Oracle® Endeca Information Discovery Integrator

Integrator Acquisition System API Guide

Version 3.2.0 • January 2016

ORACLE

Copyright and disclaimer

Table of Contents

Preface

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

The Integrator 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 programmatically configure and run IAS crawls using the IAS Server API, the Component Instance Manager API, and the Record Store API.

The guide assumes that you are familiar with the concepts of the Integrator Acquisition System, including how file systems, delimited files, JDBC databases, and custom data sources are crawled by IAS.

Who should use this guide

This guide is intended for data developers who are using the Integrator Acquisition System APIs to crawl source data and incorporate that data into an Endeca data domain.

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 APIs

This section introduces each API in the Integrator Acquisition System.

The IAS APIs

The Integrator Acquisition System includes the following APIs:

- IAS Server API — A WSDL-based API that controls crawling operations against a variety of file systems, delimited files, JDBC databases, and custom data sources.
- Component Instance Manager API — A WSDL-based API that creates, lists, and deletes Record Store instances.
- Record Store API — A WSDL-based API that modifies and controls a variety of reading, writing, and utility operations against Record Store instances.
- IAS Extension API — A Java-based API to build extensions to the Integrator Acquisition System such as data sources and manipulators. This API is for plugin developers and it is documented in the *Integrator Acquisition System Extension API Guide*.

The rest of this guide documents the WSDL-based APIs. Each WSDL-based API in the Integrator Acquisition System can be used with any programming language that has Web services support, and developers can write crawl functions in their preferred language (Java, .NET, etc.) as a Web service.

Name and location of the WSDL files

You can find the following WSDL files in `<install path>\IAS\<version>\doc\wsdl` :

- IAS Server API — `IasCrawlerService.wsdl`.
- Component Instance Manager API — `ComponentInstanceManager.wsdl`.
- Record Store API — `RecordStore.wsdl`.

Java convenience classes

For convenience, Java versions of each API are included in `<install path>\IAS\<version>\lib` :

- IAS Server API — `ias-api\eidi-api-3.2.0.jar`.
- Component Instance Manager API — `component-manager-api\component-manager-api-3.2.0.jar`.
- Record Store API — `recordstore-api\recordstore-api-3.2.0.jar`.

Each API also includes utility (helper) classes in its JAR file.

If desired, you can use the Java version of the API rather than generate client stubs from the WSDL files. The Java versions were generated using Apache CXF. For other languages (such as .NET), you must generate the client stubs in your programming language.

Java examples in the guide

Examples in this guide use the Java versions of the APIs mentioned above. This convention has an important implication in the code examples:

Most types of identifiers are set in the constructor rather than in a setter method. For example:

```
ModuleId moduleId = new ModuleId("File System");
```

If you are generating client stubs, most types of identifiers are set using a setter method. For example:

```
ModuleId moduleId = new ModuleId();
moduleId.setId("File System");
```

The specific setter usage depends on the application you use to generate client stubs. For example, setter usage varies in stubs generate with Apache Axis and Apache CXF.

Reference documentation (Javadoc) for the IAS APIs

The Javadoc provides reference documentation for both the core and utility classes. You can find the Javadoc in `<install path>\IAS\<version>\doc`:

- *IAS Server API Reference* — `ias-server-javadoc`
- *Component Instance Manager API Reference* — `component-manager-javadoc`
- *Record Store API Reference* — `recordstore-javadoc`

Generating client stubs for the IAS Web services

To create a client application that consumes any of the IAS Web services, you need the particular Web service's WSDL file to generate client stubs.

A WSDL file specifies value types, exceptions, and available methods in a Web service in a programmatic fashion. Typically, a client developer uses a tool that parses the WSDL file and generates client-side stubs (also called proxy classes) and value types. These generated files include all the code necessary to serialize and deserialize SOAP messages and make the SOAP layer transparent to the client developer. The IAS WSDL files can be used with any language that has Web services support.

Among the tools that generate client stub code from the WSDLs are the following:

- Apache CXF 2.2 or later
- Java Web Services Developer Pack (Java WSDP), version 1.4 or later
- Web Services Description Language Tool (wsdl.exe), available as part of the Microsoft .NET Framework SDK

Specify the appropriate choice below as the package name when you generate stubs for a particular Web service:

- `com.endeca.eidi.ias.api`

- `com.endeca.eidi.component.manager`
- `com.endeca.eidi.recordstore`

For example, the CXF `wsdl2java` utility takes the WSDL file and generates fully annotated Java code with one of the following commands:

- `wsdl2java -p com.endeca.eidi.ias.api -client IasCrawlerService.wsdl`
- `wsdl2java -p com.endeca.eidi.component.manager -client ComponentInstanceManager.wsdl`
- `wsdl2java -p com.endeca.eidi.recordstore -client RecordStore.wsdl`

For details on using a WSDL code-generation utility, refer to the utility's documentation.

Keep in mind that the exact syntax of a class member depends on the output of the WSDL tool that you are using. Therefore, check the client stub classes that are generated by your WSDL tool for the exact syntax of the class members.

Chapter 2

IAS Server API

This section describes the IAS Server API.

IAS Server core operations

This topic describes the IAS Server API core methods.

The following methods are provided by the API:

- `createCrawl` creates and stores a new crawl.
- `startCrawl` starts a crawl.
- `listCrawls` lists all the crawls that have been created.
- `stopCrawl` stops a crawl that is currently running.
- `deleteCrawl` deletes an existing crawl.
- `getStatus` returns the status of a specified crawl.
- `getMetrics` retrieves crawl statistics for a specified crawl.
- `getCrawlConfig` gets the configuration settings of a specified crawl.
- `listModules` returns a list of the available module IDs for data sources or manipulators. Module IDs may include any custom data source extensions or custom manipulator extensions that you installed using the IAS Extension API.

- `updateCrawl` updates the configuration settings for an existing crawl.
- `getServerInfo` returns a list of the IAS Server properties.

Note: The syntax descriptions for these operations use Java conventions. The examples in this guide use client stubs generated with Apache CXF 2.2. However, the exact syntax of a class member depends on the output of the WSDL tool that you are using.

Connecting to the IAS Server

Call the `IasCrawlerLocator.create()` method to connect to the IAS Server.

The `IasCrawlerLocator` class establishes a connection with the IAS Server. In particular, the `IasCrawlerLocator.getService()` method is the call that makes the connection. The `ServiceAddress` stores connection information including the host, and port, and context path of the IAS Server.

To create a connection to the IAS Server:

1. Create a `ServiceAddress` object and specify the host and port of the server running the IAS Server and also specify the `contextPath` of WebLogic. If you are installing into Jetty, not WebLogic, the `contextPath` can be set to an empty string.
2. Create an `IasCrawlerLocator` by calling `create()` and specifying the `ServiceAddress` object. For example:

   ```
   ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
   IasCrawlerLocator locator = IasCrawlerLocator.create(address);
   ```

3. Create an `IasCrawler` object and call `getService()` to establish a connection to the server and the Endeca IAS Service. For example:

   ```
   IasCrawler crawler = locator.getService();
   ```

You now have a connection to the IAS Server that can perform crawling operations.

Creating crawls

Use the `IasCrawler.createCrawl()` method to create a new crawl of any type (for example, file system, delimited file, or custom data source).

The syntax of the method is:

```
IasCrawler.createCrawl(CrawlConfig crawlConfig)
```

The `crawlConfig` parameter is a `CrawlConfig` object that has the configuration settings of the crawl.

To create a new crawl:

1. Make sure that you have created a connection to the IAS Server.
2. Instantiate a `CrawlId` object and set the Id for the crawl in the constructor.

   You can create an ID with alphanumeric characters, underscores, dashes, and periods. All other characters are invalid for an ID.

   For example:

   ```
   // Create a new crawl ID with the name set to Demo.
   ```

```
CrawlId crawlId = new CrawlId("Demo");
```

3. Instantiate a `CrawlConfig` object and pass in the `CrawlId` object .

   For example:

   ```
   // Create a crawl configuration.
   CrawlConfig crawlConfig = new CrawlConfig(crawlId);
   ```

4. Instantiate a `SourceConfig` object

   For example:

   ```
   // Create source configuration.
   SourceConfig sourceConfig = new SourceConfig();
   ```

5. Set the source properties and seeds in the `SourceConfig` object. Detailed information on source properties is provided in other topics.
6. Set the `SourceConfig` on the `CrawlConfig`.

   For example:

   ```
   // Set source configuration.
   crawlConfig.setSourceConfig(sourceConfig);
   ```

7. Optionally, you can set configuration options for such features as document conversion, logging, and filters for files and directories. Detailed information on these options is provided in other topics.
8. Create the crawl by calling `IasCrawler.createCrawl()` and passing the `CrawlConfig` (the configuration) object:

   For example:

   ```
   crawler.createCrawl(crawlConfig);
   ```

If the `IasCrawler.createCrawl()` method fails, it throws an exception:

- `CrawlAlreadyExistsException` occurs if a crawl of the same name already exists.
- `InvalidCrawlConfigException` occurs if the configuration is invalid. You can call `getCrawlValidationFailures()` to return the list of crawl validation errors.

To catch these exceptions, use a `try` block when you issue the method.

If the new crawl is successfully created, it can be started with the `IasCrawler.startCrawl()` method.

About the source properties for crawls

The `SourceConfig` class allows a client to specify information about the data source that is being crawled. The `SourceConfig` class uses two methods to set data source properties: `setModuleId()` and `setModuleProperties()`.

Module ID

The `setModuleId()` method sets the module ID of the data source for this crawl. A module ID is a `ModuleId` object.

The string `File System` is the module ID for a file system crawl (whose source is a file system). You must specify this module ID when you create a file system crawl.

Each crawl type has its own unique module ID. Use the `IasCrawler.listModules()` method to find out the module IDs that are available to the IAS Server.

A plug-in developer specifies the `ModuleId` for a custom data source. An IAS data developer can determine the `ModuleId` for a custom data source by running the `listModules` and task in the IAS Server Command-line Utility.

Module Properties

Each `ModuleProperty` is a key/value pair or a key/multi-value pair that provides configuration information about this data source. You specify a `ModuleProperty` by calling `setKey()` to specify a string representing the key and by calling `setValues()` to set one or more corresponding values.

You then set each `ModuleProperty` on the `SourceConfig` object by calling `addModuleProperty()`.

File system source properties and example

The `SourceConfig` object for a file system crawl requires a `ModuleId` that specifies `"File System"`, a `ModuleProperty` to specify the seeds, and additional `ModuleProperty` objects for any optional source properties.

Table 2.1: Module Properties for file system data sources

File System Module Property Key Name	Key Value Description
`seeds`	The `seeds` property is a key/multi-value pair. The key is `seeds` and the multi-value pair is one or more strings to a file or folder. File paths used as seeds must be absolute paths. Required.
`gatherNativeFileProperties`	The `gatherNativeFileProperties` property (if set to `true`) enables the crawl to gather operating system-level properties, such as Windows ACL properties (e.g., `Endeca.FileSystem.ACL.AllowRead`) or UNIX owner, group, and readable properties (e.g., `Endeca.FileSystem.IsOwnerReadable`). The default is `false`. Optional.
`expandArchives`	The `expandArchives` property (if set to `true`) enables the crawl to expand archived entries. Enabling this property creates an Endeca record for each archived entry and populates its properties. Enabling the document conversion option extracts text. Note that the crawl does not gather native file properties for archived entries even if that option is enabled. The default is `false`. Optional.

Here is an example of the source properties for a file system crawl.

```
// Connect to the IAS Server.
ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
IasCrawlerLocator locator = IasCrawlerLocator.create(address);
```

```
IasCrawler crawler = locator.getService();

// Create a new crawl Id with the name set to Demo.
CrawlId crawlId = new CrawlId("Demo");

// Create the crawl configuration.
CrawlConfig crawlConfig = new CrawlConfig(crawlId);

// Create the source configuration.
SourceConfig sourceConfig = new SourceConfig();

// Create a file system module ID.
ModuleId moduleId = new ModuleId("File System");

// Set the module ID in the source config.
sourceConfig.setModuleId(moduleId);

// Create a module property object for the seeds.
ModuleProperty seeds = new ModuleProperty();
// Set the key for seeds.
seeds.setKey("seeds");
// Set multiple values for seeds.
seeds.setValues("C:\\tmp\\iasdocset","C:\\tmp\\etldocset");

// Set the seeds module property on the source config.
sourceConfig.addModuleProperty(seeds);

// Create a module property for gathering native file props.
ModuleProperty nativeFileProps = new ModuleProperty();
// Set the key for gathering native file properties.
nativeFileProps.setKey("gatherNativeFileProperties");
// Set the value to enable gathering native file properties.
nativeFileProps.setValues("true");

// Set the nativeFileProps module property on the source config.
sourceConfig.addModuleProperty(nativeFileProps);

// Create a module property object for expanding archives.
ModuleProperty extractArchives = new ModuleProperty();
// Set the key for extracting archive files.
extractArchives.setKey("expandArchives");
// Set the value to enable expanding archives.
extractArchives.setValues("true");

// Set the extractArchives module property on the source config.
sourceConfig.addModuleProperty(extractArchives);

// Set the source configuration in the crawl configuration.
crawlConfig.setSourceConfig(SourceConfig);

// Create the crawl.
crawler.createCrawl(crawlConfig);
```

Note that if you retrieve a `SourceConfig` object from a configured crawl, you can call the `getModuleId()` method to get the module ID and the `getModuleProperties()` method to retrieve the list of module properties.

Source properties for a custom data source

The `SourceConfig` for a custom data source crawl contains a mandatory `ModuleId` and `ModuleProperty` objects that define the custom data source to crawl and any other optional properties that are necessary for a custom data source.

Module ID for a custom data source

A plug-in developer specifies the `ModuleId` for a custom data source. An IAS data developer can determine the `ModuleId` for a custom data source by running the `listModules` and task in the IAS Server Command-line Utility:

1. Start a command prompt and navigate to `<install path>\IAS\<version>\bin`.
2. Type `ias-cmd` and specify the `listModules` task with the module type (`-t`) option and specify and argument of `SOURCE`. For example:

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

...
```

3. In the list of data sources returned by `listModules`, locate the custom data source and Id value.

Module Properties for a custom data source

Custom data sources can use any number of module properties. A plugin developer determines what module properties are necessary for a custom data source and whether the module properties are required or optional.

An IAS data developer can check the available module properties for a custom data source by running the `getModuleSpec` task of the IAS Server Command-line Utility:

1. Start a command prompt and navigate to `<install path>\IAS\<version>\bin`.
2. Type `ias-cmd` and specify the `getModuleSpec` task with the ID of the module whose source properties you want to see. For example:

```
ias-cmd.bat getModuleSpec -id "Sample Data Source"
 Sample Data Source
 =================
 [Module Information]
 *Id: Sample Data Source
 *Type: SOURCE
 *Description: Sample Data Source for Testing

 [Sample Data Source Configuration Properties]
 Group: Basic Settings
 --------------------
 User name:
  *Name: username
  *Type: {http://www.w3.org/2001/XMLSchema}string
  *Required: true
  *Max Length: 256
  *Description: The name of the user used to log on to the repository
  *Multiple Values: false
  *Multiple Lines: false
  *Password: false
  *Always Show: true
```

```
 Password:
  *Name: password
  *Type: {http://www.w3.org/2001/XMLSchema}string
  *Required: true
  *Max Length: 256
  *Description: The password used to log on to the repository
  *Multiple Values: false
  *Multiple Lines: false
  *Password: true
  *Always Show: true

...
```

Here is an example of the source properties for a custom data source crawl.

```
// Connect to the IAS Server.
ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
IasCrawlerLocator locator = IasCrawlerLocator.create(address);
IasCrawler crawler = locator.getService();

// Create a new crawl Id with the name set to Demo.
CrawlId crawlId = new CrawlId("Demo");

// Create the crawl configuration.
CrawlConfig crawlConfig = new CrawlConfig(crawlId);

// Create the source configuration.
SourceConfig sourceConfig = new SourceConfig();

// Create a module ID for a Sample Data Source repository.
// Set the module ID in the constructor.
ModuleId moduleId = new ModuleId("Sample Data Source");

// Create a list for the module property objects.
List<ModuleProperty> cmsPropsList = new ArrayList<ModuleProperty>();

// Create a module property for username.
// Set key/values of the module property as strings in the constructor.
ModuleProperty uname = new ModuleProperty("username", "SALES\\username");

// Set the module property in the module property list.
cmsPropsList.add(uname);

// Create a module property for password.
// Set key/values of the module property as strings in the constructor.
ModuleProperty upass = new ModuleProperty("password", "endeca");

// Set the module property in the module property list.
cmsPropsList.add(upass);

// Set the module property list in the source configuration.
sourceConfig.setModuleProperties(cmsPropsList);

// Set the source configuration in the crawl configuration.
crawlConfig.setSourceConfig(SourceConfig);

// Create the crawl.
crawler.createCrawl(crawlConfig);
```

Source properties for a manipulator

The `ManipulatorConfig` for a manipulator contains a mandatory `ModuleId` and `ModuleProperty` objects that define the manipulator to run and any other optional properties that are necessary for a manipulator.

Module ID for a manipulator

A plugin developer specifies the `ModuleId` for a manipulator. An IAS data developer can determine the `ModuleId` for a manipulator by running the `listModules` and task in the IAS Server Command-line Utility:

1. Start a command prompt and navigate to `<install path>\IAS\<version>\bin`.
2. Type `ias-cmd` and specify the `listModules` task with the module type (`-t`) option and specify and argument of `MANIPULATOR`. For example:

```
ias-cmd listModules -t MANIPULATOR
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
```

3. In the list of manipulators returned by `listModules`, locate the manipulator and its ID value. That becomes the `ModuleId`.

Module Properties for a manipulator

Manipulators can use any number of module properties. A plugin developer determines what module properties are necessary for a manipulator and whether the module properties are required or optional.

An IAS data developer can check the available module properties for a manipulator by running the `getModuleSpec` task of the IAS Server Command-line Utility:

1. Start a command prompt and navigate to `<install path>\IAS\<version>\bin`.
2. Type `ias-cmd` and specify the `getModuleSpec` task with the ID of the module whose source properties you want to see. For example:

```
ias-cmd getModuleSpec -id
com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulator
Substring Manipulator
====================
[Module Information]
 *Id: com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulator

 *Type: MANIPULATOR
 *Description: Generates a new property that is a substring of another property
value

[Substring Manipulator Configuration Properties]
Group:
-------
Source Property:
 *Name: sourceProperty
 *Type: {http://www.w3.org/2001/XMLSchema}string
 *Required: true
 *Default Value:
 *Max Length: 255
 *Description:
 *Multiple Values: false
 *Multiple Lines: false
 *Password: false
```

```
 *Always Show: false

Target Property:
 *Name: targetProperty
 *Type: {http://www.w3.org/2001/XMLSchema}string
 *Required: true
 *Default Value:
 *Max Length: 255
 *Description:
 *Multiple Values: false
 *Multiple Lines: false
 *Password: false
 *Always Show: false

Substring Length:
 *Name: length
 *Type: {http://www.w3.org/2001/XMLSchema}integer
 *Required: true
 *Default Value: 2147483647
 *Min Value: -2147483648
 *Max Value: 2147483647
 *Description: Substring length
 *Multiple Values: false
 *Multiple Lines: false
 *Password: false
 *Always Show: false

Substring Start Index:
 *Name: startIndex
 *Type: {http://www.w3.org/2001/XMLSchema}integer
 *Required: false
 *Default Value: 0
 *Min Value: -2147483648
 *Max Value: 2147483647
 *Description: Substring start index (zero based)
 *Multiple Values: false
 *Multiple Lines: false
 *Password: false
 *Always Show: false
```

Here is an example of the source properties for a crawl that includes the manipulator in the above example.

```
// Connect to the IAS Server.
ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
IasCrawlerLocator locator = IasCrawlerLocator.create(address);
IasCrawler crawler = locator.getService();

// Create a new crawl Id with the name set to Demo.
CrawlId crawlId = new CrawlId("Demo");

// Create the crawl configuration.
CrawlConfig crawlConfig = new CrawlConfig(crawlId);

// Create a list for manipulator configurations, even if
// there is only one.
List<ManipulatorConfig> manipulatorList = new ArrayList<ManipulatorConfig>();

// Create a module ID for a Substring Manipulator.
// Set the module ID in the constructor.
ModuleId moduleId
= new ModuleId("com.endeca.ias.extension.sample.manipulator.substring.SubstringManipulator");

// Create a manipulator configuration.
ManipulatorConfig manipulator = new ManipulatorConfig(moduleId);

// Create a list for the module property objects.
List<ModuleProperty> manipulatorPropsList = new ArrayList<ModuleProperty>();
```

```
// Create a module property for sourceProperty.
// Set key/values of the module property as strings in the constructor.
ModuleProperty sp = new ModuleProperty("sourceProperty", "Endeca.Document.Text");

// Set the module property in the module property list.
manipulatorPropsList.add(sp);

// Create a module property for targetProperty.
// Set key/values of the module property as strings in the constructor.
ModuleProperty tp = new ModuleProperty("targetProperty", "Truncated.Text");

// Set the module property in the module property list.
manipulatorPropsList.add(tp);

// Create a module property for length.
// Set key/values of the module property as strings in the constructor.
ModuleProperty length = new ModuleProperty("length", "20");

// Set the module property in the module property list.
manipulatorPropsList.add(length);

// Set the module property list in the manipulator configuration.
manipulator.setModuleProperties(manipulatorPropsList);
manipulatorList.add(manipulator);

// Set the list of manipulator configurations in the crawl configuration.
crawlConfig.setManipulatorConfigs(manipulatorList);

// Create the crawl.
crawler.createCrawl(crawlConfig);
```

Setting text extraction options

The `TextExtractionConfig` class specifies document conversion parameters to override default values.

Note: The phrases *text extraction* and *document conversion* mean the same thing.

The `TextExtractionConfig` class has methods to set these document conversion options:

- Whether document conversion should be performed. The default for file system crawls is `true`. The default for custom data source extensions defaults to `false` unless the extension developer implements an interface that supports binary content. If set to `true`, the next options can be used.
- Whether to use local file copies to perform the text extraction (file system crawls only).
- The time that IAS Server waits for text extraction results from the IAS Document Conversion Module before retrying.

To set the text-extraction options:

1. Make sure that you have already created a `SourceConfig`, a `CrawlConfig`, and set the name and the seeds (if required for the source type) for the crawl.
2. Instantiate an empty `TextExtractionConfig` object

   For example:

   ```
   TextExtractionConfig textOptions = new TextExtractionConfig();
   ```

3. Call the `setEnabled()` method to set a Boolean value to enable text extraction:

   ```
   // Enable text extraction for this crawl.
   ```

```
textOptions.setEnabled(true);
```

4. For file system crawls, you can use the `setMakeLocalCopy()` method to set a Boolean indicating whether files should be copied to a local temporary directory before text is extracted from them. The default for `setMakeLocalCopy()` is `false`. Custom data source extensions may also make local copies if the extension developer implemented the `BinaryContentFileProvider` interface of the IAS Extension API.

   ```
   // Enable use of local file copying.
   textOptions.setMakeLocalCopy(true);
   ```

5. If desired, call the `setTimeout()` method and specify an integer to set amount of time (in seconds) IAS waits for text extraction on a document to finish before attempting again. The default is 90 seconds.

   ```
   // Set timeout to 120 seconds.
   textOptions.setTimeout(120);
   ```

6. Call the `CrawlConfig.setTextExtractionConfig()` method to set the populated `TextExtractionConfig` object in the `CrawlConfig` object:

   ```
   // Set the text extraction options in the configuration
   crawlConfig.setTextExtractionConfig(textOptions);
   ```

7. Create the file system crawl:

   ```
   crawler.createCrawl(crawlConfig);
   ```

Note that if you retrieve a `TextExtractionConfig` object from a configured crawl, each of the set methods has a corresponding get method, such as the `getTimeout()` method.

Filtering files and folders

The API provides classes to specify inclusion and exclusion filters for files and folders.

You add include and exclude filters to the crawl configuration to ensure that the IAS Server processes the proper files and folders when running a crawl.

Note: Custom data sources built using the IAS Extension API do not support filters.

Keep in mind that if you use both include and exclude filters, the exclude filters take precedence. For additional detailed information about how filters interact with each other and Endeca properties, see the "About filters" topic in the *Integrator Acquisition System Developer's Guide*.

The filter classes are the following:

- `WildcardFilter` for filtering based on a wildcard value.
- `RegexFilter` for filtering based on a regular expression value.
- `DateFilter` for filtering based on a datetime value.
- `LongFilter` for filtering based on a long value.

For all filters, you must specify a property against which the filter is applied. The property is typically a standard property generated by IAS (such as the `Endeca.FileSystem.Name` property), but it can also be a custom property.

Some of the classes used for creating filters are the following:

- `ComparisonOperator` provides comparison operators, such as EQUAL, NOT_EQUAL, LESS, and GREATER.
- `Filter` is the base type for all filters, providing for an optional filter scope property.
- `FilterScope` provides enumerations for the FILE and DIRECTORY filter scopes.

After you create a filter, you must set it in a `SourceConfig` object, which in turn is set in the `CrawlConfig` configuration object.

Creating wildcard filters

The `WildcardFilter` class specifies a wildcard as an inclusion or exclusion filter.

A `WildcardFilter` is a filter that applies a wildcard to a particular property. The wildcard matcher uses the question-mark (?) character to represent a single wildcard character and the asterisk (*) to represent multiple wildcard characters. Matching is case insensitive: this is not configurable (If case sensitivity is required, consider using a regular expression). In the example below, the filter applies to the `Endeca.FileSystem.Name` property.

To create a wildcard filter:

1. Make sure that you have created a `SourceConfig` and a `CrawlConfig`.
2. Instantiate a new, empty `WildcardFilter` object:

   ```
   WildcardFilter filter = new WildcardFilter();
   ```

3. Call the `setPropertyName()` method (inherited from the `Filter` class) to set the name of the property against which the filter is applied:

   ```
   // filter on the file name
   filter.setPropertyName("Endeca.FileSystem.Name");
   ```

4. Use the `setWildcard()` method to set the wildcard:

   ```
   // exclude Word files
   filter.setWildcard("*.doc");
   ```

5. Use the `setScope()` method (inherited from the `Filter` class) to set the filter scope. You can set the scope to files (as in the following example), or to folders (`FilterScope.DIRECTORY`).

   ```
   // set the scope of the filter for only files
   filter.setScope(FilterScope.FILE);
   ```

6. Create a list of `Filter` objects and use the `add()` method (inherited from the `List` interface) to add the wildcard filter.

   ```
   List<Filter> filterList = new ArrayList<Filter>();
   filterList.add(filter);
   ```

7. Use the `SourceConfig.setExcludeFilters()` method to set the populated list in the `SourceConfig` configuration object. If this were an inclusion filter, you would use the `SourceConfig.setIncludeFilters()` method instead.

   ```
   // Set the filter in the source configuration.
   sourceConfig.setExcludeFilters(filterList);
   ```

8. Use the `CrawlConfig.setSourceConfig()` method to set the populated `SourceConfig` in the main `CrawlConfig` configuration object.

```
// Set the source config in the crawl configuration.
crawlConfig.setSourceConfig(sourceConfig);
```

Note that the `WildcardFilter` class has a `getWildcard()` method to retrieve a wildcard value. In addition, the `SourceConfig` class has the `getExcludeFilters()` and `getIncludeFilters()` methods to retrieve the filters from the source configuration.

Creating regular expression filters

The `RegexFilter` class specifies a regular expression as an inclusion or exclusion filter.

A `RegexFilter` is a filter that applies a regular expression to a particular record property. Matching is case sensitive by default (this is not configurable through the API). In the example below, the filter applies to the `Endeca.FileSystem.Name` property.

IAS implements Sun's `java.util.regex` package to parse and match the pattern of the regular expression. Therefore, the supported regular-expression constructs are the same as those in the documentation page for the `java.util.regex.Pattern` class:

```
http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html
```

This means that you can use any of the following constructs:

- Escape characters, such **\t** for the tab character.
- Character classes (simple, negation, range, intersection, subtraction). For example, **[^abc]** means match any character except a, b, or c, while **[a-zA-Z]** means match any upper- or lower-case letter.
- Predefined character classes, such as **\d** for a digit or **\s** for a whitespace character.
- POSIX character classes (US-ASCII only), such as **\p{Alpha}** for an alphabetic character, **\p{Alnum}** for an alphanumeric character, and **\p{Punct}** for punctuation.
- Boundary matchers, such as **^** for the beginning of a line, **$** for the end of a line, and **\b** for a word boundary.
- Logical operators, such as **X|Y** for either X or Y.

For a full list of valid constructs, see the Pattern class documentation page referenced above.

To create a regex filter:

1. Make sure that you have created a `SourceConfig` (see the following example) and a `CrawlConfig`.

```
SourceConfig sourceConfig = new SourceConfig();
```

2. Instantiate a new, empty `RegexFilter` object:

```
RegexFilter filter = new RegexFilter();
```

3. Use the `setPropertyName()` method (inherited from the `Filter` class) to set the name of the property against which the filter will be applied:

   For example:

```
// Filter on the file name.
filter.setPropertyName("Endeca.FileSystem.Name");
```

4. Call the `setRegex()` method to set the regular expression:

   For example:

   ```
   // Exclude executable and help files.
   filter.setRegex(".*\.(exe|bin|dll|hlp)$");
   ```

5. Use the `setScope()` method (inherited from the `Filter` class) to set the filter scope. You can set the scope to files (as in the following example), or to directories (`FilterScope.DIRECTORY`).

   For example:

   ```
   // Set the scope of the filter for only files.
   filter.setScope(FilterScope.FILE);
   ```

6. Create a list of `Filter` objects and add the regex filter to it.

   For example:

   ```
   List<Filter> filterList = new ArrayList<Filter>();
   filterList.add(filter);
   ```

7. Use the `SourceConfig.setExcludeFilters()` method to set the populated list in the `SourceConfig` configuration object. If this were an inclusion filter, you would use the `SourceConfig.setIncludeFilters()` method instead.

   For example:

   ```
   // Set the filter in the source configuration.
   sourceConfig.setExcludeFilters(filterList);
   ```

8. Use the `CrawlConfig.setSourceConfig()` method to set the populated `SourceConfig` in the main `CrawlConfig` configuration object.

   ```
   // Set the source config in the crawl configuration.
   crawlConfig.setSourceConfig(sourceConfig);
   ```

Note that the `RegexFilter` class has a `getRegex()` method to retrieve a regex value. In addition, the `SourceConfig` class has the `getExcludeFilters()` and `getIncludeFilters()` methods to retrieve the filters from the source configuration.

Creating date filters

The `DateFilter` class specifies a date against which files and folders can be filtered.

A `DateFilter` uses a datetime value to filter temporal-based properties, such as the `Endeca.FileSystem.ModificationDate` property (used in the example below).
The filter also uses a comparison operator that specifies how the operands are compared, using the enumerations:

- `BEFORE`
- `AFTER`

For example, if you create a date exclude filter that performs a `BEFORE` comparison against the `Endeca.FileSystem.ModificationDate` property, then files that have been modified before the date reference are excluded.

To create a date filter:

1. Make sure that you have created a `SourceConfig` and a `CrawlConfig`.

   For example:

   ```
   SourceConfig sourceConfig = new SourceConfig();
   ```

2. Instantiate a new, empty `DateFilter` object:

   ```
   DateFilter filter = new DateFilter();
   ```

3. Use the `setPropertyName()` method (inherited from the `Filter` class) to set the name of the property against which the filter will be applied:

   ```
   // Filter on the last-modified date.
   filter.setPropertyName("Endeca.FileSystem.ModificationDate");
   ```

4. Use the `setReferenceValue()` method to set the date/time value. Note that the Java API takes a `Date` object as its parameter and the WSDL-generated classes take a `XMLGregorianCalendar` object:

   For example:

   ```
   // Create a Date object.
   Date date = new Date();
   // set the time to noon on May 1, 2009
   date.setYear(2009);
   date.setMonth(5);
   date.setDay(1);
   date.setTime(12,0,0);
   filter.setReferenceValue(date);
   ```

5. Call the `setOperator()` method to specify that the filter will exclude files that have an earlier modification date:

   For example:

   ```
   // Exclude files with an earlier modification date.
   filter.setOperator(DateComparisonOperator.BEFORE);
   ```

6. Call the `setScope()` method (inherited from the `Filter` class) to set the filter scope. You can set the scope to files or to directories (`FilterScope.DIRECTORY`).

   For example:

   ```
   // Set the scope of the filter for only files.
   filter.setScope(FilterScope.FILE);
   ```

7. Create a list of `Filter` objects and use the `add()` method to add the date filter.

   For example:

   ```
   List<Filter> filterList = new ArrayList<Filter>();
   filterList.add(filter);
   ```

8. Use the `SourceConfig.setExcludeFilters()` method to set the populated list in the `SourceConfig` configuration object. If this were an inclusion filter, you would use the `SourceConfig.setIncludeFilters()` method instead.

   For example:

   ```
   // Set the filter in the source configuration.
   sourceConfig.setExcludeFilters(filterList);
   ```

9. Use the `CrawlConfig.setSourceConfig()` method to set the populated `SourceConfig` in the main `CrawlConfig` configuration object.

   For example:

   ```
   // Set the source config in the crawl configuration.
   crawlConfig.setSourceConfig(sourceConfig);
   ```

Note that the `DateFilter` class has a `getReferenceValue()` method to retrieve the `XMLGregorianCalendar` object. In addition, the `SourceConfig` class has the `getExcludeFilters()` and `getIncludeFilters()` methods to retrieve the filters from the source configuration.

Creating long filters

The `LongFilter` class specifies a long value against which files can be filtered. `LongFilter` extends the `ComparableValueFilter` class.

A `LongFilter` is a comparison filter that specifies a value (as a long) to be compared against a numerical property, such as the `Endeca.File.Size` property (used in the example below).The filter uses a comparison operator that specifies how the operands are compared, using the enumerations:

- `EQUAL`
- `GREATER`
- `GREATER_EQUAL`
- `LESS`
- `LESS_EQUAL`
- `NOT_EQUAL`

For example, if you create a long exclusion filter that performs a GREATER comparison against the `Endeca.File.Size` property, then files whose size is greater than the reference value are excluded.

To create a long filter:

1. Make sure that you have created a `SourceConfig` and a `CrawlConfig`.

   For example:

   ```
   SourceConfig sourceConfig = new SourceConfig();
   ```

2. Instantiate a new, empty `LongFilter` object:

   ```
   LongFilter filter = new LongFilter();
   ```

3. Use the `setPropertyName()` method (inherited from the `Filter` class) to set the name of the property against which the filter will be applied:

   ```
   // filter on the file size, which is in bytes
   filter.setPropertyName("Endeca.File.Size");
   ```

4. Use the `setReferenceValue()` method to set the long value to compare against the property:

   ```
   // exclude files larger than ~1GB
   filter.setReferenceValue(1000000000);
   ```

5. Call the `setOperator()` method (inherited from the `ComparableValueFilter` class) to specify that the filter will apply only to files that have a size greater than the reference value:

```
// exclude files with a size larger than the reference value
filter.setOperator(ComparisonOperator.GREATER);
```

6. Call the `setScope()` method (inherited from the `Filter` class) to set the filter scope. You can set the scope to files or to directories (`FilterScope.DIRECTORY`).

   For example:

   ```
   // set the scope of the filter for only files
   filter.setScope(FilterScope.FILE);
   ```

7. Create a list of `Filter` objects and use the `add()` method to add the filter.

   ```
   List<Filter> filterList = new ArrayList<Filter>();
   filterList.add(filter);
   ```

8. Use the `SourceConfig.setExcludeFilters()` method to set the populated list in the `SourceConfig` configuration object. If this were an inclusion filter, you would use the `SourceConfig.setIncludeFilters()` method instead.

   ```
   // set the filter in the source config
   sourceConfig.setExcludeFilters(filterList);
   ```

9. Use the `CrawlConfig.setSourceConfig()` method to set the populated `SourceConfig` in the main `CrawlConfig` configuration object.

   ```
   // set the source config in the main config
   crawlConfig.setSourceConfig(sourceConfig);
   ```

Note that the `LongFilter` class has a `getReferenceValue()` method to retrieve the long value and a `getPropertyName()` method to retrieve the Endeca property. In addition, the `SourceConfig` class has the `getExcludeFilters()` and `getIncludeFilters()` methods to retrieve the filters from the source configuration.

About the output properties for crawls

The `OutputConfig` class specifies whether the output from a crawl is stored in a Record Store instance or an output file.

The `OutputConfig` class uses two methods to set the properties: `setModuleId()` and `setModuleProperties()`.

Module ID

The `setModuleId()` method sets the module ID of the output type. You specify a string value to indicate the type of output. You can set the string to `File System` if you want the crawl output to go to a file system or set it to `Record Store` if you want the output to go to a Record Store instance.

You can set one output option per crawl configuration.

Module Properties

Each `ModuleProperty` is a key/value pair or a key/multi-value pair that provides configuration information about this an output type.

You specify a `ModuleProperty` by calling `setKey()` to specify a string representing the key and by calling `setValues()` to set one or more corresponding values.

You then set each`ModuleProperty` on the `SourceConfig` object by calling `addModuleProperty()`.

Record Store output properties and example

The `OutputConfig` class configures a crawl to write crawl output to a Record Store instance.

Table 2.2: Module Properties for Record Store output

Record Store Property Key Name	Key Value Description
`host`	The name of the host on which the Record Store is running. The default is `localhost`.
`port`	The port number on which the Record Store is listening. The default is `8510`.
`contextPath`	The WebLogic context path of the service location. This path is required for IAS installed into WebLogic. The path should be an empty string for IAS installed into Jetty. The default is an empty string.
`isPortSsl`	Specify how to interpret the `port` setting. A value of `true` means that `port` is an SSL port and the API uses HTTPS for connections. A value of `false` means that `port` is a non-SSL port and the API uses HTTP for connections. The default is `false`. Specify `false` if you enabled HTTPS redirects.
`instanceName`	The name of the Record Store instance that you want to write output to. The default is `<crawlID>`.
`isManaged`	A Boolean value that indicates whether the Record Store instance is managed or not. Management ties a Record Store instance to its corresponding crawl configuration. Specifying `true` indicates that a Record Store instance is created if you run a crawl and a Record Store instance does not already exist. Specifying `true` also indicates that a Record Store instance is deleted if you delete the corresponding crawl configuration. The default is `true` (is managed).

Here is an example of the output properties for a crawl writing to a Record Store instance.

```
// Create the output configuration.
OutputConfig outputConfig = new OutputConfig();

// Create a Record Store module ID.
ModuleId moduleId = new ModuleId("Record Store");

// Set the module ID in the output configuration.
outputConfig.setModuleId(moduleId);

// Create a module property object.
```

```
ModuleProperty host = new ModuleProperty();
// Set the key for specifying the host name.
host.setKey("host");
host.setValues("localhost");

// Create a module property object.
ModuleProperty port = new ModuleProperty();
// Set the key for specifying the port number.
port.setKey("port");
port.setValues("8401");

// Create a module property object.
ModuleProperty contextPath =new ModuleProperty();
contextPath.setKey("contextPath");
contextPath.setValues("");

// Create a module property object.
ModuleProperty instanceName = new ModuleProperty();
// Set the key for specifying the instance name of the Record Store.
instanceName.setKey("instanceName");
instanceName.setValues("RS1");

// Create a module property object.
ModuleProperty isManaged = new ModuleProperty();
// Set the key for specifying whether the Record Store is managed.
isManaged.setKey("isManaged");
isManaged.setValues("true");

// Create a list for the module property objects.
List<ModuleProperty> outputPropsList = new ArrayList<ModuleProperty>();

// Set the module property objects in the list.
outputPropsList.add(host);
outputPropsList.add(port);
outputPropsList.add(contextPath);
outputPropsList.add(instanceName);
outputPropsList.add(isManaged);

// Set the module property in the output config (if not already done).
outputConfig.setModuleProperties(outputPropsList);

// Set the output configuration in the main crawl configuration.
crawlConfig.setOutputConfig(outputConfig);

// Create the crawl.
crawler.createCrawl(crawlConfig);
```

Record file output properties and example

The `OutputConfig` class configures a crawl to write output to a record output file.

Table 2.3: Module Properties for record output files

File System Property Key Name	Key Value Description
`outputPrefix`	The prefix of the output file (`CrawlerOutput` is the default prefix). Optional.

File System Property Key Name	Key Value Description
`outputDirectory`	The name and path of the output directory under the IAS Server's workspace directory. The default name of `outputDirectory` is `output` and the default name of *<crawlID>* is used to create a subdirectory for each crawl. This ensures each crawl has a unique subdirectory for its output. For example, if you use the default value for `outputDirectory` and have a *<crawlID>* of `FileSystemCrawl`, the resulting directory structure is `IAS\workspace\output\FileSystemCrawl\`.
`outputXml`	A Boolean value that sets the output format to either XML or binary. Specifying `true` sets the output to XML. Specifying `false` sets the output to binary. The default is `false`.
`outputCompressed`	A Boolean value that indicates whether the output file should be compressed. Specifying `true` compresses the output. The default is `false` (not compressed). Optional.

Here is an example of the output properties for a file system crawl.

```
// Create the output configuration.
OutputConfig outputConfig = new OutputConfig();

// Create a file system module ID.
ModuleId moduleId = new ModuleId("File System");

// Set the module ID in the output configuration.
outputConfig.setModuleId(moduleId);

// Create a module property object.
ModuleProperty outputPrefix = new ModuleProperty();
// set the key for the output prefix
outputPrefix.setKey("outputPrefix");
outputPrefix.getValues().add("newPrefix");

// Set the outputPrefix module property on the output config.
outputConfig.addModuleProperty(outputPrefix);

// Create a module property object.
ModuleProperty outputDirectory = new ModuleProperty();
// Set the key for the output directory.
outputDirectory.setKey("outputDirectory");
outputDirectory.setValues("output");

// Set the outputDirectory module property on the output config.
outputConfig.addModuleProperty(outputDirectory);

// Create a module property object.
ModuleProperty outputXml = new ModuleProperty();
// Set the key for specifying whether output is in XML format.
outputXml.setKey("outputXml");
outputXml.setValues("true");

// Set the outputXml module property on the output config.
outputConfig.addModuleProperty(outputXml);

// Create a module property object.
ModuleProperty outputCompressed = new ModuleProperty();
// Set the key for specifying whether output is compressed.
```

```
outputCompressed.setKey("outputCompressed");
outputCompressed.setValues("true");

// Set the outputCompressed module property on the output config.
outputConfig.addModuleProperty(outputCompressed);

// Set the output config in the main crawl configuration.
crawlConfig.setOutputConfig(outputConfig);

// Create the crawl.
crawler.createCrawl(crawlConfig);
```

Listing crawls

Call the `IasCrawler.listCrawls()` method to list the existing crawls.

The syntax of the method is:

```
IasCrawler.listCrawls()
```

The method returns a `List<CrawlId>` object, which has zero or more `CrawlId` objects. Each `CrawlId` has the name of a crawl.

To list the set of existing crawls:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Use the `IasCrawler.listCrawls()` method to return a list of crawl names.

   For example:

   ```
   List<CrawlId> crawlList = crawler.listCrawls();
   ```
3. Call the `CrawlId.getId()` method to get the actual name (as a string) of each crawl.

You can also use the following to print out the number of crawls:

```
System.out.println("There are " + crawler.listCrawls().size() + " crawls configured");
```

The `IasCrawler.listCrawls()` method does not throw an exception if it fails.

Starting a crawl

Call the `IasCrawler.startCrawl()` method to start a crawl.

The syntax of the method is:

```
IasCrawler.startCrawl(CrawlId crawlId, CrawlMode crawlMode)
```

The `crawlId` parameter is a `CrawlId` object that has the crawl ID set. The `crawlMode` parameter is one of the following `CrawlMode` data types:

- `CrawlMode.FULL_CRAWL` performs a full crawl and creates a crawl history.
- `CrawlMode.INCREMENTAL_CRAWL` performs an incremental crawl and updates the crawl history. There are several cases in which the `CrawlMode` automatically switches over from `INCREMENTAL_CRAWL` to run a `FULL_CRAWL`. A full crawl runs in the following cases:
  - If a crawl has not been run before.

- If the document conversion option has changed - either by being enabled or disabled.
- If the repository properties have changed.
- If any filters have been modified, added, or removed.
- If any seeds have been removed.
- If you are writing records to a Record Store instance that contains no generations.

This method does not return a value.

To start a crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Instantiate a `CrawlId` object and then set its ID in the constructor.

   For example:

   ```
   // Create a new crawl ID with the name set to Demo.
   CrawlId crawlId = new CrawlId("Demo");
   ```

3. Call the `IasCrawler.startCrawl()` method with the crawl ID and the appropriate crawl mode. To catch exceptions, use a `try` block with the appropriate `catch` clauses.

   For example:

   ```
   try {
      crawler.startCrawl(crawlId, CrawlMode.INCREMENTAL_CRAWL);
   }
   catch (CrawlNotFoundException e) {
      System.out.println(e.getLocalizedMessage());
   }
   ```

If the `IasCrawler.startCrawl()` method fails, it throws an exception:

- `CrawlInProgressException` occurs if the IAS Server is already running the specified crawl.
- `CrawlNotFoundException` occurs if the specified crawl (the `crawlId` parameter) does not exist or is otherwise not found.
- `InvalidCrawlConfigException` occurs if the configuration is invalid. You can call `getCrawlValidationFailures()` to return the list of crawl validation errors.
- `EidiException` occurs if other problems prevent the crawl from running.

Stopping a crawl

Call the `IasCrawler.stopCrawl()` method to stop a crawl.

The syntax of the method is:

```
IasCrawler.stopCrawl(CrawlId crawlId)
```

The `crawlId` parameter is a `CrawlId` object that contains the name of the crawl to stop.

To stop a crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)

2. Set the name for the crawl to stop by first instantiating a `CrawlId` object and then its ID.

   For example:

   ```
   // Create a new crawl Id with the name set to Demo.
   CrawlId crawlId = new CrawlId("Demo");
   ```

3. Call the `IasCrawler.stopCrawl()` method with the crawl ID. To catch an exception, use a `try` block with the appropriate `catch` clause.

   For example:

   ```
   try {
      crawler.stopCrawl(crawlId);
   }
   catch (CrawlNotFoundException e) {
      System.out.println(e.getLocalizedMessage());
   }
   ```

The `IasCrawler.stopCrawl()` method throws a `CrawlNotFoundException` if the specified crawl (the `crawlId` parameter) does not exist or is otherwise not found.

When the stop request is issued, the crawl first goes into a `STOPPING` state and then (when it finally stops) into a `NOT_RUNNING` state.

Note: Stopping a crawl means that:

- The IAS Server produces no record output for the stopped crawl (and all Record Store transactions roll back).
- Crawl history returns to its previous state before the crawl started.
- Metrics do not roll back to their state before the crawl started.

Deleting crawls

Call the `IasCrawler.deleteCrawl()` method to delete an existing crawl.

The syntax of the method is:

```
IasCrawler.deleteCrawl(CrawlId crawlId)
```

The `crawlId` parameter is a `CrawlId` object that contains the name of the crawl to be deleted.

Note: You cannot delete a crawl that is running.

To delete a crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Set the name for the crawl to be deleted by first instantiating a `CrawlId` object and then setting Id in the constructor.

   For example:

   ```
   // Create a new crawl Id with the name set to Demo.
   CrawlId crawlId = new CrawlId("Demo");
   ```

3. Call the `IasCrawler.deleteCrawl()` method with the `CrawlId` object. To catch exceptions, use a `try` block with the appropriate `catch` clauses, as in this example:

```
try {
   crawler.deleteCrawl(crawlId);
}
catch (CrawlNotFoundException e) {
   System.out.println(e.getLocalizedMessage());
}
```

If the `IasCrawler.deleteCrawl()` method fails, it throws an exception:

- `CrawlInProgressException` occurs if the crawl is running.
- `CrawlNotFoundException` occurs if the specified crawl (the `crawlId` parameter) does not exist or is otherwise not found.
- `EidiException` occurs if a problem is encountered that prevents the crawl from being deleted.

Listing modules available to a crawl

Call the `IasCrawler.listModules()` method to return a list of modules you can include in a crawl. Modules include the default crawl types provided by IAS and any data source extensions and manipulator extensions you may have created using the IAS Extension API.

The syntax of the method is:

```
IasCrawler.listModules(ModuleType moduleType)
```

where `moduleType` is an enumeration value of either:

- `SOURCE` to return data sources
- `MANIPULATOR` to return manipulators

The method returns a `List<ModuleInfo>` object, which has zero or more `ModuleInfo` objects. Each `ModuleInfo` has the name and ID of a data source or manipulator.

To list the modules available to a crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Call the `IasCrawler.listModules()` method and specify an enumeration value to return either data sources or manipulators.

   For example:

   ```
   List<ModuleInfo> modules = crawler.listModules(ModuleType.SOURCE);
   ```

3. For each `ModuleInfo` object:
   (a) Call the `ModuleInfo.getModuleId()` method to get the ID of the module (the data source or manipulator).
   (b) Call the `ModuleInfo.getModuleType()` method to get the type of the module (the data source or manipulator).
   (c) Call the `ModuleInfo.getDescription()` method to get the description of the module (the data source or manipulator).

(d) Call the `ModuleInfo.getDisplayName()` method to get the display name of the module (the data source or manipulator).

For example:

```
List<ModuleInfo> moduleInfoList = modules.getModuleInfo();
for (ModuleInfo moduleInfo : moduleInfoList) {
   System.out.println(moduleInfo.getDisplayName());
   System.out.println(" *Id: "+ moduleInfo.getModuleId().getId());
   System.out.println(" *Type: "+ moduleInfo.getModuleType());
   System.out.println(" *Description: " + moduleInfo.getDescription());
   System.out.println();
}
```

The `IasCrawler.listModules()` method does not throw checked exceptions if it fails.

Retrieving crawl configurations

Call the `IasCrawler.getCrawlConfig()` method to retrieve the configuration settings of a crawl.

The syntax of the method is:

```
IasCrawler.getCrawlConfig(CrawlId crawlId, Boolean fillInDefaults)
```

Where:

- `crawlId` is a `CrawlId` object that contains the name of the crawl for which the configuration is to be returned.
- `fillInDefaults` is a Boolean flag that, if set to `true`, fills in the default value for any setting that has not been specified. If a setting is a password, `true` returns the name but not the value. If the flag is set to `false`, it does not modify the value for any setting.

If you retrieve a crawl configuration that contains a `ModuleProperty` for a password property, the crawl configuration retrieves the value as a zero length list.

The method returns a `CrawlConfig` object, which contains the following:

- `sourceConfig` - a `SourceConfig` object that contains the seeds, filters, and specific information about the systems from which content is fetched or whether file properties from the native file system should be gathered for file system crawls.
- `manipulatorConfig` - a list of `ManipulatorConfig` objects. Each `ManipulatorConfig` specifies a manipulation that is performed in a particular crawl.
- `textExtractionConfig` - a `TextExtractionConfig` object that contains the text extraction options, such as whether text extraction should be enabled and the number of retry attempts.
- `outputConfig` - an `OutputConfig` object that contains the output options, such as whether the records are written to a Record Store instance or a record output file, the path of the output directory and the output format (binary or XML).
- `crawlthreads` - a property indicating the number of threads per crawl.
- `loggingLevel` - a property indicating the logging level.

To get the configuration settings of a crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)

2. Set the name for the crawl by first instantiating a `CrawlId` object and then setting its Id.

   For example:

   ```
   // Create a new crawl Id with the name set to Demo.
   CrawlId crawlId = new CrawlId("Demo");
   ```

3. Call the `IasCrawler.getCrawlConfig()` method with the crawl ID and the default settings Boolean flag.

   For example:

   ```
   CrawlConfig crawlConfig = crawler.getCrawlConfig(crawlId, true);
   ```

4. Process the returned `CrawlConfig` according to the requirements of your application.

The `IasCrawler.getCrawlConfig()` method throws a `CrawlNotFoundException` if the specified crawl (the `crawlId` parameter) does not exist or is otherwise not found. To catch an exception, use a `try` block with the appropriate `catch` clause.

Updating crawl configurations

Call the `IasCrawler.updateCrawl()` method to change the configuration settings for an existing crawl.

The syntax of the method is:

```
IasCrawler.updateCrawl(CrawlConfig crawlConfig)
```

The `crawlConfig` parameter is a `CrawlConfig` object that has the configuration settings of the crawl.

If you update a crawl configuration and specify an empty `ModuleProperty` for a password property, the crawl configuration reuses the password stored on IAS Server.

Note: You cannot change the configuration if the crawl is running.

To update the configuration settings of an existing crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Set the name for the crawl to be modified by first instantiating a `CrawlId` object and then setting its ID in the constructor.

   For example:

   ```
   // Create a new crawl Id with the name set to Demo.
   CrawlId crawlId = new CrawlId("Demo");
   ```

3. Call the `IasCrawler.getCrawlConfig()` method to retrieve the current configuration.

   For example:

   ```
   CrawlConfig crawlConfig = crawler.getCrawlConfig(crawlId, false);
   ```

4. Change the configuration settings as desired.
5. Update the file system crawl by using the `IasCrawler.updateCrawl()` method with the previously created `crawlConfig`.

   For example:

```
crawler.updateCrawl(crawlConfig);
```

If the `IasCrawler.updateCrawl()` method fails, it throws an exception:

- `CrawlInProgressException` occurs if the crawl is running.
- `CrawlNotFoundException` occurs if the specified crawl (the `crawlId` parameter) does not exist or is otherwise not found.
- `InvalidCrawlConfigException` occurs if the configuration is invalid.

To catch these exceptions, use a `try` block when you call the method.

Getting crawl metrics

Call the `IasCrawler.getMetrics()` method to return the metrics of a crawl. Metrics can be returned for a running crawl or (if the crawl is not running) for the last complete crawl.

The syntax of the method is:

```
IasCrawler.getMetrics(CrawlId crawlId)
```

The `crawlId` parameter is a `CrawlId` object that contains the name of the crawl for which metrics are to be returned.

The method returns a `List<Metric>` object, which (if not empty) will have one or more `Metric` objects. A `Metric` is a key-value pair that holds the value of a particular metric. The keys are the metric's ID (a `MetricId` enum class). See the *IAS Server API Reference (Javadoc)* for the list of `MetricId` enumerations.

The `CRAWL_STOP_CAUSE MetricId` has one of the following values:

- COMPLETED
- FAILED
- ABORTED

If a crawl fails, the `CRAWL_FAILURE_REASON MetricId` provides a message from the IAS Server explaining the failure.

Your application can print out all or some of the metric values.

To get the metrics of a crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Set the name for the crawl by first instantiating a `CrawlId` object and then setting its ID.

   For example:

   ```
   // Create a new crawl ID with the name set to Demo.
   CrawlId crawlId = new CrawlId("Demo");
   ```

3. Call the `IasCrawler.getMetrics()` method with the crawl ID.

   For example:

   ```
   List<Metric> metricList = crawler.getMetrics(crawlId);
   ```

4. Print the metrics by retrieving the values from the `Metric` objects. For example, if you want to print the number of records that have been processed so far by a running crawl, the code would be:

```
if (crawler.getStatus(demoCrawlId).getState().equals(CrawlerState.RUNNING)) {
    List<Metric> metricList = crawler.getMetrics(crawlId);
    for (Metric metric : metricList) {
        MetricId id = metric.getMetricId();
        if (id.equals(MetricId.TOTAL_RECORDS)) {
            System.out.println("Total records: " + metric.toString());
        }
    }
}
```

The `IasCrawler.getMetrics()` method throws a `CrawlNotFoundException` if the specified crawl (the `crawlId` parameter) does not exist or is otherwise not found.

Getting the status of a crawl

Call the `IasCrawler.getStatus()` method to retrieve the status of a crawl.

The syntax of the method is:

```
IasCrawler.getStatus(CrawlId crawlId)
```

The `crawlId` parameter is a `CrawlId` object that contains the name of the crawl for which status is to be returned.

The method returns a `Status` object, which will have the status of the crawl as a `CrawlerState` simple data type:

- NOT_RUNNING
- STOPPING
- RUNNING

To get the status of a crawl:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Set the name for the crawl by first instantiating a `CrawlId` object and then setting its ID in the constructor.

   For example:

   ```
   // Create a new crawl ID with the name set to Demo.
   CrawlId crawlId = new CrawlId("Demo");
   ```

3. Declare a `CrawlerState` variable and initialize it by calling the `IasCrawler.getStatus()` method with the crawl ID. Note that the status is actually returned by the `State.getState()` method.

   For example:

   ```
   CrawlerState state;
   state = crawler.getStatus(crawlId).getState();
   ```

4. Print the status.

   For example:

   ```
   System.out.println("Crawl status: " + state);
   ```

The `IasCrawler.getStatus()` method throws a `CrawlNotFoundException` if the specified crawl (the `crawlId` parameter) does not exist or is otherwise not found. To catch an exception, use a `try` block with the appropriate `catch` clause.

Retrieving IAS Server information

Call the `Ias.getServerInfo()` method to get the server properties of the IAS Server.

The syntax of the method is:

```
IasCrawler.getServerInfo()
```

The method returns a `List<Property>` object, which contains `Property` objects with host machine and IAS Server information.

To retrieve information about the IAS Server:

1. Make sure that you have created a connection to the IAS Server. (An `IasCrawler` object named `crawler` is used in this example.)
2. Use the `IasCrawler.getServerInfo()` method to return the server information.

   For example:

   ```
   List<Property> serverInfo = crawler.getServerInfo();
   ```
3. Call the `Property.getKey()` and `Property.getValue()` methods to get the property key-value pairs.

The returned server properties (`Property` objects) contain the following key-value information:

Property key	Property value
eidi.version	The version of the IAS Server.
eidi.workspace	The path of the IAS Server workspace directory
os.arch	The hardware architecture on which the operating system is running (such as `amd64`), as specified in the IAS Server's JVM.
os.name	The operating system of the machine on which the IAS Server is running (such as `Windows 2003`), as specified in the IAS Server's JVM.
os.version	The version of the operating system of the machine on which the IAS Server is running (such as `5.2`), as specified in the IAS Server's JVM.

The `Ias.getServerInfo()` method does not throw an exception if it fails.

Chapter 3

Component Instance Manager API

This section documents the Component Instance Manager (CIM) API.

Component Instance Manager client utility classes

The Component Instance Manager API provides client utility classes for the manipulation of objects.

ComponentInstanceManagerLocator class

The `ComponentInstanceManagerLocator` class creates a connection to a Component Instance Manager server. The steps to create a connection are:

1. Create a `ServiceAddress` object and specify the host and port of the server running the Component Instance Manager, and if you installed IAS into WebLogic, also specify the `contextPath`. If you installed IAS into Jetty, set the `contextPath` to an empty string.
2. Call the `create()` method on `ComponentInstanceManagerLocator` and pass in the `ServiceAddress` object. For example:

   ```
   ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
   ComponentInstanceManagerLocator locator = ComponentInstanceManagerLocator.create(address);
   ```

3. Call the `getService()` method to make a connection to the Component Instance Manager service on that server:

   ```
   ComponentInstanceManager cim = locator.getService();
   ```

Component Instance Manager core operations

The Component Instance Manager API has a `ComponentInstanceManager` interface, which is used to create, list, and delete Record Store instances. In this release, Record Store components are the only supported component type.

The following Component Instance Manager core operations are provided by methods in the `ComponentInstanceManager` interface:

- `createComponentInstance()` creates a component instance of the given type with the given ID.
- `deleteComponentInstance()` deletes the given component instance.
- `listComponentInstances()` lists all component instances defined in the system.
- `listComponentTypes()` lists all component types defined in the system.

Note: The syntax descriptions for these operations use Java conventions. The exact syntax of a class member depends on the output of the WSDL tool that you are using.

Creating a component

Call the `ComponentInstanceManager.createComponentInstance()` method to create a component instance of the given type (a `RecordStore`) with the given ID (a Record Store instance name).

The syntax of the method is:

```
ComponentInstanceManager.createComponentInstance(ComponentTypeId componentTypeId,
ComponentInstanceId componentInstanceId)
```

The `componentTypeId` parameter is a `ComponentTypeId` that should be set to `"RecordStore"`.

The `componentInstanceId` parameter is a `ComponentInstanceId` that is the Record Store instance name.

To create a component:

1. Create a `ServiceAddress` object and specify the host and port of the server running the Component Instance Manager, and if you installed IAS into WebLogic, also specify the `contextPath`. If you installed IAS into Jetty, set the `contextPath` to an empty string.
2. Call the `create()` method on `ComponentInstanceManagerLocator` and pass in the `ServiceAddress` object. For example:

   ```
   ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
   ComponentInstanceManagerLocator locator = ComponentInstanceManagerLocator.create(address);
   ```
3. Create a `ComponentInstanceManager` object and call `getService()` to establish a connection to the server and the Component Instance Manager service. For example:

   ```
   ComponentInstanceManager cim = locator.getService();
   ```
4. Create a Record Store instance by calling `createComponentInstance()` and specifying `RecordStore` and a Record Store instance name. For example:

   ```
   cim.createComponentInstance(new ComponentTypeId("RecordStore"),
   new ComponentInstanceId("rs1"));
   ```

Deleting a component

Call the `ComponentInstanceManager.deleteComponentInstance()` method to delete a specified component instance (a Record Store).

The syntax of the method is:

```
ComponentInstanceManager.deleteComponentInstance(ComponentInstanceId componentInstanceId)
```

The `componentInstanceId` parameter is a `ComponentInstanceId` that is the Record Store instance name that you want to delete.

To delete a component:

1. Create a `ServiceAddress` object and specify the host and port of the server running the Component Instance Manager, and if you installed IAS into WebLogic, also specify the `contextPath`. If you installed IAS into Jetty, set the `contextPath` to an empty string.

2. Call the `create()` method on `ComponentInstanceManagerLocator` and pass in the `ServiceAddress` object. For example:

```
ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
ComponentInstanceManagerLocator locator = ComponentInstanceManagerLocator.create(address);
```

3. Create a `ComponentInstanceManager` object and call `getService()` to establish a connection to the server and the Component Instance Manager service. For example:

```
ComponentInstanceManager cim = locator.getService();
```

4. Delete a Record Store instance by calling `deleteComponentInstance()` and specifying a Record Store instance name. For example:

```
cim.deleteComponentInstance(new ComponentInstanceId("rs1");
```

If the `ComponentInstanceManager.deleteComponentInstance()` method fails, it will throw an exception:

- `ComponentInstanceNotFoundException` is thrown if the Component Instance Manager does not contain the component instance.
- `ComponentManagerException` is thrown if there was an error stopping the component instance.

To catch these exceptions, use a `try` block when you call the method.

Listing component instances

Call the `ComponentInstanceManager.listComponentInstances()` method to list all component instances in the Endeca IAS Service. In this release, components are Record Store instances that are running in the Endeca IAS Service.

The syntax of the method is:

```
ComponentInstanceManager.listComponentInstances()
```

The method returns a list of `ComponentInstanceDescriptor` objects. Each `ComponentInstanceDescriptor` object represents a single component (that is, a Record Store instance) and is made up of the following:

- `TypeId` object. This is the component type. For example, in this release, it is always `RecordStore`.
- `InstanceId` object. This is the user-specified name of an instance.
- `InstanceStatus` object. This is the status of a Record Store instance. This value can be one of the following constants: `RUNNING`, `FAILED`, or `STOPPED`.

To list component instances:

1. Create a `ServiceAddress` object and specify the host and port of the server running the Component Instance Manager, and if you installed IAS into WebLogic, also specify the `contextPath`. If you installed IAS into Jetty, set the `contextPath` to an empty string.
2. Call the `create()` method on `ComponentInstanceManagerLocator` and pass in the `ServiceAddress` object. For example:

```
ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
ComponentInstanceManagerLocator locator = ComponentInstanceManagerLocator.create(address);
```

3. Create a `ComponentInstanceManager` object and call `getService()` to establish a connection to the server and the Component Instance Manager service. For example:

```
ComponentInstanceManager cim = locator.getService();
```

4. Call `listComponentInstances()` and then create a `for` loop to loop over all component instances. Inside the loop, get the `TypeId`, `InstanceId`, and `InstanceStatus` and print them to system out (or elsewhere). For example:

```
for (ComponentInstanceDescriptor desc : cim.listComponentInstances()) {
    System.out.println(desc.getInstanceId() + " of type " + desc.getTypeId()
+ " has status " + desc.getInstanceStatus());
}
```

Listing component types

Call the `ComponentInstanceManager.listComponentTypes()` method to list all component types in the Endeca IAS Service. In this release, there are only components of type `RecordStore`.

The syntax of the method is:

```
ComponentInstanceManager.listComponentTypes()
```

The method returns a list of `ComponentTypeDescriptor` objects. Each `ComponentTypeDescriptor` object is made up of a `TypeId` object and an `InstallPath` object.

Each `TypeId` has the component type, for example, `RecordStore`. Each `InstallPath` is a string representing the absolute path to the WAR file implementing the component itself, for example, `C:\Oracle\Endeca\IAS\<version>\components\RecordStore.war`.

To list component types:

1. Create a `ServiceAddress` object and specify the host and port of the server running the Component Instance Manager, and if you installed IAS into WebLogic, also specify the `contextPath`. If you installed IAS into Jetty, set the `contextPath` to an empty string.
2. Call the `create()` method on `ComponentInstanceManagerLocator` and pass in the `ServiceAddress` object. For example:

```
ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
ComponentInstanceManagerLocator locator = ComponentInstanceManagerLocator.create(address);
```

3. Create a `ComponentInstanceManager` object and call `getService()` to establish a connection to the server and the Component Instance Manager service. For example:

```
ComponentInstanceManager cim = locator.getService();
```

4. Call `listComponentTypes()` and then create a `for` loop to loop over all component types in the system. Inside the loop, get the `TypeId` and `InstallPath` and print them to system out (or elsewhere). For example:

```
for (ComponentTypeDescriptor desc : cim.listComponentTypes()) {
    System.out.println(desc.getTypeId() + " installed at " + desc.getInstallPath());
}
```

Chapter 4

Record Store API

This section documents the Record Store API.

Record Store client utility classes

The Record Store API provides client utility classes to manage a Record Store and perform read/write operations.

The Record Store API includes a set of client utility classes that are useful for working with objects, such as the creation of record collections. Java versions of these classes are included in the `recordstore-api-3.2.0.jar` library.

A brief overview of these classes is given below. For details on the signatures and arguments, refer to the *Record Store API Reference (Javadoc)*.

RecordStoreLocator class

The `RecordStoreLocator` class creates a connection to a Record Store server. The steps for obtaining a connection are:

1. Create a `ServiceAddress` object and specify the host and port of the server running the Record Store, and if you installed IAS into WebLogic, also specify the `contextPath`. If you installed IAS into Jetty, set the `contextPath` to an empty string.
2. Call the `create()` method on `ComponentInstanceManagerLocator` and pass in the `ServiceAddress` object. For example:

   ```
   ServiceAddress address = new ServiceAddress("localhost", 8401, contextPath);
   RecordStoreLocator locator = RecordStoreLocator.create(address, "MyCrawl");
   ```

3. Call the `ServiceLocator.getService()` method to make a connection to the Record Store service on that server:

   ```
   RecordStore rs = locator.getService();
   ```

The class also has other getter and setter methods for configuring communication with a Record Store instance.

RecordStoreWriter class

The `RecordStoreWriter` class provides methods for writing records to a Record Store instance.

The class has two `write()` methods that allow you to write one record at a time or a list of records all at once.

You can create a baseline writer with this method:

```
RecordStoreWriter writer = RecordStoreWriter.createWriter(
    recordStore, tId, 100);
```

RecordStoreReader class

The `RecordStoreReader` class provides methods for reading baseline and delta records from a Record Store instance.

The `RecordStoreReader` class does not have a reader for reading individual records by their ID. To perform this type of read, use the `RecordStore.readRecordsById()` method from the WSDL (core operations).

You can create a reader with this method:

```
RecordStoreReader reader = RecordStoreReader.createBaselineReader(
    recordStore, tId, gId, 100);
```

The `RecordStoreWriter` and `RecordStoreReader` classes are useful because they handle batching and un-batching of records.

Record Store core operations

This topic presents an overview of the Record Store API core methods.

The Record Store API has a `RecordStore` interface, which is used to make calls to a Record Store instance.

The following Record Store core operations are provided by methods in the `RecordStore` interface:

- `startTransaction()` starts a transaction of type READ or READ_WRITE and returns the transaction ID.
- `startBaselineRead()` creates a read cursor for reading a baseline generation from a Record Store instance.
- `startDeltaRead()` creates a read cursor for an incremental read from a Record Store instance.
- `readRecords()` performs the actual read operation for a read cursor set up by either the `startBaselineRead()` or the `startDeltaRead()` method.
- `endRead()` ends a baseline or incremental read operation performed by a `readRecords()` method.
- `readRecordsById()` reads specific records from a Record Store instance, based on a list of their record IDs.
- `writeRecords()` writes a set of records to a Record Store instance. The method returns an integer that indicates how many records were actually written.
- `commitTransaction()` commits an active (uncommitted) transaction.
- `rollbackTransaction()` rolls back an active (uncommitted) transaction.
- `listActiveTransactions()` returns a `List` of `TransactionInfos` that contain the ID, type, status, and generation ID of each active transaction.

- `listGenerations()` returns a `List` of `GenerationInfos` for each record generation currently in the Record Store.
- `getLastCommittedGenerationId()` gets the ID of the last-committed record generation.
- `getWriteGenerationId()` gets the ID of the current generation.
- `setLastReadGenerationId()` sets state for a specific client by setting the ID of the last generation read by the client.
- `getLastReadGenerationId()` gets the ID of the last-read generation that was set for a specific client.
- `listClientStates()` returns a `List` of `ClientStateInfos` for each client. Each `ClientStateInfo` object contains a client ID, a transaction ID, a generation ID of the last read generation, and a Boolean to indicate if the state is committed.
- `getConfiguration()` returns the configuration settings of a specified Record Store instance.
- `setConfiguration()` sets the configuration settings of a specified Record Store instance.
- `clean()` runs the Record Store Cleaner, which removes all records that are no longer necessary. This method allows cleaning to occur on an external schedule.

Note: The examples in this guide use client stubs generated with Apache CXF 2.2. However, the exact syntax of a class member depends on the output of the WSDL tool that you are using.

Getting and setting a Record Store instance configuration

Use the `getConfiguration()` and `setConfiguration()` methods to get a Record Store instance configuration and configure settings for the Record Store instance.

To get and set a Record Store instance configuration:

1. Create a connection to a Record Store server by calling the `create()` method and passing in a `ServiceAddress` object and a Record Store Instance name:

   ```
   ServiceAddress address = new ServiceAddress(host, port, contextPath);
   RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);
   ```

2. Create a Record Store instance by calling the `getService()` method:

   ```
   RecordStore recordStore = locator.getService();
   ```

3. Return the `config` object for the new Record Store instance by calling the `getConfiguration()` method:

   ```
   RecordStoreConfiguration config = recordStore.getConfiguration(false);
   ```

4. Enable compression by calling the `setRecordCompressionEnabled()` method:

   ```
   config.setRecordCompressionEnabled(true);
   ```

5. Set the modified configuration for the Record Store instance by calling the `setConfiguration()` method:

   ```
   recordStore.setConfiguration(config);
   ```

Example of getting and setting a Record Store instance configuration

```
ServiceAddress address = new ServiceAddress(host, port, contextPath);
RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);

RecordStore recordStore = locator.getService();

RecordStoreConfiguration config = recordStore.getConfiguration(false);

config.setRecordCompressionEnabled(true);

recordStore.setConfiguration(config);
```

Running a baseline read of the last-committed generation

Call the `startBaselineRead()` method to create a cursor for a baseline read to be consumed by the `readRecords()` method.

To run a baseline read of the last-committed generation:

1. Create a connection to a Record Store server by calling the `create()` method and passing in a `ServiceAddress` object and a Record Store Instance name:

   ```
   ServiceAddress address = new ServiceAddress(host, port, contextPath);
   RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);
   ```

2. Create a Record Store instance by calling the `getService()` method:

   ```
   RecordStore recordStore = locator.getService();
   ```

3. Start a `READ` transaction by calling the `startTransaction()` method:

   ```
   TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);
   ```

4. Return a `ReadCursorId` object by calling the `startBaselineRead()` method:

   ```
   ReadCursorId readCursorId = recordStore.startBaselineRead(transactionId, null);
   ```

5. Loop over the records returned by `readRecords()` until all records from the read cursor are read:

   ```
   List<Record> records;

   do {

       records = recordStore.readRecords(readCursorId, numRecordsPerFetch);

       // do something with the records

   } while (!records.isEmpty());
   ```

6. End the `READ` transaction by calling the `endRead()` method:

   ```
   recordStore.endRead(readCursorId);
   ```

7. Commit the transaction by calling the `commitTransaction()` method:

   ```
   recordStore.commitTransaction(transactionId);
   ```

Example of running a baseline read

```
ServiceAddress address = new ServiceAddress(host, port, contextPath);
RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);
```

```
RecordStore recordStore = locator.getService();

TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);

ReadCursorId readCursorId = recordStore.startBaselineRead(transactionId, null);

List<Record> records;

do {

    records = recordStore.readRecords(readCursorId, numRecordsPerFetch);

    // do something with the records

} while (!records.isEmpty());

recordStore.endRead(readCursorId);

recordStore.commitTransaction(transactionId);
```

Running a delta read

Call the `startDeltaRead()` method to create a cursor for a delta (incremental) read to be consumed by the `readRecords()` method.

To run a delta read:

1. Create a connection to a Record Store server by calling the `create()` method and passing in a `ServiceAddress` object and a Record Store Instance name:

   ```
   ServiceAddress address = new ServiceAddress(host, port, contextPath);
   RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);
   ```

2. Create a Record Store instance by calling the `getService()` method:

   ```
   RecordStore recordStore = locator.getService();
   ```

3. Start a `READ` transaction by calling the `startTransaction()` method:

   ```
   TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);
   ```

4. Create a `ReadCursorId` object by calling the `startDeltaRead()` method:

   ```
   ReadCursorId readCursorId
   = recordStore.startDeltaRead(transactionId, startGeneration, endGeneration);
   ```

5. Loop over the records returned by `readRecords()` until all records from the read cursor are read:

   ```
   List<Record> records;

   do {

       records = recordStore.readRecords(readCursorId, numRecordsPerFetch);

       // do something with the records

   } while (!records.isEmpty());
   ```

6. End the `READ` transaction by calling the `endRead()` method:

   ```
   recordStore.endRead(readCursorId);
   ```

7. Commit the transaction by calling the `commitTransaction()` method:

```
recordStore.commitTransaction(transactionId);
```

Example of running a delta read

```
ServiceAddress address = new ServiceAddress(host, port, contextPath);
RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);

RecordStore recordStore = locator.getService();

TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);

ReadCursorId readCursorId = recordStore.startDeltaRead(transactionId, startGeneration, endGeneration)
;

List<Record> records;

do {

    records = recordStore.readRecords(readCursorId, numRecordsPerFetch);

    // do something with the records

} while (!records.isEmpty());

recordStore.endRead(readCursorId);

recordStore.commitTransaction(transactionId);
```

Maintaining client read state in the Record Store

Use the `getLastCommittedGenerationId()` and `setLastReadGenerationId()` methods to store the `GenerationId` that the client last read.

To maintain client read state in the Record Store:

1. Create a connection to a Record Store server by calling the `create()` method and passing in a `ServiceAddress` object and a Record Store Instance name:

```
ServiceAddress address = new ServiceAddress(host, port, contextPath);
RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);
```

2. Create a Record Store instance by calling the `getService()` method:

```
RecordStore recordStore = locator.getService();
```

3. Start a `READ` transaction by calling the `startTransaction()` method:

```
TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);
```

4. Get the last-committed generation by calling the `getLastCommittedGenerationId()` method:

```
GenerationId gid = recordStore.getLastCommittedGenerationId(transactionId);
```

5. Return a `ReadCursorId` object by calling the `startBaselineRead()` method:

```
ReadCursorId readCursorId = recordStore.startBaselineRead(transactionId, gid);
```

6. Loop over the records returned by `readRecords()` until all records from the read cursor are read:

```
List<Record> records;

do {

    records = recordStore.readRecords(readCursorId, numRecordsPerFetch);
    // do something with the records

} while (!records.isEmpty());
```

7. End the `READ` transaction by calling the `endRead()` method:

```
recordStore.endRead(readCursorId);
```

8. Set the last-read generation ID by calling the `setLastReadGenerationId()` method:

```
recordStore.setLastReadGenerationId(transactionId, clientId, gid);
```

9. Commit the transaction by calling the `commitTransaction()` method:

```
recordStore.commitTransaction(transactionId);
```

10. At a later point, start a new `READ` transaction for an incremental read by calling the `startTransaction()` method:

```
TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);
```

11. Get the last-committed generation by calling the `getLastCommittedGenerationId()` method:

```
GenerationId gid = recordStore.getLastCommittedGenerationId(transactionId);
```

12. Create a `ReadCursorId` object by calling the `startDeltaRead()` method:

```
ReadCursorId readCursorId
= recordStore.startDeltaRead(transactionId, startGeneration, endGeneration);
```

13. Loop over the records returned by `readRecords()` until all records from the read cursor are read:

```
List<Record> records;

do {

    records = recordStore.readRecords(readCursorId, numRecordsPerFetch);

    // do something with the records

} while (!records.isEmpty());
```

14. End the `READ` transaction by calling the `endRead()` method:

```
recordStore.endRead(readCursorId);
```

15. Set client state by calling the `setLastReadGenerationId()` method:

```
recordStore.setLastReadGenerationId(transactionId, clientId, endGenerationId);
```

16. Commit the transaction by calling the `commitTransaction()` method:

```
recordStore.commitTransaction(transactionId);
```

Example of maintaining client read state in the Record Store

```
ServiceAddress address = new ServiceAddress(host, port, contextPath);
```

```
RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);

RecordStore recordStore = locator.getService();
// Run a baseline read

TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);

GenerationId gid = recordStore.getLastCommittedGenerationId(transactionId);

ReadCursorId readCursorId = recordStore.startBaselineRead(transactionId, gid);

List<Record> records;

do {

    records = recordStore.readRecords(readCursorId, numRecordsPerFetch);

    // do something with the records

} while (!records.isEmpty());

recordStore.endRead(readCursorId);

recordStore.setLastReadGenerationId(transactionId, clientId, gid);

recordStore.commitTransaction(transactionId);


...


// Run a delta read at a later point

TransactionId transactionId = recordStore.startTransaction(TransactionType.READ);

GenerationId startGenerationId = recordStore.getLastReadGenerationId(transactionId, clientId);

GenerationId endGenerationId = recordStore.getLastCommittedGenerationId(transactionId);

ReadCursorId readCursorId
= recordStore.startDeltaRead(transactionId, startGenerationId, endGenerationId);

List<Record> records;

do {

    records = recordStore.readRecords(readCursorId, numRecordsPerFetch);

    // do something with the records

} while (!records.isEmpty());

recordStore.endRead(readCursorId);

recordStore.setLastReadGenerationId(transactionId, clientId, endGenerationId);

recordStore.commitTransaction(transactionId);
```

Performing an incremental write

Use the `writeRecords()` method to write an incremental set of records to the Record Store.

To perform an incremental write:

1. Create a connection to a Record Store server by calling the `create()` method and passing in a `ServiceAddress` object and a Record Store Instance name:

   ```
   ServiceAddress address = new ServiceAddress(host, port, contextPath);
   RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);
   ```

2. Create a Record Store instance by calling the `getService()` method:

   ```
   RecordStore recordStore = locator.getService();
   ```

3. Start a `READ_WRITE` transaction by calling the `startTransaction()` method:

   ```
   TransactionId transactionId = recordStore.startTransaction(TransactionType.READ_WRITE);
   ```

4. Write a batch of records by calling the `writeRecords()` method:

   ```
   recordStore.writeRecords(recordBatch1);
   ```

   Repeat this step to write other batches of records to the Record Store.

5. Commit the transaction by calling the `commitTransaction()` method:

   ```
   recordStore.commitTransaction(transactionId);
   ```

Example of performing an incremental write

```
ServiceAddress address = new ServiceAddress(host, port, contextPath);
RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);

RecordStore recordStore = locator.getService();

TransactionId transactionId = recordStore.startTransaction(TransactionType.READ_WRITE);

recordStore.writeRecords(recordBatch1);

recordStore.writeRecords(recordBatch2);

recordStore.commitTransaction(transactionId);
```

Performing a baseline write

Create a `deleteAllRecord`, then use the `writeRecords()` method to write a baseline set of records to the Record Store.

To perform a baseline write:

1. Create a connection to a Record Store server by calling the `create()` method and passing in a `ServiceAddress` object and a Record Store Instance name:

   ```
   ServiceAddress address = new ServiceAddress(host, port, contextPath);
   RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);
   ```

2. Create a Record Store instance by calling the `getService()` method:

```
RecordStore recordStore = locator.getService();
```

3. Start a `READ_WRITE` transaction by calling the `startTransaction()` method:

```
TransactionId transactionId = recordStore.startTransaction(TransactionType.READ_WRITE);
```

4. Create a new record called `deleteAllRecord` with a property value of `DELETE`:

```
Record deleteAllRecord = new Record();

deleteAllRecord.addPropertyValue(new PropertyValue("Endeca.Action", "DELETE"));
```

5. Add `deleteAllRecord` as the first record in a record batch:

```
recordBatch1.addFirst(deleteAllRecord);
```

6. Write the first batch of records by calling the `writeRecords()` method:

```
recordStore.writeRecords(recordBatch1);
```

   Repeat this step to write other batches of records to the Record Store.

7. Commit the transaction by calling the `commitTransaction()` method:

```
recordStore.commitTransaction(transactionId);
```

Example of performing a baseline write

```
ServiceAddress address = new ServiceAddress(host, port, contextPath);
RecordStoreLocator locator = RecordStoreLocator.create(address, instanceName);

RecordStore recordStore = locator.getService();

TransactionId transactionId = recordStore.startTransaction(TransactionType.READ_WRITE);

Record deleteAllRecord = new Record();

deleteAllRecord.addPropertyValue(new PropertyValue("Endeca.Action", "DELETE"));

recordBatch1.addFirst(deleteAllRecord);

recordStore.writeRecords(recordBatch1);

recordStore.writeRecords(recordBatch2);

recordStore.commitTransaction(transactionId);
```

Sample Writer client example

This sample program shows how to write records to the Record Store.

The `SampleWriter.java` class is an example of how to use the core and client utility classes to write records. The sample Java program creates one record and writes it to the Record Store.

The code works as follows:

1. The PROPERTY_ID variable uses the setting of the Record Store instance `idPropertyName` configuration property, which is used to identify the records.

```
public static final String PROPERTY_ID = "Endeca.FileSystem.Path";
```

2. A sample record is created with the `Record` class and added to the records Collection.

```
Collection<Record> records = new LinkedList<Record>();
  Record record = new Record();
  record.addPropertyValue(new PropertyValue(PROPERTY_ID, "id1"));
  record.addPropertyValue(new PropertyValue("property.name", "property.value"));
  records.add(record);
```

3. Using the `RecordStoreLocator` utility class, a connection is made to the Record Store Server.

```
ServiceAddress address = new ServiceAddress(iasHost, iasPort, contextPath);
  RecordStoreLocator locator = RecordStoreLocator.create(address, "rs1");
  RecordStore recordStore = locator.getService();
```

4. In a `try` block, a READ_WRITE transaction was created by the `RecordStore.startTransaction()` core method and the `RecordStoreWriter.createWriter()` method is used to create a writer. This example writer writes a maximum of 100 records per transfer.

```
try {
  System.out.println("Setting record store configuration ...");
  recordStore.setConfiguration(config);

  System.out.println("Starting a new transaction ...");
  tId = recordStore.startTransaction(TransactionType.READ_WRITE);

  RecordStoreWriter writer = RecordStoreWriter.createWriter(recordStore, tId, 100);
...
```

5. The writer first writes a "Delete All" record, then writes the sample record, and finally closes the writer. Note that the record is written twice (the first time as part of a collection and the second as an individual record), in order to demonstrate both methods.

```
System.out.println("Writing records ...");
  writer.deleteAll();
  writer.write(records);
  writer.close();
```

6. The client program uses the `RecordStore.commitTransaction()` core method to commit the write transaction.

```
System.out.println("Committing transaction ...");
  recordStore.commitTransaction(tId);

  System.out.println("DONE");
```

After the transaction is committed, the Record Store contains a new record generation.

SampleWriter.java

```
package com.endeca.eidi.recordstore.sample;

import com.endeca.eidi.EidiConstants;
import com.endeca.eidi.record.PropertyValue;
import com.endeca.eidi.record.Record;
import com.endeca.eidi.recordstore.RecordStore;
import com.endeca.eidi.recordstore.RecordStoreException;
import com.endeca.eidi.recordstore.RecordStoreLocator;
import com.endeca.eidi.recordstore.RecordStoreWriter;
import com.endeca.eidi.recordstore.TransactionId;
import com.endeca.eidi.recordstore.TransactionType;
import com.endeca.eidi.service.ServiceAddress;

/**
```

```
 * SampleWriter is an example of how to use the Record Store core and client
 * utility classes to write records. It creates records and writes them to the
 * Record Store.
 */
public class SampleWriter {

    // This should match the idPropertyName in your record store configuration.
    public static final String ID_PROPERTY_NAME = "Endeca.Id";

    public static void main(String[] args) {
        if (args.length != 2 && args.length != 3) {
            System.out.println("Usage: <ias host> <ias port> [ias context path]");
            System.exit(-1);
        }

        String iasHost = args[0];
        int iasPort = Integer.parseInt(args[1]);
        String contextPath = (args.length == 3) ? args[2] : EidiConstants.DEFAULT_CONTEXT_PATH;

        ServiceAddress address = new ServiceAddress(iasHost, iasPort, contextPath);
        RecordStoreLocator locator = RecordStoreLocator.create(address, "rs1");
        RecordStore recordStore = locator.getService();

        TransactionId transactionId = null;
        try {
            System.out.println("Starting a new transaction ...");
            transactionId = recordStore.startTransaction(TransactionType.READ_WRITE);

            RecordStoreWriter writer = RecordStoreWriter.createWriter(recordStore, transactionId);

            System.out.println("Writing records ...");

            // Start by deleting all records in the new Record Store generation.
            // This should be done when doing a baseline write to the Record Store.
            // It should not be done when doing an incremental import into the Record
            // Store.
            writer.deleteAll();

            // Write a record to the Record Store
            writer.write(createRecord(
                    ID_PROPERTY_NAME, "record1",
                    "fruit", "apple",
                    "color", "red"));

            // Write another record to the Record Store
            writer.write(createRecord(
                    ID_PROPERTY_NAME, "record2",
                    "fruit", "banana",
                    "color", "yellow"));

            // Close the RecordStoreWriter. This will flush the client
            // side record buffer.
            writer.close();

            System.out.println("Committing transaction ...");
            recordStore.commitTransaction(transactionId);

            System.out.println("DONE");
        } catch (RecordStoreException exception) {
            exception.printStackTrace();
            if (transactionId != null) {
                try {
                    recordStore.rollbackTransaction(transactionId);
                } catch (RecordStoreException anotherException) {
                    System.out.println("Failed to roll back transaction.");
                    anotherException.printStackTrace();
                }
            }
```

```
        }
    }

    private static Record createRecord(String... nameValuePairs) {
        if (nameValuePairs.length % 2 != 0) {
            throw new IllegalArgumentException("Missing property value for property " +
                    nameValuePairs[nameValuePairs.length-1]);
        }
        Record record = new Record();
        for (int i = 0; i < nameValuePairs.length; i = i + 2) {
            record.addPropertyValue(new PropertyValue(nameValuePairs[i], nameValuePairs[i + 1]));
        }
        return record;
    }
}
```

Sample Reader client example

This sample program shows how to read records from the Record Store.

The `SampleReader.java` class is an example of how to use the core and client utility classes to read records. The sample program gets the ID of the last-committed generation and reads its records from the Record Store.

The code works as follows:

1. Using the `RecordStoreLocator` utility class, a connection is made to the Record Store Server.

   ```
   ServiceAddress address = new ServiceAddress(iasHost, iasPort, contextPath);
     RecordStoreLocator locator = RecordStoreLocator.create(address, "rs1");
     RecordStore recordStore = locator.getService();
   ```

2. In a `try` block, the `RecordStore.startTransaction()` core method creates a READ transaction and then the `RecordStore.getLastCommittedGenerationId()` core method gets the ID of the last generation that was committed to the Record Store.

   ```
   TransactionId tId = null;
     try {
       System.out.println("Starting a new transaction ...");
       tId = recordStore.startTransaction(TransactionType.READ);

       System.out.println("Getting the last committed generation ...");
       GenerationId gId = recordStore.getLastCommittedGenerationId(tId);
   ```

3. The `RecordStoreReader.createBaselineReader()` utility method is used to create a baseline reader. The reader transfers a maximum of 100 records per transfer.

   ```
   System.out.println("Reading records ...");
     RecordStoreReader reader
   = RecordStoreReader.createBaselineReader(recordStore, tId, gId, 100);
     int count = 0;
   ```

4. In a `while` loop, the `hasNext()` method tests whether the reader has another record to read. If true, the `next()` method retrieves the record, the record is written out, and the record-read count is increased by one. When there are no more records to read, the `close()` method closes the reader, and the number of records is printed out.

   ```
   while (reader.hasNext()) {
     Record record = reader.next();
     System.out.println("  RECORD: " + record);
     count++;
   ```

```
  }
reader.close();
System.out.println(count + " record(s) read");
```

5. The client program uses the `RecordStore.commitTransaction()` core method to commit the read transaction.

```
System.out.println("Committing transaction ...");
  recordStore.commitTransaction(tId);

  System.out.println("DONE");
```

SampleReader.java

```
package com.endeca.eidi.recordstore.sample;

import com.endeca.eidi.EidiConstants;
import com.endeca.eidi.record.Record;
import com.endeca.eidi.recordstore.GenerationId;
import com.endeca.eidi.recordstore.RecordStore;
import com.endeca.eidi.recordstore.RecordStoreException;
import com.endeca.eidi.recordstore.RecordStoreLocator;
import com.endeca.eidi.recordstore.RecordStoreReader;
import com.endeca.eidi.recordstore.TransactionId;
import com.endeca.eidi.recordstore.TransactionType;
import com.endeca.eidi.service.ServiceAddress;

/**
 * SampleReader is an example of how to use the Record Store core and client
 * utility classes to read records. It gets the ID of the last-committed
 * generation and reads its records from the Record Store.
 */
public class SampleReader {

    public static void main(String[] args) {
        if (args.length != 2 && args.length != 3) {
            System.out.println("Usage: <ias host> <ias port> [ias context path]");
            System.exit(-1);
        }

        String iasHost = args[0];
        int iasPort = Integer.parseInt(args[1]);
        String contextPath = (args.length == 3) ? args[2] : EidiConstants.DEFAULT_CONTEXT_PATH;

        ServiceAddress address = new ServiceAddress(iasHost, iasPort, contextPath);
        RecordStoreLocator locator = RecordStoreLocator.create(address, "rs1");
        RecordStore recordStore = locator.getService();

        TransactionId transactionId = null;
        try {
            System.out.println("Starting a new transaction ...");
            transactionId = recordStore.startTransaction(TransactionType.READ);

            System.out.println("Getting the last committed generation ...");
            GenerationId gId = recordStore.getLastCommittedGenerationId(transactionId);

            System.out.println("Reading records ...");
            RecordStoreReader reader
= RecordStoreReader.createBaselineReader(recordStore, transactionId,
                    gId);
            int count = 0;
            while (reader.hasNext()) {
                Record record = reader.next();
                System.out.println("  RECORD: " + record);
                count++;
            }
```

```
            reader.close();
            System.out.println(count + " record(s) read");

            System.out.println("Committing transaction ...");
            recordStore.commitTransaction(transactionId);

            System.out.println("DONE");
        } catch (RecordStoreException exception) {
            exception.printStackTrace();
            if (transactionId != null) {
                try {
                    recordStore.rollbackTransaction(transactionId);
                } catch (RecordStoreException anotherException) {
                    System.out.println("Failed to roll back transaction.");
                    anotherException.printStackTrace();
                }
            }
        }
    }
}
```

Index