xMatters Integration Agent Guide

(version 5.1)

(x) matters

Current to:

xMatters version 5.1.0 patch 013
xMatters Integration Agent version 5.1.8.2

For more information about the latest changes, features, and fixes, and instructions on how to install xMatters patches, consult the release notes at the xMatters Community site: support.xmatters.com.

Thursday, January 25, 2018

Contacting xMatters, inc.:

You can visit the xMatters web site at: http://www.xmatters.com

Corporate Headquarters

12647 Alcosta Blvd, Suite 425
San Ramon, CA 94583
Telephone: 925.226.0300
Facsimile: 925-226-0310

Client Assistance:

Online: http://support.xmatters.com
International: +1 925.226.0300 and press 2
US/CAN Toll Free: +1 877.XMATTRS (962.8877)
EMEA: +44 (0) 20 3427 6333
Australia/APJ Support: +61-2-8038-5048 opt 2

Other Resources:

Join the xMatters Community: http://support.xmatters.com

Table of Contents

Chapter 1: Introduction

Welcome to the xMatters Integration Agent

The xMatters Integration Agent provides a method for interacting with xMatters components. Through its support for industry standard web services, the Integration Agent provides the following benefits:

- Standardized communication protocol
- Simpler implementation and reduced training requirements via SOAP/XML
- Streamlined deployment using standard corporate tools for load balancing and resiliency

Flexible scripting environment (JavaScript) that can be used to perform any number of activities (command line execution, JDBC connections, file read/write, etc.). This makes the Integration Agent a flexible processing engine supplying information/services for the requesting systems.

Seamless integration with management systems, including the ability to initiate xMatters events and respond to requests from xMatters Action Script

Note: *For a real-world example of an Integration Agent deployment, review the data flow diagram and associated text in the next section.*

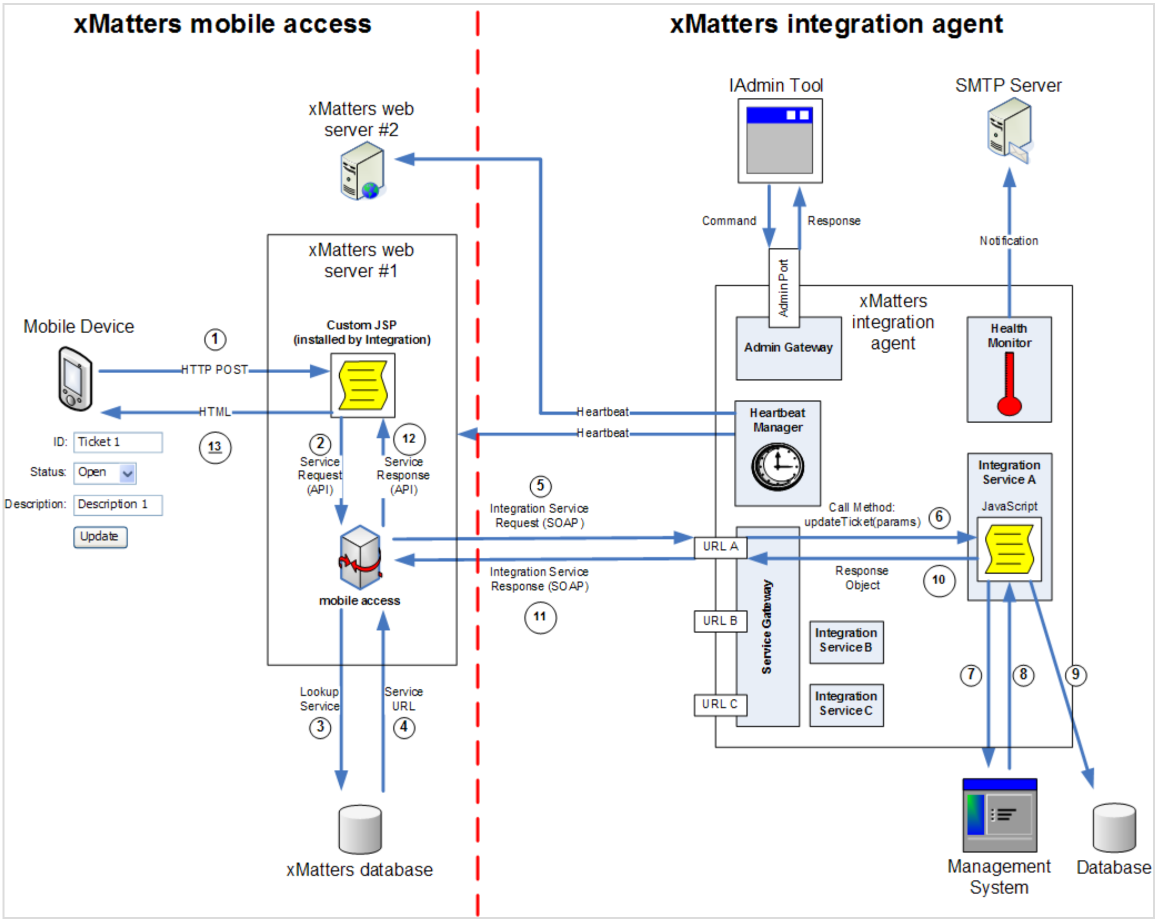
Architectural Overview

The xMatters Integration Agent facilitates bi-directional communication (both in terms of data flow and initiation), between xMatters and one or more management systems. This functionality can be divided into two sets of core features:

- An interface for web-based clients, such as the xMatters mobile access, to submit requests to a management system.
- An interface for management systems to submit requests to xMatters.

Data Flow 1: Communication Initiated by Web-Based Client

The following diagram shows the end-to-end data flow activity of the Integration Agent when communication is initiated by a web-based client (in this case, the xMatters mobile access):



xMatters Integration Agent Data Flow Diagram I

The numbers below correspond to those in the data flow diagram:

1. Via a mobile device (such as a BlackBerry), a user accesses the xMatters mobile access sample integration and is presented with a set of available Integration Services requests, including one named List Tickets. After selecting List Tickets, the user can click a particular ticket and modify its information. The user commits the modification by clicking **Update**, which submits a request structured as an HTTP POST to an integration-specific JSP hosted by the mobile access component.
2. The JSP transforms the posted request into a Java call to the mobile access component client. The call specifies the Integration Service to access (`default,sample`), the name of the function being performed by the Integration Service (*`updateTicket`*), and the parameters to the function (`description=Description 1, status=Open, id=Ticket 1`).
3. The client queries the database to determine which Integration Agent is providing `sample` (note that several Integration Agents may provide the same service).
4. The database responds with URLs for all Integration Agents providing `sample`.
5. The mobile access component client selects one of the URLs and then submits an Integration Service Request SOAP message that includes the name of the Integration Service function (*`updateTicket`*) and its parameters.
6. Each Integration Service listens for requests to its dedicated URL. The Integration Service Request is forwarded to `sample`'s implementation, which calls the JavaScript method named *`updateTicket`*.
7. *`updateTicket`* contacts the management system to update the ticket.

Note: *The actions of the* `updateTicket` *method here are only for illustration purposes. The sample implementation uses a simple XML file to simulate a management system database.*

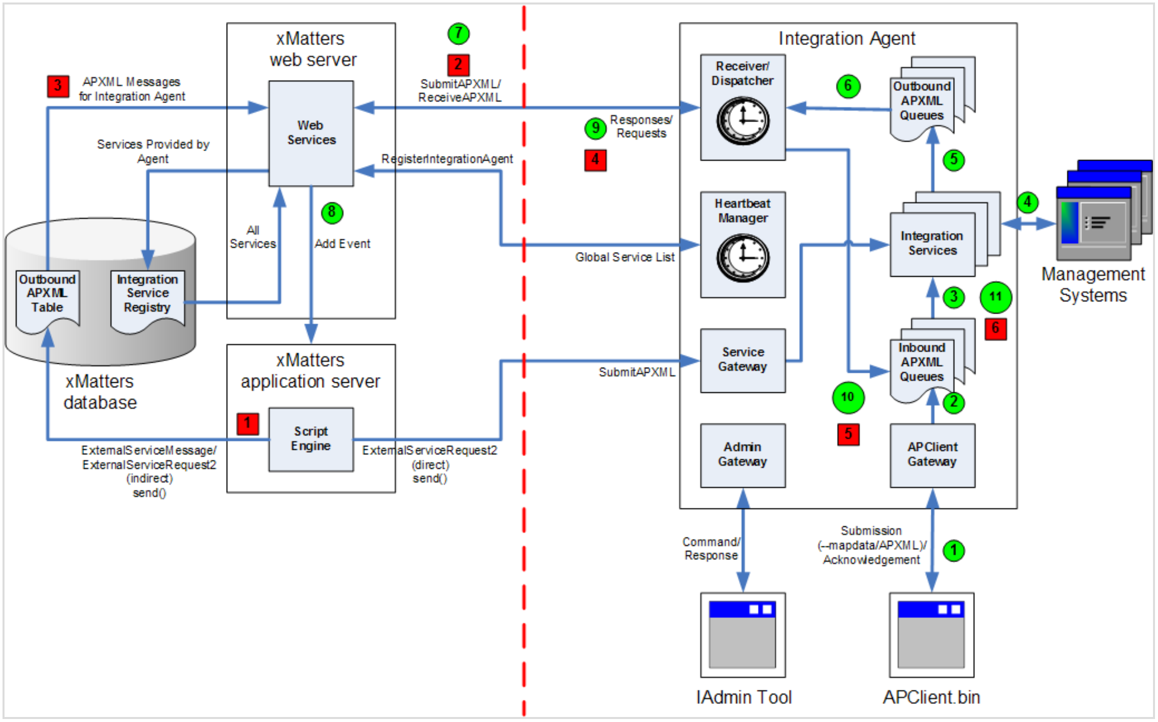
The implementation of an Integration Service is highly flexible; for example, management systems can be contacted, custom Java classes can be used, file systems can be accessed, and web services can be called.

8. The management system returns a success or failure response (in the management system's native format).
9. `updateTicket` writes an audit entry to the company's database.
10. `updateTicket` transforms the management system's response into a serializable Java object, and returns this as the result of the Integration Service Request.
11. The Integration Agent serializes the `updateTicket` response and returns it to the mobile access component client as part of the SOAP response body.
12. The client deserializes the SOAP response body into the associated Java object and returns this to the JSP.
13. The JSP uses the response object to determine whether the operation succeeded. If it has, the JSP generates an HTML page that relists the current tickets, formatted for the mobile device.

Note: *For information about the Health Monitor, see "Health Monitor" on page 48; for the IAdmin tool, see "Runtime Administration" on page 40.*

Data Flow 2: Communication Initiated by Management System

The following diagram shows the end-to-end data flow activity when communication is initiated by a management system:



xMatters Integration Agent Data Flow Diagram II

The numbers below correspond to the circled numbers in the data flow diagram, and explain the data flow for management system-initiated request to the Integration Agent:

1. An event (e.g., ticket creation) occurs within the management system and initiates some custom code (e.g., a script, a macro, or a Java program). The custom code uses the `APClient.bin` program to submit a `--mapdata` request to the Integration Agent. The
   `--mapdata` request specifies the Integration Service that is being targeted as well as additional data tokens describing the event. `APClient.bin` contacts the Integration Agent via an HTTP POST to the APClient Gateway component.
2. The APClient Gateway authenticates the request (i.e., checks access password and IP address ACL) and then applies the targeted Integration Service's data map to the `--mapdata` tokens to create an APXML message that will be submitted to xMatters. The APXML message is then added to the targeted Integration Service's inbound queue and a response is returned to `APClient.bin`. The response can be processed by the management system. All queues are persisted to disk so that APXML messages are never lost even if the Integration Agent server is rebooted.
3. The targeted Integration Service processes messages from its inbound queue in first in, first out (FIFO) order, as well as by priority. Each Integration Service is configurable with respect to the resources that it allocates for processing the various priorities of inbound messages. Once the submitted APXML message is selected for processing, it is passed to the targeted Integration Service's Input Action Scripting (IAS) framework.
4. For IAS, the submitted APXML message is passed to the `apia_input` method of the targeted Integration Service's JavaScript implementation. The implementor of the Integration Service is free to decide which actions should be performed in response to a submitted APXML message. Since the `apia_input` method is JavaScript, it can use a variety of methods (e.g., HTTP, SOAP, or custom API libraries), to interact with other management systems (e.g., to add additional information to the submitted APXML message or annotate the management system's ticket with an updated status).
5. Typically, at the end of IAS, the submitted (and perhaps now enriched) APXML message is forwarded to xMatters to initiate an event in response to the management system's event. Each Integration Service has a persisted outbound queue that stores APXML messages that are to be delivered to xMatters.
6. Periodically, the Receiver/Dispatcher component polls each Integration Service's outbound queue for messages. The outbound messages are processed with respect to their priorities and in such a way as to maintain fairness (i.e., to prevent a high throughput Integration Service from monopolizing bandwidth usage and starving low throughput Integration Services).
7. The selected outbound messages are sent as a batch to an xMatters web server via the `SubmitAPXML` web service call. The web server to which the messages are submitted is based on the result of the last heartbeat sent by the Heartbeat Manager.
8. The xMatters web server processes each submitted message in FIFO order and generates the necessary calls to satisfy the request specified by the message (e.g., add event). If an error occurs (e.g., a submitted message specifies an unknown command), the web server stops further message processing and returns an ERROR response to the Receiver/Dispatcher. The messages that were processed before the error remain unchanged (i.e., they do not need to be resubmitted).
9. The response to each `SubmitAPXML` web service call is a list of response APXML messages that correspond to the submitted APXML messages. These response messages are either OK or ERROR messages. OK messages indicate that the corresponding submitted APXML message was accepted by xMatters. ERROR messages indicate that the corresponding submitted APXML message was rejected by xMatters; the Receiver/Dispatcher will attempt to resubmit the problematic message three times in case the error is transient. If a submitted APXML message is rejected three times, it is logged and then removed from the system so that subsequent message processing can occur.
10. Both OK and ERROR response messages are added to the corresponding Integration Service's inbound queue where they will be processed in a similar fashion to the originally submitted APXML messages.
11. When an OK or ERROR response message is selected for processing, it is passed to the targeted Integration Service's Response Action Scripting (RAS) framework. The RAS framework is implemented by the `apia_ response` JavaScript method, which is analogous to the `apia_input` method used by IAS. RAS can be used to update the status of the management system's ticket or send additional APXML messages to xMatters.

As shown in this example, an Integration Service's IAS framework is used to process APXML messages that originate at a management system and are submitted via `APClient.bin`, and its RAS framework is used to process the corresponding APXML response messages that originate at xMatters and are submitted via the `SubmitAPXML` web service call's response.

xMatters can also submit APXML messages to an Integration Service independently of the responses that it normally returns via the `SubmitAPXML` response. In general, an Integration Service's RAS framework is used to process any APXML messages that originate at xMatters. The numbers below correspond to the numbers within squares in the data flow diagram, and explain the data flow for xMatters-initiated request to the Integration Agent:

1. An Action Script running on an xMatters Application Service makes an `ExternalServiceRequest2 call`, which is a request to an Integration Service to perform some action (e.g., query a management system). The `ExternalServiceRequest2` script object is transformed into an APXML message, which will be processed by the targeted Integration Service's RAS framework. There are two mechanisms by which the APXML message is submitted to the targeted Integration Service:
   - If the targeted Integration Service is configured to allow direct `ExternalServiceRequest2` calls, then the xMatters Application Server calls the Integration Service's `SubmitAPXML` web service method (note that this method is unrelated to xMatters web server's `SubmitAPXML` web service method). The `SubmitAPXML` method bypasses the Integration Service's inbound queues and allows the submitted APXML message to be immediately processed by the Integration Service's RAS framework. Any APXML message that is returned by the Integration Service's `apia_response` JavaScript method is immediately sent to the Application Server as the response to the `SubmitAPXML` call.
   - If the targeted Integration Service is configured to allow indirect `ExternalServiceRequest2` calls, then the APXML message is stored in an outbound message table in the database.

Note: *This example assumes that the targeted Integration Service is configured to allow indirect* `ExternalServiceRequest2` *calls.*

2. Periodically, the Integration Agent's Receiver/Dispatcher component calls the xMatters web server's `ReceiveAPXML` web service method.
3. In response to the `ReceiveAPXML` call, the web server queries the database for any APXML messages that target Integration Services that are being actively provided by the requesting Integration Agent.
4. The selected APXML messages are returned to the Receiver/Dispatcher component and removed from the database.
5. The Receiver/Dispatcher component processes each APXML message by adding it to the targeted Integration Service's inbound queue, where it will be processed in a similar fashion to APXML messages submitted via `APClient.bin`.
6. When an APXML message that originates at xMatters is selected for processing, it is passed to the targeted Integration Service's Response Action Scripting (RAS) framework. Typically, the Integration Service responds to a request from xMatters by contacting a management system and then sending an APXML message to xMatters that contains the response; this response is added to the Integration Service's outbound queue and processed in the same fashion as previously described.

Version compatibility

This version of the xMatters Integration Agent (5.1.8.2) is compatible with xMatters Premises deployments and xMatters On-Demand.

Previous versions

While the Integration Agent is backwards-compatible (i.e., the most recent release works with all patch levels of xMatters 5.x), some older versions of the Integration Agent may not work with more recent versions of xMatters. The following table identifies the available patch levels of xMatters, and indicates the minimum compatible version of the Integration Agent.

xMatters versions	Minimum xMatters Integration Agent version
5.0 initial release	5.0 (initial release)
5.0 patch 001	5.0 patch 001
5.0 patch 002	5.0 patch 002
5.0 patch 003	5.0 patch 003
5.0 patch 004	5.0 patch 004
5.0 patch 005	5.0 patch 005
5.0 patch 006, 007, and 008.	5.0 patch 006
5.0 patch 009 and 010	5.0 patch 007
5.1	5.1.0
5.1	5.1.1
5.1 patch 001	5.1.2
5.1 patch 002	5.1.3
5.1 patch 003, 004	5.1.4
5.1 patch 005, 006, 007	5.1.5
5.1 patch 008	5.1.6

Note: *If you are using the Integration Agent with xMatters On-Demand, update to the latest Integration Agent release.*

Chapter 2: Installation

About Installing

Before installing the xMatters Integration Agent, review this chapter to familiarize yourself with the installation process. You may also want to review the Configuration chapter to learn more about configurable items.

Supported Operating Systems

The xMatters Integration Agent supports the following operating systems:

Platform	Manufacturer	OS	Versions Used For Testing
64-bit	AMD, Intel	Windows	Server 2003, 2008 R2, 2012 R2
		LINUX	Various, including Red Hat / CentOS 6 and 7
Sparc	Sun	Solaris	10
Itanium	HP	HPUX	11.31 On HP-UX Itanium deployments, the default wrapper.conf file must be modified. For more information, see "Post-installation tasks and troubleshooting" on page 15.
POWER Processor	IBM	AIX	7.1

Pre-Installation Requirements Checklist

You can prepare for installation by reviewing the following requirements checklist:

Pre-Installation Requirements Checklist

Item	Requirement
User account	User account on the Integration Agent server (on Windows, this account must have permissions to create Services).
Hard disk space	500 MB of free hard disk space.
RAM	Minimum 640 MB of available memory.
Server and supported operating system	See above.

Item	Requirement
Web Service login	The Integration Agent requires an authorized xMatters Web Services User to register with the web servers. This includes the user name, password, and company name. When you add a web services license to a new or existing installation, a default Web Service User account is created named IA_User. **NOTE:** You must reset the password of the IA_User account through the xMatters web user interface (**Users > Find Web Service Users**) before the account can be used (otherwise, the Integration Agent heartbeat fails with an AUTHENTICATION_ERROR). **Allowed Web Services**: Submit APXML, Receive APXML, Register Integration Agent
Proxy server details	You can configure the Integration Agent to communicate with the xMatters server via a proxy server. Make a note of your proxy server's IP address or hostname, port, and authentication details. If your proxy server uses NTLM v1 authentication, note the domain name to use for the proxy server. (NTLM v2 is not currently supported.)
Server Hostname/IP	Hostname or IP address of the Integration Agent server. If the Integration Agent is configured for 'direct' external-service-request mode, then the Integration Agent must be accessible from the xMatters hosted instance.
TCP Ports	A free TCP port on the Integration Agent server for each of the following components: ◦ Integration Services (the default TCP port setting is 8081) ◦ IAdmin (default is 8082) ◦ APClient (default is 2010) ◦ Internal message broker (default is 61618)
SMTP server details (if Health Monitor enabled)	If the Health Monitor will be enabled, the address of an SMTP server (hostname/IP address and port (default is 25)) reachable without user/password authentication from the Integration Agent server.
Email address for Health Monitor messages (if enabled)	If the Health Monitor will be enabled, the email address of the person to which Integration Agent Health Monitor messages will be sent.
X.509 certificate (if SSL enabled)	If SSL will be enabled for Integration Services and the installed default self-signed certificate is not sufficient, an X.509 certificate in a keystore file is required.

Installing on Windows

This section guides you through a Window-based installation of the Integration Agent.

To install on Windows operating systems:

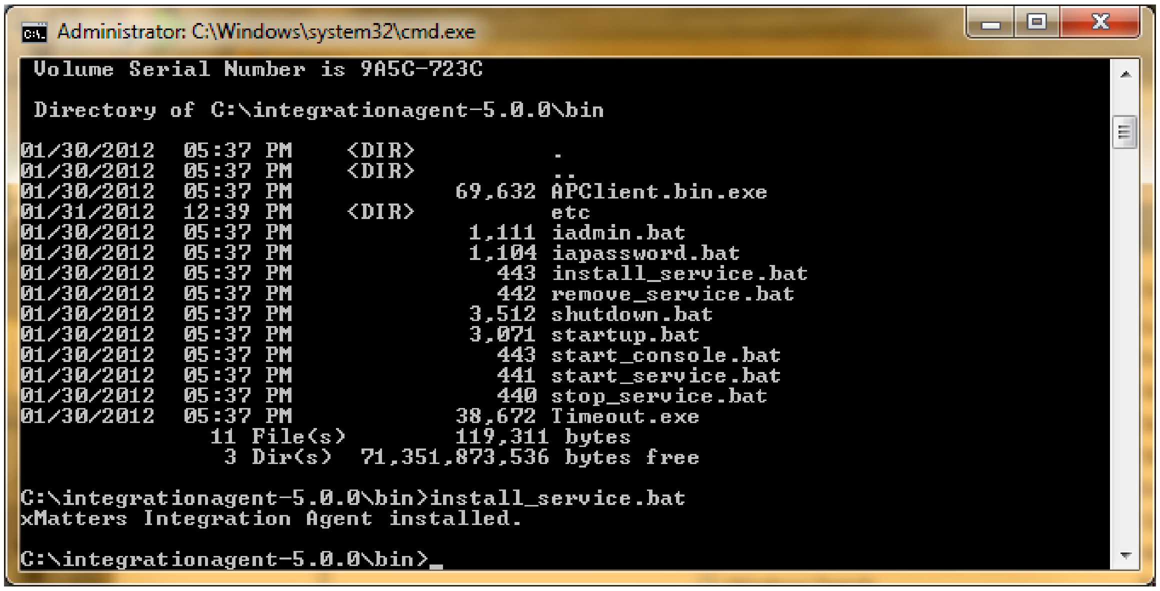
1. Download the Integration Agent archive file and the latest Integration Agent Utilities package from the xMatters web site.

- Note that the archive file is different depending on the platform to which you are installing; ensure that you download the correct Integration Agent for your operating system.

2. Extract the Integration Agent archive to a server on your LAN. Unless otherwise indicated in the documentation for your specific integration, it is recommended that you install the Integration Agent on the same server as your management system.
    - The folder to which you install the Integration Agent is referred to throughout this document as `<IAHOME>`.
3. Navigate to the `<IAHOME>\conf` folder in the extracted archive, and delete the existing `wspasswd` file.
4. Navigate to `<IAHOME>\bin`, and run the following command, replacing <password> with the Web Service Login password referred to in the "Pre-Installation Requirements Checklist" on page 10. If the password is incorrect, the Integration Agent will be unable to register with xMatters.

```
iapassword.bat --new <password> --file conf/wspasswd
```

5. Open the `<IAHOME>\conf\IAConfig.xml` file in a text editor, and specify the settings to configure the Integration Agent.
    - For a description of the settings and their recommended values, see "Integration Agent Configuration File" on page 18; the XML file also contains extensive comments to assist you with the requirements for each setting.
6. Save and close the `IAConfig.xml` file.
7. Install the Integration Agent Utilities as described in "Integration Agent utilities" on page 14.
    - The Integration Agent may fail to start if the utilities are not installed.
8. To install the Integration Agent as a Windows Service, navigate to the `<IAHOME>\bin` directory again and run `install_service.bat`.

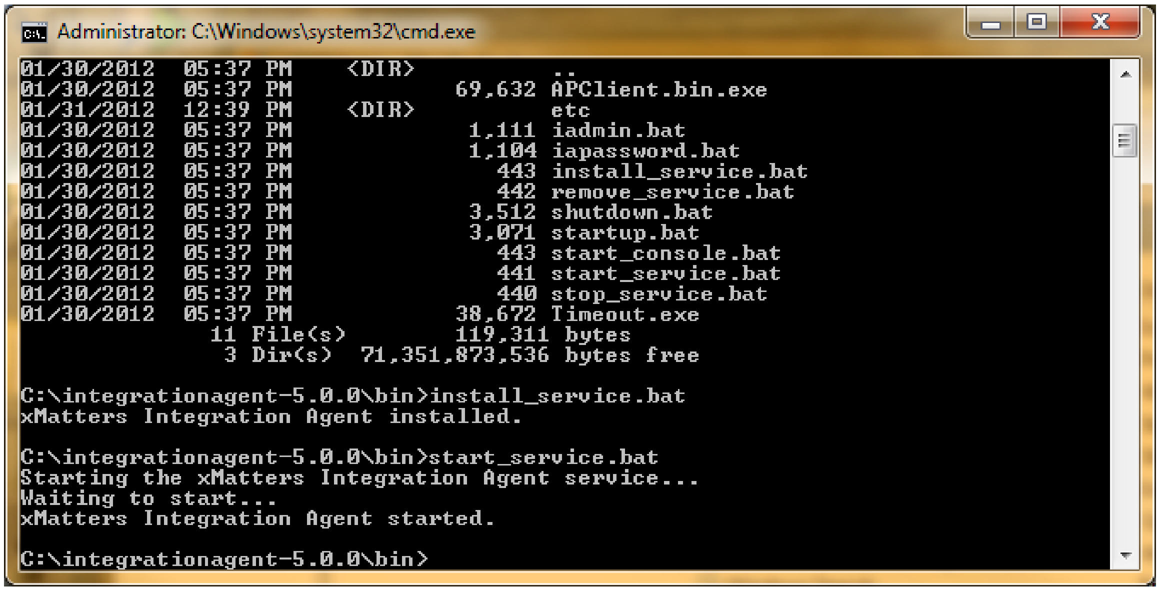


Administrator: C:\Windows\system32\cmd.exe

```
 Volume Serial Number is 9A5C-723C

 Directory of C:\integrationagent-5.0.0\bin

01/30/2012  05:37 PM    <DIR>          .
01/30/2012  05:37 PM    <DIR>          ..
01/30/2012  05:37 PM            69,632 APClient.bin.exe
01/31/2012  12:39 PM    <DIR>          etc
01/30/2012  05:37 PM             1,111 iadmin.bat
01/30/2012  05:37 PM             1,104 iapassword.bat
01/30/2012  05:37 PM               443 install_service.bat
01/30/2012  05:37 PM               442 remove_service.bat
01/30/2012  05:37 PM             3,512 shutdown.bat
01/30/2012  05:37 PM             3,071 startup.bat
01/30/2012  05:37 PM               443 start_console.bat
01/30/2012  05:37 PM               441 start_service.bat
01/30/2012  05:37 PM               440 stop_service.bat
01/30/2012  05:37 PM            38,672 Timeout.exe
              11 File(s)        119,311 bytes
               3 Dir(s)  71,351,873,536 bytes free

C:\integrationagent-5.0.0\bin>install_service.bat
xMatters Integration Agent installed.

C:\integrationagent-5.0.0\bin>_
```

Command to Install Integration Agent as a Windows Service

9. To start the Windows Service, run `start_service.bat` (alternatively, you can start the xMatters Integration Agent from the Windows Services panel).



```
01/30/2012  05:37 PM    <DIR>          ..
01/30/2012  05:37 PM            69,632 APClient.bin.exe
01/31/2012  12:39 PM    <DIR>          etc
01/30/2012  05:37 PM             1,111 iadmin.bat
01/30/2012  05:37 PM             1,104 iapassword.bat
01/30/2012  05:37 PM               443 install_service.bat
01/30/2012  05:37 PM               442 remove_service.bat
01/30/2012  05:37 PM             3,512 shutdown.bat
01/30/2012  05:37 PM             3,071 startup.bat
01/30/2012  05:37 PM               443 start_console.bat
01/30/2012  05:37 PM               441 start_service.bat
01/30/2012  05:37 PM               440 stop_service.bat
01/30/2012  05:37 PM            38,672 Timeout.exe
              11 File(s)        119,311 bytes
               3 Dir(s)  71,351,873,536 bytes free

C:\integrationagent-5.0.0\bin>install_service.bat
xMatters Integration Agent installed.

C:\integrationagent-5.0.0\bin>start_service.bat
Starting the xMatters Integration Agent service...
Waiting to start...
xMatters Integration Agent started.

C:\integrationagent-5.0.0\bin>
```

Command to Start Integration Agent as a Windows Service

Installing on Linux, Solaris, and AIX

This section guides you through a Unix-based installation of the Integration Agent.

Note: *The archive files include path names longer than 100 characters. As a result, you must use a GNU-compatible version of tar to extract the tar archives.*

To install on Unix-based operating systems:

1. Download the Integration Agent archive file from the xMatters web site.
   - Note that the archive file is different depending on the platform to which you are installing; ensure that you download the correct Integration Agent for your operating system.
2. Extract the Integration Agent archive to a server on your LAN. Unless otherwise indicated in the documentation for your specific integration, it is recommended that you install the Integration Agent on the same server as your management system.
   - The folder to which you install the Integration Agent is referred to throughout this document as `<IAHOME>`.
3. Navigate to the `<IAHOME>/conf` folder in the extracted archive, and delete the existing `wspasswd` file.
4. Navigate to `<IAHOME>/bin`, and run the following command, replacing <password> with the Web Service Login password referred to in the "Pre-Installation Requirements Checklist" on page 10. If the password is incorrect, the Integration Agent will be unable to register with xMatters.

```
./iapassword.sh --new <password> --file conf/.wspasswd
```

5. Open the `<IAHOME>\conf\IAConfig.xml` file in a text editor.
6. Specify the settings to configure the Integration Agent.
   - For a description of the settings and their recommended values, see "Integration Agent Configuration File" on page 18; the XML file also contains extensive comments to assist you with the requirements for each setting.
7. Save and close the `IAConfig.xml` file.
8. Install the Integration Agent Utilities as described in "Integration Agent utilities" on page 14.
   - The Integration Agent may fail to start if the utilities are not installed.

Note that you must start the Integration Agent daemon manually (for details, see "Starting the Integration Agent" on page 38).

Integration Agent utilities

The Integration Agent Utilities bundle (IAUtils) is a collection of scripts and binary code that provides required functionality for many integrations. The bundle is distributed separately from the main Integration Agent installer, but must be installed at the same time. If the IAUtils package is not present when the Integration Agent is started, the startup may fail.

Note: *Prior to the Integration Agent version 5.1.2 release, the installer included the IAUtils package. The utilities were removed from the installer because some versions of integrations required a specific version of IAUtils, and upgrading or reinstalling the Integration Agent would overwrite the required version and cause the integration to stop working.*

Functionality

The purpose of the utilities bundle is to add to the capabilities of the Integration Agent. For example, the Integration Agent has the ability to send incident messages to the xMatters server, and to receive user responses. This core functionality does not allow user data to be synchronized from a management system to xMatters, or for the Integration Agent to send queries to xMatters about existing events. These capabilities and more are provided by the code in the IAUtils JavaScript files.

The following is an overview of the functionality provided by the IAUtils scripts; for more detailed information, refer to the comments within the files:

`wsutil.js` exposes the following functions via the "WSUtil" class:

- `sendReceive()`: send a SOAP request; for example, to a management system's SOAP endpoint.
- `restSendReceive()`: send a REST request; for example, to a management system's REST endpoint.
- `base64Encode()`: exposes the functionality provided by `java.lang.String.encoder.encodeBase64()`.
- `formatStringForE4X()`: remove XML headers from a string (the JavaScript engine in Java 1.7 does not tolerate XML headers).

If the Integration Agent is configured to use a proxy server when sending messages to xMatters, `WSUtil.sendReceive()` and `WSUtil.restSendReceive()` will use the same proxy configuration.

`xmattersws.js` includes all of the functionality of the WSUtil class and also exposes the following functionality via the "xMattersWS" class:

- `syncSendReceive()`: send user, group, coverage, team, or device data to xMatters.
- `queryIncident()`: ask xMatters whether notifications are pending for a specified incident ID.
- `SendDelAPXML()`: tell xMatters to terminate pending notifications corresponding to a specified incident ID.
- `submitAPXML()`: send an incident to xMatters, subject to deduplication. Optionally, tell xMatters to first delete any pending notifications for the specified incident ID.
- `addEventTokensFromObject()`: add data from a JavaScript object to an APXML document.
- `initializeResponse()`: prepare a response to an ESR2 message from xMatters.
- `convertTokensToString()`: convert all tokens from an APXML document into a parenthesis-grouped set of name-value pairs.
- `getPassword()`: extract data from a file encrypted via the com.alarmpoint.integrationagent.security.EncryptionUtils package.

The above scripts expose the functionality provided by the `integrationagent-utils.jar` file. Consequently, integration service scripts can access the IAUtils functionality by loading the `wsutil.js` or the `xmattersws.js` scripts from the `webservices` and `xmatters` folders within `integrationservices\lib\javascript\`.

The `integrationservices\lib\javascript\xmatters` folder also contains several JavaScript files containing utility functions used by data synchronization integrations. See the comments within the files for descriptions of their implemented functions. For usage examples, see the data load integrations available from support.xmatters.com.

Downloading and installing

You can download the IAUtils package (`integrationagent-utils.jar`) from the same location as the Integration Agent (support.xmatters.com).

Note: *If you are using an integration that requires a specific version of the IAUtils, follow the instructions in the integration guide for that integration.*

To install the IAUtils, extract the `integrationagent-utils.zip` archive file to `<IAHOME>`, and allow the extracted `lib` and `integrationservices` folders to merge with the ones already present in `<IAHOME>`.

Post-installation tasks and troubleshooting

The following sections identify any post-installation tasks that must be completed prior to starting or configuring the Integration Agent.

Access control list (required for all On-Demand installations)

If you are deploying the Integration Agent for use with xMatters On-Demand, you must ask Client Assistance to add your Integration Agent to the access control list (ACL). This allows us to whitelist the IP address where your requests are coming from. For more information or to add your Integration Agent to the ACL, go to the xMatters Support site at support.xmatters.com and click **Submit Request**.

Updating wrapper.conf

Many parameters within wrapper.conf have names beginning with "wrapper.java.additional", followed by a number. For example,

```
wrapper.java.additional.6
```

These parameters must comply with the following rules, or they will be ignored:

- The numbers must be unique; for example, there cannot be two instances of wrapper.java.additional.6
- The numbers must be consecutive; for example, if there is a wrapper.java.additional.6 and a wrapper.java.additional.8, but no wrapper.java.additional.7, then all parameters after wrapper.java.additional.6 will be ignored.)
- If a parameter begins with "#", it is treated as a comment and the above rules do not apply to it.

HP-UX deployments

On HP-UX Itanium, the number 6 is reserved, and the line must be modified to:

```
wrapper.java.additional.7=-Xbootclasspath/a:%INSTALL_DIR%/jre/lib/tools.jar
```

This applies to all users of HP-UX, not only those upgrading an existing installation. The included `wrapper.conf` file uses "6" for the sequence number and this must be modified on ALL HP-UX Itanium deployments. The parameters rules detailed above still apply.

AIX deployments

As of the Integration Agent 5.1.8.2 release, the numbers 7 and 8 are reserved for the following properties:

```
wrapper.java.additional.7=-Dcom.ibm.jsse2.convertSSLv3=true
wrapper.java.additional.8=-Dcom.ibm.jsse2.overrideDefaultTLS=true
```

If you have previously enabled additional properties beyond the six default parameters, you will need to modify them accordingly. The parameter rules detailed above still apply.

SSL certificates

When starting up the Integration Agent, you may encounter an error message similar to the following:

```
[Heartbeat-1] ERROR - The xMatters Web Server
https://<HOST>.xmatters.com/api/services/AlarmPointWebService is unavailable or completely
rejected the heartbeat.
org.apache.axis2.AxisFault: sun.security.validator.ValidatorException:
PKIX path building failed: sun.security.provider.certpath.SunCertPathBuilderException: unable
to find valid certification path to requested target
```

This occurs when the SSL certificates were not imported properly.

To resolve this error:

1. Stop the Integration Agent.
2. Ensure that the required SSL certificates have been imported into `<IAHOME>\jre\lib\security\cacerts`.
    - For additional instructions or troubleshooting tips, contact xMatters Support.
3. Restart the Integration Agent.

Chapter 3: Configuration

This chapter describes how to define configuration settings for the xMatters Integration Agent and Integration Services using their XML configuration files.

Configuration Files

The Integration Agent uses two types of XML configuration files:

- Integration Agent configuration file
- Integration Service configuration files

The following sections provide details about these configuration files.

Note: *The tables in the following sections use "@" to denote the name of an attribute (i.e, to distinguish attributes from sub-elements, which are enclosed in angle brackets (<>). This is a notational convenience and does not make up part of the attribute name.*

Integration Agent Configuration File

The Integration Agent configuration file is named `IAConfig.xml` and is located at:

- **Windows:** `<IAHOME>\conf`
- **Unix:** `<IAHOME>/conf`

The configuration file is annotated to assist you in defining important settings.

Note: *Before using the Integration Agent, you must run the IAPassword utility twice; once to create a .wspasswd file for the Web Services User, and once to create a .passwd file (or any named file that matches the configuration in IAConfig.xml) for the Integration Agent. For details, see"The IAPassword Utility" on page 44 .*

The following table describes elements and attributes in the configuration file (`IAConfig.xml`):

Element	Description & Attributes
id	Each Integration Agent has a unique ID (a string with leading and trailing whitespace removed); if this parameter is not specified, a default agent ID is auto-generated in the format `<computer-name>/<ip>:<service-gateway port>`. For example: `<id>workstation-198-bsmith/10.2.0.121:8081</id>` You can assign a custom agent ID, but ensure that any < and `&` characters in the name are replaced with their XML entity names; i.e., `"<"` and `"&"` respectively. Each agent ID must be unique across all collaborating Integration Agents.

Element	Description & Attributes
proxy-config	You can configure the Integration Agent to communicate with the xMatters server via a proxy server using the settings in this section. If this section is omitted (i.e., commented out, as it is by default), the Integration Agent will not use a proxy server. **Sub-elements:** **<proxy-enabled>**: Specifies whether the proxy configuration settings are enabled; this must be set to *true* for the Integration Agent to use the proxy settings. **<proxy-host>**: The IP address or hostname of the proxy server you want the Integration Agent to use. **<proxy-port>**: The port to use on the proxy server. **<proxy-auth-username>**: If required, the username used to authenticate with the proxy server. If the proxy server does not require authentication, this field must be commented out. **<proxy-auth-password>**: If required, the password used to authenticate with the proxy server. If the proxy server does not require authentication, this field must be commented out. **NOTE**: If the proxy-auth-username and proxy-auth-password tags are not commented out, the Integration Agent will send an authorization header to the proxy server, even if the tags are empty. In this case, the header will contain only a ":" character, which may cause the proxy connection to be refused. **<proxy-auth-ntlm-domain>**: If the proxy server uses NTLM v1 authentication, the domain name to use for the proxy server. If the proxy server does not use NTLM v1 authentication, this setting should be omitted or left blank. **NOTE**: NTLM v2 is not currently supported. If you are using a `wrapper.conf` file which specifies proxy settings, be aware that they will override the proxy settings in `IAConfig.xml`. Use the `wrapper.conf` file that was shipped with the Integration Agent, as older wrapper files may not be forward-compatible.
web-services-auth	The Integration Agent uses this account to register itself with the xMatters server, and to send and receive incidents and responses. The specified account must exist as a Web Services User in xMatters, and must have the "Register Integration Agent", "Receive APXML", and "Send APXML" privileges. **Sub-element:** **<user>**: Login name (user name) for the xMatters Web Service account. **<password>**: Path (absolute or relative) to a file containing the login password for the account. The file is encrypted using the iapassword application, and must be created for the Web Service User (the default is `/conf./.wspasswd`). **<company>**: Company that includes the xMatters Web Service User account. This value can be left blank if the deployment does not have more than one company. The Integration Services that are provided by this Integration Agent belong to the specified Company.

Element	Description & Attributes
heartbeat	The Integration Agent can periodically send a heartbeat signal to your xMatters web server to identify the Integration Services it provides. The configuration of the `heartbeat` element identifies the web servers the Integration Agent should target with these registration attempts. During a heartbeat, if the Integration Agent successfully registers with one of the primary servers, the heartbeat is considered successful. If none of the primary servers accepts a registration attempt, the Integration Agent attempts to use the secondary servers list for a specified duration, after which the primary servers will be retried. **Note**: The URLs specified for each web server must point to the location of the AlarmPointWebService, and begin with either http:// or https://. These URLs cannot have a query or fragment component (the URLs must be resolvable from the Integration Agent). For example: `http://localhost:8888/api/services/AlarmPointWebService` **Sub-element:** **<interval>**: Duration (in seconds) between successive registration/heartbeats (value: positive integer). If you are using the Integration Agent with xMatters On-Demand, this value is ignored. **<primary-servers>**: The URL for each primary xMatters web server; all primary servers are assumed to be connected to the same database. The Integration Agent will only send a heartbeat signal to the next URL if it fails to register with the previous one listed. **<secondary-servers>**: The URL for each secondary xMatters web server; all secondary servers are assumed to be connected to the same database. If you are using the Integration Agent with xMatters On-Demand, do not specify any secondary servers. **<primary-retry-attempts>**: Number of times the Integration Agent will attempt to register with the primary servers before failing over to the list of secondary servers. **<recovery-interval>**: Duration (in seconds) that the Integration Agent will attempt to register with the secondary servers before attempting to register with the primary servers again. A value of 0 indicates that the primary servers will never be retried. Regardless of this value, if the Integration Agents fails to register with any of the secondary servers, it will immediately retry the primary servers. **Note**: For more information about the messages associated with these settings, see "Health Monitor" on page 48.

Element	Description & Attributes
apxml-exchange	The Integration Agent periodically submits and receives APXML messages to and from the xMatters server to which the last successful heartbeat was sent based on the configuration of the `apxml-exchange` element. If the last heartbeat was unsuccessful, no APXML messages are exchanged. **Note**: If you are using the Integration Agent with xMatters On-Demand, this element is ignored. The polling frequency in the On-Demand service is controlled by xMatters. **Sub-element:** **<interval>**: Duration (in seconds) between successive APXML exchanges (value: positive integer). **<size>**: The maximum number of APXML messages that are sent and received during each exchange. This count applies seperately to the submitted and received messages (i.e., the total number of exchanged messages can be twice this limit).
ip-authentication	If enabled, only clients with IP addresses that match a listed address are allowed to submit requests to the Integration Agent. This includes the IP addresses of xMatters mobile access-enabled web servers, xMatters Application Server Nodes, and users of `APClient.bin`. **Attribute and sub-element:** **@enable**: Controls whether requests are authenticated by IP address (values: true, false). **<ip>**: 0 or more IP addresses, each identifying an authorized client. Addresses can include wildcards (e.g., `192.168.168.*` would authorize all IP addresses beginning with `192.168.168`).
password-authentication	If enabled, only clients that provide the correct password are allowed to submit requests to the Integration Agent. This includes xMatters mobile access-enabled web servers, xMatters Application Server Nodes, and users of `APClient.bin`. **Attribute and sub-element:** **@enable**: Controls whether requests are authenticated by password (values: true, false). **<password>**: Path (absolute or relative) to a file containing the access password. The file is encrypted using the iapassword application and must be created before you can use the Integration Agent (the default is `/conf./.passwd`).

Element	Description & Attributes
external-service-request	This determines the behavior of xMatters when `ExternalServiceRequest2`'s `send()` method is called and this Integration Agent is the target. In direct mode, xMatters makes an immediate web service call to this Integration Agent (this requires network privileges for node-initiated communication). In indirect mode, xMatters queues an APXML message for processing by this Integration Agent via the `apxml-exchange` mechanism (this does not require network privileges for node-initiated communication). Direct mode is the preferred method because it reduces the latency of `ExternalServiceRequest2` processing. However, when this Integration Agent is behind a firewall that prevents externally-initiated communication, the indirect method is required. **Attributes:** **@mode** – Controls whether the node processes `ExternalServiceRequest2` messages via web services or the apxml-exchange mechanism (values: direct, indirect).
request-timeout	Whenever a method in the Integration Service's JavaScript implementation is called, it must be processed within this timeout (in seconds); otherwise, it is interrupted and a timeout exception is returned to the client (value: positive integer). This applies to Integration Service Requests, `ExternalServiceRequest2` requests, and Input and Response Action Scripting.
admin-gateway	Web Services gateway exposed to the IAdmin utility. **Attributes:** **@ssl** – whether administration requests should be encrypted (values: true, false). **@host** – value is always "localhost" (do not change). **@port** – unused TCP port; must be different from the ports of the other `gateway` elements (value: integer from 0 to 65535).
service-gateway	Web services gateway exposed to the xMatters web servers (Integration Service Requests are submitted to child paths of this URL). **Attributes:** **@ssl** – whether Integration Service Requests should be encrypted (values: true, false). **@host** – resolvable from the xMatters web servers (value: hostname (e.g., www.company.com) or IPv4 address (e.g., 192.168.54.32)). **@port** – unused TCP port; must be different from the ports of the other `gateway` elements (value: integer from 0 to 65535).

Element	Description & Attributes
apclient-gateway	HTTP gateway exposed to Management Systems (either directly or via `APClient.bin`). **Attributes:** **@ssl** – whether Management System submissions should be encrypted (values: true, false). **Note:** *APClient.bin supports only HTTP communication with the Integration Agent, even if the value of the --http-post parameter is an https:// URL (however, the Integration Agent can communicate back to the Management System using any of the secure protocols supported by the Management System's API.).* *If the Integration Agent is configured to use HTTPS for Management System communication (see the apclient-gateway entry in "Integration Agent Configuration File" on page 18), all APClient.bin submissions will be rejected. To communicate with the Integration Agent, the Management System must form its own HTTPS requests as described in "Input APXML").* **@host** – hostname (e.g., localhost) or IPv4 address (e.g., 127.0.0.1) of the Integration Agent. **@port** – unused TCP port; must be different from the ports of the other `gateway` elements (value: integer from 0 to 65535).
emergency-contact	Controls where the Integration Agent sends emergency notification emails when a serious condition occurs (e.g., heartbeat or Integration Service failure; for details, see "Health Monitor Events" on page 49). **Attributes and sub-elements:** **@enabled** – whether the Health Monitor is active (values: true, false). When set to true, the Health Monitor will send an email message to the specified email address every time an issue is detected or resolved. **<contact>/<to>** – email address to which Health Monitor messages are sent. **<contact>/<from>** – email address from which to send Health Monitor messages. **<smtp-relay>/@host** – hostname for the SMTP server through which the Integration Agent sends Health Monitor messages; must be resolvable from the Integration Agent server (value: hostname (e.g., localhost) or IPv4 address (e.g., 127.0.0.1)). **<smtp-relay>/@port** – port (typically 25) for the SMTP server through which the Integration Agent sends Health Monitor messages (value: integer from 0 to 65535).
service-configs	Integration Service configuration files are organized within a file structure rooted at `<IAHOME>/integrationservices`. **Attributes and sub-elements:** **@dir** – value is always "../integrationservices" (do not change). **<path>** – 0 or more path elements that specify the Integration Service configuration files that the Integration Agent loads (paths must be relative to `<IAHOME>/integrationservices`, and may be Unix- or Windows-formatted).

Note: *Even if you are running the Integration Agent on Windows, it is recommended that you use Unix-style file path formatting, which works on both platforms.*

Integration Service Configuration File

File names for Integration Service configuration files are user-defined in the format `<filename>.xml` (e.g., `sample.xml`).

The path to each Integration Service configuration file loaded by the Integration Agent is specified in the agent's configuration file (for details, see the `service-configs` element in the table titled "Configuration Settings" in "Integration Agent Configuration File" on page 18).

Note: *The Integration Agent installation includes an annotated sample Integration Service configuration file located at* `<IAHOME>/integrationservices/sample/sample.xml`. *If during installation you chose not to install the sample integration, the path to its configuration file is commented out in the* `IAConfig.xml` *file so that it is not loaded into the Integration Agent.*

The following table describes elements and attributes in the configuration file:

Integration Service Configuration Settings

Elements	Description & Attributes
domain	Event Domain to which the Integration Service belongs. Event Domains can be any combination of letters (a-z or A-Z), numbers (0-9), dashes ("-"), and underscores ("_").
name	Each service must have a name that is unique, regardless of case, within the Event Domain for the Integration Service. Names must begin with a letter (a-z or A-Z), followed by any combination of letters (a-z or A-Z), numbers (0-9), or dashes ("-").
clients **(optional element)**	Specifies the clients that are expected to make requests to the Integration Service. Only Integration Services that support the xMatters mobile access as a client are registered with xMatters. For Integration Services that are designed to support management systems only via APClient Requests, the exclusion of xMatters mobile access as a client eliminates the need for the additional configuration as described in the "Adding an Integration Service" section of the *xMatters mobile access guide*. **Note:** If this element is omitted, the Integration Service is assumed to support both xMatters mobile access and APClient requests. Additionally, the clients setting does not affect the actual endpoints (e.g., web services) that the Integration Service exposes. **Sub-elements**: • <client> – 0 or more elements that identify the clients of the Integration Service. Possible values: mg, apclient, where mg refers to the xMatters mobile access and apclient refers to management systems making APClient requests.
initial-state	Possible values: active, suspended. All services are loaded (i.e., parsed and configured), but only active services process requests by running input action, response action, or mobile access scripts. A suspended service queues APClient requests and rejects mobile access requests. Suspending a service initially is useful for debugging a configuration since parsing and validation are always performed, regardless of initial state.

Elements	Description & Attributes
concurrency **(optional element)**	For improved performance, services can concurrently process inbound APXML messages (from both APClient and xMatters). Messages are grouped in two stages: first, by priority; and, second, by apia_ process_group token. Messages with the same apia_process_group token are processed sequentially in first come, first served order, while messages with different apia_process_group tokens are processed concurrently. **Sub-elements**: • **<normal-priority-thread-count>** – maximum number of normal priority groups that can be processed concurrently. • **<high-priority-thread-count>** – maximum number of high priority groups that can be processed concurrently.
script	Integration Service Requests are implemented by corresponding methods in a JavaScript file; this element defines the location of the script and other properties. **Attribute and sub-element**: • **@lang**: value is always “js” (do not change). • **<file>**: relative path (resolved against the directory containing this file) of the script implementing the service (and may be Unix- or Windows-formatted). • **<optimization>**: values: integer ranging from -1 through 9 (default 9) OR the string value 'debug'; -1 means the JavaScript is always interpreted every time the script is run (this is the slowest performance); 0 means the JavaScript is pre-compiled but no optimizations are performed; 9 means the JavaScript is pre-compiled for maximum efficiency. Setting this to 'debug' sets the optimization value to -1, and displays the debugger whenever an event is injected.
javaclient **(optional element)**	An Integration Service can make use of the mapped-input, input-action and response-action data from a JavaClient integration by specifying the path to the AlarmPoint Java Client integration XML. This element is optional; however, if specified, the non-legacy mapped-input and constants elements in this configuration are ignored. **Sub-element**: • **<file>**: relative path (resolved against the directory containing this file) of the AlarmPoint Java Client integration configuration file (and may be Unix- or Windows-formatted).

Elements	Description & Attributes
classpath **(optional element)**	Integration Service scripts have access to all classes and JARs stored in `<IAHOME>/lib`. However, to prevent conflicts and enhance security, an Integration Service should load its own classes and resources from an unshared directory. The classpath element allows an Integration Service to specify multiple paths that will be added to the service's classpath during the processing of an Integration Service Request. Note that although this classpath augments the default classpath (which is available to all services), the augmented classpath is exclusive to this service. **Note:** JDBC drivers cannot be specified using this element; instead, they must be placed in `<IAHOME>/lib/integrationservices` where they will be automatically available without further configuration. **Sub-element**: • **<path>**: 0 or more relative paths (resolved against the directory containing this file) that specify the location of JAR files, class files, or other resources. Each path may include * and ? wildcards to refer to multiple files or directories. **Notes:** • Subdirectories are not recursively searched. • Trailing \ and / are ignored (e.g., "classes/test/" is the same as "classes/test").
mapped-input	Non-legacy Integration Services use the `mapped-input` element to define how an APClient map-data request is transformed into an APXML message. The first `map-data` token is always treated as an `agent_client_id` APXML token. Subsequent `map-data` tokens are transformed in order according to the following parameter sub-elements. If too few `map-data` tokens are supplied, the unused parameter sub-elements are ignored. Conversely, if too many `map-data` tokens are supplied, the unused tokens are ignored. **Attributes and sub-elements**: • **@method**: resulting APXML message's `method` element value. • **@subclass**: resulting APXML message's `subclass` element value; this attribute is optional and if not specified, the resulting APXML message will have no `subclass` element. • **<parameter>**: 0 or more parameters, each defining an APXML token key, and ordered the same as the `map-data` tokens (beginning with the second `map-data` token). The resulting APXML token's value derives from the corresponding `map-data` token. The optional `type` attribute can have values of `auto`, `string`, or `numeric` and defines the resulting APXML token's type.

Elements	Description & Attributes
constants	Non-legacy Integration Services use the constants element to add or replace tokens in a submitted APXML message. In the case of a map-data submission, the constants are applied to the APXML message that results from applying the mapped-input. **Sub-elements**: • **<constant>**: 0 or more constants, each identifying an APXML token (order is unimportant). The `name` attribute defines the APXML token's key. The optional `type` attribute can have values of `auto`, `string`, or `numeric`, and defines the APXML token's type. If not specified, the `type` attribute defaults to `auto`. The optional `overwrite` attribute can have values of `true` or `false`, and controls whether the constant overwrites the APXML token if it already exists. If not specified, the `overwrite` attribute defaults to `false`.

Note: *Even if you are running the Integration Agent on Windows, it is recommended that you use Unix-style file path formatting, which works on both platforms.*

Adding fault tolerance to the Integration Agent

You can configure fault tolerance in your deployment by configuring multiple Integration Agents to share a common database in a master/slave configuration. In this state, any Integration Agent can receive injections from the management system.

When a message is injected, the Integration Agent adds it to a queue which is managed by an external database. The messages are removed from the queue by the "master" Integration Agent, processed, and sent to xMatters. If the master fails, one of the "slaves" assumes the role of the master, accepting the next message from the queue and processing it. Because the Integration Agents share a common set of queues which are not disrupted by the failure of any one Integration Agent, this feature is also referred to as a shared persistent queue.

Features and limitations

- Inbound messages are temporarily stored in an external database before being processed and sent to xMatters.
- Processed messages are also stored in the database until they can be sent to xMatters.
- At least one slave Integration Agent is required, ideally on a physically separate server (if permitted by the architecture of any integrations).
- Inbound messages can be submitted to the master or the slave Integration Agents (by a load balancer, for example), but only the master can remove the message from the queue and process it.
- The slaves must run the same version of the Integration Agent and the same integration scripts as the master.
- The iadmin command only works on the master Integration Agent; on slaves, the iadmin command will return a `Read timed out` error.
- When the Integration Agents are upgraded, the shared persistent queue information must be manually restored to each Integration Agent.
- Because the shared queue uses JDBC connections to an external database, the queues are not disrupted by the failure of any single Integration Agent, provided the database is accessible to the other Integration Agents after the master fails.
- Event deduplication will be performed even if a load balancer is used to distribute injected messages to multiple Integration Agents because injected messages are all processed by the master.

- If the master fails, event deduplication will be disrupted because deduplication is performed in-memory by the master Integration Agent, and statistics are lost if the master fails.
- Only Oracle and Microsoft SQL Server databases are supported in this configuration.
- When the master Integration Agent fails or is stopped, any messages in the inbound or outbound queues will remain in the queues until a new master is activated. A slave will only become the master, and begin releasing and processing queued messages, the first time it receives a new message (e.g., from `apclient.bin`).
- Callbacks from xMatters will always be directed back to the master Integration Agent; i.e., not the Integration Agent that initially received the incident request from the management system. This will cause callbacks to fail if the master Integration Agent is not connected to the management system that initiated the notification request.

Planning for fault tolerant mode

When the Integration Agent is configured to use a shared persistent queue (as it is in fault-tolerant mode), an external database is required. The database requirements of the Integration Agent are generally modest, but some system owners may want to determine some specific information before configuring the Integration Agent to use an external database.

Overview

Database requirements for the Integration Agent's stored persistent queues are typically dominated by the volume of outbound messages waiting to be sent to xMatters. These queues can be tested by first disabling communication between xMatters and the Integration Agent, and then sending a calculated quantity of messages from the management system to the Integration Agent. The effect on the built-in database can be determined by observing the files in the Integration Agent's .activeMQ-data folder. If the test is performed with the Integration Agent already configured to use a shared persistent queue, then the database administrator can use the tools appropriate to the DBMS in question to determine the database space requirements. Typical inbound queue requirements can be accommodated by doubling the estimate for the outbound queues.

Database usage

The Integration Agent uses its database to store the contents of its inbound and outbound queues. These queues store the following categories of data:

- Incident injections received from the management system.
- Processed incidents to be sent to xMatters, which may include enrichment data retrieved from the management system.
- Responses from xMatters, including annotations, user response choices, etc.

Incident injections and responses from xMatters are stored in the inbound queues, while processed incidents are stored in the outbound queues until they can be sent to the xMatters server.

Note: *For detailed information about the queues, see "Inbound Queue Model" on page 103 and "Outbound Queue Model" on page 108.*

Approximating database requirements

You can approximate maximum database requirements by simulating a worst-case network outage (i.e., between the Integration Agent and the xMatters server) and then observing the effect on the Integration Agent database. You can do this using the built-in database, or using a staging instance of the DBMS that will be used for the shared persistent queue.

Built-in database

The Integration Agent's built-in database manager (kahaDB) stores its data in `<IAHOME>/.activemq-data`. The queue data is typically written to a file named `db.data`, which grows as messages are added to the queues. This size of this file can be used to approximate the database size requirements for the shared persistent queue.

External database

If you are performing the test using an external DBMS (Oracle and SQL Server are supported), then the database administrator will be able to determine the effect of the test on the data tables. The specific tables used for the shared persistent queue are:

- ACTIVEMQ_ACKS
- ACTIVEMQ_LOCK
- ACTIVEMQ_MSGS

Performing the test

To create the test conditions, use the following steps to simulate a network outage on a staging Integration Agent:

1. In the `<primary-servers>` section of the `<IAHOME>/conf/IAConfig.xml` file, configure the Integration Agent to point to a non-existent (or unreachable) xMatters endpoint.
2. Start the Integration Agent.
3. Inject a representative quantity of incident or change messages from the management system.
4. Observe the effect on the database.

The number of messages injected should represent the longest expected network outage, multiplied by an above-average incident injection rate from the management system.

Note that in many integrations, the Integration Agent will send a query to the management system, supplementing the original injection with data from the response, before adding the message to the outbound queues. This will cause the outbound queues to contain significantly more information than what was initially injected, and may also cause a delay before the outbound queues reach their final size.

Estimating storage requirements

Using the above method, you can estimate storage requirements for outbound queues fairly accurately. Inbound queues can be more difficult to test, for the following reasons:

- The above method prevents messages from xMatters from reaching the inbound queues.
- Management system injections will not remain in the inbound queues unless you modify the Integration Agent's integration service scripts. These changes are difficult to describe concisely as the scripts vary widely between integrations.

Injections from the management system generally contain less information than is forwarded to xMatters, however, so it is probably safe to allow the same space for inbound queues as for outbound. In other words, doubling the data store requirements for the outbound queues should allow for the inbound queue requirements as well.

One condition to be aware of is the buildup of messages in the queues due to the Integration Agent's inability to process or deliver them. For example, if xMatters is configured to send an annotation to the management system every time a notification is delivered to a user and the management system cannot accept the annotations quickly enough, the annotations may build up in the Integration Agent's inbound queues over a period of many hours. This should be considered a pathological condition: if the management system's performance cannot be improved, then the business requirement for these delivery notifications should be weighed against the possibility of service disruption.

Configuring Integration Agents for shared persistent queue operation

Before configuring an Integration Agent for a shared persistent queue, ensure that all injected messages have been processed and sent to xMatters. Any messages that were persisted by the Integration Agent's built-in queue will not be available once the Integration Agent starts using the shared queue.

Note: *You must perform the following steps on each of the master and slave Integration Agents in your deployment.*

Before configuring the shared persistent queue, ensure that the Integration Agent is operating correctly, and communicating with xMatters.

To configure a shared persistent queue:

1. Stop the Integration Agent (for details, see "Stopping the Integration Agent" on page 39).
2. Navigate to the `<IAHOME>/conf` folder, and back up the following files:
   - `activemq.xml`
   - `spring-config.xml`
3. Delete the following folders and their contents:
   - `<IAHOME>/.activemq-data`
   - `<IAHOME>/.mule`
4. Open the `<IAHOME>/conf/activemq.xml` file in a text editor.
5. Locate and uncomment the following section:

```
<bean class="com.alarmpoint.integrationagent.config.IAPropertyPlaceholderConfigurer">
  <property name="isEncrypted" value="true"/>
  <property name="locations">
    <list>
      <value>file:/conf/dbpassword.properties</value>
    </list>
  </property>
</bean>
```

6. Locate the **<persistenceAdapter>** node.
7. Comment out the entire line that starts with **<kahaDB**; i.e., replace

```
<kahaDB directory="file:./.activemq-data" journalMaxFileLength="20mb"/>
```

   With

```
<!-- <kahaDB directory="file:./.activemq-data" journalMaxFileLength="20mb"/> -->
```

8. Uncomment the line starting with **<jdbcPersistenceAdapter**; i.e., replace:

```
<!--<jdbcPersistenceAdapter dataDirectory="activemq-data" dataSource="#oracle-ds"/>-->
```

   With:

```
<jdbcPersistenceAdapter dataDirectory="activemq-data" dataSource="#oracle-ds"/>
```

9. If you will be using SQL Server (and not Oracle), change the `dataSource` attribute to **#sqlserver-ds**.
   - The node should now be similar to the following (for Oracle):

```
<persistenceAdapter>
 <!-- <kahaDB directory="file:./.activemq-data" journalMaxFileLength="20mb"/> -->
 <jdbcPersistenceAdapter dataDirectory="activemq-data" dataSource="#oracle-ds"/>
</persistenceAdapter>
```

   Or (for SQL Server):

```
<persistenceAdapter>
 <!-- <kahaDB directory="file:./.activemq-data" journalMaxFileLength="20mb"/> -->
 <jdbcPersistenceAdapter dataDirectory="activemq-data" dataSource="#sqlserver-ds"/>
</persistenceAdapter>
```

10. Locate the line that begins with **<transportConnector** and replace `localhost` with the Integration Agent server's IP address.
11. If you believe that port 61618 is being used by another process (which is unlikely), replace the port number.
    - The line should now resemble the following:

```
<transportConnector name="default" uri="tcp://192.168.1.234:61618"/>
```

Note: *If you change the transportConnector's IP address and/or port, you must also change the activeMqConnectionFactory and activeMqConnectionFactoryOutbound settings in the spring-config.xml file. If these settings do not match, the Integration Agent will not start. For more information, see "Configuring the ActiveMQ connections", below.*

12. Locate the line that begins with **<bean id="oracle-ds"** if you are using Oracle, or **<bean id="sqlserver-ds"** if you are using SQL Server.
13. Modify the **url**, **username**, and **password** properties to match the desired database. The node should now appear similar to the following, where <IP_address>, <port>, and <username> are the correct values for your database installation:

```
<property name="url" value="jdbc:jtds:sqlserver://<IP_address>:<port>"/
<property name="username" value="<username>"/>
<property name="password" value="${db.password}"/>
```

14. Save and close the `activemq.xml` file.
15. Open a command line and run `<IAHOME>bin/iapassword` to create a file that will contain the encrypted password; in the following example command, replace **<filename>** with the actual file name you want to use for the text file:

```
bin/iapassword.bat --new my_password --file bin/<filename>.txt
```

16. In a text editor, open the file containing the encrypted password.
17. Copy the password.
18. Create an empty file named **dbpassword.properties** (in `<IAHOME>/conf/`).
19. Add the following text to the file:

```
db.password=
```

20. Paste the encrypted password after "db.password=". The text in the file should now look similar to the following:

```
db.password=XkhXfkeOrFaVOQA/qkARjA==
```

21. Save the file.

Note: *Patching the Integration Agent after applying these instructions will overwrite any configuration changes; you must reconfigure the queue again after applying any future patch.*

Configuring the ActiveMQ connections

The next step in configuring fault tolerance via the shared persistent queue is to define the Integration Agent connections for the ActiveMQ connection factory in the `spring-config.xml` file.

Note that the `spring-config.xml` file can be shared across all of the Integration Agents in your deployment. Once you have successfully configured the connections, you can copy the file to the other Integration Agents rather than individually configuring each one.

To configure the connections:

1. Stop the Integration Agent.
2. Open the `<IAHOME>/conf/spring-config.xml` file in a text editor.
3. Locate the section beginning with **<bean id="activeMqConnectionFactory"**.
4. Within the section, comment out the line with a single TCP address; i.e., replace the following line:

```
<property name="brokerURL" value="failover:
(tcp://localhost:61618)?jms.redeliveryPolicy.maximumRedeliveries=-
1&amp;jms.prefetchPolicy.queuePrefetch=1"/>
```

With:

```
<-- <property name="brokerURL" value="failover:
(tcp://localhost:61618)?jms.redeliveryPolicy.maximumRedeliveries=-
1&amp;jms.prefetchPolicy.queuePrefetch=1"/> -->
```

5. Uncomment the line with multiple TCP addresses; i.e., replace the following line

```
<!--property name="brokerURL" value="failover:
(tcp://host1:61616,tcp://host2:61617,tcp://host3:61618)?jms.redeliveryPolicy.maximumRedeliver
ies=-1&amp;jms.prefetchPolicy.queuePrefetch=1"/-->
```

With:

```
<property name="brokerURL" value="failover:
(tcp://host1:61616,tcp://host2:61617,tcp://host3:61618)?jms.redeliveryPolicy.maximumRedeliver
ies=-1&amp;jms.prefetchPolicy.queuePrefetch=1"/>
```

6. Replace each instance of **host#:** (i.e., host1, host2, host3, etc.) with the IP addresses of your master and slave integration agents. The order in which the Integration Agents are listed is not important, but each IP address and port must match the address of one of your Integration Agents configured to use the shared persistence queue.
7. Locate the section beginning with **<bean id="activeMqConnectionFactoryOutbound"**, and repeat steps 4 through 6 for the TCP addresses in this section.
8. Save and close the `spring-config.xml` file.
9. Start the Integration Agent in console mode (using `<IAHOME>/bin/start_console.sh` or `<IAHOME>\bin\start_console.bat`).
10. Confirm the JDBC datastore configuration was successful by verifying that the database includes the following new tables:

```
ACTIVEMQ_ACKS
ACTIVEMQ_LOCK
ACTIVEMQ_MSGS
```

Note: You may encounter an error similar to the following on startup:

```
2011-09-13 14:31:55,871 [Thread-2] WARN - Could not create JDBC tables; they could already
exist. Failure was: ALTER TABLE ACTIVEMQ_ACKS DROP PRIMARY KEY Message: Incorrect syntax near
the keyword 'PRIMARY'. SQLState: S1000 Vendor code: 156
2011-09-13 14:31:55,871 [Thread-2] WARN - Failure details: Incorrect syntax near the keyword
'PRIMARY'. java.sql.SQLException: Incorrect syntax near the keyword 'PRIMARY'.
...trailing stack trace...
```

You can safely ignore this error as harmless, but you may need to restart the Integration Agent (once only).

Verifying correct fault tolerance operation

After you have completed the shared persistence queue configuration steps, you can use the following points to ensure that your deployment is operating correctly and that fault tolerance has been enabled.

- When starting the first Integration Agent in console mode, expect to see the following messages:

```
Successfully started Mule with config located at ./conf/mule-config.xml


Successfully completed Integration Agent bootstrap process. Integration Agent is running.
```

- The first Integration Agent should not show any error messages or exceptions in the console as it is starting. Note that this Integration Agent will be the master, meaning that it can receive messages from the management system and add them to the inbound queue, and will also retrieve messages from the inbound queue, process them, and send them to xMatters.

- Any subsequently started Integration Agents will be slaves, meaning that they can only receive messages from the management system and add them to the inbound queue. You should not expect these Integration Agents to remove messages from the queue unless the master becomes unavailable.
- When starting the slave Integration Agents in console mode, expect to see the following message before the Success messages above:

```
[WrapperSimpleAppMain] WARN - Unable to read the JMS queues due to an exception...


java.lang.NullPointerException... at org.apache.activemq.broker.jmx.BrokerView.getQueues
(BrokerView.java:185)
```

- To test the injection process, run the following command in `<IAHOME>\bin`:

```
apclient.bin --map-data ping bsmith "hi from the master" 192.168.1.234 "sent by xmatters-
master at %time%"
```

  In the master Integration Agent's console window, you should see messages indicating that the injection was queued for processing and then another message similar to the following:

```
[inbound.ping.ping.normal-3] INFO - Component ping_ping has received the following request
from endpoint jms://inbound.ping.ping.normal...
```

- When the master is stopped, one of the slaves will start processing the messages in the queue as soon as the slave receives a new injection. The slave's console should show the following messages (after the first injection is received by the slave):

```
[ActiveMQ Transport: tcp:///<master-IP>:61618] WARN - Transport (/<master-IP>:61618) failed to
tcp://<master-IP>:61618 , attempting to automatically reconnect due to: java.io.EOFException
[Thread-2] INFO - ActiveMQ 5.4.2.5-XMATTERS JMS Message Broker (localhost) is starting
[Thread-2] INFO - Connector default Started
[Thread-2] INFO - ActiveMQ JMS Message Broker (localhost, ID:<Slave-name>-55874-1394821550270-
3:1) started
```

  These messages indicate that the former slave has assumed the master's duties because the former master is unreachable.

Load balancing and Integration Agents

If you are configuring multiple Integration Agents to provide fault tolerance, you may want to front the Integration Agents with a load balancer.

When setting up a load balancer between a management system and multiple Integration Agents, be sure to configure the load balancer in failover mode. This will ensure that notification requests are always handled by the same Integration Agent, which will facilitate correct handling of callbacks. The load balancer should only switch to a different Integration Agent if the "master" Integration Agent fails.

If you configure the load balancer in round-robin mode, the result will be failed callbacks.

Note: *In Shared Persistent Queue configuration, the Integration Agent that is targeted by the load balancer may not be the same Integration Agent that communicates with the xMatters instance. This is expected behavior and does not indicate a problem.*

Troubleshooting

This section contains information about common errors and solutions involving the configuration of the shared persistent queue.

Issue:

The Integration Agent fails to start and logs the following messages:

```
[Thread-2] INFO - ActiveMQ JMS Message Broker (localhost, ID:vic-vw-rename-56606-
1394840456721-2:1) started

[WrapperListener_start_runner] FATAL - Exit code 88: The Mule server took longer than 120
seconds to start...
```

Solution:

The most common cause to this issue is a mismatch between the settings in `spring-config.xml` and `activemq.xml`; most likely the server and port specified in `spring-config.xml` does not match the URLs in `activemq.xml`. Check your URLs and ensure that you follow the steps outlined in the preceding sections.

Note that if this happens on a slave Integration Agent and a master is already running, the slave will start, but failover will not succeed when the master is stopped. The messages on the slave will resemble the following:

```
[Thread-2] INFO - ActiveMQ JMS Message Broker (localhost, null) stopped

Exception in thread "Thread-2" java.lang.RuntimeException: java.io.IOException: Transport
Connector could not be registered in JMX: Failed to bind to server socket: tcp://<slave-
IP>:61618 due to: java.net.BindException: Cannot assign requested address: JVM_Bind
```

Issue:

The Integration Agents do not log specific information about the shared persistent queue during startup.

Solution:

To log specific shared persistent queue information, open the `<IAHOME>/conf/log4j.xml` file in a text editor and uncomment the following sections:

```
<logger name="org.springframework">
  <level value="INFO"/>
</logger>

<logger name="org.mule">
  <level value="INFO"/>
</logger>
```

Add the following lines:

```
<logger name="org.apache.activemq.broker">
  <level value="DEBUG"/>
</logger>
```

Save and close the file.

The master's console should now show a set of messages similar to the following when a slave is started:

```
[ActiveMQ Transport: tcp:///<Slave-IP>:55852] DEBUG - Setting up new connection id: ID:<Slave-
name>-55848-1394820787586-0:1, address: /<Slave-IP>:55852

[ActiveMQ Transport: tcp:///<Slave-IP>:55852] DEBUG - localhost adding consumer: ID:<Slave-
name>-55848-1394820787586-0:1:-1:1 for destination:
topic://ActiveMQ.Advisory.TempQueue,topic://ActiveMQ.Advisory.TempTopic

[ActiveMQ Transport: tcp:///<Slave-IP>:55853] DEBUG - Setting up new connection id: ID:<Slave-
name>-55848-1394820787586-0:2, address: /<Slave-IP>:55853

[ActiveMQ Transport: tcp:///<Slave-IP>:55853] DEBUG - localhost adding consumer: ID:<Slave-
name>-55848-1394820787586-0:2:-1:1 for destination:
topic://ActiveMQ.Advisory.TempQueue,topic://ActiveMQ.Advisory.TempTopic
```

Issue:

Cannot find specific information about the inbound and outbound queues.

Solution:

The integration agent logs should include specific information about the queues as long as the `log4j.xml` file has been configured as described in the preceding section.

For example, for information about the "ping" integration's inbound queue, look for lines similar to the following:

```
[BrokerService[localhost] Task] DEBUG - inbound.ping.ping.normal toPageIn: 1, Inflight: 0,
pagedInMessages.size 0, enqueueCount: 24, dequeueCount: 23
```

This message indicates that a message has been submitted to the Integration Agent's inbound queue, and is ready for processing by the integration service scripts. It also shows that 24 messages have been submitted to the inbound queue since the Integration Agent was started up, and 23 of them have been released for processing.

For information about the oubound queue, look for lines similar to the following:

```
[BrokerService[localhost] Task] DEBUG - outbound.ping.ping.normal toPageIn: 0, Inflight: 0,
pagedInMessages.size 14, enqueueCount: 16, dequeueCount: 2
```

This message indicates that 16 messages have been processed and sent to the outbound queue in preparation for transmission to xMatters since the Integration Agent was started, 2 of them have been sent to xMatters, and the other 14 are ready to be sent during the next APXML exchange. Note that the messages in the queues may be been placed there by any of the Integration Agents, and not necessarily by the Integration Agent that is logging the information.

Chapter 4: Administration

This chapter covers Integration Agent administration subjects.

Note: *You can execute Integration Agent batch and script files, including the IAdmin tool, from any directory by providing a fully-qualified path.*

Starting the Integration Agent

The following sections describe how to start the Integration Agent. For help with resolving startup problems, see "Troubleshooting Common Problems" on page 54.

Windows

You can start the Integration Agent as a Windows Service (named xMatters Integration Service), or run it as a standard Windows application.

Note: *When running the Integration Agent as a Windows Service, there is no console output. This means that you must use the log files to monitor agent activities (see"Integration Agent Log" on page 51).*

To start the Integration Agent as a Windows Service

1. Do one of the following:
    - Open Windows **Administrative Tools > Component Services,** right-click xMatters Integration Agent, and then click **Start**.
    - Double-click the `start_service.bat` file (located at: `<IAHOME>\bin`).
    - From a command line, run `start_service.bat` (located at: `<IAHOME>\bin`).
    - From a command line, run `startup.bat` (located at: `<IAHOME>\bin`). This command starts the service and issues a `get-status` request through IAdmin (for details, see "IAdmin Commands" on page 41)

The following table summarizes the exit codes for `startup.bat`:

Startup.bat Exit Codes

Exit Code	Description
0	Success.
30	Service already started.
35	Service not installed.
40	Service not stopped.
45	Service failed to start, but did not have a SERVICE_EXIT_CODE (e.g., problem with Windows Services Manager).
50	`get-status` failure (check logs).

Note: *For more exit codes, see "Troubleshooting Common Problems" on page 54.*

To start the Integration Agent as a standard Windows Application:

From a command line, run `start_console.bat` (located at: `<IAHOME>\bin`).

Unix

You can start the Integration Agent as a Unix daemon, or run it as a standard Unix application.

To start the Integration Agent as a Unix daemon:

Run the following from a command line:

```
./<IAHOME>/bin/start_daemon.sh
```

To start the Integration Agent as a Unix application:

Run the following from a command line:

```
./<IAHOME>/bin/start_console.sh
```

In both cases, the Integration Agent is run as the current user. When running the Integration Agent as a Unix daemon, there is no console output. This means that you must use the log files to monitor agent activities.

Error code 45

Typically, the Integration Agent is started by a non-privileged user. If the Integration Agent daemon is started using a root account and the hosting machine needs to be restarted for any reason, the service may not start after boot up and return an error code 45. This is caused by the root account owning some of the required folders and denying access to the non-privileged user.

To repair an error code 45 installation:

1. Log in as the root account and navigate to the `<IAHOME>` folder.
2. Run the following commands:

```
chown -Rf xm:xm .mule
chown -Rf xm:xm .activemq-data
chown -Rf xm:xm log/AlarmPoint.txt
```

3. Log out of the root account.
4. Restart the Integration Agent daemon as a non-privileged user.

Stopping the Integration Agent

The following sections describe how to stop the Integration Agent.

Windows

To stop the Integration Agent Service, do one of the following:

- Open Windows **Administrative Tools > Component Services,** right-click xMatters Integration Agent , and then click **Stop**.
- Run `stop_service.bat` (located at: `<IAHOME>\bin`) from a command line.
- Run `shutdown.bat` (located at: `<IAHOME>\bin`) from a command line.
  - This commands suspends the agent, waits 30 seconds if there are pending requests, terminates any requests, and then stops the service.

To stop the Integration Agent running as a Windows program:

Press **Ctrl+C** in the console window running the agent.

Unix

To stop the Integration Agent daemon, do one of the following:

- From a command line, run `./<IAHOME>/bin/stop_daemon.sh`.
- From a command line, run `./<IAHOME>/bin/shutdown.sh`.
  - This commands suspends the agent, waits 30 seconds if there are pending requests, terminates any requests, and then stops the service.

To stop the Integration Agent running as a Unix application:

Press **Ctrl+C** in the console window running the agent.

Runtime Administration

The Integration Agent includes a command line tool named IAdmin located at:

- **Windows:** `<IAHOME>\bin\iadmin.bat`
- **Unix:** `<IAHOME>/bin/iadmin.sh`

You can use this utility to issue commands to the Integration Agent after it has started.

Integration Service Runtime States

Each Integration Service running within the Integration Agent has a runtime state. These states are specific to the Integration Services running within a single Integration Agent.

Note: *The states displayed in the xMatters web user interface may be different because they represent the states of an Integration Service across all Integration Agents providing that service.*

The following table summarizes possible runtime states for Integration Services:

Integration Service Runtime States

State	Description
ACTIVE	Indicates that the Integration Service is able to access and process requests. An active Integration Service may be suspended via the IAdmin tool.

State	Description
SUSPENDED	Indicates that the Integration Service is properly configured, but has been manually set to deny requests. For example, the service might be suspended if a management system is undergoing maintenance (e.g., code upgrade). This allows you to work on one service and reload it without impacting other system components. A suspended Integration Services may be resumed and become active via the IAdmin tool. Note the following: ■ A suspended service rejects all mobile access component requests, external service requests, and APClient requests. ■ A suspended service rejects any mobile access component or direct external service requests. ■ A suspended service accepts, but does not process through input or response action scripting, APClient requests, indirect external service requests, and APXML responses from xMatters. Messages sent to a service in a SUSPENDED state will be queued, and then processed when the service is set to an ACTIVE state. ■ Outbound messages from a suspended service are still forwarded to xMatters through the message exchange process.
ERROR	Indicates that the Integration Service is improperly configured, in most cases due to a malformed configuration file or script syntax error. Once the cause of the error is identified and corrected, an Integration Service may be reloaded and become active via the IAdmin tool or by restarting the Integration Agent.

Note: *The IAdmin tool's* `suspend` *or* `resume` *command changes the service's runtime state between SUSPENDED and ACTIVE (for more information, see "IAdmin Commands" on page 41).*

IAdmin Commands

This section describes the available commands that can be appended to the iadmin command line tool.

get-status

You can use the `get-status` command to perform troubleshooting, or to view status information about the Integration Agent. The get-status command displays the following information:

- Version and build number
- Release date
- Agent start date/time
- Agent identifier
- Event Domain list
- Integration Services list (per Event Domain)
  - Integration Service name
  - Clients
  - Integration Service Request URL
  - Integration Service Start Date/Time

    - Last active (last time a request was received)
    - Status (ACTIVE/SUSPENDED/ERROR)
    - Number of requests being processed
    - Inbound and outbound APXML queue sizes
- xMatters Primary Server List
    - Server URL
    - Server connectivity status (see related table below)
    - Last Heartbeat
- xMatters Secondary Server List
    - Server URL
    - Server connectivity status (see related table below)
    - Last Heartbeat

Server Connectivity Status

Command Syntax	Description
UNKNOWN	No connection attempt has yet been made, or the attempt has not been completed.
FAILED	No connection can be made between the Integration Agent and any of the primary or secondary xMatters web servers. The Integration Agent continues sending heartbeats, and a Health Monitor notification is sent when the heartbeat recovers.
PRIMARY_ CONNECTED	There is a connection between the Integration Agent and a primary xMatters web server, but the heartbeat generates an error. The Integration Agent continues to send heartbeats to the primary servers, but functionality may be limited until the heartbeat is fully accepted. A Health Monitor notification is sent when the heartbeat is fully accepted (consult the Integration Agent log for details).
PRIMARY_ ACCEPTED	The Integration Agent has successfully sent a fully accepted heartbeat to a primary xMatters web server, and the Integration Agent is fully functional.
SECONDARY_ CONNECTED	No connection can be made between the Integration Agent and any of the primary xMatters web servers, but a connection can be made to a secondary web server, and the heartbeat generated an error. The Integration Agent continues to send heartbeats to the secondary servers, but functionality may be limited until the heartbeat is fully accepted. A heartbeat to the primary servers is reattempted based on the configured recovery interval, or if the secondary heartbeat fails. AHealth Monitor notification is sent when the primary heartbeat recovers or the secondary heartbeat is fully accepted (consult the Integration Agent log for details).
SECONDARY_ ACCEPTED	No connection can be made between the Integration Agent and any of the primary xMatters web servers. However, the heartbeat was fully accepted by a secondary web server, and the Integration Agent is fully functional in failover mode. The Integration Agent continues to send heartbeats to the secondary servers. A heartbeat to the primary servers is reattempted based on the configured recovery interval, or if the secondary heartbeat fails. A Health Monitor notification is sent when the primary heartbeat recovers.

Suspend, Resume, Reload, and Purge Commands

The following table summarizes the other available IAdmin commands:

Command Syntax	Description
display-settings	Displays the settings that are currently in use for the Integration Agent.
suspend <domain> <service>	Suspends the specified Integration Service; incoming requests to the service are refused, but pending requests are maintained.
suspend all	Suspends all active Integration Services; incoming requests to all services are refused, but pending requests are maintained.
suspend-now <domain> <service>	Suspends the specified Integration Service; incoming requests to the service are refused, and pending requests are immediately terminated.
suspend-now all	Suspends all active Integration Services; incoming requests to all services are refused, and pending requests are immediately terminated.
resume <domain> <service>	Resumes the specified Integration Service; only SUSPENDED or ACTIVE services can be resumed.
resume all	Resumes all suspended Integration Services.
reload <domain> <service>	Reloads the configuration file for the specified Integration Service: • If the IAConfig.xml file includes the specified service, the service is reloaded (or created and loaded if the service is new). • If the IAConfig.xml file does not include the specified service, but the service had been previously created and loaded, then it is removed.
reload all	Reloads the Integration Agent's configuration file. This effectively removes any Integration Services that are no longer included in IAConfig.xml, creates and loads any new Integration Services, and reloads any existing Integration Services. Additionally, all of the Integration Agent configuration setttings are updated, except for the admin-gateway, heartbeat-interval, and id elements.
purge <domain> <service>	Removes all inbound and outbound APXML messages from the specified Integration Service; APXML messages that are being processed are maintained.
purge all	Removes all inbound and outbound APXML messages from all Integration Services; APXML messages that are being processed are maintained.

Note: *Unlike other IAdmin commands,* `purge` *can be executed even if the Integration Agent is not running.*

Error Codes

The IAdmin command line tool returns an exit code number (meant to be used in automation scripts) indicating either success, or the type of error encountered, as described in the following table:

Error Code	Description
0	Success (no pending requests)

Error Code	Description
30	Success, at least one pending request
35	Invalid arguments; check logs for details
40	IA Config error; check logs for details
45	Command failed; check logs for details

IAdmin Logging

The IAdmin tool captures all administrative commands and responses and produces output to the console and to a rolling set of log files located at:

- **Windows:** `<IAHOME>\log\AlarmPointIAdmin.xml.#` and `<IAHOME>\log\AlarmPointIAdmin.txt.#`
- **Unix:** `<IAHOME>/log/AlarmPointIAdmin.xml.#` and `<IAHOME>/log/AlarmPointIAdmin.txt.#`

Note: *The .txt files are designed to be human-readable, whereas the XML log files are designed to be used by a log-viewing utility (e.g., Chainsaw). In some versions of the Integration Agent, XML logging is disabled. For details, see "Enabling XML Logging" below.*

The results of administrative commands are displayed on the console and captured in the log files. Additionally, exit codes are captured in the log files. If errors occur during the execution of a command, a brief description of the error is displayed on the console; further details are contained in the log files.

Enabling XML Logging

In some versions of the Integration Agent, XML logging is disabled by default (this can increase performance in high-throughput scenarios). However, you can enable XML logging and parse the resulting log files with tools such as Chainsaw (the files are in standard log4j XML format).

To enable XML logging:

1. Open the agent's `conf/log4j.xml` file in a text editor.
2. In the **root category** definition near the end of the file, un-comment the following line:

```
<appender-ref ref="xmlAppender"/>
```

3. Save and close the file.
4. Open the agent's `conf/cli/log4j.xml` file in a text editor and repeat steps 2 and 3.

The IAPassword Utility

As described in "Integration Service Configuration File" on page 24 and "Applying Constants to APXML Messages" on page 100, both the Integration Agent configuration file and the Integration Services' configuration files may include configuration settings that refer to files containing encrypted data.

Note: *Before using the Integration Agent, you must run the IAPassword utility twice: once to create a .wspasswd file, and once to create a .passwd file (or any named file that matches the configuration in IAConfig.xml). For more information, see ."Integration Agent Configuration File" on page 18*

For example, the xMatters Web Service User Account that the Integration Agent uses whenever it calls the `RegisterIntegrationAgent`, `SubmitAPXML`, or `ReceiveAPXML` web service methods is configured by the `web-services-auth` element of the Integration Agent's configuration file, as shown here:

```
<web-services-auth>
  <user>IA_User</user>
  <password><file>./.wspasswd</file></password>
  <company>Default Company</company>
</web-services-auth>
```

To avoid storing the Web Service User's password in cleartext, the Integration Agent will decrypt the `.wspasswd` file and use the single string contained within as the `password` element's actual value.

Similarly, an Integration Service's configuration file can contain `encrypted-constant` elements whose values also come from encrypted files, as shown in the following example:

```
<constants>
  <constant name="device" type="string" overwrite="false">localhost</constant>
  <constant name="my_first_constant">This is an auto-typed constant...</constant>
  <encrypted-constant name="my_second_constant" type="string"             overwrite="true">
    <file>/tmp/.constant</file>
  </encrypted-constant>
</constants>
```

The IAPassword utility is a platform-independent Java program that an Integration Agent administrator can use to create and modify the contents of encrypted files. The installer stores the IAPassword program in the following location:

- **Windows**: `<IAHOME>/bin/iapassword.bat`
- **Unix**: `<IAHOME>/bin/iapassword.sh`

IAPassword accepts the following command-line parameters:

IAPassword Parameters

Parameter	Required	Description
–new <string>	Yes	Specifies the string that will be stored in the encrypted file.
–old <string>	Only if the file already exists	Specifies the current string that is stored in the encrypted file.
–file <path>	No	The path of the encrypted file: • Can be specified as absolute, or as relative to the installation folder (i.e., rather than relative to the current directory) • If not specified, the default value is `<IAHOME>/conf/.passwd` (if you want to change the Web Service password, see the example below)

For example, to change the Web Service User's password referred to in the previous example, issue the following command:

```
iapassword --new "My New Password" --old ia_complex --file conf/.wspasswd
```

This command changes the contents of the file `<IAHOME>/conf/.wspasswd` to the string "`My New Password`" (without quotes).

To create the `encrypted-constant` file from the previous example, issue the following command:

```
iapassword --new "This is a string constant..." --file /tmp/.constant
```

Whenever `iapassword` is executed, it logs messages via log4j to the `<IAHOME>/log/ AlarmPointIAPassword.txt` and `<IAHOME>/log/AlarmPointIAPassword.xml` files. This logging behavior can be changed by modifying the log4j configuration file, `<IAHOME>/conf/cli/ iapassword/log4j.xml`.

Portable Filtering and Suppression Module

The Portable Filtering and Suppression Module is a built-in module that maintains a rolling record of previously injected events, and allows for the suppression of duplicates (also referred to as "deduplication"). This helps avoid disruption of xMatters traffic due to inadvertent loads that can result when, for example, improperly configured management systems inject duplicated events.

Configuration

To configure the module, add your required filters to the `deduplicator-filter.xml` file (located at `<integration agent>/conf/`). You can add any number of filters, each of which must consist of the following filter attributes:

- **predicates:** a list of predicates that are considered relevant for the purpose of correlation.
  - Each predicate must be wrapped in <predicate> tags (see the Filter Use Cases section, below, for an example of the syntax).
  - There must be at least one predicate in the list.
  - If the predicate does not exist in the message from the management system, the message will not match the filter and will not be suppressed.
  - Predicate matching is case insensitive.
- **suppression_period:** how long, in seconds, that the system should suppress duplicates.
- **window_size:** the maximum number of unique events to record.

Filtering Process

When the integration service processes a message, the filter records the values of the specified predicates. For example, in the use cases below, the "node" and "person_or_group_id" predicates.

Each subsequent message is processed by the integration service and suppressed if its predicates are identical to the initial message AND the suppression period has not expired AND the number of messages with predicates that did not match the initial message is less than the specified window size.

In other words, once the integration service has processed a message, the filter will block all subsequent messages unless:

- their predicates differ from those in the initial message OR
- the suppression period has expired OR
- the number of unique messages exceeds the window size.

Filter Use Cases

Assume that the `deduplicator-filter.xml` file includes the following filter:

```
<filter name="default">
  <predicates>
     <predicate>node</predicate>
     <predicate>person_or_group_id</predicate>
  </predicates>
  <suppression_period>180</suppression_period>
  <window_size>1000</window_size>
</filter>
```

The following use cases are based on this filter configuration.

Use Case 1: Duplicates Detected (Suppression Period elapses)

1. At 14:00:00 an event is injected with predicates (node, "Node123"), (person_or_group_id, "bsmith").
2. The event is not considered a duplicate.
3. At 14:02:00 an event is injected with predicates (node, "Node123"), (person_or_group_id, "bsmith").
4. The event is considered a duplicate and is suppressed.
5. At 14:04:00 an event is injected with predicates (node, "Node123"), (person_or_group_id, "bsmith").
6. The event is not considered a duplicate, because the suppression period has elapsed (the suppression period is calculated based on the original event, not the second event).

Use Case 2: Duplicate Not Detected (Irrelevant Predicates)

1. At 14:00:00 an event is injected with predicates (node, "Node123"), (location, "NYC").
2. The event is not considered a duplicate.
3. At 14:01:00 an event is injected with predicates (node, "Node123"), (location, "NYC").
4. The event is not considered a duplicate, because while the predicates are identical for both events, the `location` predicate is not relevant for the purpose of correlation for this filter.

Use Case 3: Events Not Suppressed (Window Rolls)

1. At 14:00:00 an event is injected with predicates (node, "Node123"), (person_or_group_id, "bsmith").
2. By 14:01:00, 1000 distinct events are injected
3. None of these 1000 events are suppressed.
4. At 14:02:00 another event with predicates (node, "Node123"), (person_or_group_id, "bsmith") is injected.
5. The event is not suppressed, because the first event (injected at 14:00:00) was purged from the list when the 1001st event was injected.

How to use Filtering and Suppression with the sample integration

The Integration Agent includes an example (located at `<integration agent>/integrationservices/sample/sample.js`) of how to use the Filtering and Suppression Module with the `default|sample` integration.

Note: *Before working with this example, read the WARNING section below.*

To use the module with an existing integration, copy the desired code from the sample integration and paste it into the existing integration. Modify the code to achieve the desired behavior (ensure that you include the appropriate import statement).

The following example code snippet from `sample.js` shows the relevant parts:

```
// import the correct code
importClass(Packages.com.alarmpoint.integrationagent.util.
   EventDeduplicator);
// xMatters user or group that will receive notifications of event
     overflow
var stormRecipient = "bsmith";
// number of events when we have to send a notification due to
     overflow
var stormTriggerCount = 1000;
...
function apia_input(apxml)
{
  // aggregate all instances under the filter named "default"
  var recentlyReceived = EventDeduplicator.getInstance().
    recentOccurrenceCount(apxml, "default")
  // if we haven't got more than 1 event in the programmed timeout
  if (recentlyReceived <= 1)
  {
     // actually do the processing of the event
     ...
    }
    else if (recentlyReceived == stormTriggerCount)
    {
           // modify the event to indicate the duplicated events might
             be flooding the system

           return createStormNotification(apxml, recentlyReceived);

       ...
    }
    // else ignore it, or send an email, or log, or whatever else might
     be appropriate
    ...
```

Edit the `deduplicator-filter.xml` file to specify the predicates to be filtered, as well as the related the suppression period and window size.

WARNING

The sample code snippet above sends an event storm notification to the domain/service `messaging|flood` that needs to be created if the example is not modified before use (which is not recommended). You must add a new integration service called `flood` under the messaging domain.

The integration service uses a relay service (i.e., relaying the forwarding to that same `messaging|flood` service). This means that when enabling the `sample/sample.xml` integration service, `sample/messaging.xml` must be included in `<integration agent>/conf/IAConfig.xml`.

Note that the sample integration may be overwritten during upgrades and should not be used for production.

Disabling the filtering and suppression module

The Integration Agent version 5.1 patch 005 introduced the ability to disable the filtering and suppression module on a per integration basis.

To disable filtering in version 5.1.5 (and later):

1. Open the integration's `configuration.js` file in a text editor.
2. Comment out the line containing the DEDUPLICATION_FILTER_NAME variable.
3. Save and close the file.
4. Restart the Integration Agent.

If the DEDUPLICATION_FILTER_NAME variable does not exist in the `configuration.js` file, use the method described below to disable the module.

Disabling older versions

For versions of the Integration Agent prior to 5.1.5, you can disable the filtering and suppression module by adding a predicate to the filter that is never created by the integration service. For example:

```
<predicates>
    <predicate>no_matches</predicate>
<predicate>
```

Provided that the integration does not include a predicate called "no_matches", the integration service will not suppress any messages.

Health Monitor

The Integration Agent includes a component called the Health Monitor that sends messages about important agent events to a specified email address (for details about configuring Health Monitor settings, see "Integration Agent

Configuration File" on page 18). Note that the Health Monitor sends messages in the order their associated events occurred.

Health Monitor Events

Health Monitor messages may involve events related to the heartbeat settings configured in the heartbeat element of the Integration Agent configuration file, or related to a specific Integration Service. The following table summarizes events that trigger Health Monitor messages:

Health Monitor Triggering Events

Event	Description
Connection failure	Occurs when no connection can be made between the Integration Agent and any of the primary or secondary xMatters web servers. The Integration Agent continues sending heartbeats, and a notification is sent when the heartbeat recovers.
Primary Web Server Connected (with an error)	Occurs when a connection is made between the Integration Agent and a primary xMatters web server, but the heartbeat generates an error. The Integration Agent continues to send heartbeats to the primary servers, but functionality may be limited until the heartbeat is fully accepted. A notification is sent when the heartbeat is fully accepted (consult the Integration Agent log for details).
Secondary Web Server Connected (with an error)	Occurs when no connection can be made between the Integration Agent and any of the primary xMatters web servers, but a connection can be made to a secondary web server, and the heartbeat generated an error. The Integration Agent continues to send heartbeats to the secondary servers, but functionality may be limited until the heartbeat is fully accepted. A heartbeat to the primary servers is reattempted based on the configured recovery interval, or if the secondary heartbeat fails. A notification is sent when the primary heartbeat recovers or the secondary heartbeat is fully accepted (consult the Integration Agent log for details).
Primary Web Server Accepted	Occurs when the Integration Agent successfully sends a fully accepted heartbeat to a primary xMatters web server, and the Integration Agent is fully functional.
Secondary Web Server Accepted	Occurs when no connection can be made between the Integration Agent and any of the primary xMatters web servers. However, the heartbeat was fully accepted by a secondary web server, and the Integration Agent is fully functional in failover mode. The Integration Agent continues to send heartbeats to the secondary servers. A heartbeat to the primary servers is reattempted based on the configured recovery interval, or if the secondary heartbeat fails. A notification is sent when the primary heartbeat recovers.
Service request failure	Occurs whenever an unanticipated exception (i.e., exceptions other than those such as timeouts, unavailable services, etc.) is thrown during the processing of an Integration Service request.
Service failure	Unhandled exception that occurs within an Integration Service, but not related to an Integration Service request.

Event	Description
Outbound queue threshold exceeded (or reduced below threshold)	Occurs when the outbound queue size has exceeded its configured threshold (and when the queue size is subsequently reduced below its threshold). The outbound queue maximum threshold is set in the IAConfig.xml file (for details, see "Integration Agent Configuration File" on page 18).

Sample Integration Agent Health Monitor Messages

In the following examples, angle brackets denote placeholders for actual values:

- At <timestamp>, No connection could be made between Integration Agent <URL> and any of the primary or secondary xMatters Web Servers. The Integration Agent will continue sending heartbeats, and a notification will be sent when the heartbeat recovers.
- At <timestamp>, Integration Agent <URL> successfully sent a fully accepted heartbeat to primary xMatters Web Server <URL>. The Integration Agent is fully functional. A notification will be sent if the heartbeat status changes.
- At <timestamp>, A connection was made between Integration Agent <URL> and primary xMatters Web Server <URL>, but the heartbeat generated the following error: The xMatters Server at <URL> rejected the heartbeat/registration with the following reason: AUTHENTICATION_ERROR. The Integration Agent will continue to send heartbeats to the primary servers, but Integration Agent functionality may be limited until the heartbeat is fully accepted. A notification will be sent when the heartbeat is fully accepted. Consult the Integration Agent log for further details.
- At <timestamp>, No connection could be made between Integration Agent <URL> and any of the primary xMatters Web Servers. The heartbeat was fully accepted by secondary xMatters Web Server <URL>. The Integration Agent is fully functional in failover mode. The Integration Agent will continue to send heartbeats to the secondary servers. A heartbeat to the primary servers will be reattempted every 600 seconds (0 indicates indefinitely), or if the secondary heartbeat fails. A notification will be sent when the primary heartbeat recovers.
- At <timestamp>, No connection could be made between Integration Agent <URL> and any of the primary xMatters Web Servers. A connection was made to secondary xMatters Web Server <URL>, but the heartbeat generated the following error: The xMatters Server at <URL> rejected the heartbeat/registration of the following Integration Services: Domain: ping, Name: ping, Reason: UNKNOWN_SERVICE . The Integration Agent will continue to send heartbeats to the secondary servers, but Integration Agent functionality may be limited until the heartbeat is fully accepted. A heartbeat to the primary servers will be reattempted every 600 seconds (0 indicates indefinitely), or if the secondary heartbeat fails. A notification will be sent when the primary heartbeat recovers or the secondary heartbeat is fully accepted. Consult the Integration Agent log for further details.

Note: *For help resolving heartbeat and Integration Service issues, see "Troubleshooting Common Problems" on page 54.*

Health Monitor Fault Tolerance

The Health Monitor employs two forms of fault tolerance:

- **Outbound message persistence**: Health Monitor messages are stored in a persistent queue. If the Integration Agent is shut down for any reason, queued Health Monitor messages are sent upon restart.
- **Resend Policy**: If the SMTP server is unavailable, the Health Monitor atttempts to resend the message every 30 seconds for five minutes. After five minutes without success, the failure to send the message is logged as an error and the message is discarded.

Integration Agent Log

The Integration Agent uses log4j version 1.2.14 for logging. By default, logging is configured to produce only warnings and higher to the console and rolling log files located at:

- **Windows:** `<IAHOME>\log\AlarmPoint.xml.#` and `<IAHOME>\log\AlarmPoint.txt.#`
- **Unix:** `<IAHOME>/log/AlarmPoint.xml.#` and `<IAHOME>/log/AlarmPoint.txt.#`

The text log files are designed to be human-readable; the XML files are designed for use by a log-viewing utility (e.g., Chainsaw). However, in some versions of the Integration Agent, XML logging is disabled (for details, see "Enabling XML Logging" on page 44).

Note: *This section covers log4j configuration items only as they pertain to the Integration Agent. For full details about log4j and its settings, refer to its documentation (see* `http://logging.apache.org/log4j/1.2/manual.html`*).*

Default Log Entry Format

Log entries are in the following format:

```
<date> <time> <thread> <log_level> <log_message>
```

Note: *This format is defined in the* `log4j.xml` *file (located at:* `<IAInstall_Dir>/conf`*); refer to the log4j documentation for details about modifying the format.*

Example

The following represents a log entry in the default format:

```
2007-11-06 17:24:49,649 [connector.http.0.receiver.2] DEBUG - Successfully processed
Integration Service Request without any exceptions.  Returning the result to the caller.
```

Logging Configuration File

The logging configuration file is named `log4j.xml` and is located at:

- **Windows:** `<IAHOME>\conf`
- **Unix:** `<IAHOME>/conf`

Changes to the log file are automatically detected within 10 seconds.

log4j Categories

Most Integration Agent log messages appear under the `com.alarmpoint.integration` category hierarchy. Each major component or activity (e.g., Health Monitor, heartbeats, Integration Service Requests, etc.) logs to a dedicated sub-category. The advantage to this approach is that logging can be focused on a specific aspect of the Integration Agent's activities.

The following table summarizes selected log4j categories:

Logging Categories

Category	Logs
com.alarmpoint.integrationagent	All Integration Agent activity; by default, this is the only active logging category, and is set to “WARN”. Note: in the entries that follow, **<root_AP_ cat>** represents **com.alarmpoint.integrationagent**
<root_AP_cat>.admin	IAdmin tool requests
<root_AP_cat>.apclient	`APClient.bin` submissions
<root_AP_cat>.health	Health Monitor activity
<root_AP_cat>.health.mail	Health Monitor mailer activity
<root_AP_cat>.heartbeat	Heartbeats activity
<root_AP_cat>.messaging	`apxml-exchange` activity
<root_AP_cat>.services	All Integration Services activity
<root_AP_cat>.services.MyDomain	Example for specific integration; logs all Integration Services in the MyDomain Event Domain.
<root_AP_cat>.services.MyDomain.test_service_1	Example for specific Integration Service (test_ service1) in the MyDomain Event Domain.

To expose a specific logging category, open the log4j configuration file and uncomment one or more logging categories.

Example

By default, the Health Monitor logging category is inactive because it is commented out in the `log4j.xml` file:

```
<!--
   <logger name="com.alarmpoint.integrationagent.health">
    <level value="ALL"/>
   </logger>
-->
```

To enable Health Monitor logging, uncomment the entry and save the file:

```
   <logger name="com.alarmpoint.integrationagent.health">
    <level value="ALL"/>
   </logger>
```

Adding an integration service

Adding an integration service to the Integration Agent is a three-step process:

1. Place the integration service configuration file (and any additional files) in the `<IAHOME>/integrationservices` folder, or a subfolder.
   - Ensure that the event domain and name in the integration service configuration file matches the domain and name specified in the xMatters web servers.
2. Open the `IAConfig.xml` file and add a new path element to `service-configs` that references the integration service configuration file from step 1.
   - The path is relative to the `<IAHOME>/integrationservices` folder (on Windows, `<IAHOME>\integrationservices`)
3. Restart the Integration Agent or use the IAdmin tool to issue the following command

```
reload <domain> <service_name>
```

Example

For this example, assume the following configuration:

- The new integration service is named `service_new` and is in the `default` event domain.
- The integration service's configuration and associated file are installed in `<IAHOME>/integrationservices/service_new`
- The integration service configuration file is called `service_new.xml`.

Open the `IAConfig.xml` file and add a new path element to `service-configs` for `service_new`:

```
<service-configs dir="../integrationservices">
  <path>service_new/service_new.xml</path>
</service-configs>
```

Note: *The* `service-configs` *element can have multiple path elements, depending on the number of integration services added.*

To finish adding the service, restart the Integration Agent or use the IAdmin tool to issue the following command

```
reload default service_new
```

Autoloaded Integration Services

Normally, the Integration Services that are loaded by the Integration Agent must be specified within the `<service-configs>` element of the Integration Agent Configuration File (`IAConfig.xml`). The autoloaded Integration Service feature is a mechanism to deploy new Integration Services without requiring `IAConfig.xml` modifications.

When the Integration Agent starts or is reloaded, it checks for any XML files within the following folder:

```
<IAHOME>/integrationservices/autoloaded
```

Any file within this folder with the `.xml` (case-sensitive) extension is assumed to be an Integration Service configuration file and is loaded by the Integration Agent as if its path had been specified in the `IAConfig.xml`'s `<service-configs>` element.

Integration Services that are specified within the `<service-configs>` element are always loaded by the Integration Agent before autoloaded Integration Services. As a result, the Integration Agent ignores any autoloaded Integration Service with a fully-qualified name (i.e., `<domain>|<name>`) that matches an existing Integration Service.

For example, assume the installation folder has the following structure:

```
<IAHOME>/
  integrationservices/
    sample/
      sample.xml
      sample.js
    ping/
      ping.xml
      ping.js
    autoloaded/
      del.xml
      del/
        del.js
      ping.xml
      ping/
        ping.js
```

The configuration file contains the following `<service-configs>` element:

```
<service-configs dir="../integrationservices">
  <path>sample/sample.xml</path>
  <path>ping/ping.xml</path>
</service-configs>
```

When the Integration Agent starts, it will load the `default|sample` and `ping|ping` Integration Services as specified by the `<service-configs>` element. The Integration Agent will then scan the `integrationservices/autoloaded` directory and attempt to load the Integration Service configuration files `del.xml`, which defines the `del|del` Integration Service, and `ping.xml`, which defines a modified version of the `ping|ping` Integration Service.

Since no `del|del` Integration Service was specified by the `<service-configs>` element, this Integration Service will be loaded. However, since a `ping|ping` Integration Service already exists, the autoloaded `ping|ping` Integration Service will be ignored even though its implementation may differ from the existing `ping|ping` Integration Service.

Troubleshooting Common Problems

The following section describes how to approach common problems you may encounter when using the Integration Agent.

Startup Issues

On startup, the Integration Agent validates its configuration, including the following:

- Presence of the `log4j.xml` configuration file
- Presence and well-formedness of the `IAConfig.xml` file
- Availability of the Admin and Integration Agent port

Note: *Not all incorrect configurations prevent startup. For example, problems with the xMatters web server URLs or the Integration Service configuration files are logged as errors, but do not prevent the Integration Agent from starting.*

Under Windows, the service state is “Starting” during this initial validation, and does not change to “Started” until the validation is complete. If the agent cannot start, the reason for the failure is written to the Windows system event log and an exit code is set on the xMatters Integration Agent Service (if the agent was started as a console application, the agent’s exit code is set as the batch file’s exit code).

To see the Window’s service exit code, run the following from a command line:

```
sc query apia
```

The code will appear in the `SERVICE_EXIT_CODE` field. Alternatively, if the `shutdown.bat` script is used to start the Integration Agent, the batch file will return the service exit code.

Under Unix, the initial validation is part of the daemon process. Any problems preventing startup cause the daemon to terminate. Details regarding startup problems are sent to the `syslog` daemon through the `user` facility at the `fatal` level and with the identity `apia`. Note that the `syslog` daemon must be configured by an administrator to perform logging.

Additionally, on both Windows and Unix all startup logging is written to the console and a special log file named `apia.txt` located in the `log` directory. After startup is complete, logging reverts to the standard Integration Agent log files (for details, see "Integration Agent Log" on page 51).

The following table summarizes startup exit codes:

Startup Exit Codes

Startup Exit Code	Description
0	Success
30	Service already started
35	Service not installed
40	Service not stopped
45	Service failed to start, but did not have a SERVICE_EXIT_CODE
50	get-status failure (check logs)
60	Missing configuration file
61	Unreadable configuration file (i.e., file is locked)
65	Malformed configuration file
70	Missing or unreadable log4j properties file
80	Unable to bind to Admin Gateway port
85	Unable to bind to Web Services Gateway port
86	Malformed Mule configuration file
87	Mule startup error
88	Mule timeout error
90	Nonspecific startup error (check logs)

Unix Process Issues

In some cases, a user may be unable to start the Integration Agent due to an “AlarmPoint_Integration_Agent is already running” exception when the Integration Agent is not actually running. This is because the Integration Agent may not have been stopped properly (e.g., due to a power failure). To verify that the Integration Agent is not running, search for a process with the argument containing the keyword “wrapper” or “java” that refers to the Integration Agent’s install directory.

On some Unix systems, this can be done by reviewing the output of the following commands:

```
ps -ef | grep wrapper
ps -ef | grep java
```

If such processes exist, then the Integration Agent is running and must be stopped before it is restarted.

If no such process exists, then deleting the file located at `<IAHOME>/lib/mule-1.4.3/bin/ .apia.piv` should allow the Integration Agent to start.

Integration Service Request Issues

The following sections cover the two main types of Integration Service issues that can occur.

Integration Service Request Receives No SOAP Response

Usually this type of issue indicates one of the following:

- **Improperly installed Integration Service:** For example, caused by a problem with the Integration Agent or Integration Service configuration
- **Integration Service addressing problem:**Integration Agent is reachable, but the URL used to specify the Web Service Gateway is invalid (e.g., incorrectly named Integration Service)
- **Network failure:**Integration Agent unreachable from the client (e.g., the xMatters mobile access)

To troubleshoot Integration Service issues:

1. Ensure the Integration Agent has started, and issue a `get-status` command using IAdmin.
2. Verify that the Integration Service appears in the list of services and is ACTIVE.
   - If the service does not appear in the list or has an ERROR_ACTIVE or ERROR_INACTIVE status, then consult the Integration Agent log to determine the nature of the configuration problem.
3. Attempt to reach an Integration Service by opening a web browser and typing the URL of the service followed by `?wsdl` in the address bar. For example, assume that an Integration Service has the following configuration:
   - **Name:** service_new
   - **Event Domain:** default
   - **IAConfig.xml:** <service-gateway ssl="true" host="www.company.com" port="8081"/>

To attempt to contact this service, type the following into a web browser's address bar (preferably from a computer in the same location as the client):

```
https://www.company.com:8081/service_new?wsdl
```

If the Integration Service is properly installed and accessible, then a response similar to the following is returned:

```
<?xml version="1.0" encoding="UTF-8" ?> <wsdl:definitions target...<snip>
```

If this response is received, it is likely that the problem is related to an incorrectly configured Integration Agent client (e.g., the xMatters mobile access)

If this response is not received, it is likely that the problem is related to a connectivity issue (e.g., connection prevented by a firewall) between the Integration Agent client and the Integration Agent.

Integration Service Receives SOAP Error Response

This indicates that the Integration Service is able to receive requests, but not process them.

To troubleshoot this issue:

1. Ensure the Integration Agent has started, and issue a `get-status` command using IAdmin.
2. Verify that the Integration Service appears in the list of services and is ACTIVE.
   - If the service is in any other state, then the expected behavior is for the service to deny the request and respond with a SOAP error indicating that the service is not able to process the request.

3. In the Integration Agent's log configuration file, set (or add) a DEBUG category for the specific Integration Service. For example, for an Integration Service named service_new in the default Event Domain, the `log4j.xml` entry would be similar to the following:

```
<logger name="com.alarmpoint.integrationagent.services.default.service_new">
  <level value="DEBUG"/>
</logger>
```

4. Save the log configuration file and wait at least 10 seconds to allow the change to be detected.
5. Submit a new Integration Service request, and then review the log file to determine the nature of the problem.

Heartbeat Issues

Usually this type of issue indicates one of the following:

- **Incorrect Integration Agent configuration:** For example, the xMatters web server URL is improperly specified
- **Connectivity issue** between xMatters web server and the Integration Agent
- **Incorrect xMatters mobile access configuration:** For example, the mobile access component has not been configured for one of the Integration Services

To troubleshoot heartbeat issues:

1. Ensure the Integration Agent has started, and issue a `display-settings` command using IAdmin.
2. Verify that the xMatters web server URL appears in the list of server URLs.
   - If the URL does not appear in the list, then this indicates a problem in the IAConfig.xml file (e.g., the xMatters web server's URL is malformed or not specified).
3. In the Integration Agent's log configuration file, set the heartbeat category to DEBUG; for example:

```
<logger name="com.alarmpoint.integrationagent.heartbeat">
  <level value="DEBUG"/>
</logger>
```

4. Restart the Integration Agent and wait for an attempt to send a heartbeat to the xMatters web server that is rejecting the heartbeats.
5. Consult the Integration Agent's log file and locate the ERROR entry associated with the heartbeat attempt. The ERROR message will either indicate a connectivity failure or provide one of the following reasons for the heartbeat's rejection:
   - **UNKNOWN_DOMAIN**: indicates that one of the Integration Service Event Domains has not been configured on the xMatters web server
   - **UNKNOWN_SERVICE**: indicates that one of the Integration Service names has not been configured within the Event Domain on the xMatters web server
   - **REGISTRATION_ACL_FAILED**: indicates that this Integration Agent's ID has not been configured on the xMatters web server
   - **UNKNOWN_APPLICATION_ERROR**: indicates that an unexpected error occurred
   - **SERVICE_DENIED**: Depending on the error message, the web services user account in xMatters does not have the "Receive APXML" and "Send APXML" permissions (error message contains reference to "ReceiveAPXML") or does not have the "Register Integration Agent" permission (error message references a rejected heartbeat/registration).

Integration Service Configuration Issues

An Integration Service with a runtime state of ERROR has a problem with its configuration file. The most likely cause is a syntax error in the Integration Service JavaScript.

To determine the Integration Service configuration issue:

1. Review the Integration Agent log file (`AlarmPoint.txt`).
2. Locate entries pertaining to parsing the Integration Agent configuration.
3. Within these entries, locate log entries that refer to parsing the Integration Service configuration file causing the issue. One of these entries will contain an error and stack trace identifying the problem.

Example

The following log excerpt shows the context in which Integration Service configuration errors appear within the Integration Agent log file (the excerpt has been edited for brevity).

The first three log entries show that the Integration Service configuration is being parsed. The final entry shows that the Integration Service's JavaScript contains a syntax error.

```
2007-11-06 17:28:59,567 [WrapperSimpleAppMain] INFO - Starting to
        initialize Integration Agent using IA Config file
        C:\sandbox\INTEGR~1\distr\INSTAL~1\conf\IAConfig.xml.
2007-11-06 17:28:59,598 [WrapperSimpleAppMain] INFO - Starting to
        parse the Integration Service Config files in directory ../
        integrationservices.
2007-11-06 17:28:59,598 [WrapperSimpleAppMain] INFO - Parsing
        Integration Service Config file netcool/netcool-admin.xml.
2007-11-06 17:28:59,614 [WrapperSimpleAppMain] ERROR - The script for
        Integration Service (default,test_service_1) could not be created
        due to an exception.
                ScriptCreationException: The script for Integration Service
                (default,test_service_1) could not be created due to an
                exception.
                        Caused by: org.mozilla.javascript.EvaluatorException:
                        missing } after function body
                        (C:\sandbox\integrationagent\distr\installation\
                                integrationservices\netcool\netcool-admin.js#71)
```

Queue Analysis

The Integration Agent uses JMS queues to store both inbound and outbound APXML messages. (Inbound messages are those that need to be processed by an integration service, and outbound messages are those that need to be sent to xMatters.) Each integration service has its own set of queues; for example, the `sample` integration in the `default` domain has the following queues:

- inbound.default.sample.normal
- inbound.default.sample.high
- outbound.default.sample.normal
- outbound.default.sample.high

The first component of a queue name indicates its type (inbound or outbound); the second component is the integration service's domain; the third component is the integration service's name; and, the fourth component is the priority (normal or high).

Queue monitoring

You can monitor the queues in two ways:

- analyzing the log files
- via a JMX connection

Log file analysis

When you enable queue monitoring messages in the Integration Agent log files, the Integration Agent will write messages for all queues to the log several times per minute, similar to the following:

```
inbound.ping.ping.normal toPageIn: 0, Inflight: 0, pagedInMessages.size 0, enqueueCount: 2,
dequeueCount: 2
```

In the above example, the "ping" integration service's normal-priority inbound queue has received two messages and has successfully sent both of them to the integration service for processing. (The information categories contained in the message are documented by Apache's ActiveMQ project; the important values for queue analysis are enqueueCount and dequeueCount.)

If a specific queue's enqueueCount is significantly larger than its dequeueCount, and the difference remains or grows in subsequent messages, there may be a problem with the integration that is delaying notifications. Possible causes include:

- A processing step is taking more time than it should. Typically, this will be a task that involves external communication, such as a request to an external web service.
- A processing step is failing and is causing the integration service to retry processing the injected message. Check the Integration Agent log file for error messages.
- Processing of each injected message is taking a reasonable amount of time, but the volume of messages is very high. Consider increasing the number of threads allocated to the integration service, if it is configured for multi-threaded operation, or make it multi-threaded if it is not. Information on how to make an integration service multi-threaded is available on the xMatters Community site at support.xmatters.com.

To enable the queue monitoring messages:

1. Open the `<IAHOME>\conf\log4j.xml` file in a text editor.
2. Locate the following line:

```
<!-- Health Monitor --->
```

3. Replace the line with the following code:

```
<!-- Detailed ActiveMQ Logging  -->
<logger name="org.apache.activemq.broker">
<level value="DEBUG"/>
</logger>
<!-- Health Monitor -->
```

4. Save and close the file (you do not need to restart the Integration Agent).

JMX connections

To monitor queues via JMX connection, you will need to configure the Integration Agent to allow JMX connections, and then use a tool such as jVisualVM (a component of the free Oracle Java Development Kit) to monitor the queues. In addition to the information mentioned above, this will provide information for each queue, such as the average and maximum queue time, and aggregate information for all queues and system statistics such as memory usage. This capability also requires that the mbeans extension be installed in jVisualVM.

JMX support is disabled by default. To configure the Integration Agent to allow JMX connections, contact xMatters support at support.xmatters.com..

The Integration Agent times out during startup

In some cases, when Integration Agent failover occurs (or after an Integration Agent stops functioning properly), resources are held in a 'locked' state by database sessions that are left open despite termination of the server process.

If the Integration Agent subsequently times out during startup, you should use the following steps to validate and kill expired sessions on resources that can prevent normal startup. Select the instruction set that corresponds to your database type.

Steps for SQL Server 2005/2008

1. Using a SQL Server client, log in as a privileged user, which is typically the sa user, but can be any user with the following:

- Ability to query: sys.dm_tran_locks, sys.partitions, sys.sysprocesses
- Ability to execute: KILL

2. Target the database used for the Integration Agent (as configured in DB shared Queue) by executing the following query:

```
USE <Database for IA>;
```

3. Execute the following query:

```
SELECT
    request_session_id, hostname, loginame, login_time,
    object_name(P.object_id) as TableName
FROM
    sys.dm_tran_locks L
    join sys.partitions P on L.resource_associated_entity_id = p.hobt_id
    join sys.sysprocesses SP on SP.spid = L.request_session_id
WHERE object_name(P.object_id) = 'ACTIVEMQ_LOCK'
GROUP BY
    request_session_id, hostname, loginame, login_time, object_name(P.object_id);
```

4. For each request_session_id, execute the following query (replace <request_session_id> with the result from step 3):

```
KILL <request_session_id>;
```

Steps for Oracle 10g/11g

1. Using an Oracle client, log in as a privileged user, which is typically the system user, but can be any user with the following:
   - Ability to query: v$lock, v$sessions, dba_objects
   - Ability to execute: ALTER SYSTEM KILL SESSION '<sid>,<serial#>' IMMEDIATE;
2. Execute the following query:

```
SELECT s.sid, s.serial#, machine, username, s.logon_time, object_name TableName
   FROM v$lock l JOIN dba_objects o ON l.id1 = o.object_id
                                JOIN v$session s ON l.sid = s.sid
 WHERE o.object_name = 'ACTIVEMQ_LOCK';
```

3. For each row identified, execute the following query:

```
ALTER SYSTEM KILL SESSION '<sid>,<serial#>' IMMEDIATE;
```

Chapter 5: Service API

Integration Services that are implemented with JavaScript (i.e., non-legacy services) can use the Service API to access the following functionality that is not included in the standard JavaScript environment:

- Writing service-specific log messages as part of the Integration Agent's logging system.
- Sending Integration Service Requests to other Integration Agents with automatic URL binding.
- Sending APXML messages to xMatters.

The Service API is exposed to an Integration Service's JavaScript via a global variable named `ServiceAPI`.

Note: *The classes and methods that comprise the Service API are described in detail in the JavaDoc located at* `<IAHOME>/docs/service_api/javadoc`. *The following sections are intended to provide an overview of the Service API's major functionality. Integrators who plan to use the Service API should consult the JavaDoc for additional details.*

Configuration

The following function allows integrators to retrieve the integration service name, event domain name, and Integration Agent ID from within the integration service javascript:

```
ServiceAPI.getConfiguration()
```

For example:

```
var config = ServiceAPI.getConfiguration();
config.getName(); // Integration service name. i.e.. sample-relevance-engine
config.getDomain(); // Event domain name. ie.. applications
config.getAgentId(); // IA ID e.g. localhost.localdomain/127.0.0.1:8081
```

Note: *Agent IDs can be configured and stored in the* `<IAHOME>\conf\IAConfig.xml` *file; for more information, see "Integration Agent Configuration File" on page 18.*

Logging

Each Integration Service has its own Log4j logging category (for details, see "Integration Agent Log" on page 51), which is of the following form:

```
com.alarmpoint.integrationagent.services.<domain>.<name>
```

For example, the Sample integration that is part of the default domain has the following category:

```
com.alarmpoint.integrationagent.services.default.sample
```

Activity that is specific to an Integration Service (e.g., processing a Integration Service Request) is logged using the service's category, and the specifics of how and where to log the message are controlled via the Integration Agent's Log4j configuration file located at `<IAHOME>/conf/log4j.xml` (for details about how to customize this file, see "Logging Configuration File" on page 51).

The Service API's `getLogger()` method returns an `org.apache.log4j` logger that uses the service's category. When the JavaScript implementation of an Integration Services uses this logger, the logging messages will be filtered and formatted as if the Integration Agent had produced them.

The following example demonstrates how an Integration Service can use the Service API to log anticipated exceptions as warnings, rather than the normal behavior of allowing such exceptions to propagate out of the JavaScript where they would be caught by the Integration Agent and logged as errors:

```
function apia_example(params) {
        var user = params.get("user");
        var pwd = params.get("password");

        try {
accessManagementSystem(user, pwd);
return "Access granted.";
}
catch (ex) {
        ServiceAPI.getLogger().warn("An unsuccessful attempt was made to
access " + "the Management System using the account: "+user);
return "Access denied.";
        }
}
```

Note: *In the latter example, the* `accessManagementSystem()` *method checks that the submitted user/pwd specifies a valid Management System account and throws an exception if the account is invalid.*

Agent-to-Agent Requests

Each Integration Service exposes a web service method named `IntegrationServiceRequest` (ISR) that allows clients to execute specific JavaScript methods and receive the results. The primary client for ISRs is xMatters mobile access, but an Integration Service can also make an ISR to another Integration Service (either on the same IA or on a remote IA) via the Service API.

Note: *You can make or forward a lighter-weight request to a service on the same Integration Agent; for details, see "Service-to-Service Requests" on page 64.*

To make an ISR, an Integration Service must know the following information:

- The targeted Integration Service's domain and name (e.g., `default|sample`).
- The ISR actions provided by the targeted Integration Service (e.g., `ping`).
- The parameters that the targeted action requires (e.g., `device`).
- The response type returned by the targeted action (e.g., a `java.lang.String`).

The following example demonstrates how an Integration Service can use the Service API to request that another Integration Service ping a remote server:

```
function apia_example(params) {
        var isr = ServiceAPI.createIntegrationServiceRequest();
        isr.setDomain("default");
        isr.setName("sample");
        isr.setAction("ping");
        isr.setToken("device", "www.myserver.com");

        var result = isr.send();

        ServiceAPI.getLogger().info("Ping response: "+result);
}
```

In the latter example, `ServiceAPI.createIntegrationServiceRequest()` returns an object of type `com.alarmpoint.integrationagent.script.api.IntegrationServiceRequest`
, which is a container for specifying the information required to make an ISR. The actual ISR is initiated by `isr.send()` according to the following rules:

- If the ISR targets a specific Integration Agent (via `isr.setAgentId()`):
  - If the targeted Integration Agent provides the requested Integration Service in an active state, then the ISR is sent to this Integration Service.
  - If the targeted Integration Agent does not provide the requested Integration Service in an active state, then the ISR fails

- If the ISR does not target a specific Integration Agent:
  - If the requested Integration Service is active in the local Integration Agent, then the ISR is sent to this local Integration Service
  - If the requested Integration Service is not active in the local Integration Agent, but is active in at least one remote Integration Agent, then the ISR is sent to one of these randomly-selected remote Integration Agents.
  - If the requested Integration Service is not active in any Integration Agent, then the ISR fails.

This means that when making an ISR, the caller can control whether the Integration Service request should be sent to a specific Integration Agent, or whether it should be sent to any Integration Agent providing the Integration Service. The Service API automatically binds the ISR to a specific web service gateway URL, formulates the SOAP request (including any necessary access password), and deserializes the SOAP response to a Java object.

Service-to-Service Requests

In some cases when authoring an integration, it may be desirable to encapsulate integration points as separate Integration Services. To have these Integration Services communicate, you can configure Service-to-Service requests, as described in this section.

Requests can be made to another service within the same Integration Agent without having to create an IntegrationServiceRequest. The Service-to-Service request sends an APXML message into a targeted service's inbound queue.

To make a Service-to-Service request, an Integration Service must know the following information:

- The targeted Integration Service's domain and name (e.g., default|sample)
- The APXML that is expected by the targeted Integration Service

The following example demonstrates how an Integration Service can use the Service API to request that another Integration Service on the same Integration Agent ping a server:

```
function apia_example(apxml) {
      ServiceAPI.sendAPXML(apxml, "ping", "ping");
}
```

Note: *This example assumes that the APXML passed to* `apia_example` *already includes the tokens that are required by the ping Integration Service (e.g.,* `request_text` *and* `device`*).*

The sendAPXML method does not wait for the targeted service to process the APXML before returning.

Sending APXML to xMatters

Normally, an Integration Service receives an APXML message from `APClient.bin`, pre-processes it, and then forwards it to xMatters for further processing. With the Service API, an Integration Service can create its own APXML messages and submit them to xMatters. For example, in response to a single `APClient.bin` submission, an Integration Service can create multiple APXML messages to inject events across several Event Domains. An Integration Service can also create APXML messages in response to Integration Service Requests from xMatters.

The following example demonstrates how an Integration Service can use the Service API to inject an event in the messaging domain:

```
function apia_example(params) {
        var apxml = ServiceAPI.createAPXML();
        apxml.setMethod("Add");
        apxml.setSubclass("Event");
        apxml.setToken("agent_client_id", "messaging");
        apxml.setToken("apia_priority", "high");
        apxml.setToken("recipients", "bsmith");
        apxml.setToken("situation", "Server down");
        apxml.setToken("device", "192.168.0.99");
        apxml.setToken("incident_id", "TICKET-0100-3020");
        apxml.setToken("contact_type", "voice");

        ServiceAPI.sendAPXML(apxml);
}
```

In this example, `ServiceAPI.createAPXML()` returns an object of type `com.alarmpoint.integrationagent.apxml.APXMLMessage`, which is a container for specifying the contents of an APXML message. The method and subclass are set to indicate that the message's function is to create an event in xMatters. The `apia_priority` token is set to indicate that the message should be placed in the Integration Service's high priority outbound queue. The remaining tokens specify the values that are used by the messaging domain's PROCESS script.

Note: *This example shows the APXML message's* `agent_client_id` *token set to identify the Event Domain in which the event will be created (i.e., messaging). However, be aware that the current implementation of sendAPXML (apxml) always overwrites* `agent_client_id` *with the ID of the Integration Service sending the APXML (as documented in the Service API's JavaDoc). This means, for example, that an Integration Service can only inject events within its own Event Domain.*

Chapter 6: APXML Reference

Introduction to the APXML Reference

xMatters XML (APXML) is an XML format used to exchange information between the Integration Agent and xMatters. It is primarily a container for typed key-value pairs (also called tokens).

Example

Consider the following APXML message, which is sent by the Integration Agent to xMatters to trigger a `ping` event:

```
<?xml version="1.0"?>
<transaction id="99">
  <header>
    <method>Add</method>
    <subclass>Event</subclass>
  </header>
  <data>
    <agent_application_id>Agent_001</agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
    <recipients type="string">bsmith,auser</recipients>
    <recipients-list-item-delimiter>,</recipients-list-item-delimiter>
    <situation>Server down.</situation>
    <device>192.168.1.10</device>
  </data>
</transaction>
```

All APXML messages begin with a `transaction` element that has a numeric `id` attribute with a value in the range 1-2147483647. The `id` attribute does not need to be unique, and has no effect on xMatters; it is a mechanism that enables an application to correlate a set of APXML messages.

The `header` element defines the purpose of the message (e.g., Add Event, Delete Incident, etc.). The `method` element is required and defines the broad purpose of the message (e.g., Add, Delete, etc.). Some methods may also require a `subclass` element to provide further specificities for the method.

The remaining portion of the APXML message, within the `data` element, defines the parameters for the method and any additional information that the receiver of the message requires.

APXML Methods

The allowable methods/subclasses for APXML messages sent to an integration depend on what the receiver recognizes. APXML messages generated via APClient.bin and sent to an integration can specify only methods/subclasses that are recognized by that integration. If other methods/subclasses are specified, an error may result, or it may be transformed to a default method/subclass.

APXML messages that are sent to xMatters are normally generated by an integration, either in response to an APCient.bin submission, or a request from xMatters. The following table summarizes the methods/subclasses that xMatters recognizes:

Supported input methods

Method	Subclass	Description	Required Tokens
Add	Action	Creates an xMatters event	
Add	Event	Creates an xMatters event	
Del	Action	Deletes an xMatters event	

Method	Subclass	Description	Required Tokens
Del	Event	Deletes an xMatters event	
Del	Incident	Deletes an incident	
Response	n/a	Submits a response to an ExternalRequest	

Note: *xMatters recognizes these methods/subclasses regardless of case; however, methods/subclasses defined by integrations may be case sensitive.*

For every APClient.bin submission, the Integration Agent responds with an APXML message. The following table summarizes the methods/subclasses that are used for Integration Agent responses:

Integration agent response methods

Method	Subclass	Description	Relevant Tokens
Agent	OK	Returned by the Integration Agent when a message is accepted and queued for processing	
Agent	ERROR	Returned by the Integration Agent when a message is rejected	

xMatters responds to every submitted APXML message that is sent to it by the Integration Agent with an APXML message. These response messages are sent to the applicable integration service for Response Action Scripting. The following table summarizes the methods/subclasses that are used for xMatters responses:

xMatters response methods

Method	Subclass	Description	Relevant Tokens
OK	n/a	Returned by xMatters when a message is accepted and queued for processing	
ERROR	n/a	Returned by xMatters when a message is rejected	

xMatters may asynchronously (i.e., without being triggered by an Integration Agent submission) send APXML messages to the Integration Agent. These messages are triggered by an xMatters Action Script and are requests to an integration service to perform some action.The following table summarizes the methods/subclasses that are used for xMatters requests:

xMatters request methods

Method	Subclass	Description	Relevant Tokens
Send	n/a	Produced by the ExternalServiceMessage send method	
Request	n/a	Produced by the ExternalServiceRequest2 send method	

APXML Tokens

An APXML token is a key-value pair represented by an XML element (the key) with textual content (the value). The key is case-insensitive and since it is an XML element, it cannot contain any spaces or special characters. The value is case-sensitive, and within the Integration Agent is always treated as a character string. However, when an APXML message is processed by xMatters, the value may be interpreted as a non-string type (e.g., integer, long, floating-point, etc.).

The type that xMatters should use for the value can be specified using the token's `optional` type attribute. If the token does not have the `type` attribute, then xMatters auto-types the value; that is, xMatters automatically determines the value's type based on its appearance. In some cases, this auto-typing may be incorrect. For example, consider the following social security number token:

```
<ssn>111 222 333</ssn>
```

Since no explicit type is provided, xMatters will auto-type the value as an integer rather than a string. To prevent this from occurring, the token can be rewritten as:

```
<ssn type="string">111 222 333</ssn>
```

Additionally, a token can be explicitly typed as a number using `type="numeric"` attribute.

The following table summarizes reserved APXML tokens that should not be used for other purposes:

Supported Input Methods Message for xMatters

Name	Value	Description
agent_application_id	non-empty string	ID of the Integration Agent that generated the message
agent_client_id	non-empty string	ID of the integration service
apia_password	non-empty string	Password or constant identifying the path to the encrypted password file containing the authentication password of the event submitter. (Used in conjunction with the password-authentication parameter in the IAConfig.xml file.)
apia_priority	normal | high	Priority of the message
apia_process_group	<blank> | non-empty string | apia_unique_group	Controls concurrent processing of the messsage
apia_source	apia_apclient: <ip> apia_ alarmpoint: <url>	Identifies the source of the message
recipients	list of non-empty strings	Identifies the event recipients
event_id	non-empty string	Identifies the event
incident_id	non-empty string	Identifies the incident (maximum length is 250 characters)

Note: *Specific integrations may reserve other tokens. Additionally, the actual tokens that are required in an APXML message depend on the message's method/subclass, as discussed in the "APXML Methods" section above.*

xMatters Usage of APXML Tokens

Some APXML tokens represent lists. When xMatters receives an APXML message and converts the APXML tokens into event tokens, xMatters needs to know the delimiting character. This delimiting character is specified by an additional APXML token whose key is the list token's key with `-list-item-delimiter` appended. The value of this additional token is the delimiting character of the associated list. For example, the following APXML tokens are used to define a comma-separated list:

```
<dates>3 June 2008, 7 December 2007, 12 October 1984</dates>
<dates-list-item-delimiter>,</dates-list-item-delimiter>
```

If a list token does not have an explicit delimiter, then its value will be treated as a single string (i.e., xMatters will not interpret it as a list). The exception is the `recipients` token, which will be interpreted as a comma-separated list if no explicit delimiter is defined.

Chapter 7: Integration Agent Client Requests

About this Chapter

This chapter describes how Integration Agent clients (e.g., xMatters mobile access) can make Integration Service requests and receive responses. These requests and responses are implemented as SOAP messages that are sent between the Integration Agent client (e.g., the xMatters mobile access) and the Web Service Gateway exposed by each Integration Service.

URLs

Each Integration Service exposes its own Web Service Gateway to which an Integration Agent Client (e.g., xMatters mobile access) may submit Integration Service Requests. The Web Service Gateway URLs use either the HTTP or HTTPS protocol, depending on the `ssl` attribute of the `service-gateway` element in the Integration Agent configuration file (for details, see "Integration Agent Configuration File" on page 18). Similarly, the host and port for the URLs comes from the `host` and `port` attributes of the `service-gateway` element.

Example

Assume that the Integration Agent configuration file contains the following `service-gateway` element:

```
<service-gateway ssl="true" host="www.company.com" port="8081"/>
```

In this case, all Integration Service URLs would begin with:

```
https://www.company.com:8081
```

The actual Integration Service URL would be formed by adding the Integration Service's domain and name to the end of the base URL. If the Integration Service's domain is "default" and its name is "sample", then the URL would be:

```
https://www.company.com:8081/default_sample
```

Security

To restrict which Integration Agent clients are allowed to submit requests to the Integration Service URLs, the Integration Agent employs an access control list (ACL) via the `ip-authentication` and `password-authentication` elements of the Integration Agent configuration file (for details, see "Integration Agent Configuration File" on page 18). If `ip-authentication` is enabled, then before an Integration Service processes a request, it check whether the client's IP address is authorized.

If `password-authentication` is enabled, then before an Integration Service processes a request, it checks whether the SOAP body contains an `apia_password` element whose value matches the Integration Agent's access password. If either of these authorizations fail, the request is denied and the attempted access is logged to the Integration Agent log (see "Integration Agent Log" on page 51).

Note: *If you are using the Integration Agent with the xMatters mobile access, review the Access Control List information provided in the* xMatters mobile access Guide.

Additionally, that guide includes information regarding the Integration Agent Access Control page available to Super Admin (root) accounts.

Requests

Each Integration Service exposes a Web Service Gateway with a single method named `IntegrationServiceRequest`. This method takes three parameters; the first parameter identifies the name of one of the JavaScript methods implemented by the Integration Service; the second parameter identifies the optional Integration Agent access password; the third parameter is a set of key-value pairs specifying the JavaScript method's parameters.

The order of the JavaScript parameters is not important, as each is identified by its key. Each key is a string and each value is an arbitrary Java type serialized by XStream (an open-source XML and Java serialization/deserialization project). It is important that any custom objects (i.e., not those in the standard Java runtime) are available on the Integration Agent's classpath so that they can be deserialized when the request is received by the Integration Agent.

Example Request

This example shows an Integration Service request made by the sample integration when an xMatters mobile access user updates a ticket. The request specifies that the `updateTicket` JavaScript method should be called with the following parameters:

updateTicket Parameters

Key	Value
`description`	`Description 1`
`status`	`Open`
`id`	`Ticket 1`

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <ap:IntegrationServiceRequest xmlns:ap="http://www.alarmpoint.com/webservices/schema"
xmlns="http://www.alarmpoint.com/webservices/schema">
      <ap:actionName>updateTicket</ap:actionName>
      <ap:apia_password>123456</ap:apia_password>
      <ap:params>
        <entry>
          <string>description</string>
          <string>Description 1</string>
        </entry>
        <entry>
          <string>status</string>
          <string>Open</string>
        </entry>
        <entry>
          <string>id</string>
          <string>Ticket 1</string>
        </entry>
      </ap:params>
    </ap:IntegrationServiceRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

Note: *The SOAP request message does not identify the Integration Service; the URL to which the request is sent identifies the Integration Service.*

Responses

For the final step in processing an Integration Service request, the Integration Agent calls the associated Integration Service JavaScript method. This method can return any JavaScript primitive type or object. The Integration Agent transforms the method's return value into the corresponding Java primitive type or object, which is then serialized by XStream and returned as part of the SOAP response body.

It is the client's responsibility to deserialize the Integration Service response. For example, in the mobile access component, the mobile access component Client automatically deserializes the response and returns the Java primitive type or object to the JSP.

It is important that any custom objects (i.e., not those in the standard Java runtime) that are serialized by the Integration Agent be available on the client's classpath so that they can be deserialized when the client receives the response.

Example Response

This example shows an Integration Service response that corresponds to the previous request example. The response from `updateTicket` is simple, specifying a single Boolean value indicating the success or failure of the operation:

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <IntegrationServiceRequestResponse xmlns="http://www.alarmpoint.com/webservices/schema">
      <boolean>true</boolean>
    </IntegrationServiceRequestResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

A more detailed example is the response from `getTickets`, which specifies a list of structured data for the available tickets:

```
<soapenv:Envelope
        xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <IntegrationServiceRequestResponse xmlns="http://www.alarmpoint.com/webservices/schema">
      <list>
        <map>
          <entry>
            <string>description</string>
            <string>Description 1</string>
          </entry>
          <entry>
            <string>status</string>
            <string>Open</string>
          </entry>
          <entry>
            <string>id</string>
            <string>Ticket 1</string>
          </entry>
        </map>
        <map>
          <entry>
            <string>description</string>
            <string>Description 2</string>
          </entry>
          <entry>
            <string>status</string>
            <string>Closed</string>
          </entry>
          <entry>
            <string>id</string>
            <string>Ticket 2</string>
          </entry>
        </map>
      </list>
    </IntegrationServiceRequestResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Chapter 8: Integration Agent APIs

This chapter identifies and explains the different interfaces available with the Integration Agent.

Customizing utility scripts

As of version 5.1.5, the Integration Agent includes a set of scripts that provide utility functions for REST-based integrations. The scripts are installed to the `<IAHOME>\lib\integratonservices\javascript` folder, and include the following files:

- `event.js`
- `apxml.js`
- `xmio.js`
- `xmutil.js`

These scripts are shared, and can be loaded by any integration service. This means that any modifications to the scripts will affect all integration services using them. If you want to modify the utility scripts, you can avoid affecting all integration services by making a local copy and loading it into the integration scripts.

NOTE: Each successive load statement will override any previous statements and loading `event.js` will also load all of the other utility scripts. Therefore, any load statement in the integration script referring to a specific utility script must appear *after* any calls to `event.js`.

As an example, the following instructions describe how to load a local copy of the `xmio.js` utility script into the sample relevance engine integration service.

To use a customized utility script:

1. Copy `<IAHOME\lib\integrationservices\javascript\xmio.js` to `<IAHOME>\integrationservices\applications\sample-relevance-engine`.
2. Open the `<IAHOME>\integrationservices\applications\sample-relevance-engine\sample-relevance-engine.js` file in a text editor.
3. Locate the LAST occurrence of a load statement referring to either `event.js` or `xmio.js`.
   - For example:

```
load("lib/integrationservices/javascript/event.js");
OR
load("lib/integrationservices/javascript/xmio.js");
```

4. Append the following new load statement:

```
load"(integrationservices/applications/sample-relevance-engine/xmio.js");
```

5. Save and close the file.
6. Restart the Integration Agent.
7. Test that the functionality has not changed.
8. Make your changes to the `xmio.js` file.
9. Restart the Integration Agent, and then test that your changes work as intended.

Creating communication plan events

You can use the Integration Agent to target communication plan forms and create events via the xMatters REST API. This feature offers the following benefits:

- If your management system is not capable of accessing the REST API directly, you can use the integration agent to integrate with specific communication plan forms.
- You can create an integration to take advantage of the extra features available in xMatters On-Demand, including communication plan forms and scenarios, the xMatters mobile applications, and push notifications.
- Configuring callbacks with the Integration Agent allows you to receive callbacks from behind a firewall.
- You can use the communication plan designer instead of scripting to create notification messages.
- Creating communication plan events with the Integration Agent allows you to use Integration Agent features and capabilities such as retries and queuing.

Overview

The following sections describe the process by explaining how to configure the sample communication plan and its related components included with the Integration Agent. You can find the samples in the following location:

```
<IAHOME>/integrationservices/applications/sample-relevance-engine
```

The configuration process requires the following (each step identifies the related sample file):

1. Import the communication plan (`SampleRelevanceEngine.zip`) and form (Confirmation Message) in xMatters, and configure an integration service on the applications event domain.
2. Set the parameters in the `configuration.js` file to point to the communication plan you want to target (`configuration.js`).
3. Set the connection information in the integration service file (`sample-relevance-engine.xml`), and then do one of the following:
   - If you are using the HTTP interface to create events, identify the JavaScript file used to transform the data mappings and other information into the post JSON body. For more information, see "HTTP interface configuration" on page 84.
   - If you are using a command line call (APClient.bin) to create events, configure the map data entries to specify the properties you want to send. For more information, see "Mapped input parameters" on page 85.
4. (Optional) Modify the Javascript file to customize the parameter mappings, callbacks, and incident process (`sample-relevance-engine.js`).

Prerequisites

Before using the samples, you will need to create or configure the following components and options:

REST API User

Because the Integration Agent to communication plan integration uses the REST API, you will need to have a user in your xMatters deployment that is able to authenticate REST API calls. (They do not need to be able to access the web user interface or receive notifications.) This user is referred to throughout the following sections as the REST API user.

You will also need to store this user's password in an encrypted password file within the Integration Agent directory. For instructions on how to create this file, see "The IAPassword Utility" on page 44.

Recipient

Make sure that you have configured at least one user in xMatters with an active device so you can send a test notification after you have configured the integration.

Indirect mode

This feature requires that the Integration Agent be running in indirect mode, which will allow xMatters to communicate with an Integration Agent behind a firewall. You can configure this option in the `IAConfig.xml` file, as described in "Integration Agent Configuration File" on page 18.

Install the sample files

Before you can use the sample files, you must import and configure the communication plan, and add the integration service to the 'applications' event domain.

Note that while the following instructions are tailored for the samples, they can also serve as an overall guideline when designing an integration between a management system and a communication plan form using the Integration Agent.

Import the communication plan

To begin, import the sample communication plan file (`SampleRelevanceEngine.zip`) into your xMatters On-Demand deployment. For more information, including a description of the import process, consult the xMatters help at `http://help.xmatters.com/ondemand`.

Once you have finished importing, complete the following steps to enable and prepare the communication plan to receive requests.

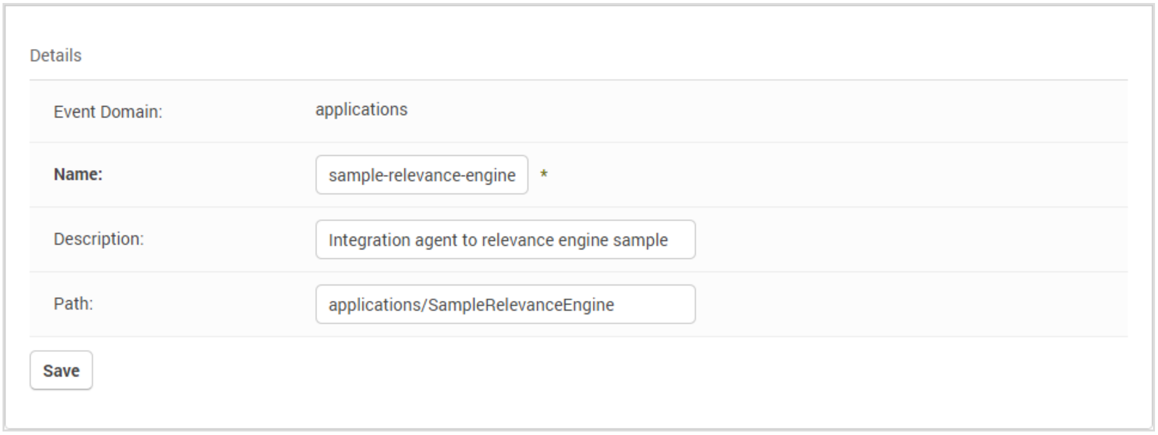
To prepare the sample communication plan:

1. In the list of communication plans, click the **Engine Disabled** button beside the sample communication plan to enable it.
2. Click the **Edit** drop-down list and select **Forms**.
3. Click the **Not Deployed** drop-down list and select the **Create Event Web Service**.
   - The list label changes to "Web Service Only".
4. Click the **Web Service Only** drop-down list and select **Access Web Service URL**.
   - Note that the Access Web Service URL option appears twice in the list; make sure you select the first option, immediately beneath the Create Event Web Service.
5. Copy the contents of the URL field at the top of the dialog box to a text file; you will need this URL to configure the connection with the Integration Agent.
6. Click **Close**.
7. Click the **Web Service Only** drop-down list and select **Permissions**.
8. In the Form Permissions dialog box, add your REST API user to the list of initiators, and then click **Save Changes**.

The sample form is now ready to receive events from the Integration Agent. Note that you can also configure callbacks for this sample form (see "Configure outbound integration webhooks" on page 83 for more information), but you must configure the integration service first as described below.

Configure integration service

When you have finished configuring the communication plan, you will need to create a new integration service named "sample-relevance-engine" in the applications event domain on your deployment, using the xMatters web user interface. The following image illustrates the settings you should specify for the integration service, assuming you have not moved the samples from their default location (the Path field can be left blank):



Details

Event Domain:	applications
Name:	sample-relevance-engine *
Description:	Integration agent to relevance engine sample
Path:	applications/SampleRelevanceEngine

Configure the integration file

Most xMatters integrations include a `configuration.js` file that provides the connection information the Integration Agent requires to send requests into xMatters, and the sample communication plan integration files include a pre-configured version.

To see or modify the sample communication plan integration configuration, open the `integrationservices/applications/sample-relevance-engine/configuration.js` file in a text editor, and review the following settings:

Setting	Description
WEB_SERVICE_URL	The web service URL specific to the targeted communication plan form that you retrieved when importing the sample.
CALLBACKS	A list of all the outbound integration webhooks (a.k.a callbacks) you want the integration service to return. For more information about this setting, see "Configure outbound integration webhooks", below.
INITIATOR	The REST API user to use when authenticating Integration Agent requests with the REST API.
PASSWORD	The location of the encrypted password file containing the password of the REST API user (the initiator).
DEDUPLICATION_FILTER_NAME	The name of the deduplication filter to use for events in this integration service. For more information, see "Portable Filtering and Suppression Module" on page 46.

When you are finished, save and close the file.

Configure outbound integration webhooks

Outbound integration webhooks (formerly and sometimes currently known as "callbacks") allow web applications and integrations to extract information from an xMatters event, and take action based on the extracted properties. You can configure webhooks for event status changes, notification deliveries, user responses, and event comments (annotations).

Note: *For more information about using outbound integration webhooks, see the xMatters On-Demand help at http://help.xmatters.com.*

Once you have identified the types of webhooks you want to receive, you need to configure the integration file (`configuration.js`) to request them.

The web user interface also contains outbound integration configuration for communication plan forms. If you configure the webhooks within the integration file, the Integration Agent will automatically register using the Integration Agent ID.

To configure the webhooks in the integration file:

1. Open `configuration.js` in a text editor.
2. Specify the types of outbound integrations you want in the CALLBACKS parameter.
   - The following setting illustrates the syntax for the parameter, and will request all available webhook types:

```
CALLBACKS = ["status", "deliveryStatus", "response", "annotation"];
```

3. Save and close the file.

You will need to add handling to the Javascript file (`sample-relevance-engine.js`) that will process the webhooks and parse the information. The default sample is already configured to process three types of webhook (the "annotation" type is relatively new) and record information about the event to the Integration Agent log file. For more information, see "apia_callback(msg)" on page 86.

Configure integration service file

Targeting a communication plan form with the integration agent uses an XML-based integration service file, as described in "Integration Service Configuration File" on page 24. Use this file to set the following parameters:

- The name of the event domain; when targeting a communication plan, this should always be "applications".
- The name of the integration service configured for the targeted communication plan, as described in "Install the sample files" on page 82.
- The name and location of the integration file containing the data transformation information, as described in "Customize your integration" on page 86.
- The mapped-input parameters specifying the values you want to inject in to xMatters; these are only required for command line (APClient.bin) integrations.

HTTP interface configuration

When targeting a communication plan with the Integration Agent's HTTP interface, the apia_http function in a JavaScript file transforms the incoming data into a correctly-formatted REST API call (i.e., a JSON object) that generates notifications using the targeted communication plan.

Example

To configure the sample communication plan, open the `sample-relevance-engine.xml` file in a text editor.

The <script> parameter should point to the Javascript file used to transform the data mappings and other information into the post JSON body; in this case, the `sample-relevance-engine.js` file.

To receive callbacks for this integration, the <domain> value must be set to "applications", and the value of the <name> parameter must be set to the name of the integration service. The mapped-input section contains default parameters for injecting events via the command line (APClient.bin). You can remove or comment out this section if you are using the HTTP interface.

Mapped input parameters

When targeting a communication plan with the command line interface (APClient.bin), you use mapped-input parameters to specify the values you want to inject into xMatters. When you pass in the values in the command line, the Integration Agent transforms them into a correctly-formatted REST API call (i.e., a JSON object) that generates notifications using the targeted communication plan.

Each parameter maps to a communication plan property or to one of the following top-level message elements:

- **recipients**: Maps to the Recipients section of a form, which identifies the notification targets.
- **xmresponses**: Maps to the responses for the communication plan.
- **xmconferences**: Maps to the Conferences section of a communication plan form.
- **xmpriority**: Sets the priority of the conferences, responses, and notifications. (In the sample communication plan integration configuration files, this has been modified to take a parameter called "priority". For more information about this modification, see "Customize your integration" on page 86.)

Example APClient.bin configuration

To configure the mapped input parameters for the sample communication plan, open the `sample-relevance-engine.xml` file in a text editor.

The first few settings identify the target of the Integration Agent requests. For communication plan targets, the <domain> value must be set to "applications", and the <name> parameter must be set to the name of the integration service. The <script> parameter should point to the Javascript file used to transform the data mappings and other information into the post JSON body; in this case, the `sample-relevance-engine.js` file.

The mapped-input section contains the following default parameters:

```
<mapped-input method="add">
  <parameter>recipients</parameter>
  <parameter>priority</parameter>
  <parameter>building</parameter>
  <parameter>city</parameter>
</mapped-input>
```

You can see from this configuration that the command line input expects to receive four parameters in a specific order: first the targeted recipients, then the priority of the notification (as mentioned above, this has been remapped from 'xmpriority'), and finally a building and a city, which are properties on the sample communication plan form.

If you had other properties on the form, you could add them as parameters within the mapped-input section. You could also add a reserved parameter to further customize the integration. Note that if you are creating an event via the command line, the syntax requires you to specify the values in the same order in which they are listed in the mapped-input section.

Create a sample event via the command line

To create a communication plan event using the command line (APClient.bin), submit a call to your Integration Agent using the following syntax:

```
APClient.bin --map-data <integrationService> <parameters>
```

Note: If the name of the event domain and the name of the integration service are not the same, use the following syntax:

```
APClient.bin --map-data "<eventDomain>|<integrationService>" <parameters>
```

The parameters you include are based on the settings in the mapped-input section in the integration service file. For example, assume that you wanted to create an event based on the Sample Relevance Engine with the following properties:

Property	Value
recipients	bsmith
xmpriority	high
form properties	building: Building B city: Vancouver

Your command line entry would use the following syntax and values. The entry has line breaks only to help with legibility; in an actual entry, all content would be on a single line:

Unix:

```
apclient.bin --map-data 'applications|sample-relevance-engine' '[{"targetName":"bsmith"}]'
  'high' '["Building A"]' 'Vancouver'
```

Windows:

```
apclient.bin --map-data "applications|sample-relevance-engine" "[{\"targetName\":\"bsmith\"}]"
high
  "[\"Building A\"]" Vancouver
```

Note: *For more information about command lines calls to the Integration Agent, see "APClient.bin" on page 95.*

Create an event via the HTTP interface

The sample communication plan integration includes a shell script that uses Curl to inject an XML file with the following properties:

```
<properties>
    <building>"Building A", "Building B"</building>
    <city>Victoria</city>
</properties>
```

To inject the sample event, launch the shell script on a Curl-enabled system.

Note: *For more information about the HTTP interface and calls to the Integration Agent, see "HTTP Interface" on page 90.*

Customize your integration

Each integration service used by the applications event domain requires a Javascript file to ensure that the injected parameters can be properly transformed into a post JSON body for the REST API. The included `sample-relevance-engine.js` file is pre-configured to work with the sample communication plan, but you can also use it as a template when building other integrations as it includes examples of all the necessary components and functions used to transform the data.

The sample file includes examples of new functions that the Integration Agent uses to parse the included parameters into a post JSON body.

apia_callback(msg)

This function works similar to apia_response (as described in "Response Action Scripting" on page 110), but the parameter is a Javascript object that contains the callback information from xMatters.

A callback for status from the example injection might resemble the following:

```
{
  "status":"active",
  "apia_priority":"normal",
  "eventidentifier":"202000",
  "username":"admin",
  "date":"15-01-05 21:29:46.267",
  "agent_application_id":"sample/127.0.1.1:8081",
  "apia_source":"alarmpoint: ",
  "apia_process_group":"apia_default_group",
  "agent_client_id":"applications|sample-relevance-engine",
  "additionalTokens":{
                      "building":"Building A"
                     },
  "incident_id":"INCIDENT_ID-202000",
  "xmatters_callback_type":"status"
}
```

The `sample-relevance-engine.js` contains the following example of how to use the apia_callback function to access the values within the callback and print it to the Integration Agent log when integration service logging is enabled:

```
function apia_callback(msg)
{
  var str = "Received message from xMatters:\n";
  str += "Incident: " + msg.incident_id;
  str += "\nEvent Id: " + msg.eventidentifier;
  str += "\nCallback Type: " + msg.xmatters_callback_type;
  IALOG.info(str);
}
```

Command line functions

The following functions apply to integrations using APClient.bin injection and mapped-input parameters.

For the following sections, assume the following command line entry, targeting the sample communication plan included with the Integration Agent:

```
APClient.bin --map-data 'applications|sample-relevance-engine' '[{"targetName":"admin"}]'
'medium' '["Building A"]' 'Vancouver'
```

If you are submitting the command line entry on a Windows system, you will need to use the double-quote syntax:

```
APClient.bin --map-data "applications|sample-relevance-engine" "[{\"targetName\":\"admin\"}]"
"medium" "[\"Building A\"]" "Vancouver"
```

Linux systems will accept either format.

apia_event(form)

This function works similar to apia_input (as described in "Integration Service Scripts" on page 104), except that where the apia_input function accepts an APXML parameter, the apia_event function takes a Javascript object that can be serialized to the xMatters post JSON body.

The function uses the map data submissions to set the properties on the communication plan, but you can also use the function to set the defaults on the target form.

Note: *Any properties set using the apia_event function will override any values passed in via the command line.*

The function refers to form properties using the following syntax:

```
form.properties.<propertyName>
```

For example, in the included sample form, you would refer to the two properties as follows:

```
form.properties.building
form.properties.city
```

The following table illustrates the required formatting and quoting rules for the different types of properties available on a communication plan form:

Property Type	Example
Boolean	`"true"` or `"\"true\""`
Combo Box	`combo1` or `'combo 2'` or `"combo example 3"`
Hierarchy	`'["hierarchy level 1", "hierarchy level 2", "hierarchy level 3"]'` or `"[\"hierarchy level 1\", \"hierarchy level 2\", \"hierarchy level 3\"]"`
List	`'["list item 1", "list item 2"]'` or `"[\"list item 1\", \"list item 2\"]"`
Number	5
Text	`textExample1` or `'text example 2'` or `"text example 3"`

Note that you can also use the apia_event function to specify default recipients. As with properties, any recipients defined within the Javascript file will override the recipients passed in via the command line. The syntax to set the recipients is as follows:

```
form.properties.recipients = '[{"targetname": "<User ID>"}]'
```

The formatting and quoting rules are also similar, as illustrated in the following examples:

```
'[{"targetname": "admin"}, {"targetname": "bsmith"}]'
```
or, as required on Windows systems:
```
"[{\"targetname\": \"admin\", \"targetname\": \"bsmith\"}]"
```

Example

As an example of the apia_event function, you could add the following code to the `sample-relevance-engine.js` file:

```
function apia_event(form)
{
  // The following would print this to the IA log:
  // Building is Building A
  ServiceAPI.getLogger().info("Building is " + form.properties.building);

  // This would change the city value from Vancouver to Victoria
  form.properties.city = "Victoria";
  return form;
}
```

The above code would log the name of the building that is passed into APClient.bin in the Integration Agent log file, and would set the value of the city property to "Victoria", overriding anything passed in on the command line.

apia_remapped_data()

This function allows you to reconfigure the reserved keyword mappings, and use a different parameter to target a top-level form attribute. You can use this function to remap one or more of the top level attributes in the post JSON body.

Syntax

```
function apia_remapped_data() {
  return {
    "<top-level JSON attribute>" : "<mapped-input parameter name>"
  }
}
```

For an explanation of this syntax, consider the following example of the JSON used to create an event based on the sample communication plan form:

```
{
  "recipients":
    [
      {
        "targetName": "bsmith"
      },
    ],
  "priority": "high",
  "conferences":
    [
      { "name":"P1M1" }
    ],
  "responses":
    [
      "a1b73279-465f-4f18-a44b-47993c3f75b9",
      "75f789c2-87b2-4c63-91de-ea6e5834e91d"
    ]
  "properties":
    {
      "Building":["Building A", "Building B"],
      "City":"Victoria",
    },
}
```

You can see that the top-level attributes are:

- recipients
- priority (a mapped-input parameter of 'xmpriority' automatically maps to this attribute)
- conferences (a mapped-input parameter of 'xmconferences' automatically maps to this attribute)
- responses (a mapped-input parameter of 'xmresponses' automatically maps to this attribute)
- properties (any mapped-data parameter that is not recipients, xmpriority, xmconferences, or xmresponses is assumed to be a property)

For example, xMatters will always use the 'xmpriority' mapped-input parameter to set the value of the 'priority' attribute in the JSON object.

Imagine that you are building an integration with a management system and want to pass in a parameter named 'priority' and map it to the priority attribute (instead of passing in an 'xmpriority' parameter). The following code, from the included sample integration service file, shows how this has already been configured for the sample communication plan:

```
<mapped-input method="add">
  <parameter>recipients</parameter>
  <parameter>priority</parameter>
  <parameter>building</parameter>
  <parameter>city</parameter>
</mapped-input>
```

The code included in the sample Javascript file illustrates the syntax used to remap the 'priority' top level attribute to the 'priority' parameter.

```
function apia_remapped_data() {
  return {
    "priority" : "priority"
  }
}
```

Note that the top-level attribute on the left is represented by its name in the JSON object, and not the 'xmpriority' parameter.

With these changes in place, xMatters uses the values passed in for the priority parameter to identify the value for the 'priority' attribute in the JSON body.

HTTP Interface

The HTTP interface allows a management system to inject an incident into the Integration Agent via direct HTTP request, with minimal formatting constraints, and without using the APClient.bin executable.

The advantages of using this interface include:

- The management system can send data to the Integration Agent even if it is not located on the same server. This allows cloud-based integrations, such as ServiceNow or BMC Remedy OnDemand, to access the Integration Agent.
- The management system can submit arbitrarily-formatted data via HTTP request.

The disadvantages of using this interface include:

- If asynchronous execution is disabled (as it is by default), the Integration Agent will not be able to process submitted data until the previous request has completed processing. Consequently, injections may be discarded if the integration is not carefully designed to avoid blocking conditions, or if no thread is immediately available to process the data.
- A load balancer is strongly recommended between the management system and the integration agent (when appropriate), and between the Integration Agent and the xMatters web servers.

Implementation

To use this interface, the management system and Integration Agent must be configured to send and receive HTTP requests.

Management system configuration

The management system must be configured to send an HTTP request to the Integration Agent. The target URL is constructed according to the following syntax:

```
http|https://<IntAgentIP>:<serviceGatewayPort>/http/<eventDomain>_<integrationService>
```

The Integration Agent IP address, service gateway port, Event Domain and Integration Service name must be defined in the `<IAHOME>\conf\IAConfig.xml` file. For example, the HP NNM i-series integration uses the following URL:

```
http://localhost:8081/http/hpnmmi_hpnnmi
```

Integration agent configuration

In addition to defining the Event Domain and Integration Service name in the `IAConfig.xml` file, the Integration Agent must have an apia_http() method defined. This is typically located at:

```
<IAHOME>\integrationservices\<eventDomain>\<integrationService>.js
```

For example, the HP NNM i-series integration uses the following default location:

```
<C:\Program Files\xMatters\Integration Agent\integrationservices\hpnnmi\hpnnmi.js
```

If a particular response is not expected by the management system, it is strongly recommended that you enable asynchronous execution of HTTP requests.

To enable asynchronous execution, include the following section in the integration configuration file (usually named `<integration_name>.xml`), before the <script> section:

```
<async-execution>true</async-execution>
```

To apply your changes, reload your integration or restart the Integration Agent.

If asynchronous execution is enabled, the Integration Agent enqueues the request and returns an HTTP 200 status without waiting for the request to be processed by the integration script.

Using the apia_http method

This method allows integrators to manipulate the incoming data and format it into JSON so it can be posted to xMatters (for example, by using the POST trigger REST API method). The HTTP entity sent via the HTTP interface can vary widely depending on its source and implementation. This function may need to handle incoming JSON, key-value pairs, XML, or some other format and syntax particular to the host system.

Due to this variety of potential implementations, the actual code contained within the apia_http function is impossible to predict. To assist integrators creating their own implementations, the apia_http function included in the `sample-relevance-engine.js` file provides an example of how to send a message through the Integration Agent via its HTTP interface. The example implementation is tied to the `sample-relevance-engine-http-client` shell script; this script contains a curl request that injects a sample XML document to the sample communication plan included with the Integration Agent.

For more information about creating communication plan events via the Integration Agent, see "Creating communication plan events" on page 80.

The apia_http method accepts two parameters: httpRequestProperties and httpResponse.

httpRequestProperties

This parameter accepts the following methods:

- `Enumeration propertyNames()`: Returns a set of available properties
- `String getProperty(String key)`: Returns the value of the specified property ("key")

For example, the following code will return the REQUEST_BODY of the HTTP request:

```
var requestBody = httpRequestProperties.getProperty("REQUEST_BODY");
```

Assuming that the request body contains an XML document, the resulting data can be used to construct an APXML message for submission to xMatters with minimal processing.

Sample HTTP request

The following is an example of an HTTP request (abbreviated for clarity) as received by the Integration Agent. It illustrates the REQUEST_BODY property in the context of the request as a whole:

```
{
  http.request=/http/hpnnmi_hpnnmi,
  REQUEST_BODY=
    <env:Envelope xmlns:env='http://www.w3.org/2003/05/soap-envelope'
        xmlns:wsa='http://www.w3.org/2005/08/addressing'
        xmlns:wse='http://schemas.xmlsoap.org/ws/2004/08/eventing'>
      <env:Header>
        <wsa:Action>Notification</wsa:Action>
      </env:Header>
      <env:Body><omitted></env:Body>
    </env:Envelope>,
  REMOTE_ADDR=/127.0.0.1:49529,
  Host=localhost:8081,
  User-Agent=JBossRemoting - 2.2.3.SP2,
  Transfer-Encoding=chunked,
  Connection=true,
  http.version=HTTP/1.1
}
```

httpResponse

You can use the httpResponse parameter to return the custom HTTP response back to the management system.

Note: *This parameter has the* `org.mule.providers.http.HttpResponse` *type. Refere to the Mule ESB documentation for more details.*

To simplify working with the httpResponse parameter, add the following lines to your code:

```
importClass(Packages.org.apache.commons.httpclient.Header);
importClass(Packages.org.apache.commons.httpclient.HttpVersion);
importClass(Packages.org.mule.providers.http.HttpResponse);
```

You can choose to return a custom HTTP response when an HTTP request could not be processed by the Integration Agent and has been discarded. To do this, implement the `apia_discard()` function, create an httpResponse object, initialize it by setting status, headers, and response body, and return. For example:

```
function apia_discard(httpRequestProperties, exception, attempts, time) {
var response = new org.mule.providers.http.HttpResponse();
response.setStatusLine(HttpVersion.HTTP_1_1, 400);
response.setBodyString("The request was discarded due to " + exception + " after " + attempts
+ " attempts");
return response;
}
```

Troubleshooting

To verify that an integration is installed correctly and accessible to the management system, use a browser on the management system server to access the following URL:

```
http://<IntAgentIP>:8081/http/<integrationName>_<integrationService>
```

For example:

```
http://localhost:8081/http/hpnmmi_hpnnmi
```

The following error response indicates that the integration is listening and accessible:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:apia="http://www.xmatters.com/apia_http_hpnnmi/">
  <soapenv:Header/>
    <soapenv:Body>
      <apia:TriggerResponse>
      <status>Error</status>
      <description>Request action not found.</description>
    </apia:TriggerResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

Input APXML Interface (APClient Requests)

A management system can initiate communication with an Integration Agent by submitting APXML messages that instruct the Integration Agent to perform various actions, which may include communication with remote servers and xMatters. APXML messages are submitted to the Integration Agent's APClient Gateway, which exposes an HTTP listener. The listener's URL has the following form:

```
[http|https]://<hostname or IP address>:<port>/agent
```

The exact form of the URL is determined by the `<apclient-gateway>` element in `IAConfig.xml`. For example, an Integration Agent with `<apclient-gateway ssl="false" host="localhost" port="2010"/>` would have the following APClient Gateway URL:

```
http://localhost:2010/agent
```

A Management System can issue an HTTP GET or POST command to the APClient Gateway URL. The following table lists the URL parameters that the APClient Gateway recognizes:

APClient Gateway Recognized Parameters

Term	Value	Mode	Description
`message`	A fully-formed APXML message, with or without the <?xml version="1.0"...?> preamble.	message	Specifies the singular message to submit.
`transactionid`	An integer in the range [0,2147483647].	mapped data	Specifies a client-defined ID that can be used to track the message (default is 0). Used to set the resulting APXML message's <transaction id="..."> element.
`mapdata`	The first instance identifies the target's Integration Service in the form <domain> or <domain>|<name>. Subsequent instances depend on the targeted Integration Service.	mapped data	Represents the ordered tokens to which the targeted Integration Service's datamap is applied to form the resultingAPXML message.

A single HTTP GET/POST is allowed to use only one of the submission modes (message or mapped data); for a mapped data submission, at least one `mapdata` parameter must be specfied. If the parameter values are submitted via HTTP GET, they must be URL encoded to remove any reserved or special characters (e.g., &, =, and non-ASCII).

For example, the following APXML message can be sent to the sample ping integration to create an event in xMatters:

```
<?xml version="1.0"?>
<transaction id='99'>
  <header>
    <method>Add</method>
    <subclass>Event</subclass>
  </header>
  <data>
    <agent_client_id>ping</agent_client_id>
    <recipients>bsmith</recipients>
    <situation>Server down.</situation>
    <device>localhost</device>
    <incident_id>TICKET-0100-0302</incident_id>
  </data>
</transaction>
```

In mapped data mode, a management system can submit this APXML message using the following HTTP GET:

```
http://localhost:2010/agent?transactionid=99
    &mapdata=ping
     &mapdata=bsmith
     &mapdata=Server+down%2E
     &mapdata=localhost
     &mapdata=TICKET-0100-0302
```

In message mode, a management system can submit this APXML message using the following HTTP GET:

```
http://localhost:2010/agent?message=%3C%3Fxml+version%3D%271%2E0%27%3F%3E
%3Ctransaction+id%3D%2799%27%3E
%3Cheader%3E
%3Cmethod%3EAdd%3C%2Fmethod%3E
```

```
%3Csubclass%3EEvent%3C%2Fsubclass%3E
%3C%2Fheader%3E
%3Cdata%3E
%3Cagent%5Fclient%5Fid%3Eping%3C%2Fagent%5Fclient%5Fid%3E
%3Crecipients%3Ebsmith%3C%2Frecipients%3E
%3Csituation%3EServer+down%2E%3C%2Fsituation%3E
%3Cdevice%3Elocalhost%3C%2Fdevice%3E
%3Cincident%5Fid%3ETICKET-0100-0302%3C%2Fincident%5Fid%3E
%3C%2Fdata%3E
%3C%2Ftransaction%3E
```

Note: *In these examples, the URL is formatted for ease of reading; the actual URL would be a single line.*

Regardless of the submission mode, the Integration Agent responds to the HTTP GET/POST with an APXML message that indicates the success/failure of the submission. An APXML submission is successful if the resulting APXML message is queued for processing by the targeted Integration Agent. The submission may still fail if it specifies invalid data or some other runtime exception occurs (e.g., database or network failure). The response to a successful APXML submission is an Agent/OK APXML message with no data tokens.

The following example demonstrates the response to a successful APXML submission to the sample pingintegration:

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="99">
  <header>
    <method>Agent</method>
    <subclass>OK</subclass>
  </header>
  <data/>
</transaction>
```

Note: *The* `transactionid` *in the response APXML message will match the submitted APXML message's transactionid.*

The response to an unsuccessful APXML submission is an Agent/ERROR APXML message with data tokens that describe why the submission was unsuccessful.

The following example demonstrates the response to an unsuccessful APXML submission that specified both message and mapped data:

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="1">
  <header>
    <method>Agent</method>
    <subclass>ERROR</subclass>
  </header>
  <data>
    <incoming_message>/agent?message=...&amp;mapdata=ping...</incoming_message>
    <errordetail>java.lang.IllegalArgumentException:   The payload...</errordetail>
    <errorcode>INVALID_SUBMISSION</errorcode>
    <errortext>The payload defines both message and data map
            submissions; only one submission method is allowed per
            request.</errortext>
  </data>
</transaction>
```

Note: *Some of the values in the latter example have been truncated for readability.*

The following table summarizes the error codes returned when an APXML submission fails:

Error Codes Returned on APXML Submission Failure

Error Code	Description
INVALID_SUBMISSION	Unrecognized or missing URL parameters.
INVALID_MESSAGE	A message submission with malformed APXML or unrecognized Integration Service target (i.e., agent_client_id).
DATAMAP_FAILED	An unrecognized Integration Service target (i.e., the first token), or an Integration Service target without a data map.
AGENT_ERROR	Any other error during processing, including denial of service (e.g., invalid access password).

APClient.bin

While it is possible for a client to submit requests directly to the APClient Gateway via HTTP GET/POST, the request formatting and URL encoding can be cumbersome. `APClient.bin` is a native binary (i.e., operating system-specific) program that provides a simplified interface to the APClient Gateway. It is located in `<IAHOME>/bin` and is named `APClient.bin` for Unix systems and `APClient.bin.exe` for Windows systems.

If `APClient.bin` is moved to another location, the APClient Gateway must be configured to use port 2010, or a folder named `etc` must be created in the same folder containing `APClient.bin` and within `etc` must be located a file named `.runtime.xml` with the following contents (replace 2010 with the actual APClient Gateway port):

```
<?xml version="1.0" encoding="UTF-8"?>
  <alarmpoint-agent-runtime version="1.0">
     <http port="2010"/>
  </alarmpoint-agent-runtime>
```

`APClient.bin` accepts command-line parameters and formulates the corresponding HTTP POST to the Integration Agent's APClient Gateway. The Integration Agent's APXML response message is written to stdout or, optionally, to a file.

The following table summarizes the OS exit codes that `APClient.bin` returns to the caller:

OS Exit Codes APClient.bin Returns to Caller

Exit Code	Description
0	The Integration Agent accepted the connection request and HTTP POST data. Note that this does not mean that the submission succeeded; the Integration Agent may still have responded with an Agent/ERROR APXML message.
10	Socket communications error (could not establish a connection).
20	Error parsing the command-line parameters.
30	Not enough available RAM to continue operation.
40	File-related error (e.g., a file could not be opened).
50	Error with the request.

Exit Code	Description
250	Unexpected error.

The following table summarizes the command-line parameters that `APClient.bin` accepts and that can be used for both message and map data submissions (note that all parameters are optional):

Command-line Parameters that APClient.bin Accepts

Parameters	Values	Description
`--post-file`	Relative or absolute file path	Redirects output to the specified file.
`--http-post`	APClient Gateway URL	Allows submissions to remote/arbitrary Integration Agents (`etc/.runtime.xml` is ignored).
`--recover-file`	Relative or absolute file path	Retries submitting the previously-failed submissions that were recorded in the specified file.
`--help`	N/A	Displays usage information.
`--version`	N/A	Displays version information.

The `--recover-file` parameter is part of `APClient.bin`'s recovery mechanism. If `APClient.bin` is unable to contact the targeted Integration Agent, it writes a recovery message to exactly one of the following files, in the order listed:

```
logs/APClient.log
./APClient.log
. /tmp/APClient.log or C:\tmp\APClient.log\
```

If the file already exists, the recovery message is appended to the end of the file.

When the `--recover-file` parameter is used to refer to one of these recovery files, `APClient.bin` renames the recovery file to `APClient.log.recover` and then attempts to resubmit the failed submissions recorded by the recovery messages. Any submissions that fail to be resubmitted are logged in a new recovery file using the same process as previously described. Once you have verified that all recovered submissions were successfully resubmitted, you can delete `APClient.log.recover`.

In addition to the parameters that have been described, `APClient.bin` also recognizes parameters that are specific to message and map data submissions; the following subsections describe how to use `APClient.bin` to make these submissions.

Notes:

APClient.bin supports only HTTP communication with the Integration Agent, even if the value of the --http-post parameter is an https:// URL (however, the Integration Agent can communicate back to the Mangement System using any of the secure protocols supported by the Management System's API.).

If the Integration Agent is configured to use HTTPS for Management System communication (see the apclient-gateway entry in "Integration Agent Configuration File" on page 18), all APClient.bin submissions will be rejected. To communicate with the Integration Agent, the Management System must form its own HTTPS requests as described in "Input APXML").

Message Submissions

The following example demonstrates how `APClient.bin` is used to generate a message submission to the sample `ping` Integration:

```
APClient.bin '<?xml version="1.0"?>
                <transaction id="99">
                  <header>
                    <method>Add</method>
                    <subclass>Event</subclass>
                  </header>
                  <data>
                    <agent_client_id>ping</agent_client_id>
                    <recipients>bsmith</recipients>
                    <situation>Server down.</situation>
                    <device>localhost</device>
                    <incident_id>TICKET-0100-0302</incident_id>
                  </data>
                </transaction>'
```

Note: *In this example, the command is formatted for ease of reading; the actual command would be on a single line.*

The APXML message that is passed to `APClient.bin` must be quoted. Single quotes will work with most Unix shells and do not require modication of the APXML. However, for Windows and some Unix shells, double quotes must be used and any double quotes in the APXML message must be changed to single quotes, as shown in the following example:

```
APClient.bin.exe "<?xml version='1.0'?>
                <transaction id='99'>
                  <header>
                    <method>Add</method>
                    <subclass>Event</subclass>
                  </header>
                  <data>
                    <agent_client_id>ping</agent_client_id>
                    <recipients>bsmith</recipients>
                    <situation>Server down.</situation>
                    <device>localhost</device>
                    <incident_id>TICKET-0100-0302</incident_id>
                  </data>
                </transaction>"
```

Note: *Any of the optional parameters (e.g., http-post), must appear before the APXML message.*

The `--submit-file` parameter specifies a file within which each line is an APXML message to submit; its value is a relative or absolute file path. This parameter is a convenient way to perform multiple message submissions. The following example shows the contents of a file that could be used with the `--submit-file` parameter to send two consecutive message submissions to the sample ping Integration:

```
<?xml version="1.0"?><transaction id="99">...<incident_id>TICKET-
  0100-0302...</transaction>
<?xml version="1.0"?><transaction id="100">...<incident_id>TICKET-
  0100-0303...</transaction>
```

Note: *Some of the values in the example have been truncated for readability. Additionally, it is not necessary to add quotes around each APXML message.*

Map Data Submissions

The following example demonstrates how `APClient.bin` is used to submit a map data submission to the sample ping integration:

```
APClient.bin --map-data-transaction-id 99 --map-data ping bsmith
  "Server down." localhost TICKET-0100-0302
```

The `--map-data-transaction-id` parameter maps directly to the `transactionid URL` parameter. It is optional and the transaction ID defaults to 1 if not specified. The `--map-data` parameter is a space-separated list of strings, each of which maps to a `mapdata URL` parameter. If one of the strings contains a space, such as "Server down", then it must be surrounded by single or double quotes; otherwise, each component of the string will be treated as its own `mapdata URL` parameter.

Note: *When present,* `--map-data` *must always be the last parameter on the* `APClient.bin` *command line.*

The following table summarizes the `APClient.bin` parameters that are relevant to map data submissions:

Command-line Parameters that APClient.bin Accepts

Parameter	Values	Description
`--map-data-transaction-id`	An integer in the range [1,2147483647]	Specifies the resulting APXML message's transaction ED. Defaults to 1 if not specified.
`--map-data`	One or more (optionally quoted) strings	Specifies the space-separated list of `mapdata` values. The first value identifies the targeted integration service. The first value identifies the targeted integration service or, if their names are not identical, the event domain and the integration service enclosed in quotes and separated by a pipe: `"applications|sample-relevance-engine"`
`--map-data-file`	A relative or absolute file path	Specifies a file containing lines of the form: `<transactionid><mapdata_1> <mapdata_2> : : :` Each line is submitted as a map data submission.

The `--map-data-file` parameter is a convenient way to perform multiple map data submissions. The following example shows the contents of a file that could be used with the `--map-data-file` parameter to send two consecutive map data submissions to the sample ping Integration:

```
99 ping bsmith "Server down." localhost TICKET-0100-0302
100 ping bsmith "Server down." 192.168.168.55 TICKET-0100-0303
```

Note: *It is still necessary to add quotes around any mapdata parameter that contains spaces. If the name of the event domain and the name of the integration service are not the same, they must be separated by a pipe and quoted. See the "--map-data" section for an example.*

Remote Integration Agent submissions

You can use `APClient.bin` to submit an event to a remote Integration Agent. Copy `APClient.bin` to the source system, and then submit a map-data request using the following syntax:

```
APClient.bin --http-post http://<IPAddress>:2010 --map-data <IntegrationService>
<EventDetails>
```

Where:

- `<IPAddress>` is the IP address of the target Integration Agent.
- `<IntegrationService>` is the event domain and integration service you want to target (e.g., `applications|sample-relevance-engine`).
- `<EventDetails>` is the map data for the event.

Applying Mapped Input to Map Data

When the Integration Agent receives a map data submission, either directly through an HTTP GET/POST or indirectly via `APClient.bin`, it uses the first map data value to determine the targeted Integration Service and the <mapped-input> element from this Integration Service's configuration file to transform the remaining map data values into an APXML message.

Example

Assume a client issues the following command:

```
APClient.bin --map-data-transaction-id 99 --map-data ping bsmith "Server down." "" TICKET-
0100-0302 999 test1 test2
```

This creates a map data submission with the following map data values:

- ping
- bsmith
- Server down.
- n/a
- TICKET-0100-0302
- 999
- test1
- test2

Since the first map data value is "ping", the Integration Agent uses the `<mapped-input>` element from the ping Integration Service. The sample ping Integration Service has the following `<mapped-input>` element:

```
<mapped-input method="add" subclass="event">
  <parameter type="string">recipients</parameter>
  <parameter type="string">situation</parameter>
  <parameter type="string">device</parameter>
  <parameter type="string">incident_id</parameter>
  <parameter type="numeric">my_first_constant</parameter>
  <parameter>my_second_constant</parameter>
</mapped-input>
```

Based on this `<mapped-input>` element, the Integration Agent creates an add/event APXML message with transaction ID 99 (from the submission's `--map-data-transaction-id` parameter). The Integration Agent then applies each `<parameter>` element, in the same order in which they appear within the `<mapped-input>` element, to the remaining map data values.

Each `<parameter>` element defines an APXML token key whose value is the corresponding map data value. If the map data value is an empty string (this is the case with the fourth map data value in this example), the corresponding `<parameter>` element is ignored.

The optional `type` attribute can be used to explicitly type the resulting APXML token; if omitted, the token is auto-typed. If there are fewer `<parameter>` elements than map data values, the additional map data values are ignored. Similarly, if there are more `<parameter>` elements than map data values, the additional `<parameter>` elements are ignored.

For this example, the Integration Agent creates the following APXML message:

```
<?xml version="1.0"?>
<transaction id="99">
  <header>
    <method>add</method>
    <subclass>event</subclass>
  </header>
  <data>
```

```
      <recipients type="string">bsmith</recipients>
      <situation type="string">Server down.</situation>
      <incident_id type="string">TICKET-0100-0302</incident_id>
      <my_first_constant type="numeric">999</my_first_constant>
      <my_second_constant>test1</my_second_constant>
    </data>
  </transaction>
```

The resulting APXML message has the following properties:

- There is no `<device>` APXML token because the corresponding map data value is an empty string.
- The `<my_second_constant>` APXML token is auto-typed since the defining `<parameter>` element has no type attribute.
- There is no APXML token for the "test2" map data value because there are seven map data values that need transformation and only six `<parameter>` elements.

Once transformed into an APXML message, a map data submission is processed in exactly the same fashion as a message submission; therefore, the following sections apply to both submission types.

Applying Constants to APXML Messages

Each Integration Service's configuration file contains a `<constants>` element, which is a well-defined and secure location for Integration Services to define certain APXML tokens that will always be added to a submitted APXML message.

To better understand the need for the `<constants>` element, consider the following situation and the resulting implications: an Integration Service receives an APXML message instructing it to create a ticket within a password-protected Management System. To specify the access password, clients might be required to include the password in their submission, but this has the disadvantage of disseminating multiple copies of the password (one to each client), which is both a security risk (clients must securely store and transmit the password to evade hackers/eavesdroppers), and a maintenance issue (clients must be updated whenever the password changes).

Alternatively, the password could be included as part of the service's JavaScript implementation, which eliminates the need the to inform clients of password changes; however, this is also a potential security risk (the password may be stored in cleartext in the source code), and a maintenance issue (the source code must be inspected and changed wherever the password changes).

The `<constants>` element resolves this issue. When a Management System access password as an encrypted constant, clients are no longer required to provide the password, the password is securely stored in a single location, and the password can be easily changed without accessing into source code. Instead, the Service's implementation will extract the password from the submitted APXML message whenever it needed to access the Management System.

Clients do not have to explicitly submit an APXML token that is defined as a constant. If a client explicitly submits an APXML token that is defined as a non-overwriteable constant, the token's value is replaced with the value defined in the Integration Service's configuration (i.e., the token is guaranteed to have the value defined in the configuration).

It is sometimes useful to relax this condition and allow clients to overwrite a constant's value; in this case, the overwriteable constant acts as a default value. For example, assume the Management System account that is used by an Integration Service is defined by `mgmt_user` and `mgmt_pwd` APXML tokens. The Integration Service could define a default Management System account as overwriteable `mgmt_user/mgmt_pwd` constants, in which case clients that submitted APXML messages without `mgmt_user/mgmt_pwd` tokens would use the default account, while all other clients would use the account they specified.

The sample `ping` Integration Service configuration contains the following `<constants>` element:

```
<constants>
  <constant name="device" type="string" overwrite="false">localhost</constant>
  <constant name="my_first_constant">This is an auto-typed constant...</constant>
  <constant name="my_second_constant" type="string"
     overwrite="true">This is a string constant...</constant>
```

```
    </constants>
```

Each <constant> element has `name` and `type` attributes, which directly correspond to the resulting APXML token's key.and type. The `type` attribute is optional, in which case the APXML token is auto-typed (for details, see "APXML Tokens" on page 70).

The optional `overwrite` attribute defines the behavior of the constant when a client explicitly submits the resulting APXML token. If `overwrite` is omitted or "false", then the constant is ignored if a client explicitly submits an APXML token with the same name (i.e., both the type and value of the submitted token is preserved). If `overwrite` is "true", then the resulting APXML token will always replace any explicitly submitted token with the same key (i.e., the explicitly submitted token is ignored).

For example, assume the following APXML message is submitted to the sample `ping` Integration Service:

```
<?xml version="1.0"?>
<transaction id="99">
  <header>
    <method>add</method>
    <subclass>event</subclass>
  </header>
  <data>
    <recipients type="string">bsmith</recipients>
    <situation type="string">Server down.</situation>
    <incident_id type="string">TICKET-0100-0302</incident_id>
    <my_first_constant type="numeric">999</my_first_constant>
    <my_second_constant>test1</my_second_constant>
  </data>
</transaction>
```

After applying the constants, the submitted APXML message results in the following APXML message:

```
<?xml version="1.0"?>
<transaction id="99">
  <header>
    <method>add</method>
    <subclass>event</subclass>
  </header>
  <data>
    <recipients type="string">bsmith</recipients>
    <situation type="string">Server down.</situation>
    <incident_id type="string">TICKET-0100-0302</incident_id>
    <my_first_constant type="numeric">999</my_first_constant>
    <my_second_constant type="string">This is a string constant...</my_second_constant>
    <device type="string">localhost</device>
  </data>
</transaction>
```

Although the `device` <constant> element has an implicit overwrite="false", a `device` APXML token is added because it was not originally in the submitted APXML message. The `my_first_constant` APXML token is unaffected, even though it differs from the my_first_constant <constant> element, since this constant has overwrite="false". In contrast, the `my_second_constant` APXML token is overwritten by the definition of the `my_second_constant <constant>` element since this constant has overwrite="true".

There is also a corresponding `<encrypted-constant>` element that is processed in the same way as a <constant> element, except that the value of the resulting APXML token is the decrypted contents of an encrypted file created by `iapassword`. For example, suppose the following `iapassword` command is used to create an encrypted file located at `/tmp/.constant`:

```
iapassword --new "This is a string constant..." --file /tmp
             /.constant
```

The sample `ping` Integration Service's `<constants>` element could be modified to use the encrypted file as follows:

```
<constants>
  <constant name="device" type="string" overwrite="false">localhost
</constant>
  <constant name="my_first_constant">This is an auto-typed constant...</constant>
```

```
    <encrypted-constant name="my_second_constant" type="string"
       overwrite="true"><file>/tmp/.constant</file>
    </encrypted-constant>
  </constants>
```

Note: *The* `<encrypted-constant>` *element contains a* `<file>` *sub-element whose path can be absolute or relative. Relative paths are resolved against the location of the Integration Service's configuration file.*

The APXML message that results after applying the modified `<constants>` element would be exactly the same as in the previous example since in both cases the value of the `my_second_constant` APXML token is "This is a string constant...".

Configuring Auto Recovery of APClient.bin Messages on Integration Agent Startup

When APClient.bin is executed but a connection cannot be made to the Integration Agent, the message is saved in the `APClient.log` file. The `--recover-file` parameter to APClient.bin can be used to resubmit these messages, or the file can be placed in the recovery directory and the Integration Agent will recover those APClient.bin messages. When auto recovery is enabled, APClient.bin processes messages on a first-in first-out basis. This means that messages queued while the Integration Agent is unavailable will be processed before any new messages submitted after the Integration Agent starts.

Several parameters set when creating the APClient Gateway determine this feature's behavior. These configuration items are located in the Integration Agent's `spring-config.xml` file located at `<ia_home>/conf`. The following is a sample:

```
  <!--
   | Create the APClient gateway UMO, which exposes an HTTP endpoint to the APClient and
  enqueues Integration Service requests.
   +-->
  <bean id="apclientGateway" class="com.alarmpoint.integrationagent.
  APClientGatewayImpl" scope="singleton" init-method="initialize">
   <property name="logger" value="com.alarmpoint.integrationagent.apclient"/>
   <property name="muleHelper"><ref local="muleHelper"/></property>
   <property name="servicesManager"><ref local="servicesManager"/></property>
   <property name="automaticRecovery" value="false" />
   <property name="recoveryInterval" value="10" />
   <property name="recoveryFileDir" value="file:./bin/logs" />
   <property name="recoveryLockFile" value="file:./bin/logs/alock" />
   <property name="recoveryFileCharset" value="UTF-8" />
  </bean>
```

The following table describes these parameters:

Property	Value	Function
automaticRecovery	Boolean	Value of "true" enables automatic recovery; value of "false" disables the features.
recoveryInterval	long	Amount of time in seconds between the completion of the previous recovery of APClient.bin messages and the next check for recovery files.
recoveryFileDir	string	Location of the recovery directory. If automaticRecovery is enabled, this directory must exist; otherwise, the Integration Agent will fail to start.
recoveryLockFile	string	File that the Integration Agent locks when it is processing a recovery file. The specified lock file must match the location and name of the lock file that is used by APClient.bin to synchronize the recovery mechanism's access to an active recovery file with APClient.bin.

Property	Value	Function
recoveryFileCharset	string	Encoding used for characters in the recovery file.

Notes

- The default value for recoveryFileDir and recoveryLockFile is the name and location of the lock file that APClient.bin will use if it is executed from `<ia_home>/bin`. If APClient.bin is used with an integration or run from a batch file, this setting should be changed to the directory where APClient.bin will be executed.
- recoveryFileDir and recoveryLockFile are global settings; if multiple processes execute APClient.bin from different directories, only one location can be chosen.
- To determine the name and location of recoveryFileDir and recoveryLockFile, run APClient.bin as normal while the Integration Agent is not active and note where these files are created.
- The recovery directory can have more than one recovery file. A recovery file named `APClient.log` will be considered the current file to process. All other recovery files *must* begin with APClient.log and have a suffix of **.x** where **x** is an integer (e.g., APClient.log.7). The recovery files will then be processed in ascending numerical order.

Inbound Queue Model

The Integration Agent uses a system of queues to ensure that incoming requests are processed in the correct order. The queues also provide fault tolerance if a message cannot be processed, or if the Integration Agent is interrupted (eg, by a power outage.)

Each integration service has a two inbound queues, labeled "normal" and "high." These queues correspond to the value of the incoming request's "apia_priority" token. For example, the generic integration service has an "inbound.generic.generic.normal" queue and an "inbound.generic.generic.high" queue. If the request does not have an "apia_priority" token when the Integration Agent receives it, then one will be created and assigned a value of "normal."

The high queue does not receive additional resources compared to the normal queue; these queues simply allow the user to differentiate requests from each other so they can prevent time-sensitive requests being delayed by the processing of less-urgent requests. For example, if the Integration Agent has 1,000 requests in its normal queue, the next request might have a significant wait before being processed. If the request has a high priority, however, and is sent to the "high" queue, then the message will be processed much sooner. For more information on this behavior, see the discussion of concurrency settings in the next section.

Each queue is partitioned into a variable number of sub-queues to handle the "apia_process_group" token values within the requests. If several requests are in the integration service's normal queue and they all have the same "apia_process_group" token, they will all be put in the same sub-queue and processed in FIFO order. If the request does not have an "apia_process_group" token when the Integration Agent receives it, one will be created and assigned a value of "apia_default_group".

Requests whose "apia_default_group" tokens do not match will be processed concurrently, as described in the following section.

The following is a brief overview of how the Integration Agent uses its inbound queues, along with the messages that appear in the Integration Agent log. For the sake of simplicity, assume that all requests have the default value in their "apia_process_group" tokens.

- When a request is received (e.g., from the apia_gateway or the service gateway), the httpConnector.receiver component determines the integration service to which it belongs, and sends the message to the appropriate queue.

```
[httpConnector.receiver.3] INFO - Component generic_generic has received the following request
from endpoint http://10.2.0.126:8081/http/generic_generic... (request data omitted)
[httpConnector.receiver.3] INFO - The Integration Service (generic, generic) is enqueueing the
following message for processing: ...
```

```
[ActiveMQ Transport: tcp:///127.0.0.1:43351] DEBUG - localhost Message ID:vic-vw-jb-ia-test-
43350-1490220977967-1:1:3:1:1 sent to queue://response.generic.generic.normal
```

- When the integration service has an available worker process (thread) available, the request is copied from the queue and sent to the integration service:

```
[inbound.generic.generic.normal-1] INFO - Component generic_generic has received the following
request from endpoint jms://inbound.generic.generic.normal...
```

For more information about the various APXML request types, see "Integration Service Scripts" on page 104.

Note: The request has not yet been removed from the inbound queue.

- If the worker process finishes processing the request with success, the request is removed from the inbound queue and the worker process becomes available to handle the next request:

```
[httpConnector.receiver.3] INFO - Component generic_generic has finished processing the
request from endpoint http://10.2.0.126:8081/http/generic_generic...
[inbound.generic.generic.normal-1] INFO - Component generic_generic has finished processing
the request from endpoint jms://inbound.generic.generic.normal...
[ActiveMQ Transport: tcp:///127.0.0.1:43351] DEBUG - queue://response.generic.generic.normal
remove sub: QueueSubscription: consumer=ID:vic-vw-jb-ia-test-43350-1490220977967-1:1:2:1
```

In this context, "success" is determined by execution of the final instruction in the function, at which point the Integration Agent will handle any returned data. Then the worker process will become available once again to handle new requests. At this time, the original request is removed from the inbound queue.

For more information about handling of returned data, see the Integration Service Scripts section, below.

If the request times out, or if an exception is thrown during processing, then the request is kept on the queue and a copy is once again sent to the appropriate function. By default, the Integration Agent will try two more times for a total of three processing attempts. After the third attempt, the Integration Agent will attempt to send the request to the apia_ discard function, if one exists. (Most integration services do not have an apia_discard function.) See the "apia_discard method" section of this document for details.

After the Integration Agent attempts to invoke apia_discard, the original request is removed from the inbound queue and the worker process becomes available for re-use.

The Integration Agent uses ActiveMQ to manage its inbound and outbound queues. By default, ActiveMQ uses a file-based DBMS called KahaDB to manage the queue contents. This ensures that the queue contents are persisted if the Integration Agent is restarted or otherwise interrupted. ActiveMQ can optionally be cofigured to use an external Oracle or SQLServer DBMS, which can be shared with other Integration Agents to provide fault tolerance; i.e., if one Integration Agent stops working, another will take over responsibility for the items in the queues. For more information, see "Configuring Integration Agents for shared persistent queue operation" on page 29.

Integration Service Scripts

When the Integration Agent pulls a request from one of its inbound queues and assigns it to a worker process, the worker process will attempt to find an appropriate function to handle the request using the following rules:

- If the request is of type APXML, and the apia_source token begins with "apclient" or "integration", the message is handled by the apia_input function.
- If the request is of type APXML, and the apia_source token begins with "alarmpoint", the message is handled by the handleResponse function.
- If the request is not of type APXML, it is sent to the apia_http function. Unlike the other functions, apia_http will accept any form of data that can be sent to it via an HTTP request, typically SOAP or JSON The other functions accept only APXML data.

Multi-Threading and Concurrency

Since the Integration Agent provides multi-threading capability, each integration service can have a specific number of threads allocated to its "normal" and "high" priority requests.

The number of threads is defined in the "concurrency" section of the integration service's XML definition file (e.g., `<IAHOME>\integrationservices\generic\generic.xml`). For example:

```
<concurrency>
  <normal-priority-thread-count>3</normal-priority-thread-count>
  <high-priority-thread-count>3</high-priority-thread-count>
</concurrency>
```

Each message has a "priority" attribute, as defined by the apia_priority token. This token and the apia_process_group token should be created and assigned an appropriate value *before* the request is sent to the Integration Agent, to ensure that Add and Del requests are handled correctly and in FIFO order. If the apia_priority token is not populated when the request is received, the token will be created and assigned the default value of "normal", at the time when the script output is sent to the outbound queues. Similarly, the apia_process_group token will be automatically created and provided the default value of "apia_default_group" if it doesn't already exist.

To summarize, requests will be handled concurrently only if the following conditions are true:

- Concurrency settings in the integration service's XML definition file are greater than "1".
- Requests are pre-populated with apia_priority and apia_process_group tokens with appropriate values.

Priority

As described in the Inbound Queue Model section, the normal and high priority queues are partitioned according to the values of the requests' apia_process_group tokens. Any messages with matching apia_priority and apia_process_group tokens will be processed sequentially in FIFO order, but requests with non-matching values in these tokens will be processed concurrently. Using the concurrency settings shown in the example above, the integration service will be able to process up to six requests concurrently, assuming that each of the normal and high priority queues has at least three messages with individual apia_process_group tokens.

As previously mentioned, the high queue does not receive more resources than the normal queue by default. This design minimizes delays for more urgent requests. For example, if all requests take one second to process, and the normal queue has 1,000 requests already in it, then the next message sent to the queue would be delayed by 1,000 seconds before being processed. But if the high queue were reserved for the processing of urgent messages, then delays would be minimized for these messages.

This model allows integration designers to manage special handling of high-priority requests. However, if the majority of requests are assigned a high priority token, then the high priority queue will fill up more quickly than the normal priority queue, which would be counterproductive. More threads can be assigned to the high queue if required via the concurrency settings detailed above.

Non-APXML messages (e.g., SOAP or JSON messages sent to the apia_http function via the service-gateway) do not have apia_priority or apia_process_group tokens, and are always managed via the normal queue.

Script Timeouts

Once a request is assigned to a worker thread, the Integration Agent allows 30 seconds (by default) for processing to finish. This is typically plenty of time, but if the request entails time-consuming operations (such as calls to an external web service or requests for enrichment data), the script may time out. For more information, see "Errors and Retries While Processing Inbound Queue" on page 122.

Return Values

Assuming that the script process succeeds, any returned value is returned to the originator of the request. Specifically,

- The apia_input function is expected to return an APXML message, which is automatically directed to the Integration Agent's outbound queues. See the "Outbound Queue Model" section for details. Alternatively, apia_ input is allowed to return null.
- The apia_http function is expected to return an HttpResponse object (as defined by www.w3.org). apia_http is also allowed to return null, which case the Integration Agent automatically generates an HTTP 200 "OK" response. The response is sent to the requester (i.e., the external entity that sent the HTTP request to the Integration Agent's service-gateway interface.)
- APXML requests from xMatters with a method token value of "OK" are not expected to return anything. These are automatically directed to the apia_response function and are intended only to acknowledge that an APXML message was received by xMatters.
- Similarly, APXML requests from xMatters with a method token value of "ERROR" are not expected to return anything. These are directed to the apia_response function and indicate that xMatters was not able to accept an APXML message from the Integration Agent.
- APXML requests from xMatters with a method token value of "SEND" are not expected to return anything. These are directed to the apia_response function and contain an APXML ExternalServiceMessage (ESM) from the xMatters server.
- APXML requests from xMatters with a method token value of "REQUEST" are required to return an APXML response. These requests are directed to the apia_response function and contain an APXML ExternalServiceRequest (ESR) from the xMatters server.

Regardless of whether the script returns anything, it is considered to have terminated successfully when the last instruction in the function has finished executing. If this doesn't happen within the request-timeout period, a ServiceTimeoutException is thrown and the Integration Agent resubmits the original request from the inbound queue. For more information, see "External Service Message/Request Processing" on page 114.

Reserved Function Names

The following function names are reserved to provide specific functionality:

apia_input

- **input**: APXML, received automatically from the IA's inbound queues
- **output**: APXML, sent automatically to the IA's outbound queues via the "return" instruction
- **purpose**: processing of notification requests from a management system, typically via apclient.bin or via direct HTTP POST.

apia_http

- **input**: SOAP or JSON data (typically, but it can be anything that can be transported via an HTTP request)
- **output**: HttpResponse object (as defined by www.w3.org)
- **purpose**: processing of non-APXML notification requests from a management system, via direct HTTP POST.

apia_discard

- **input**: see the apia_discard method section of this document
- **output**: APXML or HttpResponse object (depending whether apia_discard was called by apia_input or apia_http)
- **purpose**: allows a request to be handled by alternative logic, if the normal logic times out or throws a RetriableException. For more information, see "Errors and Retries While Processing Inbound Queue" on page 122.

apia_response

- **input**: APXML message (from the xMatters instance)
- **output**: APXML (only if the input includes a "method" token with a value of "REQUEST")

- **purpose**: handles messages from the xMatters instance. Despite its name, apia_response may receive messages that are not responses; e.g., the message may be a message delivery annotation. For more information, see "Response Action Scripting" on page 110.

The following function names are also reserved and described in detail in "Lifecycle hooks" on page 124.

- apia_startup
- apia_shutdown
- apia_suspend
- apia_resume
- apia_interrupt
- apia_webservice_init *(deprecated and no longer in use, but still reserved)*

Handoff to Outbound Queues

As previously mentioned, the data returned by the apia_input function is automatically added to the Integration Agent's outbound queues, as seen in these Integration Agent log messages:

```
[generic|generic-1] INFO - The Integration Service (generic, generic) is enqueueing the
following APXML message for delivery to xMatters: (message omitted)...
[ActiveMQ Transport: tcp:///127.0.0.1:46881] DEBUG - localhost Message ID:vic-vw-jb-ia-test-
46880-1490390510511-1:1:3:1:1 sent to queue://outbound.generic.generic.normal
```

The apia_http function cannot automatically send an APXML message to the outbound queues simply by returning an appropriate value. However, any function (including apia_input, apia_discard, etc.) can send APXML data to the outbound queues by explicitly calling the ServiceAPI.sendAPXML function.

The ServiceAPI.sendAPXML function can be called directly, or by using the xMattersWS convenience functions defined in the utilities bundle, available for download from the same web page as the Integration Agent. The xMattersWS submitApxml function provides additional functionality to the ServiceAPI.sendAPXML function, including synchronous deletion of existing notifications. For this reason, use of the xMattersWS submitApxml function is recommended over the ServiceAPI.sendAPXML function.

APXML data automatically receives some processing before it is placed in the outbound queues:

- The agent_client_id token is automatically set to the event domain and integration service name of the current integration. This is noteworthy because clients occasionally have tried to spoof the event domain and/or integration service name, and were not expecting this data to be overridden during the transition to the Integration Agent's outbound queues.
- If it does not already exist, the company token is added, and its value is set to the <web-services-auth>/<company> element in `IAConfig.xml`.
- If it does not already exist, the apia_priority token is created and its value set to "normal".

Non-APXML data does not get handled by the Integration Agent's outbound queues. Specifically, non-APXML requests from the Integration Agent to xMatters are sent directly (e.g., via the GET(), Send(), etc utility functions in xmio.js). These functions do not provide fault tolerance, except for the Integration Agent's built-in retry mechanism which is activated if a script throws a retriable exception. For more information, see "Errors and Retries While Processing Inbound Queue" on page 122.

It is possible to add retry logic when invoking the xmio utility methods, but this is not recommended for the following reasons.

Firstly, there is a limit on the execution time allocated to the integration script (30 seconds by default). After this limit is exceeded, the original request is retrieved from the inbound queue and execution begins from scratch. This implies that if xmio.post() were invoked from a while() loop, for example, with an exit condition based on a successful response from xmio.post, the contents of the while loop would be executed only once in the event of a network timeout error, and then the integration script would time out. Therefore the while loop would achieve nothing.

Secondly, subsequent inbound messages cannot be processed by the Integration Agent until processing of the previous message has been completed. Increasing the script timeout setting is likely to cause upstream errors. For example, if the script timeout were increased to 60 seconds, and two HTTP requests were sent to the Integration Agent from the management system, the management system's original HTTP request would time out before the Integration Agent had prepared an appropriate response.

In most integrations that accept HTTP data, a custom response is provided to the management system to indicate whether processing was successful. If the original request times out while the integration script is still processing it, the management system will not receive a response at all. In the best case, this will simply mean that the incident ticket will not be updated with a "submitted to xMatters" annotation. In less favourable cases, the management system will fail to handle the resulting exception, and may result in reliability problems.

These reasons apply equally to SOAP requests sent from the Integration Agent; e.g., via the wsutil sendReceive() function.

In summary, only APXML data is sent via the Integration Agent's outbound queues. Other types of outbound messages (such as SOAP and REST requests) are not sent via the outbound queues, and there is no fault tolerance associated with non-APXML outbound messages or requests.

Outbound Queue Model

Each Integration Service has its own persistent outbound queue, which stores APXML messages that are to be sent to xMatters. Before an APXML message is added to an Integration Service's outbound queue, the Integration Agent adds or resets the following APXML tokens:

- `company_name` is set to the Company specified by the `web-services-auth>/<company>` element in the Integration Agent's configuration file.
- `agent_client_id` is set to `<domain>|<name>` (e.g., for the sample `ping` Integration Service, all outbound APXML messages contain `<agent_client_id>ping|ping</agent_client_id>`)
- `apia_priority` is set to "normal" if the APXML message does not already contain an `apia_priority` token with a recognizable value (i.e., normal or high).

Periodically, as configured by the `<apxml-exchange>/<interval>` element in the Integration Agent's configuration file, an exchange of APXML messages occurs between the Integration Agent and xMatters. The exchange is initiated by the Integration Agent via calls to AlarmPoint's `SubmitAPXML` and `ReceiveAPXML` web service methods.

Before calling `SubmitAPXML`, the Integration Agent uniformly selects, in first-come-first-served order, a set of normal and high priority messages from each Integration Service's outbound queue, up to (approximately) the total configured by the `<apxml-exchange>/<size>` element in the Integration Agent's configuration file. These selected messages are then sent as part of the `SubmitAPXML` call.

For example, assume an Integration Agent contains three Integration Services with the following outbound queue contents:

Example: Initial Outbound Queue Sizes

Service	Normal Priority Count	High Priority Count
A	10	2
B	3	3
C	1	12

If the Integration Agent is configured to send at most 18 APXML messages during a single exchange, then each Integration Service is given an allocation of six APXML messages. Additionally, each Integration Service's quota is equally divided between normal and high priority messages. Any unused allocation is then redistributed across the Integration Services that still have unselected outbound APXML messages.

In this example, the Integration Agent would make the following initial selections:

Example: First Allocation Phase

Service	Allocated	Selected Normal	Selected High	Unused Allocation
A	6	3	2	1
B	6	3	3	0
C	6	1	3	2

After initial selection, there are three unused allocations, so these are redistributed among the two services (A and C) that have remaining unselected APXML messages. To be fair, the unused allocation is rounded up so that all remaining services are given the same allocation.

In this example, the Integration Agent would make the following secondary selections:

Example: Second Allocation Phase

Service	Allocated	Selected Normal	Selected High	Unused Allocation
A	2	1	0	1
B	0	0	0	0
C	2	0	1	1

This allocation/re-allocation process repeats until the original allocation is exceeded or there are no more unselected outbound APXML messages.

In this example, the Integration Agent would make the following tertiary (and final) selections:

Example: Third Allocation Phase

Service	Allocated	Selected Normal	Selected High	Unused Allocation
A	1	1	0	0
B	0	0	0	0
C	1	0	1	0

Example: Final Queue Allocation Sizes

Service	Total Selected Normal	Total Selected High
A	5	2
B	3	3
C	1	5

As demonstrated in this example, the limit on the number of submitted APXML messages (18) is approximate since a total of 19 messages were selected. The reason for this discrepancy is the rounding up that occurs to maintain fairness in the allocation. The limit on the total number of APXML messages that are submitted during a single exchange should be chosen based on network bandwidth and the performance of xMatters with respect to processing the `SubmitAPXML` call. The intention is to submit APXML messages and have them processed in a timely fashion so that high priority outbound APXML messages that arrive after the message exchange begins will be handled as quickly as possible during the next exchange.

APXML messages are submitted to the xMatters web server that accepted the last Integration Agent heartbeat. If no such server exists, the message exchange process still occurs (i.e., log messages are produced), but there is no effect. If the heartbeat interval is considerably longer than the message exchange interval, and the ‘current’ web server becomes unavailable before a new heartbeat is sent, the message exchanger will log an error when it attempts to call the web server's SubmitAPXML web service method.

In either case (no current xMatters web server or a failed current web server), the selected outbound APXML messages are returned to the outbound queues. Additionally, because the outbound queues are persistent, even if the Integration Agent is abnormally shut down in the middle of an APXML exchange, the selected outbound APXML messages that were unsuccessfully sent are recovered when the Integration Agent restarts.

The only case in which an outbound APXML message will be discarded is if the xMatters Web Server to which the message was submitted returns an error indicating a problem processing that specific message. In this case, xMatters does not process any of the other submitted messages that were selected after the problematic message.

The Integration Agent returns the unprocessed messages, including the problematic message, to their respected outbound queues and resubmits them during the next message exchange. If xMatters continues to reject the problematic message after three reattempted submissions, the Integration Agent will log the problematic message and discard it. This allows the subsequent outbound APXML messages to be submitted and (potentially) successfully processed by xMatters.

The next section describes how the Integration Agent processes APXML messages that it receives from xMatters during the message exchange process.

Response Action Scripting

For each APXML message that the Integration Agent submits to xMatters during the message exchange process, xMatters returns a corresponding APXML message. If the submitted APXML message was successfully processed, xMatters returns an OK APXML message, and if the submitted APXML message could not be processed, xMatters returns an ERROR APXML message. The response message may also contain additional information related to the request. AlarmPoint's `SubmitAPXML` web service method accepts and returns the following APXML message types:

APXML Message Types

Method	Submitted
Add	Event

Method	Submitted
Delete	Event
Delete	Incident
Response	N/A

Note: *The method and subclass values are case-insensitive.*

xMatters responds to an Add/Event APXML message by returning a modified version of the submitted message, which includes the ID of the resulting xMatters event, as shown in the following example:

Submission

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="99">
  <header>
    <method>add</method>
    <subclass>event</subclass>
  </header>
  <data>
    <recipients type="string">bsmith</recipients>
    <situation type="string">Server down.</situation>
    <device type="string">localhost</device>
    <incident_id type="string">TICKET-0100-0302</incident_id>
    <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
    <apia_source type="string">apclient: /127.0.0.1:4063</apia_source>
    <apia_priority type="string">normal</apia_priority>
    <apia_process_group type="string">apia_default_group</apia_process_group>
    <company_name>Default Company</company_name>
  </data>
</transaction>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="99">
  <header>
    <method>OK</method>
  </header>
  <data>
    <recipients>bsmith</recipients>
    <situation>Server down.</situation>
    <device>localhost</device>
    <incident_id>TICKET-0100-0302</incident_id>
    <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
    <apia_source>apclient: /127.0.0.1:4063</apia_source>
    <apia_priority>normal</apia_priority>
    <apia_process_group>apia_default_group</apia_process_group>
    <company_name>Default Company</company_name>
    <apm_transmit_checksum>11aeab07f69fbde8e49da71dfbe</apm_transmit_checksum>
    <event_id>200019</event_id>
  </data>
</transaction>
```

Note: *In this example, AlarmPoint's APXML processor has returned auto-typed tokens instead of the explicitly typed tokens that were submitted. Additionally, an* `apm_transmit_checksum` *is added, but this is for internal use only.*

xMatters responds to a Delete/Event APXML message by returning a modified version of the submitted message without any additional information, as shown in the following example:

Submission

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="100">
  <header>
    <method>delete</method>
    <subclass>event</subclass>
  </header>
  <data>
    <event_id>200019</event_id>
    <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
    <apia_source type="string">apclient: /127.0.0.1:4617</apia_source>
    <apia_priority type="string">normal</apia_priority>
    <apia_process_group type="string">apia_default_group</apia_process_group>
    <company_name>Default Company</company_name>
  </data>
</transaction>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="100">
  <header>
    <method>OK</method>
  </header>
  <data>
    <event_id>200019</event_id>
    <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
    <apia_source type="string">apclient: /127.0.0.1:4617</apia_source>
    <apia_priority type="string">normal</apia_priority>
    <apia_process_group>apia_default_group</apia_process_group>
    <company_name>Default Company</company_name>
  </data>
</transaction>
```

Note: *If the event does not exist, xMatters still returns an OK APXML message.*

xMatters responds to a Delete/Incident APXML message by returning a modified version of the submitted message without any additional information, as shown in the following example:

Submission

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="101">
  <header>
    <method>delete</method>
    <subclass>event</subclass>
  </header>
  <data>
    <incident_id>TICKET-0100-0302</incident_id>
    <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
    <apia_source type="string">apclient: /127.0.0.1:4617</apia_source>
    <apia_priority type="string">normal</apia_priority>
    <apia_process_group type="string">apia_default_group</apia_process_group>
    <company_name>Default Company</company_name>
  </data>
</transaction>
```

Response

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="101">
  <header>
    <method>OK</method>
  </header>
  <data>
    <incident_id>TICKET-0100-0302</incident_id>
    <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
```

```
      <apia_source type="string">apclient: /127.0.0.1:4617</apia_source>
      <apia_priority type="string">normal</apia_priority>
      <apia_process_group>apia_default_group</apia_process_group>
      <company_name>Default Company</company_name>
    </data>
```

Note: *If the incident does not exist, xMatters still returns an OK APXML message.*

Response APXML messages that are submitted by the Integration Agent are an exception to the general rule of xMatters responding with an OK APXML message. For Response APXML messages, if xMatters successfully processes the response, it does *not* return an APXML message. Response APXML messages are covered in more detail in the next section, which describes the `ExternalServiceRequest2` processing mechanism.

If xMatters cannot process a submitted APXML message, it responds with a modified version of the submitted message, which includes additional tokens that describe the reason for the failure. For example, xMatters returns the following ERROR APXML message if an attempt is made to submit an APXML message with an unrecognized method/subclass:

Submission

```
  <?xml version="1.0" encoding="UTF-8"?>
  <transaction id="102">
    <header>
      <method>erase</method>
      <subclass>event</subclass>
    </header>
    <data>
      <event_id>200019</event_id>
      <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
      <agent_client_id>ping|ping</agent_client_id>
      <apia_source type="string">apclient: /127.0.0.1:1467</apia_source>
      <apia_priority type="string">normal</apia_priority>
      <apia_process_group type="string">apia_default_group</apia_process_group>
      <company_name>Default Company</company_name>
    </data>
  </transaction>
```

Response

```
  <?xml version="1.0" encoding="UTF-8"?>
  <transaction id="100">
    <header>
      <method>ERROR</method>
    </header>
    <data>
      <errorcode>HUB_POST</errorcode>
      <errordetail>&lt;message&gt;The message specified a method
                  UNKNOWN and subclass Event, which is unrecognized.&lt;
                  /message&gt;</errordetail>
      <errortext>Could not process message.</errortext>
      <incoming_message><![CDATA[<?xml version="1.0" encoding="UTF-8"?><transaction
                  id="102"><header><method>...</transaction>]]></incoming_message>
      <apm_transmit_checksum>11aead703f2b341a47cd0b90b42</apm_transmit_checksum>
      <event_id>200019</event_id>
      <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
      <agent_client_id>ping|ping</agent_client_id>
      <apia_source>apclient: /127.0.0.1:1467</apia_source>
      <apia_priority>normal</apia_priority>
      <apia_process_group>apia_default_group</apia_process_group>
      <company_name>Default Company</company_name>
    </data>
  </transaction>
```

As described in the previous section, if xMatters returns an ERROR APXML message, the problematic message and all subsequent messages are requeued and eventually resubmitted or discarded. If xMatters returns an OK APXML message, the Integration Agent adds this response message to the targeted Integration Service's inbound queue. However, before doing so, the Integration Agent resets the `apia_source` token to a string of the form "`alarmpoint: <url>`" where `<url>` is replaced with the URL of the xMatters web server that returned the response.

Note: *The priority and process groups of the submitted APXML message are carried over to the response APXML message. This is so that the Integration Service will process the response message with the same urgency and sequencing as the original (submitted) APXML message.*

Once the response APXML message is added to an Integration Service's inbound queue, it is processed in much the same way as APXML messages that are submitted via `APClient.bin`. The difference occurs when the Integration Service begins processing the response message from its inbound queue. In this case, the response message's `apia_source` token's value begins with `"alarmpoint"`, so the Integration Service processes the message via its Response Action Scripting (RAS) mechanism.

An Integration Service implements RAS via a method named "`apia_response`" in its JavaScript implementation. In every other way, `apia_response` is analogous to the `apia_input` method (for details, see "Integration Service Scripts" on page 104). An Integration Service can use RAS to update the integrated Management System (e.g., with the ID of the created event), make a remote web service call, or to submit additional APXML messages to xMatters.

The following section describes the way in which RAS applies to APXML messages that are *not* returned as responses by xMatters.

External Service Message/Request Processing

Response Action Scripting (RAS) applies to any APXML message that is sent by xMatters to an Integration Service. As described above, these messages may be responses to APXML messages submitted by an Integration Service to xMatters. Additionally, xMatters may send APXML messages to an Integration Service independently of any submitted messages. In particular, the Action Script objects `ExternalServiceMessage` (ESM) and `ExternalServiceRequest2` (ESR) may target an Integration Agent-hosted service (i.e., an Integration Service).

For example, the `ping` Script Package, which is included with xMatters, contains a handler Action Script with that performs the following operations when the recipient of a `ping` event returns an unrecognized response:

```
@event::report("Received unrecognized response: " & $reply, $notification.id)
@externalMessage = @event::createExternalServiceMessage()
$externalMessage.message = "Received unrecognized response from "
   & $response.recipient_target & ": " & $reply
@externalMessage::send()
```

In this example, the xMatters Application Node that is running the handler script prints a log message and then creates an ESM object. Since the ESM object is created by an event that originated from an Add/Event APXML message submitted by the ping Integration Service, the ESM's service provider is initialized with the source Integration Agent's ID and the ESM's service name is initialized with "`ping|ping`".

When `@externalMessage::send()` is called, xMatters checks the service name and since it has the form `<domain>|<name>`, xMatters knows that the message is targeting an Integration Service. xMatters converts the ESM into a Send APXML message whose tokens are the ESM's variables (e.g., message). xMatters then adds the Send APXML message to a database table that represents a queue of APXML messages that are to be sent to Integration Agents during the message exchange process. This queue is referred to as the APIA outbound queue and is different than the Integration Service outbound queues that were referred to in the “Outbound Queue Model” section.

When an Integration Agent calls the `ReceiveAPXML` Web Service method, xMatters returns any APIA outbound queue APXML messages whose `agent_application_id` matches the requesting Integration Agent's ID and whose `agent_client_id` specifies an active Integration Service that the Integration Agent is hosting. Additionally, if there are any APXML messages that have been waiting for an excessively long time (default is five minutes) in the APIA outbound queue and they are targeting an active Integration Service that the Integration Agent is hosting, xMatters may choose to return these messages to the requesting Integration Agent. This process of allowing old APIA outbound queue messages for potential processing by Integration Agents that were not originally targeted is both a fault tolerance mechanism (e.g., in case the originally targeted Integration Agent is shut down), and a resource usage limiting mechanism that prevents the APIA outbound queue from growing uncontrollably.

The total number of APXML messages that xMatters will return during a single `ReceiveAPXML` call is limited by the `<apxml-exchange>/<size>` value specified in the Integration Agent's configuration file. Since this limitation applies separately to the `SubmitAPXML` and `ReceiveAPXML` calls, the total number of APXML messages that are exchanged at one time may be twice the `<apxml-exchange>/<size>` value.

The following example shows the APXML message that the `ping` Integration Service would receive as a result of the previous example's `@externalMessage::send()` call:

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction>
  <header>
    <method>Send</method>
  </header>
  <data>
    <incident_id>TICKET-0100-0302</incident_id>
    <message>Received unrecognized response from bsmith|Work Email:SILLY</message>
    <agent_application_id>vic-schow/192.168.168.71:8081</agent_application_id>
    <originator>e7e923b1-8df7-4868-94f3-674083b84f27</originator>
    <agent_client_id>ping|ping</agent_client_id>
    <request_id>e7e923b1-8df7-4868-94f3-674083b84f27</request_id>
  </data>
</transaction>
```

When the Integration Agent receives an APXML message from xMatters via `ReceiveAPXML`, it adds an `<apia_priority>normal</apia_priority>` token to the message, unless the message already contains an `apia_priority` token with a recognized value (i.e., normal or high). That is, the Integration Agent respects the message's priority if one is set.

Additionally, the Integration Agent sets the `apia_source` token to a string of the form "`alarmpoint: <url>`" where `<url>` is replaced with the URL of the xMatters web server to which the `ReceiveAPXML` call was made. Finally, the APXML message is added to the targeted Integration Service's inbound queue and subsequently processed via RAS, as described in "Response Action Scripting" on page 110.

An ESR also generates an APXML message that is sent by xMatters to an Integration Service. For example, the ping Script Package's handler Action Script performs the following operations when a recipient of a ping event returns a "PING DEVICE" response:

```
# perform the external ping request
@serviceRequest = @event::createExternalServiceRequest2()
$serviceRequest.request_text = "pingdevice"
$serviceRequest.device = $event.device
$requestTimedOut = false
@serviceRequest::send()
$timeout = 10
UNTIL($serviceRequest.completed, $timeout)
```

As was the case with ESMs, the ESR object is initialized with the source Integration Agent's ID and the source Integration Service's domain and name. When `@serviceRequest::send()` is called, xMatters checks the service name and since it has the form `<domain>|<name>`, xMatters knows that the request is targeting an Integration Service. xMatters converts the ESR into a Request APXML message whose tokens are the ESR's variables (e.g., `request_text` and `device`).

Once the Request APXML is created, xMatters must deliver it to the targeted Integration Service, and it is as this point that ESM and ESR processing differ. Rather than immediately adding the Request APXML message to the APIA outbound queue, from where it would eventually be sent to an Integration Agent during a `ReceiveAPXML` call, xMatters first checks whether the targeted Integration Agent allows direct External Service Request calls. The External Service Request mode that an Integration Agent supports is configured by the `<external-service-request>/@mode` setting in the agent's configuration file, and this value is submitted by each Integration Agent whenever it sends a heartbeat to xMatters.

If the targeted Integration Agent allows direct External Service Requests, the xMatters Application Node immediately calls the targeted Integration Service's `SubmitAPXML` web service method, passing the Request APXML message as a parameter. When an Integration Service receives the `SubmitAPXML` call, it extracts the Request APXML message and processes it using the same RAS mechanism that described previously.

The only difference is that the Request APXML message bypasses the Integration Service's inbound queue (i.e., it is immediately passed to RAS), and the APXML message that is returned by the Integration Service's `apia_response` JavaScript method is returned as the response to the `SubmitAPXML` web service call (i.e., the response is immediately returned to AlarmPoint). xMatters processes any returned APXML message in the same fashion that it processes APXML messages submitted by the Integration Agent via the message exchange process.

Note: *During RAS processing of an ESR, any APXML messages that an Integration Service sends to xMatters via the Service API's* `sendAPXML(...)` *method will still be sent to xMatters via the message exchange process. To achieve the full benefit of direct External Service Request processing, the Integration Service's* `apia_response` *method should explicitly return an APXML message rather than using the* `sendAPXML(...)` *method.*

If the targeted Integration Agent does not allow direct External Service Requests, the xMatters Application Node adds the Request APXML message to the APIA outbound queue, and the message is delivered to the Integration Agent via the message exchange process. There is an important difference between the Send APXML messages that are generated by ESMs and the Request APXML messages that are generated by ESRs: while the priority of Send APXML messages defaults to "normal", the priority of Request APXML messages is always "high".

The following example shows the APXML message that the ping Integration Service would receive as a result of the previous example's `@externalReqeust::send()` call:

```
<?xml version="1.0" encoding="UTF-8"?><transaction>
  <header>
    <method>Request</method>
  </header>
  <data>
    <request_text>pingdevice</request_text>
    <device>localhost</device>
    <incident_id>TICKET-0100-0302</incident_id>
    <originator>ec59ee13-dc3f-4ecf-99b1-00923f762ffd</originator>
    <request_id>ec59ee13-dc3f-4ecf-99b1-00923f762ffd</request_id>
    <completed>false</completed>
    <successful>false</successful>
    <agent_application_id>vic-schow/192.168.168.71:8081
    </agent_application_id>
    <agent_client_id>ping|ping</agent_client_id>
    <apia_priority>high</apia_priority>
  </data>
</transaction>
```

Note: *The only difference between a Request APXML message that is sent as a direct External Service Request call and a Request APXML message that is sent as an indirect External Request call is the presence of the* `apia_priority` *token. Request APXML messages that are sent directly do not have an* `apia_priority` *token since the are not queued.*

To provide fault tolerance, if the targeted Integration Agent allows direct External Service Request calls, but xMatters is unable to successfully call the targeted Integration Service's `SubmitAPXML` method (e.g., due to a network failure or application error), xMatters will subsequently try to send the Request APXML message to any Integration Agent that actively hosts the targeted Integration Service.

These failover requests are sequentially sent to each eligible Integration Agent (in random order), using the agent's configured External Service Request mode until either a direct `SubmitAPXML` succeeds or the Request APXML message is added to the APIA outbound queue. If all attempts fail, the ESR's `completed` variable is set to `true` and its `successful` variable is set to `false` so that the Action Script calling `@externalMessage::send()` will be notified of the failure.

The benefit of direct External Service Request processing is that the Action Script initiating the request will receive a response as quickly as possible since there is no delay in waiting for the message exchange process to occur or the targeted Integration Service's inbound and outbound queues to be processed. However, xMatters deployments that use direct External Service Request processing must have firewall settings that allow connections to be initiated from xMatters Application Nodes to Integration Agents, and from Integration Agents to xMatters web servers. For deployments that use indirect External Service Request, firewall settings need only allow connections to be initiated from Integration Agents to xMatters web servers.

Legacy Integration Services

The Integration Agent provides connectivity between management systems and xMatters using the mechanisms of the APClient Gateway, Input Action Scripting, and Response Action Scripting, . The Java Client provides similar functionality, but without the firewall-friendly communication and JavaScript support that the Integration Agent provides. In order to leverage past investments, the Integration Agent can host an Integration Services whose implementation is based on an existing Java Client integration.

For example, the JavaClient contains a sample ping integration whose definition is stored in the file `<JavaClient_Dir>/etc/integrations/ping.xml`. The following XML is an excerpt from the JavaClient's ping integration:

```
<?xml version="1.0" encoding="UTF-8"?>
<alarmpoint-agent-client id="ping">
  <mapped-input method="add" subclass="action">
    <parameter index="1" type="string">person_or_group_id</parameter>
    <parameter index="2" type="string">situation</parameter>
    <parameter index="3" type="string">device</parameter>
    <parameter index="4" type="string">incident_id</parameter>
    <parameter index="5" type="string">company</parameter>
  </mapped-input>
  <response-filter type="ok-error-request" />
  <response-action name="RequestAction" method="request" consume="true">
    <script language="java">
    <![CDATA[

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.InputStream;

... snip

// ////////////////////////////////////////////////////////////////////
//
// The main routine for handling PING starts here.
//
// ////////////////////////////////////////////////////////////////////

if( (APDT_request_text != void) &&
  (APDT_request_text.equalsIgnoreCase( "PingDevice" )) )
{
  // Make sure we know all the information about the initial request
  // in order to create a proper response.
  //
  if( (APDT_device != void) && (APDT_request_id != void) &&
      (APDT_originator != void) )
  {
    // Presume nothing works.
    //
    String result = "Ping Failed";

    try

... snip

  // The request text was blank; let the log know.
  //
  AGENT_LOG_MESSAGE = "Agent Client: PingDevice - " +
    "Unknown request_text = '" + APDT_request_text + "'";
}
]]>
```

```
      </script>
    </response-action>
    <input-action name="Add" method="Add">
      <module type="filter">
        <configuration>
          <filters />
        </configuration>
      </module>
    </input-action>
    <input-action name="Add Action" method="Add" subclass="Action">
      <module type="filter">
        <configuration>
          <filters />
        </configuration>
      </module>
    </input-action>
  </alarmpoint-agent-client>
```

An Integration Service can use the javaclient element of its configuration file to refer to an existing Java Client integration, as shown in the following example:

```
  <javaclient>
    <file>/apagent/etc/integrations/ping.xml</file>
  </javaclient>
```

In this example, the sample `ping` Integration Service has been changed so that its implementation is based on the JavaClient's `ping` integration. When the APClient Gateway receives map data submissions for the `ping` Integration Service, it will apply the Java Client `ping`'s mapped-input to create a resulting APXML message. The `ping` Integration Service's own mapped-input element is ignored. Similarly, since the Java Client ping's mapped-input element is also used to specify constants, the ping Integration Service's own constants element is also ignored. Even though the ping Integration Service is based on the Java Client's implementation, all submitted APXML messages are still processed according the inbound queue model described in "Inbound Queue Model" on page 103.

Instead of applying the JavaScript-based Input Action Scripting to submitted APXML messages, the `ping` Integration Service will use the Java Client's BeanShell-based Input Action Scripting. Java Client integration log messages, which are normally specified via the `AGENT_LOG_MESSAGE` variable, are output by the Integration Agent as INFO level messages using the relevant Integration Service's logging category. Similarly, Java Client APXML response messages, which are normally specified via the `APXML_RESPONSE` variable and have type `com.invoqsystems.apcom.APMessage`, are converted by the Integration Agent into equivalent `com.alarmpoint.integrationagent.apxml.APXMLMessage` objects and stored in the relevant Integration Service's outbound queue for eventual delivery to xMatters.

Similarly, instead of applying the JavaScript-based Response Action Scripting to APXML message received from xMatters, the `ping` Integration Service will use the JavaClient's BeanShell-based Response Action Scripting. Logging and response APXML messages are handled in the same way as BeanShell-based Response Action Scripting, except that, in the case of direct External Service Requests, the response APXML message will be returned directly to the xMatters Application Node (in the same manner as JavaScript-based Response Action Scripting; refer to the *xMatters AlarmPoint Java Client Guide* for more information on how to configure Java Client integrations).

Note: *The Integration Agent does not support the response-filter, module, or consume features features that are part of xMatters Java Client integrations.*

Summary and Security Issues

This section references a diagram that summarizes the steps that occur within the Integration Agent when it processes APClient submissions. This diagram can be accessed via the xMatters Community support site at:

```
  http://support.xmatters.com/docs/DOC-3070
```

Note: *If the above link does not work, search the Community site for "Integration Agent flowchart".*

On the flowchart, notice that requests to legacy and non-legacy Integration Services are handled in a similar fashion, with the only differences being in the conversion of map data submissions to APXML and the targets used for Input and Response Action Scripting. In particular, both legacy and non-legacy Integration Services are scrutinized by the same security framework.

APClient requests are authenticated in the same way as Integration Service/mobile access component Requests (see "Security" on page 74). To prevent unauthorized APClient submissions, the Integration Agent can check that the submission comes from a known IP address (via the `ip-authentication` ACL) and/or contains a recognized password (via the `password-authentication` configuration). For further details on these settings see "Integration Agent Configuration File" on page 18.

Since the Java Client has no analog to the Integration Agent's security framework, legacy integrations have no configuration for IP addresses or access passwords. By its nature, IP authentication requires no explicit changes to legacy integrations; if `ip-authentication` is enabled, the Integration Agent administrator need only ensure that the legacy integration clients are included in the IP address ACL.

If the Integration Agent requires an access password, then legacy integrations must either include the password with their submissions (via the `apia_password` APXML token), or the legacy Integration Service must be configured to add the password as a constant. The former requires changing legacy integration clients, which may be infeasible, while the later requires no changes to client code and is recommended given the following caveat:

If a legacy Integration Service is configured with an `apia_password` constant, then password authentication for that specific Integration Service is essentially circumvented. Any APClient will be able to submit requests to the integration unless IP authentication is enabled. However, this circumvention applies only to Integration Services that explicitly add an `apia_password` constant; other Integration Services will remain secure.

The following example demonstrates how the legacy ping Integration Service, when hosted in an Integration Agent that requires password authentication, can work seamlessly without client changes. The example depends on the following Integration Agent installation:

Files for Legacy ping Integration Service Example

File Path	Description
`<IAHOME>/conf/IAConfig.xml`	Configuration file
`<IAHOME>/conf/.passwd`	Encrypted Integration Agent access password
`<IAHOME>/integrationservices/ping/ping.xml`	Ping Integration Service configuration file
`<IAHOME>/integrationservices/ping/legacy/ping.xml`	Legacy ping Integration Service configuration file from Java Client

Assume the Integration Agent has the following setting in its `IAConfig.xml` file:

```
<password-authentication enable="true">
  <password>

    <file>.passwd</file>
  </password>
</password-authentication>
```

This setting indicates that APClient submissions must contain a recognized `apia_password` token. Without any changes, the following request will be rejected by the Integration Agent because it is missing the password:

```
APClient.bin --map-data ping bsmith "Server down." localhost
      TICKET-0100-0302
```

The legacy `ping` Integration Service configuration file has the following `mapped-input` element:

```
    <mapped-input method="add" subclass="action">
      <parameter index="1" type="string">action_script_set</parameter>
      <parameter index="2" type="string">person_or_group_id</parameter>
      <parameter index="3" type="string">situation</parameter>
      <parameter index="4" type="string">device</parameter>
      <parameter index="5" type="string">incident_id</parameter>
    </mapped-input>
```

Suppose an `apia_password` parameter is added to the `mapped-input` element:

```
    <mapped-input method="add" subclass="action">
      <parameter index="1" type="string">action_script_set</parameter>
      <parameter index="2" type="string">person_or_group_id</parameter>
      <parameter index="3" type="string">situation</parameter>
      <parameter index="4" type="string">device</parameter>
      <parameter index="5" type="string">incident_id</parameter>
      <parameter index="6" type="string">apia_password</parameter>
    </mapped-input>
```

If the Integration Agent's access password is "my secret", then the following request will be accepted by the Integration Agent:

```
    APClient.bin --map-data ping bsmith "Server down." localhost TICKET-0100-0302 "my secret"
```

As previously mentioned, changing the client requests may not be feasible. The alternative is to note that, as shown in Integration Agent Submission flowchart, constants are applied before authentication. As a result, the Integration Agent's access password can be automatically added to each Integration Service request, regardless of whether it is included explicitly by the client. For the `ping` Integration Service example, the following `constants` element could be added to the `ping` Integration Service's configuration file (the non-legacy configuration file):

```
  <constants>
    <constant name="apia_password" overwrite="false">my secret</constant>
  </constants>
```

In this case, the Integration Agent's access password is stored in cleartext, which may be undesirable from a security audit standpoint.

Alternatively, the ping Integration Service's `apia_password` constant can refer to an encrypted file:

```
  <constants>
    <encrypted-constant name="apia_password" overwrite="false">
      <file>.passwd</file>
    </encrypted-constant>
  </constants>
```

Specifically, this encrypted-constant refers to the same encrypted file that the Integration Agent uses for its password authentication.

As a result, if the Integration Agent's access password is changed, the `ping` Integration Service needs only be reloaded or the Integration Agent restarted for the new access password to be added to all APClient requests sent to the `ping` Integration Service.

Deleting Events

The default Integration Agent installation includes an autoloaded `del` Integration Service:

```
  <IAHOME>/integrationservices/autoloaded/del.xml
```

The `del` Integration Service is intended for use by Management Systems making APClient requests via `APClient.bin` or HTTP POST. Consequently, the `del` Integration Service does not need to be configured in the xMatters web server as is required for xMatters mobile access Integration Services.

Note: *The "Del" integration service exists only as a legacy feature of the Integration Agent, and should not be used unless there is a compelling reason. "Delete" requests that are submitted via this integration service are not reliably delivered to xMatters in the correct sequence with respect to any "Add" requests. For this reason, integration services should use the "sendDelAPXML" function, implemented in the* `xmattersws.js` *script in the Integration Agent Utilities bundle. For an example, see the processRequest function in the BMC Remedy integration (version 4.1.)*

Management Systems can use map data submissions to the `del` Integration Service to delete events by either `incident_id` or `event_id`. The first map data parameter (i.e., the parameter immediately following the `del` Integration Service's name) corresponds to `incident_id` and the second map data parameter corresponds to `event_id`. Only one of these parameters needs to be set.

For example, to delete all events with `incident_id=TICKET-1234`, a Management System can make the following submission:

```
APClient.bin --map-data del TICKET-1234
```

In response, the Integration Agent will send the following Del/Action APXML message to AlarmPoint:

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="1">
  <header>
    <method>del</method>
    <subclass>action</subclass>
  </header>
  <data>
    <incident_id type="string">TICKET-1234</incident_id>
    <company_name>Default Company</company_name>
    <agent_client_id>del|del</agent_client_id>
  </data>
</transaction>
```

Since the second map data parameter is omitted, no `event_id` token is included in the Del/Action APXML message.

To delete the event with `event_id=200302`, a Management System can make the following submission:

```
APClient.bin --map-data del "" 200302
```

In response, the Integration Agent will send the following Del/Action APXML message to AlarmPoint:

```
<?xml version="1.0" encoding="UTF-8"?>
<transaction id="1">
  <header>
    <method>del</method>
    <subclass>action</subclass>
  </header>
  <data>
    <event_id type="string">200304</event_id>
    <company_name>Default Company</company_name>
    <agent_client_id>del|del</agent_client_id>
  </data>
</transaction>
```

Note: *Since the first map data parameter is* `""`*, no* `incident_id token` *is included in the Del/Action APXML message.*

These example map data submissions assume that there is no other Integration Service in the `del` Event Domain. If this is not the case, the `APClient.bin` submissions must use the `del` Integration Service's fully-qualified name; for example:

```
APClient.bin --map-data "del|del" TICKET-1234
```

Note: *Windows and some Unix shells interpret the pipe character (|) as a command. To prevent this, the Integration Service name* `del|del` *may need to be enclosed in quotes or have the pipe character escaped (depending on the operating system).*

Errors and Retries While Processing Inbound Queue

APXML messages enter an Integration Service's inbound queue through the following submission mechanisms:

- APClient Gateway submissions (via direct HTTP POST/GET or APClient.bin, or Auto Recovery of APClient.bin)
- Service-to-Service submissions (via the ServiceAPI.sendAPXML method)
- xMatters SubmitAPXML responses (OK and ERROR)
- xMatters External Service Messages
- xMatters Indirect External Service Requests

From the inbound queue, each message is processed by the Integration Service's Input or Response Action Scripting (IAS or RAS), depending on the submission mechanism. APClient Gateway and Service-to-Service submissions are processing through IAS, while all other submissions are processed through RAS.

The processing of APXML messages from an inbound queue is transactionalized: each message is temporarily removed from the queue, processed, and then either permanently removed or restored depending on the outcome of processing. If the Integration Agent is suddenly stopped while processing messages, the in-process messages will not be lost, and their processing will restart when the Integration Agent resumes operation.

Note: *Messages that were in-process when the Integration Agent stopped are processed anew when the integration restarts. As a result, scripts must be written to handle partial updates (e.g., to management state) left by restarted messages.*

When the processing of an APXML message completes without any errors or interruptions, the message is permanently removed from the inbound queue. Depending on the nature of the exception and the Integration Service's implementation, several things can occur if the script throws an exception, as discussed below.

A script can be written to indicate that an error is transient and that processing of the message should be retried after a configurable delay by throwing an instance or subclass of com.alarmpoint.integrationagent.exceptions.retriable.RetriableException. The simplest way to create a RetriableException is to use the following constructor:

```
public RetriableException(Throwable cause, int maxAttempts, long delayMillis);
```

Where:

- `cause` is the transient exception caught by the script (e.g., database connection unavailable).
- `maxAttempts` is the maximum number of times that the message can be (re)processed before being discarded.
- `delayMillis` is the minimum time (in milliseconds) that the message must wait before being re-processed.

For example, the following input action script allows a message to be processed up to 3 times when it encounters a (presumably transient) database exception:

```
importClass(Packages.com.alarmpoint.integrationagent.exceptions.retriable.
 RetriableException)

function apia_input(apxml) {
  try {
    update_db(apxml);
  } catch (ex) {
    throw new RetriableException(ex, 3, 5000);
  }
```

Note: *The classes and methods that support retriable exceptions are described in detail in the JavaDoc located at <ia_home>/docs/retriable_exceptions/javadoc. Integrators who plan to use retriable exceptions should consult the JavaDoc for additional details.*

When a script throws a RetriableException and the processing of the message has been attempted fewer than `maxAttempts` times, a warning will be logged and the message will be restored to its original position in the inbound queue. However, the message will not be re-processed until the delay has elapsed. Since the Integration Agent guarantees first-in-first-out ordering of messages in the same process group, this also means that the processing of subsequent messages in the same process group will be delayed until the problematic message is permanently removed from the inbound queue (either because processing eventually succeeded, processing resulted in a non-retriable exception, or processing resulted in too many retriable exceptions). Messages that are in other process groups are not affected by the delay.

Note: *The attempt-to-process count and retry delay for a particular message is reset whenever the Integration Agent restarts. For example, if two attempts are made to process a message and the Integration Agent restarts while waiting for the third attempt, upon restart the message's attempt-to-process count will be reset to 1 and there will be no delay before the first (re)attempt to process the message.*

If a script throws a non-retriable exception or a RetriableException whose retry policy is no longer valid (i.e., the attempt-to-process count is greater than or equal to `maxAttempts`), an attempt is made to call a method named `apia_discard` in the Integration Service's JavaScript implementation.

The `apia_discard` method

The `apia_discard` method is a way for Integration Services to implement their own error handling logic before a message is permanently removed from the inbound queue due to an error. `apia_discard` implementations may log the error, create new xMatters events to signal the error, or annotate Management System logs. APXML messages returned by this method are forwarded to xMatters in the same fasion as `apia_input` and `apia_response`.

Inputs

The `apia_discard` method accepts the following parameters:

```
function apia_discard(apxml, ex, numAttempts, firstAttemptTimestamp);
```

Where:

- `apxml` is of type `com.alarmpoint.integrationagent.apxml.APXMLMessage` and is a copy of the message that was initially passed to `apia_input/apia_response` (i.e., any changes made to `apxml` by these methods are lost when `apxml` is sent to `apia_discard`).
- `ex` is an instance or subclass of `java.lang.Throwable`, and is the last exception thrown by `apia_input/apia_response` prior to calling `apia_discard`.
- `numAttempts` is the current attempt-to-process count for `apxml`.

Note: *In some circumstances, most notably when the Integration Agent is restarted between the last exception thrown by* `apia_input/apia_response` *and the subsequent call to* `apia_discard`*, the full context of the exception, including the number of attempts and first attempt timestamps, may be lost. In this case, a* `java.lang.RuntimeException` *with a message regarding the loss of exception context, will be passed to* `apia_discard` *along with a* `numAttempts` *count of 0 and an approximate* `firstAttemptTimestamp`.

- `firstAttemptTimestamp` is the time (approximate, in milliseconds from epoch) of the first attempt to process `apxml`. Like `numAttempts`, this is reset when the Integration Agent restarts.

Outputs

Implementations of `apia_discard` can access the ServiceAPI and return APXML message results that can then be injected into xMatters. In short, there is no difference in the scope of available operations between `apia_discard` and `apia_input/apia_response`.

If an Integration Service determines that an error is not important, its `apia_discard` method may return without throwing an exception. If this is the case, the message is permanently removed as if the original call to `apia_input/apia_response` were successful.

An Integration Service may also throw an exception from its `apia_discard` method. This exception may be the same exception passed to `apia_discard` or a new exception (either intentional or unintential). In either case, the exception is logged as an error and the message is permanently removed.

Finally, if an Integration Service does not implement an `apia_discard` method, either because it is a Legacy Integration Service or its JavaScript implementation has no `apia_discard` method, a warning is logged and the message is permanently removed.

Note: *The message is always removed from the inbound queue, regardless of whether* `apia_discard` *processing is not implemented or succeeds/fails. The only circumstance under which a message will be reprocessed by* `apia_discard` *is if the Integration Agent is restarted mid-process.*

Timeouts

A timeout will occur if the input or response action scripting request takes longer than the <request-timeout> parameter configured in the `IAConfig.xml` file. After this period has elapsed, the integration agent cancels the execution of the request, and will retry the request two more times, with a ten-second delay between attempts.

If the request cannot be completed after a total of three attempts, a WARN message is logged indicating that the request was terminated due to the <request-timeout> parameter being exceeded. If the apia_discard method is present, it is called and the integration can decide whether to forward the APXML to xMatters by returning it in the method:

```
function apia_discard(apxml, ex, numAttempts, firstAttemptTimestamp)
{
  ServiceAPI.getLogger().warn("apia_discard called...");
  return apxml;
}
```

If the apia_discard method is not present, an error message is logged indicating that the APXML could not be processed.

Lifecycle hooks

The Integration Agent lifecycle hooks allow scripted responses to startup, shutdown, suspend, interrupt and resume events. When one of these events is requested (eg, when the Windows service receives a shutdown command,) the appropriate JavaScript function is invoked by the Integration Agent.

The following methods are available:

- `apia_startup()`: triggered when the Integration Service is first started, typically during startup of the Integration Agent.
- `apia_shutdown()`: triggered when the Integration Agent receives a shutdown request. Note that apia_shutdown() is called regardless of when or if the apia_startup() is called.
- `apia_suspend()`: triggered when the Integration Service is suspended using the "iadmin suspend" command.
- `apia_resume()`: triggered when the Integration Service is resumed using the "iadmin resume" command.
- `apia_interrupt()`: triggered when the Integration Agent is forcibly shut down or the "iadmin suspend-now" command is issued.

Note that all of these methods are optional.

Implementation

The following is an example Javascript method illustrating the apia_startup method:

```
function apia_startup()
  {
    log.debug("The Integration Service Startup Hook has been triggered.");
    return true;
  }
```

For more examples, consult one of the xMatters integrations, such as the xMatters (IT) engine for HP NNM i-series integration.

Appendix: Error and exit codes

For ease of reference, this appendix comprises the various error and exit codes you can use to monitor your Integration Agents. These codes are also included in their associated sections elsewhere in this document.

Startup issues

The following table summarizes startup exit codes:

Startup Exit Codes

Startup Exit Code	Description
0	Success
30	Service already started
35	Service not installed
40	Service not stopped
45	Service failed to start, but did not have a SERVICE_EXIT_CODE
50	get-status failure (check logs)
60	Missing configuration file
61	Unreadable configuration file (i.e., file is locked)
65	Malformed configuration file
70	Missing or unreadable log4j properties file
80	Unable to bind to Admin Gateway port
85	Unable to bind to Web Services Gateway port
86	Malformed Mule configuration file

Startup Exit Code	Description
87	Mule startup error
88	Mule timeout error
90	Nonspecific startup error (check logs)

APXML and APClient.bin

The following table summarizes the OS exit codes that `APClient.bin` returns to the caller:

OS Exit Codes APClient.bin Returns to Caller

Exit Code	Description
0	The Integration Agent accepted the connection request and HTTP POST data. Note that this does not mean that the submission succeeded; the Integration Agent may still have responded with an Agent/ERROR APXML message.
10	Socket communications error (could not establish a connection).
20	Error parsing the command-line parameters.
30	Not enough available RAM to continue operation.
40	File-related error (e.g., a file could not be opened).
50	Error with the request.
250	Unexpected error.

The following table summarizes the error codes returned when an APXML submission fails:

Error Codes Returned on APXML Submission Failure

Error Code	Description
INVALID_SUBMISSION	Unrecognized or missing URL parameters.
INVALID_MESSAGE	A message submission with malformed APXML or unrecognized Integration Service target (i.e., agent_client_id).
DATAMAP_FAILED	An unrecognized Integration Service target (i.e., the first token), or an Integration Service target without a data map.
AGENT_ERROR	Any other error during processing, including denial of service (e.g., invalid access password).

Heartbeat failures

Consult the Integration Agent's log file and locate the ERROR entry associated with the heartbeat attempt. The ERROR message will either indicate a connectivity failure or provide one of the following reasons for the heartbeat's rejection:

- **UNKNOWN_DOMAIN**: indicates that one of the Integration Service Event Domains has not been configured on the xMatters web server
- **UNKNOWN_SERVICE**: indicates that one of the Integration Service names has not been configured within the Event Domain on the xMatters web server
- **REGISTRATION_ACL_FAILED**: indicates that this Integration Agent's ID has not been configured on the xMatters web server
- **UNKNOWN_APPLICATION_ERROR**: indicates that an unexpected error occurred
- **SERVICE_DENIED**: Depending on the error message, the web services user account in xMatters does not have the "Receive APXML" and "Send APXML" permissions (error message contains reference to "ReceiveAPXML") or does not have the "Register Integration Agent" permission (error message references a rejected heartbeat/registration).

www.xmatters.com

Online Support: http://support.xmatters.com

International: **+1 925.226.0300** and press **2**

US/CAN Toll Free: **+1 877.XMATTRS (962.8877)**

EMEA: **+44 (0) 20 3427 6333**

Australia/APJ Support: **+61-2-8038-5048** opt **2**

xMatters enables any business process or application to trigger two-way communications (voice, email, SMS, etc.) throughout the extended enterprise. The company's cloud-based solution allows for enterprise-grade scaling and delivery during time-sensitive events. More than 1,000 leading global firms use xMatters to ensure business operations run smoothly and effectively during incidents such as IT failures, product recalls, natural disasters, dynamic staffing, service outages, medical emergencies and supply-chain disruptions.