Oracle® WebCenter Sites

Developing a Java Adapter and Plug-In
for Content Integration Platform

11*g* Release 1 (11.1.1)

February 2012

Oracle® WebCenter Sites: Developing a Java Adapter and Plug-In for Content Integration Platform,
11*g* Release 1 (11.1.1)

Primary Author: Tatiana Kolubayev

Contributor: Valentin Vakar, Chandrashekar Avadhani, Suman Saha

Table of Contents

About This Guide

This guide shows developers how to extend Oracle WebCenter Sites: Content Integration Platform to publish from systems of their own choice to Oracle WebCenter Sites.

Applications discussed in this guide are former FatWire products. Naming conventions are the following:

- *Oracle WebCenter Sites* is the current name of the product previously known as *FatWire Content Server*. In this guide, *Oracle WebCenter Sites* is also called *WebCenter Sites*.
- *Oracle WebCenter Sites: Content Integration Platform* is the current name of the application previously known as *FatWire Content Integration Platform*. In this guide, *Oracle WebCenter Sites: Content Integration Platform* is also called *Content Integration Platform*, or *CIP*.

Content Integration Platform integrates with Oracle WebCenter Sites according to specifications in the *Oracle WebCenter Sites 11g Release 1 (11.1.1.x) Certification Matrix*. For additional information, see the release notes for Content Integration Platform. Check the WebCenter Sites documentation site regularly for updates to the *Certification Matrix* and release notes.

Audience

Users of this guide must be Java developers with proficiency in the C++ programming language.

Related Documents

For more information, see the following documents:

- *Oracle WebCenter Sites Administrator's Guide for Content Integration Platform for File Systems and Microsoft SharePoint*
- *Oracle WebCenter Sites Administrator's Guide for Content Integration Platform for EMC Documentum*

Conventions

The following text conventions are used in this guide:

- **Boldface** type indicates graphical user interface elements that you select.
- *Italic* type indicates book titles, emphasis, or variables for which you supply particular values.
- `Monospace` type indicates file names, URLs, sample code, or text that appears on the screen.
- **`Monospace bold`** type indicates a command.

Third-Party Libraries

Oracle WebCenter Sites and its applications include third-party libraries. For additional information, see *Oracle WebCenter Sites 11gR1: Third-Party Licenses*.

Chapter 1

Integrating with Custom Source Systems

This chapter outlines methods for extending Oracle WebCenter Sites: Content Integration Platform to support content delivery from custom source systems to Oracle WebCenter Sites.

This chapter contains the following sections:

Customizing WebCenter Sites: Content Integration Platform

Content Integration Platform (CIP), by default, delivers content from file systems, Microsoft SharePoint, and EMC Documentum to Oracle WebCenter Sites. Developers can extend Content Integration Platform to publish from a system of their own choice, such as a database or custom content management system, by writing a Java-based implementation consisting of a source adapter and plug-in(s), or just the adapter. Both the adapter and the plug-ins are supported by the Content Integration Agent component (Figure 1, on page 9).

A Java source adapter must be written for each source system whose content will be delivered to Oracle WebCenter Sites. The adapter queries the source system to retrieve its metadata and binary content. (The adapter must be registered with Content Integration Agent by means of a statement in the `catalog.xml` file.)

A plug-in is required only if items retrieved by the adapter must be processed before they are published to WebCenter Sites. Processing 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* adapter. Like the adapter, 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 adapter and plug-ins.
- Native source adapter (`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`.

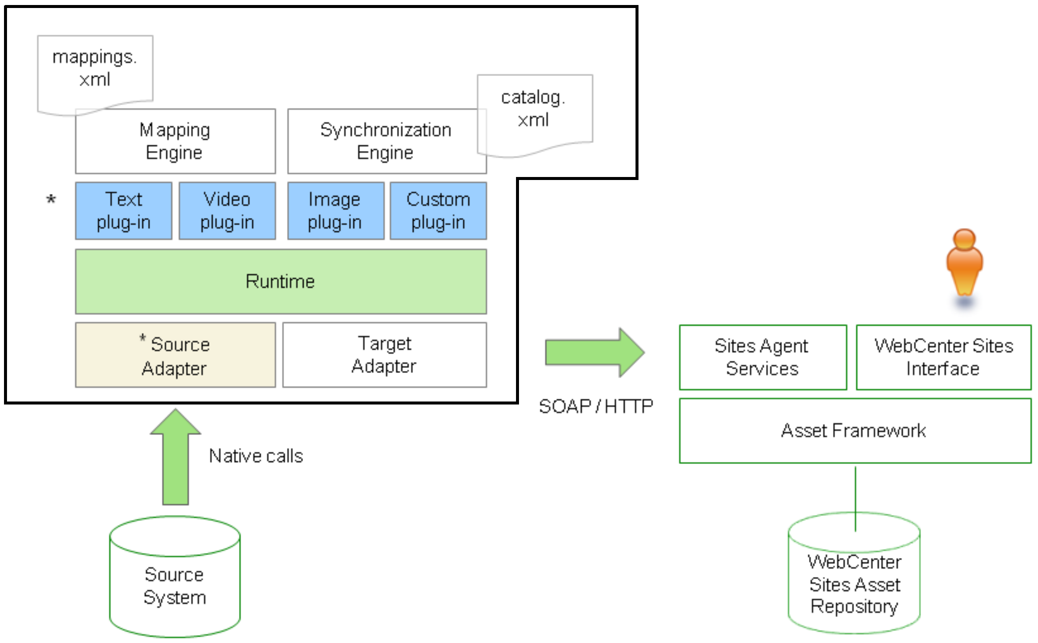
Once the Java-based adapter is created and the JVM is enabled, the C++ Agent runtime system can use the JVM to call Java code via the native adapter (similar process for plug-ins). For system architecture, see Figure 1B, on page 9.

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 following guides:

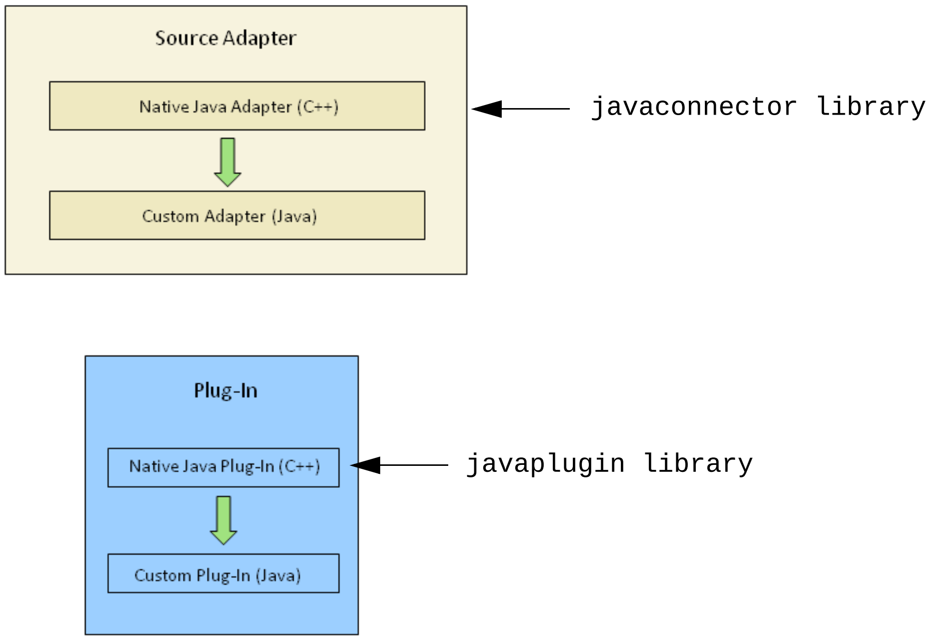
- *Oracle WebCenter Sites Administrator's Guide for Content Integration Platform for File Systems and Microsoft SharePoint*
- *Oracle WebCenter Sites Administrator's Guide for Content Integration Platform for EMC Documentum*

Figure 1: Content Integration Platform

A. Content Integration Agent



B. Source Adapter and Plug-In



Chapter 2

Creating Adapters and Plug-Ins

This chapter provides instructions for creating a complete integration solution to support content delivery from any source system to WebCenter Sites.

This chapter contains the following sections:

Overview

Creating an adapter 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* adapter. You can implement and deploy as many plug-ins as necessary.

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

1. Reproduce the objects' metadata in WebCenter Sites 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 adapter implementation, or in the `mappings.xml` file.

I. Creating a Java Source Adapter

Publishing from an unsupported source system to WebCenter Sites requires you to create a Java-based source adapter. (A plug-in is not required unless objects retrieved by the adapter 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 adapter to work.

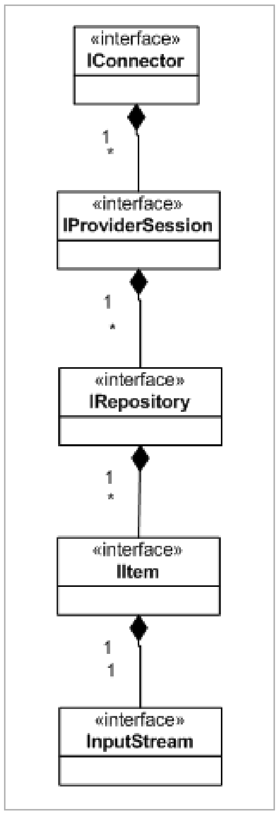
To create a Java source adapter

1. Implement the adapter:

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.

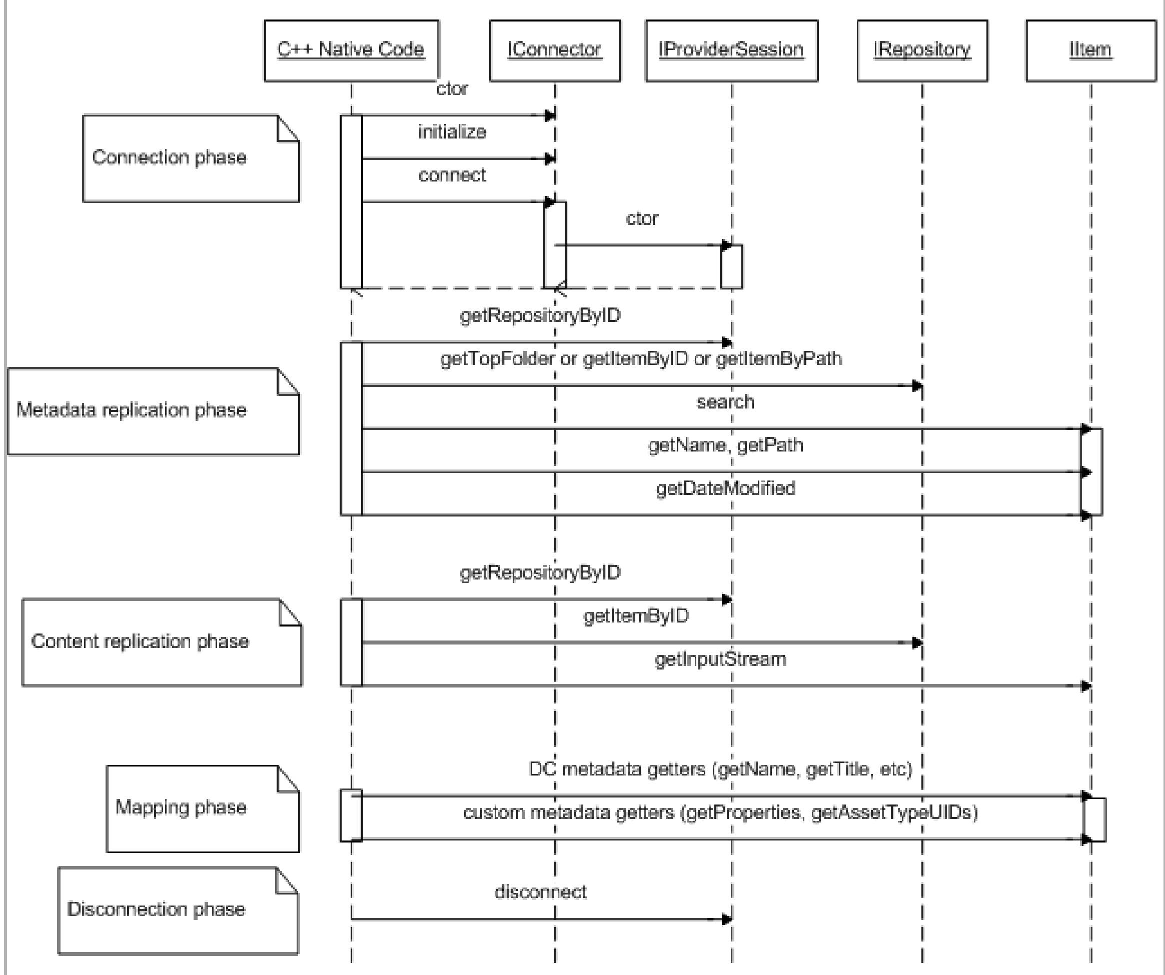
Figure 2 shows the relationships among the interfaces. The entry point for the adapter's code is a factory class: the `IConnector` interface implementation.

Figure 2: Adapter and plug-in class diagram



There are different phases in an adapter's lifetime. Depending on the phase, different methods are invoked. Figure 3 shows the sequence of calls during each phase.

Figure 3: Source adapter calls sequence



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

Analyzing Figure 3**:** Source adapter 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 adapter:

   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`	Adapter's unique identifier.
`provider-specific_parameters`	Set of parameters that will be passed to `IConnector.login` during the call.

c. Deploy the adapter:

Place the adapter'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

Adapter classes are loaded by different class loaders to prevent collisions with different implementations and loading/unloading features. We strongly advise placing all adapter `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 adapter), 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 19.

II. Creating a Java Plug-In

A plug-in is not required unless objects retrieved by the adapter must be processed before they are published to the WebCenter Sites 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* adapter. You can create and deploy as many plug-ins as necessary.

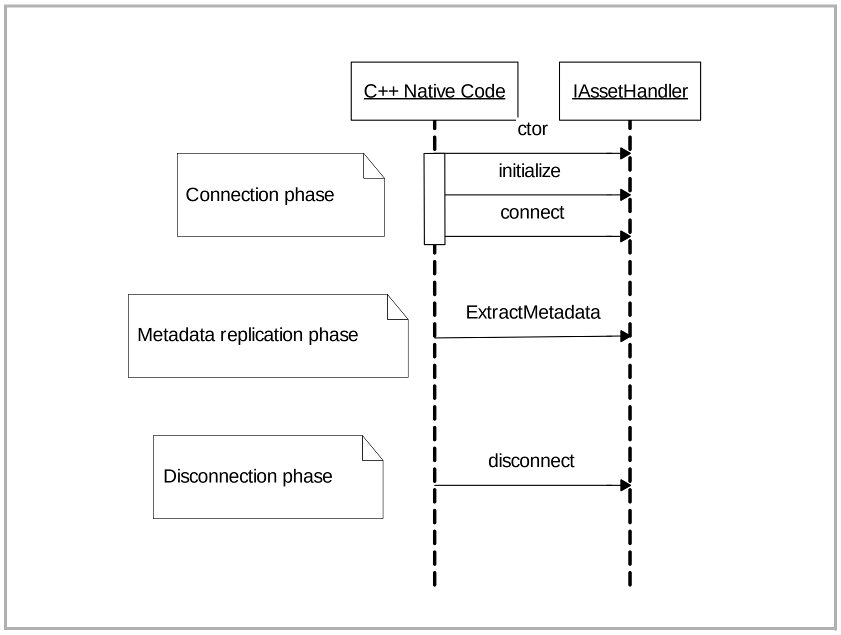
Creating a plug-in is similar to creating a adapter. The steps are as follows:

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

b. If you are using Content Interation Platform for EMC Documentum, complete this step. Otherwise, skip to step c on page 18.

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 Content 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 then 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.extracMetadata` call, then the item is discarded from future processing and not sent to the target adapter.

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. *`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).

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 adapter 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. Verify that `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>`

Parameter	Description

`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. 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 Sites Agent Services**

  To get more data about an error in the Sites 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.