Endeca® Platform Services

Content Adapter Developer's Guide

Version 6.1.1 • December 2011

Contents

Copyright and disclaimer

Preface

Endeca® InFront enables businesses to deliver targeted experiences for any customer, every time, in any channel. Utilizing all underlying product data and content, businesses are able to influence customer behavior regardless of where or how customers choose to engage — online, in-store, or on-the-go. And with integrated analytics and agile business-user tools, InFront solutions help businesses adapt to changing market needs, influence customer behavior across channels, and dynamically manage a relevant and targeted experience for every customer, every time.

InFront Workbench with Experience Manager provides a single, flexible platform to create, deliver, and manage content-rich, multichannel customer experiences. Experience Manager allows non-technical users to control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

At the core of InFront is the Endeca MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. InFront Integrator provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. InFront Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Endeca InFront, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

About this guide

This guide covers the Content Adapter Development Kit and related API elements.

It describes how to use the Content Adapter Development Kit to:

- Easily create direct connections between enterprise content sources and the Endeca Information Transformation Layer (ITL).
- Create the means for transforming records in the Endeca ITL.

Who should use this guide

This guide is intended for Java developers who create Java manipulators and content adapters for use with the Endeca Information Access Platform, and pipeline developers who implement these Java manipulators and content adapters.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Endeca Customer Support

The Endeca Support Center provides registered users with important information regarding Endeca software, implementation questions, product and solution help, training and professional services consultation as well as overall news and updates from Endeca.

You can contact Endeca Standard Customer Support through the Support section of the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Chapter 1

About the Content Adapter Development Kit

This section introduces the Content Adapter Development Kit (CADK) and describes content adapters, Java manipulators, record adapters, and Perl manipulators. The documentation provided here assumes that you are comfortable programming in Java.

About the Content Adapter Development Kit

The CADK API is a collection of Java classes that provides developers with a flexible mechanism for adding content adapters and Java manipulators to the pipeline.

You can use different methods to prepare, change or clean your data, and to efficiently manipulate records as part of Forge data processing. The CADK lets you create your own ways of transforming both the incoming data, and the records themselves.

With the CADK API, you can create:

- Content adapters in Java, in order to transform the incoming source data.

   **Note:** The kit contains a sample content adapter.

- Java manipulators to modify the records that have already been preprocessed by existing components in your pipeline. For example, you can add properties to the records, or modify them in other ways.

About Java manipulators

A Java manipulator is your own code in Java that takes records from any number of pipeline components in Forge or, optionally, your source data, and changes it according to your processing requirements. A Java manipulator can then write any records you choose to its output.

For example, a Java manipulator can write the “transformed” records into its output, so that the records can be passed to the next pipeline component in Forge.

Java manipulators are the most generic way of modifying your data and records in the pipeline. In other words, content adapters represent a specific case of Java manipulators. You create Java manipulators with the Content Adapter Development Kit API.

A Java manipulator has the following characteristics:

- It adheres to the **Adapter** interface provided with the CADK. It uses the interface described in the *Content Adapter API (Javadoc)*, and requires you to import all the Java classes supplied with the Content Adapter Development Kit.
- It can have any number of inputs. It can take either source data, or, most importantly, it can take the records from record adapters or other components in the pipeline, and transform them further, according to your needs.
- It can have only one output. Do not emit empty records from your Java manipulator, that is record objects with no `PVal` objects specified. This can cause Forge to abort processing after acquiring the empty record.
- It has a simplified debugging mechanism. You import the JVM `java.util.logging.Logger` into your Java manipulator code. You can then make calls to the `Logger` object in different parts of your code to print the logging messages that are logged in the Forge log.
- It can transform your records in any way you want; it isn't limited to just changing `Pval` values.

When to use Java or record manipulators in your pipeline

This section provides an overview of pipeline components and helps you identify when to use them.

Once the source data passes through a record adapter, or, optionally, through your own content adapter, it gets turned into Endeca records. The records may need to be modified further, within the Forge pipeline.

For example, you may determine that some properties of your records must be cleaned or changed.

For the purpose of changing the records, the Endeca software offers a variety of methods:

- Java manipulators—Java manipulators are the most generic way of cleaning or changing your records in the pipeline; they use Java classes to specify the records transformations you need.
- Record manipulators—Record manipulators contain expressions, which are evaluated against each record as it flows through the pipeline. When Forge evaluates an expression, it may change the current record. The changes take a variety of forms, from adjustments of property values to creation of new data.
- Perl manipulators—Perl manipulators use Perl to manipulate source records as part of Forge's data processing. You can use a Perl manipulator to add, remove, and reformat properties, join record sources, and other such tasks.

You can use any one of these methods, based on your preferences. Use the following guidelines to decide which type of manipulator you need:

- If you prefer to use Java, use Java manipulators for adding, removing or changing properties of your records. Create Java manipulators with the CADK in cases when your records require any further transformations, after passing through other pipeline components, such as record and content adapters.
- If you prefer to use Perl, use Perl manipulators to perform your source record management tasks.
- If you prefer to use Data Foundry expressions and would rather edit XML expressions directly within the files (or within the **Expressions Editor** in Developer Studio) use record manipulators to transform your records.

In general, record manipulators are far more limiting than Java manipulators, in what they let you do with them. On the other hand, transforming records with record manipulators is faster than using Java manipulators, which in turn are faster than Perl manipulators.

Implementing a Java manipulator

This section provides a high-level overview of the process for implementing a Java manipulator.

To implement a Java manipulator, follow these steps:

1. Install the Endeca Platform Services package, which includes the CADK.
2. Write the Java manipulator, using the CADK Java classes, and compile it into a JAR file.
3. Deploy the Java manipulator that you created in a Forge pipeline, by configuring the Java manipulator as a pipeline component in Developer Studio, and then run Forge.

Related Links

About writing a Java manipulator on page 21
To write a Java manipulator you use a collection of Java classes from the CADK. Ensure that all Java manipulators that you write adhere to the `com.endeca.edf.adapter.Adapter` interface defined in the CADK.

Running a Content Adapter and a Java Manipulator on page 25
This chapter describes how you run your content adapter and Java manipulator in a Forge pipeline. To run your content adapter, you use a record adapter that you create in Developer Studio. To run your Java manipulator, you use a Java manipulator pipeline component that you create in Developer Studio.

Example of a Java manipulator

This example outlines the behavior of a sample Java manipulator.

The Java manipulator `SampleJavaManipulator.java` looks for a record property specified by the `SourceProp` pass through. If it finds that property, it creates a new property, specified by the `TargetProp` pass through, by copying the value of the source property into the target. It uses the first arbitrary source value if multiple source values exist. Next, the code emits all records to the next component in the pipeline, regardless of whether the property for the record was transformed.

Related Links

SampleJavaManipulator.java example on page 33
`SampleJavaManipulator.java` is a working sample manipulator that you can use as a shell code for writing your own Java manipulators.

About content adapters

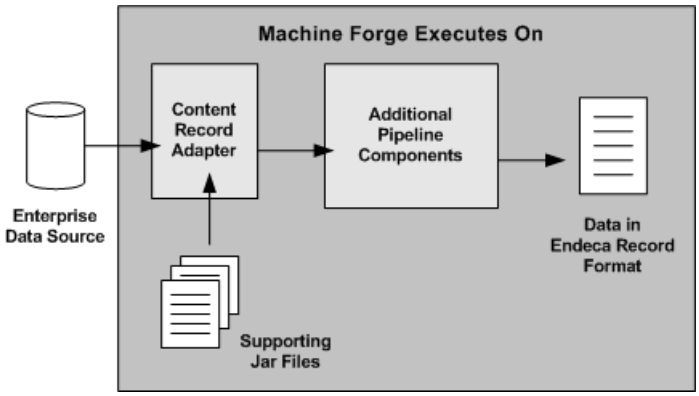
A *content adapter* is a custom record adapter that you write in Java in order to extract data from a data source and load it into Forge.

You need your own content adapter in cases when you cannot use any of the standard record adapters for data extraction. For instance, you may decide to create your own content adapter if your data is in the format that is not supported by any of the record adapters shipped with the Endeca software.

Content adapters extract data from other formats. Adding your own content adapters results in improved integration with a wider variety of enterprise content sources.

For example, if you want to access Lotus Notes data, you could write and compile a `NotesAdapter`, and then deploy it in a Forge pipeline by harnessing it in a record adapter. The content adapter code parses the source data, allowing the record adapter to feed the parsed data into the pipeline.

The illustration below shows how enterprise data is transformed into Endeca records using the content adapter:



When to use content adapters

When you are building your Endeca implementation, you need to determine how you can import your source data, and whether a content adapter is necessary.

In general, regardless of the format of your source data, you need to first load it into the Data Foundry by using a record adapter. After the data is loaded through the record adapter, it can be processed further by Forge and indexed by Dgidx.

The Endeca software is packaged with predefined record adapters. The record adapters provide the build-in facility for processing data from many formats, including delimited, XML, binary, and others.

To decide whether you need a content adapter:

1. Determine if any of the standard record adapters can handle your source data. For information on record adapters, see the *Endeca Forge Guide* and the *Developer Studio Help*. If you can use a record adapter, import your source data with it.
2. If the format of your source data cannot be handled by any of the predefined record adapters shipped with Endeca, create your own content adapter in Java to process your data, using the CADK.

Note: Instead of a content adapter that will transform your source data, you can create a Java manipulator for the same purpose, since Java manipulators are a general version of the content adapters.

Here is how content adapters differ from Java manipulators:

- Content adapters cannot take any Forge components as an input. They can only read data from multiple external sources. (You need a separate content adapter for each data source).
- Java manipulators can read in source data as well, but the main purpose of Java manipulators is to process records that already exist within the pipeline in Forge.

To summarize, Java manipulators can take either source data, or records that have already been passed through record adapters or manipulators, and transform the records further. Content adapters can only transform source data.

Implementing a content adapter

This section provides a high-level overview of the process for implementing a content adapter.

To implement a content adapter, follow these steps:

1. Install the Endeca Platform Services package, which includes the CADK.
2. Write the content adapter, using the content adapter Java classes, and compile it with `adapter.jar`.
3. Once you write a content adapter, you deploy it in a Forge pipeline by using the Developer Studio user interface, and then run Forge.

Related Links

About writing a content adapter on page 20
To write a content adapter you use a collection of Java classes from the CADK, and ensure that your content adapter method adheres to the `Adapter` interface defined in the CADK.

Running a Content Adapter and a Java Manipulator on page 25
This chapter describes how you run your content adapter and Java manipulator in a Forge pipeline. To run your content adapter, you use a record adapter that you create in Developer Studio. To run your Java manipulator, you use a Java manipulator pipeline component that you create in Developer Studio.

About record adapters

Record adapters are standard methods for manipulating the source data and loading it into the pipeline. Standard record adapters are packaged with the Endeca software and accommodate a number of formats.

Record adapters let you load and save records in a variety of formats, including delimited, binary, ODBC (Windows only), JDBC, and Microsoft Exchange. Each record adapter format has its own set of attributes.

Use record adapters for the following purposes:

- You need at least one record adapter to read in the source records. The configuration of the record adapter depends on the data format of the source data. If the format of your source data has a matching standard record adapter, use it for your source data. If the standard record adapters cannot handle the format of your source data, write your own content adapter.
- Use record adapters specifically for changing the incoming source data, and not the records within the pipeline. If you want to change the records themselves, you can use any of the following methods: Perl manipulators, record manipulators or Java manipulators.
- You can use many record adapters in your pipeline: you transform data in each format through its own record adapter.

Note: For detailed information about record adapters, see the *Endeca Forge Guide*.

About Perl manipulators

Perl manipulators use Perl to efficiently manipulate source records as part of Forge data processing. If you prefer to use Perl instead of Java, you can create a Perl manipulator to add, remove, and reformat properties, join record sources, and other such tasks.

 **Note:** For detailed information about Perl manipulators, see the *Developer Studio Help*.

Here is the code that appears in the **Expressions editor** of the record manipulator; it represents a Perl manipulator:

```
<EXPRESSION LABEL="" NAME="PERL" TYPE="VOID" URL="">
<EXPRBODY>
<![CDATA[#extract fields from date:
  my @dates = get_props_by_name("Message Date");

  foreach $date (@dates) {
   my $value = $date->value();
   $value =~ /(\w{3}) (\w{3}) (\d\d) (\d\d):(\d\d):\d\d (\d{4})/;
   add_props(new Zinc::PropVal("Message Date (Weekday)", $1));
   add_props(new Zinc::PropVal("Message Date (Month)", $2));
   add_props(new Zinc::PropVal("Message Date (Day)", $3));
   add_props(new Zinc::PropVal("Message Time (Hour)", $4));
   add_props(new Zinc::PropVal("Message Time (Minute)", $5));
   add_props(new Zinc::PropVal("Message Date (Year)", $6));
  }

  remove_props("Message Date");

  #extract information about sender from X-Yahoo-ProfData:
  my @senderInfos = get_props_by_name("X-Yahoo-ProfData");
    foreach $senderInfo (@senderInfos) {
   my $value = $senderInfo->value();

   if($value =~ /(\d*)\/(\w)\/(.*)/) {
    add_props(new Zinc::PropVal("Sender Age", $1));
    add_props(new Zinc::PropVal("Sender Location", $3));

    my $sex;
    if($2 eq "M") {
     $sex = "male";
    }else{
     $sex = "female";
    }
    add_props(new Zinc::PropVal("Sender Gender", $sex));
   }
  }
    #copy subject to "P_Name", the default display key :
   my @subjects = get_props_by_name("Subject");

  foreach $subject (@subjects) {
   add_props(new Zinc::PropVal("P_Name", $subject->value()));
  }]]>
</EXPRBODY>
</EXPRESSION>
```

About record manipulators

Record manipulators are installed with Endeca and handle a variety of source data formats. Use them for transforming the records and loading them into the pipeline.

Record manipulators contain XML expressions which Forge compares with each record as it flows through the pipeline. When Forge evaluates an expression, it may change the current record. The changes take a variety of forms, from adjustments of property values to creation of new data.

The CADK reference implementation pipeline contains a record manipulator that transforms properties such as the message data and email subject line into Endeca properties.

Record manipulators are similar to Java manipulators in what they do (transform records in the pipeline). The differences between Java manipulators and record manipulators are as follows:

- When you create record manipulators, you add Data Foundry expressions. You modify files directly, by using either Perl or XML expressions, within the **Expression editor** of the Developer Studio. (Alternatively, you can reference your external Perl code in a Perl manipulator).
- When you create Java manipulators, you have a cleaner way of creating your own methods for changing the records, using a family of Java API classes in the CADK designed for this purpose. You write your own Java code and then incorporate it as a Java manipulator in your pipeline.

Note: For detailed information about record manipulators and expressions used for them, see the *Data Foundry Expression Reference* and the *Endeca Forge Guide*.

Chapter 2

Installing the Content Adapter Development Kit

This section describes how to set up the CADK.

About installing the CADK

The CADK is installed by default as part of the Endeca Platform Services.

Note: The *Endeca Platform Services Installation Guide* contains detailed instructions on how to install this package. The guide also explains how to uninstall the Endeca Platform Services components.

CADK system requirements

The CADK runs on the same machine as the rest of the Endeca Platform Services components, and has two specific requirements in addition to those required by Platform Services.

Note: For details on the Endeca Platform Services component requirements, see the *Endeca Platform Services Installation Guide*.

In particular, the CADK has these requirements:

- The machine you are running the CADK on must have the Java 2 Platform Standard Edition 5.0 (aka JDK 1.5) or 6.0 (aka JDK 1.6) installed. You will have the correct version of JDK with the Platform Services default installation, because it includes a suitable version of the JDK in the `$ENDECA_ROOT/j2sdk` directory.
- If you intend to use the build script provided with the CADK, you must have Apache Ant version 1.5.4 or later installed.

CADK directory structure reference

The CADK is installed in the `$ENDECA_ROOT/cadk` directory on UNIX platforms (`%ENDECA_ROOT%\cadk` on Windows). It has the following directory structure:

Directory	Details
`src`	The directory to which the reference implementation Java class is written when you compile using Ant. Also contains the source code for the sample `MBox¬Adapter` content adapter.
`lib`	A directory that contains `adapter.jar`.
`reference`	A directory that contains sample content adapter code, as well as the compiled version.
`build.xml`	This Ant build file, along with the directory structure, makes it possible for you to compile and JAR the reference implementation's example content adapter or Java manipulator from the command line in the installation directory. You can copy this file and the associated directory structure to another location to use as a template when you build your own Java manipulator or content adapter. However, for the sake of stability, we recommend that you do not develop your own Java manipulators or content adapters within the Endeca directory structure.

Chapter 3

Writing Content Adapters and Java Manipulators

This chapter describes how you write, compile, and build a Java manipulator and a content adapter (in Java), using the reference implementation as an example.

About the content adapter framework

This section provides a high-level overview of the content adapter framework.

The content adapter framework is a collection of Java classes that you use to create a Java manipulator or a content adapter that adheres to the `Adapter` interface defined in the CADK. Later, you will be able to execute the Java manipulator, or the content adapter that is harnessed in a record adapter, in a Forge pipeline.

Classes relevant to the `Adapter` interface are documented in the *CADK Javadoc*, which you can obtain from the Endeca Developer Network (EDeN) at *http://eden.endeca.com*.

Related Links

Sample Content Adapter and Java Manipulator Code on page 29
This section contains the code for the `MBoxAdapter.java` content adapter, which is included as part of the CADK. It also contains the code for the `SampleJavaManipulator.java` Java manipulator.

CADK core classes

This section provides a brief overview of the CADK core classes.

Note: For detailed information, see the *CADK Javadoc*.

Class	Details
`Adapter` interface	The `Adapter` interface is a set of classes and methods to which your Java code must adhere. In other words, any Java manipulator that you write must implement the `Adapter` interface.
`AdapterConfig` class	Contains configuration information handed to the `Adapter` class via the execute method. The configuration data `AdapterConfig` contains is

Class	Details
	essentially a multimap of name-value pairs, where a given name may have one or more values associated with it.
`AdapterHandler` class	Used by a Java manipulator or a content adapter to emit records. In the case of a Java manipulator, this class represents your "connection" to Forge. The Java manipulator gets records from the `AdapterHandler` and emits records to it. You can also use the `AdapterHandler` to obtain information about Forge state, and to obtain the configuration from Forge.
`PVal` class	A name/value pair, a collection of which constitutes an Endeca record.
`Record` class	A collection of data in the format of `PVals` (that is, name/value pairs).
`AdapterException` class	Represents all exceptions thrown by `Adapter`-related classes. Specific exceptions are either inherited from this class or are accessible via Java's chained exception facility.

About writing a content adapter

To write a content adapter you use a collection of Java classes from the CADK, and ensure that your content adapter method adheres to the `Adapter` interface defined in the CADK.

A typical content adapter works as follows:

- The configuration information for the content adapter is passed through in the corresponding record adapter. The content adapter uses this information to connect to a data source. For example, in the CADK reference implementation, the source data file's name and location are passed through.
- The configuration information that is passed through in the record adapter is used to construct an `AdapterConfig` object.
- The `AdapterConfig` object is then handed to the `com.endeca.edf.adapter.Adapter` object's `Adapter.execute` method.
- `AdapterHandler`, meanwhile, is used to emit the records that Forge will process. A record created by the content adapter is processed by calling `emit` on the `AdapterHandler` parameter passed to `Adapter.execute`. Records emitted in this way flow from the record adapter (that is used for harnessing the content adapter) to the next pipeline component.

  For example, the CADK reference implementation contains `MBoxAdapter.java`, which is a basic example of a content adapter. This content adapter connects to a data source, extracts properties from it and then constructs Endeca records with properties extracted from that source. In particular, this content adapter reads email messages from an mbox file and converts each message to an Endeca record.

  Each record is processed by calling the emit method on the given adapter handler. The emitted records flow from the record adapter harnessing this content adapter to the next pipeline component.

Related Links

Running a Content Adapter and a Java Manipulator on page 25
This chapter describes how you run your content adapter and Java manipulator in a Forge pipeline. To run your content adapter, you use a record adapter that you create in Developer Studio. To run your Java manipulator, you use a Java manipulator pipeline component that you create in Developer Studio.

About writing a Java manipulator

To write a Java manipulator you use a collection of Java classes from the CADK. Ensure that all Java manipulators that you write adhere to the `com.endeca.edf.adapter.Adapter` interface defined in the CADK.

A typical Java manipulator works as follows:

- It implements the `com.endeca.edf.adapter.Adapter` interface. This enables your Java manipulator to be understood by Forge.
- It imports the Java classes from the CADK. It also imports the `java.util.logging.Logger` from the JVM.
- It defines its own logging by using the JVM `Logger` class. Defining logging is optional, although it is recommended.
- It specifies zero or more pass throughs. The pass through mechanism lets you specify arbitrary, key-value configuration pairs in the pipeline using the Developer Studio, and use those pairs in the Java manipulator. For example, the name and value pairs that you specify may "represent" the record properties that will be changed with your Java manipulator method. The `AdapterConfig` class defines a list of methods that you can use to conveniently handle the pass throughs.

  You can make the manipulator issue an error if a property that you wanted specified was not specified, or you can assign the pass through some default value.

- It takes inputs from one or more pipeline components, such as record adapters or other pipeline components. You can use the `getRecord` and `getNumInputs` methods on the `AdapterHandler` class for this task. For example, the `getNumInputs` method returns the number of record inputs that are used by the Java manipulator.
- Next, the code defines a loop within which your Java code for cleaning or changing the records runs.
- Once you are done changing or cleaning each record, use the `emit` method of the `AdapterHandler` class to send the records to the next step in the pipeline (by passing the records to the `AdapterExecute` class).

About compiling and building with Ant

You can use Apache Ant to compile and build your Java manipulator or content adapter, or you can use Java commands.

In the example that follows, you compile and build the reference application (`MBoxAdapter.java`) in the CADK directory. However, you should develop your own Java manipulator or content adapter outside of the Endeca directory structure. This ensures that your work is not inadvertently removed when you make changes to your Endeca software.

The simplest way to compile and build the reference implementation Java manipulator or content adapter is to run ant from the CADK directory. This builds `reference-adapter.jar` and places it in an install directory that Ant creates.

The name `reference-adapter.jar` can be changed by building with the `jarname` flag, for example:

```
ant -Djarname=MyNamedAdapter.jar
```

Similarly, the install location can be changed by building with the installpath flag, for example:

```
ant -Dinstallpath=my-install-path
```

Note: Ant relies on both `build.xml` settings and a particular directory structure. When you are ready to develop your own Java manipulator or content adapter, you can set up a similar directory structure for your own work and copy and modify `build.xml` appropriately.

Compiling and building with Java commands

You can use Apache Ant to compile and build your Java manipulator or content adapter. Alternatively, you can compile and build the project in separate steps, using Java commands.

For example, to compile the `MBoxAdapter.java` provided with the reference implementation:

Note: The instructions for creating JAR files for Java manipulators are the same.

1. Compile the `MBoxAdapter` with javac and the `adapter.jar` provided with the CADK in the class path (using a version of javac corresponding to the Java virtual machine version). For example:

   ```
   javac -classpath $ENDECA_ROOT/cadk/lib/adapter.jar MBoxAdapter.java
   ```

2. Create a JAR file from the class files generated by javac, as follows:

   ```
   jar cf reference-adapter.jar MBoxAdapter.class
   ```

3. If you are compiling a content adapter, enter the path of the JAR file in the path specified in the **Java properties class path** text box on the **Record Adapter editor General** tab, on every machine on which Forge will run.

   If the JAR file is not found, Forge terminates and logs an error that the content adapter class could not be located similar to the following:

   ```
   ERR: [Edf]: (AdapterRunner): Adapter class not found:
   MBoxAdapter

   WRN: [Adapter Input]: RecordAdapter 'Adapter Input':
   file 'INPUT' is empty or does not exist.

   ERR: [Edf]: Forge failed with 1 error and 1 warning.
   ```

   Forge inserts any logs from the custom adapter's logger into the Forge log files, specifying that the messages came from the content adapter.

4. If you are compiling a Java manipulator, enter the path of the JAR file in the path specified in the **Class path** field on the **Java Manipulator editor General** tab, on every machine on which Forge will run.

Related Links

Error Handling in the Content Adapter Development Kit on page 37
This section discusses how errors should be handled in the CADK.

Deploying a Java manipulator in the Forge pipeline on page 26
You use Developer Studio to configure a Java manipulator for transforming and producing records. A single Java manipulator can take records from several record servers.

Running a Content Adapter and a Java Manipulator on page 25
This chapter describes how you run your content adapter and Java manipulator in a Forge pipeline. To run your content adapter, you use a record adapter that you create in Developer

Studio. To run your Java manipulator, you use a Java manipulator pipeline component that you create in Developer Studio.

Chapter 4

Running a Content Adapter and a Java Manipulator

This chapter describes how you run your content adapter and Java manipulator in a Forge pipeline. To run your content adapter, you use a record adapter that you create in Developer Studio. To run your Java manipulator, you use a Java manipulator pipeline component that you create in Developer Studio.

Prerequisites

If you have not been doing your content adapter or Java manipulator development on a machine running the Endeca Information Access Platform, you will need to move the JAR file before continuing.

This chapter assumes that you have been doing your content adapter or Java manipulator development on a machine running the Endeca Information Access Platform. If this is not the case, you need to move the JAR file for your content adapter or Java manipulator to a machine running the Endeca Information Access Platform.

Creating a record adapter to harness your content adapter

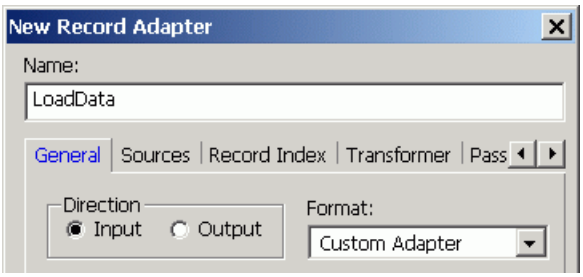
When you create a record adapter for loading data, you specify **Input** as the direction of data flow, along with the format and location of the source data.

Record adapters read and write record data. The record adapter describes where the data is located, the format, and various aspects of processing. Each file of source data requires its own record adapter.

To access your content adapter through a record adapter, follow these steps:

1. In Developer Studio, open the project that will contain the content adapter.
2. In the **Project Explorer**, double-click **Pipeline Diagram**.
3. In the **Pipeline Diagram editor**, click **New** > **Record** > **Adapter**.
   The Record Adapter editor opens.
4. In the **Name** text box, type a unique name for this record adapter. (In the example, we use `LoadData`.)
5. In the **General** tab, do the following:
   a) In the **Direction** frame, choose **Input**.

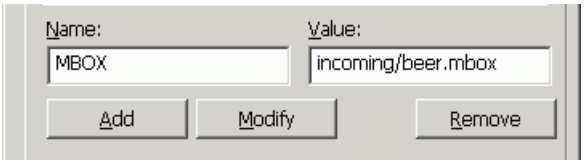
b) In the **Format** list, choose **Custom Adapter**.



c) In the **Java properties** frame, enter information according to the following requirements:

- **Java home** must point to a version 1.5.0 or greater install of the JDK.
- **Class** specifies the fully-qualified name of the adapter class, such as `com.acme.MyAdapter`.
- **Class path** specifies the absolute path to a JAR file (or a set of JAR files, separated by colons '`:`' on UNIX and semi-colons '`;`' on Windows) containing the classes required by the content adapter.

6. In the **Pass Throughs** tab, enter elements that become name-value pairs in the `AdapterConfig` given to the content adapter via the `Adapter.execute` method. In most cases this configuration is used to initialize the content adapter by connecting to a data source.



Deploying a Java manipulator in the Forge pipeline

You use Developer Studio to configure a Java manipulator for transforming and producing records. A single Java manipulator can take records from several record servers.

Java manipulators read records from multiple inputs, transform specific properties of the records, and write the results to a records output that can be passed to another component in the pipeline. A Java manipulator is not limited to changing properties on existing records. It can produce records in any form you choose.

Typically, the records that the Java manipulator takes as inputs come from one or more record servers, such as record adapters that already exist in your pipeline.

When you create a Java manipulator, you specify the Java class used for cleaning or changing the record properties, configuration pass throughs for the manipulator (typically these are the properties in the records that will be transformed), and the record sources that the Java manipulator should take as its input.

To deploy your Java manipulator in the pipeline:

1. In Developer Studio, open the project that will contain the Java manipulator. In the **Project Explorer**, double-click **Pipeline Diagram**, and then click **New** > **Java Manipulator**.
   The **New Java Manipulator editor** displays.
2. In the **Name** field, enter a name for your Java manipulator. The name must be unique among the pipeline components.

3. (Optional) Under the **General** tab, in the **Java home** field, specify the location of the Java Runtime Engine (JRE).

   If you do not specify this attribute, Forge obtains the location of the JRE by using the attribute specified to the Forge, the `--javaHome` flag. If it is not available, Forge uses the `ENDECA_ROOT/j2sdk` directory, which is installed as part of the Endeca Platform Services package. As a final fallback, Forge uses the `JAVA_HOME` environment variable.

4. Under the **General** tab, in the **Class field**, specify the name of the Java class that will be used by this Java manipulator. This is the class that you built with the CADK.

   For example, use this class for the sample Java manipulator:

   ```
   com.endeca.soleng.javamanipulator.SampleJavaManipulator
   ```

5. (Optional) Under the **General** tab, in the **Class path** field, specify the absolute path to the JAR file that contains the class specified by the `Class` attribute.

   The JAR file must contain the class and all other classes it depends on. If you do not specify this attribute, the Java manipulator uses the default class path of `ENDECA_ROOT/lib/java`. If you use this attribute, the specified class path is prefixed to the default classpath.

   For example, you can specify this class path for the `SampleJavaManipulator.jar`:`lib\SampleJavaManipulator.jar`

   Note: When running your pipeline, you can override the **Java home** and **Class path** settings using command-line options.

6. Under the **Sources** tab, specify which record servers in the pipeline are providing records to your Java manipulator. You can specify multiple record sources.

   Important: Make sure that the record servers (such as record adapters) are configured in the required order for processing. Developer Studio arranges the record servers in alphabetical order for processing by Forge; therefore, you may have to rename them so that they are processed by Forge in the order that your Java manipulator expects them.

7. Under the **Pass Throughs** tab, specify any pass through arguments that your Java manipulator expects. For example, you can specify the properties of the records that you want to be cleaned or changed by your Java manipulator. Enter the name and value pair for each record property that you will be using as your pass through.
8. (Optional) Use the **Comment** tab to describe your Java manipulator.
9. Click **OK**.
   The Java manipulator is added to the Forge pipeline.

About using other pipeline components for additional processing

Depending upon your requirements, you may want to add another pipeline component, such as a Java manipulator, a record manipulator, or a Perl manipulator to your pipeline.

Related Links

When to use Java or record manipulators in your pipeline on page 10

This section provides an overview of pipeline components and helps you identify when to use them.

Running the pipeline

You can run the pipeline through Forge using the **EAC Admin Console** page, a control script, or the command line.

After you add your content record adapter or your Java manipulator and do any additional requisite pipeline editing, it is time to run the pipeline through Forge.

 **Note:** See the *Endeca Application Controller Guide* for information on the EAC.

To do this:

Run the pipeline using either of the following methods:

- In the Endeca Application Controller (EAC) environment, start the **Forge** component in Endeca Workbench, using the **EAC Admin Console** page.
- Run the CADK pipeline in Forge with a command line similar to the following examples:

  Windows

  ```
  %ENDECA_ROOT%\bin\forge -vw --javaClasspath
  %ENDECA_ROOT%\cadk\reference\sample_cadk_data\
  adapters\MBoxAdapter.jar
  --javaHome C:\jdk1.5.0_05 Pipeline.epx
  ```

  UNIX

  ```
  $ENDECA_ROOT/bin/forge -vw --javaClasspath
  $ENDECA_ROOT/cadk/reference/sample_cadk_data/
  adapters/MBoxAdapter.jar
  --javaHome /usr/local/jdk1.5.0_05 Pipeline.epx
  ```

Upon successful completion, Forge produces binary output to the location specified in the pipeline's output record adapter.

Appendix A

Sample Content Adapter and Java Manipulator Code

This section contains the code for the `MBoxAdapter.java` content adapter, which is included as part of the CADK. It also contains the code for the `SampleJavaManipulator.java` Java manipulator.

MBoxAdapter.java example

`MBoxAdapter.java` is a basic example of a content adapter. It reads email messages from an `mbox` file and converts each message into an Endeca record.

Each record is processed by calling the `emit` method on the given adapter handler. Records emitted in this way flow from the record adapter harnessing the content adapter to the next pipeline component.

```
//Some standard java classes we'll need:
import java.lang.String;
import java.lang.reflect.Array;

import java.util.logging.Logger;
import java.util.regex.*;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

//Endeca classes to import:
import com.endeca.edf.adapter.Adapter;
import com.endeca.edf.adapter.AdapterConfig;
import com.endeca.edf.adapter.AdapterHandler;
import com.endeca.edf.adapter.AdapterException;
import com.endeca.edf.adapter.Record;
import com.endeca.edf.adapter.PVal;

/**
 * This is a simple example content adapter which reads email messages
 * from an mbox file and converts each message to an Endeca record.
 *
 * To compile this class type:
 *
 * javac -classpath $ENDECA_ROOT/lib/java/adapter.jar MBoxAdapter.java
 */
public class MBoxAdapter implements Adapter
```

```
{
 //Constants:

 //Enumeration of the property names we'll use:
 private static String PROP_NAME_SENDER    = "Sender";
 private static String PROP_NAME_DATE     = "Message Date";
 private static String PROP_NAME_SENDER_INFO  = "Sender Info";
 private static String PROP_NAME_BODY     = "Body";

 /**
  * This is the logger for our adapter. Forge will insert any logs from
  * this logger into its log files, specifying that they came from
  * "MBox Adapter".
  */
 private static Logger log = Logger.getLogger("MBox Adapter");

 /**
  * This regular expression will be used to extract fields from the From
  * line of each message. It matches the word "From" followed by a
  * space, followed by a sequence of non-whitespace characters which
  * constitute the sender, followed by some whitespace and a string of
  * exactly 24 characters which consitutes the date and possibly followed
  * by a string of characters containing other information about the
  * sender.
  */
 private static Pattern fromLineRegex
  = Pattern.compile("From (\\S*)\\s*(.{24})(.*)");

 /*********************************
  * Adapter Interface Implementation
  *********************************/

 public void execute(AdapterConfig config, AdapterHandler handler)
  throws AdapterException
 {
  //First process the configuration passed to this adapter by forge:

  //Get the paths of the mbox files to process:
  String[] mboxFiles = config.get("MBOX");

  //Check to make sure at least one file was specified:
  if(mboxFiles == null)
   throw new AdapterException("You must specify at least one filename.");

  //Now that we have processed the configuration, we're ready to

  //Loop over each of the files to parse:
  int nFiles = Array.getLength(mboxFiles);
  for(int n=0; n<nFiles; n++) {
   try{
    //Open the current file:
    BufferedReader reader
     = new BufferedReader(new FileReader(mboxFiles[n]));

    //process the current file:

    //The first line of each message should look like
    //"From <sender> <date> <more info>":
    String curFromLine = reader.readLine();

    if(curFromLine == null) {
```

```
     log.warning("mbox file '" + mboxFiles[n] + "' was empty.");
     continue; //Continue on to next file.
    }

    //Loop over the messages in the file:
    while(curFromLine != null) {
     //Report our progress:
     log.info("Processing message: " + curFromLine + "...");

     //Extract fields from the from line:

     Matcher matcher = fromLineRegex.matcher(curFromLine);

     if(!matcher.matches()) {
      log.warning("Invalid From line syntax in file '"
            + mboxFiles[n] + "': " + curFromLine);
      break; //Abort this file.
     }

     String sender = matcher.group(1);
     String date = matcher.group(2);
     String senderInfo = matcher.group(3);

     //Create a new Record for this message and add the from line
     //fields as properties:
     Record record = new Record();

     record.add(new PVal(PROP_NAME_SENDER, sender));
     record.add(new PVal(PROP_NAME_DATE, date));

     if(!senderInfo.equals(""))
      record.add(new PVal(PROP_NAME_SENDER_INFO, senderInfo));

     processHeaders(reader, record); //process the message headers.

     //The rest of the message is the message body. This method will
     //read that in, add it to the record and return the From line of
     //the next message, if any:
     curFromLine = processBody(reader, record);

     handler.emit(record); //Emit the completed record.
    }

    reader.close(); //close the current file.
   }catch(IOException e) {
    //There was a problem processing the current file, but maybe the
    //others will work; we'll log an error and continue:
    log.severe("Error processing mbox file '" + mboxFiles[n] + "': "
          + e.getMessage());
   }
  }
}

/**************************
 * MBoxAdapter Implementation
 **************************/

/**
 * This method extracts all of the header fields from an mbox message
 * and adds them to the specified record.
 *
```

```
 * @param reader BufferedReader to extract headers from
 * @param record Record to add header properties to
 *
 * @throws IOException
 */
private void processHeaders(BufferedReader reader, Record record)
 throws IOException
{
 //Loop until we reach a blank line, which indicates the end of the
 //headers, or we reach the end of the input stream:
 while(true) {
  String line = reader.readLine();

  if(line == null)
   break;

  if(line.equals(""))
   break;

  //Each header has the form "Name: value". Extract the name and
  //value from the current header:
  int colonPos = line.indexOf(':');

  if(colonPos == -1) {
   log.warning("Invalid message header format. Expected a colon in "
         + "the line '" + line + "'");
   continue; //Move on to next header.
  }

  record.add(new PVal(line.substring(0, colonPos)
            , line.substring(colonPos + 1)));
 }
}

/**
 * Beginning at the current position of the reader, this method reads
 * in a message body until it reaches a blank line followed by a "From"
 * line indicating the start of the next message, or the stream runs out
 * of data. Once it is done reading in the body, it adds the body text
 * to the specified record as a property, and returns the "From" line of
 * the next message, if any.
 *
 * @param reader  Reader to read body from
 * @param record  Record to add body to
 *
 * @return  The "From" line of the next message in the reader stream, or
 *          <code>null</code> if there are no more messages
 *
 * @throws IOException
 */
private String processBody(BufferedReader reader, Record record)
 throws IOException
{
 String body = "";
 String fromLine = null;

 while(true) {
  String line = reader.readLine();

  if(line == null)
   break;
```

```
   if(line.equals("")) {
    fromLine = reader.readLine();

    if(fromLine == null)
     break;

    //If the line begins with "From " then it is a From line:
    if(fromLine.regionMatches(true, 0, "From ", 0, 5))
     break; //A new message was found.

    //not a from line...
    line += fromLine;
    fromLine = null;
   }

   body += line; //Append line to body.
  }

  //Add the body to the record:
  record.add(new PVal(PROP_NAME_BODY, body));
  return fromLine;
 }
}
```

SampleJavaManipulator.java example

`SampleJavaManipulator.java` is a working sample manipulator that you can use as a shell code for writing your own Java manipulators.

This sample shows you how to use the `com.endeca.edf.adapter.Adapter` interface and its classes and methods in order to:

- Acquire multiple record sources, by using `getRecord` and `getNumInputs` methods of `AdapterHandler`.
- Specify which properties you want to modify via pass throughs, by using `AdapterConfig`.
- Enable logging and debugging, by using the `JVM java.util.logging.Logger` class.
- Prepare records for future processing in the pipeline, by using the `emit` method of the `AdapterHandler`.

This Java manipulator looks for a record property specified by the `SourceProp` pass through. If it finds that property, it creates a new property, specified by the `TargetProp` pass through by copying the value of the source property into the target. It uses the first arbitrary source value if multiple values exist. Next, the code emits all records to the next component in the pipeline, regardless of whether the property for the record was transformed.

```
package com.endeca.soleng.javamanipulator;
import java.util.logging.Logger;
import java.util.logging.Level;

import com.endeca.edf.adapter.Adapter;
import com.endeca.edf.adapter.AdapterConfig;
import com.endeca.edf.adapter.AdapterException;
import com.endeca.edf.adapter.AdapterHandler;
import com.endeca.edf.adapter.PVal;
import com.endeca.edf.adapter.Record;
```

```
/**
*
 * This is a sample Java Manipulator; you can use this code as a shell for
 writing your own manipulators.
 * This manipulator looks for a property specified by the "SourceProp" pass
 through.  If it finds that property it creates a new property, specified
by the "TargetProp" pass through,
 * copying the source's value into the target.
 * It will use the first arbitrary source value if multiple values exist.

 *
 */
public class SampleJavaManipulator implements Adapter
{
// all Java Manipulators must implement com.endeca.edf.adapter.Adapter

 // define constants for our passthrough names
 private static final String PASSTHROUGH_SOURCEPROP = "SourceProp";
 private static final String PASSTHROUGH_TARGETPROP = "TargetProp";

 // grab a logger from java.util.logging so we can write to the Forge logs

 private Logger logger = Logger.getLogger(this.getClass().getName());

 // because they implement Adapter, all Java Manipulators must have this
 // execute(AdapterConfig, AdapterHandler) method
 public void execute(AdapterConfig config, AdapterHandler handler)
   throws AdapterException
 {
  // read passthrough values from the config object;
  // the config object contains all our configuration.
  String sourceprop = config.first(PASSTHROUGH_SOURCEPROP);
  String targetprop = config.first(PASSTHROUGH_TARGETPROP);

  // validate passthrough values
  if (sourceprop == null)
   throw new AdapterException(PASSTHROUGH_SOURCEPROP + " passthrough not
specified; aborting");

  if (targetprop == null)
   throw new AdapterException(PASSTHROUGH_TARGETPROP + " passthrough not
specified; aborting");

  // loop through all input sources of this manipulator
  for (int inp = 0; inp != handler.getNumInputs(); inp++)
  {

   // create this flag for testing in our while loop, below
   boolean hasMoreRecords = true;
   long currentRecord = 0;

   // loop through all of the records for the current input
   while(hasMoreRecords)
   {

    // get the records, one by one, from the current input source.
    Record rec = handler.getRecord(inp);

    // when we are out of records, the last record will be null.
    if (rec != null)
```

```
    {
     ++currentRecord;
     String sourcepropValue = null; // placeholder for our source property


     // each record is a collection of properties.
     // loop through the properties to find the one we want.
     // if we are looking for multiple different properties of a single
     // record, it may be more efficient to stuff them into a Map.
     for (PVal prop : rec)
     {
      if (prop.getName().equals(sourceprop))
      {
       // we found the property we are looking for.  Save its value.
       sourcepropValue = prop.getValue();
       break;
      }
     }

     // If we found the sourceprop, copy its
     // value into another prop, add the new prop to the
     // record, and write to the Forge log.
     if (sourcepropValue != null)
     {
      rec.add(new PVal(targetprop,sourcepropValue));
      if (logger.isLoggable(Level.INFO))
       logger.info("Input source: " + inp + "; Record " + currentRecord +
": found " + sourceprop + "=\"" + sourcepropValue + "\", copying to " +
targetprop);

     }

     // we must emit() each record we want to pass out of this
    // manipulator into the next downstream component (e.g. PropertyMapper)

     handler.emit(rec);
    } else {
     hasMoreRecords = false;
    }
   }
  }
 }
}
```

Appendix B

Error Handling in the Content Adapter Development Kit

This section discusses how errors should be handled in the CADK.

Error handling in the CADK

To handle errors and enable logging in the CADK, use the following recommendations:

- A Java manipulator or a content adapter that you create should be implemented to throw an `AdapterException` if an error (such as failing to connect to a data source) occurs. Forge logs the message portion of the exception with a log level of `error`. If a Java manipulator or a content adapter simply wants to quit without issuing an error in Forge, it should return from the `execute` method.
- Never implement a Java manipulator or a content adapter to call `System.exit`.
- The `AdapterHandler` may throw an `AdapterException`, which, in general, should not be caught by a Java manipulator or a content adapter. However, in some cases it may be necessary to catch this exception (usually for the purpose of formatting the error message).

  Catching and re-throwing the message is acceptable. If you are doing so, it is important to note that the `AdapterHandler` throws a sub-class of `AdapterException`, `ShutdownRequestAdapterException`, which is used to signal that Forge will not process any more records. This can happen under several scenarios, the most common of which occurs when Forge is run with the `-n` flag.

  If the `ShutdownRequestAdapterException` is caught and an `AdapterException` is thrown in its place, Forge logs an error message. Normally this is not the desired effect. To avoid this, a content adapter's `try ... catch` block should be similar to the following:

```
try {
  ...
  adapterHandler.emit(record);
}
catch (ShutdownRequestAdapterException e) {
  throw e;
}
catch (AdapterException e) {
  throw new
AdapterException(reformat(e.getMessage()));
}
```

A `try ... catch` block of this form does not log an error message in Forge if the `ShutdownRequestAdapter Exception` is thrown by the `AdapterHandler`.

- In general, once the `AdapterHandler` throws an exception, further calls to `AdapterHandler.emit` are likely to trigger additional exceptions.

Logging messages

You can write error messages to a `java.util.logging.Logger` object in order to have them display in Forge's logs.

To log messages, use these recommendations:

1. Import the `JVM java.util.logging.Logger` into your Java manipulator code, and define your own logging object.
2. Write log messages you want displayed in Forge's logs to your `java.util.logging.Logger` object.

   This is generally done when implementing logging and debugging messages for Java manipulators and content adapters.

Note: Do not use `System.out` or `System.err` for logging because messages written to either of these streams are not displayed in Forge's logs.

For example, The `MBoxPlugin` content adapter logs informative messages to a static `java.util.logging.Logger` class. In this case, the `Logger` is named `MBox Adapter`; logger names should normally be based on the package name or class name of the logged component.

Messages written to this logger are displayed in the Forge log, provided Forge is run with the appropriate verbosity (in this case `-vi` or lower; for details on Forge's verbosity levels, see the flag reference in the *Endeca Forge Guide*). Adapter log messages appear in the Forge logs with the name of the logger in parenthesis.

In the case of `MBoxPlugin`, messages appear as follows when Forge is run with the `-vi` command line argument:

```
...
INF: [Edf]: (AdapterRunner): Adapter class: MBoxPlugin
INF: [Edf]: (MBoxPlugin): Creating <n> records.
...
INF: [Edf]: (MBoxPlugin): Finished!
...
```

The `AdapterRunner` log message declares the content adapter that was loaded.

Log levels reference

The log levels in `java.util.logging.Logger` map roughly to those in Forge, as the following table shows:

For more information about Forge logging, see "The Forge Logging System" in the *Endeca Forge Guide*.

java.util.logging.Level	Forge
SEVERE	ERROR
WARNING	WARN
INFO	INFO
CONFIG	INFO
FINE	DEBUG
FINER	DEBUG
FINEST	TRACE

Debugging Java manipulators

You can use the Forge log and JVM `Logger` class to help debug your Java manipulators.

Use the following features to assist you in debugging Java manipulators:

1. Import the `JVM java.util.logging.Logger` class, so that you can use its `severe`, `warning`, `info`, `config`, `fine`, `finer` and `finest` methods in your Java manipulator code to print log messages to the Forge log file.

   Here is the relevant code from the sample that enables logging:

   Note: Do not use `System.out` or `System.err` for logging because messages written to either of these streams are not displayed in Forge logs.

   ```
   //import the logging API that comes with JVM.
   import java.util.logging.Logger;
   ...
   //Define the logger
   Logger log = Logger.getLogger(this.getClass().getName());
   ```

   The `Logger` class has several methods which can be used to log messages of different severity levels. An example of one of these from the sample is the line:

   ```
   //Print a message using the log to your console, letting you know that
   forge has entered the TextCleaner javamanipulator
   log.info("Inside the TextCleaner adapter");
   ```

2. View the Forge log to locate the command that was used to call a specific JVM running your Java manipulator.

   In general, any messages that are printed using the `JVM Logger` class are automatically picked up by the Forge logging system and output into the Forge log file.

The name of the log file is `Edf.Pipeline.RecordPipeline.JavaManipulator.<JM_NAME>.log`. It is located in the Forge working directory. In the log file name, *JM_NAME* is the name of your Java manipulator.

Note: Once an error or warning occurs in the Java processing, it gets passed to the Forge processing console and is displayed and logged in the `Edf.Pipeline.RecordPipeline.JavaManipulator.<JM_NAME>.log` file.

In addition to the Java manipulator logging, for cases in which logging does not catch all relevant information, you can insert additional components to your pipeline, or create an additional pipeline.

For example, you can insert an output record adapter in the pipeline, after the Java manipulator, to write the records to a file that you can examine.

Or, you can create an additional simple pipeline. This pipeline can write the records emitted by the Java manipulator that you are debugging into an XML file. The pipeline will contain two items: the Java manipulator that you are testing, and an XML output record manipulator that writes all records produced by the test Java manipulator to an XML output file.

Passing JVM arguments to Forge

You can use the `--javaArgument` flag with Forge to pass VM arguments to the JVM running the Java manipulator.

To pass VM arguments with the `--javaArgument` flag:

1. From Endeca Workbench, access the **EAC Admin Console** page.
2. On the **Components** tab, select the **Forge** component.
3. Use the `--javaArgument` flag to pass the desired VM arguments.

   For example, the following string sets a 2 GB max heap for the JVM:

   ```
   --javaArgument -Xmx2g
   ```

Index