FatWire

Content Integration Platform

Version 1.5

Creating a Java Connector and Plug-In

Publication Date: Jan. 12, 2010

FatWire® SOFTWARE

FatWire Content Integration Platform: Creating a Java Connector and Plug-In
Publication Date: Jan. 12, 2010
Product Version 1.5

FatWire Technical Support
www.fatwire.com/Support

FatWire Headquarters
FatWire Corporation
330 Old Country Road
Suite 207
Mineola, NY 11501
www.fatwire.com

Table of

Contents

Chapter 1

Integrating with Custom Source Systems

This chapter outlines how developers can extend FatWire Content Integration Platform to support publishing from custom source systems to FatWire Content Server and FatWire TeamUp.

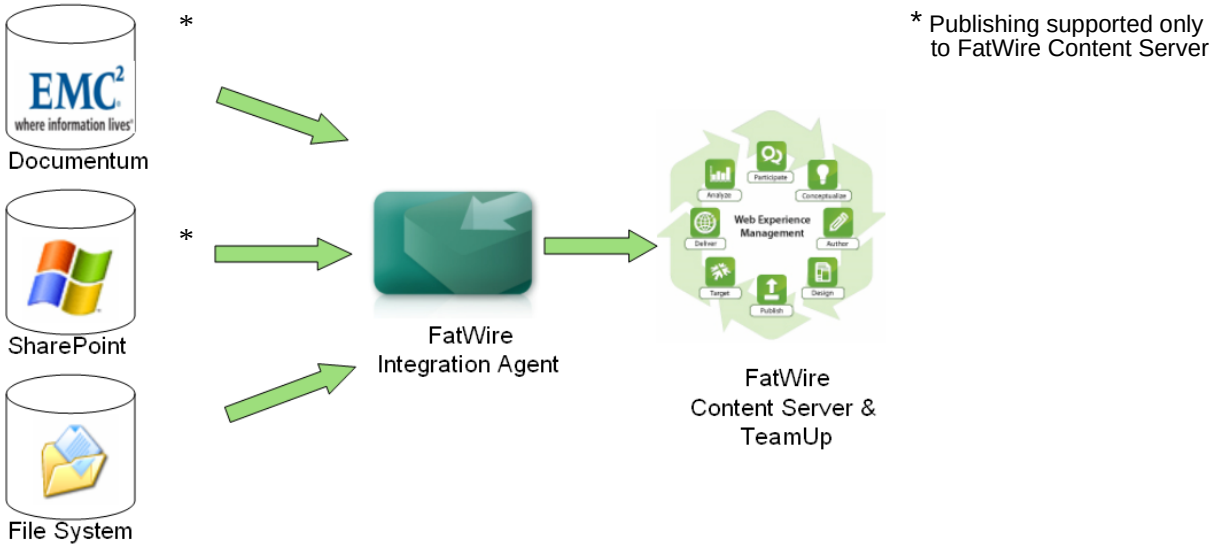
This chapter contains the following sections:

Overview

FatWire Content Integration Platform (CIP) brings content from various source systems to FatWire Content Server and FatWire TeamUp. Publishing from the following systems is supported out-of-the-box: Documentum Content Server, Microsoft SharePoint, and file systems on Unix and Windows (Figure 1). Default integration solutions are listed in the *Supported Platform Document* (at `http://e-docs.fatwire.com`).

While Content Integration Platform provides out-of-the box integration solutions for commonly used source systems, its true value is its applicability to any source system.

Figure 1: CIP architecture



Customizing FatWire Content Integration Platform

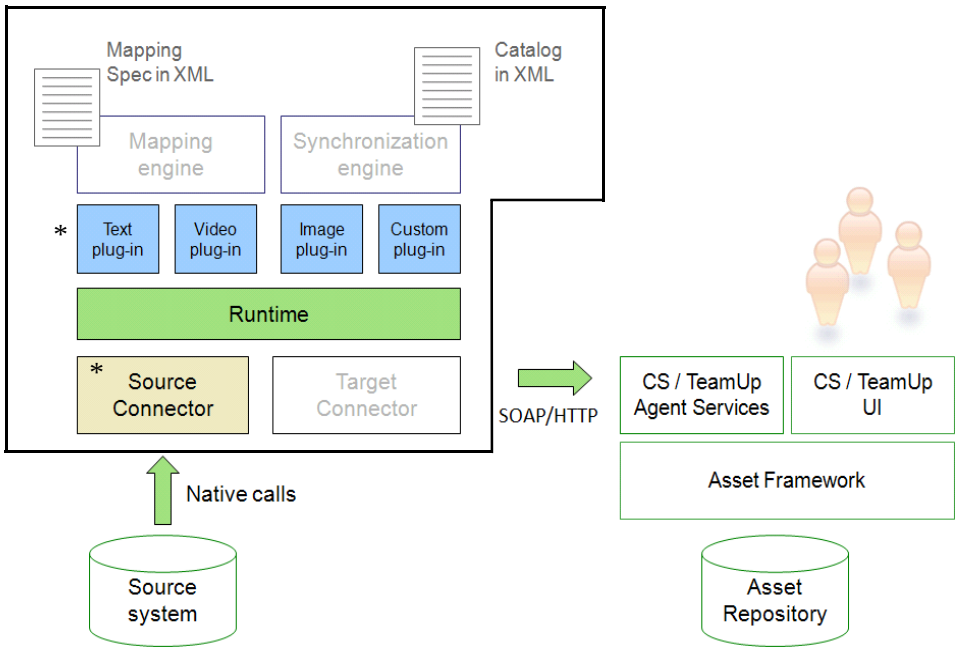
Developers can extend Content Integration Platform to publish from systems of their own choice (such as a database or custom content management system) by writing a Java-based implementation: a source connector and plug-in(s), or just the plug-in(s). Both the connector and the plug-ins are supported by the Content Integration Agent component (Figure 2, on page 7. See also "Content Integration Agent," on page 8).

A Java source connector must be written for each source system whose content will be published to the target system (FatWire Content Server or TeamUp). The connector queries the source system in order 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.)

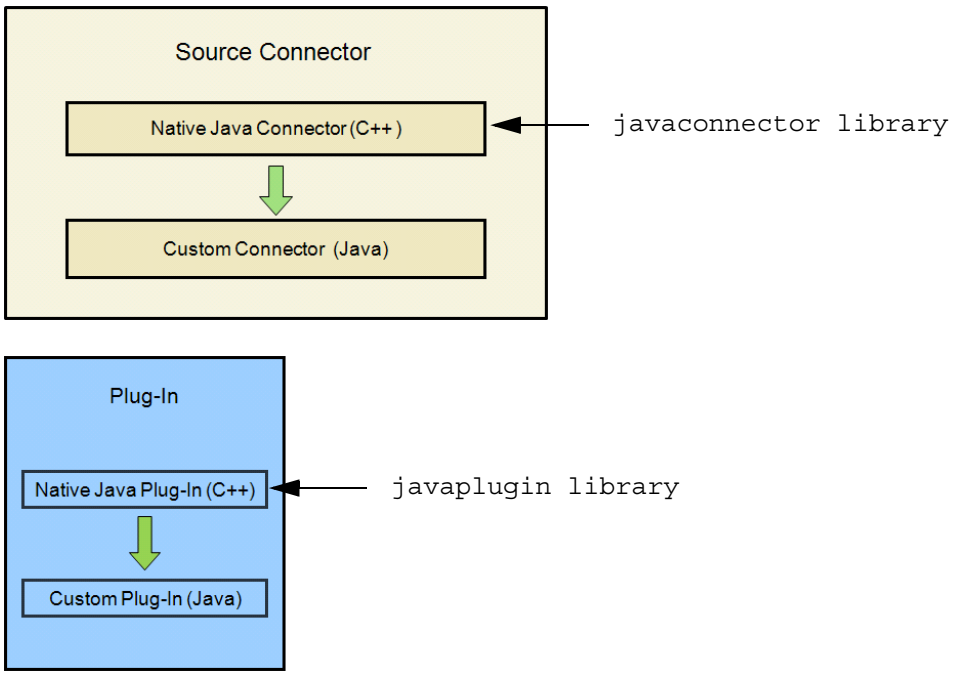
A plug-in is required only if items retrieved by the connector must be processed before they are published to the target system. Processing an item could include for example, extracting thumbnails from image files or performing a validation step while publishing. Typically, plug-ins are written to support 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 within the Content Integration Agent (in the `types.xml` file).

Figure 2: Content Integration Platform

A. Content Integration Agent



B. Source Connector and Plug-In



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 2B, on page 7.

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

Chapter 2

Creating Connectors and Plug-Ins

This chapter provides instructions for creating a complete integration solution to support publishing from custom source systems to FatWire Content Server and FatWire TeamUp.

This chapter contains the following sections:

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 target system and be mapped to the target system. The following steps are required:

- If the target system is FatWire Content Server:
  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.
- If the target system is FatWire TeamUp, map object types and attributes to their respective TeamUp instances. 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 (or TeamUp) 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.) For the list of currently supported source systems, see the *Supported Platform Document* for CIP 1.5 (available at `http://e-docs.fatwire.com`).

Note

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

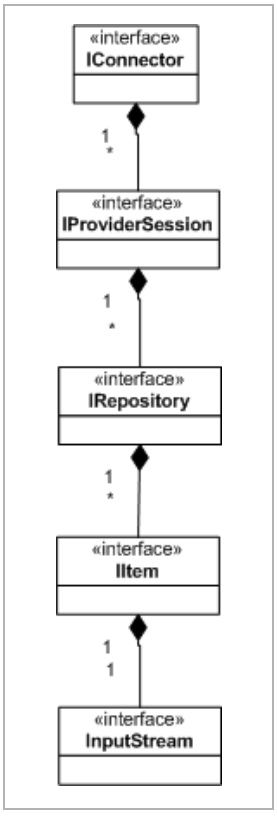
To create a Java source connector

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.

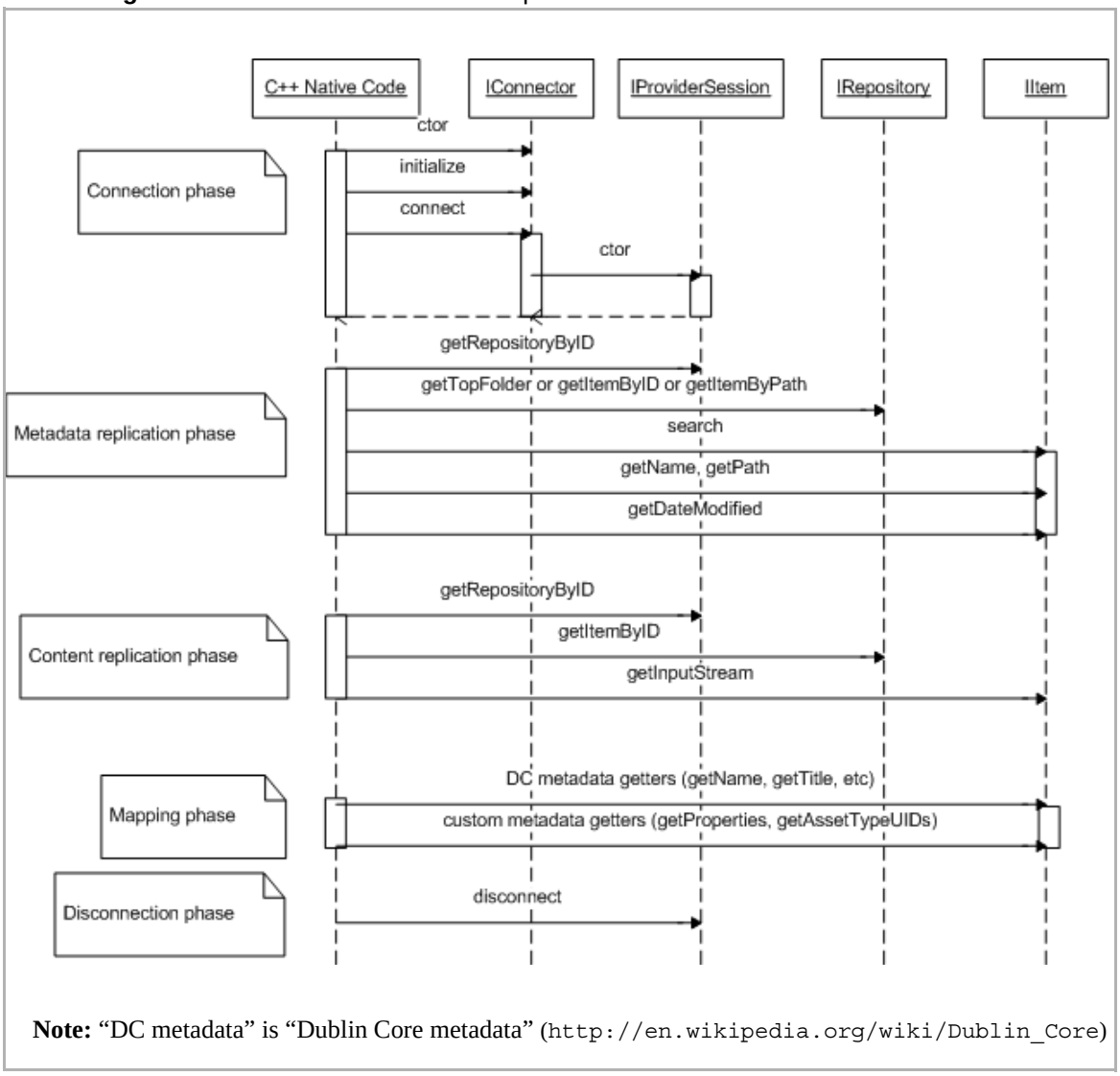
   Figure 3 shows the relationships among the interfaces. The entry point for the connector's code is a factory class: the `IConnector` interface implementation.

   Figure 3: Connector and plug-in class diagram



There are different phases in a connector's lifetime. Depending on the phase, different methods are invoked. Figure 4 shows the sequence of calls during each phase.

Figure 4: Source connector calls sequence



Note: "DC metadata" is "Dublin Core metadata" (`http://en.wikipedia.org/wiki/Dublin_Core`)

Analyzing Figure 4

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
`connector_id`	Any unique identifier.
`connector_descriptive_name`	Any descriptive name (does not have to be unique).
`connector_class_name`	Name of the `IConnector` implementation created.
`connector-specific_parameters`	Set of parameters that will be passed to `IConnector.initialize` 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
`provider_id`	Any unique identifier.
`provider_descriptive_name`	Any descriptive name (doesn't have to be unique).
`connector_id`	Connector's unique identifier.
`provider-specific_parameters`	Set of parameters that will be passed to `IConnector.login` 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 target 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.

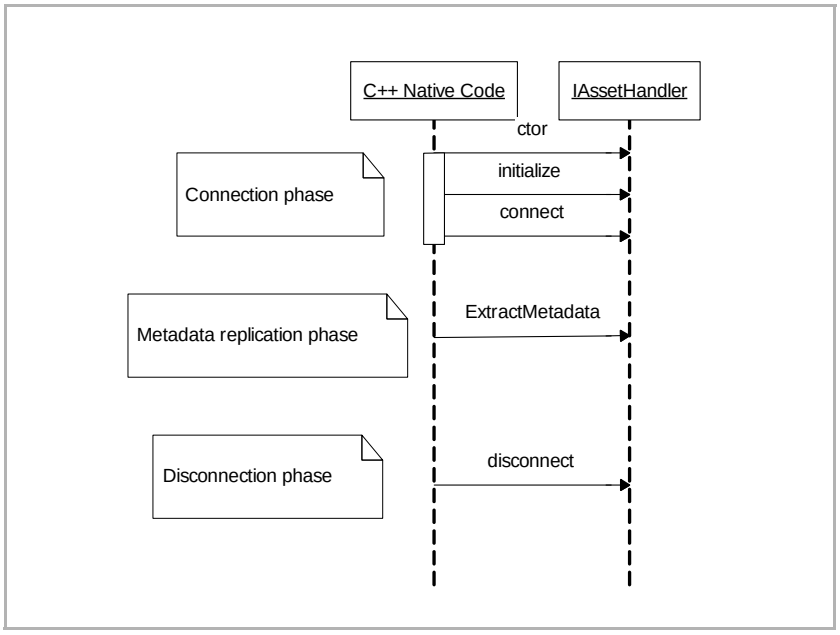
Creating a plug-in is similar to creating a connector. The steps are as follows:

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 5 shows the calls sequence in a plug-in's lifetime.

Figure 5: 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
`handler_id`	Custom plug-in's unique identifier.
`plugin-specific parameters`	Plugin-specific parameters that are passed when the plug-in is initialized.

   b. 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. *`MIMEtype`* must be of the form `<major_type/minor_type>`, e.g., `text/plain`. A wild-card symbol (`*`) can also be used. For example: • To enable the plug-in for all text files, specify: `text/*` • To enable the plug-in for all items, specify: `*/*`
`ext`	File extension, e.g., `.txt` 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 `handler` element, in the previous step).

c. 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 *`param_id`*'s.
`Java_VM_argument`	Java VM argument to be passed to the Java VM created within the Agent runtime process. **Example:** `<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`

IV. Completing the Integration

Complete one of the following procedures, depending on whether you are integrating with Content Server or TeamUp.

If you are integrating with Content Server

1. Reproduce the publishable objects' data model in Content Server:

   Create (or re-use) a flex family to store the object type definitions, the attributes, and the objects, themselves. (Typically, the flex family is named after the source system.)

   Examples of flex families are available in the *Content Integration Platform Administrator's Guide* (see "Flex Family Specifications" in any one of the appendices). Instructions for creating flex families are available in the *Content Server Developer's Guide*.

2. Map the object types and attributes to their respective instances in the flex family. Your options are to create the map directly in the connector implementation, or to customize the `mappings.xml` file. Information about mapping is available in the *Content Integration Platform Administrator's Guide*.

3. Test your implementation. For instructions on publishing to FatWire Content Server, refer to the *Content Integration Platform Administrator's Guide.*

If you are integrating with TeamUp

1. Map the object types and attributes to their respective TeamUp instances. Your options are to create the map directly in the connector implementation, or to customize the `mappings.xml` file. Information about mapping is available in the *Content Integration Platform Administrator's Guide*.

2. Test your implementation. For instructions on publishing to FatWire Content Server, refer to the *Content Integration Platform Administrator's Guide.*

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.