All components must be self-recovering. The components extend engine classes that perform much of the recovery, but your component should never simply throw an exception or ever call System.exit() for any reason.
- All parameters must be in the run time catalog (dictionary), described by the MasterInfo class associated with your component. Specifically, do not create your own properties objects.
- All components must be ISO I18N compliant. The engine provides facilities to assist in this. Never call getBytes() without an encoding.

Build Environment

Your component is stored in a directory structure off of an assigned location. The following directory structure uses iWay guidelines.

Directory	Description
\src	Root of the Java source. The package structure descends from here.
\ini	Root of the JNI elements. The package structure descends from here.
\test	junit tests root from here. Under \test are the packages needed and \data.
\doc	Documentation.

An example of an ANT script that can generate a component jar is in *Sample ANT Build Script* on page 184.

MasterInfo Class

A MasterInfo class exists for each protocol. The MasterInfo describes the parameters needed to configure a listener and an emitter. To prepare a MasterInfo, extend the MasterInfo base class with a specific instance. A MasterInfo is static, so that the tables appear once in memory, but are instanced as needed to make available their methods to validate entered parameters.

When you develop a component, you provide a reference to the static instance of your MasterInfo in the addListener() and addEmitter() calls.

The following syntax is an example of a MasterInfo class.

```
package com.ibi.config;
import com.ibi.edaqm.XD;
import java.util.*;
public class MasterInfoTick extends MasterInfo
{
    public static final MasterInfoTick INFO = new MasterInfoTick("Tick",
"Tick", "Passes input on a predefined interval");
    private MasterInfoTick(MasterType type) {
        super(type);
    }
    private MasterInfoTick(String innerName, String patternName, String
description) {
        super(innerName,patternName,description);
    }
    public static final MasterPropertyInfo RETRY =
    new MasterPropertyInfo(XD.KW_TIMEOUT, "Execution Interval", "2.0",
PropertyType.DECIMAL_TENTHS, MasterPropertyInfo.REQUIRED,
                           "Interval between cycled requests",
"#listener.schedule.timeout");
    public static final MasterPropertyInfo FILE =
    new MasterPropertyInfo(XD.KW_FILE, "Input File", "", PropertyType.FILE,
MasterPropertyInfo.OPTIONAL,
                           "Name of a file to use for repeating input, if
empty, use builtin document", null);
    public static final MasterPropertyInfo INTYPE =
    new MasterPropertyInfo("type", "Operation Type", "tree",
PropertyType.CHOICE, MasterPropertyInfo.OPTIONAL,
                           "Type of message to pass through: tree is
preparsed, bytes are parsed each pass", null);
    static {
        Map typeValues = new TreeMap();
        typeValues.put("tree", "tree: preparsed");
        typeValues.put("bytes", "bytes: parsed each pass");
        INTYPE.setAllowedValues(typeValues);
    }
    // List of all the master properties
    private static final MasterPropertyInfo[] PRIVATE_VALUES =
    { CommonMasterProps.ACTIVE,
        RETRY,
        FILE,
        INTYPE,
        CommonMasterProps.XALOG,   // removed own parm and use common,
SOC-727
        CommonMasterProps.COUNT,
        CommonMasterProps.MAX_LIFE};
    private static final List VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
    /**
     * Returns a List of the master's properties.
     * @return a List of the master's properties.
     */
    public List getMasterProperties() {
        return VALUES;
    }
    // List of all the master properties
    private static final MasterPropertyInfo[] PRIVATE_REPLYTO_VALUES =
    {};
    private static final List REPLYTO_VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_REPLYTO_VALUES));
    public List getReplytoProperties() {
        return REPLYTO_VALUES;
    }
}
```

Installing Components

Components are packaged into jars. These jars can contain any number of listeners and emitters. An example of a jar with more than one component is the iWay FIX jar which contains two listeners, but no emitter.

Some components provide both listeners and emitters, such as Oracle AQ. Some provide only listeners, such as FIX. Still others provide only emitters, such as Print.

Your components are assumed to be in the appropriate packages. You must also provide a loading registration class, in any package. A manifest entry identifies the loading class.

The manifest contains an iXTEComponent section, used by iWay Service Manager to load the component. It might look like:

```
Manifest-Version: 1.0
Created-By: Apache Ant 1.5.1
Built-On: EDARDB March 16 2005 1622
Version: 5.5.xxxx
Level: Development
Label: IW55xfoc.0190
Build: 0190
Name: IXTEComponent
Register: com.ibi.example.Register
```

Naturally, you want to set information fields as appropriate. The component installer looks for the IXTEComponent name section.

The Register attribute identifies the class which is run by the engine during startup to load and prepare the component. Note that the .class extension is not used.

Note: The manifest command on the console can display a manifest.

An ANT script might be used to construct the jar:

```
<target name="jar" depends="compile" >
   <mkdir dir="${build.dir}/meta-inf"/>
   <tstamp/>
   <manifest file="${build.dir}/meta-inf/Manifest.mf">
      <attribute name="Built-On" value="${env.COMPUTERNAME} ${TODAY} ${TSTAMP}"/>
      <attribute name="Version"  value="${version}"/>
      <attribute name="Level"    value="${level}"/>
      <attribute name="Label"    value="${label}"/>
      <attribute name="Build"    value="${build.num}"/>
      <attribute name="Root"     value="${build.root}"/>
      <section name="IXTEComponent">
          <attribute name="Register" value="com.ibi.edaqm.Register"/>
      </section>
   </manifest>
   <echo message="Building ${engine}.jar"/>
   <jar
      jarfile="${out.dir}/${engine}.jar"
      basedir="${build.dir}"
      manifest="${build.dir}/meta-inf/Manifest.mf"/>
</target>
```

The Register class must implement com.ibi.common.IComponentRegister. This interface provides two methods: register() and unregister(). The register() method is called when the engine needs to load the component. Unregister() must be implemented, but is reserved for future use.

The register() method is called by the engine before the trace mechanism has been established. It is passed a reference to a class that implements the com.ibi.common.IComponentManager interface.

```
public interface IComponentRegister
{
    /**
     * Register the listener(s) and emitter(s) with the
     * iWay Service Manager engine.
     * This method may be called by a static initializer so you should program
     * accordingly.
     *
     * @param icomponentmanager
     * The component manager's interface provides methods to interact with the
     * engine to register the components.
     */
    public abstract void register(IComponentManager icomponentmanager);
    public abstract void unregister(IComponentManager icomponentmanager);
}
    /*
     * Unregister reserved for future use.
     */
```

The component manager is a part of iWay Service Manager that loads and unloads components.

```
public interface IComponentManager
{
public String getLanguage();
  public int addEmitter(com.ibi.config.MasterInfo mi, String shortName,
             String patternName, char separator);
  public int addListener(com.ibi.config.MasterInfo mi, String shortName,
             String patternName);
  public int addExit(String fullName, String shortName);
  public int  addFunction(String name, String className);
}
```

The IComponentManager interface offers the methods that the register() method can use to identify listeners and emitters to iWay Service Manager. Once these methods have been called, the component can be used by iWay Service Manager. The component appears in the iWay Service Manager Console, sorted into its lists in alphabetic order appropriate to the language. Such loaded external components are indistinguishable at run time from built-in iWay Service Manager components.

```
import com.ibi.common.*;
public class Register implements com.ibi.common.IComponentRegister
{
    public Register()
    {
    }
    public void register (com.ibi.common.IComponentManager cm)
    {
       cm.addListener(com.ibi.config.MasterInfoFoobar.INFO,"foobar","Foobar");
       cm.addEmitter(com.ibi.config.MasterInfoFoobar.INFO,"foobar","Foobar",'\0');
       cm.addExit("com.pkg.agents.fooagent","FOOAGENT");
    }
    public void unregister (com.ibi.common.IComponentManager cm)
    {
    }
}
```

The register method can also add exits to the configuration, as is seen in the addExit() call in the example. The exit's type is automatically determined and the exit is defined for use under the short name provided in the call.

The methods addListener(), addEmitter(), and addExit() are described in detail in the appropriate java for the server.

Installing Your Protocol Component

To install your protocol component, add the .jar file to the etc/manager/extensions directory. If this directory does not exist, you must create it. The server startup service adds the .jar files in this directory to the classpath at startup time, and checks each .jar file beginning with the letters “iw” in this directory for the appropriate manifest entry.

The environment entry IWAY7 is set up by the server installation to locate the root of your installation. Note that the actual name of the installation root variable is subject to change. During server execution, the value of the root variable is available in special register *iwayhome*.

Providing Protocol Metadata

Both the listener and the emitter require metadata. There is only one metadata file for any single protocol, and the name must end with the protocol name. In the example of a line protocol handler for the console, the protocol name is line. The sample protocol metadata object is shown in the following code. Only a single parameter is provided - a suffix to be applied to each output line. Parameters can be far more sophisticated, supporting additional types such as lists, integers, and so forth.

```
package com.ibi.config;
import com.ibi.edaqm.XD;
import java.util.Map;
import java.util.List;
import java.util.Collections;
import java.util.Arrays;
import java.util.TreeMap;
/**
 * The MasterInfoFile class is a concrete MasterInfo implementation.
 * It defines the master of type Line.
 *
 */
public class MasterInfoLine extends MasterInfo
{
  // Create the one and only instance of this object
  public static final MasterInfoLine INFO = new MasterInfoLine("line", "Line",
    "Line Handler");
  /**
   * Constructs a new MasterInfo
   * @param type the type of the master.
   */
  private MasterInfoLine(MasterType type)
  {
    super(type);
  }
  private MasterInfoLine(String innerName, String patternName, String description)
  {
    super(innerName,patternName,description);
  }
  public static final MasterPropertyInfo SUFFIX =
      new MasterPropertyInfo("prefix", "Suffix", "", PropertyType.STRING,
                             MasterPropertyInfo.OPTIONAL,
                             "A suffix to be added to each line", null);
```

```
/*************************************************************************/
/* Properties specific for this master                                   */
/*************************************************************************/
  // List of all the master properties
  private static final MasterPropertyInfo[] PRIVATE_VALUES =
   { CommonMasterProps.ACTIVE,   // only support active flag
     SUFFIX
   };
  private static final List VALUES =
      Collections.unmodifiableList(Arrays.asList(PRIVATE_VALUES));
  /**
   * Returns a List of the master's properties.
   * @return a List of the master's properties.
   */
  public List getMasterProperties()
  {
    return VALUES;
  }
  public String getReplyToDesc()
  {
    return "Prefix";
  }
  /**
   * Returns description for the replyto type.
   */
  public String getReplyToDescription()
  {
    return "Emit to the console with System.out";
  }
  // List of the reply-to properties
  private static final MasterPropertyInfo[] PRIVATE_REPLYTO_VALUES = {SUFFIX};
  private static final List REPLYTO_VALUES =
    Collections.unmodifiableList(Arrays.asList(PRIVATE_REPLYTO_VALUES));
  public List getReplytoProperties()
  {
    return REPLYTO_VALUES;
  }
  public boolean hasDestination()
  {
    return false;
  }
}
```

Writing an iWay Service Manager Emitter

iWay Service Manager emitters provide the output exchange between iWay Service Manager documents produced by the system and the actual destination medium. Emitters are provided for each medium that the facility supports, such as MQ Series, the local file system, FTP, TIBCO Rendezvous, and so on.

Emitters are bound into the system using the addEmitter() method of the component interface.

The emitter itself must be in the package com.ibi.edaqm, and it must be named according to the pattern XDxxxEmit.java.

During emit, the user or configurator creates an XDReply for the appropriate address, and passes into it a map containing parameters. In many cases, the XDReply can be used for more than a single emitter. For example, the XDReply constructed for a queuing emitter can be used in association with any queuing emitter. System-specific parameters may need to be supplied, but common parameters are commonly named and validated.

When the emitter is used, the XDReply is passed into the emitter, and serves as its source of parameters.

The emitter itself extends com.ibi.edaqm.XDEmit, which provides all of the logical services needed by the emitter to interact with the engine. The specific emitter must support four methods:

- **Constructor**, which is passed the XDReply object and the XDWorker object. You can use the XDWorker to emit traces, and for other information, such as access to special registers.
- **Emit**, which is passed by the XDDocument to be emitted. The document can be an XML tree or it can be a flat document. The emitter calls upon the parental emitter to run preemitters and encryptors or compressors. The emit routine usually does the actual writing to the external media. If your emitter does its actual output in the emit() method, commit() and rollback() serves no purpose. In such a case, the output cannot be controlled by the current transaction.
- **Commit** is called to complete the emission activity. For a file emitter, commit closes the file. For MQ Series, commit issues the MQ Commit call.
- **Rollback** prevents completion of the emission activity. When this is received by the file emitter, as an example, the file is closed and deleted.

Associated with each emitter is a separator character used to divide the two parts of the address to which the emitter emits. The addEmitter() call specifies this character. For example, email uses the @ character (name@emailservice), while HTTP uses the : character (// host:port). If you have no separator, use the '\0' character.

A very simple emitter is shown in the following code. This emitter takes the document and displays in the console using System.out. Although this is not a real emitter, it demonstrates the major components.

1. An XDReply, built by the configuration or manually by an agent, contains the addressing information. If you are simply using the console to configure your listener, you do not have to be concerned with the source of the reply object. The reply object has the address and any parameters defined in the MasterInfoLine that provides the metadata for this emitter.

2. The document to be emitted arrives in the emit() call. You prepare for emit here.
3. The commit() call tells the emitter to actually emit the output.

```
package com.ibi.edaqm;
public class XDLineEmit extends XDEmit
{
    XDTickMaster master = null;
    String prename = null;  // stick this in front of each message
    String suffix="";
    String message;  // message to put out is created here
    // the line emitter example uses an address of name (part1) which it simply
    // prepends to each string
1.  public XDLineEmit(XDReply reply, XDWorker worker) throws XDException
    {
        master = (XDTickMaster)worker.getMaster();
        super(reply);
        this.worker = worker;
        prename = "["+reply.part1+"]: ";
        if (!reply.useMaster)   // was this from the Master itself?
        {
           worker.setLogger(),debug("worker is not XDTickWorker");
           htParams = reply.htParams;  // pick up user's parameter map to XDReply
           if (htParams == null)
           {
               return;
           }
           String s = (String)htParams.get("suffix");
           if (suffix != null)
           {
               suffix = "["+s+"]";
           }
```

```
            // do something with the parms here
            // close (!reply.useMaster)
        }
        else
        {
            // extract parameters from your own master here if this is meaningful
            String s = master.getSuffix();
            if (suffix != null)
            {
                suffix = "["+s+"]";
            }
        }
    }
2.  public void emit(XDDocument outdoc) throws XDException
    {
       review(outdoc);
       try
       {
           byte[] b = preEmit(worker.dictionary, outdoc);  // get what to emit
           message = prename+new String(b)+suffix;
       }
       catch (XDException x)
       {
           logger.debug("  ");
           worker.logger.debug("document setup got error "+x);
           throw new XDException((Exception)x);
       }
    }
    // actually issue the message
3.  public void commit()
    {
        logger.debug(message);
    }
    public void rollback()
    {
        message=null;
    }
}
```

Writing an iWay Service Manager Listener

iWay Service Manager listeners receive messages on an appropriate protocol, and pass them on to the document processing sequence. Good practice dissociates the receipt of the message of its handling, which is performed by the processing sequence.

Listeners always consist of two classes:

- XDxxxMaster, which extends XDMaster. The xxx represents the protocol name, as defined by the addListener() pattern. The XDxxxMaster receives control from the XDManager class, which maintains state for each protocol. Some masters are simply created by the manager, while others are run in threads. The master isStartable() attribute (which defaults to true) controls whether the master is started or simply instanced by the manager. See *XDTickMaster* on page 109.
- XDxxxWorker, which is controlled by the XDxxxMaster. There is one XDxxxWorker for each thread of execution or document that can be handled in parallel. Each XDxxxWorker extends XDWorker, which manages the processing sequence. See *XDTickWorker* on page 111.

The example shown is for a channel master/worker pair that watches clock ticks. While functionally and syntactically correct, it is simplified for example use.

The developed channel master/worker pair directly extends XDMaster and XDWorker, which are documented in the supplied Javadoc. iWay strongly discourages direct extension of superclassed protocols such as XDFileMaster and XDFileWorker should you need to develop your own variation. The implementation of specific protocols can vary from release to release and direct extension can result in your protocol not performing correctly as releases change.

XDTickMaster

```
package com.ibi.edaqm;
import java.util.*;
import java.net.*;
import java.io.*;
import com.ibi.config.MasterInfoTick;
public class XDTickMaster extends XDMaster implements Runnable
{
  public XDTickMaster()
  {
    init("Tick");
  }
  public XDTickMaster (HashMap xdm, XDManager mgr, XDControl xdc)
  {
    super(xdm, mgr, xdc);   // store properties and set common values
    init("Tick");
  }
  private static final int TREE=0;
  private static final int BYTES=1;
  private byte[]  inputArray= null;
  private XDNode  treeRoot= null;
  private String fileName = null;
  private int inputType = 0;
  /**
   * Initialize the master (channel). Gather parms
   * and check that the master can run.
   *
   * @exception XDException
   */
  public void init() throws XDException
  {
    initGlobalFields(); // fills in general parameter we need for execution
    fileName = property(MasterInfoTick.FILE);
    String t = property(MasterInfoTick.INTYPE);

    if (!isPresent(t))
    {
      t="tree";
    }
    if (t.equals("tree"))
    {
      inputType = TREE;
    }
    else if (t.equals("bytes"))
    {
      inputType=BYTES;
    }
    else
    {
       throw new XDException("Specified input type '"+t+"' not valid");
    }
    if (isPresent(fileName))
    {
      File f = new File(fileName);
      if (!f.exists())
      {
        throw new XDException("Specified file "+fileName+" does not exist");
      }
      if (inputType == TREE)
      {
        try
        {
          XDParser parser = new XDParser();
```

2. Initialize the master initGlobalFields() to load common properties, then load your own. All properties are visible to you by the property() call.
3. Start up the routine. If you throw an exception, iWay Service Manager retries your master later.
4. The manager starts a thread to do the work. If you want the manager to simply call your run method without giving it a separate thread, add the metadata method:

   ```
   boolean isRunnable()
   {
       returns false;
   }
   ```

   The default is true.
5. When the manager needs to stop your master, it calls two methods. The super.stopAll() gets the stopActivity flag in XDMaster, which this code tests at the top of its run loop.
6. awaitTimeoutCycle() helps you poll by "sleeping" for the timeout period established in your configuration. Each second it checks for the stopActivity flag, and returns immediately if it is set. You can provide your own logic here to configure the loop. awaitTimeoutCycle() is a convenience, but it is not required.

XDTickWorker

```
package com.ibi.edaqm;
import java.io.*;
import java.net.*;
import java.util.*;
import com.ibi.common.ISpecRegManager;
public class XDTickWorker extends XDWorker
{
  private XDMaster Master;
  XDTickWorker(XDTickMaster master) throws Exception
  {
    super(master);
    this.Master = master;
  }
  void resetWorker()
  {
    resetBaseWorker();
    isBusy = false;
    makeWorkerAvailable();
  }
  public void run()
  {
    int  rc=0;
    for (;;)
    {
      if (stopFlag || Master.stopActivity)
      {
        break;
      }
      try
      {
        try
        {
          waitForDocument();
          if (stopFlag)   // in case stopFlag set while waiting
          {
            break;
          }
        }

        catch (InterruptedException e)
        {
          logger.error("Exception waiting for document: "+e);
          resetWorker();
          continue;
        }
        xddIn.reset();
        rc = XDException.SUCCESS;
        try
        {
          parseAndValidate(xddIn);
        }
        catch (XDException e)
        {
          rc = e.getType();  // what class of exception do we have?
          logger.error("Exception from ParseAndValidate rc="+rc+": " + e);
          rc = XDException.FAIL;
        }
        catch (Exception e)
        {
          logger.error("Exception from ParseAndValidate: " + e,e);
          rc = XDException.FAIL;
        }
```

master thread when told to stop. It finishes any work in process and then returns.

2. The stop() method is called by the master when the master is ready to stop. This example signals to run loop to stop before starting another piece of work.
3. The master passes work to the worker (process loop) by calling here. Set isBusy to prevent the worker from being selected for another message while this one is in process.
4. When the message is complete, this tells the master that the worker can be reassigned to another message.

Writing an Extension Command

Extensions can offer commands through the command handler. Once offered, the command is available to users using any command interface (for example, command prompt console, Telnet command console, scheduler, and so on). An extension command has the following format:

```
name operand*
```

where:

name

Identifies the command and the extension. For example, *scheduler* and *bam*. As a best practice, try to emulate existing commands where possible.

To add an extension command, your code must extend the following class:

```
com.ibi.edaqm.XDCommandBase
```

For example:

```
public class CmdSchedule extends XDCommandBase implements IXDCommandItem
{
  Your Code}
```

The command is registered with the system in the register method for the extension, as:

```
XDCommandHandler.setExtensionCommand("name");
```

where:

name

Is the same value that you specified for the extension command.

The command received control at its execute method. For example:

```
void execute(StringTokenizer tokenizer, String inputLine) throws XDException
```

where:

tokenizer

Can break up the line. The original line is also available in the inputLine parameter.

The XDCommandBase class contains several methods to assist in writing the extension command. The say() and sayerr() methods allow you to communicate with the user. Although the say() method can be used to provide information at any time, the sayerr() method passes an error, which your command should then return. The sayerr() method offers the opportunity to provide an error code.

The following is sample syntax for the sayerr() method:

```
void sayerr(FailCause fc, int specific, String msg)
FailCause: A value for the failure type
Specific: an integer added to the base value - usually 0
Msg: Explanation
```

You should use XDCommandBase services only from extension commands that have been properly loaded.

The Failure causes are described in the FailCause enum:

SECURITY	1000	A general security error class.
SYNTAX	2000	A general syntax error (add a specific value).
UNKNOWN	2001	An unknown keyword has been detected.
MISSINGINFO	2002	Information needed for this command is missing.
SYNTAXVALUE	2015	A numeric value is not valid or out of range.
SYNTAXDUPL	2016	Duplicate switch.
SYNTAXXML	2018	A document is not a well formed XML.
SYNTAXURL	2019	A URL is not valid.
RESOURCE	3000	A resource needed for the command is not available.
FILENOTFOUND	3500	A file needed is not found.
SPECIFICATION	5000	Cannot use the command in this situation.
IO	6000	IO error.

These values are meant to avoid random error codes. If your error meets a specific, already enumerated, category, then you must make the specific cause 0. If it simply falls into a category, then you can add a value to make the error more specific. Be careful not to overrun another category.

Note: Additional standard error classes may be created in future releases.

Your code can also take advantage of the following static method to expand a partial token to a full token:

```
String com.ibi.xdutils.StringUtils.recognizer(String[] candidates, String
token)
```

The array must contain the candidates in ASCII order.

For example, to expand the partial input switch in the following flow command:

```
private static String[] flowtoks={"-commit","-flat","-map","-output","-xml
"};
which = recognizer(flow, which);
```

This example expands -o to -output.

Adding Help

You can add help by adding the following lines to your translation file:

```
cmd.<commandname>=\t\%1 One line explanation of the command
subcmd.name=fuller explanation
```

The last line responds to the help name.

To explain a subcommand, use the following line:

```
subcmd.name.subcommand=fuller explanation of subcommand
```

If you want to hide a subcommand, then do not add this line.

The following is an example of sample syntax that is placed in the com.ibi.commands package.

```
/**
 * reverse <string>
 * reverses the string entered on the command line
 */
package com.ibi.commands;
import java.util.StringTokenizer;
import com.ibi.edaqm.XDCommandBase;
import com.ibi.edaqm.XDCommandHandler.FailCause;
public class CmdReverse extends XDCommandBase {
  public void execute(StringTokenizer tokenizer, String inputLine)
         throws Exception
  {
    if (inputLine.trim().length() < 8)  // account for the command name
    {
      sayerr(FailCause.MISSINGINFO,000,"No string to reverse");
      return;
    }
  StringBuilder sb = new StringBuilder(inputLine.substring(8));
  String r = sb.reverse().toString();
  say(r);
  }
}
```

Running a Process Flow

The execution component of iWay Service Manager (iSM) is the process flow. Process flows are run in response to input routing decisions or at other times as required. The mechanism described in this section allows applications to run process flows. It is first necessary to prepare the way with a dictionary and a worker, and then to provide a compiled process flow in internal form. These methods generally assume that the process flows have been built by iWay Designer and published to the registry.

As a best practice it is recommended that applications use only the existing components, and avoid use of these methods unless no other options are available.

Understanding the PFlowExecutor API

The PFlowExecutor API is designed to simplify the process of running a process flow from an iSM component or a standalone Java program.

Since a process flow requires an XDWorker to execute, the PFlowExecutor constructor takes an XDWorker instance as an argument. Runtime components of the server, like services and preparsers, generally have access to an XDWorker instance and can pass this. However, in some cases, like console pages, functions, and standalone programs, no worker is available and it is the responsibility of the caller to create one. Normally, this will be an XDMTLclWorker, as shown in the example below.

Methods

```
XDPFlowState runProcessByName(String flowName, XDDocument docIn, boolean
cache, boolean commit, boolean overrideTimeout) throws XDException
```

Given the name of an available system flow and an input document, execute the flow. Returns an XDPFlowState object that describes execution status.

Parameter	Type	Description
flowName	String	Name of the flow to execute. The named flow must be defined at the system level for the worker that will execute the flow. When using a local worker, make sure that the flow is available in the dictionary used to create the local master.
docIn	XDDocument	Input to the flow. This may be either an XML or flat XDDocument.
cache	boolean	Workers are normally related to instances of XDMaster, which can maintain a cache of reusable process flow objects. If this argument is set as true, the PFlowExecutor will attempt to retrieve the flow from the master's cache and will cache the flow after execution. When the caller expects to reuse a worker or to use workers related to a common instance of XDMaster, this option can greatly improve performance.
commit	boolean	Determines whether the worker should commit or rollback the local transaction after flow execution is completed. This is the equivalent of the local transaction setting in a configured channel.
overrideTimeout	boolean	If true, a flow with no timeout set will be assigned a 10 second timeout.

```
XDPFlowState runProcess(String flowName, XDNode flowNode, XDDocument docIn,
boolean commit, boolean cache, boolean overrideTimeout) throws XDException
```

Given either a pflowloc node from an XDictionary or the root of a compiled flow, execute the flow with the input document provided. Returns an XDPFlowState object that describes execution status.

Parameter	Type	Description
flowName	String	Name of the flow to execute. The named flow must be defined at the system level for the worker that will execute the flow. When using a local worker, take care that the flow is available in the dictionary used to create the local master. When flowNode is supplied, this parameter is not required unless cache is true.
flowNode	XDNode	This can be either a pflowloc node as used in the system defines area of the runtime dictionary or the root of a compiled flow (see compileGUIFlow method below).
docIn	XDDocument	Input to the flow. This may be either an XML or flat XDDocument.
cache	boolean	Workers are normally related to instances of XDMaster, which can maintain a cache of reusable process flow objects. If this argument is set as true, the PFlowExecutor will attempt to retrieve the flow from the master's cache and will cache the flow after execution. When the caller expects to reuse a worker or to use workers related to a common instance of XDMaster, this option can greatly improve performance. When cache is true, be sure to supply flowName.
commit	boolean	Determines whether the worker should commit or rollback the local transaction after flow execution is completed. This is the equivalent of the local transaction setting in a configured channel.
overrideTimeout	boolean	If true, a flow with no timeout set will be assigned a 10 second timeout.

```
XDNode compileGUIFlow(String guiFlow, String configId) throws XDException
```

Given the raw output of iWay Designer, compile this into an executable form. Returns the root of the compiled flow. This is supplied as a convenience for running tests from design tools.

Parameter	Type	Description
guiFlow	String	String containing a process flow as output from iWay Designer.
configId	String	The name of the iWay Service Manager configuration where the compiler should check for dependencies when compiling the GUI flow.

```
List<XDDocument> getOutDocs()
```

After flow execution, retrieve output documents from the flow.

```
String getReturnState()
```

For convenience, after flow execution, return the execution status as a string.

```
void reset()
```

If using the PFlowExecutor more than once, call reset() after each flow execution to clear the results of the previous flow.

Understanding the XDPFlowState Object

These functions return a XDPflowState object. This object is used internally by the process flow interpreter, which uses it to manage transitions. You can determine if the flow succeeded by examining the type of the state object. The method is:

```
int getType()
```

Returns a code as listed and described in the following table:

Value	Name	Purpose
0	XDPFlowState.NORMAL	The process flow was successful.
1	XDPFlowState.COMMIT	The process flow was successful/reserved for future use.
2	XDPFlowState.RETRY	The process flow ended with retry.
3	XDPFlowState.FAIL	The process flow ended with failure.

Success is indicated by a normal return. Any other return indicates a failure. The transactional operation (commit or rollback as appropriate, if configured in the call) will have taken place before the return from the return from the flow execution.

Example #1: RunPFlowSampleAgent

This service demonstrates the use of PFlowExecutor when an instance of XDWorker is available.

```
package com.ibi.agents;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import com.ibi.common.IXLogger;
import com.ibi.config.ConfigurationException;
import com.ibi.edaqm.XDAgent;
import com.ibi.edaqm.XDDocument;
import com.ibi.edaqm.XDException;
import com.ibi.edaqm.XDNode;
import com.ibi.edaqm.XDUtil;
import com.ibi.pflow.PFlowExecutor;
import com.ibi.pflow.XDPFlowState;
/**
 *
 * Attempts to execute the specified process flow using the PFlowExecutor
API.
 * If the flow outputs more than one document, additional
 * documents will be output as siblings to the first.
 *
 */
public final class RunPFlowSampleAgent extends XDAgent
{
  public RunPFlowSampleAgent()
  {
    super();
  }
  public String getDesc()
  {
    return "Attempts to execute the specified process flow using the
PFlowExecutor API.";
  }
  public String getLabel()
  {
    return "Sample PFlow Agent";
  }
  private static String[] flowNameParm = { "flowname", "string", "Name of
the process flow,
          defined at the system level, to execute", "",    "yes", "",
"Process Flow" };
  private static String[][] parmsMeta = { flowNameParm, };
  String[][] parmsEnums = { null, };
  private HashMap parmMap; // parms are stored here
  public void init(String[] parms) throws XDException
  {
    // merge input parameters, account for required parms, and make final
parameter map
    parmMap = initParms(parms, parmsMeta);
  }
  /**
   * Provides metadata to the Adapter Designer as to which flow edges are
expected to be followed.
   * If you plan to return fixed names, return them in the array of this
method. These names will
   * appear in the designer as standard returns.
   *
   */
  public String[] getOPEdges()
  {
    String[] r = {
     "success",
     XDAgent.EX_FAIL_NOTFOUND,
     XDAgent.EX_FAIL_OPERATION,
    };
```

Example #2: RunPFlowSample

This standalone Java program runs a process flow by creating a local worker based on an existing iWay Service Manager configuration.

```
package com.ibi.samples;
import java.util.HashMap;
import java.util.List;
import com.ibi.edaqm.XD;
import com.ibi.edaqm.XDDocument;
import com.ibi.edaqm.XDException;
import com.ibi.edaqm.XDMTLclMaster;
import com.ibi.edaqm.XDNode;
import com.ibi.edaqm.XDWorker;
import com.ibi.pflow.PFlowExecutor;
import com.ibi.pflow.XDPFlowState;
/*
 * Agent to run a pflow by name as a stand-alone command. Note that
 * this is linked with the supplied lib from the server install.
*/
public class RunPFlowSample
{
 private String dict;
 private String configName;// name of the configuration
 private String flowName;  // name of the flow
 private String input;     // the input to process
 private boolean isXML;    // is theinput document XML?
 private List<XDDocument> outDocs;
 public RunPFlowSample(String dict, String configName,
                              String flowName, String input,
                              boolean isXML)
 {
  this.dict = dict;
  this.configName = configName;
  this.flowName = flowName;
  this.input = input;
  this.isXML = isXML;
 }
 public int runFlow()
 {
     XDWorker worker = null;
  // set up a local worker, based on the supplied dictionary and
configuration
  try
  {
   worker = getLocalWorker(dict, configName);
  }
  catch (XDException xde)
  {
   logger.error.println("Error: Unable to construct worker from dictionary
file");
   return -1;
  }
  // verify requested flow exists -- this uses an internal accessor
  XDNode pflowlocNode =
worker.getMaster().getDictionary().findSystemPFlowNode(flowName);
  if (pflowlocNode == null)
  {
   logger.error.println("Error: Unable to find requested pflow in
dictionary");
   return -1;
  }
  // set up XDDocument, flat or XML, for input
  XDDocument inDoc = new XDDocument(worker);
  if (isXML)
  {
   try
   {
    inDoc.toXML(input);
```

Chapter 5

Programming Objects

iWay Service Manager operates on and with several objects that appear in various points during a document flow. Some of these key objects are described in this section. Some detail is included, however, more detail is provided in other sections of this manual and in system Javadoc.

In this chapter:

XDDocument

An XDDocument is the carrier of the message being processed. Documents usually hold XML messages, but can be set to flat mode to hold byte arrays or strings.

During the message life cycle in iWay Service Manager, it is first read from a protocol, then passed to a preparser if any is defined. Once past the preparser, an XDDocument is immediately created to carry the message through the remainder of the system. Documents can be moved from one XDDocument to another, for example, a business processing agent performs some operation on the incoming XDDocument and loads the results into an outgoing XDDocument.

The XDDocument carrying the final state of the document is passed to the emitter to be carried from the system to its destination.

The XDDocument carries XML messages as a tree of XDNodes, described below. The XDDocument itself provides some convenience methods for interacting with the tree. A full node and tree manipulation API is provided by the XDNode class itself.

SOAP business agents are identical to regular business agents. The system properly handles the envelope. The getRoot() method of the document deals only with the payload.

XDDocuments always hold a reference to the listener under which they function. This enables document operations to properly handle I18N considerations and to obtain other configuration information that may be needed. Therefore, never use the empty constructor when creating your own XDDocument. Always use the constructor that accepts an XXMaster, XDWorker, and so on. In any exit, you can accomplish this by entering:

```
XDDocument d = new XDDocument(this);
```

The document provides a variety of useful methods.

XDDocument Methods

Method	Description
addLine	Adds a data line to a flat document.
addPreEmitter	Adds a preemitter to the document.
addReplyTo	Adds one destination to an output document.
addSibling	Adds a preconstructed document as a sibling of this document. Can be a standard document (one containing an XML tree) or a flat document that holds non-XML.
copyTo	Copies the data (tree or flat data) from the document to an output document.
createSiblingDocument	Creates a new document that is a sibling of this document. The sibling document can carry a different tree and have different destination addresses. During output, all output document siblings are sent. This method allows a business agent to produce multiple outputs.
findByName	Locates a subtree within the document tree. This method is provided for convenience, and a full complement of tree manipulation methods are provided by the XDNode interface.
flatten	Flattens the tree in the document to a string.
getEncoding	Returns the IANA encoding for this document.
getJavaEncoding	Returns the Java encoding for this document. Usually this is the same as the IANA encoding.

Method	Description
getRoot	Returns the root of an XDNode tree that is the basis of the document.
isFlat	Returns true if this is a flat document, false if this is an XML document.
moveTree	Moves the payload from an input document to an output document. The payload can be XML or flat.
printTree	Flattens the tree in the document to a string with embedded carriage returns. This is useful only for tracing. If the document is large or supports staging, a truncated tree may be generated.
resetPreEmitters	Eliminates all associated preemitters.
setEncoding	Sets the IANA encoding for this document. Internal tables attempt to determine the appropriate Java encoding.
setFlat	Sets the document to flat or XML.
setInError	Sets the document error state. Any emit is sent to the errorTo address, if available.
setReplyTo	Adds a reply destination. Any single XDDocument can hold one or more reply addresses, and each sibling document can hold its own unique set of addresses.
setRoot	Sets the root of an XDNode tree as the basis of the document.
xnode	Returns a list of nodes meeting the XPATH expression passed as a parameter. The expression is supported as a selector into the current document, and does not support the full XPATH syntax.

Documents can carry non-XML information. This can be string or byte arrays. The form of the document is determined by the type of data stored; the flatten() routines always convert the contents to displayable form for tracing. Non-XML documents carry information between channel components just as do XML documents.

Controlling Listeners From Exits

It is possible to control listener activity from exits. Control can be accomplished by posting a message to a listener using postMessage.

Messages that can be posted are:

Message	Description
ACTIVATE	Starts the listener if it is in STOP or PASSIVE state.
STOP	Stops the listener. This is the equivalent of using the stop command from the console. Any outstanding messages are completed before the stop takes effect.
PASSIVATE	Listener stops accepting messages, although it remains active. The listener continues to check for a STOP request.
PULSE	Listener in PASSIVE state accepts one cycle of messages and reenters PASSIVE state. This is not supported by all listeners.

The XDControlAgent exposes these messages to the agent flow. However, you can issue them from your own code as needed. Users are strongly cautioned that use of posted messages can seriously affect performance and can get the system "out of state" in a way that is hard to debug.

Sibling Documents

Siblings are subdocuments associated with the document being processed. A sibling is created by an exit, which is responsible for setting any emit parameters and values into the document. Once created, the sibling is carried through the processing stream by the host document to which it is attached.

When a host document reaches the end of the processing flow, each sibling is individually emitted. Although siblings can have siblings, and any document can have any number of siblings, there is no logic currently applied to the position of the document in the sibling "tree"; each is treated individually in the order accessed.

Use of siblings is rare – the most common use is to create documents for emit that require different preemitters yet are created during the main document life cycle.

Siblings do not automatically inherit the attributes of the document to which they are applied. Attribute inheritance can be accomplished by creating a document and adding it as a sibling of the exit output document.

```
XDDocument ed = new XDDocument(this,inDoc);
ed.setRoot(<what I want emitted>);
outDoc.addSibling(ed);
```

Alternately, you might create a document and add your own errorTo or replyTo addresses using the appropriate XDDocument method calls.

A convenient use of siblings is to send error messages to one destination and the document to another. In such a case, set the new sibling error state to true via setInError(true). When the main document is emitted, the sibling is emitted to the errorTo address(es).

A process flow provides an iterator that can send each sibling through subsequent processing steps. The iterator “breaks” the sibling relationship, enabling the flow to work with each sibling as an individual document.

Making Error Documents

Two convenience classes are provided to assist in construction of error documents. These documents are emitted to the errorTo address(es). The first, XDErrorDocument, constructs a standard error document carrying, where possible, the original input in the <data> section of the document. The second, XDUEDocument, sets the data section of the error document to the contents of the document passed in as a template. These classes extend XDDocument, and may assist you in creating standard error documents; there is no requirement to create error documents by this approach.

The following code example constructs an error document and passes it plus the payload contents of the template document on for further action. The payload is carried in the <data> section of the error document in a form appropriate to the type of payload being carried.

```
public String execute(XDDocument docIn, XDDocument docOut) throws XDException
{
   // sample for copy process only; remove this and insert your code here
   copyTree(docIn, docOut);
   XDNode enode = new XDNode("testerror");
   docIn.setRoot(enode); // add my error tree, use docin for convenience
   XDUEDocument ed = new XDUEDocument(
     getWorker(), // link into this flow
     docIn, // source of addresses and data (model)
     XDUEDocument.PIPE_AGENT, //flow position of error
     "test", // actual source of error
     null, // additional attributes to include
     -222, // my error code
     "This is error string"); //my error description
   docOut.addSibling(ed); // chain to main document
   return "success";
}
```

To reprocess the contents of the error document, you can configure the ErrorFilter to extract the contents of the <data> portion of the standard error document and pass it on through the system.

Document Attachments

XDDocuments provide methods for conveniently working with attachments. These methods enable an exit or protocol component to:

- Determine whether a document has attachments.
- Obtain a specific attachment associated with the document.
- Add an attachment to the document.

Attachments can optionally be carried by both XML and flat documents. That is, all operations on the document content continue to work; the extra payload is carried with the document, and is copied from document to document during exit chaining and process flows. When the document is emitted, the standard multipart preemitter, if configured, emits the attachment part ignoring the XML or flat content. You can easily write a preemitter that mixes the two, depending upon your application needs. Conditional routing services apply to the carried document XML payload, and do not respect attachments carried along by the document.

Multipart documents reaching a listener are parsed by the provided multipart preparser, and the attachments are made available to the XDDocument API as discussed in this section. The API presumes that an incoming multipart document has been preparsed by the multipart preparser, although you can add attachments to any document. If you do not parse an incoming multipart document with the provided preparser, the incoming attachments are not available to the attachment API.

Details of the API are discussed in the provided Javadoc.

The API supports several methods:

Method	Description
addAttachment	Adds one attachment to the document, which has been prepared by the openMultiPart() call. You provide a name, headers, and the content data.
closeMultiPart	Finalizes any added attachments. Failure to call this method results in the loss of added attachments, and possibly unpredictable contents of the document.

Method	Description
getAttachmentBytes	Returns a byte array of the contents of the attachment.
getAttachmentCount	Returns the number of attachments. A document with no attachments returns 0.
getAttachmentHeaders	Returns a map of the headers for a specific attachment. All header keys are returned as lowercase.
openMultiPart	Preparses the document to receive new attachments. These can be added to a document with no attachments, or to one that already has attachments.

You need to call openMultiPart only to manipulate attachments, and not to get attachments.

When adding an attachment, the header map should contain all information needed to process the attachment, which varies by MIME type. You must include the content-type header. Handlers are provided for most of the standard types.

The following code example, taken from an exit, adds an attachment to a document, and then retrieves the attachment.

```
try
 {
    int count = doc.openMultiPart();  /* get the count of current attachments */
    Map headers = new HashMap(4);
    headers.put("content-type","text/plain");   /* set the content type */
    docOut.addAttachment("Attach"+(count+1), headers,
                         ("Attachment data "+count).getBytes());
    count = doc.closeMultiPart();
    byte[] in = doc.getAttachmentBytes(count-1);
    debug("got back '"+new String(in)+"'");
 }
 catch (Throwable t)
 {
    throw new XDException(t);
 }
```

This example should return the text of the added attachments. Naturally, there is no limit to the number of attachments inserted into a document that has been opened for multipart operations.

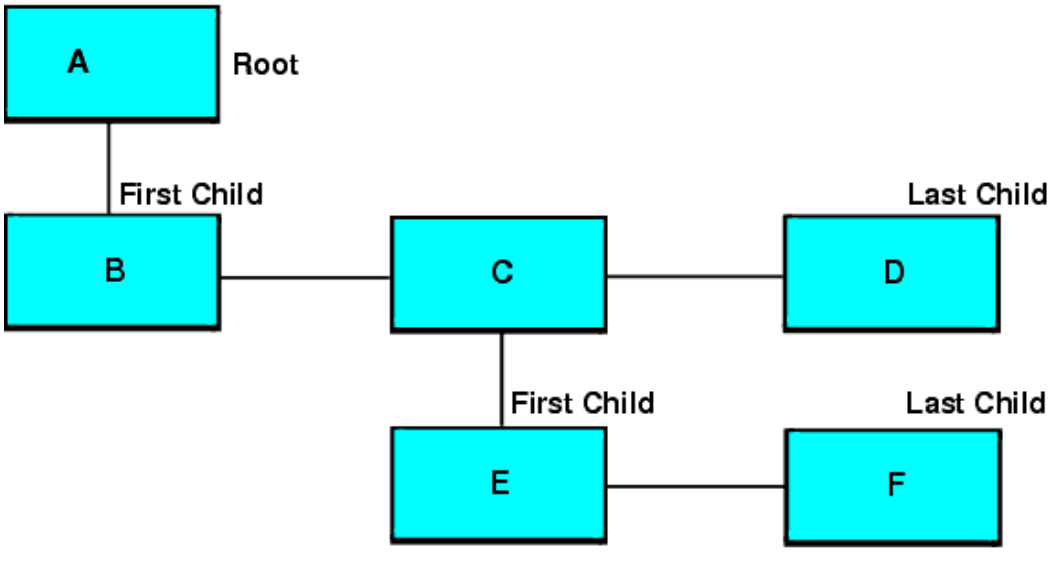
An iWay Service Manager tool, CreateMultipart, is provided to help you construct test documents. For more information, see *Supplied Tools* on page 219.

XDNode

An XDNode is a single element of an XML tree. A complete document is a tree of XDNodes. The XDNode class and tree are designed for fast parsing and searching, and for easy manipulation in an application. Methods are available to convert between XDNode trees and standard JDOM trees. All server operations are performed on trees of XDNodes.

Each XDNode represents one "element" of the XML document, including the attributes, comments, and processing instructions that apply to it. Method calls are provided to get and set this information, along with the node's name, value, relationships with other nodes in the tree, and so on.

A document of type XML encapsulates a tree of XDNodes. Each XDNode is the root of a subtree or is a leaf. The usual task of a business agent is to create one or more output XML documents in the form of XDNode trees, and add each to an XDDocument. The base output XDDocument is provided to the business agent, and the business agent can create one or more siblings if multiple output documents are to be emitted.




```
<A><B/><C><E/><F/><C ATR='IWAY'>cvalue</C><D/></A>
```

The XDNode class provides a set of standard tree manipulation methods, all of which are explained in the associated Javadoc. Key methods are listed in the tables below. When considering a node, the XML has three parts: name, value, and attributes.

```
<NAME ATTRIBUTE="ONE">VALUE</NAME>
```

Important: Do not allow a node to be in two trees at the same time, or a tree to be in two documents at the same time.

The standard search of nodes is down-right recursive. Some search calls have the appendix "AsRoot", such as, findByNameAsRoot(). These methods treat the nodes in which they are called as a subtree, and do not search to the right on their root node.

Results of Node Calls for Node C

Method	Returns
getParent()	Node A
getFirstChild()	Node E
getName	"C"
getRight	Node D
getValue()	"Cvalue"
getAttribute("ATR")	"iWay"
getAttribute("x")	Null

XDNode Methods That Deal With Nodes

Method	Description
findByFullNameAsRoot	Locates a node by the child name below the current node.
findByName findByNameAsRoot	Locates a descendent (following) node of a specified name.
findByNameEnd findByNameEndAsRoot	Locates a node with a name that ends with the given sequence.
findByNameIC findByNameICAsRoot	Locates a node by name, case insensitive.
findByValue findByValueAsRoot	Locates a descendent (following) node with a specified value.

Method	Description
findChild	Locates a node by name that is a direct descendent of this node.
getFirstChild/getLastChild	Returns the associated child of the node.
getParent/getLeft/getRight	Returns an associated node.
setLastChild	Adds a node as the last child of a parent node.
snipNode	Eliminates the node from the tree. Optionally, this method can replace the snipped node with a new node. The nodes may be the roots of trees.
swap	Swaps two nodes in the tree.
XDNode.newNode	Static constructor for a node. Faster than creating a new XDNode.

XDNode Methods That Deal With Values

Method	Description
appendValue	Appends the passed sequence to the current value of the node.
getAttribute	Returns the value of the attribute associated with the key. If there is no such attribute, null is returned.
getName	Returns the name associated with the node.
getValue	Returns the value associated with the node. If no value can be found, null is returned. If a mapped value points into another part of the document, the XPATH is followed to the value.
setAttribute	Adds an attribute to the node or deletes an attribute.
setProcessingInstruction	Adds one processing instruction to the node.
setValue/setName	Sets the appropriate name or value data.

Method	Description
toflat	Flattens this node and its children to a string.
toString	Flattens the node in the document to a string.

XDNode Methods That Return Lists

Method	Description
findAllByAttributeValue	Locates all nodes with a specific attribute of a specific value.
findAllByFullName	Locates all nodes meeting a path specification.
findAllByName	Locates all nodes with the passed name.
findAllChildrenByName	Locates all nodes of the given name that are children of a specific node.
getAllChildren	Returns all children of the node.
getComments	Returns a list of any comments that apply to this node.
getProcessingInstructions	Returns a list of processing instructions applicable to this node.

Namespace Support

The XDNode API has been augmented with namespace support. The old constructor creates an element without a namespace or prefix.

```
XDNode level1Element = new XDNode("elemName");
    // level1Element.getNamespaceURI() returns null
    // level1Element.getLocalName() return null
    // level1Element.getName() returns "elemName"
```

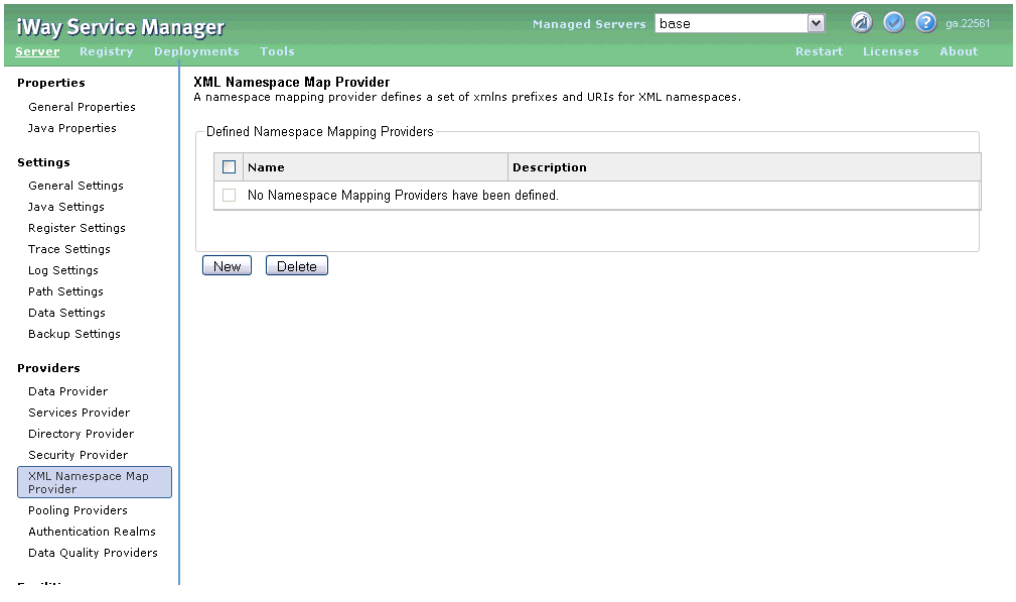
A new constructor creates a namespace-aware element. The application is responsible to create the XML namespace declaration explicitly on the node or on one of its ancestors.

```
XDNode level2Element = new XDNode("http://ibi.com", "elemName",
"ns:elemName");
                    level2Element.setAttribute("http://www.w3.org/2000/
xmlns/", "ns", "xmlns:ns", "http://ibi.com");
    // level2Element.getNamespaceURI() returns "http://ibi.com"
    // level2Element.getLocalName() return "elemName"
    // level2Element.getName() returns "ns:elemName"
```

The second and third argument of the constructor overlap somewhat, but they were chosen to match the arguments provided by a SAX2 parser. Notice there is no verification of the namespace prefix "ns". As you can see from the example, the XDNode is allowed to be inconsistent momentarily. An xmlns declaration must exist for the prefix on this node or an ancestor, otherwise the node will serialize to an invalid XML document. An xmlns declaration is just a regular attribute with the required namespace and qualified name. The DOM handles namespaces exactly same way.

The numerous finder methods in XDNode are not all namespace-aware, although many have been augmented for this use. Fortunately, XPATHContainer, the handler of xpath within the server, fully supports namespaces right away.

Functions using namespaces refer to them through a namespace provider, which maintains maps of alias to URI.



Use of namespaces in the XPATH function is described in the Runtime Functions documentation.

XDNode and XDDom

Background

XML is represented in memory by trees of values and their characteristics, including the name by which the value is known and attributes associated with that value. The path to the value can uniquely select that value from amongst any others in the tree, or it can identify sets of related values.

The actual representation of the XML tree in memory is constructed by a parser that accepts a serialized tree and converts it into the memory representation. Nothing in the XML specification requires that a particular shape for the tree be implemented; usually the tree shape is dependent upon its use. In iWay core, the tree is represented by related elements called XDNodes, each containing one name and possibly one value, along with any associated attributes. The XDNodes are consistent, and differ only in the chaining of their relationships (i.e. is there a child?) and the values and attributes stored in the node.

An alternative implementation of nodes, which generally goes by the name DOM, represents the tree in a different manner. In the DOM representation nodes of different types are constructed differently, and each component of the tree is represented by a related node of the proper type.

The XDNode is particularly simple for use in Java programs, however over time the DOM representation has become a standard. It is used as a common form for passing an XML tree between unrelated components. Without the ability to directly pass a DOM tree, components that need this canonical form must parse a serialized XML document themselves. This leads to slower operation and usually increased memory use as the XDNode tree must be flattened and reparsed to obtain the needed DOM. In some cases the reverse operation must be performed when the generalized component has completed its work and the XDNode tree needs to be constructed to represent that result.

To reduce this overhead, iWay Software developed several components that apply standard DOM operations directly to XDNode trees. The most common of these is the xpath and XSLT operation. However, in some situation, it makes more sense to call the standard implementation of a standard rather than re-implement it with XDNode. For example, it is better to call the XML Digital Signature implementation bundled with JDK 1.6 rather than re-implement it from scratch. What is needed is a way to call the externally developed component based on the DOM without incurring all the overhead.

Accordingly, iWay provides a DOM interface to the XDNode tree. DOM itself is simply an interface and does not mandate any specific memory object structure. Every effort was made to insure that XDNode performance was not compromised, and that when operating as a DOM the performance be at least as good as that of Java standard DOM. The result of this is the XDDom facility.

XDDom supports a large portion of the DOM specification both on the read and write side. Technically, The iWay DOM implements most DOM Level3 methods on DOM Level1 interfaces (i.e. no Level3 features like load and save). The few methods that were not implemented will throw when called.. In practice, this is rarely a limitation since Level 2 and 3 classes are rarely used. Performance is much better than the standard DOM implementation because XDDom assumes the caller is always right.

Use of external DOM components (such as XSLT and XMLDSig) has demonstrated that XDDom does meet the DOM specification. In the future, it's possible that situations may arise that require implementation of some portion of the interface that has not yet been covered.

Running an XSLT Transformation

Extensible Stylesheet Language Transformation (XSLT) is a standard transformation facility for XML documents. iWay Service Manager (iSM) provides an internal ability to run XSLT transformations in an efficient manner. The transformations accept the input as a string or an XDNode as the root of the tree to be transformed, and return either a string or an XDNode representing the result of the transformation.

All three functions accept an XSLT name. This is interpreted as either the name of a transformation defined and published for the iSM instance or as the path to the XSLT file. If the path is relative, then it is assumed to be relative to the <*iway_home*>\config directory.

Note: A transformation might not be available until iSM restarts. A defined transformation will be compiled and cached as a thread-safe Templates object at iSM startup. Other named XSLT templates will be compiled and cached on first use.

The transformation calls can be made in any XDExitBase component, which include services (agent nodes), preparsers, preemitters, reviewers, xalog exits, and so on.

The following functions are available:

- **String xsltTransformToString(String xsltName, String input)** - generates an exception

  The input is an XML document in flattened form, and the result is an XML document in flattened form.

- **XDNode xsltTransformToXML(String xsltName, String input)** - generates an exception

The input is an XML document in flattened form. The output is an XDNode-based tree suitable to be directly inserted into an output XDDocument.

- **XDNode xsltTransformToXML(String xsltName, XDNode input)** - generates an exception

  The input is an XML tree in XDNode-based tree form. The output is an XDNode-based tree suitable to be directly inserted into an output XDDocument. The input tree can simply be taken from the input XDDocument of the component using getRoot() or can be constructed using XDNode facilities.

DOM Implementation

Once you have an XDNode, you can ask for the DOM Node corresponding to this XDNode. Method name conflicts makes it undesirable to implement the DOM interface directly on the XDNode, therefore the DOM Node is a separate Java Object.

```
Node domNode = xdnode.getDomNode();
```

The relationship is maintained between the XDNode and its DOM Node. Changes you make in one will be reflected in the other and vice-versa. The way we achieve this is to store all the information in the XDNode. The DOM implementation is mostly accessors inside the XDNode. The DOM Node is created on demand and is remembered. You will get back the same DOM Node if you call getDomNode() repeatedly on the same XDNode, keeping referential identity. This is efficient if you need to call multiple DOM methods because the DOM objects will only be created once and only for the parts of the tree that are actually traversed.

The XDNode API supports an optional string value per Element. In the DOM, a string value is represented as a Text or CDATASection child Node. We go through great effort to make these two views consistent. Assigning a value to an XDNode will create a Text or CDATASection child Node in the DOM depending on the isCDATA() flag. Conversely, inserting a Text or CDATASection as the first child of a DOM Node will modify the XDNode value and its isCDATA() flag. Assigning a new string value to that first child will also change the XDNode value. We call this effect " Implied Text". The XML serialization is identical for both views:

```
XDNode xdNode = new XDNode("elem");
    Node domNode = xdNode.getDomNode();
    xdNode.setValue("xdValue");
    // domNode.getNodeValue() returns "xdValue"
    domNode.setNOdeValue("domValue");
    // xdNode.getValue() returns "domValue"
    // xdNode now flattens to <elem>domValue</elem>
```

DOM Mixed Content

The DOM has some concepts that did not exist in XDNode. Examples are mixed element content and the DOM Document. These require special considerations.

In a business transaction environment where iWay products operate, mixed element content is rare. Nevertheless, some APIs like XML Digital Signatures must preserve that information. XDNode can support a rigid form of mixed content where the appearance of the nodes is dictated by the type of Node: comments before processing instructions before single string value before child Elements. The DOM is more general and allows us to intersperse all these nodes in any order.

The existing code using the XDNode API does not expect to see anything other than Elements in the children list. To preserve that contract and the fantastic performance of the get accessors, we maintain the same pointers as before: firstchild, lastchild, left and right. This doubly linked list contains child Elements exclusively. To store the mixed content we add 4 new pointers: mixedFirst, mixedLast, mixedLeft and mixedRight. This new doubly linked list contains the complete mixed content in the order it appears. The items in the new list are XDNodes or one of the 4 new subclasses: XDNodeText, XDNodeCDATA, XDNodeComment, XDNodeProcInst. New get and set accessors were added for the 4 new pointers.

```
XDNodeText xdText = new XDNodeText("some text value");
    XDNodeComment xdComment = new XDNodeComment("some comment");
    XDNodeProcInst xdPI = new XDNodeProcInst("target", "data");
    XDNodeCDATA xdCDATA = new XDNodeCDATA("some cdata value");
    XDNode xdMixedElem = new XDNode("mixedElem");
    xdMixedElem.setMixedLast(xdText);
    xdMixedElem.setMixedLast(xdComment);
    xdMixedElem.setMixedLast(xdPI);
    xdMixedElem.setMixedLast(xdCDATA);
    // BEWARE IMPLIED TEXT BREAKS REFERENTIAL IDENTITY IN THE XDNODE VIEW
    // xdMixedElem.getMixedFirst() returns an XDNodeImpliedText instance,
not XDText
    // xdMixedElem.getMixedLast() returns xdCDATA
    // xdPI.getMixedLeft() returns xdComment
    // xdPI.getMixedRight() returns xdCDATA
```

Most of the complexity of XDNode mixed content can be avoided if all you want to do is add some comments or processing instructions. The old methods are still there but they were modified to mean append to the end of the mixed content list. Beyond the obvious impact on the ordering, this also means a node maintains the comments and processing instructions inside of it, not above. New accessors have been added to make it more efficient to create the nodes.

```
xdNode.setComment("<!-- inefficient comment -->");
    xdNode.setProcessingInstruction("<?piTarget inefficient pi?>");
    xdNode.addText("some text");
    xdNode.addCdata("some cdata");
    xdNode.addComment("a comment");
    xdNode.addProcessingInstruction("piTarget", "piData");
```

If addText() is called before any other content is added, it will set the string value of the XDNode. If addText() is called when there is already a child node, it will create an XDNodeText child node instead, faithfully representing the mixed content created. Similarly, addCdata() will set the string value or create an XDNodeCDATA node depending on when it is called.

We expect the new pointers will be rarely accessed directly except internally through the DOM interface. Indeed, mixed content is mostly for comments and processing instructions which can be safely ignored (like the XDNode API is optimized to do). One place where it cannot be ignored is flattening. The serialization methods have been improved to serialize mixed content correctly without affecting performance in the normal case.

```
// flattening xdMixedElem above produces
    <mixedElem>some text value
        <!-- some comment -->
        <?target data?><![CDATA[some cdata value]]>
    </mixedElem>
```

DOM Document

In DOM, the DOM Document contains the information to recreate the XML Declaration, comments and processing instructions that precede the root Element and finally the root Element itself. The parent of the DOM root Element is the DOM Document or null if the root Element is not parented. Many standard APIs out there expect a DOM Document instead of the root Element. In XDNode, the notion of an all encompassing DOM Document is missing. We artificially recreated this notion with an XDNodeRootDoc. The weird thing about an XDNodeRootDoc is that you can add children but the child parent field remains null. Therefore the parent of the XDNode root Element is still null preserving the old XDNode contract. We do not expect users to create an XDNodeRootDoc frequently. Instead, ask for the DOM Document encompassing the XDNode. The method getDomDocument() returns its current DOM Document if it exists. The method getRootDocument() goes up the tree root, and always returns a DOM Document creating one if not found.

```
// returns the DOM Document above the root element or null if missing
    Document domDoc = xdnode.getDomDocument();
    // always returns a DOM Document, create one if missing
    Document domDoc = xdnode.getRootDocument();
```

The iWay XML parser creates an XDNodeRootDoc when it parses an XML document. It will actually contain the information preceding the root element as you expect. XDHandler.getResult() returns the root element as before. You can retrieve the XDNodeRootDoc with one of the methods above.

```
XDParser parser = new XDParser();
          parse.parseIt(xmlStringInput);
          XDNode xdNode = parser.getResult();
    XDNodeRootDoc xdRootDoc = xdNode.getRootDocument();
```

Creating a Tree from DOM

In general, converting a DOM Node to an XDNode requires the expensive serialization and reparse. In the very common case that we know the DOM Node was created by XDDom, we can be a lot faster. You can do this by casting the DOM Node and calling the getXDNode() method. Notice we can convert most DOM Nodes but not Attributes. The reason is simple. XDNodes are never attributes. They contain attributes but they cannot be attributes themselves.

```
Node domNode = ...
    Attr attrib = ...
    XDNode xdNode = ((XDDomNode)domNode).getXDNode();
    // ((XDDomNode)attrib).getXDNode() always returns null
    // If you convert back to the DOM, you get the same DOM Node
    // xdNode.getDomNode() returns domNode
```

The DOM is a set of interfaces. It cannot define constructors because it defines no classes. So how do you create a DOM tree? You call the DOM factory methods on a DOM Document. So how do you get a DOM Document to begin with? The DOM Level3 has a solution based on an environment variable but we did not implement it. We are stuck calling the XDDomDocument constructor directly. Please do not call any DOM constructor directly except for XDDomDocument. This is very, very bad style. Do this instead:

```
XDDomDocument doc = new XDDomDocument();        Element root =
doc.createElement("root");
        doc.appendChild(root);
        Attr attrib1 = doc.createAttribute("attrib1");
        attrib1.setValue("val1");
        root.setAttributeNode(attrib1);
        root.setAttribute("attrib2", "val2");
        Element elem1 = doc.createElement("elem1");
        Text text1 = doc.createTextNode("text1");
        elem1.appendChild(text1);
        root.appendChild(elem1);
        Comment comment1 = doc.createComment("comment1");
        root.appendChild(comment1);
        Text text2 = doc.createTextNode("text2");
        root.appendChild(text2);
        ProcessingInstruction procInst1 =
           doc.createProcessingInstruction("target1", "data1");
        root.appendChild(procInst1);
        Text text3 = doc.createTextNode("text3");
        root.appendChild(text3);
        CDATASection cdata1 = doc.createCDATASection("cdata1");
        root.appendChild(cdata1);
        Text text4 = doc.createTextNode("text4");
        root.appendChild(text4);
```

It might seem unnatural that DOM does not define constructors but there is another huge advantage. Most if not all libraries using the DOM are built to support any DOM implementation. This means they cannot call constructors either. They must go through the DOM factory methods. All we have to do is pass our XDDomDocument somehow and we know that all DOM objects created by that library will create DOM objects from our own DOM implementation. Because we maintain the relationship between the XDNode view and the DOM View, we know that we can create an XDNode tree, pass the DOM view to a DOM library like XSLT which modifies the tree, safely cast the result to an XDDomNode to get the XDNode back, and voila. The XSLT library manipulated the XDNode tree without knowing it.

XDNode with the DOM

In most cases, conversion between XDNodes and the equivalent objects in DOM is a straightforward matter. The following illustrates the use of XDDom with the standard API for Xpath.

XDNode with XPATH

Suppose *indoc* is an XDDocument that looks like this:

```
<root>
  <child1>
    <grandchild>123</grandchild>
  </child1>
  <child1>
    <grandchild>234</grandchild>
  </child1>
  <iw:child1 xmlns:iw='http://www.iwaysoftware.com'>
    <iw:grandchild>456</iw:grandchild>
  </iw:child1>
</root>
```

We want to use the core Java XPath API (javax.xml.xpath) to access the grandchild element in the iWay namespace. To get the value of the iw:grandchild node, we could write code like this:

```
String expr =
"//*[local-name() = 'child1' and namespace-uri() =" +
"'http://www.iwaysoftware.com']";
XPath xpath = XPathFactory.newInstance().newXPath();
Node n = indoc.getRoot().getRootDocument();
String result = xpath.evaluate(expr, n)
```

In the fourth line, we obtain the root of the XDDocument as a DOM node by calling the getRootDocument() method of the XDNode. Unlike the getDomNode() method, this ensures that the returned node is a document and, hence, that absolute XPath statements will treat this node as the root.

Suppose that instead of getting the value of the node as a String, we wanted the node itself. For this, we could change the above example like this:

```
XDDomNode xddn =
(XDDomNode) xpath.evaluate(expr, n, XPathConstants.NODE);
XDNode xdn = xddn.getDomNode();
```

If we wanted all the child1 nodes in the default namespace, we could do this:

```
expr = "//child1";
NodeList nl = (NodeList) xpath.evaluate(expr, n, XPathConstants.NODESET);
for (int j = 0; j < nl.getLength(); j++)
{
XDDomNode xddn = (XDDomNode) nl.item(i);
}
```

XDNode with XSLT

In the past, using XSLT within iSM has required flattening the input document to a string (serializing) and reparsing the result into a new tree (deserializing). With XDDom, it is possible to transform using DOMSource and DOMResult objects to avoid these steps. Suppose the file c:/myxform.xsl contains the transformation we want to apply to XDDocument xddoc:

```
Transformer t;
TransformerFactory tf = TransformerFactory.newInstance();
.newTransformer(
t = tfnew StreamSource(
new FileReader(("c:/myxform.xsl"))
);
Node inNode = xddoc.getRoot().getRootDocument();
DOMSource ds = new DOMSource(inNode);
Node outDomDoc = new XDDomDocument();
DOMResult dr = new DOMResult(outDomDoc);
t.transform(ds, dr);
XDDomNode domRoot = (XDDomNode)outDomDoc. getDocumentElement();
XDNode result = (domRoot == null) ? null : domRoot.getXDNode();
```

1. After setting a Transformer in the usual way, we get the root of our input XDDocument as a DOM Document node just as we did above in the xpath examples and use this in a DOMSource.
2. Next, we create an "empty" XDDomDocument to use as the target for our transformation result. The root element will be added as a child of this DOM Document. The XDDomDocument is set into a DOMResult.
3. We apply the transform, using the DOMSource and DOMResult.
4. To retrieve the result, we first need to access the DOM root element, then get its underlying XDNode. The transform might have returned without creating a root element, hence the check for null.

Other APIs that use DOMSource and DOMResult should work similarly.

XDOrgData

An XDOrgData holds the "original" data that the business agent is processing. This object is of interest only to protocol component writers. There is a built-in XDOrgData object in the worker, and you can load it and reference it as needed. The object handles several types of messages, so that you can store your incoming message in whatever form is most convenient - byte[], String, or tree. Usually, your protocol worker uses only the store() method; the retrieval methods are used by the internal system to operate on the data itself.

XDOrgDate Methods

Method	Description
getSafeStoreKey	Returns the stored safe store key.
setObject	Adds an arbitrary object to the persistent object store. Use this to preserve information across business agent calls.

Method	Description
setSafeStoreKey	Stores a string that can be used as a key to a safe store holding this message. Used only for protocols for which safe store applies.
store	Stores the data into the object. Can accept a byte[], String, or XDNode tree.

XDSpecReg

Special registers are used to hold information about the document process, and the environment in which it is being handled. For example, the time of arrival can be checked in an exit by looking up a designated register.

A more complete description of the registers and the API that applies to them is found in *iWay API Reference* on page 147.

XDExitBase

An XDExitBase is the base object for all message processing exits.

This class provides parameter parsing, metadata reporting, and other common services for all exits. This ensures that all exits operate in a consistent manner. Key services available to your exit through the XDExitBase are shown in the following table:

XDExitBase Methods

Method	Description
getObject	Retrieves an object from the object store.
isDebug	Returns true if the business agent is running a listener in debug mode.
isQA	Returns true if the business agent is running a listener in QA mode.
setObject	Adds an arbitrary object to the persistent object store. Use this to preserve information across business agent calls.
systemProperty	Accepts a String as a property name and returns the setting in the engine dictionary.

Method	Description
trace	Adds a line to the trace output. The trace line is routed as appropriate to the trace level.

Chapter 6

iWay API Reference

Your exits and components can call upon the services of iWay Service Manager to obtain information and to perform actions. *Programming Objects* on page 123 discusses the objects that iWay Service Manager uses, for example, documents and trees. This chapter describes how to manipulate these objects and how to take other actions that facilitate your application purpose.

In this chapter:

Tracing

Tracing is the key to diagnosing problems and thus to application reliability. iWay Service Manager provides a full complement of tracing services, oriented to diagnostic analysis of the running system.

Experience teaches that tracing can also be a major source of performance degradation. This comes primarily from the string processing needed to form meaningful trace messages. Before you build a message that has several parts, check the following method to determine if the trace message is emitted

```
logger.is TraceLevel Enabled()
```

where:

TraceLevel

Is the specified trace level.

If the trace message is not emitted, then there is little purpose to construct it. Too much tracing is as bad as too little. Messages with several values to assist the debugger are easier to work with than several small messages that may get separated on a trace display.

Line feeds and tabs can be included in messages to make them easier to understand. Never start or end a trace message with a tab or line feed.

Tracing in iSM-level servers has been changed from prior releases, although exits written to the older trace() method calls will continue to function. In iSM, trace calls mirror the calls used in other familiar tracing systems, although with distinctions.

In most Java applications, trace levels are hierarchical. Any specific level sets all "lower" levels. In a server, this is expected to run for extended periods of time. The plethora of trace messages that would be generated in such a case when attempting to isolate problems could be enormous. To this end, iWay uses a mechanism based on trace categories. As an example, setting the PARSE category on traces only parsing operations and does not imply that all debugging levels will be consequently set.

When exits are entered, the trace has been enabled for the desired settings. All exits extend a standard base, and this base is a logger. As a result, you do not need to specifically refer to a logger when making trace calls. The protected field logger is available should you wish to use it.

```
debug("this is a debug message")
```

and

```
logger.debug("this is a debug message")
```

are equivalent.

When coding a listener or emitter, you must specifically refer to the logger in your calls, as these objects do not themselves act as loggers.

All levels provide two calls:

Call	Description
level(String)	Issues a trace message at that level in the context of the logger.
is*Level*Enabled	Returns true if the trace level is enabled.

For example, *debug(String)* and *isDebugEnabled()* are defined for the debug level.

At the error level, the additional call is provided:

Call	Description
error(String,Throwable)	Traces an error message with a stack trace

Trace levels supported are:

- **BIZERROR** level shows error messages associated with processing a document. Rule violations are BIZERROR level. You should not use this level to report problems such as a missing jar or things relating to the environment.
- **DATA** level shows the incoming and outgoing documents as they pass to and from the protocol channel.
- **DEBUG** level reports data that is helpful for debugging situations. It shows logic that follows the path of a document. You should not trace too much at DEBUG level.
- **DEEP** is used for detailed logic tracing. Trace whatever you need to understand why the document processed as it did. Stack traces are reported by the system in DEEP level.
- **ERROR** level shows error messages relating to operation of the adapter. This level is always displayed.
- **INFO** provides messages you think the user always needs to see. The server traces almost no messages at INFO level.
- **PARSE** displays the internal operations of the many parsers in the server. For example, character by character lexical analysis for the function compiler or the XPATH compiler is displayed at this level.
- **TREE** shows the document as it enters and leaves the system in XML form. Intermediate processing as a document evolves is done at TREE level. iWay makes every attempt to avoid tracing secure information in trees being traced. This includes attempting to locate elements with password values and replacing those values with a fixed number of asterisks.
- **WARN** shows important information pertaining to some situation, but that does not fall into the category of an error.

In addition to control using the consoles, any trace level can be set on or off using the command window when the server is running as an executable process. The command

```
set level on|off
```

where the level specifies the level to be controlled, will turn on or off tracing.

Use `set display` to see what levels are enabled.

Interacting With Log4J

Some components interact with externally-developed components that use the Log4J tracing system. The iWay server can route such traces to its own logs, using a Log4J configuration file as described in *Interacting With Log4J* on page 227. Log4J is classed by iWay as an external log system, and is turned on in the server by setting the **external** level.

Special Registers

Special registers are named fields of information carried through the system and available to all components and exits. Special registers are created by:

- iWay Service Manager initialization to reflect configuration information. This includes registers defined by the configurator (console). Defining special registers as part of the configuration is one of the mechanisms that iWay Service Manager provides to make it possible to deploy applications and process flows to various configurations.
- Document intake, reflecting the source of the document and dispatching information.
- Any exit that calls special register functions to store values for use later.

Special register values can be substituted into documents during transformations by use of the SREG() built-in function offered in the iWay Transformer and iWay Designer. Additionally, any configuration parameter can accept the SREG() specification. Special registers are also available as global variables to any process flow. The special service EvalWalk substitutes values in documents while processing in the channel (workflow).

If you use a special register in your own code, be careful that the register is available at the time you request it. For example, a document-specific register such as SREG(msgid) is not available until the document arrives to be processed; it would not be available to an exit init() method.

Although the specific special registers available can vary based upon protocols and situations, a complete list of these registers currently defined is displayed by the XDQAAgent(). This business agent displays the document and other information, and is available as a standard iWay Service Manager business agent when the server is installed.

Registers fall into several categories. You can reference registers in all categories, but normally should create registers only in HDR and USER categories.

Category	Description
CFG	Entered at the console. Should never be added or changed by an exit.
DOC	Applies to the document. Examples are source and protocol.
HDR	Applies to a document delivery header. For example, HTTP headers are carried as special registers; MQ Series RFH2 values are special registers. When a document is emitted, HDR registers are used to form the delivery header.
SYS	System information, such as listener name, configuration name, and so on.
USER	A variable that you might create to pass information from one part of the system to another.

The register type can be referenced by the static constant XDSpecReg object, such as XDSpecReg.HDR.

Register names must be unique, regardless of the register type. Creating a register named "acklevel" at USER level replaces one at HDR level.

Register names must conform to the requirements for XML element names. The names are further restricted for HDR registers if the register will be used to form a message header for a protocol that itself has restricted character conventions. Use of the namespace designator '.' is supported for registers in which such namespaces are meaningful. In MQ Series, the namespace character is used to group RFH2 information. Otherwise, such characters are treated simply as part of the name for lookup purposes. Thus for MQ, the section of the RFH2 from which a value was extracted will appear as the namespace.

All non-SYS registers are reset for each document.

Special Register Hierarchy

Registers are stored in a hierarchy of levels. The levels are, in order highest to lowest:

1. **Configuration** level includes registers defined in the runtime console.
2. **Protocol** (listeners, also called masters) are registers associated with the channel as a whole.
3. **Message** (also called worker) are registers associated with the message as it passes through the system. Process flow actions can set registers a message level, in which case they remain associated with the message until its exit from the system.
4. **Flow** level registers are associated with a process flow, but do not exist beyond the flow. To pass a register past the flow, you must set the register as Message level.

5. **Thread** level is defined for the thread of the process flow (edge) on which it is set.

A process flow cannot set registers into the configuration or protocol levels, as this affects operations outside of the scope of the flow.

When searching for a register, the "nearest" special register repository is searched first, proceeding upward until the register is found. So in a flow, the Thread level is searched first, then upward through thread levels until the flow level is reached. Then upward until the register is found or the register is determined not to be stored anywhere; in this case the default is returned.

Emit operations defined in a flow inherit the existing registers as the emit object is encountered. Changes to registers subsequent to executing the emit object do not affect the emit.

Methods for Special Registers

Special Registers are classified into namespaces. Namespaces are simply groups of registers categorized by the first token of their name, which terminates in a dot. The namespace is considered part of the name and must be used for searches, etc. Many components can be configured to store their registers or take their registers from specific namespaces. For example, an nHTTP emit will attempt to make HDR registers into HTTP headers, and will store headers into the HDR portion of the registry hierarchy. Without a namespace the headers would become mixed with other headers in the system, and it would be impossible to separate those dealing with one nHTTP emit from another.

A special process flow agent, **XDSregNamespaceAgent**, provides namespace support services such as copying, renaming, and deleting.

Name	Description
lookupSpecialRegister(name,default)	Returns the value of the special register or the default if the register is not defined.
lookupSpecialRegisterObject	Special registers are actually stored as objects, which in turn can hold values of different types, such as Integer or Tree. You can use this method to retrieve the actual object, and then use the XDSpecReg object methods to extract specific information.

Name	Description
storeSpecialRegister(name, value)	Stores the value as a special register under the name key at the USER level. Storing null eliminates the register.

storeSpecialRegister(name, value, type)	Stores a special register of the specified type. For example,
	`storeSpecialRegister ("john","121",XDSpecReg.HDR);`
	Appears in an HTTP post as john=121.
	`String x = lookupSpecialRegister("john","000"); logger.debug("john is "+x);`
	Prints john is 121.

deleteSpecialregister(name)	Deletes the special register.
getValue()	Returns the value from the special register object. This is returned as a String and is the usual result of getting the value via lookupSpecialRegister().
getObject()	Returns the object holding the value. This can be any object of meaning to an application. The object has to be set on a new special register or assigned to the register via the setObject() method.
setObject()	Sets the object stored in the register. The object reference is marked as volatile and can be locked by synchronization methods. For example, to set its own object, the application can lock on the register, get the current object, update that object, and set the object back.

Although the list of special registers is updated as needs arise, you can count on at least the following special registers that can be accessed from any exit. All special register names are lowercase.

Name	Description
<property name>	Each property associated with a JMS queue is stored in a special register under the name of the property. This includes user properties.
backoutcount	Count of the number of times the message has previously been returned by an MQQueue.get() call as part of a unit of work, and subsequently backed out. MQ only.
config	For example, if you are running the base configuration, this is "base".
correlid	For queue-based listeners, this is the correlation ID. If the value is defined as allowing binary (for example, IBM MQ Series) it may be the base64(value) representation.
destination	Reply queue name.
eos	'1' if a streamer is running and has read end of stream, '0' if not at end of stream, and undefined if not streaming.
expiry	Message expiry in .1 seconds. MQ only.
format	Message format. MQ only.
from	Incoming email sender.
groupid	For queue-based listeners that support groups, this is the group ID.
ip	For TCP-based listeners, this is the dot-form IP address of the client.
msgid	For queue-based listeners, this is the message ID. If the msgid is defined as allowing binary (for example, IBM MQ Series) it may be the base64(value) representation.
name	Name of this server. Names are set by the –n startup parameter.
outmsgid	MQ Series only. Sets the outbound msgid to be emitted.
persistence	"queue", "persistent", or "non-persistent" to describe the message persistence. MQ, JMS, and SONIC only.

Name	Description
priority	Message priority. JMS, SONIC, and MQ only.
putapplicationname	Name of the writing application. MQ only.
protocol	Name of the protocol of the listener that received this document.
source	Origin of the document. For files, this is the file name, for TCP-based listeners, this is the host name of the client, for queue-based listeners, this is the queue name.
subject	Subject of incoming email.
tid	Transaction ID from this document. This is set to a unique value regardless of whether or not the document is operating in a controlled transaction.
type	Queue message type. For example, request, reply, or report.
userid	Identifier of the user that put this message. MQ only.

The configuration console shows special registers appropriate to the listener type being configured.

You can learn which special registers are defined and their current values by executing the XDQAAgent or using the Debug service of a process flow. All special registers are dumped to the output location along with the current document.

Special Register Callbacks

Some registers may return values that are "expensive" in terms of system resource such as time or memory. It may be desirable to not actually load the value unless the register is actually checked by an _sreg() call or programmed lookup. An example is a reverse DNS lookup on a connected socket.

To accommodate this situation, register callbacks can be used. A register callback is simply a method that is called when the value of the register is looked up. So, in our example above, the callback method would make the reverse DNS call only if the user actually checks the value.

To establish a callback, first construct the special register. Then call the following:

```
void setCallback(com.ibi.common.IXSpecRegCallback callback, Object savedO)
```

This passes in the method reference and an arbitrary object that will be passed to the callback method when the callback is invoked. This will be the actual reference and not a clone; the application is responsible for protecting the value of the object.

When the register is looked up by the program, the callback is invoked. The callback is defined in the IXSpecregCallback interface. The method getSregValue is called. The method is passed to the register object, the name of the register (if known), the current value, and the object stored when the callback was established. The callback must return the value of the register to be returned by the original lookup.

```
public Object getSregValue(XDSpecReg sreg, String name, Object currentVal,
Object savedO)
{
  return  "the answer"
}
```

Pooling

iWay Service Manager offers a pooling service to simplify the pooling of objects such as connections. The basis of the pooling facility is the pool set, a facility maintained by the engine to house and monitor individual pools. Pools are shared at the system level, and in JCA situations they can be shared at the connection factory level—for example, across instances of iWay Service Manager.

There is one distinct, critical requirement for an object to be pooled. It cannot maintain stateful information between uses. A pooled object cannot rely on the information from any prior invocation or expect to be used by the same consumer during any future invocation. Pooled objects do not necessarily have to be thread safe. They just need to clean up their class variables before being returned to the pool.

Each pool (XDPool) in the set is built to house one type of item pool. Each item pooled is represented in a user-written class that extends XDPoolItem. The XDPool provides methods to set, request, and delete items from the pool. The XDPoolItem is required to implement the term() method to close down the item; term() is called by the engine for any XDPoolItem in a pool when the engine shuts down, allowing your pooling class to close. The actual XDPoolItem being pooled must represent the resource appropriately. For example, the MQ Connection pool maintains a pool by a name that relates to the thread ID, because in MQ the queue manager is thread related. The users of the pool (the XDMQEmitter) moves the item from the available queue manager connections pool to the thread pool, keyed by TID, during the time a transaction is in progress.

Each pool is given a name, by which you request the pool from the manager. If no pool exists, an empty one is created for you. You do not need to take any special action to establish the pool itself. Each pooled item is identified by its key, which can be any String. If you request a pooled item for a key that has not yet been pooled, you get a null return. At this point, you create a pool item of the key and add it to the pool.

Once you have the item, you can use the pooled object that it represents. If this is a new object, you need to set it up.

In this example, the pool holds some object of meaning (we call it MYObject) to the application.

This example pool also shows how to keep a pooled counter. One common use of a pool is to keep a named counter with no object attached.

Be careful of the scope of synchronization; the example shows how to minimize long duration locks on a central resource to avoid serializing the server.

```
MyObject poolUserMethod()
{
    XDPool pool;         // the pool itself
    MyPoolItem pitem;    // the specific ABC item
    int counter;
    String key = "ABC";  // id of item in the pool - first get pool by object type
    pool = worker.getPool("MYOBJECT");
    MyObject myobj;   // the copy I will work on - now use it or set it up
    synchronized (pool)
    {
        pitem = (MyPoolItem)pool.getItem(key);  // get or add
        if (pitem==null) // do I need a new one?
        {
            pitem = new MyPoolItem();
            pool.addItem(key,pitem);    // add the new object to the pool
        }
    }
    // now use the pooled item
    synchronized (pitem)
    {
        myobj = pitem.get();
        if (myobj != null)
        {
            debug("MyObject "+key+" recovered from pool");
         }
           else
           {
            counter=pitem.getHandle();  // just a serializer to show how
            myobj = new MyObject(key);  // make the object to pool
            pitem.set(myobj);
            debug("MyObject "+key+" created and added to pool");
           }
    }
           return myobj;  // ready for caller's use
   }
```

```
// pool for MyObject instances, keyed by String
class MyPoolItem extends XDPoolItem
{
    boolean badItem = false;  // set if we can't connect
    int count=0;
    MyObject myobj;  // some object I am pooling
    MyPoolItem()
    {
    }
    public void term()
    {
        if (myobj != null)
        {
            try
            {
                myobj.term(); // call my thing to shut it down
            }
            catch (Exception e)
            {
            }
        }
        myobj=null;
    }
    MyObject get()  // gets the object from the pool
    {
        return myobj;
    }
    void set(MyObject myobj) // adds the object to the pool
    {
        this.myobj = myobj;
    }
    int getHandle()    // just to show how to keep a pooled counter
    {
        return ++count;
    } // end MyPoolItem
} // end poolUserMethod
```

Getting Server Information

Your exit can find out the state of the server by using the methods shown here.

Name	Description
getTID	Within business processing agents, returns the transaction ID assigned to this workflow.
isDebug	Returns true if debug tracing is set on.
isQA	Returns true if QA mode is set on.

An agent, XDChanInfoAgent, is available for use in process flows. It returns much of the same information as is returned by the **info** and other commands issued from the command shell.

In addition to the specific method calls, server information is available in read-only special registers. These are shown on the Register Settings page of the console. Among the registers are:

Name	Description
iwayversion	The version of the server.
iwayhome	The root location where the server is installed.
iwayconfig	The name of the configuration currently running. The console may show the name of the master configuration, but within any configuration, this register is set to the current configuration name.
iwayworkdir	The path to the configuration, through iwayhome.
iway.pid	The detectable process ID of the server.
iway.serverip	The IP address of the server.
iway.serverhost	The host name of the server.
iway.serverfullhost	Fully qualified, host name of the server, extended with the appropriate domain information as needed.
IBSE-port	Port on which SOAP messages are managed.
locale	The encoding in IANA form.

To aid analysis, the thread status was added to the report. A sample report is shown here.

```
?xml version="1.0" encoding="ISO-8859-1 " ?>
<info>
  <channels>
    <master name="internal" state="active" type="internal" completed="0"
failed="0" active="0" available="1">
      <user mean="0.0" variance="0.0" ehrlang="1.0"/>
      <cpu mean="0.0" variance="0.0" ehrlang="1.0"/>
      <threads group="internal">
        <thread name="W.internal.1" state="timed_waiting"/>
      </threads>
    </master>
    <master name="file1" state="active" type="file" completed="0"
failed="0" active="1" available="3">
      <user mean="0.0" variance="0.0" ehrlang="1.0"/>
      <cpu mean="0.0" variance="0.0" ehrlang="1.0"/>
      <threads group="file1">
        <thread name="W.file1.1" state="runnable"/>
        <thread name="W.file1.2" state="timed_waiting"/>
        <thread name="W.file1.3" state="timed_waiting"/>
      </threads>
    </master>
    <master name="mqin" state="active" type="mq" completed="0" failed="0"
active="0" available="1">
      <user mean="0.0" variance="0.0" ehrlang="1.0"/>
      <cpu mean="0.0" variance="0.0" ehrlang="1.0"/>
      <threads group="mqin">
        <thread name="W.mqin.1" state="runnable"/>
      </threads>
    </master>
  </channels>
  <internalqs>
    <queue name="internal1" size="0"/>
  </internalqs>
</info>
```

QA Mode

The server runs in either QA or standard mode. You set the mode in the systems properties page of the console. QA mode adds some Quality Assurance services to the server to assist in validating your software. In addition to some extra tracing, the most visible aspect of QA mode is that the dates in emitted documents, where recognized by the server, are replaced with a constant string. This facilitates regression testing by simplifying the comparing of outputs over time.

Your own software can check for QA mode and take any special action appropriate to QA and regression testing. The method

```
isQA()
```

returns true for QA mode. You can use the QA check in any exit.

Evaluating Expressions

iWay Service Manager supports built-in functions for parameter entry. These functions perform some expressible operation on their operands, and are evaluated to a string result, which in turn becomes the parameter. Expressions are of two classes, those that can be evaluated at init() time and those that must be evaluated at message execution time. An expression can only be evaluated at init() time if all of its operands depend only on values not related to the message being processed. An example of such an expression is the FILE(<filename>) function, if the name of the file is constant, and is not itself taken from a special register or from the message.

Init time functions are automatically evaluated by the time that your init() function is called in an exit. You simply get the result of the evaluation. For execution time evaluation, or in a protocol listener or emitter, you must evaluate the expression yourself. The engine provides an evaluation service to do the actual evaluation.

The complete list of built-in functions, which can be combined, are explained in iWay Service Manager configuration documentation. Some examples of built-in functions are shown in the following table.

Function	Returns	Description
_ALL	True/False	Modifier for COND.
_ANY	True/False	Modifier for COND.
_ATTCNT()	Integer	Number of attachments.
_COND	True/False	Evaluates expression for truth.
_CONTAINS(string, string2)	True/False	If substring is contained.
_COUNT(parm1)	Integer	Number of elements in expression.
_ENDSWITH(string1, string2)	True/False	Does parameter end with value?
_FILE	String	Gets contents of a file.
_FLATOF()	String	Returns flat value of the document.
_ISERROR()	True/False	Is document in error state?
_ISFLAT()	True/False	Is document flat?

Function	Returns	Description
_ISXML()	True/False	Is document XML?
_LDAP	String	Looks up value in an LDAP.
_LENGTH(string1)	Integer	Returns value length of the parameter.
_SREG	String	Returns value of a special register.
_STARTSWITH(string1, string2)	True/False	Does parameter start with value?
_SUBSTR(string1, startindex [,endindex])	String	Returns substring of parameters.
_XPATH	String	Gets value(s) from an XML document.

The evaluate() service evaluates any expression of any built-in functions described in iWay Service Manager configuration documentation.

You can pass any String value to the evaluate() service. The method returns a String of the evaluation, which your program should use as the value of the parameter. Evaluation is done right to left, inside to outside. So an expression of

```
FILE(SREG(XPATH(/root/childone)))
```

returns the contents of the file whose name is in a special register the name of which is in the incoming document. This example is for illustration, and such complexity is probably a poor idea in a real application.

To evaluate, use the static method evaluate()

```
String XDUtil.evaluate(String parm, XDDocument doc, ISpecregMgr ism)
```

where:

parm

Is the expression to be evaluated. The parm must be a properly formed expression. Null is returned if the expression cannot be evaluated. The evaluation of a literal is the literal itself.

doc

Is the XDDocument that is to be the subject of any XPATH built-in function. Even if you know that there are no XPATH function calls, pass the current document if you have one. In listeners or preparsers, this must be null, as no document yet exists.

ism

A special register manager. You can obtain the local register manager by calling getSRM() in your context.

logger

Standard context logger. You can object the local logger by calling getLogger().

For example:

```
File c:\x.txt in Windows contains the value "ABC".
  String r = XDUtil.evaluate("COND(FILE(c:\\x.txt),EQ,DEF)",null,this);
  logger.debug("val is "+r);  // prints "false".
```

User Functions

Special-purpose functions can be written and added to the system. They must be in the classpath and properly registered. The actual writing of functions is explained in *Writing Management Components* on page 73.

Exit Attributes

Exits when configured as run time instances have attributes such as the name and (in most cases) a comment that is either the comment entered on the configuration screen or the default as defined in the metadata for the exit (getDesc()). Your program can obtain this information through the getAttributes() method.

This following code returns a Map of the attributes that can be queried by the standard MAP collections class methods. For example, the name assigned when the instance is configured can be obtained by:

```
Map attr = getAttributes();
String name = (String) attr.get("name");
```

Storing Objects

Each master maintains its own object store, which is simply a map relating keys to objects. Once stored, the objects remain until explicitly deleted. This provides a way to pass information between exits and even between workflows. Using the object store, you can implement stateful workflows. If your input provides a correlation ID (for example, JMS or MQ) this often makes a good key for maintaining state information.

Name	Description
getObject	Retrieves an object stored under a key.

Name	Description
setObject	Adds an arbitrary object to the object store under a designated key. Setting the null object removes the key from the object store.

Service Functions

iWay Service Manager provides programmers with several service functions to simplify programming. These are static in the XD object and the xdutils package. All are documented in Javadoc.

Name	Description
XD.entityExpand	Given a string with XML parsible entities, such as &, replaces the entities with the actual character.
XD.entityReplace	Replaces characters in a string with their XML entity representation.
XD.normalizePath	Given a file path, converts the backslashes to Java standards, and ends the path with a slash.
XDServices String operations	Normalizing partial text strings, converts numbers with abbreviations such as 4kb to integers.
XDServices file operations	Writes and copies files.
XDServices regular expressions	Converts between DOS wildcard expressions and standard regular expressions.

Note: There are also a variety of methods for constructing date strings as used by the internal components. These are explained in the Javadoc for the XD and XDTime objects.

Creating Files With Unique Names

A frequent need arises to create files with unique names matching some pattern. Although Java provides some services for creating unique file names, the engine provides additional services to create file names meeting specific design patterns.

Patterns supported include serial number and time stamp. Each is represented in the pattern by a character: # for serial number and * for time stamp. The serial number created in response to the # is sized according to the number of # characters with rollover to 1. For example, MYFIL###.OUT generates unique files from MYFIL001.OUT to MYFILE999.OUT.

To create a unique file name, you first create a pattern object XDUniqueFile with the pattern you desire:

```
void XDUniqueFile(String directory, String pattern)
```

and then call nextFile() on the pattern:

```
String dir = "SREG(iwayhome)/myarea"; // this might come from parameters
dir = XDUtils.evaluate(dir,null,this);
XDUniqueFile uf = new XDUniqueFile(dir,"F###.TXT");
File useFile = uf.nextFile();
```

The supported meta characters are listed below.

Name	Description
*	Current timestamp in milliseconds.
#	Serializing number, modulus the antilog of the number of consecutive # characters. For example, AB## becomes AB00..AB99, and then recycles to AB00. The number is stored on a disk, and is continuous for any specific pattern across starts.
^	Similar to the # character, but not stored on a disk. This restarts at 0 each time the system starts, and applies irrespective of any pattern. Faster than # as it avoids a disk access.

Safe Store Facility

Several protocols require the use of a safe store, a guaranteed storage of messages before the next step of the protocol can proceed. Although the details of a safe store mechanism may vary among protocols and the medium chosen to hold the safe store, the steps of a safe store are generally common. To this end, the server provides a common interface for safe store mechanisms, com.ibi.common.IXDSafeStore. A default implementation of the safe store mechanism is provided by the com.ibi.edaqm.XDFileSafestore class, which supports a constructor of the path in a file system to be used for the safe store.

Supported methods are:

Method	Description
add	Adds a record to the safe store. An arbitrary key is required.
getMessage	Returns the message associated with a given key.
iterator	Obtains an iterator for the records in the safe store. This is usually required during recovery operations to facilitate recovery of the safe stored information. The iterator next() method returns the key of the next record in the safe store.
remove	Removes a keyed record from the safe store.

Details of the XDISafeStore interface and the FileSafestore implementation are contained in the provided Javadoc.

Converting Between JDOM and XD Trees

iWay Service Manager works internally with XD trees, described earlier. Some external services you may want to use require JDOM trees. A utility class is available to convert between the two. To use it, make an instance of XDJDOMUtilities in the com.ibi.edaqm package. The methods exposed are:

```
public Document converXDtoJDOM(XDNode root)
```

Converts the tree rooted in the passed node to a JDOM document.

```
public XDNode convertJDOMtoXD(Document doc)
```

Converts the JDOM document to an XDNode tree.

This example makes a JDOM, converts it to an XDNode tree, and then goes the other way.

```
try
{
   SAXBuilder builder = new SAXBuilder();
   Document doc = builder.build(in);   // make a JDOM doc
   in.close();
   XMLOutputter outputter = new XMLOutputter();
   FileOutputStream out = new FileOutputStream(fileParsedJDOM);
   outputter.output(doc, out);   // print JDOM doc
   XDJDOMUtilities jdu = new XDJDOMUtilities();
   XDNode   xnode = jdu.convertJDOMtoXD(doc);
   out = new FileOutputStream(fileJDOMToXD);
   out.write(XD.doflat(xnode,true).getBytes());  // print XD tree
   doc = jdu.convertXDtoJDOM(xnode);   // back to JDOM
   out = new FileOutputStream(fileXDToJDOM);
   outputter.output(doc, out);   // print JDOM doc
}
catch (Exception e)
{
   logger.debug("Exception received: "+ e); e.printStackTrace();
}
```

Multithread Considerations

The server runs channels and process flows in full multithread and multiprocessor modes to the degree configured and for which hardware is available. The following list shows the normal rules of multithread programming that are applied:

- Protecting resources from multiple updates.
- Protecting resources from references during updates.

Java language provides facilities to assist in this, and many books are available to provide advice.

Testing a multithread program is often difficult, as it is hard to simulate the many possible interactions between threads and with the hardware or operating system that result in "scrambling" of the order of execution. The bugs that arise are often referred to, jokingly, as Heisenbugs, as they seem to vanish when an attempt is made to search for them. The server provides a small utility to assist in the search for such tasks.

The com.ibi.xdutils package provides a simple method to cause random, distributed thread yields. It is discussed here as it may assist programmers in testing software and in developing their own, more sophisticated, and application-specific approaches.

The assert statement must be added to the code as shown below:

```
assert(com.ibi.xdutils.XDAssert.heisenbug())
```

This causes a random thread yield by issuing a Sleep for a short, random millisecond period. To enable this code, add -ea (-enableassertions) to the startup command for Java in the server start script.

If assertions are not enabled, the assert statement is ignored by the Java Virtual machine. The asserts can be left in the application code and enabled as needed without causing a significant production overhead.

Problems caused by bugs in multithread handling often become apparent by scrambling the order of the thread execution.

Chapter 7

iWay Service Manager Rules System

Document validation enables any document to be validated against sets of rules specified on a per-document basis. The rules are encoded in a rule file that is addressed using the system document dictionary.

Rules apply to document nodes in the XML tree, and optionally, to their children. Built-in rules can be specified in any combination, and specialized rules can be coded in Java and loaded by the engine as required. This chapter discusses how to encode a rules file, how to use the built-in rules, and how to code specialized rules if needed. iWay Software provides a complete set of built-in rules as needed to validate HIPAA documents.

The last section of this document describes how to write rules in Java for special situations.

In this chapter:

Rules File

Rules validation picks up where schema validation leaves off, providing context and content analysis. The content of the document can be examined against assertions, and values can be examined in the light of other values in the document. Services exist to validate content against tables and databases, and exits can be written to perform specialized analysis if needed.

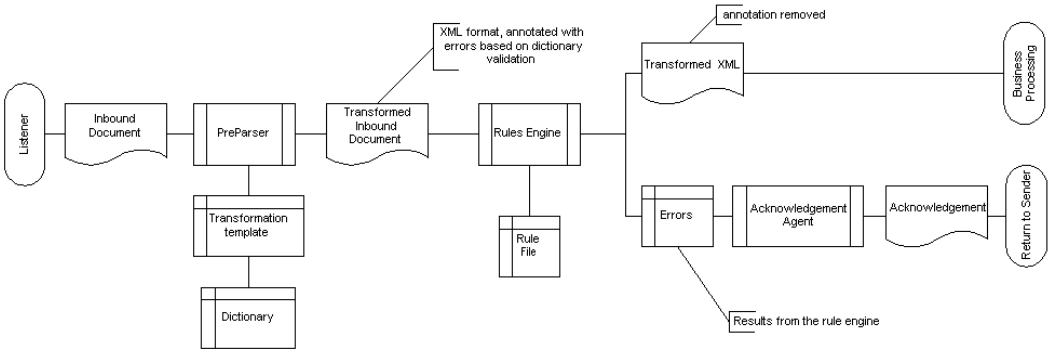
The rules file is an XML document. One file should exist for each document to be validated. The outer tag should be the document name, and under this tag are *rule* tags which may be enclosed within *using* tags. For example, for a HIPAA835, the file (very shortened) might look like this:

```
<HIPAA835>
<using>
  <rule tag="_835.DTM" method="checkDTM" code="405" cd="R020305,C0203,P0506"/>
  <rule tag="_2100.DTM" method="checkDTM" code="036,050,232,233"/>
</using>
</HIPAA835>
```

The rules are expressed in an XML document tied into the iWay Service Manager dictionary through <validation> tags, which associate one or more rule sets with the specific document entry. The outer tag of the rules document should be explanatory, describing the type of document, but the actual tag is ignored. The rule document itself is a structure containing specific rule applications.

All attributes of rules must be specified in lowercase.

The following diagram illustrates the document flow and how the rules are applied in that flow.



The document entry is the outer tag of the rules file. The name "document" is arbitrary and can be replaced with any meaningful, XML-legal name.

<*document*>**entry**

Attribute	Description
ackagent	Java program class to be called to construct the acknowledgement. This is a standard iWay Service Manager business process agent. For more information, see the *iWay Service Manager Component Reference Guide*.

<*using*>**entry**

Attribute	Description
class	Java program class containing all <rule>s within the section, unless overridden by a class= attribute in the <rule> entry itself.
other	Any unrecognized attribute is passed to each rule in the rule attributes. For example, to apply the "radix=','" attribute to all rules, it can be specified here rather than on each rule. Naturally, rules that do not use the "radix" attribute would ignore it.

The rule *tag* and *method* attributes are required. The remaining attributes are rule-specific and their inclusion is based on the rule itself. The Validator uses the required tags to identify the rule in question and to identify the node or nodes of the document to which it applies. Common <rule> tags are:

<*rule*>**entry**

Attribute	Description
class	Rule class to which this rule belongs. This corresponds to a Java object class, and each rule is a method of the class. If this is omitted, the class from the enclosing USING tag is used.
method	Specific rule.
name	Rule identification; should be a unique name. This is used in trace messages to specify which rule caused a violation. If omitted, no unique identification can be given.
stag	For HIPAA documents, this is a specification subsection tag. For more information, see the *iWay Format Adapter for HIPAA* manual.
tag	Names the right-most parts of the tag to which this rule applies. The rule applies to any node of the document that meets the tag criteria. For example, "DTM" would cause this rule to be applied to every DTM in the incoming document. "X.DTM" applies to all DTM parts prefixed by X. Tags are case sensitive. If omitted, stag must be used.
usage	Specify usage="M" (mandatory) to specify that there must be a value in the identified node. This check is applied before the actual rule logic is executed.

The rule document is located by the <validation> tag value in the dictionary *system* section, and is identified with the specific document in its <document> entry. Rules validation is performed for document input (<in_validation>) and output (<out_validation>). If several tags are found in the document description, each rules validation is performed in the order in which the tags are found.

A section of the dictionary that illustrates this is:

```
<system>
   <validation>
       <name file="hipaa835.xml">hipaa835</name>
   </validation>
</system>

<document>hipaa-835
   <in_validation>hipaa835</in_validation>
   <agent>XDHipaaAgent</agent>
</document>
```

General Rule Set

General rules are provided by the engine for use by any rule set. The rules are located in com.ibi.edaqm.XDRuleBase, which extends com.ibi.edaqm.XDRuleClass, the base of all rules. To include these rules, your own rule class should extend XDRuleBase instead of EXRuleClass.

isN

isN validates that a node is numeric with an optional leading sign.

```
<rule tag="xx" method="isN" />
```

Attribute	Description
max	Maximum number of digits permitted, not including sign. Optional.
min	Minimum number of digits required, not including sign. Optional.

isR

isR validates that a node is numeric with an optional leading sign and a single decimal point.

```
<rule tag="xx" method="isR" />
```

Attribute	Description
max	Maximum number of digits permitted, not including sign or radix. Optional.
min	Minimum number of digits required, not including sign or radix. Optional.
radix	Single character to be used to separate the decimal parts of the real value. The default is a period character. The radix attribute can be taken from the USING entry.

isDate

isDate validates that a node is a CCYYMMDD format.

```
<rule tag="xx" method="isDate" />
```

Attribute	Description
max	Maximum positions permitted. If omitted, 8 is assumed.
min	Minimum number of positions required. If omitted, 8 is assumed.

isTime

isTime validates that a node is in HHMM[SS] format.

```
<rule tag="xx" method="isTime" />
```

Attribute	Description
none	NA

Writing Rules in Java

Rules can be written in Java, loaded by the system at startup, and applied by specification in a rule. A rule class extends XDRuleClass, and can make use of any of its services. Each public method in the rule class that meets the rule signature can be applied by name as a rule. The rule methods can make use of service methods in the parental XDRuleClass.

Rules apply to the input document, which is represented as a tree of XDNodes. An XDNode is a self-contained representation of the element with its attributes, value, and relationships to other nodes. XDNode objects support a complete set of access methods to both obtain these values and to walk the tree. When the rule method is called, it is passed a reference to the node for which it was called, along with the rule parameters. Your rule method can walk the tree to locate related nodes and to evaluate them as appropriate.

In addition to writing rules, you can replace the existing search rules with your own search methods. The default search method supplied by iWay Software searches either a rule-supplied list carried in an attribute, or a delimited text file. You might replace this, for example, with a search method that tests lists in a relational data base or other storage media.

In this example, a node is checked to determine whether its value is included in the attribute's collection. If not, it is an error.

On entry to the rule, parameters are passed:

Parameter	Description
attributes	HashMap of rule attributes. The rule method can check for any attributes that it desires. A HashMap is a fast implementation of a Hashtable that does not serialize.
node	Node identified by the tag attribute in the rule. The rule method is called once for each node that matches the tag specification.
value	Data value of the addressed node. This differs from the node.getValue() return if the tag contained a subfield address (for example, tag=x:2).

```
import java.util.*;
import com.ibi.edaqm.*;
public class XDMyRules extends XDRuleClass
{
    public XDMyRules()
    {
    }
    public void specialRule(XDNode node, String value, HashMap attributes)
                throws XDException
    {
        trace(XD.TRACE_DEBUG, "specialRule called with parms: " +
              node.getFullName() + ", " + attributes.toString());
        String testValue = (String)attributes.get("value");
        if (value.equals(testValue))
        {
          node.setAssociatedVector(new XDEDIError(4,0,error,"explanation"));
          throw new XDException(XDException.RULE,
          XDException.RULE_VIOLATION,
          "node value "+value+" is not 'Value'");
        }
    }
}
```

Rule violations should throw an XDException describing the violation.

The parental class provides a group of services to assist in preparing rules:

Method	Description
boolean isYYYYMMDD(String date)	Validates that a date is formatted correctly.
boolean isInList(String list, String value)	Value must be in the list.
void trace(int level, String msg)	Text of the message is written to the system trace file. The level should be XD.TRACE_DEBUG, XD.TRACE_ERROR, or XD.TRACE_ALL.

Rules can also use all methods in XDNode to address the values in the passed node and the tree in general. Details of XDNode, XDException, and other available engine programming services can be found elsewhere in iWay Software documentation.

Rule violations must be returned as XDExceptions of class XDException.RULE. Two causes are available, XDException.RULE_SYNTAX if the rule is in error, and XD.RULE_VIOLATION if the data violates the rule. Syntax errors cause the document to be aborted, as it is presumed that rules should have been debugged. Violations should be posted to the node by the rule, and the engine continues to process the document. Violations are traced by the engine, and affect the later acknowledgement generation.

The error itself is posted to the node using the standard XDNode service setAssociatedVector(Object o) which records an object with the node. The special EDIError object contains the following elements:

Element	Description
class	Class of the error. Should be 4 for a syntax error, resulting in an AK4.
errorcode	Code to be returned in the ack AKx (997).
explanation	String explaining the error, for tracing use.
reserved	Must be 0.

To test an XML file for rules, the root of the rules file needs to match the root tag of the XML file. For instance if the XML file is as below:

```
<Customer>
<CustomerName>John Smith</CustomerName>
<bdate>11/77/2007</bdate>
</Customer>
```

The rules file might look like this:

```
<Customer>
   <using radix=",">
      <!-- End User Header Record Validation -->
<!--  Your rule goes here -->
      <rule tag="bdate" method="isDate" schema="Customer"/>
<!-end of rule(s) -->
   </using>
</Customer>
```

The rule needs to be created under `Registry->Components->Rules.`

Writing Rule Search Routines in Java

Short lists can be searched by built-in rule engine code. Longer lists, in which the values in the list are obtained not from the attribute directly, but instead from an external source, requires a rule list searcher tailored to the source. Lists might be obtained from a:

- Simple file.
- Database with values loaded at startup.
- Database with an access at each search request.

Each list might need its own search logic, tailored to the source and format of the list itself. To accommodate this, the rule engine allows list-specific search routines to be developed and added to the system. These routines are loaded at system initialization, and terminated at system closedown. Each must offer a search method that determines whether the passed value is valid.

Search routines must extend the XDRuleList class, which is part of the edaqm package: com.ibi.edaqm.XDRuleClass. The routine must offer three methods in the manner common to all XD extensions:

- init(String[] args) is called once at system initialization.
- term() is called once at system termination. It is not guaranteed to be called.
- search(String value) is called when the rule is executed.

The Rule List search code is identified in the <preload> section of the <system> area of the dictionary. The Preloads console page manages this section.

The following preload

```
<preload>
   <name file="RuleFileList(c:\ziplist.txt)"
         comment="validates zip codes">ziplist
   </name>
</preload>
```

specifies that a rule can be written, such as:

```
<rule tag="xxx" code="@ziplist" method="checklist"/>
```

The ziplist routine might load a list from a text file.

A simplified example procedure to load a file containing codes is shown below:

```
import com.ibi.edaqm.*;
import java.util.*;
import java.io.*;
/**
 * A rule list handler is a routine called to enable users search lists during
 * execution or the checkList rule. checkList() is a generally available rule to
 * test whether the contents of a document field are valid. The rule list handler is
 * invoked when the code= attribute indicates the name of a coder routine rather
 * than a simple list.
 * For example, <I>code="@list1"</I> will cause the search routine of the list1
 * class to be invoked.
 * The file read by this procedure consists of tokens separated by a new line, white
 * space or commas.
 */
public class XDRuleListFile extends XDRuleList
{
     String[] list;
     ArrayList al = new ArrayList(127);
     public XDRuleListFile()
     {
     }
     /**
      * The init method is called when a rule is loaded. It can perform any
      * necessary initialization, and can store any persistent information in the
      * object store.
      *
      * @param parms Array of parameter string passed within the start command
      * init-name(parms).
      */
     public void init(String[] parms) throws XDException
     {
         if (parms == null)
         {
             throw new XDException(XD.XDException.RULE, XDException.RULE_SYNTAX,
                    "no parms sent to " + name);
         }
         try
         {
             File f = new File(parms[0]);
             FileInputStream fs = new FileInputStream(f);
             long len = f.length();
             byte[] b = new byte[(int)len];
             fs.read(b);
             fs.close();
             String data = new String(b);
             StringTokenizer st = new StringTokenizer(data, ", " +XD.NEWLINE);
```

```
            while (st.hasMoreTokens())
            {
                String part = st.nextToken();
                al.add(part);
            }
        }
        catch (FileNotFoundException e)
        {
            throw new XDException(XDException.RULE, XDException.RULE_SYNTAX,
                   "list file "+parms[0] + " not found");
        }
        catch (IOException eio)
        {
            throw new XDException(XDException.RULE, XDException.RULE_SYNTAX,
eio.toString());
        }
    }
    /**
     * The term() method is called when the worker is terminated. It is NOT
     * guaranteed to be called, and applications should not rely upon this
     * method to update databases or perform other critical operations.
     */
    public void term()
    {
    }
       /**
        * Search the given value to determine whether it is in the list.
        *
        * @param value String to test against the list
        * @return true if found, false otherwise
        */
       public boolean search(String value)
       {
           return al.contains(value);
       }
}
```

Appendix A

Building iWay Service Manager Components

This appendix describes how to structure a .jar file containing your iWay Service Manager (iSM) components.

In this appendix:

Building iSM Components Overview

For demonstration purposes, the example in this appendix references a company called *xyzcorp*, which has a unique URI identifier of *com.xyz*. You are required to build agents (services) that will appear in the agents directory within the .jar file, for a sample project called *myproject*.

Understanding the Registration Class

The registration class instructs iWay Service Manager (iSM) where to locate your components within the. jar file (for example, com.xyz.agents.Agent1):

```
public void register (IComponentManager cm)
{
    cm.addExit("com.xyz.agents.Agent1", "FirstAgent");
}
```

The manifest of your .jar file points to the registration class, which is named *Register* and stored as *com.xyz.myproject.Register*. For example:

```
Manifest-Version: 1.0
Ant-Version: Apache Ant 1.9.5
Created-By: 24.79-b02 (Oracle Corporation)
Built-On: mycomputer October 28 2015 1116
Version: 1
Build: 1
Implementation-Version: 7.0.5
Name: IXTEComponentRegister: com.yourid.iwudpx.Register
```

The sample ANT build script applies a manifest to the .jar file, where the most important portion is the *IXTEComponent* section. iSM reads the manifest, uses this information to locate your registration class, and then executes the class to locate specific components.

The registration class identifies and locates the components (for example, agents, preparsers, iFL, and so on) for your project-specific development.

In addition to the .jar files you need to build your project, you must include the .jar files from the <*ism_home*>\lib directory such as iwcore.jar. By convention, all iWay-specific .jar files start with the letters *iw*.

Sample ANT Build Script

Although the sample ANT build script is created by iWay, it may not apply to your own build environment as is. You will need to modify the build script for directory paths, .jar file locations, project names, and so on.

```
<?xml version="1.0" ?>
<!-- ================================================ -->
<!-- Sample build script, set myproject to the value in engine -->
<!-- USE THIS ANT SCRIPT AS A STARTING POINT. -->
<!-- ================================================ -->
<project name="myproject" default="extension" basedir=".">
  <!-- ======================================== -->
  <!-- Macros used in the build -->
  <!-- ======================================== -->
  <property environment="env"/>
  <property name="build.sysclasspath" value="ignore"/>
  <property name="build.dir" value="classes" />
  <!-- out.dir is where I want the result -->
  <property name="out.dir" value="." />
  <!-- mycompany is my id; used to locate the register class -->
  <property name="mycompany" value="companyidentifier"/>
  <!-- engine is the name of my project extension; used to locate the
   register class and name the jar-->
  <property name="engine" value="myproject" />

  <!-- ============================================== -->
  <!-- The path below is becomes the compile classpath -->
  <!-- This MUST appear below the macro definitions -->
  <!-- ============================================== -->
  <path id="project.classpath">
     <fileset dir="${out.dir}">
        <include name="**/*.jar"/>
        <include name="**/*.zip"/>
     </fileset>
  </path>
  <!-- ================================ -->
  <!-- compile -->
  <!-- ================================ -->
  <target name="compile">

   <mkdir dir="${build.dir}"/>
   <javac
    classpathref="project.classpath"
    srcdir="${basedir}/src"
    destdir="${build.dir}"
```

```
    failonerror="${javac.failonerror}"
    debug="on"
    optimize="on"
    fork="yes"
    memorymaximumsize="256m"
    deprecation="off"
   >
    <include name="**/*.java"/>
   </javac>
  </target>
  <!-- jar it up; note where the register class will be found -->
  <target name="jar" depends="compile" >
    <mkdir dir="${build.dir}/META-INF"/>
    <tstamp/>
    <manifest file="${build.dir}/META-INF/MANIFEST.MF">
      <attribute name="Built-On" value="mycomputer ${TODAY} ${TSTAMP}"/>
      <attribute name="Version"  value="1"/>
      <attribute name="Build"    value="1"/>
      <attribute name="Implementation-Version"   value="7.0.5"/>
      <section name="IXTEComponent">
       <attribute name="Register
       "value="com.${mycompany}.${engine}.Register"/>
      </section>
    </manifest>

    <echo message="Building ${engine}.jar"/>
    <echo message="Building directory ${build.dir}"/>
   <jar
    jarfile="${out.dir}/${engine}.jar"
     basedir="${build.dir}"
     manifest="${build.dir}/META-INF/MANIFEST.MF"
   />
  </target>
  <!-- ================================================================-->
  <!-- clean -->
  <!--================================================================-->
  <target name="clean">
    <echo>Clean ${ant.project.name}</echo>
    <delete file="${out.dir}/${engine}.jar"/>
    <delete dir="${build.dir}"/>
  </target>
  <!-- ================================================================-->
  <!-- ***KICKER*** -->
  <!--================================================================-->
  <target name="myproject" depends="jar" description="BUILD"/>
</project>
```

Appendix B

Standard Business Exits

iWay provides pre-written exits, some written by iWay itself and some developed by iWay Service Manager users and distributed by iWay. The complete range of these exits is not included in this manual, however, we have listed some to give you an idea of the kinds of services that iWay exits provide.

All iWay exits are written using the facilities discussed in this manual.

In this appendix:

Business Exits Terminology

Each exit has a short name, by which it is commonly referred to, and a class name which is the actual module name in the provided software. When you configure an exit, the short names of all defined exits of the appropriate type are presented for selection. The short name can be any name meaningful to you.

The following topics describe included common exits using the short name commonly associated with that exit. For example, the XDSnipAgent is shown with the common short name Snip. The *iWay Service Manager* manual describes how to use the short name and the class name to make the exit available for actual use.

Business Agents

Provided standard business agents facilitate the development of business activities. When used alone, many of these business agents alone are sufficient to provide a complete business service. They can also be used in stacks and process flows as components of business processing.

Alternate Routing

Class name: XDIsReachable

Tests whether a specific IP target is reachable. If the target can be reached, the agent returns success. This agent is useful for supporting alternate routes such that a message can be directed to a primary or a backup host.

The edges returned are:

success	The host is reachable.
fail_connect	The host is not reachable.
fail_partner	The host is not known. For example, the host is not identifiable in the DNS.

Base64 Services

Class name: XDBase64Agent

This agent encodes or decodes a portion of the document. It is possible to store non-unicode bytes into an internal document even though under XML rules such a document cannot be serialized or deserialized (parsed). This agent enables base64 fields to be replaced with bytes, and such bytes to be represented in base64.

The edges returned are:

success	Successful conversion.
fail_parse	The input could not be converted.

Call Adapter

Class name: XDAdapterAgent

This agent calls an iWay adapter. To use this agent, the adapter must have been configured in an iWay Explorer program.

The edges returned are:

success	Successful operation of the adapter.
fail_operation	The adapter failed and did not report the direct cause.
fail_security	The adapter has returned a security violation.
fail_partner	The adapter reports that it cannot reach the needed system or that system has returned an error.
cancelled	The operation has been cancelled due to a timeout or other cause.

Check Schema

Class name: XDCheckSchema

Evaluates the input document against a specified schema. Optionally this agent can route the document to the fail edge or can simply set the appropriate document state that can later be tested with the iFL statement `_hasschemaerror()`.

The edges returned are:

success	Successful.
fail_parse	The schema could not be parsed or the document fails the schema check.
fail_missingschema	The schema could not be found.
fail_format	The input document was inappropriate for the test (e.g. the input was not XML).

Control

Class name: XDControlAgent

This agent enables or disables a listener. Configuration takes the name of the listener, and the operation to be performed. Operations supported are:

Operation	Description
start	Named listener is started, if necessary. It is enabled for acquiring messages.
passivate	Named listener is disabled for acquiring messages. This may not take effect immediately. The listener remains active and no resources are released.
pulse	Named listener is started/enabled to acquire messages. It runs through one acquisition cycle (for example, reading all rows in a RDBMS table) and is then stopped. Pulse is not guaranteed to work for all listeners. iWay recommends that this message be used only with extreme caution.
stop	Named listener is stopped. All resources are released.

Using the control agent, the activity on one listener can control another. For example, the detection of a message in an MQ Series queue can trigger an RDBMS listener to begin working with a relational table.

Control Character Filter

Class name: CCharFilter

Some input documents contain control characters such as tabs or bells. This agent replaces or removes such characters.

The edges returned are:

success	Successful conversion.

Copy

Class name: XDCopyAgent

This agent duplicates the existing document. It allows a comment to be emitted and a delay to be configured. Unless the delay or comment is needed, iWay recommends the use of the Move service instead.

Deflate/Compress Document

Class name: XDDeflateAgent

This agent compresses the document for transmission or storage. The document is flattened to a byte stream, and compression algorithms are applied. The output of this agent is a flat document.

The user can select the algorithm to be used.

none	No compression is used.
smallest	The smallest possible output is desired, regardless of the execution times.
fastest	The fastest execution time is desired, even if the degree of compression is reduced.
standard	A compromise between smallest and fastest.
Huffman	An entropic encoding is applied.

The edges returned are:

success	Successful deflation.
fail_operation	Deflation could not be performed.

Emit at EOS

Class name: XDEmitEOSAgent

This agent retains the previous document that was passed through the agent, and emits it as EOS (End of Stream). This allows the document to be sent to emitters only in the EOS state.

The edges returned are:

success	Successful.

Entag a Flat Document

Class name: XDEntagAgent

This agent adds an element tag around a flat document, and optionally, base64 encodes the document. If the input document is not flat, this agent is ignored.

The edges returned are:

success	Successful.

EvalWalk (Evaluate Input)

Class name: EvalWalk

Implements the iWay Active Document within an agent.

The exit evaluates each attribute and entity value in the document, executing all iWay functions. For example, a node:

```
<node1/ATR1="_ISERROR()">SREG(ip)</node1>
```

might become:

```
<node1/ATR1="false">123.45.678.910</node1>
```

Fail

Class name: XDFailAgent

The failure business agent always returns an XDException. If the parameter RETRY is used, the exception calls for a retry of the input, if possible. This business agent is useful when debugging rollback logic in a customer business agent.

FFSField

Class name: XDColAgent

This business agent generates an output <eda> <response> document based upon the incoming document. It also uses the properties file as described under FFSRow. The <response> document contains divided rows, with each metadata column (name, type) described in each row. This format is similar to that of many other database systems. It is a convenient exit for transformations.

FFSHotScreen

Class name: XDHotScreenAgent

The hotscreen business agent does not format its responses into an XML file. Rather, it accepts for relay any output generated by the iWay Full Function Server. This can include reports, server-produced XML files, or HTML tables.

FFSRow

Class name: XDStdAgent

This business agent accesses the iWay Full Function Server, and generates a response document that confirms to the eda dtd <response>. The business agent locates the appropriate iWay Data Server by using the server property file. For a TCP server named *myserv*, the statements might look like:

```
Jlink.myserv.protocol = tcp
Jlink.myserv.host = somehost
Jlink.myserv.service = 1234
```

The form of the property file is a sequence of lines in the form

```
<name>.<token>=value
```

where the name parameter is the business agent name. For the iWay business agents, the name is jlink.<server>.

Get Channel Information

Class name: XDChanInfoAgent

Information about the current state of the system can be obtained in XML form. Each channel (called a master) is included in the XML output, along with statistics on its current state.

```
<info>
   <channels>
      <master name="internal" state=" active " type="Internal" completed="0"
           failed="0" active="0" available="1">
         <user mean="0.0" variance="0.0" ehrlang="1.0"/>
         <cpu mean="0.0" variance="0.0" ehrlang="1.0"/>
         <threads group="internal">
            <thread name="W.internal.1"/>
         </threads>
      </master>
      <master name="file1" state=" active " type="FILE" completed="0"
           failed="0" active="0" available="4">
         <user mean="0.0" variance="0.0" ehrlang="1.0"/>
         <cpu mean="0.0" variance="0.0" ehrlang="1.0"/>
         <threads group="file1">
            <thread name="W.file1.1"/>
            <thread name="W.file1.2"/>
            <thread name="W.file1.3"/>
            <thread name="W.file1.4"/>
         </threads>
      </master>
   </channels>
   <internalqs>
      <queue name="internal1" size="0"/>
   </internalqs>
</info>
```

The edges returned are:

success	Successful extraction.

Inflate/Decompress Document

Class name: XDInflateAgent

The document contains a deflated message which is to be inflated, making the document usable. If the document is intended to be XML, it is parsed.

The edges returned are:

success	Successful completion.

fail_operation	The input data is not deflated in a form that is understood by the inflater.
fail_parse	Following successful inflation, the document could not be parsed into XML. the configuration must specify that the output is to be XML.

Jdbc

Class name: XDJdbcAgent

The JDBC business agent uses industry-standard JDBC to generate the standard <eda> <response> result. Because JDBC standards limit the available database operations that can be performed, the business agent is correspondingly limited. For example, this business agent cannot process the <focus> tag of the input document.

The business agent can, however, avail itself of any configured JDBC driver. This includes the iWay SAP, IMS, and transaction server drivers as well as drivers from third-party providers.

LocalMaster

Class name: LocalMasterAgent

Passes the document to a separate, named workflow and awaits the result. The agent accepts the name of a configured LOCAL protocol which can be configured in any manner desired. The routes are pooled for efficiency.

Unlike using an Internal Emitter to route a document to the Internal protocol, this call is synchronous. Any errors reported by the named workflow are reflected in the calling flow, and any timeout configured for the flow includes time in the called flow.

LocalMasterAgent differs from simply calling an external process flow in the normal manner in that a complete workflow is available including preparsers, reviewers, and so on.

It is strongly recommended that this agent not be used unless no other means is available to accomplish the application purposes.

Log Event

Class name: XDXALogEvent

The transaction activity log records events during the life of the server. Events include start and stop of the channel, each operation within the channel, and errors. User events can be added to the log using this agent.

If no activity log driver is operational, this agent is a functional NOP.

The only edge returned is:

success	Successful completion.

Manage Attachments

Class name: XDAttachOps

Performs supported operation on a document. This service can delete all attachments or a single attachment.

Manipulate Register Namespace

Class name: XDSREGNamespaceAgent

Registers can be set into namespaces. The namespace is the first portion of the register name, for example, *request.mimetype*. The mimetype special register is considered to be in the request namespace. The namespace is simply the first token of a multi-token register name.

Among the operations that can be performed by this service are to copy, rename or delete a namespace or test for the existence of any registers in a namespace.

Move

Class name: XDMoveAgent

Moves the input to the output without duplicating the information. This is the fastest means of copying from input to output.

Parse to XML

Class name: XDToXML

Parses a flat document into an XML document. If the input is already XML, this service performs no action.

This service is useful if a flat (non-XML) input has been received and must be changed into XML for subsequent operations.

Two possible returns are supported:

- success indicates successful parsing, in which case the output document consists of the newly parsed material.

  or

- fail_parse, in which the case output is the original input.

Parsing

Class name: XDToXML

Parses a flat document into an XML document.

The edges returned are:

success	Successful deflation.
fail_parse	The agent was unable to parse the input document.

QA

Class name: XDQAAgent

Emits a flattened copy of the input document to a file named in the init() parameters. The business agent outputs the document (XML or flat) either in QA mode or always, depending upon the setting of a parameter. If the QA mode is not on (set in the Diagnostic System Properties Console Configuration Page) and the always parameter is not set, this business agent acts as a move business agent. This business agent is designed to work as a chained business agent during debugging. The document and all special registers are included in the output.

Registers

Class name: XDSREGAgent

Sets or deletes one or more special registers under program control. The registers can be set to any of the supported scopes (message, flow, or thread) and can be of any defined category (hdr [header], user, or doc). Normally, registers cannot be set above the Message scope. The value to be assigned to any register can be an iFL expression which is evaluated when the service is executed.

The order of register assignment is undefined. You cannot, for example, assume that the following two assignments:

first	10
second	sreg (first) +1

will be executed such that *first* is assigned before *second*. The results of such a setting strategy is unpredictable. If such a setting is required, you must use two copies of the register agent.

Run a Shell Command

Class name: XDRunCmdAgent

The agent accepts a shell command such as "copy x y" and passes it to the shell for execution. The command is operating system/shell specific. The command is evaluated for iFL expressions, so that the command can be built from components.

The agent awaits completion of the command, and upon successful completion it returns on the success edge. Failure to complete the command causes a return on the fail_operation edge. The agent cannot detect a successful execution of the command that completes with the intended result, only that the command did indeed return successfully.

The return document contains the result of the execution as the value of the <cmdoutput> cdata element. The interpretation of the output, if any, is the responsibility of the flow.

```
<status>
  <command>attrib c:\iway7\*.*</command>
  <cmdoutput>    ![CDATA[
     A          C:\iway7\iway7.cmd
     A          C:\iway7\license.xml
     A          C:\iway7\uninstall.exe
     ]]>
  </cmdoutput>
  <exitcode>0</exitcode>
</status>
```

Users are cautioned that not all shell commands can be executed. For example, MS Windows cannot reliably interpret and execute some built-in commands. What commands can be executed are shell-specific.

The edges returned are:

success	Successful deflation.

fail_operation	The shell reported that the command failed.

Run Preemitter

Class name: XDPreemitAgent

Preemitters are intended to convert a document in preparation for emission. This agent executes a named preemitter (which must have previously been defined and configured) and generates formatted output.

The edges returned are:

success	Successful deflation.
fail_preemit	The preemitter reported an error.

Set Constant

Class name: XDConstantAgent

This agent replaces the input document with a constant document. Because all parameters are evaluated with the iWay Functional Language, this agent is frequently used to modify the input or load a file from a disk.

The edges returned are:

success	Successful.
fail_parse	An iFL expression could not be evaluated.

Set Document State

Class name: XDDocAgent

Sets or resets the current document's state.

The states that can be set are:

- Error: Whether the document is in error state. If this is set when the document leaved the flow, the document is sent to the errorTo address. This state can be tested in iFL by the _iserr() function.

Status can be set to SET, RESET or LEAVE.

- Schema Error: If the document has passed through a schema check, this state reflects whether and error was reported.

  Status can be set to SET, RESET or LEAVE.

Snip

Class name: XDSnipAgent

Sends a subtree of the input document to the output document. This business agent is especially useful in pulling bodies from documents that surround the body with a header structure.

SQL

Class name: XDSQLAgent

The XDSQLAgent enables you to execute an arbitrary SQL statement, specified as a service configuration parameter. The service returns an EDA response document, as defined elsewhere.

This agent differs from the more complete RDBMS agents and adapters in that it does not interrogate an iWay response document to obtain its SQL; instead it accepts the SQL as a configuration parameter and modifies the statement based on the current document or environment.

The SQL expression can be modified during execution by iWay standard parameter expressions.

This agent also returns a variety of edges any combination of which can be wired. It can return an edge of the SQL XOPEN state, for example, 42S02 for “Table not found”.

The edges returned are:

cancelled	The operation was terminated by a cancel.
duplicate	The operation detected a duplicate condition such as inserting a tuple with a unique key that already exists in the database.

fail_notfound	The operation was unable to succeed due to a milling component of the SQL DML statement, such as a missing index or table.
fail_operation	The operation failed due to invalid parameters of other error.
notfound	The operation did not return an error but the number of rows affected was 0. note that both notfound and success are returned to permit the application interpret the result as appropriate. **Note:** Both notfound and success are returned to allow the application interpret the result as appropriate.
success	The operation completed successfully.
timeout	The SQL operation terminated due to a timeout.

Note: Different SQL implementations may return different results to the service depending on how the conditions are interpreted.

The service sets the SQLcount special register to assist the application in handling return situations. For example, if a select returns no rows, the service returns candidate edges of notfound and success. The application can test the special register on a success edge to route to the appropriate logic. The count in this register is obtained by interrogating the SQL system during a non-select operation (insert, update, or delete) or by counting the returned rows for a select.

Store/Load XML

Class name: XDSREGTreeAgent

The XML contents of the message are stored into a named special register, or can be replaced from a named special register. The XML can be stored from the root or the child of a named parent within the tree. Similarly, the stored tree can be loaded (restored) to the root or as the child of a named parent. There is no requirement that the stored tree be restored to the same location from which it was taken.

Transform

Class name: XDTransformAgent

The transform named in the business agent initialization parameter is applied to each document. The transform name should be an alias specified in the *transforms* section of the engine configuration. The transformation can be either an iWay transform or an XSLT transform.

The transform business agent is especially useful as part of a sequence of business agents in which the output is chained to another business agent for execution of a business process.

```
XDTransformAgent(hipaa270)
```

XML To/From Special Register

Class name: XDSregTreeAgent

This agent assigns the tree in the current XML document, or a subtree, to a special register or extracts a tree in a register and assigns it to the current document. Note that some iFL expressions, including XPATH(), can operate upon a tree in a register.

The edges returned are:

success	Store or extract was successful.
fail_duplicate	The node to extract has siblings and a single root node cannot be determined.
fail_notfound	The node to be operated upon was not found in the document.
fail_operation	Invalid parameters of other error.

ZIP Document Contents

Class name: XDZipOutAgent

The current contents of the input document are replaced with a ZIP-compressed and structured content. This can be written to a file or transmitted; standard ZIP tools can be used to decompress the created output.

The edges returned are:

success	Zip operation performed.

Protocol Business Agents

Protocol business agents are standard business agents that can emit a document to a designated target or can read from a source. In the standard situation for output, a <replyto> should be used, however for some circumstances, a protocol business agent can be employed. In all cases, if an emitter can be used it should be used in preference to a protocol business agent.

Each protocol business agent takes initialization parameters appropriate to the protocol type.

Protocol Agents Result Document

```
<emitstatus>
   <protocol>name</protocol>
   <status>status code</status>
   <parms>stringOfParms</parms>
   <native>nativeSystemErrorCode</native>
   <text>error descriptive text</text>
   <name>nameOfCreatedFile</name>
   <channel>name of the channel</channel>
   <nodename>name of the pflow node</nodename>
   <retries>numberOfRetries</retries>
   <timestamp>timeOfCompletion</timestamp>
</emitstatus>
```

AQ

The document is posted using Oracle AQ.

Class name: XDAQEmitAgent

Parameter	Description
host	Network name of the machine where the target system resides.
password	Password of this user.
port	Service port that Oracle uses to exchange messages.
qtable	Database table used to manage queues.
queue	Queue to receive documents.
sid	Name of Oracle database service.
user	User name of access to the target system.

The result of the post appears in the <native> section of the <emitstatus> result.

EMAIL

The EMAIL emit sends the document to a designated recipient. The document always goes to the recipient specified in the parameters.

Class name: XDEmailEmitAgent

Parameter	Description
from	Author.
host	Email host.
subject	Subject of this message.
to	Intended recipient as an email address.

File

The File Emit writes the contents of the current document or other specified information to the file system. The source specification can be blank or can specify any iWay expression, including xpath(). If the document is XML and xpath() is specified, each value meeting the xpath() criteria is written as a file; the assigned names are allocated sequentially based on the entered pattern.

Class name: XDFileEmitAgent

Parameter	Description	
directory	Directory to which the file is to be written.	
pattern	File name pattern. In the pattern, a * character is replaced by the current timestamp. So T*.out might become T2002-06-25_12:02:24:43.out. A # is considered a unique number. The number of # characters controls the length.	
Run Preemitter	If a preemitter exists on this route for output, this allows it to be bypassed for this emit operation. Perform this action to enable the output file to include non-processed (raw) information.	
Return	Specifies the desired output of this service:	
	status	Status document which contains the name for each file written by this service.
	input	Input document as received.
	swap	Swap the input document, but any information written as a result of an xpath() from the document is replaced with the file name to which it was written.
Source	Source of the data will be written to the file. If this is omitted, the input document is written to the file. If this contains an xpath() expression, for example, xpath(/root/x/value), each located value in the input document is written to a file. This parameter can contain any value, including a constant. For example, specifying file(/x/y.txt) will result in the contents of the named file being written.	

Output Edges	
success	Standard success edge.
fail_parse	Could not parse the xpath expression of other entered function.
fail_operation	File write failure occurred.

Output Edges	
notfound	No information meeting the source criteria was located.

File Read

The local file input accepts a request to read a file from the local file system and outputs this request as an XDTextDocument. The transform business agent and standard output can accept and operate upon this output. The input business agent can operate on flat or XML data, and can emit data in either form.

Class name: XDFileReadAgent

Parameter	Description
basepath	Optional directory path to be tried if the passed in name is not absolute.
encode	Encoding to be performed on the input. Can be "asis" or "base64". If omitted, no encoding is performed.
entag	Optional tag to be set around the input. If omitted or empty, no tag is used.
format	Format of the input document. Optional. Can be "flat" or "xml". Default is flat.
tagname	Tag of input document whose value is the file name to read.

The input document must be:

```
<filename>pathToFile</filename>
```

assuming that the tagname is set to filename.

If the input format is XML or an entag parameter is used, the output is in XML form. Otherwise, the output is a flat document.

Note: The XDFileReadAgent assumes that the input data format is flat by default, or if the entag option is set to any value even if the XML input data format is selected.

FTP

Class name: XDFTPEmitAgent

Parameter	Description
directory	Directory on the host system to receive the output.
duration	Time to wait (in seconds) before retrying if the FTP system cannot be reached.
host	Host system of FTP target.
maxretries	Maximum number of time to retry. 0 is the default value.
password	Password on the host system.
pattern	File name pattern. In the pattern, a * character is replaced by the current timestamp. So T*.out might become T2002-06-25_12:02:24:43.out.
port	Integer port number. 21 is the default value.
user	User ID on the host system.

FTP Read

Reads one file through FTP. The service provides many options. The key options are shown in the following table.

Class name: XDFTPReadAgent

Parameter	Description
hostname	Name of the FTP host.
username	Name of the user to log into the FTP server.
password	Password for the user to log into the FTP server.
tagname	Name of the tag from the input document that identifies the file name to be read.
enclosetag	Tag used to enclose the data read.
basepath	Option name of path to use if tagname is not absolute.
format	Indicates if data is XML or flat.

Parameter	Description
delete	Indicates if the data should be deleted after the read.

HTTP GET

The GET facility input accepts a URL in the incoming document and issues an HTTP GET through that URL. The transform business agent and standard output can accept and operate upon this output. It is presumed that some output is returned, otherwise an error is raised.

Class name: XDHTTPReadAgent

Parameter	Description
tagname	Tag of input document whose value is the URL.

The input document might be:

```
<?xml version="1.0"?>
<inurl>http://localhost:1234/xmlone.xml</inurl>
```

assuming that the tagname is set to inurl.

In addition to the URL parameter, any other parameter denoted by <parm>s are treated as GET parms, with the values specified in the parm value. For example:

```
<agent>emitHttp
   <parm name='url' required='y'/>
   <parm name='header1' required='y'/ prompt='first header'>
   <parm name='header2' required='y'/ prompt='second header'>
</agent>
```

GET parms with no value are ignored. GET parms with values of x=y format are changed such that the x is the name and the y is the value. Thus a deploy user can "fill in the blanks."

The returned information is passed on as the output document. If it begins with a **<**, it is considered XML; otherwise, a flat document is returned.

HTTP POST

The document is posted using HTTP to a designated server.

Class name: XDHTTPEmitAgent

Parameter	Description
returnresponse	An enumeration of 'RESPONSE' or 'STATUS'. This parameter is optional, and the default is 'RESPONSE'.
url	URL for this post.

In addition to the URL parameter, any other <parm>s are treated as headers, with the values specified in the parm value. For example:

```
<agent>emitHttp
   <parm name='url' required='y'/>
   <parm name='header1' required='y'/ prompt='first header'>
   <parm name='header2' required='y'/ prompt='second header'>
</agent>
```

Header parms with no value are ignored. Header parms with values of x=y format are changed such that the x is the name and the y is the value. Thus a user can "fill in the blanks."

The result of the post appears in the <native> section of the <emitstatus> result.

If the returnresponse parameter is STATUS, the status message is always generated. If it is RESPONSE (the default) the actual information returned from the POST is emitted, except in the cases of an error in which case the status information is emitted. The POST is considered successful if the POST can reach its destination and a 200 status is received.

A successful returnresponse is considered to be XML if the first character is a **<**. Otherwise, a flat document is created.

Internal

Class name: XDInternalEmitAgent

The server offers an internal queue facility to which messages can be posted. An Internal Channel can be configured to read from the queue and behave in all other ways like a standard channel. This means one channel can pass work off to another channel, which can have preparsers, splitters, process flows and emitters, and is in no way distinguishable from any other channel.

The emitter specifies whether the channel operation is to be asynchronous (default) or synchronous.

As an asynchronous emit, the emitter immediately proceeds to the next node. This is a "fire and forget" emit operation and no result is returned.

In synchronous mode, the emitter waits for the internal channel operation to complete and returns the result of its default reply to the caller. Naturally, a timeout or cancel can affect this wait.

The edges returned are:

success	The operation completed successfully.
fail_notfound	The named queue was not found. The internal channel had not been started to register the queue, or the queue name is mis-specified.
cancel	The emitter was awaiting the synchronous result when a cancel was received.
timeout	The emitter was awaiting the synchronous result and timed out.

MQ Series

Class name: XDMQEmitAgent

Parameter	Description
correlid	Correlation ID.
queuemanager	Queue Manager name.
queuename	Queue name.

The message ID appears in the <name> element of the <emitstatus> result.

Sonic

Class name: XDSonicEmitAgent

Parameter	Description
broker url	Used to reach the Sonic broker.

Parameter	Description
Other Sonic-specific parameters	Parameters to control the Sonic emit.

TIBCO RV

Class name: XDTIBRVEmitAgent

Parameter	Description
subject	Subject for the emitted message.
daemon	Access daemon.
network	Network name.
Other TIBRV-specific parameters	Parameters to control the TIBRV emit.

Preparsers

Many preparsers are available to assist you in preparing message flows. Some common ones are described in this section.

Append

Class name: XDAppend

Adds constant text to the head and/or tail of the incoming message. This preparser is often useful in adding XML tags around messages.

DelVal

Class name: XDDelVal

Delimited value handler. Useful when the incoming message is a delimited value stream such as lines of comma separated values. The first line can optionally hold field names (DIF format input). An XML document is generated from the input with a child node of each row, and children of that row in turn for each value.

Entag

Class name: XDEntag

Surrounds the incoming message with a designated tag. Optionally passes on the incoming message as CDATA to the tag. Additional features allow configuration of attributes including source, message ID, and correlation ID.

ErrorFilter

This preparser operates on iWay error documents. Error documents are a standard by which iWay Service Manager reports errors back to the user. These documents contain the complete incoming message associated with the error. The preparser extracts the original message and passes it into the system for processing.

Messages identified as not being error documents are passed on as is.

MultiPart

Class name: XDMultiPart

Divides multipart documents for later use. This preparser must be last in the chain. When used, the multipart message is broken up so that the body of the multipart message is part 0 and subsequent attachments are 1 to n. The multipart object is carried in the XDDocument and can be retrieved and interrogated by your own business agent to obtain the individual parts for processing. If the body is a MIME type of _/XML, it is parsed and carried as an XML document.

PGPDecrypt

Operates on the text of a PGP-encrypted message to decrypt it for use. The preparser accepts either a passphrase or uses a registered keyring for key purposes. The configuration determines whether the document is to be treated as XML or flat once it emerges from the preparser.

Replace Characters

Characters are replaced on a one-for-one basis. Each character in the input specification is replaced with the parallel character in the output specification.

Characters can be specified as simple characters such as %, standard encoding (\n, \t), or by using a hex specification, such as \x05. For example, to replace the carriage return (hex 0d) and the tab characters with the letter x, the specification would be:

input	\x0d\t

replace	xx

Splitter Preparsers

Splitting is performed by preparsers. A splitting preparser must be the first user-configured preparser. Splitting preparsers are provided for XML, non-XML, and EDI messages.

EDISplit

Class name: EDISplitPP

Splits EDI documents on ST segments. Each split subdocument is passed to a transform for conversion to XML.

FlatSplit

Class name: XDFlatStreamPreParser

Splits non-XML messages that are to be split on a recognized character. The configuration specifies the split character - default is the end of line for the platform. The character can be specified as any character, a special character, or a hex character.

Character	Description
X	Any character
\n	New line
\r	Carriage return
\t	Tab
\xab	Ab are hex characters such as 0A.

A standard use of this preparser is to handle individual lines of a large, delimited file.

XMLSplit

Class name: XDXMLStreamPreParser

The incoming XML document is divided into smaller documents, broken at a specified repeating child. For example:

```
<a>
 <b>
  <c>one</c
 </b>
 <b>
  <c>two</c>
 </b>
</a>
```

With a split specified as a/b, it results in three passes through the configured flow.

Pass 1

```
<a>
 <b>
  <c>one</c>
 </b>
</a>
```

Pass 2

```
<a>
 <b>
  <c>two</c>
 </b>
</a>
```

Pass 3

No data.

getStreamState() is STREAM_EOS and special register eos is defined as 1.

DelValSplit

Class name: XDDelValStream

Operates in the same manner as the delimited values preparser DelVal. One or more rows of delimited values are emitted as a tree, until the input is exhausted.

Example: Input file

```
One,Two,Three
10,20,30
100,200,300
```

Setting for one row per pass with a root of *tree* yields:

Pass 1

```
<tree>
<One>10</One>
<Two>20</Two>
<Three>30</Three>
<tree>
```

Pass 2

```
<tree>
<One>100</One>
<Two>200</Two>
<Three>300</Three>
<tree>
```

Pass 3

No data.

getStreamState() is STREAM_EOS and special register eos is defined as 1.

Preemitters

Many preemitters are available to assist you in formatting output messages. Some common ones are described in this section.

Detag

Class name: XDDETAG

Removes one set of surrounding tags.

EncryptPGP

Class name: PGPEncrypt

Encrypts an outgoing message in PGP. Accepts either a passphrase or a recipient alias based on a registered keyring.

EntityRepl

Class name: XDEntityRepl

Replaces XML entity sequences with their natural sequences. For example, < is replaced with <.

iWayTrans

Class name: XDXMLGpreEmitter

Runs an iWay transform.

Parameter	Description
encoding	How to encode the result. Choices are document encoding, file system encoding, or listener configuration encoding.
template	Defined name of the transform template

MultiPart

Class name: MultiPart

Assembles multipart documents into a single message for emitting. This preemitter must be the last in the chain. The multipart object carried in the XDDocument is reassembled into a valid multipart document.

XDCharRepl

Class Name: XDCharRepl

Replaces characters in the output with alternate characters; provides carriage return/line feed mapping.

XSLTTrans

Class name: XDXSLTPreEmitter

Runs an XSLT conversion of the output document. This must be the first preemitter in the chain. This preemitter is frequently used to run a standard XML to HTML transformation.

Parameter	Description
template	Name of the defined XSLT template.

Reviewers

Many reviewers are available to assist you in reviewing messages. Some common ones are described in this section.

EvalWalk

Class name: XDEvalReviewer

Implements the iWay Active Document within an agent.

The exit evaluates each attribute and entity value in the document, executing all iWay functions. For example, a node:

```
<node1/ATR1="_ISERROR()">SREG(ip)</node1>
```

might become:

```
<node1/ATR1="false">123.45.678.910</node1>
```

Parsing

Class name: XDFlateToXMLReviewer

Parses a flat document into XML. This reviewer is usually needed only in a flat channel, as in XML [non-flat] channels such parsing is automatic before the first reviewer is executed.

QA

Class name: XDQAReviewer

Emits a flattened copy of the input document to a file named in the init() parameters. The business agent outputs the document (XML or flat) either in QA mode or always, depending upon the setting of a parameter. If the QA mode is not on (set in the Diagnostic System Properties Console Configuration Page) and the always parameter is not set, this business agent acts as a move business agent. This business agent is designed to work as a chained business agent during debugging. The document and all special registers are included in the output.

Registers

Class name: XDSregReviewer

Sets or deletes one or more special registers under program control. The registers can be set to any of the supported scopes (message, flow, or thread) and can be of any defined category (hdr [header], user, or doc). Normally, registers cannot be set above the Message scope. The value to be assigned to any register can be an iFL expression which is evaluated when the service is executed.

The order of register assignment is undefined. You cannot, for example, assume that the following two assignments:

first	10
second	sreg (first) +1

will be executed such that first is assigned before second. The results of such a setting strategy is unpredictable. If such a setting is required, you must use two copies of the register agent.

Appendix C

Supplied Tools

Tools are "stand-alone" programs that can be run from the iWay Service Manager command window. Tools are usually provided to facilitate field service. Some of the tools available are described in this appendix.

In this appendix:

CreateMultipart

This tool constructs a simple MIME multipart document from a set of "parts". It accepts as input a standard properties object (file) describing the operations to be performed. The location of the properties file is the only parameter.

```
tool CreateMultipart <path to properties file>
```

The properties file must contain the required property *out.name*, with a value of the path to the multipart document to be constructed.

A group of properties named inX.<propertyname> describes each "part" to be included, where X is replaced with the part number 1 to n. The required property *name* for each in part has a value of the path containing the data to be included in the part. Any other properties to be copied to the message part are expressed as *header* properties. For example,

```
out.name=c:/testarea/testmp.mpart
# part 1 is an XML document as a header
in1.name=c:/testarea/testmp.xml
in1.header.Content-Type=application/xml
#part 2 is the information part one describes
in2.name=c:/testarea/test.txt
in2.header.Content-Type=text/document
in2.header.Content-Id: mydata
```

The content of the XML part for our example is

```
<mydoc>
    <request>
        <user>user1</user>
        <password>password1</password>
        <route>COPY</route>
        <input type='attach'/>
    </request>
</mydoc>
```

Running the tool command tool

```
CreateMultipart c:/testarea/testmp
```

produces this file:

```
message-id: <7754670.1104164220167.JavaMail.rb02237@Beck2000>
content-type: multipart/mixed;
boundary="----=_Part_1_7835377.1104164220042"
content-length: 593
mime-version: 1.0
------=_Part_1_7835377.1104164220042
Content-Type: application/xml
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=testmp.xml

<mydoc>
    <request>
        <user>user1</user>
        <password>password1</password>
        <route>COPY</route>
        <input type='attach'/>
    </request>
</mydoc>
------=_Part_1_7835377.1104164220042
Content-Type: text/document
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=test.txt
Content-Id: mydata

This is a test of the multipart tool

------=_Part_1_7835377.1104164220042---
```

FromWhere

The FromWhere tool explores the system loaders and classpath. It reports where an object was found and from where it has been loaded. To use the command, type:

```
tool FromWhere <name of object>
```

For example, assume that you have written an agent named com.ibi.agents.myagent, and you are unsure of from where it is loaded.

```
tool FromWhere com.ibi.agents.myagent
```

The tool displays the name of each JAR containing the object on the classpath, the location of the JAR itself (it might be in a RAR or a WAR for example) and if the object is currently loaded by the system, the location in which the classloader found the object.

Signing Configuration Files

The *iwdevkit* extension includes a utility program for signing and verifying configuration files using XML digital signature. This includes the runtime dictionary and all compiled process flows. When running in an appropriate license, the server will not start if it detects an unsigned or incorrectly signed runtime configuration nor will it execute an unsigned or incorrectly signed process flow.

To run this utility, use the server's *tool* command.

```
>tool SignTree [-s|-v] <input file> [<output file>]
```

If the output file is omitted, the input file is rewritten with an XML signature. Use -s to sign (this is the default) and -v to verify.

Files to be manipulated can be anywhere in the file system. For a given configuration, the dictionary is *<iwayhome>/config/<confignаme>/<confignаme>.xml.*

As a convenience, the xml suffix will be added by the utility if it is omitted. iWay home is the base of the iWay server installation.

Process flows are carried in the configuration's *processes* subdirectory. They are named *<name>_compiled_date.xml*, and it is these that must be signed. The *_gui* and *_image* files are used for design only, and are not needed at runtime.

The SignTree utility is also available as a Windows OS command shell program. The command file for using this is located in the /doc subdirectory of the iwdevkit extension. This file, signtree.cmd, should be placed into the iWay home directory. The command will only operate if the iwdevkit extension is in the extensions area.

Naturally, the iwdevkit extension should not be distributed to customers.

Testing Functions

The testfuncs tool enables entry of an expression in the iWay functional language. It evaluates the expression and returns the result. It is intended for technical users. To use the command, type:

```
Tool testfuncs <path to an xml document>
```

The tool supports the set subcommand to set a special register value. For example, assume you want to try how the special register works with arithmetic:

```
funcs->set aa 1
stored
funcs->sreg(aa)+2
<superroot [baseNode]>
   <arith [funcNodeMath]>+
      <sreg [funcNodeFunctionSreg]>sreg(aa)
         <p-literal [funcNodeLit]>aa</p-literal>
      </sreg>
      <x-literal [funcNodeLit]>2</x-literal>
   </arith>
</superroot>
3
funcs->
```

Another example might make it clearer:

```
funcs->_substr('abcde',2,4)
<superroot [baseNode]>
   <substr [funcNodeFunctionStr.substr]>_substr(&apos;abcde&apos;,2,4)
      <p-literal [funcNodeLit]>abcde</p-literal>
      <p-literal [funcNodeLit]>2</p-literal>
      <p-literal [funcNodeLit]>4</p-literal>
   </substr>
</superroot>
cd
funcs->
```

Each test shows the abstract syntax tree that results from the compile of the function. Problems with the compilation can usually be understood by analysis of the tree.

Some function testing requires use of special registers. The command:

```
set regname value
```

sets the specified special register to the designated value. The value operand is evaluated, so that a value can, for example, reside in a file.

If the value after evaluation begins with a left bracket, the test tool assumes that the value is to be parsed as XML and an XML tree is to be loaded into the named register. Otherwise, the register is set to a string of the input value.

To set register one to the value valone, use

```
funcs->set one valone
stored
```

A file in the root named sregdoc.xml contains an XML document. To load it into a special register named xmldoc, use the following command.

```
funcs->set xmldoc file('/sregdoc.xml')
stored xml
```

Testing xpath

The testxpath tool enables you to try xpath expressions against an std document. The xpath expression will be evaluated against the provided document and the result returned. To use the command, type:

```
Tool testxpath <sampledocument>
```

Assume the standard document is the same as that shown in testfuncs.

```
Enter command:>tool testxpath \smalldoc.xml
<?xml version="1.0" encoding="ISO-8859-1" ?>
<a>
   <top>
      <b>one</b>
      <b>two</b>
      <x>sreg(iwayhome)</x>
   </top>
</a>
```

Now enter an xpath expression against the sample document.

```
xpath->/a/top/b[1]
<superRoot [baseNode]>
   <a [baseNode]/>
   <top [baseNode]/>
   <b [baseNode]>
      <functionPredicate [filterInt]>1</functionPredicate>
   </b>
</superRoot>
values->'one'
tree-><?xml version="1.0" encoding="ISO-8859-1" ?>
<xpathresult>
   <b>one</b>
</xpathresult>
list-><b>one [parent=top]
xpath->
```

Appendix D

Debugging Tools

Regardless of how well designed and tested an application may be, there will be times when problems arise or unexpected results are returned. iWay Service Manager (iSM) provides tools to assist in testing applications and in helping to locate the cause of a problem.

Tip: Before you contact Customer Support Services (CSS), a key point to consider is the amount of information you can provide to a CSS representative. If more detailed information and guidance is provided, then the time that is required to resolve an issue will be dramatically reduced. For example, you can include a screen capture to help describe a console page issue. In this case, it would be even more helpful to draw a circle around the area in question using a simple graphics editor (for example, Windows Paint). Similarly, iSM may produce multiple trace files that are large in size. Identifying the specific trace file that contains the error for your CSS representative, and if possible, where the error seems to manifest is very helpful.

In this appendix:

Tracing

Tracing is key to diagnosing problems and thus to application reliability. iWay Service Manager provides a full complement of tracing services, oriented to diagnostic analysis of the running system. Tracing provides a step by step explanation of the activity of the server internals.

Experience teaches that tracing can also be a major source of performance degradation. The iWay console enables you to select the levels of traces that you wish to generate and display. Unless you are diagnosing a problem, you will usually want to limit the tracing to error level only.

Traces are offered in several levels, controlled independently:

Trace	Description
BIZERROR	Shows error messages associated with processing a document. Rule violations such as errors detected during analysis of EDI documents are BIZERROR level.
DATA	Shows the incoming and outgoing documents as they pass to and from the protocol channel.
DEBUG	Reports data that is helpful for debugging situations. It shows logic that tracks the path of a document.
DEEP	Used for detailed logic tracing. Stack traces are reported by the system in DEEP level. Usually you will use this level only if instructed to do so by iWay support personnel.
ERROR	Shows error messages relating to operation of the adapter. This level always displays.
EXTERNAL	External traces are disabled by default. The server provides a route for third party tools that run under control of the server (for example open source Apache code) that uses java tracing to pass their traces through the server. Control of the traces lies with the external code, for example Log4J. The iSM server issues the traces when the external trace level is enabled. **Note:** Unlike other trace levels, masking the level does not affect the generation of the trace line and the resources that the generation of the trace consumes. Such control is managed by the Log4J or other control mechanism.
INFO	Provides messages that the user always needs to see to verify activity. The server puts out almost no messages at INFO level.
TREE	Shows the document as it enters and leaves the system in XML form. Intermediate processing as a document evolves is done at TREE level.
WARN	Warnings are important messages that do not denote an error, but should be reviewed.

All except INFO and ERROR levels can be masked off, so that the log contains only brief informational and error messages.

Control of tracing is controlled in the Configuration System Properties Diagnostics page of the console. It offers the ability to set the log directory, manage the size of the log files, and turn logging on and off. Unlike most design time settings, changing trace levels takes immediate effect in the run time system. Changing the log location does not take effect until the next restart of the system.

Interacting With Log4J

Some components, both provided by iWay and your own, may make use of third-party libraries that log with Log4J. In order to capture these logs, iWay initializes the Log4J system as the server starts. If the server finds a file named "log4j.properties" in its current working directory, then it will try to configure Log4J with this file. If no such file is found, then Log4J is configured with the root logger at the INFO level and an appender that routes Log4J messages to the log of the server.

For example, the NHTTP extension uses an HTTP client from Apache Commons, which traces with Log4J. To view the wire header and context messages for the HTTP client, you can use the following log4j.properties file:

```
log4j.rootLogger=INFO, xd
log4j.appender.xd=com.ibi.logging.XDLog4jAppender
log4j.appender.xd.mapDebugToDeep=true
log4j.appender.xd.layout=org.apache.log4j.PatternLayout
log4j.appender.xd.layout.ConversionPattern=[%C{1} - %M()] - %m
log4j.logger.httpclient.wire.header=DEBUG
#log4j.logger.org.apache.commons.httpclient=DEBUG
```

In this example, rootLogger is set to the INFO level and then XDLog4JAppender is added. The `.xd` on the definition of the rootLogger is required. Since most applications that use Log4J seem to handle the DEBUG level as the server does its DEEP level, the mapDebugToDeep option on the appender writes Log4J DEBUG messages to the DEEP level of the server. Finally, the appender is set with a layout pattern that prints class, method, and message. This mirrors the iWay format.

The last two lines control specific Log4J loggers created by the HTTP client. For more information, refer to the documentation for the relevant component.

Other components used within the server are configured individually as appropriate for the component. For example, in OpenJPA you might add the following line in order to trace JPA SQL calls:

```
log4j.category.openjpa.jdbc.SQL=DEBUG
```

For a general description for tracing OpenJPA, see the following website:

http://openjpa.apache.org/builds/1.0.1/apache-openjpa-1.0.1/docs/manual/ref_guide_logging_log4j.html

Another example is the HTTP client (4) used by the nHTTP provider, which can be traced. For more information, see the following website:

http://hc.apache.org/httpcomponents-client-4.3.x/logging.html

For example, header wire and context logging would add these lines to the log4j property file.

```
log4j.logger.org.apache.http=DEBUG
log4j.logger.org.apache.http.wire=ERROR
```

As you add external information to the Log4J properties file, add only the controlling entries.

Log4J is classed by iWay as an external log system, and is enabled in the server by setting the **external** level. You can do this by using the set command from the shell.

Deferred Tracing Mode

iWay Service Manager (iSM) offers a deferred tracing mode. In this mode, most channel traces are deferred until an error is detected in the channel. The traces associated with the message are held, and if the message processing completes with no error, the traces are eliminated. If an error trace is issued during the processing of the message, then the held traces are emitted along with all subsequent traces for the message. In either case, a trace message at the debug level indicates the number of deferred traces.

Deferred trace messages are held in memory, and are formatted. Therefore, use of deferred tracing does *not* significantly affect performance and can result in increased memory use. Deferred tracing is useful in situations where a message failure is rarely detected. For example, 1 out of 100 messages raises an error. Using deferred tracing can avoid the need for large trace files when only the failure is being debugged. Deferred tracing at high trace levels (for example, debug, deep debug, and so on) is not recommended for normal production runtime. In these cases, only low trace levels (for example, info, error, and warn) are usually enabled, as recommended for general server use.

Deferred tracing can be set on a system-wide basis using the configuration console in the tracing page. Additionally, deferred tracing can be set using the following command line set option

```
set tracedefer [-m <channelname> [on | off]
```

where:

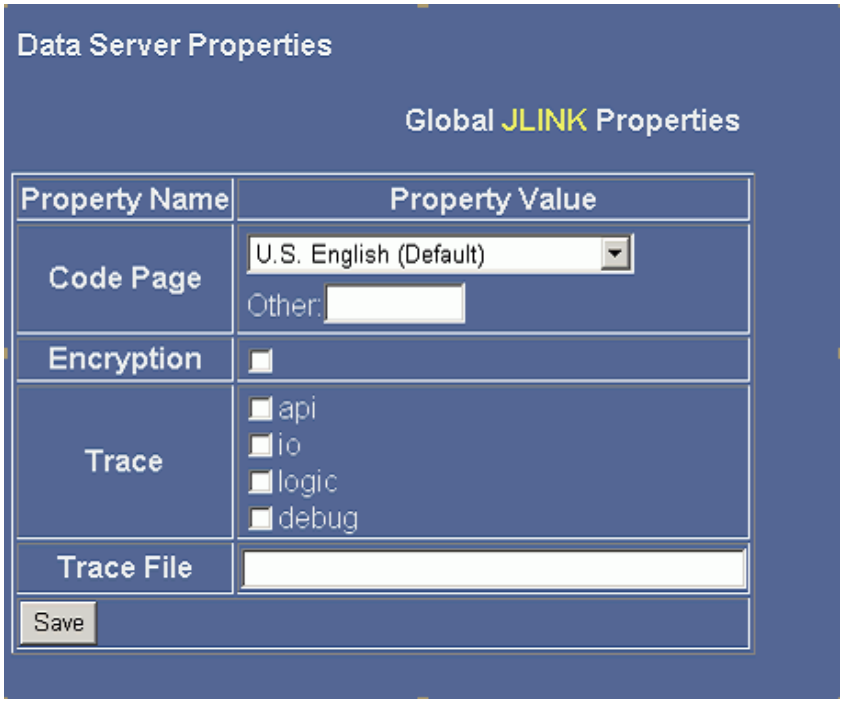
<channelname>

Is the name of the channel.

Server-wide deferred tracing is affected only when a channel is initialized. When enabled on a per-channel basis through the set command, the deferred mode activates only following the start of the next message through the channel execution thread. That is, it may take one or two messages before the mode is recognized.

Jlink Debug

A separate category called jlink debug is used to mask off trace messages arising in the iWay JDBC driver used to access the main data server. Actual tracing levels for all instance of the driver are specified in the driver settings in the Data Server Properties configuration page.



You can also specify trace levels for specific instances of the driver. The trace levels are:

- api. Causes entry/exit tracing as the application steps through JDBC calls.
- io. Traces data in and out of the system.
- logic. Traces internal activity of the driver. Equivalent to the DEBUG level of the server.
- debug. Traces internal operations of the driver. Equivalent to DEEP level of the server.

The trace file specification is an entry that enables you to route traces from the iWay JDBC driver to a specific file. If this is not specified, the traces (in most cases) appear in the standard server trace. Some more generalized services that use the iWay JDBC driver do not pass traces through the server. In these cases, specification of the external trace file enables the traces to be captured. You will be asked to send this file to iWay as part of problem resolution.

Running in a Command Shell

As a general rule, the server runs as a "service" as appropriate for the platform. For example, under Microsoft Windows, this is a Windows Service.

It is possible to run the server under the command shell. To do so, you need access to the shell. The location containing the server software (usually the bin directory under the installation point) must be in the execution path of the operating system.

To start the server, enter the following command:

```
iwsrv <configname>
```

The command is documented in other manuals and by entering the following command directly at the Windows command line:

```
iwsrv ?
```

A batch startup script is supplied with the installation for Windows and for Unix/ZOS platforms. These are named iway7.cmd and iway7.sh respectively. They support the same parameters.

Commands can also be sent using the secure telnet command channel. To enable this channel, configure a TelnetD channel on the master configuration (or any other configuration). Only the inlet/listener component matters. The remainder of the configuration is ignored. Once configured, any standard Telnet client can be used to communicate command and control to the server.

If the server is running as a service with no console access (e.g. Windows), iWay recommends configuration of a TelnetD channel to provide command, control, and analysis services.

When running in the command shell, you control the server by typed commands in addition to the run time console. These commands are designed to assist in resolving issues, and many are technical in nature. Some of the key commands are:

Command	Description
copy <from> <to>	Copies a file between locations.

Command	Description
errors	Displays the last few errors reported by the server.
info	Displays monitor information.
line	Draws a line. This makes reading of the display simpler.
manifest <jarname>	Displays the manifest of the named jar file. The name can be a full path name or a simple jar name in the classpath.
memory	Displays statistics on memory in use.
quit	Exits the server. All listeners must have been stopped.
refresh <listener name>	Restarts the listener with an updated local configuration. The listener should be started.
start [<listener name>]	Starts all listeners or the identified listener.
stats	Displays statistics. For standard operation, this shows a wall clock (response) time figure. iWay provides a measurement that extends the functionality of this command.
stop [<listener name>]	Stops all listeners or the identified listener.
threads	Lists execution “threads” currently being controlled by the server. This may not include threads started by auxiliary packages such as third party interfaces. This is often very useful after a STOP command to determine what might still be running. Options on the threads command enable you to write a current JVM status (stacktrace), track internal monitors (locks), and so on. These options can help to diagnose what might appear to be a "loop" or "work stoppage".
time	Displays the time in GMT.
tool <toolname> [parameters]	Runs the specified tool. See *Supplied Tools* on page 219.

The `show` command can display information regarding key system resources and the environment of the server. You may need to use some of the `show` commands depending on the situation being investigated. While the `show` command changes for each release, some of the most useful subcommands for debugging are listed and described in the following table:

Command	Description
show classpath	The JAR files contributing to the environment of the server are shown. This information is also available from the console. Some systems restrict classpath length, and this will be apparent if the classpath seems truncated.
show configs	Information about the operating status of configuration/ applications is displayed. This can quickly show whether a configuration is running.
show extensions	Displays information about the extensions loaded by the server. Extensions usually contain components of the system, for example, a channel protocol such as Sonic. If the extension did not load properly, you may see runtime startup errors. The reason that the extension did not load is usually shown in the listing.
show pools	Displays the contents of key internal resource pools.
show providers	Providers control resources, such as keystores or data base access connections. When problems in the resources controlled by the providers appear, this command will display usage statistics that may help isolate issues.
show queue	Displays information about the queues by protocol or channel name. All queuing systems are not necessarily supported. You can use this information to understand how many messages are on the queue, error counts, number processed so far, and so on.
show sregs (or show registers)	Displays manager and listener-level special registers and their values. Registers that are created as part of a document workflow, such as the source name of a document read by the file listener, are not shown.

Command	Description
show xalog	Displays information related to the transaction log update drivers. BAM (Business Activity Monitor) uses a transaction log driver. If issues arise in BAM or other transaction log, this can help isolate errors.

Console command help is available through use of the `help` command at the console. Entering the following command shows a list of the currently supported command:

```
help
```

Be aware that the commands (especially the informational commands) may vary with the release.

Entering the following commands provides additional information where available and appropriate:

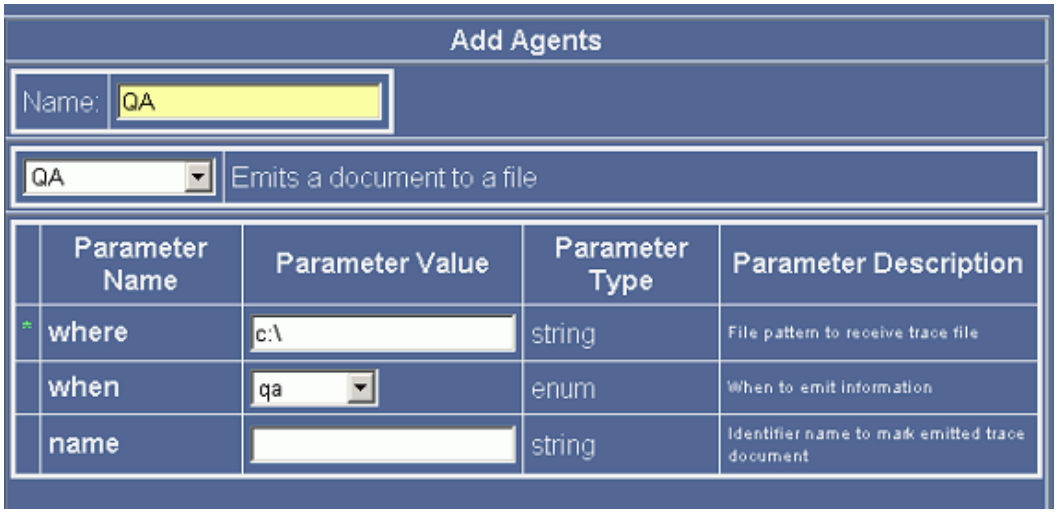
- `help <commandname>`
- `help <commandname><parameter>`

Quality Assurance Mode

QA Mode is a capability of the server supporting testing and quality assurance use. In QA mode, all dates emitted to documents by the server are replaced with a constant value. This facilitates comparing documents between runs of the server. The special register QA is set to "1", which can be tested in conditional execution statements, and the QAAgent can be configured to emit output in QA mode only.

QA Mode is set in the Configuration System Properties Diagnostics page of the configuration console.

The QAAgent is applied to a flow simply by adding it into the agent stack:



In process flows, the QA Agent is added to the flow by the debug icon on the toolbar.

The agent creates a file containing information about the state of the system when it is entered. You can elect to have the agent emit always or only in QA mode. The information is written as a file to the "where" location using standard file pattern naming. For example, in Windows: c:\qafiles\qa###.fil will cause files such as qa001.fil, qa002.fil, and so on to be written to directory c:\qafiles.

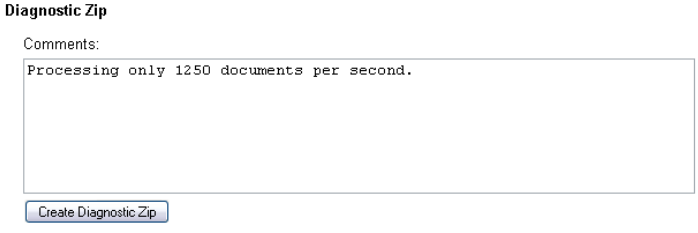
The information written includes the names and values of all defined special registers (variables), the document as it is at that point in the flow, and any attachments that exist on the document.

A sample of the emitter output for a very simple document is shown. The name set in the configuration is *debug*.

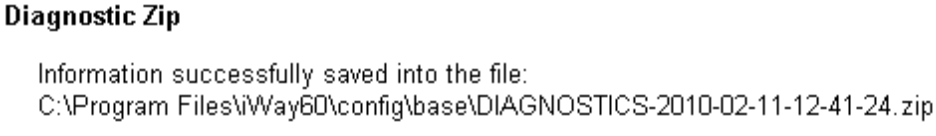
```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<debug>
   <document>
      <edxax errors="0">
         <dest>user1@mail.com</dest>
         <request reqtag="NT04" test="018">
            <connection cnctag="CNC1">
               <dsn bar="something">LOCALSR1</dsn>
               <user>edardb</user>
               <sql>
                  <query>select model from car </query>
               </sql>
            </connection>
         </request>
      </edxax>
   </document>
   <variables>
      <variable name="basename" type="SYS">xmlone</variable>
      <variable name="engine" type="SYS">base</variable>
      <variable name="name" type="SYS">xmlone.xml</variable>
      <variable name="protocol" type="SYS">FILE</variable>
      <variable name="source" type="SYS">C:\stress\xmlone.xml</variable>
      <variable name="tid" type="DOC">FILE1-FILE-
         W.FILE1.1_20040624170104434Z</variable>
   </variables>
</debug>

MultiPart attachments:
message-id: <2766626.1088096464825.JavaMail.rb02237@Beck2000>content-
type: multipart/mixed; boundary="----
=_Part_0_3472085.1088096464638"content-length: 398mime-version: 1.0------
=_Part_0_3472085.1088096464638

Content-Type: text/plainContent-Transfer-Encoding: 7bitContent-
Disposition: attachment; filename=Attach1Content-Length: 8

attach 1
------=_Part_0_3472085.1088096464638
Content-Type: text/plainContent-Transfer-Encoding: 7bitContent-
Disposition: attachment; filename=Attach2Content-Length: 8

attach 2
------=_Part_0_3472085.1088096464638---
```

Gathering Support Information

You may be asked to send diagnostic information to iWay for support. This task is simplified by the Diagnostics Create Diagnostic Zip function. You are presented with a screen in which you can describe symptoms or include other information you think will assist iWay in resolving the issue. When you run the program to create the traces, you should delete any trace files that are not needed, and set the trace levels to include at least DEEP level. You can include TREE and DATA levels if the documents are small enough; avoid this for huge documents that quickly fill the traces. Generally, fewer trace files, but each large file is easier to work with than many small files. You set the number and rollover size of the trace files.



Once you have explained the problem, click *Create Diagnostic Zip*. A file will be created including the information iWay most needs for analysis. The location of this file is written to the console.



The `diagzip` command can be issued from the console (either with the master console or a connected telnet command console). It is documented under iSM commands. This command can be used to generate a diagnostic zip in cases in which the iSM console is unavailable.

Additional information is available by issuing the following command:

```
help diagzip
```

You should package this file along with other files that may be useful such as input files.

Appendix E

Measuring Performance

This appendix describes how facilities in iWay Service Manager (iSM) can be used to measure performance.

In this appendix:

Performance Measuring Overview

Several facilities are provided in iWay Service Manager (iSM) to assist in measuring execution performance. However, no single tool provided either by iWay Software or an external source will do a complete job of providing a performance analysis. Only some approaches and techniques used frequently by application developers are discussed in this appendix. iWay Software provides tools to assist in such analysis, which are also discussed in this appendix.

The following list of principals apply to any performance measurement, and must be considered in any experimental design.

1. Java is designed to learn as it goes, which means that it has a heuristic optimizer that requires many passes through code to determine the best compilation and execution path. As a result, you must run many setup transactions before actually taking measurements. The initial transactions will be far slower than later transactions, and the measurements of those transactions will include the Java analysis time as the Java Virtual Machine (JVM) gathers information for later optimization.
2. You must run many experiments. One experiment can be configured to deviate from the expected result, so that a wide range of statistical data can be acquired.

3. All extraneous effects must be avoided. For example, running performance tools with tracing enabled will simply measure tracing, which is provided to help debug problems, and will skew any results.
4. You will need some understanding of the dynamics of a JVM, its optimization, and the tools available with the JVM. These are not further discussed in this appendix.

Built-in Statistics

The generation and display options of the measurement package assist in analyzing the behavior of iSM. The built-in statistics record execution and memory heap usage.

Note: Any information and any formats described in this document are release-dependent, and subject to change.

Memory Command

The memory command shows the amount of memory being used at the time the command is issued. The JVM has a heap that represents the runtime data area from which memory is allocated for all memory requirements. It is created during the JVM start-up. Heap memory for objects is reclaimed by an automatic memory management system, which is known as a garbage collector. The garbage collector runs automatically, but the GC command can be used to force it to run for analysis purposes.

The memory display shows the following components:

Field	Description
init	Represents the initial amount of memory (in bytes) that the JVM requests from the operating system for memory management during startup. The JVM may request additional memory from the operating system and may also release memory to the system over time. The value of init may be undefined (0) on some platforms.
used	Represents the amount of memory currently used.
committed	Represents the amount of memory (in bytes) that is guaranteed to be available for use by the JVM. The amount of committed memory may increase or decrease over time. The JVM may release memory to the system, causing committed to be less than init. However, committed will always be greater than or equal to used.

Field	Description

max	Represents the maximum amount of memory (in bytes) that can be used for memory management. Its value may be undefined. The maximum amount of memory may change over time if defined. The amount of used and committed memory will always be less than or equal to max if max is defined. A memory allocation may fail if it attempts to increase the used memory such that:
	`used > committed`
	even if the following would still be true (for example, when the system is low on virtual memory):
	`used <= max`

The heap display is more accurate than the memory display issued without the measurement package installed. The original (standard) information is shown below in addition to the new heap information.

```
Enter command:>memory
STR00X35: memory used 8244K, free 591K,
        nodes: cache 1001 allocated 8607, reclaimed 116, destroyed 116
        namespace 0 namespace reclaim 0
        Heap: init=0K  committed=8244K  max=65088K  used=7662K
Enter command:>
```

The key value is the word *used*, which shows how much is currently allocated. As it approaches *max*, the garbage collector may start, and the performance may be eroded.

Threads Command

The threads command provides the state of the logical threads, which are usually mapped one-for-one to physical operating systems. Threads are shown along with their state.

The following table lists and describes the possible thread states.

Thread State	Description
`NEW`	This state represents a new thread, which is not yet started.

Thread State	Description
RUNNABLE	This state represents a thread which is executing in the underlying JVM. In this case, executing in JVM does not necessarily mean that the thread is always executing in the operating system as well. It may wait for a resource from the operating system, such as the processor, while being in this state.
BLOCKED	This state represents a thread that has been blocked and is waiting for a monitor to enter or re-enter a synchronized block or method. A thread gets into this state after the Object.wait method is called.
WAITING	This state represents a thread in the waiting state, where the wait is over only when some other thread performs some appropriate action. A thread can get into this state either by calling a wait method with no time-out option. The thread depends on the actions of some other thread.
TIMED_WAITING	This state represents a thread which is required to wait at max for a specified time limit. A thread can get into this state by calling a wait method with a time-out option. That is, the thread can run either when another thread completes some action or a clock expires.
TERMINATED	This state represents a thread which has completed its execution either by returning its execution method after completing the execution or by throwing an exception which caused the termination of the thread.

Deadlock Detector

The deadlock detector finds cycles of threads that are in a deadlock and waiting to acquire locks. That is, it finds threads that are blocked waiting to enter or re-enter a synchronization block after a wait call, where each thread owns one lock while trying to obtain another lock already held by another thread.

Usually, a thread is deadlocked if it is part of a cycle in the relation *is waiting for lock owned by*. In the simplest case, thread A is blocked waiting for a lock owned by thread B, and thread B is blocked waiting for a lock owned by thread A.

This is an intrusive operation, and should be used only in cases in which it is suspected that messages are locked up in the system.

To enable the monitoring, use the following command:

```
THREAD MONITOR ON
```

To disable the monitoring, use the following command:

```
THREAD MONITOR OFF
```

While the monitor is enabled, entering the threads command will display information regarding any detected deadlocks, showing the thread name(s) and the lock name(s). The thread names will indicate the components that are deadlocked. Deadlocks on external components, such as a locked data base, are not available.

Performance Measuring Statistics

During the run, statistics with wall clock and CPU time are generated. On the summary page, all times are reported in second precision to four places. It is possible to develop a report for greater precision.

Wall clock times can in some cases show useful information. These times provide a measure of performance for a single message as experienced by the sender. They do not give any information regarding the throughput capacity of the server.

CPU and User times describe the actual execution time expended on messages. Implementation of these measurements is platform and JVM dependent. In many cases the CPU and User times are the same, as the JVM may not differentiate between the two. On those platforms in which the JVM does differ, the CPU time can be expected to be the greater of the two.

User time represents the CPU time that the current thread has executed in User mode, executing Server Manager Instructions.

CPU time is the sum of User time and System time, and includes the time spent setting up for JVM services such as locks, network operations, i/o operations, and so on.

```
Enter command:>stats
                In seconds
  name       count    low      high      mean      variance   std.dev.   ehr num/sec
mq1a  wall:      2   0.0470   0.1560    0.1015    0.0030    0.0545       -      9.85
      cpu :      2   0.0312   0.0625    0.0469    0.0002    0.0156       -
      user:      2   0.0312   0.0625    0.0469    0.0002    0.0156       -
```

The STATS command displays the statistics summary gathered to that point. To reset to zero, use STATS RESET. iWay recommends not trusting statistics until at least several messages have been handled to completion, as the server front-loads initialization. Once the system is in a steady state, you can reset the statistics to zero.

Users are cautioned that the numbers shown on the summary page are approximate and intended for general guidance only. Brief descriptions of the fields are provided in the following table. However, a complete understanding of the message processing distributions described by these figures requires some knowledge of statistics and probability as it applies to queueing.

Field	Description
count	The number of messages handled for which statistics have been gathered.
low	The lowest time recorded for handling a message.
high	The highest time recorded for handling a message.
mean	The numeric mean of the times recorded. This is the sum of the times divided by the number of messages handled. This is also called the average.
variance	The statistical variance of the times recorded. Variance is a measure of how numbers disburse around the mean.
std. dev.	The statistical standard deviation of the times recorded. Standard deviation is a measure of how numbers disburse around the mean.
ehr	The Ehrlang Density Coefficient. This gives evidence of the randomness of the time distribution. If there are too few values to compute the coefficient, a dash (-) is displayed. If the coefficient is sufficiently close to constant, the term *const* is shown. This value is an approximation. A value of 1.0 indicates a Poisson distribution, which is the design point of the server. A very low value can indicate that the individual times recorded are wildly skewed and therefore less usable to predict behavior.

Field	Description
num/sec	The reciprocal of the mean giving the number of messages handled per second. This is shown for the wall time, and is not directly a measure of the throughput capacity of the server.

The iSM Administration Console can also display the summary of statistics. The monitor/statistics page provides a table to view this data, as shown in the following image.

Listener Name	Listener Type	Completed	Time clock	Mean (sec)	Std. Dev. (sec)	Variance (sec)	Ehrlang	Num/Sec
filein	FILE	1	Wall	0.1250	0.0000	0.0000	const	8.00
			CPU	0.1094	0.0000	0.0000	const	9.14
			User	0.0938	0.0000	0.0000	const	10.67
filelock	FILE	0	Wall	0.0000	0.0000	0.0000		
			CPU	0.0000	0.0000	0.0000		
			User	0.0000	0.0000	0.0000		

Tips:

- When working with the complexity of a document (a number greater than -1 in the complexity field), a good rule of thumb is to use the number of digits in the field. For example, a message of 172 nodes would get a complexity measure of 3, while one of 1459 would get a measure of 4. For most purposes, this provides a reasonable value for analysis purposes.
- Traces consume time and memory in the server. For valid statistics, disable all trace. You can use the set command from the console to do this, or use the console configuration.

Additional Information:

For more information on queuing theory, the use of the available statistics, and the interpretation of the presented fields, many books are available.

Kushner, Harold J.; *Heavy Traffic Analysis of Controlled Queueing and Communications Networks*. New York, Springer; (June 8, 2001).

Using iFL to Measure Multiple Node Duration

iWay Functional Language (iFL) provides the _now('Un') function, which is used to record time. The pattern 'Un' causes the value of the best available timer to be recorded. The time recorded is not absolute, and cannot necessarily be related to any external time. Instead, all that is guaranteed are the time increments and the best available timer.

A frequent use of this function is in the preamble and postamble of the process flow node. You might use the preamble of the first node of a sequence and the postamble of the last node of the sequence to measure several nodes together. Normally, these opportunities are used to set registers. By setting a preamble register, such as start to _now('Un') and a postamble register, such as totaltime to _sub(_now('Un'), _sreg(start)), the registered total time will hold the measured time duration across the node. You might emit this time using the Trace Message Writer Service (com.ibi.agents.XDTraceAgent), write it using a File emitter, or even include it in an output document.

Activity Log Timer

Each activity in iWay Service Manager (iSM) is reported to the Activity Log Timer. This facility is configured in the iSM Administration Console, and you can select one or more drivers depending upon the tasks to be performed. As a measurement and analysis aid, an activity facility offers a driver Time Event Logger for Performance Measurement. This driver records raw times of events, using minimal overhead. iWay Software does not recommend this driver for use in production environments. However, in development environments, it can provide significant insight into system activity.

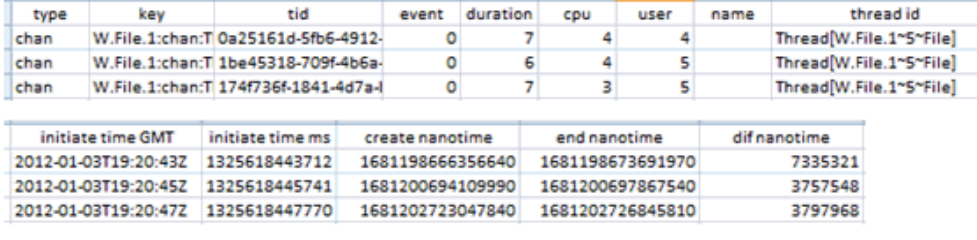
When configuring the driver, you must set the driver to active. When running the driver, statistics are written to an external file in a delimited form. This file is updated infrequently, As for performance reasons, the statistics are maintained in memory. The final statistics are flushed to the file when the QUIT command is issued. Therefore, you must terminate the server run with the STOP and QUIT sequence. Any other means of terminating the server will result in loss of data.

The following table lists and describes the fields that are used to configure the driver.

Field	Description
Output File	Statistics are written to this file. You may use standard iWay file naming conventions such as /directory/stats###.txt which will cause a file named stats001.txt to be written for the first server start, stats002.txt for the second, and so on. You may use # (count) and * (time) patterns for the file naming.
Skip	Number of messages to skip before recording starts. Use this value to allow the optimizer of the JVM to load and operate before recording statistics. A value of 1000 will allow about 1000 messages to be ignored before statistics gathering begins.
Delimiter	Choose whether a comma or a tab character should be used for the field delimiter.
Record Events	Choose whether or not detailed events should be recorded. Message events in the channel give overall statistics for the message execution: time, cpu, and so on. Detailed events report the time of each process flow node, transform, and so on. Depending upon what is being measured, you can determine whether detailed events are required.

The file can be loaded into a spreadsheet program, such as Microsoft Excel.

The following image shows how an Excel spreadsheet of channel (non-event) activity might appear.



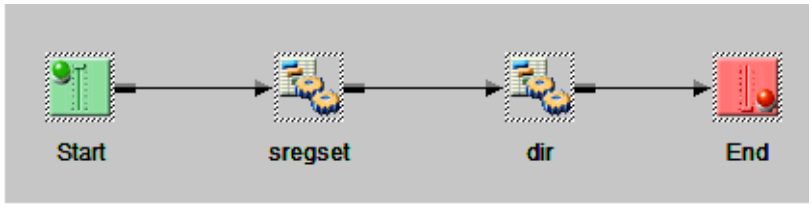
type	key	tid	event	duration	cpu	user	name	thread id
chan	W.File.1:chan:T	0a25161d-5fb6-4912-	0	7	4	4		Thread[W.File.1~5~File]
chan	W.File.1:chan:T	1be45318-709f-4b6a-	0	6	4	5		Thread[W.File.1~5~File]
chan	W.File.1:chan:T	174f736f-1841-4d7a-l	0	7	3	5		Thread[W.File.1~5~File]

initiate time GMT	initiate time ms	create nanotime	end nanotime	dif nanotime
2012-01-03T19:20:43Z	1325618443712	1681198666356640	1681198673691970	7335321
2012-01-03T19:20:45Z	1325618445741	1681200694109990	1681200697867540	3757548
2012-01-03T19:20:47Z	1325618447770	1681202723047840	1681202726845810	3797968

The following table lists and describes the fields in the spreadsheet.

Field	Description
Type	CHAN or EVENT. CHAN rows five durations for a message, while EVENT rows record for each event within the server.
Key	Used by the statistics program to manage recording. Usually not useful for analysis, but it is provided for use in keeping long term records
TID	The transaction ID of the message. A message is assigned a TID when it is first recognized by the server. The TID follows the message throughout its life.
Event	The event code for EVENT records.
Duration	The time in ms taken by the message or event.
CPU	The CPU time associated with the message or event.
User	The user time (includes I/O) associated with the message or event.
Thread ID	The thread name associated with the message or event. While keys are unique, threads are often reused.
Initiate Time GMT	The GMT time that the message or event began.
Initiate Time ms	The time in UNIX format (ms past Jan. 1, 1970).
Create Nanotime	The time at receipt of the initiation event recorded by the best available clock.
End Nanotime	The time at receipt of the end event recorded by the best available clock.
Dif nanotime	The difference between the end and start nanotime.

The following image shows a simple process flow that is published to a File listener.



Two documents are run. The stats command provides the following result:

```
File
          wall:      3    0.0000    0.2510    0.0890    0.0132    0.1147   6.76   11.24
          cpu :           0.0000    0.0156    0.0052    0.0001    0.0074 const
          user:           0.0000    0.0156    0.0052    0.0001    0.0074 const
```

The following image shows the spreadsheet that is generated by the activity driver when events are enabled.

type	key	tid	event	duration	cpu	user	name	thread id	initiate ti	initiate time ms	create nanotime	end nanotime	dif nanotime
evnt	W.File.1:(	fa7ef18a-47(	2	0	0	0	Parse	Thread[W	2012-01-(	1325695361244	1758114748788250	1758114749155110	366862
evnt	W.File.1:(	fa7ef18a-47(	7	0	0	0	Agent	Thread[W	2012-01-(	1325695361275	1758114781628010	1758114782121520	493512
evnt	W.File.1:(	fa7ef18a-47(	7	0.032	0.0156001	0.0156	Agent [sregset]	Thread[W	2012-01-(	1325695361275	1758114782325930	1758114820249020	37923090
chan	W.File.1:(	fa7ef18a-47(	0	0.078	0.0624004	0.0468		Thread[W	2012-01-(	1325695361244	1758114748288970	1758114826410230	78121265
evnt	W.File.1:(	0d511c6b-35	2	0	0	0	Parse	Thread[W	2012-01-(	1325695363289	1758116789190070	1758116794749980	5559908
evnt	W.File.1:(	0d511c6b-35	7	0	0	0	Agent	Thread[W	2012-01-(	1325695363289	1758116796929980	1758116797159410	229433
evnt	W.File.1:(	0d511c6b-35	7	0.016	0.0156001	0	Agent [sregset]	Thread[W	2012-01-(	1325695363289	1758116797321480	1758116804300710	6979238
chan	W.File.1:(	0d511c6b-35	0	0.016	0.0156001	0		Thread[W	2012-01-(	1325695363289	1758116788276190	1758116807184410	18908228
evnt	W.File.1:(	afa024c6-9f(	2	0	0	0	Parse	Thread[W	2012-01-(	1325695367330	1758120830232840	1758120830510390	277552
evnt	W.File.1:(	afa024c6-9f(	7	0	0	0	Agent	Thread[W	2012-01-(	1325695367330	1758120832118350	1758120832340850	222504
evnt	W.File.1:(	afa024c6-9f(	7	0	0	0	Agent [sregset]	Thread[W	2012-01-(	1325695367330	1758120832497530	1758120839019440	6521911
chan	W.File.1:(	afa024c6-9f(	0	0	0	0		Thread[W	2012-01-(	1325695367330	1758120830006480	1758120841793040	11786558

Note: Values that are close to zero are shown as zero.

Appendix F

iWay Functional Language

This iFL Grammar represents the allowed expressions in the language. The rules of the language are strict, and this appendix describes these rules. iFL is intended to enable expressions to be used for tests and configuration parameters, which drives the language design. This appendix assumes that readers are generally familiar with the iFL language.

In this appendix:

Functional Language Rules

```
Expr::= ExprOR | ExprOR "," Expr | ExprOR Expr
ExprOR::= ExprAND | ExprAND "OR" ExprOR
ExprAND::= ExprCompare | ExprCompare "AND" ExprAND
ExprCompare::= ExprPlusMinus | ExprPlusMinus CompareOp ExprCompare
ExprPlusMinus::= ExprMultDiv | ExprMultDiv PlusMinusOp ExprPlusMinus
ExprMultDiv::= ExprNot | ExprNot MultDivOp ExprMultDiv
ExprNot::== ExprFundamental | "!" ExprFundamental
ExprFundamental::= Literal | "(" ExprInParens | FuncCall
ExprInParens::= ExprOR | ExprOR "," ExprInParens | ExprOR ExprInParens
FuncCall::== Func ParmList
ParmList::= ")" | ExprParm ")" | ExprParm "," ParmList | "," ParmList
ExprParm::= ExprOR | ExprOR ExprParm
CompareOp::= "<" | ">" | "=" | "<=" | ">=" | "=="
PlusMinusOp::= "+" | "-"
MultDivOp::= "*" | "%"
Func::= Identifier "("
Literal::= QuotedLiteral | BareToken
QuotedLiteral::= SingleQuoted | DoubleQuoted
SingleQuoted::= SQuote SingleQuotedChars
DoubleQuoted::= DQuote DoubleQuotedChars
SQuote::= '
DQuote::= "
Escape::= \
SingleQuotedChars::= SQuote | PlainChar SingleQuotedChars| DQuote
SingleQuotedChars | EscapedChar SingleQuotedChars
DoubleQuotedChars::= DQuote | PlainChar DoubleQuotedChars| SQuote
DoubleQuotedChars | EscapedChar DoubleQuotedChars
PlainChar::= AnyChar except SQuote, DQuote, Escape
EscapedChar::= Escape AnyChar
BareToken::= Sequence of unquoted characters containing no recognized
special characters nor keywords
```

The iFL language has implicit concatenation of adjacent expressions, at top level, within parentheses and within a function parameter. Adjacent literals are concatenated at compile time to form a single value.

Spaces around operators are ignored. Spaces after comma are ignored. At top level and within parentheses, a comma is a literal comma, except it also ignores spaces to the right.

A parameter might be empty in a parameter list, it means empty literal `""`, e.g. `fn(,a,,b,)` is the same as `fn("",a,"",b,"")`

The escape character is an operator only within quoted literals, this is because unquoted Windows paths make use of the backslash as a component separator. For the same reason, there is no math division operator; standard path names use the forward slash as a component separator. Use `_div()` or `_idiv()` instead.

Functions are recognized even if the function identifier is immediately preceded with other characters, e.g. `abcsreg(reg)` is `"abc"` followed by sreg(reg). There is no space allowed between a function identifier and the open parenthesis.

The named operators *AND* and *OR* must be complete identifiers to be recognized unlike functions, e.g. `(1<2)AND""2<3` is the AND operator, but `1<2AND2<3` is not. This is the exception to the automatic concatenation rule.

Function names and named operators are case-insensitive and follow the Java Identifier rules. All built-in functions start with an underscore by convention. A handful of legacy functions are also available without the underscore. No additional functions will be made available in this form.

Minus is always a binary operator, this is to disambiguate implicit concatenation. You can obtain unary minus by quoting it or writing `0-expr`.

All binary operators are left-associative, `A op B op C` is `((A op B) op C)`. Comparison operators are also associative. For example, the expression `"1 < 2 < 3"` is the same as `"true < 3"` because clearly one is less than two. The binding is from left to right.

The following is the binding precedence from tightest to loosest:

1. Explicit grouping: ()
2. Multiplicative: *, % i.e. multiply, modulo
3. Additive: +, -
4. Comparison: <, >, =, <=, >=, ==
5. Conjunction: AND
6. Disjunction: OR

7. Concatenation: implicit or ,

Appendix G

Creating Application Consoles for iWay Service Manager

This appendix describes how to create application consoles for iWay Service Manager (iSM).

In this appendix:

Application Console Overview

iWay Service Manager (iSM) can be used as a platform for redistributable applications that are developed around pre-configured channels. In these cases, developers may need to expose some configuration options to the user, but without the full power of the iSM Administration Console. Two common scenarios are seen; either maintaining a completely separate resource for an application or providing a customized configuration of the server itself.

It is not intended that the server be a fully-fledged servlet engine. There are many such engines for that purpose, such as Apache Tomcat.

The separate resource scenario is a subset of the effort in developing the customized configuration application. The tutorial in this appendix will address that situation.

Consider the following scenario, which this tutorial is based upon:

A company might want develop an application that picks up purchase orders and sends them to its B2B hub in the proper format. When the company distributes this application to its partners, the partner should be able to specify the directory where the application should look for purchase orders, but unable to change other details of the channel configuration or have the ability to create new channels.

One solution for the developer would be to configure the iSM application using the _PROPERTY() runtime function. This function looks up a value from a standard Java properties file and then can be used in any context where runtime functions are evaluated. For example, you could set the input directory parameter of the File channel to:

```
_PROPERTY(my.properties, fc.inputdir, somedefault)
```

A properties file (for example, my.properties) can then be created using the following format:

```
fc.inputdir=c:/input
```

The partner or end-user can now edit the properties file to set the parameters that the developer wanted to expose without ever viewing the iSM Administration Console or modifying any core iSM configuration files. In simple applications, this solution may be sufficient. However, when more complex functionality is required (for example, to include monitoring capabilities or to provide a graphical user interface for configuration), a developer can create application consoles.

Application consoles use standard iSM components and development techniques to create an application-specific graphical user interface for configuration and monitoring. An application console is an HTTP channel with routes that can perform the required tasks and output HTML pages with menus and information for the user. There are many ways to design a channel like this, depending on the application requirements. This appendix describes some basic techniques and provides a sample application console, which is referred to as the red console.

Basic Application Console Techniques

The following sections describe basic techniques that can be considered when creating application consoles for iWay Service Manager.

Using the _PROPERTY() Function

The _PROPERTY() function takes three arguments:

1. The path to a properties file. In most cases, it is a best practice to store this value in a special register (SREG).

   Note: The standard special register, `iwayworkdir`, locates the configuration directory. This is a convenient place to store a property file. For example:

   ```
   sreg('iwayworkdir')/my.properties
   ```

2. The name of the required property.
3. A default value to return if the property cannot be resolved.

The properties file uses the standard format for Java properties, `key=value`, and may include comment lines beginning with `#` characters.

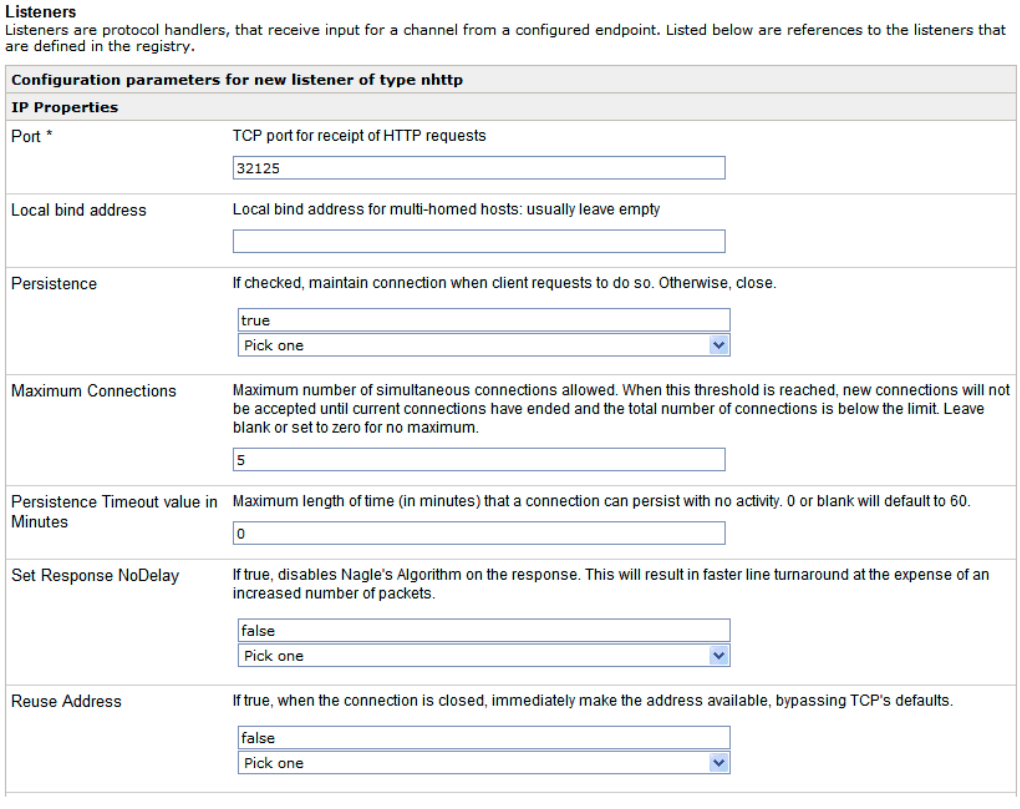
The properties file is loaded into memory when the _PROPERTY() function is first invoked. At each subsequent invocation, the last modified time of the properties file is checked. If the file has been modified since the previous invocation, the properties in memory are refreshed from the file.

Configuring an NHTTP Listener for Use With an Application Console

The NHTTP listener provides several parameters that can be useful when developing application consoles. The following table lists and describes these parameters.

Parameter	Description
Allowable Clients	Access to the NHTTP listener is restricted only to the host names and IP addresses in the supplied list.
SSL Client Authentication	When set to *true*, only clients with certificates in the trust store of the listener (for example, the trust store for the SSL Context Provider used by this listener) are allowed to connect.
GET Handling	When set to *event*, HTTP GET requests create an XML event document that is sent to the process flow. This can be useful during the development of a dynamic application console. For more information, see *Sample Application Console (Red Console)* on page 258.
Response Content Type	You must set the response content type accordingly since the application console often returns HTML documents.

The following image shows the Configuration pane of an NHTTP listener in the iSM Administration Console.



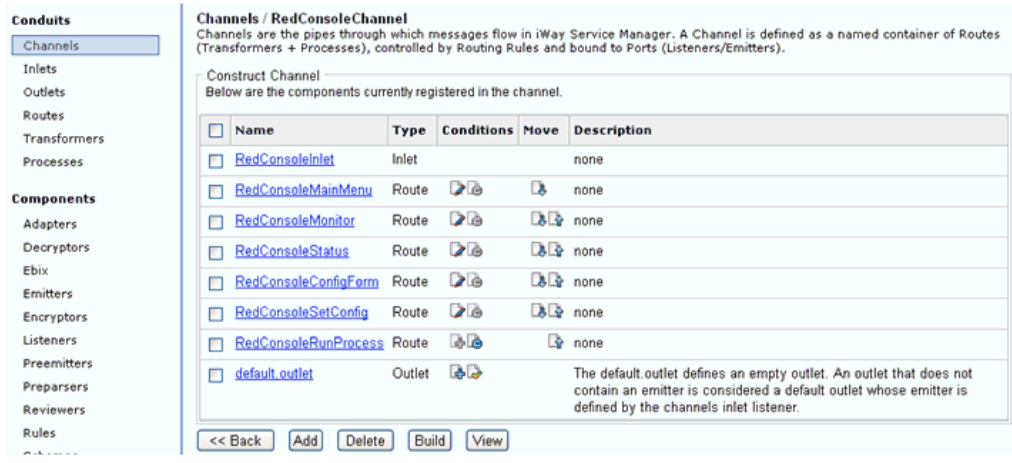
Configuring Conditional Routes

The behavior of an application console channel is determined by the URL of an incoming request. As defined by the GET Handling parameter for the NHTTP listener, the elements of the URL are included in a GET event document. In theory, it would be possible to handle all possible events in a single, complex process flow. However, in most cases, it is simpler and more extensible to use several simple processes and conditional routing. This section provides an overview of how to configure a conditional route.

1. Using the iSM Administration console, configure inlets, outlets, and routes as required.

   For more information on how to create and configure channels in iSM, see the *iWay Service Manager User's Guide*.

2. Create a channel with your inlet and outlet, and then add a route.



3. In the Conditions column, click the small icon with a blue plus sign next to the route that you added.

   The Set Condition pane opens, as shown in the following image.

4. Enter your condition using iSM runtime function syntax.
5. Click *Update* when you are finished.

   You are returned to the previous Construct Channel pane. Notice that the small icon with a blue plus sign has changed to a small blue pencil, which indicates that a condition has been specified.

6. To set the route as a default route, click the small gray icon next to the plus sign/pencil condition icon.

When you have set the route as the default, the icon changes to blue.

Sample Application Console (Red Console)

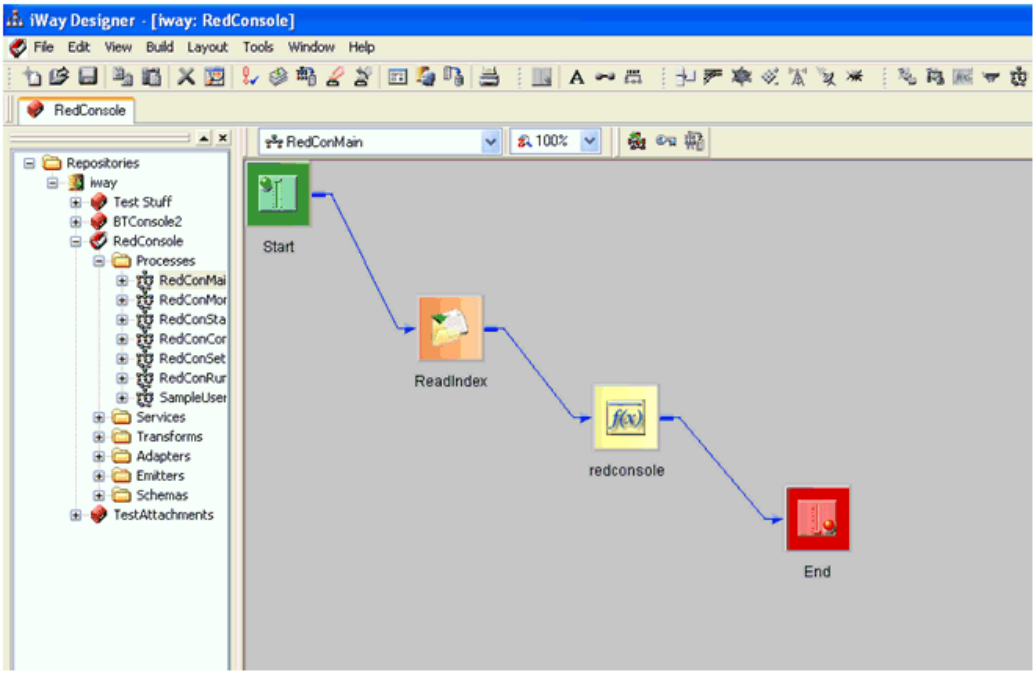
The following section describes a sample application console, which is referred to as the red console.

The red console consists of a channel with an NHTTP listener as the inlet, several conditional routes, and a default outlet. The listener is configured to handle GET requests by creating an event message that contains the context of the request, including the path requested by the URL. All but one of the routes is conditional on an XPath expression that evaluates the URL path. Each of these routes is associated with a process flow that generates a console screen by executing one or more agents and transformations. The other route is run by default and tries to run a process flow whose name matches the request path.

For example, a GET request to the NHTTP listener on `http://myhost:3125` would create this message. The router would evaluate the expression `XPATH(/http/url/path)`, find "/", and call the main menu process.

```
<http user="unknown" type="GET">
   <parms>
      <parm name="version">1.1</parm>
      <parm name="source">localhost</parm>
      <parm name="rcReq.Host">localhost:32125</parm>
      <parm name="rcReq.Cache-Control">max-age=0</parm>
      <parm name="rcReq.User-Agent">Mozilla/5.0 Windows; …</parm>
      <parm name="reqType">GET</parm>
      <parm name="rcReq.Connection">close</parm>
      <parm name="rcReq.Referer">http://localhost:32125/</parm>
      <parm name="pdm">0</parm>
      <parm name="rcReq.Accept">text/xml,application/xml,application/
         xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5</
parm>
      <parm name="action"/>
      <parm name="rcReq.Accept-Encoding">gzip,deflate</parm>
      <parm name="rcReq.Accept-Charset">ISO-8859-1,utf-8;q=0.7,*;q=0.7</
parm>
      <parm name="rcReq.Accept-Language">en-us,en;q=0.5</parm>
      <parm name="url">/</parm>
      <parm name="ip">127.0.0.1</parm>
   </parms>
   <body/>
   <url secure="false">
      <host>localhost</host>
      <port>32125</port>
      <path>/</path>
      <query/>
   </url>
   <version>1.1</version>
</http>
```

The main menu process reads an XML file containing menu options and welcome text. An XSLT stylesheet transforms this document to HTML before returning the main menu page to the user.



In addition to the channel package, the application console uses three external files. Since components in the packaged channel reference these files using relative paths, these files must be in the correct locations with respect to the iSM configuration in which the channel will run. For reference, assume that the channel is deployed to myconfig, at the absolute path /iwayhome/config/myconfig.

The myconfig directory contains the properties file, userparms.properties. This contains the names and current values of any properties you want to expose to the end user. For example:

```
name=steve
port=2200
```

These properties can be referenced in the channel configurations using the _PROPERTY() function.

The myconfig/redconsole directory contains the index.xml file, which lists the menu items needed to include on the console pages and the text for the main menu page. For example:

```
<redconsole style="redconsole">
  <menu>
    <item name="Configuration" href="/configuration"/>
    <item name="Monitor" href="/monitor"/>
    <item name="Status" href="/status"/>
  </menu>
  <text>Welcome to the Main Menu.</text>
</redconsole>
```

The myconfig/redconsole directory also contains the propsmeta.xml file, which contains metadata for properties needed to be exposed. For example:

```
<properties>
  <group name="general">
    <parm name="name" type="string"/>
    <parm name="foo" type="int"/>
    <parm name="other" type="boolean"/>
  </group>
  <group name="security">
    <parm name="sslcertalias" type="string"/>
  </group>
  <group name="http">
    <parm name="port" type="int"/>
  </group>
</properties>
```

The red console channel also uses one custom agent (service), *RedConsoleSetPropsAgent*. This agent updates the properties file given an NHTTP event document for a GET request with properties and values in the query string.

Configuring the Red Console Demo Channel

This section describes how to configure the red console demo channel and how to add an item to the channel menu.

Procedure: How to Configure the Red Console Demo Channel

To configure the red console demo channel:

1. Add the rcagent.jar file to the classpath of the iSM configuration to make the RedConsoleSetPropsAgent service available.
2. Unzip the redconsole.zip archive to the root directory of the configuration to which you are going to add the red console, for example:

   ```
   /iWay7/config/myconfig/redconsole
   ```

3. Import the channel archive into your registry using the iSM console.
4. Edit any channel properties you want to change, for example, the NHTTP port of the listener.

5. Deploy the red console channel from the registry to the configuration where you copied the property files.

Procedure: How to Add an Item to the Menu

To add an item to the menu:

1. In redconsole/index.xml, add another item to the menu, and specify the name and HREF.
2. Create a route that does what you want the menu item to do.

   Be aware of the content type you are returning. By default, the red console channel returns text/html. If you want another content type, set the `worker:resp.ct` special register in your process. The register needs the `worker:` prefix to make it available outside of the process flow.
3. Add the route to the red console channel with the following condition:

   ```
   XPATH(/http/url/path) == /path/from/URI
   ```

   This is the HREF from the index.xml file.

   Note: Since the default behavior of the red console channel is to run a process with the name specified in the URI, an alternative method would be to simply add the process to the menu as described above with `href='/myflow'`. You can then publish the process flow directly to the configuration as a system process. This avoids directly modifying the registry. However, performance is slightly slower, since it runs the process from within another process.
4. Rebuild and redeploy the channel.

   Notes on look and feel:

   1. The included processes follow the same pattern:
      a. Read the index.xml file.
      b. Run a service that generates XML.
      c. Join the index.xml file to the other XML.
      d. Run an XSLT transform that reformats the join output (mergeindex.xslt).
      e. Run a final XSLT to generate HTML output (redconsole.xslt).
   2. To change the look of the existing pages, edit the redconsole.xslt file.

      You will have to republish and rebuild the channel as a result of this step.
   3. If you want to add pages with a consistent look, edit the mergeindex.xslt file to handle the new input.

It may also be necessary to edit the redconsole.xslt file to handle the display of the new page.

Glossary

You can refer to this glossary for definitions and helpful information pertaining to terms presented in this manual. This glossary may also describe terms not used in this manual, but that pertain to iWay in general.

CACerts

A Java construct that is not part of any formal security infrastructure. This file, distributed with Java, contains trusted public certificates of known Certificate Authorities. There is, however, no formal validity to CACerts, and it use is limited in secure situations. iWay recommends that CACerts not be relied upon as delivered with Java for a secure installation; rather a valid trust store should be constructed. You can use tools to remove unwanted entries and add wanted entries to this file, making it an appropriate for use as a trust store.

Condition

An expression in iWay Functional Language that resolves to 'true' or 'false'. A condition is used to select a route for an incoming message. An example of a conditional expression is

_isroot('edi') or _root() = 'edi'

either of which selects evaluates to 'true' if the root of the current XML document is edi.

Console Command

An statement issued in the shell console or a Telnet management console that displays or sets the state of some aspect of the server.

Certificate Store

Often called a certstore. A database of public key certificates and certificate revocation elements. A revocation item makes the identified certificate unavailable for new use, often this is due to expiration date passage; the certificate is not eliminated from the store because it may be necessary to check that the revocation did not occur during the time in which the certificate was still valid, or that the certificate was not valid at the time of the message.

Decryptor

A service point of an inlet to use a security encryption system. An encryptor renders a message suitable for preparsing.

Designer

A tool provided by iWay to define and test processes.

Document

A document holds the message information as it passes through the stages of the Server. A document also carries contextual information to assist in processing the message.

Default Route

The message process route used if no explicit route is selected. Routes are selected by conditional tests associated with the route. A default route has no associated condition.

Dynamic Route

A routing method for a channel that reevaluates the available routes at each step of the message process. The routing can therefore respond to changes in the message and execution environment. This method is in contrast to fixed routing in which a route, once selected, remains in force for the duration of the processing of that message.

Emitter

The final service point of an outlet. An emitter physically sends the message to the addressed recipient.

Encryptor

A service point of an outlet to use a security encryption system.

Expression

IA statement in iWay functional language that resolves to a value. An expression can be used to provide a configuration value or control a process decision or switch. An example of an expression is xpath(/a/b) which extracts the value of the <b> node under the <a> root in the document.

Fixed Route

A route in a channel that once selected remains in force for the duration of the processing of that message.

Inlet

A component of a channel responsible for receiving messages.

Key Store

A database of key material. Key material is used for a variety of purposes, including authentication and data integrity. There are various types of keystores available, including "PKCS12" and Sun's "JKS." Some keystores can contain both encryption keys and security certificates. Formally, however, a keystore holds the private key for one or more PKI key pairs.

Listener

A component of an inlet that receives the message from the sender.

Message

A message is information in the form that it is received from or sent to the outside world.

Outlet

A component of a channel responsible for emitting messages.

Preemitter

A service point on an outlet to convert messages from document form to transport format for emitting.

Preparser

A service point of an inlet to convert messages to documents for processing.

Principal

A container of authority that contains roles. Principals are created when a user logs into a protocol that supports logons. Information in the principal is available to process flows in the test nodes

Process

A service point in the lifecycle of a document that applies business operations to the document.

Provider

A runtime component that offers centralized access to resources such as connections, keystores, namespace maps, etc. Other components access the services of a provider by its assigned name.

Route

A component of a channel responsible for passing documents through the defined processing steps. Routes are selected for execution by evaluation of the conditional expressions associated with the route. By this means routes can be associated with one or more specific classes of documents. Any boolean expression can be used to select the route for execution. The channel's default route has no associated condition, and handles documents not associated with a specific route. Routing decisions are re-evaluated for each document arriving on the channel. Routing for a channel can be fixed in which case once selected for a message the route does not change or dynamic which reevaluates routing decisions as the message progresses through execution.

Reviewer

A component that can examine and optionally change the document. A common use is header analysis. Input reviewers execute immediately following the input parse of the message into XML; output reviewers execute immediately prior to document emitting.

Rules

Perform contents-based validation of input documents. Input rules also set and control the acknowledgement agent that reports the results of the validation for formats such as EDI.

Special Register

An element of context information that can be set and used during the lifecycle of a message. Special Registers can hold and set header information for emitters. Special Register values can be used in configuration expressions and process tests via the SREG() function.

Transform

A step in the transformer stage that alters the structure of the document being processed. Transforms can be prepared using the iWay Transformer tool.

Transformer

(1) A step in the process of a document that prepares it for the business process. (2) A tool provided by iWay to define and test message transformations. The transformer tool provides for both XML and non-XML transformations.

Truststore

A database of key material. Holds the public certificates of trusted partners for message exchange. Although it is possible to share a single file with the keystore, more formally a truststore and a keystore are separate entities.

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

TIBCO, the TIBCO logo, the TIBCO O logo, FOCUS, iWay, Omni-Gen, Omni-HealthData, and WebFOCUS are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the readme file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.