xMatters Integration Agent Guide

version 5.2

(x) matters

Table of Contents

The Integration Agent

The xMatters Integration Agent facilitates closed-loop integrations with applications installed behind a corporate firewall. It provides two core functions for applications behind a corporate firewall:

- An interface for applications to take action in xMatters.
- An interface to update applications when actions take place in xMatters.

It allows you to enrich events, making it a powerful tool in the integrator's tool box. 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 apps, 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.

For more information about the deprecated premises version of the Integration Agent, see the xMatters Integration Agent 5.1.8.2 Release Notes.

Get up and running with the Integration Agent

Follow along as we walk you through how to quickly set up the Integration Agent and send a test notification to an xMatters user, using our sample communication plan. The only thing you need is a working xMatters instance with a notifiable user (in other words, a user with a working device).

These instructions are tailored for the sample plan. However, you can use them as a guideline to configure any integration based on a communication plan.

Configure your xMatters instance

First, there are a couple of things you need to do in xMatters: create a web service user and an integration user.

Create a web service user

This is a user with specific permissions in xMatters to receive user responses and notifications about event status changes for the Integration Agent. ("Web service users" are a specific type of user in xMatters that can work with the SOAP web services used by the Integration Agent; this means you can't just use a regular user and assign them a specific role or permission – you have to create a dedicated user.)

How to create a web services user

1. Log in to xMatters as a Company Administrator.
2. Click the **Users** tab, and then click **Add Web Service User** in the Web Service Users menu.
3. Give your web service user a user ID (the default user ID used by the Integration Agent is "ia_user") and password.
    - Make note of the user ID and password – you need them to configure the Integration Agent. In our examples below we'll use "ia_user" as the username and "password" as the password.

4. Give the user access to the following web services:
    - Register Integration Agent
    - Receive APXML
    - Submit APXML

Create an integration user

Integrations using a communication plan require a user who can authenticate REST web service calls when injecting events.

This user needs to be able to work with events, but doesn't need to update administrative settings. While you can use the default Company Supervisor role to authenticate REST web service calls, the best method is to create a user specifically for this integration and assign that user to the "REST Web Service User" role, which includes the necessary permissions and capabilities.

This is the account that is used to send notification requests to xMatters – we'll refer to this user as the "integration user" to distinguish it from the web services user.

How to create an integration user

1. Log in to the target xMatters system.
2. On the **Users** tab, click **Add**.
3. Enter the appropriate information for your new user. Because this user affects how messages appear for recipients and how events are displayed in the reports and Communication Center, you may want to identify the user as specific to this integration. For example, if you're integrating with the sample-plan, you might set up the user as follows:
    - First Name: Sample
    - Last Name: Integration
    - User ID: sampleplan (we'll use this in our examples below)
4. Assign the user to the REST Web Service User role and set their password (we'll use "password" in our examples below).
    - Depending on your deployment, you might need to add one of the following roles so you can log in as the integration user and access the Developer tab: Full Access User or Developer.
5. Make a note of these details; you'll need them when configuring other parts of this integration.
6. Click **Save**.

Import and configure the sample communication plan into xMatters

Next, you need to import that sample plan into xMatters, do some basic configuration and copy the URL for the inbound integration.

How to import the sample plan

1. In the extracted Integration Agent archive, locate the `<IAHOME>\integrationservices\applications\sample-plan\SamplePlan.zip` file and copy it to your local workstation.
2. Log in to your xMatters system, click the **Developer** tab, and then click **Import Plan**.
3. Click **Choose File**, and then locate the SamplePlan.zip file.
4. Click **Import Plan**.
    - Once the import is finished, the plan should be automatically enabled. If it isn't, click **Plan Disabled** to enable it.
5. Click the **Edit** drop-down list for the plan, and select **Access Permissions**.
6. Add the integration user you created above, and then click **Save Changes**.

7. In the **Edit** drop-down list, select **Forms**.
    - The forms should be automatically deployed. If they aren't, click **Not Deployed** beside the form and select **Enable for Web Service**.
8. For the **Confirmation Message** form, in the drop-down list, click the **Web Service** drop-down list then click **Sender Permissions**.
9. Add the integration user, and then click **Save Changes**.
10. Go to the **Integration Builder** tab and expand the list of inbound integrations.
11. Click the name of the integration to view its details, then select **Basic** in the **Select authentication method** drop-down list.
12. Click **Update Inbound Integration**, then scroll down to the bottom of the page, and click **Copy** beside the URL field.
13. Paste the URL into a file — you'll need it later to configure the Integration Agent.

You can now install and configure the Integration Agent then set up the sample plan to send a test message.

Install and configure the Integration Agent

If you haven't already, download the latest version of the Integration Agent and the Integration Agent Utilities (IA_Util) from the product page. Make sure you're getting the right file for your operating system!

Once you have those files, extract that package to a suitable server or workstation to install the Integration Agent, then you can configure and, finally, start the Integration Agent

Install the Integration Agent and IAUtils

1. The first step is to extract the Integration Agent archive (unzip or untar) onto your server or workstation. There are some best practices and recommendations to keep in mind:
    - If you plan to integrate with software you're already using, the simplest (and recommended) approach is to install the Integration Agent onto the same server as that software.
    - We recommend that you install the Integration Agent in `C:\xmatters` (Windows) or `/opt/xmatters` (Linux). Extracting the archive into this folder creates a subfolder named `integrationagent-#.#.#`, where #.#.# is the version number. For example:

      ```
      C:\xmatters\integrationagent-5.2.0
      ```
2. Next, extract the integrationagent-utils.zip archive to the subfolder that was created, and allow the extracted lib and integrationservices folders to merge with the ones already present.
    - If you are using an integration that requires a specific version of the IAUtils, follow the instructions in the integration guide for that integration.

Note: *We refer to the installation location (for example,* `C:\xmatters\integrationagent-5.2.0` *in the steps above) as* ***<IAHOME>*** *in our documentation. Also keep in mind that, in most cases, we show the Windows paths and commands — substitute the commands for your operating system wherever appropriate.*

Configure the Integration Agent

The steps below provide the basic settings to get you up and running quickly. You can adjust the other settings if needed for your instance.

1. Navigate into your newly installed Integration Agent, and open the `<IAHOME>\conf\IAConfig.xml` file in a text editor.
2. Edit the following configuration settings (you can leave the other settings as is for now):
    - In the `web-services-auth` section, replace the value in the `user` tags with the name of the web service user you created above, and replace the "Default Company" value in the `company` tags with the name of your

company in xMatters. (Leave the password value for now; we'll get to that later.)

- In the heartbeat section:
  - In the `primary-servers` section, change the subdomain and domain portion of the `url` value to point to the IP address (or hostname) of your xMatters hosted instance. For example, change:
    `<url>https://xyzcustomer.hosted.xmatters.com/api/services/AlarmPointWebService</url>`
    to:
    `<url>https://acmeco.hosted.xmatters.com/api/services/AlarmPointWebService</url>`

3. Save and close the file.

Add your Integration Agent to our Access Control List

When you deploy the Integration Agent, you need to ask us to add it to the access control list (ACL). This allows us to whitelist your Integration Agent so it can talk to your xMatters On-Demand instance. 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**.

Set up the sample plan files on the Integration Agent machine

The final step before sending a test message is to set up the files used by the particular integration. To do this you need to create a password file for your web service and integration users then configure the sample-plan integration's configuration.js file.

Create the encrypted password files

The passwords for the web service user and the integration user are stored in encrypted files in the same folder as the IAConfig.xml file. Fortunately, the Integration Agent includes an encryption utility to create the file.

How to create the encrypted password files

1. Open a command window, and navigate to the `<IAHOME>\bin` folder.
2. Run the following command, but replace <new_password> with the actual password of the web service user you created in xMatters:
   `iapassword.bat --new "<new_password>" --file conf/.wspasswd`
   In our example, this would be `iapassword.bat --new "password" --file conf/.wspasswd` — this is the password referenced in the IAConfig.xml file configuration. In our example, you don't need to do anything else. Just keep in mind that, if you change the name or location of this file, you'd need to update the IAConfig.xml file.
3. Now run the command for the integration user, replacing <new_password> with the password of the integration user you set up:
   `iapassword.bat --new "<new_password>" --file conf/.initiatorpasswd`
   In our example, this would be `iapassword.bat --new "password" --file conf/.initiatorpasswd`

For more information on the IAPassword encryption utility, see IAPassword Utility.

Configure the Integration Agent connection to the sample plan

You need to add the sample plan integration service to the IAConfig.xml file and change a couple settings in the integration's configuration.js file.

How to configure the files for a new integration

1. Open the `<IAHOME>\conf\IAConfig.xml` file, find the `service-configs` section, and check that `<path>applications/sample-plan/sample-plan.xml</path>` is shown and not commented out.
   - For other integrations, you'd add a `<path>applications/integration folder/IntegrationName.xml</path>` line to the section to add it to your integration services list.
2. Save and close the file.

3. Open the `<IAHOME>integrationservices\applications\sample-plan\configuration.js` file in a text editor and locate the following line:
   `WEB_SERVICE_URL = "Paste the Inbound Integration URL here"`
4. Replace the words in the double quotes with the inbound integration URL you copied from xMatters. (Make sure you keep the double quotes!)
   - We'd show you an example, but it's unique to each xMatters instance and each integration.
5. Now locate the following line:
   `INITIATOR = "admin"`
6. Replace `admin` with the name of the sample integration user account you created.
   - In our example, it would be: `INITIATOR = "sampleplan"`
7. Make sure `PASSWORD` matches the path and filename of the encrypted password file you created above for the integration user.
8. Save and close the `configuration.js` file.

You can configure the outbound integrations (or callbacks) and deduplication settings as well.

Start the Integration Agent and check the connection

You can now start the Integration Agent and make sure it's connecting to your xMatters instance.

We recommend that you start the Integration Agent in the console the first time, so you can see the console output. If you see errors when starting the Integration Agent, check out our troubleshooting tips.

How to start the Integration Agent in the console

1. In the command window, run the following command from <IAHOME>\bin:
   `start_console.bat`
   In a non-Windows systems, run `./start_console.sh` (this runs as the current user)

   The console should start (hopefully with no errors), and display the following message:
   "Successfully completed Integration Agent bootstrap process. Integration Agent is running."

2. Now open another command prompt and navigate to `<IAHOME>\bin`, and run the following command (using the IAdmin tool):
   `iadmin get-status`

You should see two indications that the connection to xMatters is valid:

- The xMatters server shows "Connectivity status: PRIMARY ACCEPTED"
- The sample-plan integration shows "Status: ACTIVE"

You can also start the Integration Agent as a Windows service or Linux daemon. When running the Integration Agent as a Windows service or Linux daemon, there is no console output. This means that you must use the log files to monitor agent activities.

How to start the Integration Agent as a service or daemon

Windows

To run the Integration Agent as a service, you first need to install the service by running <IAHOME>\bin\install_service.bat. After you've done that, do one of the following to start the Integration Agent:

- 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.

Linux

Run the following from a command line (this needs to be done manually any time the server restarts):

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

This runs as the current user.

Send a test message

The final step is to check that things are indeed up and running. To do this you can send yourself a test message, using HTTP or the command line.

Send using HTTP

Use RESTClient, cURL, PostMan or a similar utility to send an HTTP POST request to the Integration Agent.

This example sends a message to Bob Smith (bsmith), the Operations group (Operations) and Tara Sanderson's (tsanderson) email and voicemail devices.

URL:

```
http://127.0.0.1:8081/http/applications_sample-plan
```

Request body (formatted here for clarity):

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<event>
       <properties>
              <building>\"Building A\", \"Building B\"</building>
              <city>Victoria</city>
       </properties>
       <recipients>
              <targetName>bsmith</targetName>
              <targetName>Operations</targetName>
              <targetName>tsanderson, devices: ["\Email\", "\Voice Phone\"]</targetName>
       </recipients>
</event>
```

Replace one of the targeted recipients with a notifiable user, group or device that exists in your xMatters instance.

For example, if you use cURL, the command would be:

```
curl.exe -i -X POST -d "<?xml version=\"1.0\" encoding=\"UTF-
8\"?><event><properties><building>\"Building A\", \"Building
B\"</building><city>Victoria</city></properties><recipients><targetName>bsmith</targetName><t
argetName>Operations</targetName><targetName>tsanderson, devices: ["\Email\", "\Voice
Phone\"]</targetName></recipients></event>" http://127.0.0.1:8081/http/applications_sample-
plan
```

Send using 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 applications|<integrationService> <parameters>
```

The parameters you include are based on the settings in the mapped-input section in the integration service file. For example, assume you want to create an event based on the sample-plan communication plan with the following properties:

- recipients: bsmith
- xmpriority:high

- form properties:
  - building: Building B
  - city: Vancouver

Your command line entry uses the following syntax and values.

Linux:

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

Windows:

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

Explore the sample-plan integration further

There's much more you can do with the Integration Agent and communication plans. The following sections give you some examples of how you can edit the sample-plan files.

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, event comments (annotations), escalations, and targeted recipient failures.

Configure outbound integration webhooks in the integration file

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

You can configure a couple of ways:

- Using the Integration Builder in the web user interface: use this method if you want the webhook to fire based on form events.
- Using the integration's configuration file: use this method if you want the outbound integration webhook to fire on the Integration Agent that it came from. Follow the instructions below to configure this method.

Note: *If you configure the webhooks in the integration configuration file, these take precedence over those defined in the communication plan.*

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 (it requests all available webhook types):

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

3. Save and close the file.

You may need to add handling to the Javascript file (`sample-plan.js`) to process the webhooks and parse the information. For more information, see apia_callback(msg).

Mapped input parameters

When you pass in parameter values using the command line, the Integration Agent transforms them into a correctly-formatted REST API call (a JSON object) that generates notifications using the targeted communication plan.

How to customize mapped input parameters

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-plan integration files, this has been remapped to take a parameter called "priority".)

Example APClient.bin configuration

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

If you're creating an event via the command line, you need to specify the values in the same order as they're listed in the mapped-input section. So, in the sample-plan configuration, the command line input expects to receive four parameters in the following 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.

Additional functions in the sample-plan.js file

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

apia_callback(msg)

This function works similar to apia_response (see the Response action scripting section for details), but the parameter is a Javascript object that contains the callback information from xMatters.

You can see sample payloads in the Outbound integrations to the Integration Agent topic in the online help.

The `sample-plan.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

You can use the following functions when using APClient.bin injection and mapped-input parameters. The examples assume that you're targeting the sample-plan with the command line you used to send the test message.

apia_event(form)

This function works similar to apia_input (see the Input action scripting section for details), 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 override any values passed in via the command line.*

The function uses the following syntax to refer to form properties:

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

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

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

The different types of properties available on a communication plan form have required formatting and quoting rules:

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"`

You can also use the apia_event function to specify default recipients. As with properties, any recipients defined within the Javascript file overrides the recipients passed in via the command line. The syntax to set the recipients is:

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

The formatting and quoting rules are also similar, for example:

```
'[{"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-plan.js` file:

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

  // 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)

Example

For example, xMatters will always use the 'xmpriority' mapped-input parameter to set the value of the 'priority' attribute in the JSON object. Now, imagine that you're 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-plan 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"
       }
}
```

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.

Installation requirements

There are a few things to check on your side before you install the Integration Agent.

Supported operating systems

The Integration Agent supports the following operating systems:

Platform	Manufacturer	OS	Description
64-bit	AMD, Intel	Windows	Server 2008 R2, Server 2012 R2
		Linux	Various, including Red Hat / CentOS 6 and 7
Sparc	Sun	Solaris	10
Itanium	HP	HPUX	11.31 For HP-UX Itanium deployments, check the comments in the wrapper.conf file for more information.
POWER Processor	IBM	AIX	7.1

Things to set up or have at hand during the installation and configuration

Item	Description
User account	A user account on the server where you plan to install the Integration Agent. On Windows, this account must have permissions to create Services.

Item	Description
Hard disk space	500+ MB of free disk space
RAM	640+ MB of available memory
Proxy server setup	If you want the Integration Agent to communicate with xMatters 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.
TCP ports	A free TCP port on the server for each of the following components: ■ Integration services (default: 8081) ■ IAdmin (default: 8082) ■ APClient (default: 2010) ■ Internal message broker (default: 61618) We refer to some of these ports in our instructions. If you change the port, remember to replace the default port with the port you're using.
SMTP server details (if Health Monitor is enabled)	If Health Monitor is enabled, you need the hostname/IP address and port (default is 25) of an SMTP server reachable without user/password authentication by the server where the Integration Agent is installed.
Email address (if Health Monitor is enabled)	If Health Monitor is enabled, you need to provide the email address of the person who you want to receive Health Monitor messages from the Integration Agent.
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.

Integration Agent configuration files

The Integration Agent uses two types of XML configuration files:

- Integration Agent configuration file
- Integration services configuration files

You can change a number of settings in these files to customize your installation.

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

Integration Agent configuration file

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

- Windows: `<IAHOME>\conf`
- Linux: `<IAHOME>/conf`

Element	Description
id	Each Integration Agent must have a unique ID across all collaborating Integration Agents. If you don't specify this parameter, a default agent ID is auto-generated in the format <computername>/<ip>:<service-gateway port>. For example: `<id>workstation-198-bsmith/10.2.0.121:8081</id>` If you assign a custom agent ID, make sure that any **<** and **&** characters in the name are replaced with their XML entity names (in other words replace **<** with `&lt;` and **&** with `&amp;`). Any leading or trailing whitespace is removed.

Element	Description
proxy-config	Use the proxy-config settings if you need to configure the Integration Agent to communicate with xMatters via a proxy server. By default, the proxy-config section commented out, and the Integration Agent does 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-path>: If required, the path (relative to this file) that refers to a file containing the password (encrypted via iapassword) to use 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-path tags aren't commented out, the Integration Agent sends an authorization header to the proxy server, even if the tags are empty. If the tags are empty; the header contains only a ":" character, which can cause the proxy connection to be refused.* ■ <proxy-auth-ntlm-domain>: The domain name to use for the proxy server if it uses NTLM v1 authentication. If the proxy server does not use NTLM v1 authentication, omit this setting or leave it blank (NTLM v2 is not currently supported). If you're using a wrapper.conf file which specifies proxy settings, those settings 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 xMatters 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-elements:** ■ <user>: Login name (username) for the xMatters web service user 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 ./.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
heartbeat	The Integration Agent periodically sends a heartbeat request to xMatters to identify the integration services it provides. The heartbeat element settings identify the web server the Integration Agent should target with these registration attempts. The URLs specified must point to the location of the AlarmPointWebService, and begin with either http:// or https://. The URL cannot have a query or fragment component (the URL must be resolvable from the Integration Agent). For example: `http://xyzcustomer/api/services/AlarmPointWebService` **Sub-elements:** ■ <primary-servers>: The primary location of each server's RegisterIntegrationAgent Web Service. See the Health Monitor section for more information about the messages associated with these settings.
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 Application Server Nodes and users of APClient.bin. **Attributes and sub-elements:** ■ @enable: Controls whether requests are authenticated by IP address (values: true, false). ■ <ip>: IP address identifying an authorized client. You can have 0 or more IP addresses. Addresses can include wildcards (for example, 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 Application Server Nodes and users of APClient.bin. **Attributes and sub-elements:** ■ @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 ./.passwd).
external-service-request	This determines the behavior of xMatters when ExternalServiceRequest2's send() method is called and this Integration Agent is the target. We recommend that you don't change this setting without first talking to someone at support.xmatters.com.

Element	Description
request-timeout	The maximum number of seconds that this Integration Agent processes a client request before cancelling it with a timeout exception sent to the client (value: positive integer). This applies to integration service requests, ExternalServiceRequest2 requests, and input and response action scripting. It is not possible to increase the timeout for individual integrations. We strongly recommend that you do not increase this setting, since it can result in extended delays building up. See the information on script timeouts in the Input action scripting section for more information.
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 – must be an unused TCP port that is 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). ■ @port – must an unused TCP port that is different from the ports of the other gateway elements (value: integer from 0 to 65535).
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). If set to true, Apclient.bin won't be able to send notifications to the Integration Agent. Only change this setting if you have integrations in which the management system sends APXML requests directly to the APClient gateway. See the APXML reference and APClient.bin sections for more information. ■ @host – hostname (e.g., localhost) or IPv4 address (e.g., 127.0.0.1) of the Integration Agent. ■ @port – must be an unused TCP port different from the ports of the other gateway elements (value: integer from 0 to 65535).

Element	Description
emergency-contact	Where the Integration Agent sends emergency notification emails when a serious condition occurs (for example, network failure or low memory; see the Health Monitor section for details. **Attributes and sub-elements:** ■ @enabled – whether the Health Monitor is active (values: true, false). When set to true, the Health Monitor sends an email message to the specified email address every time an issue is detected or resolved. ■ <contact> – the email address to which Health Monitor messages are sent, the email address from which to send Health Monitor messages, and the subject of the email message. ■ <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.
service-configs	The configuration files for integrations 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 configuration files that the Integration Agent loads (paths must be relative to `<IAHOME>/integrationservices`, and can be Linux- or Windows-formatted).

The <outbound-queue>, <threshold> and <polling-interval> settings are covered in the Health Monitor documentation.

Integration service configuration file

This file is unique to each integration. It's located at:

- Windows: `<IAHOME>\integration-services\<integration-name>\<integration-name>.xml`
- Linux: `<IAHOME>/integration-services/<integration-name>/<integration-name>.xml`

Element	Description
domain	For integrations that use a communication plan, this must be "applications".

Element	Description
name	The name of the integration service, which must be unique (regardless of letter case). 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 ("-").
initial-state	Only active integration services process requests, though all services are loaded (parsed and configured) regardless of state. A suspended service allows requests to be added to its inbound queues, but the requests aren't processed until the service is active. Suspending a service initially is useful for debugging a configuration since parsing and validation are still performed.
concurrency (optional element)	For improved performance, integration services can concurrently process notification requests from your management system and callbacks from xMatters. Messages are grouped in two stages: first by priority then 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. We recommend you don't change these to exceed their default values. Doing so can reduce throughput.
script	Integration service requests are implemented by corresponding methods in a JavaScript file specific to the integration. This element defines the location of the script and related 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 can be Linux- or Windows-formatted).

Element	Description
classpath (optional element)	Integration service scripts have access to all classes and JARs stored in `<IAHOME>/lib`. However, to prevent conflicts and enhance security, you can specify that an integration service loads its own classes and resources from an unshared directory. The classpath element allows you to specify multiple paths that will be added to the integration service's classpath during the processing of an integration service request. 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-elements:** ■ <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	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 subelements 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 has 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.

Element	Description
constants	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.

The IAPassword utility

Both the Integration Agent and the integration services' configuration files can include configuration settings that refer to files containing encrypted data. The IAPassword utility is a platform-independent Java program that an Integration Agent administrator can use to create and modify the contents of these encrypted files.

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 example, the web service user account that the Integration Agent uses whenever it calls the RegisterIntegrationAgent, SubmitAPXML, or ReceiveAPXML web service methods is defined by the webservices-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 plain text, the Integration Agent decrypts the .wspasswd file and uses the single string contained within as the password element's actual value.

Similarly, an integration service's configuration files can contain encrypted elements (for example, integration user passwords and encrypted-constant elements) that get their values from encrypted files. Here's an example of an encrypted-constant:

```
<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>
```

You can find the IAPassword program in the following location:

- Windows: `<IAHOME>\bin\iapassword.bat`
- Linux: `<IAHOME>/bin/iapassword.sh`

IAPassword parameters

IAPassword accepts the following command-line parameters:

Parameter	Required	Description
--new <string>	Yes	Specifies the string to be stored in the encrypted file.
--old <string>	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. This can be specified as absolute or as relative to the installation folder (NOT relative to the current directory). If not specified, the default value is `<IAHOME>/conf/.passwd`. (There's an example of how to change the Web Service password below.)

Examples

For example, to change the web service user's password, use the following command:

```
iapassword --new "My New Password" --old ia_user --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, use 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\AlarmPointIAdmin.txt` file. You can change this logging behavior by modifying the log4j configuration file at `<IAHOME>/conf/cli/log4j.xml`.

Manage and troubleshoot your Integration Agent

There are some basic tasks you can follow and logs you can review to get information on the status of your Integration Agent and specific integrations, and to troubleshoot issues you might come across.

Stopping the Integration Agent

You might need to stop the Integration Agent following integration script or Integration Agent configuration changes, or to clear the inbound queues.

Follow the instructions for your operating system to stop the Integration Agent.

Windows

To stop the Integration Agent running as a Windows 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.

Linux

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 Linux application:

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

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 (if you have multiple Integration Agents configured.

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.*

Possible integration service runtime states

The IAdmin tool's suspend or resume commands changes the service's runtime state between SUSPENDED and ACTIVE.

State	Description
ACTIVE	Indicates the integration service is able to access and process requests.

State	Description
SUSPENDED	Indicates the integration service is properly configured, but has been manually set to deny requests (for example, when a management system is undergoing maintenance — this allows you to work on one service and reload it without impacting other system components). When suspended: ■ the service rejects all external service requests and APClient requests. ■ the service rejects any direct external service requests. ■ the 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 are queued, and then processed when the service is set to an ACTIVE state. ■ outbound messages from the service are still forwarded to xMatters through the message exchange process.
ERROR	Indicates the integration service is improperly configured, in most cases due to an issue in the configuration file or a script syntax error. Once the cause of the error is identified and corrected, you can reload the integration service and make it active by using the the IAdmin tool or by restarting the Integration Agent.

IAdmin tool

The Integration Agent includes a command line tool, named IAdmin, that you can use to issue commands to and get the status of the Integration Agent after it has started.

IAdmin is located at:

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

IAdmin commands

There are various commands you can use with the iadmin command line tool to manage the Integration Agent and get information on its status.

Command syntax	Description
get-status	Use the get-status command to perform troubleshooting or 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 ■ Integration services list (including integration service name, clients, integration service request URL and integration service start date/time) ■ Last request (last time an integration service request was received) for each integration service, with status (ACTIVE/SUSPENDED/ERROR), number of requests being processed, and inbound and outbound APXML queue sizes. ■ xMatters Primary Server List (including server URL, Server connectivity status, and last heartbeat
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.

Command syntax	Description
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. Unlike other IAdmin commands, `purge` can be executed even if the Integration Agent is not running.
purge all	Removes all inbound and outbound APXML messages from all integration services. APXML messages that are being processed are maintained. Unlike other IAdmin commands, `purge` can be executed even if the Integration Agent is not running.

Server Connectivity Status

Command syntax	Description
UNKNOWN	No connection attempt has been made, or the attempt has not been completed.
FAILED	No connection can be made between the Integration Agent and the primary xMatters web server. 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 (see the Integration Agent log for details).

Command syntax	Description
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.

IAdmin Logging

The results of administrative commands in the IAdmin tool are displayed in 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, and additional details are captured in the log files.

IAdmin exit 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.

IAdmin tool exit codes

Error code	Description
0	Success (no pending requests)
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

Integration Agent log

The log files for the Integration Agent and the IAdmin tool are the first place to check if you notice issues with your installation.

Integration Agent logs: Contains logs for all Integration Agent entries of a warning level or higher (by default).

- Windows: `<IAHOME>\log\IntegrationAgent.txt`
- Linux: `<IAHOME>/log/IntegrationAgent.txt`

IAdmin log: Contains the results of administrative commands in the IAdmin tool.

- Windows: `<IAHOME>\log\IntegrationAgentIAdmin.txt.#`
- Linux: `<IAHOME>/log/IntegrationAgentIAdmin.txt.#`

The Integration Agent uses log4j version 1.2.14 for logging. The information here covers items that are specific to the Integration Agent. See the log4j documentation for full details about its format and its settings.

Default log entry format

Log entries are in the following format:

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

Example

The following represents a log entry in the default format:

```
2018/04/25 15:52:01.537 -0700 PDT [applications|sample-plan-1] INFO - Calling JavaScript
method apia_http
```

Logging configuration file

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

- Windows: `<IAHOME>\conf`
- Linux: `<IAHOME>/conf`

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

Logging categories

Most Integration Agent log messages appear under the com.alarmpoint.integrationagent category hierarchy. Each major component or activity (for example, Health Monitor, heartbeats, integration service requests, etc.) logs to a dedicated subcategory. The advantage to this approach is that logging can be focused on a specific aspect of the Integration Agent's activities.

log4j Categories

The following table summarizes selected log4j categories:

Category	What it logs
com.alarmpoint.integrationagent	All Integration Agent activity. **Note:** *In the entries that follow, <root> represents com.alarmpoint.integrationagent*
<root>.admin	IAdmin tool requests
<root>.apclient	APClient.bin submissions
<root>.health	Health Monitor activity
<root>.health.mail	Health Monitor mailer activity

Category	What it logs
<root>.heartbeat	Heartbeats activity
<root>.messaging	apxml-exchange activity
<root>.services	All integration services activity
<root_AP_cat>.services.test_service_1	Example for specific integration service. This would log the test_service_1 integration service.

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

Example of exposing a logging category

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

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

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

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

Troubleshooting

If you encounter any hiccups using the Integration Agent, there are some things you can check to help fix the issue.

Startup issues

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

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

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. You can inspect the startup exit codes to investigate startup issues.

Startup exit codes

- 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 appears in the SERVICE_EXIT_CODE field.

- Alternatively, if the shutdown.bat script is used to start the Integration Agent, the batch file returns the service exit code.

- Linux: 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. The syslog daemon must be configured by an administrator to perform logging.

Additionally, on both Windows and Linux, 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.

Startup exit codes	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

Startup exit codes	Description
88	Mule timeout error
90	Nonspecific startup error (check logs)

Unix 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.

How 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/IntegrationAgent.txt
   ```
3. Log out of the root account.
4. Restart the Integration Agent daemon as a non-privileged user.

SSL certificates error

When starting up the Integration Agent, you might encounter an error if the SSL certificates were not imported properly.

Error message format and resolution

The error message is 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
```

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.

Linux 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.

How to search for the process on Linux systems

On some Linux systems, your can search for this process 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.pid` should allow the Integration Agent to start.

Integration service request issues

There are two main types of integration service issues that can occur: integration service request receives no SOAP response and integration service receives SOAP error response.

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

How to troubleshoot integration service issues

1. Make sure the Integration Agent has started then 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, check 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, if an integration service has the following configuration:

- Name: sample-plan
- Event Domain: applications
- IAConfig.xml: `<service-gateway ssl="true" host="www.company.com" port="8081"/>`

To test 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/applications_sample-plan?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, the problem is likely related to an incorrectly configured Integration Agent client. If this response is not received, the problem is likely 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.

How to troubleshoot SOAP error response issues

1. Make sure the Integration Agent has started then 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 sample-plan, the log4j.xml entry would be similar to the following:
```
<logger name="com.alarmpoint.integrationagent.services.applications.sample-plan">
       <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.

How to troubleshoot heartbeat issues

1. Make sure the Integration Agent has started then 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 either indicates a connectivity failure or provides one of the following reasons for the heartbeat's rejection:
   - UNKNOWN_DOMAIN: indicates that the integration service's event domain has not been configured on the xMatters web server
   - UNKNOWN_SERVICE: indicates that one of the integration service names has not been configured 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's JavaScript.

How to determine the integration service configuration issue

1. Review the Integration Agent log file.
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 contains 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\netc
ool-admin.js#71)
```

Next steps

Now that you have a functioning Integration Agent, you can further explore and customize your installation.

Check out our packaged integrations

Look under the hood on our of our packaged integrations that use the Integration Agent, like the one for CA Service Desk, to learn more about how integrations work with the Integration Agent.

Perform advanced management and monitoring of your Integration Agent

There are a number of ways you can further customize your Integration Agent setup and management.

Some of the things you can do are:

- Configure deduplication (suppress duplicate event notifications before the Integration Agent sends them to xMatters).
- Set up multiple Integration Agents to enhance fault tolerance.
- Initiate communication with the Integration Agent using APXML messages.
- Inject information into the Integration Agent using a direct HTTP or HTTPS request.
- Configure the Integration Agent Health Monitor.

Manage your Integration Agent

Beyond the basic Integration Agent management and troubleshooting of the Integration Agent, there are some additional options you can use to customize and monitor your Integration Agent installation, including filtering and suppression, queue analysis and the Health Monitor.

Filtering and suppression

Filtering and suppression allows for the suppression of duplicate events (also referred to as "deduplication"). You can also implement flood control within xMatters On-Demand. However, implementing filtering and suppression within the Integration Agent helps avoid disruption of network traffic due to inadvertently high loads by stopping duplicated events before they're even sent to xMatters.

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 values that are considered relevant for the purpose of correlation.
  - Each predicate must be wrapped in <predicate> tags (see Filter Use Case examples 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 number of non-matching events to count before resetting the suppression_period timer. After this number is reached, an event is not considered to be a duplicate even if the predicates match and the suppression period has not elapsed.

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 filter records the values of the "node" and "person_or_group_id" predicates. After a message is processed, the filter blocks all subsequent messages with identical predicates until either the suppression period expires or the number of unique messages exceeds the window size.

Filter use case examples

In these examples, the deduplicator-filter.xml file includes the following filter:

```
<filter name="sample-plan">
       <predicates>
              <predicate>city</predicate>
              <predicate>person_or_group_id</predicate>
       </predicates>
       <suppression_period>180</suppression_period>
       <window_size>1000</window_size>
</filter>
```

Use case 1: Duplicates detected (suppression period elapses)

1. At 14:00:00 an event is injected with predicates (city: "London", person_or_group_id: "bsmith").
   - The event is not considered a duplicate.
2. At 14:02:00 an event is injected with predicates (city: "London", person_or_group_id: "bsmith").
   - The event is considered a duplicate and is suppressed.
3. At 14:04:00 an event is injected with predicates (city: "London", person_or_group_id: "bsmith").
   - The event is not considered a duplicate because the suppression period has elapsed (180 seconds or 3 minutes). 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", city: "NYC").
   - The event is not considered a duplicate.
2. At 14:01:00 an event is injected with predicates (node: "Node123", city: "NYC").
   - The event is not considered a duplicate. Although the predicates are identical for both events, they are not identical to the filter, since the location predicate is not included in the filter.

Use case 3: Events not suppressed (window rolls)

1. At 14:00:00 an event is injected with predicates (city: "London", person_or_group_id: "bsmith").
2. By 14:01:00, 1000 distinct events are injected
   - None of these 1000 events matches the filter, so they are not suppressed. The number of events is now 1001, which means the window size is reached and the suppression period reset (the first event is purged).
3. At 14:02:00 another event with predicates (city: "London", person_or_group_id: "bsmith") is injected.
   - 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 by 14:01:00.

How to try out filtering and suppression with the sample integration

The Integration Agent sample-plan integration includes code to implement filtering and suppression. Here's an overview of how to try it out:

1. Set up the Integration Agent and sample-plan integration.
2. Edit the deduplicator-filter.xml file to look for one or more of the event properties (the sample-plan deduplicator looks for building and city)
   - Just remember that **all** of the deduplicator's predicates must be met for the event to be deduplicated.
3. Configure the sample-plan-http-inject.bat script (or its Linux .sh counterpart) with one or more recipients who exist in your xMatters instance (and make sure it is injecting all the properties in your deduplicator-filter.xml file).
4. Open up xMatters and navigate to the Reports tab, then use the script to send a test notification. Check the Reports tab to confirm that the event was created.
5. Use the script two or more times within 20 seconds to send some identical requests.
   - The sample-plan scripts and the deduplicate filter in xmutil.js check whether or not the event should be suppressed (deduplicated).
6. Check the Reports tab again to make sure there is still only one event.
   - You can also check the Integration Agent log for messages like the following one, which indicate that a notification request was blocked by the deduplicator:

     ```
     2018/05/29 15:38:15.157 -0700 PDT [applications|sample-plan-1] WARN - An event with
     properties {
     …
     } has been injected into the sample-plan service more frequently than allowed by the
     deduplication filter sample-plan. It has been suppressed.
     ```

7. You can send another test notification 20 seconds after you sent the last one, then use the Reports tab and log to confirm that a new event was created (because the suppression_period of 20 seconds has elapses).

The following code snippets from `sample-plan.js` shows the relevant parts:

```
// prepare an empty event request object
var event = XMUtil.createEventTemplate();

event.properties = {};

// Deduplicate based on form properties.
if (XMUtil.deduplicator.isDuplicate(event.properties)) {
       // Discard message, automatically adding a warning note to the IA log
       XMUtil.deduplicate(event.properties);
       return;
}

// Send to xMatters
XMIO.post(JSON.stringify(event));
// If the post succeeds, register the event with the deduplication filter.
XMUtil.deduplicator.incrementCount(event.properties);
```

The sample integration should not be used for production. 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).

Disabling the filtering and suppression module

You can disable the filtering and suppression module on a per integration basis.

To disable filtering for an integration:

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 earlier versions of the Integration Agent, 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. This disables filtering and suppression for all integrations.

Fault tolerance

You can configure multiple Integration Agents to share a common database in a master/slave setup to enhance the fault tolerance of your deployment.

With this setup, any Integration Agent can receive injections from the management system. When a message is injected, the receiving 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.

How it works

- Inbound messages are temporarily stored in an external database before being processed and sent to xMatters.
- At least one slave Integration Agent is required, ideally on a physically separate server (if permitted by the architecture of the integrations you're using).
- Inbound messages can be submitted to either 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 returns 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.
- Because injected messages are all processed by the master, event deduplication is performed even if a load balancer is used to distribute injected messages to multiple Integration Agents.
    - This means that if the master Integration Agent fails, event deduplication will be disrupted because the information require for deduplication is lost.
- 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 remains 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).
- Outbound integrations from xMatters are always directed back to the master Integration Agent, not the Integration Agent that initially received the incident request from the management system. This will cause outbound integrations to fail if the master Integration Agent is not connected to the management system that initiated the notification request.

Planning for a fault tolerant configuration

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 there is some specific information to keep in mind before configuring the Integration Agent to use an external database.

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.
- Responses from xMatters, including annotations, user responses.

Approximate database requirements

You can approximate maximum database requirements by simulating a worst-case network outage (in other words, an outage that disrupts communication between the Integration Agent and xMatters) 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 database management system that you plan to use for the shared persistent queue.

Built-in database

The Integration Agent's built-in database manager 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. The size of this file can be used to approximate the database size requirements for the shared persistent queue.

External database

You can also perform the test with the Integration Agent already configured to use a shared persistent queue. In this case, your database administrator can view the effect of the test on the data tables in a staging instance of the database management system (Oracle and SQL Server) you're using.

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 integration service's XML configuration file (for example, `<IAHome>/integrationservices/applications/sample-plan/sample-plan.xml`), find `<initial-state>active</initial-state>` and replace it with `<initial-state>suspended</initial-state>`.
2. Restart 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.

Estimating storage requirements

Using the above method, you can estimate storage requirements fairly accurately. This estimate does not include user responses or annotations from xMatters. The volume of data from these is generally not significant, but if you need to include these responses and annotations in the estimate, you can send a volume of test messages to xMatters via the Integration Agent immediately before putting the integration in suspended mode. When you restart the Integration Agent, it receives the annotations and responses that were generated after the Integration Agent was stopped, giving you an estimate of their impact on the storage requirements.

Configuring Integration Agents for shared persistent queue operation

Before configuring an Integration Agent for a shared persistent queue, make sure:

- all injected messages have been processed and sent to xMatters. Any messages that were in the Integration Agent's built-in queue will not be available once the Integration Agent starts using the shared queue.
- the Integration Agent is operating correctly, and communicating with xMatters.

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

To configure a shared persistent queue:

1. Stop the Integration Agent.

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.
   a. Comment out the entire **kahaDB** line. In other words, replace:

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

   With

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

   b. Uncomment the **jdbcPersistenceAdapter** line and make sure the dataSource atttribute matches the database type (Oracle or SQL Server). For example, replace:

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

   With:

   `<jdbcPersistenceAdapter dataDirectory="activemq-data" dataSource="#oracle-ds"/>` for Oracle, OR

   `<jdbcPersistenceAdapter dataDirectory="activemq-data" dataSource="#sqlserver-ds"/>` for SQL Server

   For Oracle, the node should now be similar to:

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

   For SQL Server, the node should now be similar to:

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

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

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

   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.

8. Locate the bean id="oracle-ds" line, if you're using Oracle, or the bean id="sqlserver-ds" line, if you're using SQL Server, and 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}"/>
   ```

9. Save and close the `activemq.xml` file.
10. Open a command line and run `<IAHOME>bin/iapassword` to create a file that contains the encrypted password:
    `bin/iapassword.bat --new my_password --file bin/<filename>.txt` (replacing <filename> with the actual filename you want to use for the password file).
11. In a text editor, open the file containing the encrypted password and copy the password.
12. Create an empty file named `dbpassword.properties` in <IAHOME>/conf/ and add the line:
    `db.password=<encrypted password>` (where <encrypted password> is the password you copied). The text in the file should now look similar to the following:

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

13. 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 updates.*

Configuring the ActiveMQ connections

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

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. For example, replace:

   ```
   <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. For example replace:

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

   With:

   ```
   <property name="brokerURL" value="failover:
   (tcp://host1:61616,tcp://host2:61617,tcp://host3:61618)?jms.redeliveryPolicy.maximumRedel
   iveries=-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 (use <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

You might 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 complete the shared persistence queue configuration steps, use the following points to check that your deployment is operating correctly and that fault tolerance has been enabled.

- The **first** Integration Agent *will be the master*, meaning that it can receive messages from the management system and add them to the inbound queue, and 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. These Integration Agents don not remove messages from the queue unless the master becomes unavailable.
- When starting the **first** Integration Agent in console mode, look for 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 starts.
- 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.applications.sample-plan.normal-3] INFO - Component applications_sample-plan has
received the following request from endpoint jms://inbound.applications.sample-
plan.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 helps ensure that notification requests are always handled by the same Integration Agent, which facilitates 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 xMatters. This is expected behavior and does not indicate a problem.*

Troubleshooting fault tolerance and shared persistent queue setup

Issue	Troubleshooting
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...`	The most common cause of 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 make sure you follow the steps outlined above. 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	Troubleshooting

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

Cannot find specific information about the queues.	The Integration Agent logs should include specific information about the queues as long as the log4j.xml file has been configured as described above. For example, for information about the sample-plan integration's inbound queue, look for lines similar to the following: `[BrokerService[localhost] Task] DEBUG - inbound.applications.sample-plan.normal toPageIn: 1, Inflight: 0, pagedInMessages.size 0, enqueueCount: 24, dequeueCount: 23` This indicates that a message was 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, and 23 of them have been released for processing. The messages in the queues may have been placed there by any of the Integration Agents, and not necessarily by the Integration Agent that is logging the information.

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, use the following steps to validate and kill expired sessions on resources that can prevent normal startup. Select the instructions 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;
```

Queue analysis

The Integration Agent uses JMS queues to store requests from the management system and from xMatters.

Each integration service has its own set of queues. For example, the sample-plan integration in the applications domain has the following queues:

- inbound.applications.sample-plan.normal
- inbound.applications.sample-plan.high
- outbound.applications.sample-plan.normal
- outbound.applications.sample-plan.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). The ActiveMQ outbound queues are not used by the Integration Agent.

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 writes messages for all queues to the log several times per minute, similar to the following:

```
dequeueCount: 2
```

In the above example, the normal-priority inbound queue of the sample-plan's integration service 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 multithreaded operation, or make it multi-threaded if it is not. Information on how to make an integration service multithreaded 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.

Health Monitor

The Integration Agent includes a component called the Health Monitor that sends messages about important agent events to a specified email address. You can configure Health Monitor settings using the Integration Agent IAConfig file. 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 IAconfig.xml file, or related to a specific integration service. The following table summarizes events that trigger Health Monitor messages:

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 generates 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 (for example, 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.

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: applications, Name: sample-plan, 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 Manage and troubleshoot your Integration Agent.*

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 attempts 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.

Service API

Integration services that are implemented with JavaScript 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-plan
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.*

Logging

Each integration service has its own Log4j logging category in the following format:

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

For example, the sample-plan integration that is part of the applications domain has the following category:

```
com.alarmpoint.integrationagent.services.applications.sample-plan
```

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`.

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.

This example shows how to use the Service API's logger to log the ID of the Integration Agent in the previous example to the log file:

```
ServiceAPI.getLogger().info("IA ID is: " + ServiceAPI.getConfiguration().getAgentId() );
```

Which writes the following output to the log:

```
2018/05/23 10:35:55.217 -0700 PDT [applications|sample-plan-1] INFO - IA ID is: vic-vw-jb-ia-
test/10.2.0.126:8081
```

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. An integration service can make an ISR to another integration service (either on the same Integration Agent or on a remote Integration Agent) 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.*

To make an ISR, an integration service must know the following information:

- The targeted integration service's domain and name (e.g., `applications|sample-plan`).
- 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("applications");
        isr.setName("sample-plan");
        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., applications|sample-plan)
- 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, "applications", "sample-plan");
}
```

Note: *This example assumes that the APXML passed to* `apia_example` *already includes the tokens that are required by the sample-plan integration service (e.g.,* `request_text` *and* `device`*).*

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

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 an 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>applications|sample-plan</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 are not used when sending event requests to xMatters (event requests are always sent to xMatters via REST requests), but are used when requests are sent to the Integration Agent's apclient gateway and when xMatters sends callbacks to the Integration Agent.

Supported input methods

The following table summarizes the methods/subclasses that xMatters recognizes:

Method	Subclass	Description
Add	Action	Creates an xMatters event
Add	Event	Creates an xMatters event

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

xMattersrequest methods

The following table summarizes the methods/subclasses that are used for xMatters requests:

Method	Subclass	Description
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.

Supported input methods message for xMatters

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

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 APXML Methods.*

Integration Agent APIs

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

Utility scripts

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-plan integration.

To use a customized utility script:

1. Copy `<IAHOME\lib\integrationservices\javascript\xmio.js` to `<IAHOME>\integrationservices\applications\sample-plan`.
2. Open the `<IAHOME>\integrationservices\applications\sample-plan\sample-plan.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. Add the following new load statement:

```
load("integrationservices/applications/sample-plan/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.

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://<IntegrationAgentIP>:<serviceGatewayPort>/http/applications_<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 sample-plan integration uses the following URL:

```
http://localhost:8081/http/applications_sample-plan
```

Integration agent configuration

In addition to defining the 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\applications\<integrationService>\<integrationService>.js
```

For example, the sample-plan integration uses the following default location:

```
<IAHOME>\integrationservices\applications\sample-plan\sample-plan.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-plan.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-plan-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 Get up and running with the Integration Agent.

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");
```

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/applications_sample-plan,
  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://<IntegrationAgentIP>:8081/http/<integrationName>_<integrationService>
```

For example:

```
http://localhost:8081/http/applications_sample-plan
```

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_sample-plan/">
  <soapenv:Header/>
    <soapenv:Body>
      <apia:TriggerResponse>
      <status>Error</status>
      <description>Request action not found.</description>
    </apia:TriggerResponse>
  </soapenv:Body>
</soapenv:Envelope>
```

HTTPS requests

For added security, you can add an SSL certificate to the Integration Agent so that it can receive notification requests via HTTPS posts.

Note: *By default, the Integration Agent is set up to serve (receive) SSL requests. The APClient.bin formats the data sent to it into an HTTP POST and then sends it to the Integration Agent via SSL; however, the SSL certificate that ships with the Integration Agent is not suitable for use by any SSL client except APClient.bin.*

The Integration Agent comes with an example keystore file that you can add a functional certificate to, or you can create a new keystore file. Within the keystore file, you can either create a new self-signed certificate or add a commercial certificate. In the example below, we describe how to create a keystore file that contains a new self-signed certificate.

Once you've created the keystore and certificate, you'll need to export the certificate from the keystore and import it into the trust store of the entity (i.e., a management system) that will be sending requests to the Integration Agent. Otherwise, the management system will not trust your self-signed certificate and it will break the connection to the Integration Agent, rather than completing the request.

To create a keystore containing a self-signed certificate:

1. On the Integration Agent server, open a command prompt and navigate to `<IAHome>`. In this example we will use "`c:\xmatters\integrationagent-5.2.2`".
2. Use the following command to create the keystore and initiate generation of the certificate:

```
C:\xmatters\integrationagent-5.2.2>jre\bin\keytool -genkey -keyalg RSA -validity 3600 -keysize
2048  -alias xmia-xyzcompany-com -ext SAN=dns:xmia.xyzcompany.com -keystore
./conf/keystore-xmia-xyzcompany-com  -storepass alarmpoint
```

- Replace the alias, the SAN (Server Alternate Name), and the keystore name with names of your choice, but they should be consistent with each other and with the values that you enter in the next step, or results may be unpredictable.
- Replace the storepass with a password of your choosing. This is the password that the Integration Agent uses to retrieve the certificate from the file when responding to HTTPS requests, so you will need to provide the password when editing the Integration Agent configuration files.

3. Follow the prompts that appear when providing the data that will be encoded in the certificate. For example:

```
What is your first and last name?
  [Unknown]: xmia.xyzcompany.com
What is the name of your organizational unit?
  [Unknown]: operations
What is the name of your organization?
  [Unknown]: Xyzcompany, inc
What is the name of your City or Locality?
  [Unknown]:  Los Angeles
What is the name of your State or Province?
  [Unknown]:  California
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=xmia.xyzcompany.com, OU=remedy, O="Xyzcompany, inc", L="Los Angeles", ST=California,
C=US correct?
  [no]:  yes

Enter key password for <xmia-xyzcompany-com>
        (RETURN if same as keystore password):
```

- The result will be a new file in `<IAHome>\conf`, called "`keystore-<name>`".

2. Make a backup copy of `<IAHome>\conf\mule-config.xml` and then open the original file with a text editor.
3. Replace the following:

```
    <connector name="httpsConnector" className="org.mule.providers.http.HttpsConnector">
        <properties>
           <property name="keyStore" value="./conf/.keystore"/>
```

with:

```
    <connector name="httpsConnector" className="org.mule.providers.http.HttpsConnector">
        <properties>
<!--            <property name="keyStore" value="./conf/.keystore"/>  -->
       <property name="keyStore" value="./conf/keystore-xmia-xyzcompany-com"/>
```

4. Replace "`alarmpoint`" in the lines that follow with the password you chose when generating the certificate. If you chose not to use the same password for the keystore and the certificate, update the "`storePassword`" property with the password for the file, and the "`keyPassword`" property with the password for the certificate.
5. Make a backup copy of `<IAHome>\conf\IAConfig.xml` and open the original file with a text editor.
6. Replace the following:

```
<service-gateway ssl="false" port="8081"/>
```

with:

```
<service-gateway ssl="true" port="8081"/>
```

7. Save the files and restart the Integration Agent.
    - If there are errors during Integration Agent startup, generate a support-zip, then restore the backups you made of the `IAConfig` and `mule-config` files. Restart the Integration Agent (which should now be functioning properly again) and contact xMatters Client Assistance.

Note: *If you are working with a commercial certificate instead of creating a new self-signed certificate, the vendor will provide directions for adding the certificate chain to an X.509 keystore. After you have prepared the keystore and configured the* `IAConfig` *and* `mule-config` *files as described above, your Integration Agent should be ready to receive HTTPS requests.*

To export the certificate from the keystore, use the following command:

```
jre\bin\keytool -export -keystore conf/keystore-xmia-xyzcompany-com -storepass alarmpoint
-alias xmia-xyzcompany-com -file xmia-xyzcompany-com.crt
```

The exported certificate (`xmia-xyzcomany-com.crt` in the above command) must now be copied to the management system and imported into its trust store. If the management system uses Java, the trust store will usually be located

within its "jre" subdirectory at `<jrehome>\lib\security\cacerts`. Make sure you make a backup copy of the `cacerts` file before importing the certificate.

If your management system does not use Java, you may need to engage its support personnel for this task. Otherwise, use the following directions to import the certificate.

To import the certificate to your management system:

1. Open a command prompt and navigate to the jre folder.
2. Run the following command to import the certificate:

```
bin\keytool -import -keystore lib/security/cacerts -storepass changeit -file
conf/xmia-xyzcompany-com.crt -alias xmia-xyzcompany-com
```

3. Answer "yes" when prompted "Trust this certificate? [no]:"
4. Verify that the input was successful with:

```
bin\keytool -v -list -keystore lib/security/cacerts -storepass changeit -alias
xmia-xyzcompany-com
```

5. You may need to restart the management system to make the trust store changes take effect. You may also need to restart the Integration Agent.
6. Send a test submission to the Integration Agent via HTTPS and verify that it was successful.
   - You can use cURL, POSTman, or something similar to send a submission to the Integration Agent using the same URL and port that the SSL client will be using. Or, use a non-production management system to send test requests to the non-production Integration Agent.

Notes:

- We recommend that you implement changes and test them on a non-production environment before moving them to production.
- Once you configure the Integration Agent to use HTTPS for management system communication, all APClient.bin submissions will be rejected. This is not an issue if you are sending data to the Integration Agent via HTTPS, since this does not require APClient.bin.
- SSL certificates expire and we recommend that you monitor or schedule when your certificate needs to be renewed or replaced.

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.

APClient gateway recognized parameters

The following table lists the URL parameters that the APClient Gateway recognizes:

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-plan 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>applications|sample-plan</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=applications%7Csample-plan
   &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%3Eapplications%7Csample-plan%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-plan integration:

```
<?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=applications%7Csample-plan...</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.*

Error codes returned on APXML submission failure

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

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 Linux 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.

OS exit codes APClient.bin returns to caller

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

Exit Code	Description
0	The Integration Agent accepted the connection request and HTTP POST data. 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.

Exit Code	Description
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.

Command-line parameters that APClient.bin accepts

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

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), 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).

- For more information about configuring the Integration Agent to receive notifications via HTTPS posts, see HTTPS requests.

Message submissions

The following example demonstrates how `APClient.bin` is used to generate a message submission to the sample-plan integration:

```
APClient.bin '<?xml version="1.0"?>
       <transaction id="99">
              <header>
                     <method>Add</method>
                     <subclass>Event</subclass>
              </header>
              <data>
                     <agent_client_id>applications|sample-plan</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 Linux shells and do not require modification of the APXML. However, for Windows and some Linux 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>applications|sample-plan</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-plan 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-plan integration:

```
APClient.bin --map-data-transaction-id 99 --map-data "applications|sample-plan" 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.*

Command-line parameters that APClient.bin accepts

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

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 enclosed in quotes and separated by a pipe: `"applications|sample-plan"`
`--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-plan 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 map-data parameter that contains spaces.*

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 integration service you want to target (for example, `applications|sample-plan`).
- `<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 "applications|sample-plan" bsmith "Server
down." "" TICKET-0100-0302 999 test1 test2
```

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

- applications|sample-plan
- bsmith
- Server down.
- n/a
- TICKET-0100-0302
- 999
- test1
- test2

Since the first map data value is "applications|sample-plan", the Integration Agent uses the `<mapped-input>` element from the sample-plan integration service. The sample 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-plan 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.

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-plan 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-plan 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.

Property	Value	Function
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.
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 (for example, 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 sample-plan integration service has an "inbound.applications.sample-plan.normal" queue and an "inbound.applications.sample-plan.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 applications_sample-plan has received the
following request from endpoint http://10.2.0.126:8081/http/applications_sample-plan...
(request data omitted)
[httpConnector.receiver.3] INFO - The Integration Service (applications, sample-plan) 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.applications.sample-plan.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.applications.sample-plan.normal-1] INFO - Component applications_sample-plan has
received the following request from endpoint jms://inbound.applications.sample-plan.normal...
```

For more information about the various APXML request types, see Input Action Scripting.

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 applications_sample-plan has finished processing
the request from endpoint http://10.2.0.126:8081/http/applications_sample-plan...
[inbound.applications.sample-plan.normal-1] INFO - Component applications_sample-plan has
finished processing the request from endpoint jms://inbound.applications.sample-plan.normal...
[ActiveMQ Transport: tcp:///127.0.0.1:43351] DEBUG - queue://response.applications.sample-
plan.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 Integration service scripting.

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 configured to use an external Oracle

or SQLServer DBMS, which can be shared with other Integration Agents to provide fault tolerance (in other words, if one Integration Agent stops working, another takes over responsibility for the items in the queues).

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 accepts 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\applications\sample-plan.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.

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.

Testing has shown that performance does not improve when concurrency settings are increased past 3 or 4 threads, so we do not recommend using higher values. This is due to a limitation of the ActiveMQ queueing software.

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.

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.

The 30-second duration is configurable via the <request-timeout> value in IAConfig.xml. We strongly recommend that you do not increase this setting, because long-running processes will delay any requests that are in the inbound queues.

For example, if an operation (such as a request to an external web service) is causing a script to timeout, it has the same impact on other requests in the integration's queues. Increasing the delay will cause the delays to accumulate, potentially resulting in notifications being delayed by tens of minutes. It's better to find and fix the cause of the issue causing the timeout.

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. 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.

Reserved function names

The following function names are reserved to provide specific functionality:

apia_input

- **input**: APXML, received automatically from the Integration Agent's inbound queues
- **output**: APXML, sent automatically to the Integration Agent'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.

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.

The following function names are also reserved and described in detail in Lifecycle hooks.

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

Fault tolerance in outbound REST requests

After successful processing, notification requests are removed from the Integration Agent's inbound queues and are sent to xMatters via the HTTP convenience functions (post, get, etc) provided by 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.

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 (for example, via the wsutil sendReceive() function).

Handling of callbacks from xMatters

Callbacks are messages that are sent from xMatters to the Integration Agent. There are four types of callback: "status", "deliveryStatus", "response", and ”annotation”. These messages are not sent directly to the Integration Agent; rather, they are queued in xMatters until the Integration Agent requests them. The request takes the form of a ReceiveAPXML request which is sent to xMatters every 10 – 60 seconds.

When an Integration Agent calls the `ReceiveAPXML` Web Service method, xMatters returns any 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.

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. The APXML message is then added to the targeted integration service's inbound queue. When processing resources are available, the message is sent to the apia_response function, which converts it to a JSON object and forwards it to the apia_callback function.

Deleting events

You can delete events by using an xMatters REST API request to set the status of any existing events to "Terminated". This involves the following two-part process:

- Send a request to xMatters to retreive a list of events corresponding to a specific incident ID
- Send a separate request to xMatters for each event in the list that you want to terminate.

This functionality is included in `XMUtil.terminateEvents`, defined in xmutil.js. The sample-terminate integration service (included with the Integration Agent) shows an example of the usage of this function.

Errors and retries while processing inbound queue

The processing of 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:

```
RetriableException( cause,  maxAttempts,  delayMillis);
```

Where:

- `cause` is the exception caught by the script (for example, “org.apache.http.NoHttpResponseException”). It is of type Throwable (see the description of the Throwable object class at https://docs.oracle.comfor more information).
- `maxAttempts` is the maximum number of times that the message can be (re)processed before being discarded. It is of type Integer.
- `delayMillis` is the minimum time (in milliseconds) that the message must wait before being re-processed. It is of type long.

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.javaException, 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.

Inputs

The `apia_discard` method accepts the following parameters:

```
function apia_discard(message, ex, numAttempts, firstAttemptTimestamp);
```

Where:

- Message is of type `com.alarmpoint.integrationagent.apxml.APXMLMessage` or an HTTP entity (see Using the apia_http method). It is an unmodified copy of the message that was initially retrieved from the inbound queue.
- `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_http`, and `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, 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:

```
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.

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.