



FatWire

# Content Integration Platform for EMC Documentum

Version 2.0

## Creating a Java Connector and Plug-In

**Publication Date:** Jun. 29, 2010

FATWIRE CORPORATION PROVIDES THIS SOFTWARE AND DOCUMENTATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. In no event shall FatWire be liable for any direct, indirect, incidental, special, exemplary, or consequential damages of any kind including loss of profits, loss of business, loss of use of data, interruption of business, however caused and on any theory of liability, whether in contract, strict liability or tort (including negligence or otherwise) arising in any way out of the use of this software or the documentation even if FatWire has been advised of the possibility of such damages arising from this publication. FatWire may revise this publication from time to time without notice. Some states or jurisdictions do not allow disclaimer of express or implied warranties in certain transactions; therefore, this statement may not apply to you.

Copyright © 2010 FatWire Corporation. All rights reserved.

The release described in this document may be protected by one or more U.S. patents, foreign patents or pending applications.

FatWire, FatWire Content Server, FatWire Engage, FatWire Satellite Server, CS-Desktop, CS-DocLink, Content Server Explorer, Content Server Direct, Content Server Direct Advantage, FatWire InSite, FatWire Analytics, FatWire TeamUp, FatWire Content Integration Platform, FatWire Community Server and FatWire Gadget Server are trademarks or registered trademarks of FatWire, Inc. in the United States and other countries.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. AIX, AIX 5L, WebSphere, IBM, DB2, Tivoli and other IBM products referenced herein are trademarks or registered trademarks of IBM Corporation. Microsoft, Windows, Windows Server, Active Directory, Internet Explorer, SQL Server and other Microsoft products referenced herein are trademarks or registered trademarks of Microsoft Corporation. Red Hat, Red Hat Enterprise Linux, and JBoss are registered trademarks of Red Hat, Inc. in the U.S. and other countries. Linux is a registered trademark of Linus Torvalds. SUSE and openSUSE are registered trademarks of Novell, Inc., in the United States and other countries. XenServer and Xen are trademarks or registered trademarks of Citrix in the United States and/or other countries. VMware is a registered trademark of VMware, Inc. in the United States and/or various jurisdictions. Firefox is a registered trademark of the Mozilla Foundation. UNIX is a registered trademark of The Open Group in the United States and other countries. Any other trademarks and product names used herein may be the trademarks of their respective owners.

This product includes software developed by the Indiana University Extreme! Lab. For further information please visit <http://www.extreme.indiana.edu/>.

Copyright (c) 2002 Extreme! Lab, Indiana University. All rights reserved.

This product includes software developed by the OpenSymphony Group (<http://www.opensymphony.com/>).

The OpenSymphony Group license is derived and fully compatible with the Apache Software License; see <http://www.apache.org/LICENSE.txt>.

Copyright (c) 2001-2004 The OpenSymphony Group. All rights reserved.

You may not download or otherwise export or reexport this Program, its Documentation, or any underlying information or technology except in full compliance with all United States and other applicable laws and regulations, including without limitations the United States Export Administration Act, the Trading with the Enemy Act, the International Emergency Economic Powers Act and any regulations thereunder. Any transfer of technical data outside the United States by any means, including the Internet, is an export control requirement under U.S. law. In particular, but without limitation, none of the Program, its Documentation, or underlying information of technology may be downloaded or otherwise exported or reexported (i) into (or to a national or resident, wherever located, of) any other country to which the U.S. prohibits exports of goods or technical data; or (ii) to anyone on the U.S. Treasury Department's Specially Designated Nationals List or the Table of Denial Orders issued by the Department of Commerce. By downloading or using the Program or its Documentation, you are agreeing to the foregoing and you are representing and warranting that you are not located in, under the control of, or a national or resident of any such country or on any such list or table. In addition, if the Program or Documentation is identified as Domestic Only or Not-for-Export (for example, on the box, media, in the installation process, during the download process, or in the Documentation), then except for export to Canada for use in Canada by Canadian citizens, the Program, Documentation, and any underlying information or technology may not be exported outside the United States or to any foreign entity or "foreign person" as defined by U.S. Government regulations, including without limitation, anyone who is not a citizen, national, or lawful permanent resident of the United States. By using this Program and Documentation, you are agreeing to the foregoing and you are representing and warranting that you are not a "foreign person" or under the control of a "foreign person."

*FatWire Content Integration Platform for EMC Documentum: Creating a Java Connector and Plug-In*

Publication Date: Jun. 29, 2010

Product Version 2.0

#### **FatWire Technical Support**

[www.fatwire.com/Support](http://www.fatwire.com/Support)

#### **FatWire Headquarters**

FatWire Corporation  
330 Old Country Road  
Suite 207  
Mineola, NY 11501  
[www.fatwire.com](http://www.fatwire.com)

Table of

## Contents

<b>1</b>	<b>Integrating with Custom Source Systems</b> .....	<b>5</b>
	Customizing FatWire Content Integration Platform .....	6
	Content Integration Agent .....	6
<b>2</b>	<b>Creating Connectors and Plug-Ins</b> .....	<b>9</b>
	Overview .....	10
	I. Creating a Java Source Connector .....	11
	II. Creating a Java Plug-In .....	14
	III. Enabling javafacility .....	17
	Troubleshooting and Debugging .....	18



## Chapter 1

# Integrating with Custom Source Systems

This chapter outlines methods for extending FatWire Content Integration Platform to support delivery from custom source systems to FatWire Content Server.

This chapter contains the following sections:

- [Customizing FatWire Content Integration Platform](#)
- [Content Integration Agent](#)

## Customizing FatWire Content Integration Platform

FatWire Content Integration Platform (CIP) brings content from EMC Documentum to FatWire Content Server. CIP can also be customized to deliver content from any source system to Content Server.

Developers can extend Content Integration Platform to publish from systems of their own choice, such as database or custom content management systems, by writing a Java-based implementation: a source connector and plug-in(s) or just the plug-in(s). Both the connector and the plug-ins are supported by the Content Integration Agent component (Figure 1, on page 7).

A Java source connector must be written for each source system. The connector queries the source system to retrieve its metadata and binary content. (The connector must be registered within the Content Integration Agent by the addition of a statement to the `catalog.xml` file.)

A plug-in is required only if items retrieved by the connector must be processed before they are published to Content Server. Processing an item could include for example, extracting thumbnails from image files or performing a validation step while publishing. Typically, plug-ins are written to support different file formats or to filter selected items from the publishing process. Any number of plug-ins can be used with *any* connector. Like the connector, a plug-in must be registered with the Content Integration Agent (in the `types.xml` file).

## Content Integration Agent

Content Integration Agent is written in C++ and provides the following components to support Java-based custom connectors and plug-ins:

- Solid runtime system.
- Pluggable interfaces, used to implement Java-based source connectors and plug-ins.
- XML files named `catalog.xml` and `types.xml`, both used to register the custom source connector and plug-ins.
- Native source connector (`javaconnector` library) and native plug-in (`javaplugin` library). Both are written in C++. They are used to make calls to Java code.
- Facilities, which are construction blocks providing some set of functionality to the Agent runtime. Content Integration Agent hosts the Java Virtual Machine in its process space in order to delegate calls from the C++ runtime environment to Java code. The JVM is enabled by registering `javafacility` in `facilities.xml`.

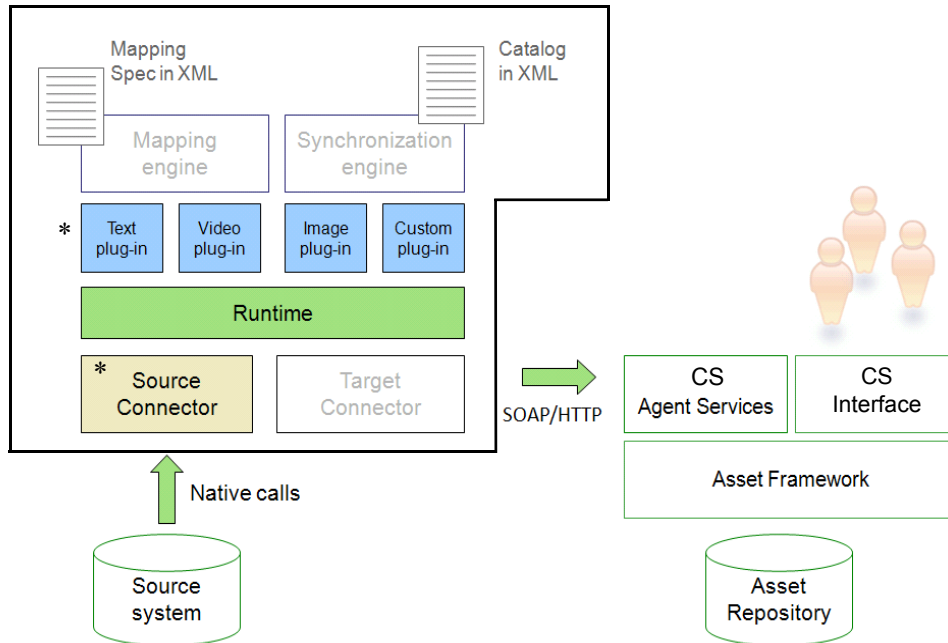
Once the Java-based connector is created and the JVM is enabled, the C++ Agent runtime system can use the JVM to call Java code via the native connector (similar process for plug-ins). For system architecture, see Figure 1B, on page 7.

Procedures for creating Java-based connectors and plug-ins are given in chapter 2, along with instructions for completing the integration.

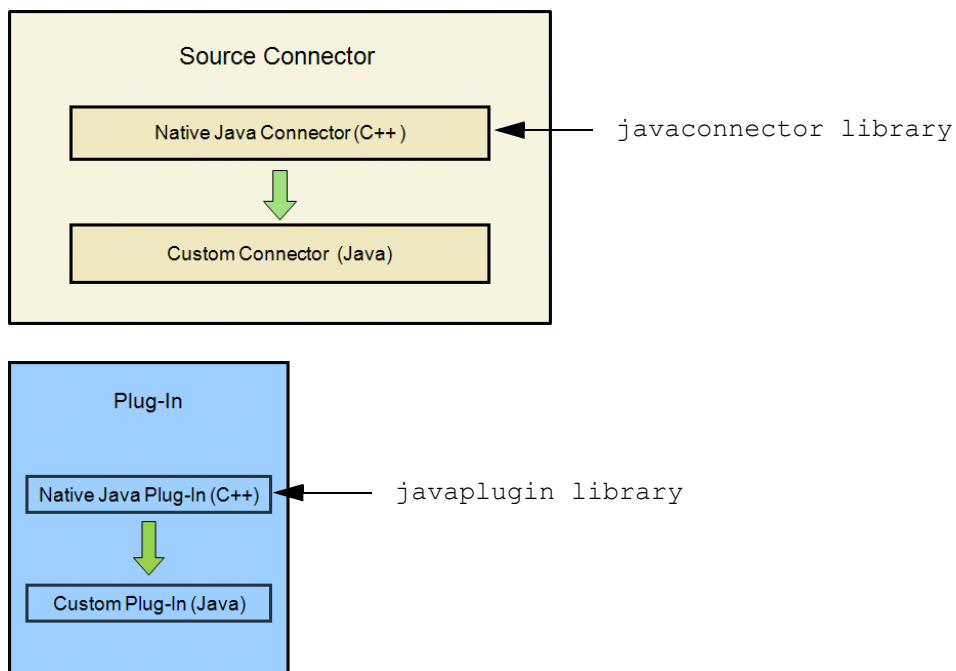
More information about Content Integration Agent can be found in the *CIP 2.0 for EMC Documentum Administrator's Guide*, available on our e-docs site at <http://support.fatwire.com>.

**Figure 1: Content Integration Platform**

**A. Content Integration Agent**



**B. Source Connector and Plug-In**







## Chapter 2

# Creating Connectors and Plug-Ins

This chapter provides instructions for creating a complete integration solution to support content delivery from any source system to FatWire Content Server.

This chapter contains the following sections:

- [Overview](#)
- [I. Creating a Java Source Connector](#)
- [II. Creating a Java Plug-In](#)
- [III. Enabling javafacility](#)
- [Troubleshooting and Debugging](#)

## Overview

Creating a connector and plug-in involves the following steps:

1. Implementing the pluggable interfaces that are provided within Content Integration Agent.
2. Registering the implementation(s) with the Content Integration Agent runtime system.
3. Registering `javafacility` in order to enable the Java Virtual Machine to delegate calls from the C++ Agent runtime to Java code.

### Note

A custom plug-in can be used with *any* connector. You can implement and deploy as many plug-ins as necessary.

Before a custom connector (or plug-in) can be successfully used, the data model for the publishable objects must exist on the Content Server system and be mapped to the Content Server system. The following steps are required:

1. Reproduce the objects' metadata in Content Server by creating a dedicated flex family (or re-using an existing flex family) to store the object types, their attributes, and the objects themselves.
2. Map object types and attributes to their respective flex family asset instances (created in the previous step). The map can be created directly in the connector implementation, or in the `mappings.xml` file.

## I. Creating a Java Source Connector

Publishing from an unsupported source system to Content Server requires you to create a Java-based source connector. (A plug-in is not required unless objects retrieved by the connector must be processed before they are published.)

### Note

If you are using a relational database, implement custom views or custom queries in order for the connector to work.

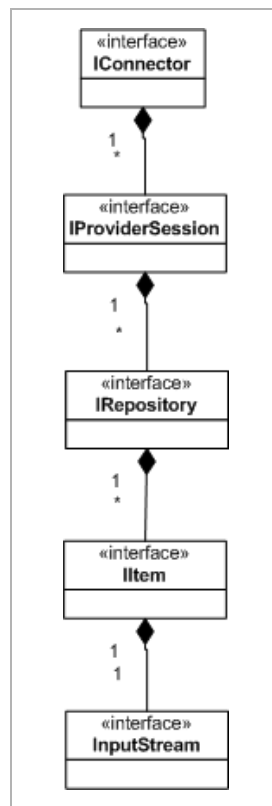
### To create a Java source connector

#### 1. Implement the connector:

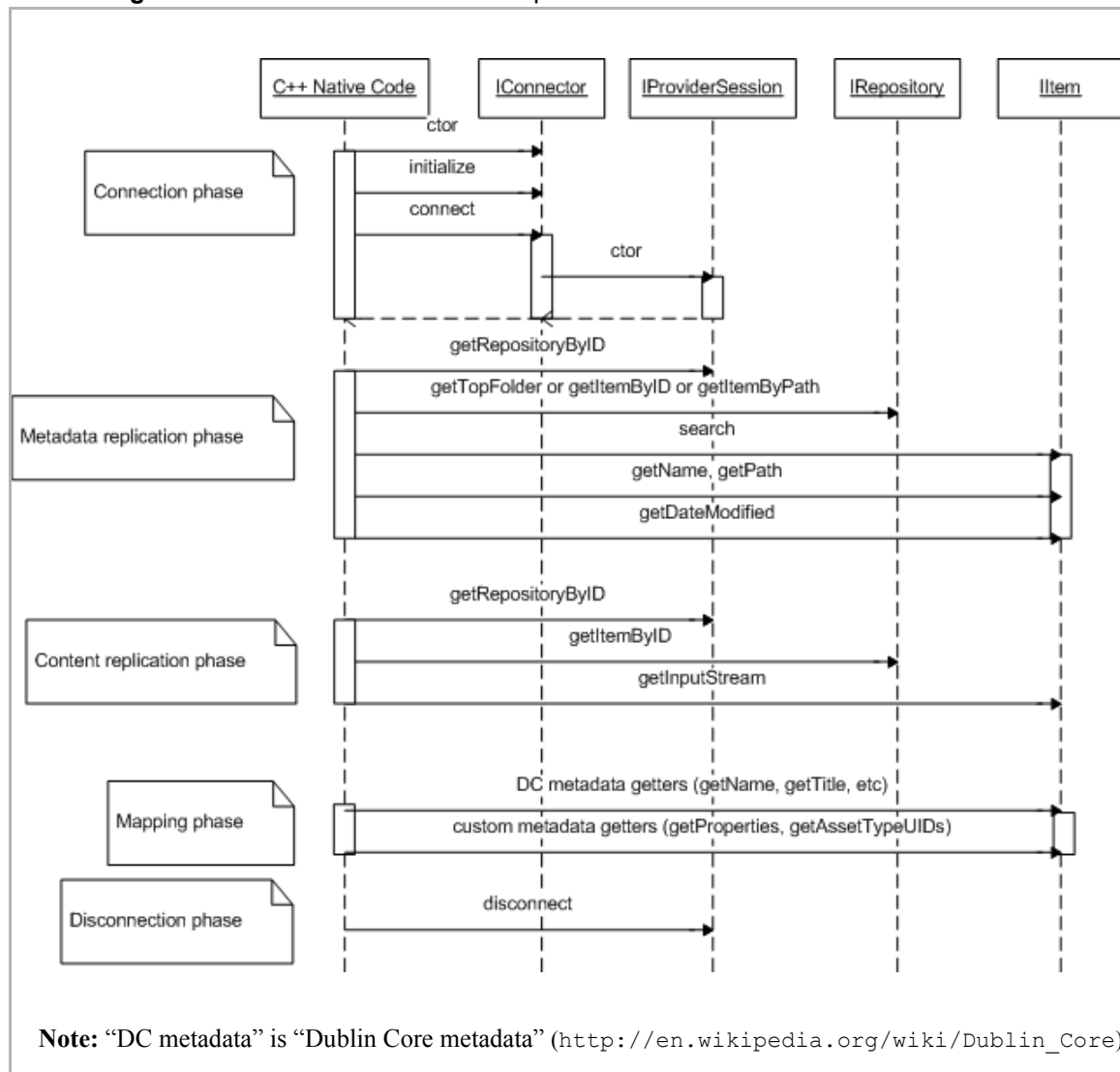
Implement the `IConnector`, `IProviderSession`, `IRepository`, and `IItem` interfaces. You can optionally implement the `InputStream` interface if items on your source system have primary binary content.

[Figure 2](#) shows the relationships among the interfaces. The entry point for the connector's code is a factory class: the `IConnector` interface implementation.

**Figure 2:** Connector and plug-in class diagram



There are different phases in a connector's lifetime. Depending on the phase, different methods are invoked. [Figure 3](#) shows the sequence of calls during each phase.

**Figure 3:** Source connector calls sequence**Analyzing Figure 3: Source connector calls sequence**

The ID, which is passed to the `getRepositoryById` function, is taken from one of the corresponding workspace elements in the `catalog.xml` file.

Depending on what you pass to the `cipcommander`, one of the following functions is invoked:

- If `-source_itemid` is passed, then `getItemById` is invoked passing the `itemid`.
- If `-source_itemid` is omitted, and `-source_path` is specified, then the `getItemByPath` function is invoked.
- If neither `-source_itemid` or `-source_path` is specified, then the `getTopFolder` function is invoked. (In this case, the entire repository is published.)

To ensure uniqueness, maintain a different `versionid`, `itemid`, and `path` for all items inside the same repository, and keep the names different for all items inside the same folder. The `path` must be in the form: `<parent path>/<this item name>`.

## 2. Register the connector:

- a. Register the `IConnector` interface implementation with Content Integration Agent by adding a 'connector' element to `catalog.xml` (located in `integration_agent/conf/`):

```
<connector id="connector_id"
  name="connector_descriptive_name">
  <library>javaconnector</library>
  <init-params>
    <param name="className">connector_class_name</param>
    connector-specific_parameters
  </init-params>
</connector>
```

Parameter	Description
<code>connector_id</code>	Any unique identifier.
<code>connector_descriptive_name</code>	Any descriptive name (does not have to be unique).
<code>connector_class_name</code>	Name of the <code>IConnector</code> implementation created.
<code>connector-specific_parameters</code>	Set of parameters that will be passed to <code>IConnector.initialize</code> during the call.

- b. Enable publishing by adding a new 'provider' element to `catalog.xml`:

```
<provider id="provider_id" name="provider_descriptive_name">
  <connector-ref refid="connector_id"/>
  <init-params/>
  provider-specific_parameters
</init-params>
</ provider >
```

Parameter	Description
<code>provider_id</code>	Any unique identifier.
<code>provider_descriptive_name</code>	Any descriptive name (doesn't have to be unique).
<code>connector_id</code>	Connector's unique identifier.
<code>provider-specific_parameters</code>	Set of parameters that will be passed to <code>IConnector.login</code> during the call.

c. Deploy the connector:

Place the connector's `jar` files into the folder `<resource>/java/<connector_id>/lib`, and the `class` files into `<resource>/java/<connector_id>/classes`.

The `<resource>` folder is located within Content Integration Agent.

**On Windows:** `<resource>` is `<%INSTALLDIR%>`

**On Unix:** `<resource>` is `<$INSTALLDIR/shared/cipagent>`

### Note

Connector classes are loaded by different class loaders to prevent collisions with different implementations and loading/unloading features. We strongly advise placing all connector `jar` and `class` files into the `<connector_id>` folder, instead of including them into the `CLASSPATH` environment variable, or the `java.class.path` property, or the `jre/lib/ext` folder.

3. If you require a Java plug-in (to process items retrieved by the connector), continue to section “[II. Creating a Java Plug-In](#).” Otherwise, enable `javafacility` (to allow the Java Virtual Machine to delegate calls to Java code from the C++ Agent runtime). For instructions, see “[III. Enabling javafacility](#),” on page 17.

## II. Creating a Java Plug-In

A plug-in is not required unless objects retrieved by the connector must be processed before they are published to the Content Server system. The main purpose of a plug-in is to modify the metadata of retrieved items, add metadata to retrieved items, and reject items.

### Note

A custom plug-in can be used with *any* connector. You can create and deploy as many plug-ins as necessary.

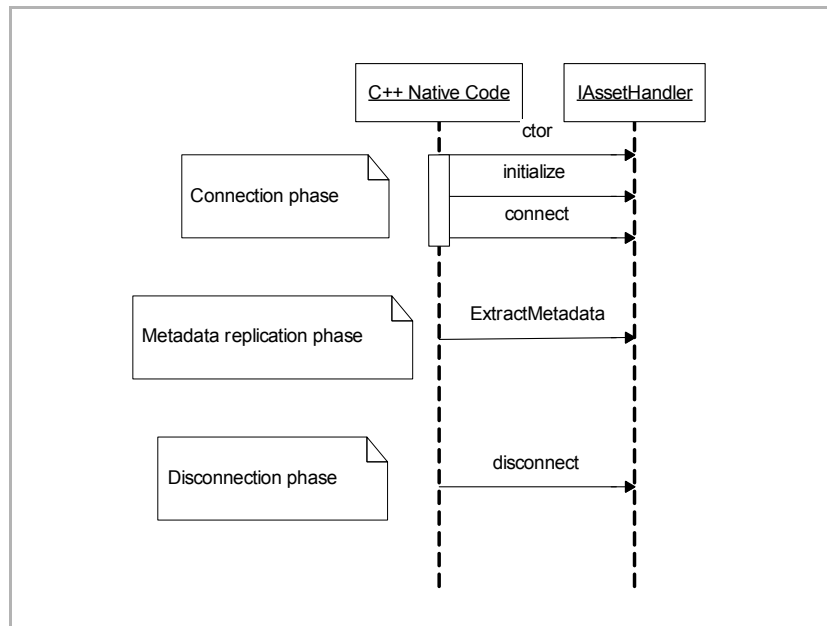
Creating a plug-in is similar to creating a connector. The steps are as follows:

#### To create a Java plug-in

1. Implement the plug-in by implementing the `IAssetHandler` interface (in Content Integration Agent).



The entry point for a plug-in is the `IAssetHandler` interface. This is the only interface which is directly used by the runtime system. In most cases `ExtractMetadata` is the only method you need to implement. [Figure 4](#) shows the calls sequence in a plug-in's lifetime.

**Figure 4:** Plug-in calls sequence

## 2. Register the plug-in with Content Integration Agent.

- a. Add a new plug-in 'handler' element to the `types.xml` file (located in the `integration_agent/conf/` folder):

```

<handler id="handler_id">
  <library>javaplugin</library>
  <init-params/>
    plugin-specific_parameters
  </init-params>
</handler>
  
```

Parameter	Description
<code>handler_id</code>	Custom plug-in's unique identifier.
<code>plugin-specific parameters</code>	Plugin-specific parameters that are passed when the plug-in is initialized.

- b. Enable the handler for the selected handler sets. Which handler set to use is specified during the publication initiation process. Each handler set contains the list of handlers, which are invoked during the metadata extraction phase in the Integration Agent. Handlers are matched by either MIME type or asset type.

MIME type has the following form: `<major type>/<minor type>` (image/jpeg, for example). There is an option to use '\*' for MIME types. It can be applied either to the `minor` part or the whole MIME type. For example, `*/*` matches all assets, while `text/*` matches only text files.

When using the `IConnectorContext.guessMIMEType` function, it will look into `mimetypes.xml` to get the corresponding MIME type for the supplied file

extension. For example the call with "txt" parameter will produce the "text/plain" result.

Asset types also support the '\*' notation, which matches all asset types.

If more than one handler matches a specific item, then both are invoked in the same sequence in which they are registered within the handler set used. If any of the matching handlers returns the null object from the `IItem.extractMetadata` call, then the item is discarded from future processing and not sent to the target connector.

- c. Enable the custom plug-in for selected object types by adding "asset-type" elements to the `types.xml` file. Items for which this plug-in is invoked will be filtered according to MIME type.

### Note

The `asset-type` element in the context of a plug-in is a MIME type group.

```
<asset-type type="MIME_type">
  <extensions>
    <ext>ext</ext>
    ...
  </extensions>
  <handler-ref refid="handler_id" />
</asset-type>
```

Parameter	Description
MIME type	MIME type of the item for which this plug-in will be invoked. <i>MIMEtype</i> must be of the form <code>&lt;major_type/minor_type&gt;</code> , e.g., <code>text/plain</code> . A wild-card symbol (*) can also be used. For example: <ul style="list-style-type: none"> <li>• To enable the plug-in for all text files, specify: <code>text/*</code></li> <li>• To enable the plug-in for all items, specify: <code>*/*</code></li> </ul>
ext	File extension, e.g., <code>.txt</code> for text files. The file extension does not directly affect the plug-in selection process. However, it is used to "guess" the MIME type for those systems where MIME type is not directly available (e.g., file system).
handler_id	Custom plug-in's unique identifier (specified in the <code>handler</code> element, in the previous step).

- d. Deploy the plug-in:

Place the plug-in's jar files into the folder `<resource>/java/<plugin_id>/lib`, and the class files into `<resource>/java/<plugin_id>/classes`.

The `<resource>` folder is located within Content Integration Agent.

**On Windows:** `<resource>` is `<%INSTALLDIR%>`



**On Unix:** <resource> is <\${INSTALLDIR}/shared/cipagent>

### Note

Plug-in classes are loaded by different class loaders to prevent collisions with different implementations and loading/unloading features. We strongly advise placing all plug-in jar and class files into the <plugin\_id> folder, instead of including them into the CLASSPATH environment variable, or the java.class.path property, or the jre/lib/ext folder.

3. If you created a custom connector but have not enabled javafacility, continue to the next section, “[III. Enabling javafacility.](#)”

## III. Enabling javafacility

Calling Java code from C++ Agent runtime requires a special facility named java to be registered in facilities.xml.

### To enable javafacility

1. Make sure facilities.xml is not commented (facilities.xml is located in the integration\_agent/conf/ folder).
2. Add the following lines:

```
<facility name="javafacility">
  <library>java</library>
  <init-params>
    <param name="VMArgparam_id">Java_VM_argument
      </param>
    <param name="VMLibraryPath">VM_library_path</param>
  </init-params>
</facility>
```

Parameter	Description
param_id	Parameter's unique id (any unique value). In order to pass multiple arguments to the JVM, construct multiple parameters with different <i>param_id</i> 's.
Java_VM_argument	Java VM argument to be passed to the Java VM created within the Agent runtime process. <b>Example:</b> <param name="VMArg0">-Xmx256m</param>
VM_library_path	Full path to the Java VM library (DLL or shared library) within the JRE/JDK installation. For example, for Sun JDK on Windows, VM_library_path is either: %JAVA_HOME%\jre\bin\server\jvm.dll - or - %JAVA_HOME%\jre\bin\client\jvm.dll

## Troubleshooting and Debugging

When developing custom components for CIP, it is often helpful to see more than just the default logging messages displayed in the production environment. Therefore, CIP Agent supports five different logging levels:

- fatal
- error
- warning
- info
- debug

Use the instructions below to debug custom components in CIP.

### Note

Do not use the settings shown below on a production system, as they can slow down the system's performance.

- **Escalating the logging level in CIP Agent**

CIP is set to `error` by default. To increase the logging level, CIP Agent must run as a console executable:

1. Stop the CIP Agent system service.
2. Run the `cipagent -t debug` command.

- **Debugging Java custom components**

To debug custom Java implementations hosted within the Agent runtime, enable remote debugging in CIP Agent. For example, to start the remote debugger on port 7007 and suspend CIP Agent to wait until a debugger attaches, add the following lines to `javafacility`:

```
<param name="VMArg1">-Xdebug</param>
<param name="VMArg2">-Xrunjdwp:transport=dt_socket,
  address=7007,server=y,suspend=y</param>
```

- **Escalating the logging level for CS Agent Services**

To get more data about an error in the CS Agent Services application, set the `DEBUG` level in the `commons-logging.properties` file for the `com.fatwire.logging.csagentservices` category. We also recommend setting the `DEBUG` logging level in the `commons-logging.properties` file for the `com.fatwire.logging.cs.db` category.