

Oracle® Endeca Information Discovery Integrator

Integrator Acquisition System Extension API Guide

Version 3.1.0 • October 2013

Copyright and disclaimer

Copyright © 2003, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. UNIX is a registered trademark of The Open Group.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

This software or hardware and documentation may provide access to or information on content, products and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Table of Contents

Copyright and disclaimer	2
Preface	5
About this guide	5
Who should use this guide	5
Conventions used in this guide	5
Contacting Oracle Customer Support	6
Chapter 1: Introduction to the IAS Extension API	7
System requirements	7
Overview of the IAS Extension API	7
About IAS extensions	8
About the IAS extension interfaces and classes	8
About IAS annotations	9
Record access interfaces	11
Common record properties	12
Chapter 2: Implementing a data source	14
Creating a data source class	14
Creating a pipeline component configuration class for a data source	16
Creating a runtime class for a data source	17
Supporting filtering in a data source	19
Supporting document conversion in a data source	19
Specifying which documents to convert	20
Full and incremental crawling modes	20
Supporting incremental acquisition in a data source	21
Chapter 3: Implementing a manipulator	23
Creating a manipulator class	23
Creating a pipeline component configuration class for a manipulator	25
Creating a runtime class for a manipulator	26
Supporting incremental acquisition in a manipulator	28
Chapter 4: Extension life cycle and threading	30
Life cycle of a data source	30
Life cycle of a manipulator	32
About threading	34
Chapter 5: Common implementation tasks	36
Stopping an extension when an acquisition stops	36
Cleaning up resources used by an extension	37
Storing state information for an extension	37
Exceptions that trigger fatal and non-fatal failures	38

Enabling logging in an extension	38
Unit testing an extension	39
Packaging an extension into a plugin	40
Installing a plugin into IAS	41
Running an extension	42
Uninstalling a plugin from IAS	42
Appendix A: Sample extensions	44
About the sample extensions	44
Sample extensions files and directories	45
Building the sample extensions	45
Unit testing the sample extensions	47
Installing the sample plugin into IAS	47
Running the sample CSV File data source	48
Running the sample Substring manipulator	51
Running the sample Blob Database data source	53
Running the sample Document Directory data source	55
Running the sample Change Tracking data source	58

Preface

Oracle® Endeca Information Discovery Integrator is a powerful visual data integration environment that includes:

The Information Acquisition System (IAS) for gathering content from delimited files, file systems, JDBC databases, and Web sites.

Integrator ETL, an out-of-the-box ETL purpose-built for incorporating data from a wide array of sources, including Oracle BI Server.

In addition, Oracle Endeca Web Acquisition Toolkit is a Web-based graphical ETL tool, sold as an add-on module. Text Enrichment and Text Enrichment with Sentiment Analysis are also sold as add-on modules. Connectivity to data is also available through Oracle Data Integrator (ODI).

About this guide

This guide describes how to implement, test, and package IAS extensions using the IAS Extension API. IAS extensions include data source extensions and manipulator extensions.

The guide assumes that you are familiar with Endeca concepts and introductory concepts of the Endeca Integrator Acquisition System. You can find an introduction to the Integrator Acquisition System in the *Integrator Acquisition System Developer's Guide*.

Who should use this guide

This guide is intended for Java developers who implement, unit test, and package extensions for use in the Integrator Acquisition System.

In this guide and in other IAS documentation, there are two developer roles who work with IAS extensions. There is an extension developer and a IAS application developer.

An extension developer creates extensions and packages extensions into one or more plugins and hands off the plugins to a IAS application developer. The IAS application developer installs the plugins. After installation, the IAS application developer can configure the extensions, and run the extensions as part of a crawl.

Conventions used in this guide

The following conventions are used in this document.

Typographic conventions

The following table describes the typographic conventions used in this document.

Typographic conventions

Typeface	Meaning
User Interface Elements	This formatting is used for graphical user interface elements such as pages, dialog boxes, buttons, and fields.
Code Sample	This formatting is used for sample code phrases within a paragraph.
<Variable Name>	This formatting is used for variable values, such as <install path>.
File Path	This formatting is used for file names and paths.

Symbol conventions

The following table describes symbol conventions used in this document.

Symbol conventions

Symbol	Description	Example	Meaning
>	The right angle bracket, or greater-than sign, indicates menu item selections in a graphic user interface.	File > New > Project	From the File menu, choose New, then from the New submenu, choose Project.

Contacting Oracle Customer Support

Oracle Customer Support provides registered users with important information regarding Oracle software, implementation questions, product and solution help, as well as overall news and updates from Oracle.

You can contact Oracle Customer Support through Oracle's Support portal, My Oracle Support at <https://support.oracle.com>.



Chapter 1

Introduction to the IAS Extension API

This section introduces the basics of the IAS Extension API.

[System requirements](#)

[Overview of the IAS Extension API](#)

[Common record properties](#)

System requirements

The IAS Extension API requires the following software installed on the system where you develop IAS extensions:

- A full IAS installation. No other Oracle products are required to create IAS extensions.
- Java Development Kit (JDK) 1.6
- A Java development environment
- If you want to open and build the sample extensions, you need Eclipse™ IDE for Java version 3.4.2 or later.
- If you want to use Ant to build the sample extensions, you need Apache Ant 1.7.1 or later.

Overview of the IAS Extension API

The IAS Extension API provides interfaces and classes to build extensions to IAS such as data source extensions and manipulator extensions. An extension developer packages extensions into a JAR and a IAS application developer installs the JAR and any additional JARs (for third-party dependencies) into the Integrator Acquisition System. After installation, the extensions are available and configurable using the IAS Server API or the IAS Server Command-line Utility.

The IAS Extension API is installed by default as part of the Integrator Acquisition System.

The components of the IAS Extension API include the following:

- The IAS Extension API, utility, and record access packages (IAS\`<version>`\lib\ias-extension-api)
- The IAS Extension samples (IAS\`<version>`\sample\ias-extensions)
- *Integrator Acquisition System Extension API Reference (Javadoc)* (IAS\`<version>`\doc\ias-extension-api-javadoc)
- *Integrator Acquisition System Extension API Guide* (this guide).

About IAS extensions

An extension developer can use the IAS Extension API to create data source extensions and manipulator extensions.

Data source extensions can access any type of data source that you want to include in the Integrator Acquisition System. For example, data source extensions might access flat files, databases, content management repositories, and so on.

Manipulator extensions transform Endeca records as part of data processing in an IAS crawl. In a typical usage, manipulators run in an IAS crawl to provide record pre-processing before an Integrator ETL graph runs.

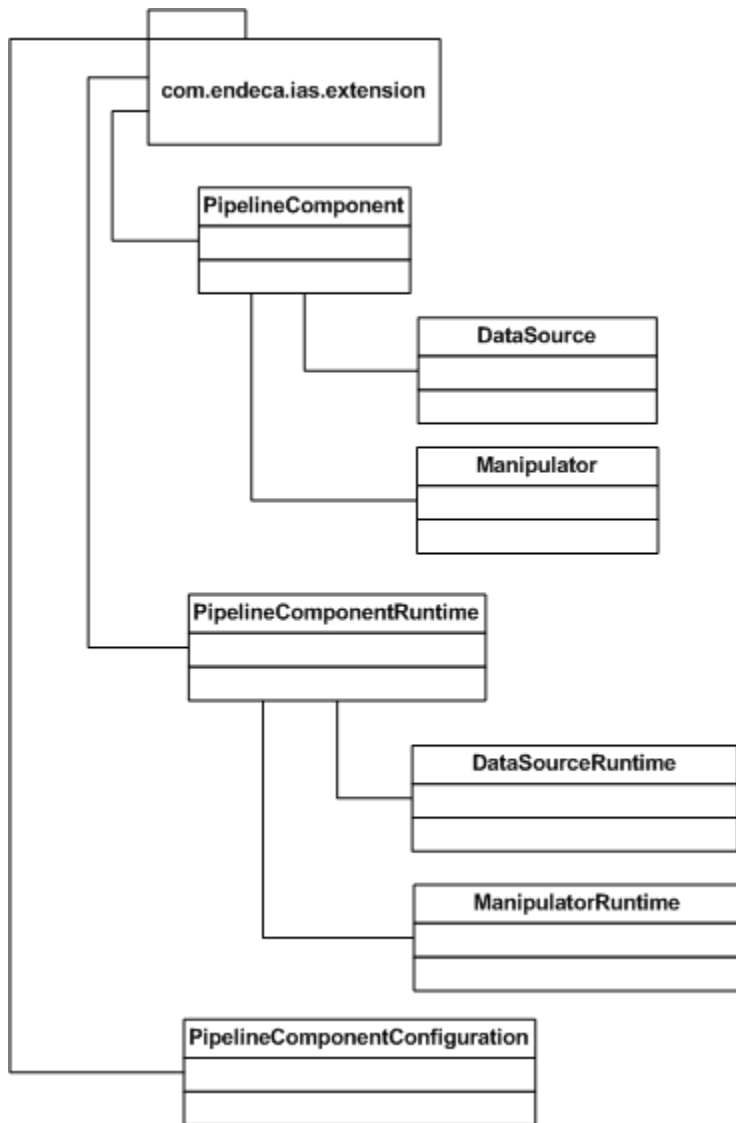
About the IAS extension interfaces and classes

The interfaces of the IAS Extension API are contained in two Java packages. The `com.endeca.ias.extension` package provides interfaces to interact with the IAS framework and interfaces to represent data source and manipulator extensions. The `com.endeca.ias.extension.annotation` package provides annotations to describe the configuration of an extension.

Primary classes in `com.endeca.ias.extension`

The following diagram shows the inheritance hierarchy of several classes you will work with in the package. The classes in `com.endeca.ias.extension` are declared in the API as abstract classes. Most of the abstract classes have default implementations of their methods. You can often use the default implementation

of the methods, or if necessary, you can override the default implementation. The key abstract classes in the package declare abstract methods which you must implement to provide extension functionality.



About IAS annotations

The IAS Extension API provides Java annotations that define extensions to the Integrator Acquisition System and also define configuration properties for an extension. Annotations can describe whether configuration properties are required or optional, whether they have a default value, and their display properties in `ias-cmd`.

Annotations define extensions

An extension requires an annotation of either `@IasDataSource` or `@IasManipulator` to indicate whether the extension is a data source or a manipulator.

After an IAS application developer installs a plugin, the Integrator Acquisition System scans for extensions by checking for classes that have an annotation of either `@IasDataSource` or `@IasManipulator` and by checking the uniqueness of extension IDs.

Here is an example annotation that defines a data source extension:

```
@IasDataSource(displayName="CSV File", description="Reads comma separated files")
public class CsvDataSource extends DataSource<CsvDataSourceConfig>
```

Here is an example annotation that defines a manipulator extension:

```
@IasManipulator(
    supportsIncrementals=true,
    deleteRecordsBypassManipulator = true,
    displayName="Substring Manipulator",
    description="Generates a new property that is a substring of another property value")
public class SubstringManipulator extends Manipulator<SubstringManipulatorConfig>
```

Annotations define configuration properties for an extension

You annotate a Java field in a `PipelineComponentConfiguration` class to expose the field as a configuration property for an extension. The fields that you annotate are available to the IAS Server Command-line Utility and the IAS Server API. If you do not annotate a field, IAS Server ignores it.

Each Java field has a IAS annotation that corresponds to the data type of the Java field. Field annotations include the following:

- `@StringProperty`
- `@BooleanProperty`
- `@DoubleProperty`
- `@IntegerProperty`

Annotations contain attributes that specify additional information about a configuration property. This information may control the order in which fields render, default values for the fields, and so on.

Here is an example annotation of two string fields:

```
@StringProperty(isRequired=true, name="inputFile", displayName="Input File",
    description="Path to the input csv file e.g. c:\\incoming\\data.csv")
private String mInputFile;

@StringProperty(isRequired=true, name="keyColumn", displayName="Key Column",
    description="Name of the column with the record key")
private String mKeyColumn;
```

Here is an example annotation for a field which is a list of strings (a multi-valued property):

```
@StringProperty(isRequired=true, name="sourcePropertyList", displayName="Source Property List")
private List<String> mSourcePropertyList;
```

Here is an example annotation for an integer field with four attributes:

```
@IntegerProperty(isRequired=false, name="startIndex", displayName="Substring Start Index",
    description="Substring start index (zero based)", defaultValue=0)
private int mStartIndex;
```

Annotations specify groups and the order of fields in a group

You can annotate an extension to organize a set of fields into a group and specify the order of fields in a group.

The `@ConfigurationGroup` annotation specifies that the fields contained within it are a group from the perspective of the IAS Server Command-line Utility.

The `groupName` attribute of `@ConfigurationGroup` specifies the label for the group, and the `propertyOrder` attribute specifies the order in which the properties display as output from tasks in IAS Server Command-line Utility. If you omit the `propertyOrder` attribute, the properties are sorted alphabetically and display alphabetically when returned from `ias-cmd`.

Here is an example group named `User Credentials` that defines three configuration properties:

```
@ConfigurationGroup(groupName="User Credentials", propertyOrder={"userName","userPassword"})
```

Annotations specify the order of multiple groups of fields

You can annotate an extension to specify the order of multiple groups of fields. As mentioned above, you specify each group with a `@ConfigurationGroup` annotation. You specify the order of multiple groups with a `@ConfigurationGroupOrder` annotation.

This may be useful if you want to enforce the order of groups and order of the fields within each group. For example, suppose a data source extension accesses a database. The first group is called `User Credentials` and it displays a `userName` property and a `userPassword` property.

Next you want a second group of fields called `Database Settings`, and it displays `serverName`, `databasePath`, and `portNumber`.

Last you want a third group called `Advanced Settings`, and it displays `settingA`, `settingB`, and `settingC`.

This scenario requires the following annotations:

```
@ConfigurationGroupOrder({
    @ConfigurationGroup(groupName="User Credentials", propertyOrder={
        "userName","userPassword"})
    @ConfigurationGroup(groupName="Database settings", propertyOrder={
        "serverName","databasePath","portNumber"})
    @ConfigurationGroup(groupName="Advanced Settings", propertyOrder={
        "settingA","settingB","settingC"})})
```

Record access interfaces

The IAS Extension API contains the `com.endeca.eidi.record` package necessary for both `ManipulatorRuntime` and `DataSourceRuntime` record processing.

Structure of an Endeca record

Endeca records are made up of property key-value pairs. Each of the key-value pairs are strings. In some cases, a property key can have multiple values. Here is a simple representation of an example record

produced by acquiring data from a file system. This representation illustrates the makeup of a record as key-value pairs. For example, the property key named `Endeca.SourceType` has a value of `FILESYSTEM`.

Record example
<code>Endeca.Action: UPSERT</code>
<code>Endeca.FileSystem.IsHidden: false</code>
<code>Endeca.FileSystem.ACL.AllowRead: fsmith</code>
<code>Endeca.FileSystem.ACL.AllowRead: Users</code>
<code>Endeca.FileSystem.Name: dir2</code>
<code>Endeca.FileSystem.Path: C:\docs\seed1\dir2</code>
<code>Endeca.FileSystem.ParentPath: C:\docs\seed1</code>
<code>Endeca.File.Size: 0</code>
<code>Endeca.FileSystem.IsDirectory:true</code>
<code>Endeca.FileSystem.ModificationDate: 1183755212473</code>
<code>Endeca.SourceType: FILESYSTEM</code>

To manipulate the key-value pairs in a record, you implement `ManipulatorRuntime.processRecord()` in a manipulator. For a list of the properties IAS can produce in a record, see "Record properties generated by crawling" in the *Integrator Acquisition System Developer's Guide*.

The Record class

The `Record` class represents an individual Endeca record. The methods in this class access the name and value of a property value.

The PropertyValue class

The `PropertyValue` class represents individual property values on an Endeca record. The methods in this class access and modify one or more values of a property.

Common record properties

The IAS Server generates certain properties whether you crawl data from a file system, delimited file, JDBC database, or custom data source.

The IAS Server generates record properties and assigns each property a qualified name, with a period (.) to separate qualifier terms. The IAS Server constructs the qualified name as follows:

- The first term is always `Endeca` and is followed by one or more additional terms.
- The second term describes a property category, for example: `Document` or `CMS`.

The IAS Server generates the following properties for all records:

Endeca Property Name	Property Value
<code>Endeca.Action</code>	The action that was taken with the document. Values are <code>UPSERT</code> (the file or folder has been added or modified) or <code>DELETE</code> (the file or folder has been deleted since the last acquisition).

Endeca Property Name	Property Value
Endeca . Id	Provides a unique identifier for each record. For data source extensions, an extension developer must add Endeca . Id to each record and assign it a value appropriate for the data source.
Endeca . SourceId	Indicates the name of the acquisition source.



Chapter 2

Implementing a data source

This section describes how to implement a data source with the IAS Extension API.

[Creating a data source class](#)

[Creating a pipeline component configuration class for a data source](#)

[Creating a runtime class for a data source](#)

[Supporting filtering in a data source](#)

[Supporting document conversion in a data source](#)

[Specifying which documents to convert](#)

[Full and incremental crawling modes](#)

[Supporting incremental acquisition in a data source](#)

Creating a data source class

You create a data source by extending the `DataSource` abstract class and other supporting classes.

A `DataSource` requires an `@IasDataSource` annotation. The annotation has several important attributes you can configure:

- `displayName`. Optional. The name of a data source as it is returned from the `listModules` task of the IAS Server Command-line Utility. If not specified, `displayName` defaults to the `name` value.
- `description`. Optional. The description of what the data source can access. The description is returned from the `listModules` task of the IAS Server Command-line Utility.
- `id`. Optional. If unspecified, the extension defaults to using the fully qualified class name as its `id`.

The `listModules` task of the IAS Server Command-line Utility and the `listModules()` method of the IAS Server API both return the attribute values you specify in the `@IasDataSource` annotations.

To create a data source extension:

1. Create a Java project in your development environment of your choice.
If you are creating several extensions in one plugin, you can use the same Java project for each extension.
2. Add the IAS Extension API libraries to your compile classpath. These include all the libraries available in `<install path>\IAS\<version>\lib\ias-extension-api`.
3. Create a subclass of `DataSource` and specify the `PipelineComponentConfiguration` subclass that the extension uses. The `DataSource` requires a zero-argument constructor.

For example:

```
public class CsvDataSource extends DataSource<CsvDataSourceRuntime,CsvDataSourceConfig>{
}
```

4. Add a `@IasDataSource` annotation to the `DataSource` class.

For example:

```
@IasDataSource(displayName="CSV File", description="Reads comma separated files")
public class CsvDataSource extends DataSource<CsvDataSourceRuntime,CsvDataSourceConfig>{
}
```

5. Implement the `getConfigurationClass()` method to return the appropriate `PipelineComponentConfiguration` subclass.

For example:

```
public Class<CsvDataSourceConfig> getConfigurationClass() {
    return CsvDataSourceConfig.class;
}
```

6. Implement the `createDataSourceRuntime()` method to create an implementation of the `DataSourceRuntime` class.

For example:

```
public CsvDataSourceRuntime createDataSourceRuntime(
    CsvDataSourceConfig config, PipelineComponentRuntimeContext context) {
    return new CsvDataSourceRuntime(context, config);
}
```

7. Implement the `getRuntimeClass()` method to return the runtime class the data source creates.

For example:

```
public Class<CsvDataSourceRuntime> getRuntimeClass() {
    return CsvDataSourceRuntime.class;
}
```

8. Optionally, override the `deleteInstance()` method. IAS Server calls `deleteInstance()` when it removes an extension. In this method, you can perform any clean up that is necessary when IAS Server calls `deleteInstance()` to remove the extension from an acquisition. The default implementation of `deleteInstance()` is empty.

Example of a data source extension

To see many of the steps above, refer to the sample data source extension in `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\csv\CsvDataSource.java`.

[Life cycle of a data source](#)

[About threading](#)

[Creating a runtime class for a data source](#)

[Creating a pipeline component configuration class for a data source](#)

Creating a pipeline component configuration class for a data source

A data source extension requires a `PipelineComponentConfiguration` class to describe the extension's configuration, to validate the data source's configuration, and to determine whether a configuration change requires a full acquisition for the data source.

To create a data source configuration:

1. In the Java project that contains your `DataSource` implementation, create a subclass of `PipelineComponentConfiguration`.

For example:

```
public class CsvDataSourceConfig extends
    PipelineComponentConfiguration<CsvDataSourceConfig>{
}
```

2. Add each field that you want available as a configuration property in the data source.

For example:

```
private String mInputFile;

private String mKeyColumn;
```

3. Add an annotation to each field. The annotation type must match the field's data type. See package `com.endeca.ias.extension.annotation` in the *IAS Extension API Reference (Javadoc)* to determine which annotations have required attributes and to determine which attributes are appropriate for the field. Also see, [About IAS annotations on page 9](#).

For example:

```
@StringProperty(isRequired=true, name="inputFile",
    displayName="Input File", description
    ="Path to the input csv file e.g. c:\\incoming\\data.csv")
private String mInputFile;

@StringProperty(isRequired=true, name="keyColumn",
    displayName="Key Column", description="Name of the column with the record key")
private String mKeyColumn;
```

4. If you want to order configuration properties within a single group, add a `@ConfigurationGroupOrder` annotation to the `PipelineComponentConfiguration` class and then add a nested `@ConfigurationGroup` annotation.

For example, here is one group of fields that display in order — `filePath`, `headerRow`, and `separator`:

```
@ConfigurationGroupOrder({@ConfigurationGroup(groupName="Basic",
    propertyOrder={"filePath","headerRow","separator"})})
public class CsvDataSourceConfig implements
    PipelineComponentConfiguration<CsvDataSourceConfig>{
}
```

If you omit `propertyOrder`, the properties are sorted alphabetically and display alphabetically.

5. If you want to order multiple groups, add additional `@ConfigurationGroup` annotations within `@ConfigurationGroupOrder` for each group of fields that you want ordered.

- Optionally, override the default implementation of `isFullAcquisitionRequired()`. The default implementation determines whether a configuration change should force full acquisition the next time an acquisition is run by comparing the old `PipelineComponentConfiguration` and the new `PipelineComponentConfiguration` using the `equals()` method. The default implementation of the `equals()` method uses reflection to compare all non-transient fields for equality.

You can write code that checks a specific property to determine if a full acquisition is required (rather than the entire `PipelineComponentConfiguration`). If you want to force a full acquisition, write code that always returns `true`.

- Optionally, override the default implementation of `validate()`. IAS Server performs data type and constraint validation (constraints may include `minValue` and `maxValue` for integer properties). Any code you write in `validate()` performs additional custom validation.

For example:

```
public List<ValidationFailure> validate(
    CsvFileDataSourceConfiguration configuration) {

    List<ValidationFailure> validationFailures =
        new LinkedList<ValidationFailure>();
    File checkFile = new File(configuration.getFilePath());

    if (!checkFile.exists()) {
        validationFailures.add(new ValidationFailure("File " +
            checkFile.getAbsolutePath() + " does not exist"));
    }

    return validationFailures;
}
```

If validation fails, the `PipelineComponentConfiguration.validate()` method returns a collection `ValidationFailure` objects.

- If it is necessary for unit testing or for the implementation of the data source runtime, you may also need to write getter and setter methods for each field that you added.

Example of a pipeline component configuration for a data source

To see many of the steps above, refer to the sample data source extension in `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\csv\CsvDataSourceConfig.java`.

Creating a runtime class for a data source

The `DataSourceRuntime` is the runtime representation of a data source instance. It is created by `DataSource.createDataSourceRuntime()` and exists for the life span of the data source.

IAS Server creates and passes a `PipelineComponentRuntimeContext` class to `DataSource.createDataSourceRuntime()`. The `PipelineComponentRuntimeContext` specifies an output channel, error channel, a state directory, and several other runtime properties.

The `ErrorChannel.discard()` method discards any invalid records from the record acquisition process. Also, in addition to discarding records, the `ErrorChannel` class processes exceptions that you catch. This processing includes incrementing the appropriate metric for a record and also logging a record in the `ias-service.log` file. The `ErrorChannel` logs events at level `WARN` and higher.

To create a runtime class for a data source:

1. In the Java project that contains the `DataSource` implementation, create a subclass of `DataSourceRuntime`.

For example:

```
public class CsvDataSourceRuntime extends DataSourceRuntime {  
    }  
}
```

2. Implement the `DataSourceRuntime` constructor.
3. Implement the abstract method `runFullAcquisition()` to define how to acquire content from the data source. The implementation depends on your custom data source.
4. Within your implementation of `runFullAcquisition()`, call `ErrorChannel.discard()` as necessary to discard any records that are invalid or have errors, and also call `OutputChannel.output()` for each record that has been processed.
5. Optionally, implement either the `BinaryContentFileProvider` interface or the `BinaryContentInputStreamProvider` interface if the data source needs to support text extraction. For guidance, see [Supporting document conversion in a data source on page 19](#).
6. Optionally, implement the `IncrementalDataSourceRuntime` interface to calculate the changes in your data source, rather than have the Integrator Acquisition System determine the changes for you. For guidance, see [Supporting incremental acquisition in a data source on page 21](#).
7. Optionally, handle requests to stop an acquisition by providing a mechanism to stop an extension's runtime object in a timely way. This may include polling `PipelineComponentRuntimeContext.isStopped()` and may include overriding `PipelineComponentRuntime.stop()`. For guidance, see [Stopping an extension when an acquisition stops on page 36](#).
8. Optionally, override `PipelineComponentRuntime.endAcquisition()` to clean up any resources used by `PipelineComponentRuntime`. For guidance, see [Cleaning up resources used by an extension on page 37](#).

Example of a data source runtime

To see many of the steps above, refer to the sample data source extension in `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\csv`.

[Creating a data source class](#)

Supporting filtering in a data source

An extension developer can add support for including or excluding content by adding filtering logic to the `DataSourceRuntime.runFullAcquisition()` method and the `IncrementalDataSourceRuntime.runIncrementalAcquisition()` method.

Filtering logic may be necessary because in this release of the Integrator Acquisition System, data source extensions do not have the same filtering features as the File System data source, Delimited File data source, and JDBC data source. In particular, data source extensions have the following limitations:

- Filter objects, as represented by the `Filter` base class in the IAS Server API, are not currently supported for a data source extension.
- Document conversion filters specified in `DocumentConversionFilters.xml` do not apply to a data source extension.

An IAS data developer encounters these filtering limitations when he or she configures a data source. If you work around these limitations, be sure to communicate the expected filtering behavior of a data source to the IAS data developer.

Supporting document conversion in a data source

You add support for document conversion by making the `DataSourceRuntime` class implement either the `BinaryContentFileProvider` interface or the `BinaryContentInputStreamProvider` interface.

The `BinaryContentFileProvider` interface allows the extension to pass a file to IAS Server so IAS Server can perform document conversion. The interface provides a `getBinaryContentFile()` method that takes a `Record` as input and uses a property on the `Record` to identify the file to read. IAS Server then reads the file directly or caches it locally (optional) and then reads the file.

The `BinaryContentInputStreamProvider` interface allows the extension to download and convert binary contents to an input stream so IAS Server can read the input stream and perform document conversion. A common scenario is one where the data source extension connects to a database to read content. The interface provides a `getBinaryContentInputStream()` method that takes a `Record` as input and uses a property on the `Record` to identify the content to read. IAS Server then caches the content locally (not optional) and reads the content as an input stream.

During the document conversion process, IAS Server examines the file, extracts the text of the file, and stores the text as the `Endeca.Document.Text` property on the `Record`. In both interfaces, the IAS Server manages file access, local file download (if enabled), temporary files, and caching.

Enabling document conversion in the data source

An extension developer needs to implement one of the binary content provider interfaces, but not both, to support document conversion. An IAS application developer specifies whether document conversion is enabled by configuring the data source in XML create configuration file, by using the IAS Server API (`TextExtractionConfig`), or by using the IAS Server Command-line Utility.

If document conversion is enabled, an IAS data developer can also specify whether IAS Server should cache the file locally before reading it.

Example code in the IAS extension samples

To see an example of how `BinaryContentFileProvider.getBinaryContentFile()` is used, see the IAS sample extension in `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\directory\DirectoryDataSourceRuntime.java`.

To see an example of how `BinaryContentInputStreamProvider.getBinaryContentInputStream()` is used, see the IAS sample extension in `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\blob\BlobDataSourceRuntime.java`.

Specifying which documents to convert

There may be cases where you want to specify which documents you want to convert. For example, you may want to create a record for a specific document but exclude that record from the document conversion process because the document size is too large.

You cannot use document conversion filters specified in `DocumentConversionFilters.xml`, because it does not apply to data source extensions.

There are two approaches for specifying which documents to convert:

- Add logic in the extension that identifies a property in the record to examine, tests the property during acquisition, and returns null from either `getBinaryContentFile()` or `getBinaryContentInputStream()` for those records you want to exclude from the document conversion process.
- Add logic in the extension that examines a configuration file, determines which property to examine, and returns null from either `getBinaryContentFile()` or `getBinaryContentInputStream()` for those records you want to exclude from the document conversion process.

Full and incremental crawling modes

The IAS Server starts a crawl in one of two modes:

- `full` mode, in which all content is processed.
- `incremental` mode, in which only new, modified, or deleted content is processed.

Crawling in full mode

Crawling in *full* mode means that IAS processes all the content in a data source according to the filtering criteria you specify. As part of crawling a data source, IAS creates metadata information and stores it in a crawl history. This history includes the ID of each record and information about all properties on the record.

Crawling in incremental mode

Crawling in *incremental* mode means that IAS processes only that content whose metadata information, stored in the crawl history, has changed since the last crawl. Specifically, IAS checks all properties on the record to see if any have changed. If any properties have changed, the IAS Server crawls the content again.

This is true in cases where IAS is calculating the incremental difference. An extension developer, using the IAS Extension API, may choose to calculate incremental changes in a data source extension.

IAS automatically determines which crawling mode is necessary. By default, IAS attempts to crawl in incremental mode. If necessary, IAS switches to crawling in full mode, if a crawl's configuration has `unavailableIncrementalSwitchesToFullCrawl` set to `true`, and any of the following conditions are true:

- A data source has not been crawled before, which means no crawl history exists.
- A Record Store instance does not contain at least one record generation. (This applies to cases where the IAS Server is configured to output to a Record Store instance rather than a file on disk.)
- Seeds have been removed from the crawl configuration (adding seeds does not require crawling in full mode).
- The document conversion setting has changed.
- Folder filters or file filters have been added, modified, or removed in the data source configuration.
- Repository properties have been changed, such as the **Gather native properties** option for file system data sources.

If `unavailableIncrementalSwitchesToFullCrawl` is set to `false` and any of the above conditions are true, the crawl fails and throw an exception.

This switch from incremental to full mode can occur no matter how you run a crawl (using the IAS Server API or the IAS Server Command-line Utility).

After you crawl a data source using `ias-cmd` or the API, a status message is returned indicating whether a full or incremental crawl ran.

Supporting incremental acquisition in a data source

There are two approaches for determining the incremental difference between acquisitions from a data source: you can either let the Integrator Acquisition System determine the incremental difference, or you can implement the `IncrementalDataSourceRuntime` interface to determine the incremental difference.

If you are accessing a data source that tracks content revisions, it can be more efficient to implement `IncrementalDataSourceRuntime` and calculate the changes in your data source extension, rather than have the Integrator Acquisition System determine the changes for you.

The Integrator Acquisition System determines the incremental difference

The Integrator Acquisition System maintains a history of each acquisition (full or incremental) that runs against a data source. This history includes the `Id` of each record and information about all properties on each record. To determine the incremental change between acquisitions, IAS compares all the properties on a given record to determine if a record has changed between acquisitions.

If an acquisition does not find content that is listed in the history, IAS treats that content as deleted. That removal is part of the incremental change. Similarly, if an acquisition finds new content, IAS adds a record to the history. That addition is part of the incremental change.

This comparison can take a significant amount of time in large data sets.

An extension developer determines the incremental difference

Some data source types and repositories provide features to identify incremental content changes. For example, many data source types such as content management systems, version control systems, and enterprise management systems have the capability to track modification dates, user changes, and other types of content changes.

If this kind of information is available, a data source extension can programmatically request it and then acquire only content that has changed as part of an incremental acquisition. This approach to identifying incremental content changes is often more efficient than having the Integrator Acquisition System create and compare metadata histories to identify the incremental difference.

You can support this approach in a data source extension by implementing the `IncrementalDataSourceRuntime` interface. This includes the following steps:

- Implement `IncrementalDataSourceRuntime.checkFullAcquisitionRequired()`.

The logic of this method should do whatever is necessary to determine whether a full acquisition is required. For example, this may involve checking whether any manipulator extension in an acquisition requires state produced by a full acquisition. If a manipulator does require state, it would return `true` from `checkFullAcquisitionRequired()`.

If the Boolean is `true`, IAS Server then sets the `AcquisitionMode` to `FULL_ACQUISITION`. If the Boolean returned from `checkFullAcquisitionRequired()` is `false`, then IAS Server sets the `AcquisitionMode` to `INCREMENTAL_ACQUISITION`.

If the `AcquisitionMode` is set to `FULL_ACQUISITION`, the IAS Server switches from an incremental acquisition to a full acquisition and calls `DataSourceRuntime.runFullAcquisition()`.

- Implement `IncrementalDataSourceRuntime.runIncrementalAcquisition()`.

Example code in the IAS extension samples

To see an example of how `IncrementalDataSourceRuntime` is used, see the IAS sample extension in

```
<install path>\IAS\<version>\sample\ias-  
extensions\src\main\com\endeca\ias\extension\sample\datasource\incremental.
```

[Storing state information for an extension](#)



Chapter 3

Implementing a manipulator

This section describes how to implement a manipulator with the IAS Extension API.

[Creating a manipulator class](#)

[Creating a pipeline component configuration class for a manipulator](#)

[Creating a runtime class for a manipulator](#)

[Supporting incremental acquisition in a manipulator](#)

Creating a manipulator class

You create a manipulator by extending the `Manipulator` abstract class and other supporting classes.

A `Manipulator` requires an `@IasManipulator` annotation. The annotation has several important attributes you can configure:

- `supportsIncrementals`. Required. A Boolean value that indicates whether the manipulator supports input from an incremental acquisition. If one manipulator in an incremental acquisition has this set to `false`, the incremental acquisition runs in full mode.
- `deleteRecordsBypassManipulator`. Required. A Boolean value that indicates whether to send deleted input records directly to the manipulator's output channel or to send deleted input records into the manipulator. A value of `true` sends records to the output channel (by passing the manipulator). A value of `false` sends records into the manipulator.
- `displayName`. Optional. The name of a manipulator as returned from the `listModules` task of the IAS Server Command-line Utility.
- `description`. Optional. The description of what the manipulator does to Endeca records that it processes. The description is returned from the `listModules` task of the IAS Server Command-line Utility.
- `id`. Optional. If unspecified, the extension defaults to using the fully qualified class name as its `id`.

The `listModules` task of the IAS Server Command-line Utility and the `listModules()` call of the IAS Server API both return the attribute values you specify in the `@IasManipulator` annotations.

To create a manipulator extension:

1. Create a Java project in your development environment of your choice.
If you are creating several extensions in one plugin, you can use the same Java project for each extension.
2. Add the IAS Extension API libraries to your compile classpath. These include all the libraries available in `IAS\<version>\lib\ias-extension-api`.

3. Create a subclass of Manipulator and specify the PipelineComponentConfiguration subclass that the extension uses.

For example:

```
public class SubstringManipulator extends Manipulator<SubstringManipulatorConfig>{
}

```

4. Add an @IasManipulator annotation to the Manipulator class and any attributes as described above.

For example:

```
@IasManipulator(
    supportsIncrementals=true,
    deleteRecordsBypassManipulator = true,
    displayName="Substring Manipulator",
    description="Generates a new property that is a substring of another property value")
public class SubstringManipulator extends Manipulator<SubstringManipulatorConfig>

```

5. Implement the getConfigurationClass() method to return the appropriate PipelineComponentConfiguration subclass.

For example:

```
public Class<SubstringManipulatorConfig> getConfigurationClass() {
    return SubstringManipulatorConfig.class;
}

```

6. Implement the createManipulatorRuntime() method to create an implementation of the ManipulatorRuntime class.

For example:

```
public ManipulatorRuntime createManipulatorRuntime(
    SubstringManipulatorConfig configuration, PipelineComponentRuntimeContext
    context) {
    return new SubstringManipulatorRuntime(context, configuration);
}

```

7. Implement the getRuntimeClass() method to return the runtime class the manipulator creates.

For example:

```
public Class<SubstringManipulatorRuntime> getRuntimeClass() {
    return SubstringManipulatorRuntime.class;
}

```

8. Optionally, override the deleteInstance() method. IAS Server calls deleteInstance() when it removes an extension from an acquisition. In this method, you can perform any clean up that is necessary when IAS Server calls deleteInstance() to remove the extension from an acquisition. The default implementation of deleteInstance() is empty.

Example of a manipulator extension

To see many of the steps above, refer to the sample manipulator extension in <install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\manipulator\substring\SubstringManipulator.java.

[Creating a pipeline component configuration class for a manipulator](#)

Creating a runtime class for a manipulator

Creating a pipeline component configuration class for a manipulator

A manipulator requires a `PipelineComponentConfiguration` class to describe the extension's configuration, to validate the manipulator's configuration, and to determine whether a full acquisition is required by the manipulator.

To create a manipulator configuration class:

1. In the Java project that contains the `Manipulator` implementation, create a subclass of `PipelineComponentConfiguration`.

For example:

```
public class SubstringManipulatorConfig extends
PipelineComponentConfiguration<SubstringManipulatorConfig> {
}
```

2. Add each field that you want available as a configuration property in the manipulator.

For example:

```
private String mSourceProperty;

private String mTargetProperty;

private int mStartIndex;
```

3. Add an annotation to each field. The annotation type must match the field's data type. See package `com.endeca.ias.extension.annotation` in the *IAS Extension API Reference (Javadoc)* to determine which annotations have required attributes and to determine which attributes are appropriate for the field. Also see, [About IAS annotations on page 9](#).

For example:

```
@StringProperty(isRequired=true, name="sourceProperty", displayName="Source Property")
private String mSourceProperty;

    @StringProperty(isRequired=true, name="targetProperty", displayName="Target Property")
private String mTargetProperty;

    @IntegerProperty(isRequired=false, name="startIndex", displayName
="Substring Start Index",
description="Substring start index (zero based)", defaultValue="0")
private int mStartIndex;
```

4. If you want to order fields within a single group, add a `@ConfigurationGroupOrder` annotation to the `PipelineComponentConfiguration` class and then add a nested `@ConfigurationGroup` annotation.

For example, here is one group of fields that display in order — `sourceProperty`, `targetProperty`, `length` and `startIndex`:

```
@ConfigurationGroupOrder({
    @ConfigurationGroup(propertyOrder
={ "sourceProperty", "targetProperty", "length", "startIndex" })
})
```

```
public class SubstringManipulatorConfig extends
PipelineComponentConfiguration<SubstringManipulatorConfig> {
}

```

5. If you want to order multiple groups, add additional `@ConfigurationGroup` annotations within `@ConfigurationGroupOrder` for each group that you want ordered.
6. Optionally, override the default implementation of `isFullAcquisitionRequired()`. The default implementation determines whether a configuration change should force full acquisition the next time an acquisition is run by comparing the old `PipelineComponentConfiguration` and the new `PipelineComponentConfiguration` using the `equals()` method. The default implementation of the `equals()` method uses reflection to compare all non-transient fields for equality.

You can write code that checks a specific property to determine if a full acquisition is required (rather than check the entire `PipelineComponentConfiguration`). If you want to force a full acquisition, write code that always returns `true`.

7. Optionally, override the default implementation of `validate()`. IAS Server performs data type and constraint validation (constraints may include `minValue` and `maxValue` for integer properties). Any code you write in `validate()` performs additional custom validation.

If validation fails, the `PipelineComponentConfiguration.validate()` method returns a collection `ValidationFailure` objects.

8. If it is necessary for unit testing or the implementation of the manipulator runtime, you may also need to write getter and setter methods for each field that you added.

Example of a pipeline component configuration for a manipulator

To see many of the steps above, refer to the sample data source extension in `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\manipulator\substring\SubstringManipulatorConfig.java`.

Creating a runtime class for a manipulator

The `ManipulatorRuntime` is the runtime representation of a manipulator instance. The `ManipulatorRuntime` is created by `Manipulator.createManipulatorRuntime()` and exists for the life span of the manipulator.

IAS Server creates and passes a `PipelineComponentRuntimeContext` class to `Manipulator.createManipulatorRuntime()`. The `PipelineComponentRuntimeContext` specifies an output channel, error channel, a state directory, and several other runtime properties.

The `ErrorChannel.discard()` methods discards any invalid records from record processing. Also, in addition to discarding records, the `ErrorChannel` class processes exceptions that you catch. This processing includes incrementing the appropriate metric for a record and also logging a record in the `ias-service.log` file. The `ErrorChannel` logs events at level `WARN` and higher.

To create a runtime class for a manipulator:

1. In the Java project that contains the `Manipulator` implementation, create a subclass of `ManipulatorRuntime`.

For example:

```
public class SubstringManipulatorRuntime extends ManipulatorRuntime {
}

```

2. Implement the `ManipulatorRuntime` constructor.
3. Optionally, override the default implementation of `ManipulatorRuntime.checkFullAcquisitionRequired()` to allow each manipulator in an acquisition to indicate whether it requires a full acquisition. This check could be necessary if a manipulator has state-based dependencies that should force a full acquisition. For guidance, see [Supporting incremental acquisition in a data source on page 21](#).
4. Optionally, override the default implementation of `prepareForAcquisition(AcquisitionMode)` if the manipulator has to prepare state to process records that result from an incremental acquisition. IAS Server passes in an acquisition mode of either `FULL_ACQUISITION` or `INCREMENTAL_ACQUISITION` based on the results of running `checkFullAcquisitionRequired()`.
5. Implement the abstract method `processRecord()` to define how to manipulate records. The implementation depends the manipulation you wish to perform.
6. Optionally, call `ErrorChannel.discard()` as necessary to discard any records that are invalid or have errors.
7. Call `OutputChannel.output()` for each record that has been processed by `processRecord()`.

For example:

```
getContext().getOutputChannel().output(record);

```

A manipulator should not modify any records that have already been output by `output()`. If you are doing significant processing between calls to `output()`, you may want to periodically call `PipelineComponentRuntimeContext.isStopped()` to see if any requests to stop the acquisition have been made while `OutputChannel.output()` is running.

8. Optionally, implement `onInputClose()` to perform any cleanup or post-processing after `processRecord()` finishes processing the last record.
9. Optionally, handle requests to stop an acquisition by providing a mechanism to stop an extension's runtime object in a timely way. This may include polling `PipelineComponentRuntimeContext.isStopped()` and may include overriding `PipelineComponentRuntime.stop()`. For guidance, see [Stopping an extension when an acquisition stops on page 36](#).
10. Optionally, override `PipelineComponentRuntime.endAcquisition()` to clean up any resources used by `PipelineComponentRuntime` or `ManipulatorRuntime` and also clean up any state-based dependencies. For guidance, see [Cleaning up resources used by an extension on page 37](#).

Example of a manipulator runtime

To see many of the steps above, refer to the sample manipulator extension in `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\manipulator\substring\SubstringManipulatorRuntime.java`.

Supporting incremental acquisition in a manipulator

A manipulator can be implemented to support record input from an incremental acquisition of a data source. Broadly speaking, *support* means that a manipulator can take record input from an incremental acquisition, process the records, and generate correct output.

More specifically, a manipulator generates correct output if it can produce records from an incremental acquisition that represent the delta between two full acquisitions.

This situation is more complicated than supporting record input from a full acquisition. During a full acquisition, a manipulator processes every record as a new record. During an incremental acquisition, a manipulator processes incremental changes that may include added records, updated records, and deleted records.

Here is a simple example that illustrates how a manipulator generates correct output for changes (creates, updates, and deletes) to an input record named record A:

- Record A is passed into a manipulator. Based on the new record A, the manipulator creates new records A1 and A2.
- An updated record A is passed into a manipulator. Based on the update to record A, the manipulator updates records A1 and A2.
- Record A is deleted from the data source. A corresponding delete record is passed into the manipulator. Based on the delete record, the manipulator deletes records A1 and A2.

Generating correct output

In simple cases, a manipulator produces correct output and can be annotated with `"supportsIncrementals=true"` if it meets all of the following criteria:

- Does the manipulator output only the records it receives? That is, does it always output records with a `recordId` that was input to it?
- Does the manipulator output all the records that it receives?
- Does the manipulator generate output for one record that depends only on the input of the same record and not on any other records?
- Does the manipulator produce the same output over time in response to a given input record?

If the answer is yes to these four questions, then a manipulator can be annotated with `"supportsIncrementals=true"`. If the answer is no to any one of these questions, then further analysis is required to determine whether correct output can be produced, and further work is required to produce correct output. This work typically involves maintaining state.

If the answer is yes to all but the second question, it may be possible to support incremental input with only a little extra work to address the records that the manipulator does not output. To be correct, a manipulator should emit a delete record for any record that it had previously output but is now not being output. (Recall that a record marked for deletion has an `Endeca.Action` property set to `DELETE`.)

This would require the manipulator to track what records it had previously output. To make such implementation easier, a manipulator can replace input records that it should not output with a delete record instead.

For example, suppose a manipulator filters records to include only those records that contain a `color` property that is set to red. Record A has the property `color` set to red, so the manipulator includes record A. In a subsequent incremental acquisition, record A has changed so that now the `color` property is set to blue.

Record A no longer meets the manipulator's filtering criteria. Instead of dropping the record, the manipulator emits a delete record for record A.

Delete records can bypass a manipulator

If a delete record does not require manipulator processing, you can set `deleteRecordsBypassManipulator` to `true` and any delete record will bypass the manipulator and be routed directly to a manipulator's `OutputChannel`. Setting `deleteRecordsBypassManipulator` to `true` allows you to avoid writing special case code to handle delete records.

Setting `deleteRecordsBypassManipulator` to `false` routes delete records into the manipulator for processing.

[About IAS annotations](#)

[Creating a runtime class for a manipulator](#)



Chapter 4

Extension life cycle and threading

This section describes several life cycle and threading issues to consider when implementing an extension to the Integrator Acquisition System.

[Life cycle of a data source](#)

[Life cycle of a manipulator](#)

[About threading](#)

Life cycle of a data source

The following actions indicate key points in the life cycle of a data source. The events take place after a data source extension has been implemented, packaged, and installed into the Integrator Acquisition System.

Action	Corresponding life cycle event
The Endeca IAS Service starts.	IAS Server instantiates any <code>DataSource</code> that is installed and correctly annotated.
The following items perform the same action. <ul style="list-style-type: none">In the IAS Server API - Calling either <code>IasCrawler.createCrawl()</code> or <code>IasCrawler.updateCrawl()</code>.In the <code>ias-cmd</code> utility - Running either <code>createCrawls</code> or <code>updateCrawls</code>.	IAS Server instantiates the <code>PipelineComponentConfiguration</code> specified by the <code>DataSource</code> . IAS Server performs data type and constraint validation and also calls <code>PipelineComponentConfiguration.validate()</code> to execute any additional custom validation code you wrote in the method's implementation.

Action	Corresponding life cycle event
<p>The following items perform the same action.</p> <ul style="list-style-type: none"> In the IAS Server API - Calling <code>IasCrawler.startCrawl()</code>. In the <code>ias-cmd</code> utility - Running <code>startCrawl</code>. 	<p>IAS Server re-creates the <code>PipelineComponentConfiguration</code> specified by the <code>DataSource</code> and re-validates the <code>PipelineComponentConfiguration</code> by calling <code>validate()</code>. (IAS Server injects configuration property values via reflection before calling <code>validate()</code>.)</p> <p>IAS Server calls <code>createDataSourceRuntime()</code> on the <code>DataSource</code> instance.</p> <p>The resulting <code>DataSourceRuntime</code> exists for the duration of the acquisition.</p> <p>If an incremental acquisition is requested, IAS Server calls <code>checkFullAcquisitionRequired()</code> on the new <code>DataSourceRuntime</code>. The check returns a Boolean value to indicate whether a full acquisition is required or not.</p> <p>If any data source or manipulator in the IAS pipeline returns <code>true</code>, IAS Server calls <code>DataSourceRuntime.runFullAcquisition()</code>.</p> <p>If all extensions return <code>false</code>, IAS Server calls <code>IncrementalDataSourceRuntime.runIncrementalAcquisition()</code>.</p> <p>After all records have been processed by all extensions, IAS Server calls <code>PipelineComponentRuntime.endAcquisition()</code>.</p> <p>IAS Server passes in an <code>AcquisitionEndState</code> value to <code>endAcquisition()</code> to indicate whether the extension succeeded, failed, or requires a full acquisition to recover from a failure.</p>
<p>The following items perform the same action.</p> <ul style="list-style-type: none"> In the IAS Server API - Calling <code>IasCrawler.deleteCrawl()</code>. In the <code>ias-cmd</code> utility - Running <code>deleteCrawl</code>. 	<p>IAS Server calls <code>PipelineComponent.deleteInstance()</code> when a data source is removed from an application as part of updating or deleting an acquisition, and <code>deleteInstance()</code> is also called if the data source type of the acquisition has changed. (The data source type is represented by a <code>ModuleId</code> setting in the IAS Server API).</p>

Life cycle of a manipulator

The following actions mark key points in the life cycle of a manipulator. The events take place after a manipulator extension has been implemented, packaged, and installed into the Integrator Acquisition System.

Action	Corresponding life cycle event
The Endeca IAS Service starts.	IAS Server instantiates any <code>Manipulator</code> that is installed and correctly annotated.
<p>The following items perform the same action.</p> <ul style="list-style-type: none"> In the IAS Server API - Calling either <code>IasCrawler.createCrawl()</code> or <code>IasCrawler.updateCrawl()</code>. In the <code>ias-cmd</code> utility - Running either <code>createCrawls</code> or <code>updateCrawls</code>. 	<p>IAS Server instantiates a <code>PipelineComponentConfiguration</code> specified by the <code>Manipulator</code> for each <code>Manipulator</code> occurrence in a given acquisition.</p> <p>IAS Server performs data type and constraint validation and also calls <code>PipelineComponentConfiguration.validate()</code> to execute any additional custom validation code you wrote in the method's implementation.</p>

Action	Corresponding life cycle event
<p>The following items perform the same action.</p> <ul style="list-style-type: none"> • In the IAS Server API - Calling <code>IasCrawler.startCrawl()</code>. • In the <code>ias-cmd</code> utility - Running <code>startCrawl</code>. 	<p>IAS Server re-creates the <code>PipelineComponentConfiguration</code> and re-validates the <code>PipelineComponentConfiguration</code> by calling <code>validate()</code>. (IAS Server injects configuration property values via reflection before calling <code>validate()</code>.)</p> <p>IAS Server calls <code>createManipulatorRuntime()</code> on the <code>Manipulator</code> instance.</p> <p>The <code>ManipulatorRuntime</code> exists for the duration of the acquisition.</p> <p>If an incremental acquisition is requested, IAS Server calls <code>checkFullAcquisitionRequired()</code> on the new <code>ManipulatorRuntime</code>. The check returns a Boolean value to indicate whether a full acquisition is required or not before record processing. If any one data source or manipulator in a IAS pipeline returns <code>true</code>, IAS runs a full acquisition. All extensions in a IAS pipeline must return <code>false</code> for IAS to run an incremental acquisition.</p> <p>Before processing records, IAS Server calls <code>prepareForAcquisition()</code> to prepare state.</p> <p>During the acquisition, IAS Server calls <code>ManipulatorRuntime.processRecord()</code> concurrently for separate records to manipulate every input record.</p> <p>After all records have been processed by all extensions, IAS Server calls <code>ManipulatorRuntime.onInputClose()</code> and then calls <code>PipelineComponentRuntime.endAcquisition()</code> on each manipulator.</p> <p>IAS Server passes in an <code>AcquisitionEndState</code> value to <code>endAcquisition()</code> to indicate whether the extension succeeded, failed, or requires a full acquisition to recover from a failure.</p>
<p>The following items perform the same action.</p> <ul style="list-style-type: none"> • In the IAS Server API - Calling <code>IasCrawler.deleteCrawl()</code> or • In the <code>ias-cmd</code> utility - Running <code>deleteCrawl</code>. 	<p>IAS Server calls <code>PipelineComponent.deleteInstance()</code> when a manipulator is removed from an acquisition.</p>

About threading

Data sources and manipulators must be thread safe.

The `stop()` method can be called concurrently when any of the following methods are running:

- `DataSourceRuntime.runFullAcquisition()`
- `ManipulatorRuntime.processRecord()`
- `ManipulatorRuntime.onInputClose()`
- `IncrementalDataSourceRuntime.runIncrementalAcquisition()`

Recommendations for data sources

The requirement to be thread safe has a few implementation implications for data sources:

- Any state that is shared with `runFullAcquisition()` needs to be synchronized with `stop()`. State may be share with `checkFullAcquisitionRequired()` and the binary content interfaces (`BinaryContentFileProvider` and `BinaryContentInputStreamProvider`).
- If you are supporting text extraction by implementing either the `BinaryContentFileProvider` interface or the `BinaryContentInputStreamProvider` interface, the data source must be thread safe because IAS Server calls `BinaryContentFileProvider.getBinaryContentFile()` or `BinaryContentInputStreamProvider.getBinaryContentInputStream()` from multiple threads.

Recommendations for manipulators

The requirement to be thread safe has a few implementation implications for manipulators:

- If possible, use only local variables or final immutable fields.
- Persist internal state across calls to `processRecord()` or `onInputClose()` only if it is absolutely necessary. If it is necessary, access state in a synchronized way.

For optimal performance, it is a good idea to minimize the time you hold locks in `processRecord()`.

Manipulators should not hold locks when calling `OutputChannel.output()` from `processRecord()`. The call to `output()` may take a while to return, which blocks other threads that are concurrently calling `processRecord()`. One way of holding locks is by using the Java `synchronize` keyword for a method. However, synchronizing `processRecord()` adversely affects performance. Synchronizing effectively makes the manipulator single threaded by preventing other threads from entering `processRecord()`.

Configuration and context synchronization

As part of the implementation of an extension, the IAS Server passes in a `PipelineComponentConfiguration` object and a `PipelineComponentRuntimeContext` object to either `DataSource.createDataSourceRuntime()` (in the case of data sources) and `Manipulator.createManipulatorRuntime()` (in the case of manipulators). The IAS Server does not modify the `PipelineComponentConfiguration` after `createManipulatorRuntime()` or `createDataSourceRuntime()` has been called.

When the IAS Server runs an acquisition, the `PipelineComponentRuntimeContext` and everything accessible from it is thread safe.

[Creating a runtime class for a data source](#)

Creating a runtime class for a manipulator



Chapter 5

Common implementation tasks

This section describes general implementation tasks that are common to both data source and manipulator extensions.

Stopping an extension when an acquisition stops

Cleaning up resources used by an extension

Storing state information for an extension

Exceptions that trigger fatal and non-fatal failures

Enabling logging in an extension

Unit testing an extension

Packaging an extension into a plugin

Installing a plugin into IAS

Running an extension

Uninstalling a plugin from IAS

Stopping an extension when an acquisition stops

When an acquisition stops, it may take time for an extension within an acquisition to stop. Therefore, Oracle recommends that you provide a mechanism to stop an extension's runtime object in a more timely way (recall that a runtime object is either a `DataSourceRuntime` or a `ManipulatorRuntime`).

There are several requests or conditions that may cause IAS Server to stop an acquisition:

- An IAS data developer requests a stop by running the `stopCrawl` task of the IAS Server Command-line Utility (`ias-cmd`).
- An IAS data developer requests a stop by calling the `stopCrawl()` method of the IAS Server API.
- An acquisition may abort because it encountered a fatal error during the acquisition processing or record manipulation processing.

When an acquisition stops, it has the following effects in the IAS Extension API:

- Calls to `PipelineComponentRuntimeContext.isStopped()` return `true`.
- Calls to `OutputChannel.output()` throw a `PipelineStoppedException` exception.
- Calls to `ErrorChannel.discard()` throw a `PipelineStoppedException` exception.
- IAS Server calls `PipelineComponentRuntime.stop()` on all data source and manipulator extensions in the acquisition.

There are several mechanisms to stop an extension's runtime in a more timely way:

- You can poll the `PipelineComponentRuntimeContext.isStopped()` method, and if it returns `true`, you throw a `PipelineStoppedException` and let the exception propagate through the system. There is example code that implements `isStopped()` in the CSV data source extension provided with the integrator Acquisition System.
- In addition to polling, you can override `PipelineComponentRuntime.stop()` on an extension. Your implementation of `stop()` should perform any tasks that help the extension stop more quickly, for example, terminating any pending network requests or closing or cancelling any output requests. This approach is particularly useful in situations where the extension is doing time-consuming work between calls to `OutputChannel.output()`.

Cleaning up resources used by an extension

The IAS Server calls `endAcquisition()` after all extensions in a acquisition have completed data acquisition and record processing. Your implementation of `endAcquisition()` can override `PipelineComponentRuntime.endAcquisition()` to perform any necessary cleanup for an extension.

The `endAcquisition()` method has an input parameter, an `AcquisitionEndState` object that indicates the state of the acquisition process when it ended. The `AcquisitionEndState` can have one of the following enumerated values:

- `SUCCESS`
- `FAILURE`
- `FULL_ACQUISITION_RECOVERY_REQUIRED`

Your implementation of `endAcquisition()` should account for each of the values of `AcquisitionEndState` that IAS might pass to the extension.

In general, this means if IAS passes `endAcquisition()` a value of `SUCCESS`, the extension typically maintains any state changes it made during the acquisition. For example, this could include a data source extension writing out timestamp information for a successful acquisition and later reading in that timestamp for a subsequent acquisition in order to determine the incremental difference between acquisitions.

If IAS passes `endAcquisition()` a value of `FAILURE`, the extension typically reverts any state changes it made during the acquisition. For example, this could include reverting timestamp information for a failed acquisition.

If IAS passes `endAcquisition()` a value of `FULL_ACQUISITION_RECOVERY_REQUIRED`, the extension could either maintain or revert state information. You want to do whatever is necessary to prepare for a new acquisition and also throw a `FullAcquisitionRecoveryRequiredException` exception.

Storing state information for an extension

The Integrator Acquisition System automatically creates directories under `<install path>\IAS\workspace\state` that you can use to store state information for a data source or manipulator extension. An extension can read, write, or delete state information from these directories as necessary.

A data source may require state information to run an incremental acquisition. For example, by relying on a file that stores the last date that the data source read from a CMS. The data source may later read from the file and pass in the date in order to run an incremental acquisition.

The path for a data source's state directory is `<install path>\IAS\workspace\state\ias\crawls\<crawlId>\source`.

The path for a manipulator's state directory is `<install path>\IAS\workspace\state\ias\crawls\<crawlId>\manipulators\<manipulatorId>`.

At end of an extension's life cycle, IAS calls `PipelineComponent.deleteInstance()` and then IAS also deletes the contents of the `state` directory.

Supporting incremental acquisition in a data source

Exceptions that trigger fatal and non-fatal failures

It is important to distinguish exceptions that indicate a fatal error and therefore stop an acquisition from exceptions that indicate a non-fatal error and continue acquisition processing.

Methods that cause a fatal error throw a `FatalExecutionException`. Fatal errors indicate a more global problem. Methods that cause a non-fatal error throw an `ExecutionException`. Non-fatal errors indicate a more local problem.

If an `ExecutionException` is thrown by a manipulator extension, the record is discarded from processing. If an `ExecutionException` is thrown by a data source extension, all records acquired to that point are discarded.

Enabling logging in an extension

You can enable logging in an extension to provide diagnostic information about the extension as it runs in an acquisition. An extension writes to the `<install path>\IAS\workspace\logs\ias-service.log` file using one of the common logging frameworks.

Supported logging frameworks

The Integrator Acquisition System supports the following logging frameworks:

- SLF4J
- Apache Commons Logging
- `java.util.logging`
- `log4j`

Oracle recommends the SLF4J framework because its parameterized logging minimizes the performance impact of disabled logging statements. For details, see the SLF4J documentation at <http://www.slf4j.org>.

Integrating logging

Integration is largely transparent. You import the logging framework into the `PipelineComponentRuntime` and call `getLogger()`. If any logging requests come in from any of the frameworks, the Integrator Acquisition System detects the requests and redirects them to `log4j`, which IAS then uses to write to `ias-service.log`.

For example, if you are using SLF4J, integration is similar to the following:

```
import org.slf4j.LoggerFactory
```

```
...
LoggerFactory.getLogger(getClass()).info("A logging message.");
```

Changing log levels

You can change log levels by modifying `log4j.logger.<loggerName>` properties in `<install path>\IAS\workspace\conf\ias-service-log4j.properties`. The default log level is set to `WARN`.

For example, this `log4j` entry sets the log level to `DEBUG` for the sample substring manipulator.

```
log4j.logger.com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulatorRuntime=DEBUG
```

After you modify, save, and close the `ias-service-log4j.properties` file, you must restart the Endeca IAS Service for the change to take effect.

The log level setting in the data source's configuration

Data sources include several advanced configuration properties by default: you do not need to implement these properties as Java fields in an `PipelineComponentConfiguration` class. A data developer can set these properties using the IAS Server Command-line Utility and the IAS Server API. The log level setting applies to high-level aspects of a data acquisition, such as logging crawl history, but the log level setting does not apply to the data source extension itself.

Unit testing an extension

Unit testing an IAS extension involves writing mock Java classes, writing unit tests, and running these tests through a tool of your choice.

Developers should mock out the following interfaces for their unit tests:

- `PipelineComponentContext`
- `PipelineComponentRuntimeContext`
- `OutputChannel`
- `ErrorChannel`

Oracle recommends the JUnit testing framework but it is not required. For information about using the JUnit framework, see <http://junit.org/>.

There is a set of sample JUnit tests included with the sample implementation. These are available in `<install path>\IAS\<version>\sample\ias-extensions`.

About testing the `PipelineComponentConfiguration`

The general strategy for unit testing the `PipelineComponentConfiguration` involves:

- Constructing an `PipelineComponentConfiguration` and invoking `setPipelineComponentContext` with a mock `PipelineComponentContext`.
- Invoking the methods on the implementation and ensuring correct behavior.

About testing the `DataSourceRuntime`

The general strategy for unit testing the `DataSourceRuntime` involves:

- Constructing a `DataSourceRuntime` with a mock `PipelineComponentRuntimeContext` that contains a mock `OutputChannel`, a mock `ErrorChannel`, and a mock `PipelineComponentContext`.
- Constructing an `IncrementalDataSourceRuntime` if a data source extension determines its own incrementals.
- Invoking `DataSourceRuntime.runFullAcquisition()` and verifying that the correct records are output to the mock `OutputChannel`.
- Invoking `IncrementalDataSourceRuntime.runIncrementalAcquisition()` and verifying that the correct records are output to the mock `OutputChannel`. This is necessary if you constructed a `IncrementalDataSourceRuntime`.
- Invoking `ErrorChannel.discard()` as necessary to discard any records that are invalid or have errors, and verifying that the invalid records are discarded to the mock `ErrorChannel`.

About testing the `ManipulatorRuntime`

The general strategy for unit testing the `ManipulatorRuntime` involves:

- Constructing a `ManipulatorRuntime` with a mock `PipelineComponentRuntimeContext` that contains a mock `OutputChannel`, a mock `ErrorChannel`, and a mock `PipelineComponentContext`.
- Invoking `ManipulatorRuntime.checkFullAcquisitionRequired()` and verifying that the manipulator correctly identifies whether a full acquisition is required.
- Invoking `ManipulatorRuntime.prepareForAcquisition()` and verifying that the manipulator performs any required preparation before the acquisition starts.
- Constructing test records, passing the test records to `ManipulatorRuntime.processRecord()` and verifying that the correct records are output to the mock `OutputChannel`.
- Invoking `ErrorChannel.discard()` as necessary to discard any records that are invalid or have errors, and verifying that the invalid records are discarded to the mock `ErrorChannel`.
- Invoking `ManipulatorRuntime.onInputClose()` and verifying that the manipulator performs any required clean up after the acquisition completes its record processing.

Packaging an extension into a plugin

A plugin is a JAR or set of JAR files that contain an extension or set of extensions. After implementing one or more extensions, you package them into one or more JAR files and distribute them, and any dependent JAR files, to an IAS data developer.

This topic assumes you have already implemented one or more extensions.

To package an extension into a plugin:

1. In your Java project, build the classes for an extension into a JAR or a set of JARs.

For an example, see the procedure in Building the sample extensions.



Note: There are several IAS JAR files that are available to the extensions in a plugin and do not need to be packaged as dependencies with a plugin. These JAR files include the following:

- ias-extension-api.jar
- eidi-api-common.jar
- commons-logging.jar
- log4j.jar
- slf4j-api.jar

Installing a plugin into IAS

After receiving a plugin (a JAR or set of JAR files) from a plugin developer, an IAS data developer installs the plugin into IAS.

The Integrator Acquisition System detects each plugin and validates the extensions within it by checking the uniqueness of extension IDs and by checking for the presence of an annotation of either `@IasDataSource` or `@IasManipulator` for each extension.

To install a plugin into IAS:

1. Stop Endeca IAS Service.
2. Navigate to `<install path>\IAS\<version>\lib\ias-server-plugins` and create a `<plugin-name>` subdirectory for each plugin.

For example: `IAS\3.1.0\lib\ias-server-plugins\ODBCDataSourceExt`

3. Copy the plugin JAR or JARs, and any dependent JAR files, to `<install path>\IAS\<version>\lib\ias-server-plugins\<plugin-name>`.
4. Repeat the steps above as necessary for multiple plugins.
5. Start Endeca IAS Service.

You can confirm that an extension is installed by running the `listModules` task of the IAS Server Command-line Utility and specifying a `moduleType` of either `SOURCE` or `MANIPULATOR`. The task returns the installed modules. For example, this task shows that a custom data source named `Sample Data Source for testing` is installed:

```
ias-cmd listModules -t SOURCE
Sample Data Source
 *Id: Sample Data Source
 *Type: SOURCE
 *Description: Sample Data Source for testing

File System
 *Id: File System
 *Type: SOURCE
 *Description: No description available for File System
 *Capabilities:
  *Binary Content Accessible via FileSystem
```

```
*Data Source Filter
*Has Binary Content
*Expand Archives
```

Running an extension

After you have implemented, unit tested, packaged, and installed an extension, you can run it as part of an acquisition to ensure it works as expected.

Before starting this procedure, you must have already packaged the extension into a plugin and installed the plugin into the Integrator Acquisition System. For installation instructions, see the *Integrator Acquisition System Installation Guide*.

To run an extension:

1. On the machine where IAS is installed, confirm that the plugin is installed by running the `listModules` task of `ias-cmd`.
 - If you installed a plugin with a data source, specify a `moduleType` of `SOURCE` to the `listModules` task.
 - If you installed a plugin with a manipulator, specify a `moduleType` of `MANIPULATOR` to the `listModules` task.

For example:

```
ias-cmd listModules -t SOURCE
Blob Database
*Id: com.endeca.ias.extension.sample.datasources.blob.BlobDataSource
*Type: SOURCE
*Description: Reads a database table containing documents
```

2. In the crawl configuration XML file, specify configuration properties as appropriate for the data source.
3. In the crawl configuration XML file, specify configuration properties as appropriate for any manipulators.
4. Start the crawl by running the `startCrawl` task of `ias-cmd`.
5. When the crawl completes, you can confirm that new records exist in a Record Store instance by running the `read-baseline` task of `recordstore-cmd`.

For example:

```
recordstore-cmd read-baseline -a Test
```

Uninstalling a plugin from IAS

Before you begin the uninstall process, back up any plugin JAR files that you want to retain from the `IAS\<version>\lib\ias-server-plugins` directory.

To uninstall a plugin from IAS:

1. Stop the Endeca IAS Service.
2. Navigate to `IAS\<version>\lib\ias-server-plugins`.
3. Delete the `<plugin-name>` subdirectory for each plugin.

For example: `IAS\<version>\lib\ias-server-plugins\ODBCDataSourceExt`

4. Repeat the steps above as necessary for multiple plugins.
5. Start Endeca IAS Service.



Sample extensions

This section describes the sample extensions that are installed with the Integrator Acquisition System.

[About the sample extensions](#)

[Sample extensions files and directories](#)

[Building the sample extensions](#)

[Unit testing the sample extensions](#)

[Installing the sample plugin into IAS](#)

[Running the sample CSV File data source](#)

[Running the sample Substring manipulator](#)

[Running the sample Blob Database data source](#)

[Running the sample Document Directory data source](#)

[Running the sample Change Tracking data source](#)

About the sample extensions

There are sample extensions that illustrate how to build, test, install, and run custom extensions to IAS:

- A sample data source that reads from a database table containing blobs. See `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\blob`.
- A sample data source that reads a folder of files. See `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\directory`.
- A sample data source that reads from a comma-separated file. This sample also illustrates how to use `ErrorChannel.discard()`. See `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\csv`.
- A sample manipulator that generates a new property based on a substring of another property value. See `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\manipulator\substring`.
- A sample data source that performs incremental acquisitions from a database. See `<install path>\IAS\<version>\sample\ias-extensions\src\main\com\endeca\ias\extension\sample\datasource\incremental`.

The sample data sources rely on data that is installed in `<install path>\IAS\<version>\sample\ias-extensions\data`.

Sample extensions files and directories

The sample extensions have the following directory structure:

```
..\..\IAS\3.1.0\sample\ias-extensions
    \build
    \data
    \lib
    \src
    .classpath
    .project
    build.xml
```

The contents are as follows:

- `build` – Contains the generated classes and JAR files for the extensions. This directory is not present after installing the Integrator Acquisition System. Building the extensions creates this directory.
- `data` – Contains a `data.csv` file for use with the CSV data source, a database for use with the Blob Database data source, and a folder of documents for the Document Directory data source.
- `lib\ias` – Contains the IAS Extension API and API dependencies.
- `lib\ias-test` – Contains dependencies for the sample extension unit tests.
- `lib\main` – Contains dependencies for the sample extensions.
- `lib\test` – Contains dependencies for the sample extension unit tests.
- `src` – Contains the Java source files and the unit tests for the extensions.
- `.classpath` – The classpath file for the Eclipse project.
- `.project` – The Eclipse project file for the extensions.
- `build.xml` – The Ant build file for the extensions.

Building the sample extensions

You can build the sample extensions using either Ant or Eclipse.



Note: The JUnit Testing Framework has been removed from this software distribution. Please download and install JUnit version 3.8.1 before building `ias-sample-extensions` using Ant. To download JUnit, see <http://sourceforge.net/projects/junit/files/junit/3.8.1/junit3.8.1.zip/download>. Once you download JUnit, extract the JAR, and place it in the `ias-extensions\lib\test` directory.

Building the sample extensions with Ant

The Ant build file has the following targets:

- `clean` – Cleans previous build output.
- `compile` – Compiles source code for the sample extensions.
- `package` – Builds the JAR file for the plugin.
- `test-compile` – Compiles source code for the unit tests.
- `test-run` – Runs unit tests. You can skip this target by passing the `-DskipTests=true` flag to Ant.
- `dist` – Cleans previous build output, builds the plugin, and runs unit tests.

To build the sample extensions with Ant:

1. Open a command prompt and navigate to the `<install path>\IAS\<version>\sample\ias-extensions` directory.
2. Run the following command:

```
ant dist
```

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\sample\ias-extensions>ant dist
```

The Ant build file runs all the targets listed above including the unit tests for the extensions. The build output is packaged to `<install path>\IAS\<version>\sample\ias-extensions\build\ias-sample-extensions`.

Building the sample extensions with Eclipse

To build the sample extensions with Eclipse:

1. Start Eclipse and select **File>Import...>General>Existing Projects into Workspace>Next**.
2. Browse to the project in `<install path>\IAS\<version>\sample\ias-extensions` and click **OK>Finish**.
3. In the Project Explorer, right-click on the `ias-sample-extensions` directory and select **Export**.
4. Select **Java>JAR file** and click **Next**.
5. In the **JAR File Specification** dialog:
 - Select the resources to export (the `ias-sample-extensions` package).
 - If it is not already checked, select **Export generated class files and resources**.
 - Select an export destination.
 - Select **Compress the contents of the JAR file**.
 - Click **Next** when you are satisfied with the specification.
6. In the **JAR Packaging Options** dialog, select any options that you want and click **Finish**.

Building the sample extensions using Eclipse to run Ant

To build the sample extensions using Eclipse to run Ant:

1. Start Eclipse and import the `ias-extensions` project as in steps 1 - 2 above.
2. In the Project explorer, expand the `ias-sample-extensions` directory.
3. Right-click `build.xml` and select **Run As>Ant Build**.

The Ant build file runs all the targets listed above including the unit tests for the extensions. The build output is packaged to `<install path>\IAS\<version>\sample\ias-extensions\build\ias-sample-extensions`.

Unit testing the sample extensions

You can separately unit test the sample extensions using Ant or Eclipse. Remember that building the extensions with `ant dist` also runs the `ant test-run` target, so in that scenario, the tests will already have run.

Unit testing the sample extensions with Ant

To unit test the sample extensions with Ant:

1. Open a command prompt and navigate to the `ias-extensions` directory.
2. Issue the following command to compile and unit test the sample extensions:

```
ant test-run
```

Unit testing the sample extensions with Eclipse

To unit test the sample extensions with Eclipse:

1. In the Project Explorer, expand the **ias-sample-extensions** project.
2. Right-click **src/test**, and select **Run as...>JUnit Test**.
3. On the JUnit tab, verify that the unit tests ran successfully (their status should appear green).

Unit testing the sample extensions using Eclipse to run Ant

To unit test the sample extensions using Eclipse to run Ant:

1. In the Project explorer, expand the `ias-sample-extensions` directory.
2. Right-click `build.xml` and select **Run As>Ant Build...**
3. Select the **test-run** target (de-select **dist [default]** to only run tests).
4. Click **Run**.

Installing the sample plugin into IAS

After you build the sample extensions into a plugin, install the resulting plugin files into the Integrator Acquisition System.

To install the sample plugin into IAS:

1. Stop the Endeca IAS Service.
2. If you built the extensions using Ant, copy the `ias-sample-extensions` directory from `IAS\<version>\sample\ias-extensions\build\` to `IAS\<version>\lib\ias-server-plugins`.
3. If you built the extensions using Eclipse, do the following:
 - (a) Create a new folder named `ias-sample-extensions` under `IAS\<version>\lib\ias-server-plugins`.
 - (b) Locate the JAR you built and copy it to `IAS\<version>\lib\ias-server-plugins\ias-sample-extensions\`.

- (c) Copy the JARs from IAS\<<version>\sample\ias-extensions\lib\main into IAS\<<version>\lib\ias-server-plugins\ias-sample-extensions. (These JARs are dependencies that the sample extensions require.)

4. Restart the Endeca IAS Service.

You can confirm that the extensions are installed by running the `listModules` task of the IAS Server Command-line Utility. The task returns the default IAS modules and the newly installed sample extensions. The sample extensions are listed here:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat listModules
Blob Database
*Id: com.endeca.ias.extension.sample.datasource.blob.BlobDataSource
*Type: SOURCE
*Description: Reads a database table containing documents
*Capabilities:
  *Has Binary Content

CSV File
*Id: com.endeca.ias.extension.sample.datasource.csv.CsvDataSource
*Type: SOURCE
*Description: Reads comma separated files
*Capabilities: None
...

Change Tracking Data Source
*Id: com.endeca.ias.extension.sample.datasource.incremental.ChangeTrackingDataSource
*Type: SOURCE
*Description: A data source that can incrementally crawl a change tracking data base
*Capabilities: None
....

Substring Manipulator
*Id: com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulator
*Type: MANIPULATOR
*Description: Generates a new property that is a substring of another property value
*Capabilities:
  *Supports Incrementals
...
```

Running the sample CSV File data source

After you install the sample extensions into the Integrator Acquisition System, you can configure and then run the sample CSV File data source.

To run the sample CSV File data source:

1. In your IAS installation, locate the default crawl configuration files provided in `<install path>\IAS\<<version>\sample\crawlConfigFiles`.
2. Make a copy of `delimitedFileCrawl.xml`, rename the file with a unique name for your environment, and save it to a local directory.
For example, copy `delimitedFileCrawl.xml` and save it as `sampleCSV.xml` within `<install path>\IAS\<<version>\sample\crawlConfigFiles`.
3. Open the new crawl configuration file in a text editor.

4. Configure the following settings:

Option	Description
crawlId	Specify a unique name to distinguish the crawl from others in IAS. For example, <code>sampleCSV</code> .
moduleId	Specify the module ID for a CSV File data source. This value must be set to <code>com.endeca.ias.extension.sample.data.source.csv.CsvDataSource</code> .

5. Create the following settings as new `moduleProperty` elements in the `sourceConfig` XML:

Option	Description
inputFile	Specify the path to the <code>data.csv</code> file. For example, <code>C:\Oracle\Endeca\IAS\3.1.0\sample\ias-extensions\data\data.csv</code> .
keyColumn	Specify the name of the column containing the record key: <code>P_WineID</code> .

6. Delete the other `moduleProperty` elements within `sourceConfig`.

After these steps, the `sourceConfig` XML should be as follows:

```

...
    <sourceConfig>
      <moduleId>
        <id>com.endeca.ias.extension.sample.datasources.csv.CsvDataSource</id>
      </moduleId>
      <moduleProperties>
        <moduleProperty>
          <key>inputFile</key>
          <value>C:\Oracle\Endeca\IAS\3.1.0\sample\ias-extensions\data\data.csv<
/ value>
        </moduleProperty>
        <moduleProperty>
          <key>keyColumn</key>
          <value>P_WineID</value>
        </moduleProperty>
      </moduleProperties>
      <excludeFilters />
      <includeFilters />
    </sourceConfig>
...

```

7. Configure the following settings within the `outputConfig` XML:

Option	Description
moduleId	Specify the output type for a crawl. Specify an id of Record Store.

Option	Description
host	Specify the fully qualified name of the host running the Record Store instance. The default value is localhost.
port	Specify the port of the Endeca IAS Service running the Record Store instance. The default value is 8401.
contextPath	<p>If you installed IAS into WebLogic Server, and you modified the default WebLogic context path, then specify the revised context path without including a forward slash. In WebLogic Server installations, the default value of contextPath is ias-server. If you installed IAS into Jetty, you can remove contextPath or specify an empty value.</p> <p>For example:</p> <pre data-bbox="868 898 1266 1066"><moduleProperty> <key>contextPath</key> <value>ias-server</value> </moduleProperty></pre>

8. Delete the other moduleProperty elements within outputConfig.

For example, after these steps, the outputConfig XML should be similar to the following:

```
...
  <outputConfig>
    <moduleId>
      <id>Record Store</id>
    </moduleId>
    <moduleProperties>
      <moduleProperty>
        <key>host</key>
        <value>mymachine.endeca.com</value>
      </moduleProperty>
      <moduleProperty>
        <key>port</key>
        <value>8401</value>
      </moduleProperty>
    </moduleProperties>
  </outputConfig>
...
```

9. Save and close the crawl configuration file.
10. Run the createCrawls task of ias-cmd to upload the crawl configuration file to IAS.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat createCrawls -f C:\Oracle\Endeca\IAS\
3.1.0\sample\crawlConfigFiles\sampleCSV.xml
Created crawl sampleCSV
```

11. Run the `startCrawl` task of `ias-cmd` to acquire data from the `data.csv` file.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat startCrawl -id sampleCSV
```

12. After the crawl has completed, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`. In this sample, IAS Server created 50 Endeca records and failed on 2 records (these are intentionally discarded records).

For example, this command returns the record count for the crawl:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>recordstore-cmd read-baseline -a sampleCSV -c
```

Running the sample Substring manipulator

The Substring manipulator examines a source property and creates a target property based on a substring of the source. After you install the sample extensions into the Integrator Acquisition System, you can add, configure, and run the sample Substring manipulator as part of the sample CSV File data source.

In this task, you configure the Substring manipulator to examine the `P_Wine` property (wine name) and create a five character `P_Wine_short` property.

To run the sample Substring manipulator:

1. Create an XML crawl configuration file that uses the sample CSV data source. For details, see [Running the sample CSV File data source on page 48](#).

(You will add the sample Substring manipulator to the sample CSV data source and then run the crawl.)

2. Open the XML crawl configuration file in a text editor.
3. Locate the empty `<manipulatorConfigs />` element and add the XML tags for a new manipulator:

```
<manipulatorConfigs>
  <manipulatorConfig>

  </manipulatorConfig>
</manipulatorConfigs>
```

4. In the `<manipulatorConfig>` element, and create Substring Manipulator with the following settings in the XML:

Option	Description
<code>moduleId</code>	Specify the module ID for a Substring Manipulator. This value must be set to <code>com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulator</code> .

So far, the XML should be as follows:

```
<manipulatorConfigs>
  <manipulatorConfig>
    <moduleId>
      <id>com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulator</id>
```

```

</moduleId>

</manipulatorConfig>
</manipulatorConfigs>

```

5. Create the following settings as new `moduleProperty` key/value pairs in the XML:

Option	Description
<code>sourceProperty</code>	Specify a <code>sourceProperty</code> of <code>P_Wine</code> .
<code>targetProperty</code>	Specify a <code>targetProperty</code> of <code>P_Wine_short</code> .
<code>length</code>	Specify a length of 5.
<code>id</code>	Specify a unique value for <code>id</code> , for example <code>Compute_P_Wine_short</code> .
<code>enabled</code>	Specify <code>true</code> to execute this manipulator as the crawl runs. (This setting enables or disables individual manipulators within a crawl.)

After these steps, the `manipulatorConfigs` XML should be as follows:

```

...
<manipulatorConfigs>
  <manipulatorConfig>
    <moduleId>
      <id>com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulator</id>
    </moduleId>
    <moduleProperties>
      <moduleProperty>
        <key>sourceProperty</key>
        <value>P_Wine</value>
      </moduleProperty>
      <moduleProperty>
        <key>targetProperty</key>
        <value>P_Wine_short</value>
      </moduleProperty>
      <moduleProperty>
        <key>length</key>
        <value>5</value>
      </moduleProperty>
    </moduleProperties>
    <id>Compute_P_Wine_short</id>
    <enabled>true</enabled>
  </manipulatorConfig>
</manipulatorConfigs>
...

```

6. Save and close the crawl configuration file.
7. Run the `updateCrawls` task of `ias-cmd` to upload the revised crawl configuration file to IAS.

For example:

```

C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat updateCrawls -f C:\Oracle\Endeca\IAS\
3.1.0\sample\crawlConfigFiles\sampleCSV.xml
Updated crawl sampleCSV

```

- Run the `startCrawl` task of `ias-cmd` to acquire data from the `data.csv` file and execute the Substring Manipulator to create the new target properties.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat startCrawl -id sampleCSV
```

- If desired, you can confirm that the new target properties exist by running the `read-baseline` task of `recordstore-cmd` and examining the records for the new properties.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>recordstore-cmd read-baseline -a sampleCSV
```

Running the sample Blob Database data source

The sample Blob Database data source illustrates an implementation of the `BinaryContentInputStreamProvider` interface. This interface provides an input stream so IAS Server can read the input stream and perform text extraction. After you install the sample extensions into the Integrator Acquisition System, you can configure and then run the sample Blob Database data source.

To run the sample Blob Database data source:

- In your IAS installation, locate the default crawl configuration files provided in `<install path>\IAS\<version>\sample\crawlConfigFiles`.
- Make a copy of `jdbcCrawl.xml`, rename the file with a unique name for your environment, and save it to a local directory.

For example, copy `jdbcCrawl.xml` and save it as `sampleBLOB.xml` within `<install path>\IAS\<version>\sample\crawlConfigFiles`.

- Open the new crawl configuration file in a text editor.
- Configure the following settings:

Option	Description
<code>crawlId</code>	Specify a unique name to distinguish the crawl from others in IAS. For example, <code>sampleBLOB</code> .
<code>moduleId</code>	Specify the module ID for a Blob Database data source. This value must be set to <code>com.endeca.ias.extension.sample.data source.blob.BlobDataSource</code> .

- Create the following settings as new `moduleProperty` elements in the `sourceConfig` XML:

Option	Description
<code>databaseDirectory</code>	Specify the path to the sample database: <code><install path>\IAS\<version>\sample\ias-extensions\data\document-db</code> .

6. Delete the other `moduleProperty` elements within `sourceConfig`.

After these steps, the `sourceConfig` XML should be as follows:

```

...
    <sourceConfig>
      <moduleId>
        <id>com.endeca.ias.extension.sample.datasource.blob.BlobDataSource</id>
      </moduleId>
      <moduleProperties>
        <moduleProperty>
          <key>databaseDirectory</key>
          <value>C:\Oracle\Endeca\IAS\3.1.0\sample\ias-extensions\data\document-db<
/val
          </moduleProperty>
        </moduleProperties>
        <excludeFilters />
        <includeFilters />
      </sourceConfig>
    ...

```

7. Configure the following settings within the `outputConfig` XML:

Option	Description
<code>moduleId</code>	Specify the output type for a crawl. Specify an id of Record Store.
<code>host</code>	Specify the fully qualified name of the host running the Record Store instance. The default value is <code>localhost</code> .
<code>port</code>	Specify the port of the Endeca IAS Service running the Record Store instance. The default value is <code>8401</code> .
<code>contextPath</code>	<p>If you installed IAS into WebLogic Server, and you modified the default WebLogic context path, then specify the revised context path without including a forward slash. In WebLogic Server installations, the default value of <code>contextPath</code> is <code>ias-server</code>. If you installed IAS into Jetty, you can remove <code>contextPath</code> or specify an empty value.</p> <p>For example:</p> <pre> <moduleProperty> <key>contextPath</key> <value>ias-server</value> </moduleProperty> </pre>

8. Delete the other `moduleProperty` elements within `outputConfig`.

For example, after these steps, the `outputConfig` XML should be similar to the following:

```

...
    <outputConfig>
      <moduleId>
        <id>Record Store</id>
      </moduleId>
      <moduleProperties>
        <moduleProperty>
          <key>host</key>
          <value>mymachine.endeca.com</value>
        </moduleProperty>
        <moduleProperty>
          <key>port</key>
          <value>8401</value>
        </moduleProperty>
      </moduleProperties>
    </outputConfig>
...

```

9. Save and close the crawl configuration file.
10. Run the `createCrawls` task of `ias-cmd` to upload the crawl configuration file to IAS.

For example:

```

C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat createCrawls -f C:\Oracle\Endeca\IAS\
3.1.0\sample\crawlConfigFiles\sampleBLOB.xml
Created crawl sampleBLOB

```

11. Run the `startCrawl` task of `ias-cmd` to acquire data from the sample BLOB database.

For example:

```

C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat startCrawl -id sampleBLOB

```

12. After the crawl has completed, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`. In this sample, IAS Server created 5 Endeca records.

For example, this command returns the record count for the crawl:

```

C:\Oracle\Endeca\IAS\3.1.0\bin>recordstore-cmd.bat read-baseline -a sampleBLOB -c
Records read: 5

```

Running the sample Document Directory data source

The sample Document Directory data source illustrates an implementation of the `BinaryContentFileProvider` interface. This interface allows the extension to pass a file to IAS Server and perform text extraction. After you install the extensions into the Integrator Acquisition System, you can configure and then run the sample Document Directory data source.

To run the sample Document Directory data source:

1. In your IAS installation, locate the default crawl configuration files provided in `<install path>\IAS\<version>\sample\crawlConfigFiles`.
2. Make a copy of `fileSystemCrawl.xml`, rename the file with a unique name for your environment, and save it to a local directory.

For example, copy `fileSystemCrawl.xml` and save it as `sampleDocDir.xml` within `<install path>\IAS\<version>\sample\crawlConfigFiles`.
3. Open the new crawl configuration file in a text editor.

4. Configure the following settings:

Option	Description
crawlId	Specify a unique name to distinguish the crawl from others in IAS. For example, sampleDocDir.
moduleId	Specify the module ID for a Document Directory data source. This value must be set to com.endeca.ias.extension.sample.datasource.directory.DirectoryDataSource.

5. Create the following settings as new moduleProperty elements in the sourceConfig XML:

Option	Description
documentDirectory	Specify the path to the documents: <install path>\IAS\<version>\sample\ias-extensions\data\documents.

6. Delete the other moduleProperty elements within sourceConfig.

After these steps, the sourceConfig XML should be as follows:

```

...
    <sourceConfig>
      <moduleId>
        <id>com.endeca.ias.extension.sample.datasource.directory.DirectoryDataSource<
/id>
      </moduleId>
      <moduleProperties>
        <moduleProperty>
          <key>documentDirectory</key>
          <value>C:\Oracle\Endeca\IAS\3.1.0\sample\ias-extensions\data\documents<
/value>
        </moduleProperty>
      </moduleProperties>
      <excludeFilters />
      <includeFilters />
    </sourceConfig>
...

```

7. Configure the following settings within the outputConfig XML:

Option	Description
moduleId	Specify the output type for a crawl. Specify an id of Record Store.
host	Specify the fully qualified name of the host running the Record Store instance. The default value is localhost.

Option	Description
port	Specify the port of the Endeca IAS Service running the Record Store instance. The default value is 8401.
contextPath	<p>If you installed IAS into WebLogic Server, and you modified the default WebLogic context path, then specify the revised context path without including a forward slash. In WebLogic Server installations, the default value of contextPath is ias-server. If you installed IAS into Jetty, you can remove contextPath or specify an empty value.</p> <p>For example:</p> <pre data-bbox="868 772 1266 940"><moduleProperty> <key>contextPath</key> <value>ias-server</value> </moduleProperty></pre>

8. Delete the other moduleProperty elements within outputConfig.

For example, after these steps, the outputConfig XML should be similar to the following:

```
...
  <outputConfig>
    <moduleId>
      <id>Record Store</id>
    </moduleId>
    <moduleProperties>
      <moduleProperty>
        <key>host</key>
        <value>mymachine.endeca.com</value>
      </moduleProperty>
      <moduleProperty>
        <key>port</key>
        <value>8401</value>
      </moduleProperty>
    </moduleProperties>
  </outputConfig>
...
```

9. Save and close the crawl configuration file.
10. Run the createCrawls task of ias-cmd to upload the crawl configuration file to IAS.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat createCrawls -f C:\Oracle\Endeca\IAS\
3.1.0\sample\crawlConfigFiles\sampleDocDir.xml
Created crawl sampleDocDir
```

11. Run the startCrawl task of ias-cmd to acquire data from the sample documents.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat startCrawl -id sampleDocDir
```

- After the crawl has completed, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`. In this sample, IAS Server created 5 Endeca records.

For example, this command returns the record count for the crawl:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>recordstore-cmd.bat read-baseline -a sampleDocDir -c
Records read: 5
```

Running the sample Change Tracking data source

The sample Change Tracking data source illustrates an implementation of the `IncrementalDataSourceRuntime` interface. This interface provides support to check whether a full acquisition is required from the Change Tracking data source. If a full acquisition is not required, then the data source provides an implementation of `runIncrementalAcquisition()` to acquire only the changed records.

After you install the extensions into the Integrator Acquisition System, you can configure and then run the sample Change Tracking data source.

To run the sample Change Tracking data source:

- In your IAS installation, locate the default crawl configuration files provided in `<install path>\IAS\<version>\sample\crawlConfigFiles`.
- Make a copy of `fileSystemCrawl.xml`, rename the file with a unique name for your environment, and save it to a local directory.

For example, copy `fileSystemCrawl.xml` and save it as `sampleChgTracking.xml` within `<install path>\IAS\<version>\sample\crawlConfigFiles`.

- Open the new crawl configuration file in a text editor.
- Configure the following settings:

Option	Description
<code>crawlId</code>	Specify a unique name to distinguish the crawl from others in IAS. For example, <code>sampleChgTracking</code> .
<code>moduleId</code>	Specify the module ID for a Change Tracking data source. This value must be set to <code>com.endeca.ias.extension.sample.data source.incremental.ChangeTrackingData source</code> .

5. Create the following settings as new `moduleProperty` elements in the `sourceConfig` XML:

Option	Description
path	Specify the path to the documents: <code><install path>\IAS\<version>\sample\ias-extensions\data\change-tracking-db.xml</code> .

6. Delete the other `moduleProperty` elements within `sourceConfig`.

After these steps, the `sourceConfig` XML should be as follows:

```

...
    <sourceConfig>
      <moduleId>

<id>com.endeca.ias.extension.sample.datasource.incremental.ChangeTrackingDataSource</id>
      </moduleId>
      <moduleProperties>
        <moduleProperty>
          <key>path</key>

<value>C:\Oracle\Endeca\IAS\3.1.0\sample\ias-extensions\data\change-tracking-db.xml</value>
          </moduleProperty>
        </moduleProperties>
        <excludeFilters />
        <includeFilters />
      </sourceConfig>
...

```

7. In the `textExtractionConfig` XML, change the value of `enabled` to `false`.
8. Configure the following settings within the `outputConfig` XML:

Option	Description
moduleId	Specify the output type for a crawl. Specify an id of Record Store.
host	Specify the fully qualified name of the host running the Record Store instance. The default value is <code>localhost</code> .
port	Specify the port of the Endeca IAS Service running the Record Store instance. The default value is <code>8401</code> .

Option	Description
contextPath	<p>If you installed IAS into WebLogic Server, and you modified the default WebLogic context path, then specify the revised context path without including a forward slash. In WebLogic Server installations, the default value of contextPath is <code>ias-server</code>. If you installed IAS into Jetty, you can remove contextPath or specify an empty value.</p> <p>For example:</p> <pre><moduleProperty> <key>contextPath</key> <value>ias-server</value> </moduleProperty></pre>

9. Delete the other moduleProperty elements within outputConfig.

For example, after these steps, the outputConfig XML should be similar to the following:

```
...
<outputConfig>
  <moduleId>
    <id>Record Store</id>
  </moduleId>
  <moduleProperties>
    <moduleProperty>
      <key>host</key>
      <value>mymachine.endeca.com</value>
    </moduleProperty>
    <moduleProperty>
      <key>port</key>
      <value>8401</value>
    </moduleProperty>
  </moduleProperties>
</outputConfig>
...
```

10. Save and close the crawl configuration file.
11. Run the createCrawls task of ias-cmd to upload the crawl configuration file to IAS.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat createCrawls -f C:\Oracle\Endeca\IAS\
3.1.0\sample\crawlConfigFiles\sampleChgTracking.xml
Created crawl sampleChgTracking
```

12. Run the startCrawl task of ias-cmd to acquire data from the sample documents.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat startCrawl -id sampleChgTracking
```

13. After the crawl has completed, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`. In this sample, IAS Server created 3 Endeca records.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>recordstore-cmd.bat read-baseline -a sampleChgTracking
[Endeca.Id=1, Endeca.Action=UPSERT, Endeca.SourceId=sampleChgTracking, DATA=base
line data...]
[Endeca.Id=3, Endeca.Action=UPSERT, Endeca.SourceId=sampleChgTracking, DATA=some
incremental data...]
[Endeca.Id=5, Endeca.Action=UPSERT, Endeca.SourceId=sampleChgTracking, DATA=some
incremental data...]
```

14. Navigate to `<install path>\IAS\<version>\sample\ias-extensions\data` and open `change-tracking-db.xml` in a text editor.
15. Update one record in the `change-tracking-db.xml` file by doing the following:
 - (a) Add a new `<changeHistory>` entry to the file as shown in the example below.
 - (b) Ensure that the `<key>` value corresponds to an existing `<row>` entry in the `<database>`.
 - (c) Modify the `<time>` value to indicate a time after the acquisition in step 12 and before the current time. (The time is expressed in UTC format. See <http://www.w3.org/TR/NOTE-datetime> for guidance about the syntax.)

For example:

```
<changeHistory>
  <key>5</key>
  <changeType>UPDATE</changeType>
  <time>2010-02-02T19:19:43.471-05:00</time>
</changeHistory>
```

Acquiring data from this file results in an incremental update to record 5.

16. Add one record in the `change-tracking-db.xml` file by doing the following:
 - (a) Add a `<row>` entry to the file and ensure the `<key>` value is unique, as shown:

```
<row>
  <key>7</key>
  <data>some incremental data...</data>
</row>
```

- (b) Add a `<changeHistory>` entry for the `<row>` as shown:

```
<changeHistory>
  <key>7</key>
  <changeType>CREATE</changeType>
  <time>2010-02-02T19:19:43.471-05:00</time>
</changeHistory>
```

- (c) Ensure that the `<key>` value corresponds to a `<row>` entry in the `<database>`.
 - (d) Ensure that the `<changeType>` value is set to `CREATE`.
 - (e) Modify the `<time>` value to indicate a time after the acquisition in step 12 and before the current time. (The time is expressed in UTC format. See <http://www.w3.org/TR/NOTE-datetime> for guidance about the syntax.)

Acquiring data from this file results in an incremental change that adds record 7.

17. Delete one record in the `change-tracking-db.xml` file by doing the following:
 - (a) Add a `<changeHistory>` entry for the `<row>` that has been removed as shown:

```
<changeHistory>
  <key>8</key>
  <changeType>DELETE</changeType>
  <time>2010-02-02T19:19:43.471-05:00</time>
</changeHistory>
```

- (b) Ensure that the `<key>` value corresponds to a `<row>` that does not exist in the `<database>`.
- (c) Ensure that the `<changeType>` value is set to `DELETE`.
- (d) Modify the `<time>` value to indicate a time after the acquisition in step 12 and before the current time. (The time is expressed in UTC format. See <http://www.w3.org/TR/NOTE-datetime> for guidance about the syntax.)

Acquiring data from this file results in an incremental change that removes record 8.

18. Save and close `change-tracking-db.xml`.
19. Run the `startCrawl` task of `ias-cmd` to acquire data from the revised sample documents.

For example:

```
C:\Oracle\Endeca\IAS\3.1.0\bin>ias-cmd.bat startCrawl -id sampleChgTracking
```

20. After the crawl has completed, you can confirm that IAS Server updated, added, and deleted the records you modified by running the `read-baseline` task of `recordstore-cmd`.