Endeca® Content Acquisition System

Extension API Guide

Version 3.1.2 • April 2013

ENDECA

Contents

Copyright and disclaimer

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Preface

The Oracle Endeca Commerce solution enables your company to deliver a personalized, consistent customer buying experience across all channels — online, in-store, mobile, or social. Whenever and wherever customers engage with your business, the Oracle Endeca Commerce solution delivers, analyzes, and targets just the right content to just the right customer to encourage clicks and drive business results.

Oracle Endeca Commerce is the most effective way for your customers to dynamically explore your storefront and find relevant and desired items quickly. An industry-leading faceted search and Guided Navigation solution, Oracle Endeca Commerce enables businesses to help guide and influence customers in each step of their search experience. At the core of Oracle Endeca Commerce is the MDEX Engine™, a hybrid search-analytical database specifically designed for high-performance exploration and discovery. The Endeca Content Acquisition System provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. Endeca Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

Oracle Endeca Experience Manager is a single, flexible solution that enables you to create, deliver, and manage content-rich, cross-channel customer experiences. It also enables non-technical business users to deliver targeted, user-centric online experiences in a scalable way — creating always-relevant customer interactions that increase conversion rates and accelerate cross-channel sales. Non-technical users can control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

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

About this guide

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

The guide assumes that you are familiar with Endeca concepts and introductory concepts of the Endeca Content Acquisition System. You can find an introduction to the Content Acquisition System in Chapter 1 of the *CAS Developer's Guide*.

Who should use this guide

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

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

An extension developer creates extensions and packages extensions into one or more plug-ins and hands off the plug-ins to a CAS application developer. The CAS application developer installs the

plug-ins. After installation, the CAS application developer can configure the extensions, and run the extensions as part of acquiring data from a data source.

Conventions used in this guide

This guide uses the following typographical conventions:

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

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

Contacting Oracle Support

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

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

Chapter 1

Introduction to the CAS Extension API

This section introduces the basics of the CAS Extension API.

System requirements

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

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

Overview of the CAS Extension API

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

The CAS Extension API is installed by default as part of the Content Acquisition System.

The components of the CAS Extension API include the following:

- The CAS Extension API, utility, and record access packages (`CAS\`*`version`*`\lib\cas-extension-api`)
- The CAS Extension sample implementations (`CAS\`*`version`*`\sample\cas-extensions`)
- *Endeca CAS Extension API Reference (Javadoc)* (`CAS\`*`version`*`\doc\cas-extension-api-javadoc`)
- *Endeca CAS Extension API Guide* (this guide).

About CAS extensions

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

Data source extensions can access any type of data source that you want to include in the Content Acquisition System. For example, data source extensions might access flat files, databases, content management repositories (that do not already have a corresponding CMS Connector), and so on.

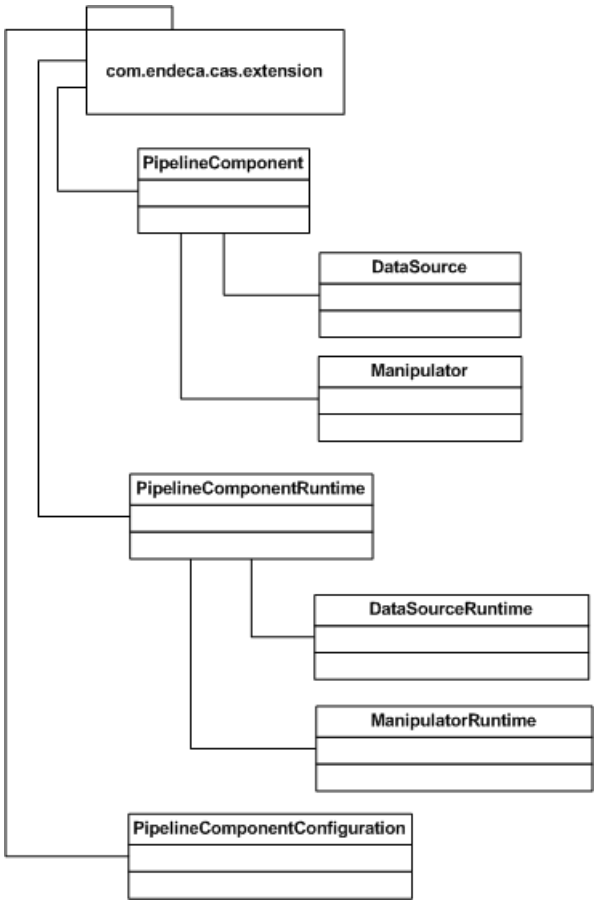
Manipulator extensions transform Endeca records as part of data processing in a CAS acquisition. In a typical usage, manipulators run in a CAS acquisition to provide record pre-processing before a Forge pipeline runs.

About the CAS extension interfaces and classes

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

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

The following diagram shows the inheritance hierarchy of several classes you will work with in the package. The classes in `com.endeca.cas.extension` are declared in the API as abstract classes. Most of the abstract classes have default implementations of their methods. You can often use the default implementation of the methods, or if necessary, you can override the default implementation. The key abstract classes in the package declare abstract methods which you must implement to provide extension functionality.



About CAS annotations

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

Annotations define extensions

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

After a CAS application developer installs a plug-in, the Content Acquisition System scans for extensions by checking for classes that have an annotation of either `@CasDataSource` or `@CasManipulator` and by checking the uniqueness of extension IDs.

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

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

Here is an example annotation that defines a manipulator extension:

```
@CasManipulator(
  supportsIncrementals=true,
```

```
  deleteRecordsBypassManipulator = true,
  displayName="Substring Manipulator",
  description="Generates a new property that is a substring of another
property value")
public class SubstringManipulator extends Manipulator<SubstringManipulator¬
Config>
```

Annotations define configuration properties for an extension

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

CAS Console renders all annotated Java fields as configuration properties on the Data Source tab. When a user specifies values for the fields in CAS Console, and saves the data source, then CAS Console sends the value of the field to CAS Server as a configuration property.

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

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

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

Here is an example annotation of two string fields:

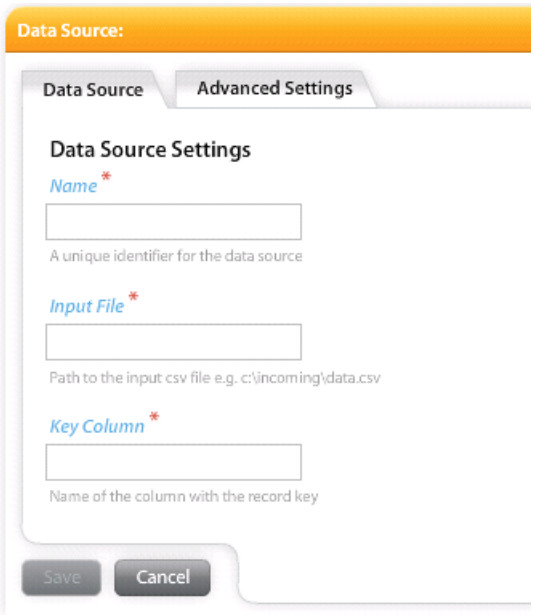
```
@StringProperty(isRequired=true, name="inputFile", displayName="Input File",

 description="Path to the input csv file e.g. c:\\incoming\\data.csv")
 private String mInputFile;

@StringProperty(isRequired=true, name="keyColumn", displayName="Key Column",

 description="Name of the column with the record key")
 private String mKeyColumn;
```

When CAS Console renders the `mInputFile` property and the `mKeyColumn` property, they display as the **Input File** and **Key Column** configuration properties shown here:



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

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

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

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

Annotations specify groups and the order of fields in a group

A group organizes fields for display as configuration properties in CAS Console. You can annotate an extension to organize a set of fields into a group and specify the order of fields in a group.

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

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

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

```
@ConfigurationGroup(groupName="User Credentials", propertyOrder={"user¬
Name","userPassword"})
```

Annotations specify the order of multiple groups of fields

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

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

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

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

This scenario requires the following annotations:

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

Record access interfaces

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

Structure of an Endeca record

Endeca records are made up of property key-value pairs. Each of the key-value pairs are strings. In some cases, a property key can have multiple values. Here is a simple representation of an example record produced by acquiring data from a file system. This representation illustrates the makeup of a record as key-value pairs. For example, the property key named `Endeca.SourceType` has a value of `FILESYSTEM`.

Record example

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

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

The Record class

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

The PropertyValue class

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

Common record properties

The CAS Server generates certain properties whether you acquire data from a file system, a CMS, or custom data source.

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

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

The CAS Server generates the following properties for all records:

Endeca Property Name	Property Value
`Endeca.Action`	The action that was taken with the document. Values are `UPSERT` (the file or folder has been added or modified) or `DELETE` (the file or folder has been deleted since the last acquisition).
`Endeca.Id`	Provides a unique identifier for each record. For data source extensions, an extension developer must add `Endeca.Id` to each record and assign it a value appropriate for the data source.
`Endeca.SourceId`	Indicates the name of the acquisition source.

Chapter 2

Implementing a data source

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

Creating a data source class

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

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

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

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

To create a data source extension:

1. Create a Java project in your development environment of your choice.

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

   ```
   public class CsvDataSource extends DataSource<CsvDataSourceRuntime,Csv¬
   DataSourceConfig>{

   }
   ```
4. Add a `@CasDataSource` annotation to the `DataSource` class.

For example:

```
@CasDataSource(displayName="CSV File", description="Reads comma separated
 files")
public class CsvDataSource extends DataSource<CsvDataSourceRuntime,Csv¬
DataSourceConfig>{

}
```

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

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

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

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

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

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

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

Example of a data source extension

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

Related Links

Life cycle of a data source on page 35
The following actions indicate key points in the life cycle of a data source. The events take place after a data source extension has been implemented, packaged, and installed into the Content Acquisition System.

About threading on page 38
Data sources and manipulators must be thread safe.

Creating a runtime class for a data source on page 21
The `DataSourceRuntime` is the runtime representation of a data source instance. It is created by `DataSource.createDataSourceRuntime()` and exists for the life span of the data source.

Creating a pipeline component configuration class for a data source on page 19

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

Creating a pipeline component configuration class for a data source

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

To create a data source configuration:

1. In the Java project that contains your `DataSource` implementation, create a subclass of `PipelineComponentConfiguration`.
   For example:
   ```
   public class CsvDataSourceConfig extends
      PipelineComponentConfiguration<CsvDataSourceConfig>{

   }
   ```
2. Add each field that you want available as a configuration property in the data source.
   For example:
   ```
   private String mInputFile;

   private String mKeyColumn;
   ```
3. Add an annotation to each field. The annotation type must match the field's data type. See package `com.endeca.cas.extension.annotation` in the *CAS Extension API Reference* (Javadoc) to determine which annotations have required attributes and to determine which attributes are appropriate for the field.
   For example:
   ```
   @StringProperty(isRequired=true, name="inputFile",
      displayName="Input File", description="Path to the input csv file e.g.
    c:\\incoming\\data.csv")
   private String mInputFile;

   @StringProperty(isRequired=true, name="keyColumn",
      displayName="Key Column", description="Name of the column with the
   record key")
   private String mKeyColumn;
   ```
4. If you want to order configuration properties within a single group, add a `@ConfigurationGroupOrder` annotation to the `PipelineComponentConfiguration` class and then add a nested `@ConfigurationGroup` annotation.
   For example, here is one group of fields that display in order — `filePath`, `headerRow`, and `separator`:
   ```
   @ConfigurationGroupOrder({@ConfigurationGroup(groupName="Basic",
      propertyOrder={"filePath","headerRow","separator"})})
   public class CsvDataSourceConfig implements
      PipelineComponentConfiguration<CsvDataSourceConfig>{

   }
   ```

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

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

   You can write code that checks a specific property to determine if a full acquisition is required (rather than the entire `PipelineComponentConfiguration`). If you want to force a full acquisition, write code that always returns `true`.
7. Optionally, override the default implementation of `validate()`. CAS Server performs data type and constraint validation (constraints may include minValue and maxValue for integer properties). Any code you write in `validate()` performs additional custom validation.
   For example:

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

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

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

  return validationFailures;
}
```

   If validation fails, the `PipelineComponentConfiguration.validate()` method returns a collection `ValidationFailure` objects.
8. If it is necessary for unit testing or for the implementation of the data source runtime, you may also need to write getter and setter methods for each user-interface field that you added.

Example of a pipeline component configuration for a data source

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

Related Links

About CAS annotations on page 11

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

Creating a runtime class for a data source

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

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

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

To create a runtime class for a data source:

1. In the Java project that contains the `DataSource` implementation, create a subclass of `DataSourceRuntime`.
   For example:
   ```
   public class CsvDataSourceRuntime extends DataSourceRuntime {

   }
   ```
2. Implement the `DataSourceRuntime` constructor.
3. Implement the abstract method `runFullAcquisition()` to define how to acquire content from the data source. The implementation depends on your custom data source.
4. Within your implementation of `runFullAcquisition()`, call `ErrorChannel.discard()` as necessary to discard any records that are invalid or have errors, and also call `OutputChannel.output()` for each record that has been processed.
5. Optionally, implement either the `BinaryContentFileProvider` interface or the `BinaryContentInputStreamProvider` interface if the data source needs to support text extraction.
6. Optionally, implement the `IncrementalDataSourceRuntime` interface calculate the changes in your data source extension, rather than have the Content Acquisition System determine the changes for you.
7. Optionally, handle requests to stop an acquisition by providing a mechanism to stop an extension's runtime object in a timely way. This may include polling `PipelineComponentRuntimeContext.isStopped()` and may include overriding `PipelineComponentRuntime.stop()`. For guidance, see *Stopping an extension when an acquisition stops* on page 41.
8. Optionally, override `PipelineComponentRuntime.endAcquisition()` to clean up any resources used by `PipelineComponentRuntime`. For guidance, see *Cleaning up resources used by an extension* on page 42.

Example of a data source runtime

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

Related Links

Creating a data source class on page 17
You create a data source by extending the `DataSource` abstract class and other supporting classes.

Supporting filtering in a data source

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

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

- Filter objects, as represented by the `Filter` base class in the CAS Server API, are not currently supported for a data source extension.
- No **Filters** tab is available in CAS Console for a data source extension.
- Document conversion filters specified in `DocumentConversionFilters.xml` do not apply to a data source extension.

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

Supporting document conversion in a data source

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

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

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

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

Enabling document conversion in the data source

An extension developer needs to implement one of the binary content provider interfaces, but not both, to support document conversion. A CAS application developer specifies whether document conversion is enabled by configuring the data source in CAS Console, using the CAS Server API (`TextExtrac¬tionConfig`), or using the CAS Server Command-line Utility.

If document conversion is enabled, a CAS application developer can also specify whether CAS Server should cache the file locally before reading it.

Example code in the CAS extension samples

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

To see an example of how `BinaryContentInputStreamProvider.getBinaryContentInput¬Stream()` is used, see the CAS sample extension in `<install path>\CAS\`*`version`*`\sample\cas-extensions\src\main\com\endeca\cas\extension\sample\datasource\blob\BlobDataSourceRuntime.java`.

Specifying which documents to convert

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

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

There are two approaches for specifing which documents to convert:

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

Full and incremental crawling modes

The CAS Server crawls a data source in one of two modes:

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

Crawling in full mode

Crawling in *full* mode means that CAS processes all the content in a data source according to the filtering criteria you specify. As part of crawling a data source, CAS creates metadata information and

stores it in a crawl history. This history includes the Id of each record and information about all properties on the record.

Crawling in incremental mode

Crawling in *incremental* mode means that CAS processes only that content whose metadata information, stored in the crawl history, has changed since the last crawl. Specifically, CAS checks all properties on the record to see if any have changed. If any properties have changed, the CAS Server crawls the content again. This is true in cases where CAS is calculating the incremental difference. An extension developer, using the CAS Extension API, may choose to calculate incremental changes in a data source extension.

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

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

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

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

After you click **Start** in CAS Console, you can click the link under **Acquisition Status** to see a status message indicating whether a full or incremental crawl is running. After you crawl a data source using the API, the status message is returned.

Incremental mode and MDEX compatible output

An incremental crawl processes only data records. It does not process any configuration stored in the Endeca Configuration Repository (such as dimensions and properties, precedence rules, and so on), and it does not crawl dimension value records. By contrast, a full crawl processes data records, configuration in the Endeca Configuration Repository, and dimension value records.

Related Links

Introduction to CAS and Crawling Data Sources This part contains the following sections:

Supporting incremental acquisition in a data source

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

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

The Content Acquisition System determines the incremental difference

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

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

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

An extension developer determines the incremental difference

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

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

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

- Implement `IncrementalDataSourceRuntime.checkFullAcquisitionRequired()`.

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

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

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

- Implement `IncrementalDataSourceRuntime.runIncrementalAcquisition()`.

Example code in the CAS extension samples

To see an example of how `IncrementalDataSourceRuntime` is used, see the CAS sample extension in `<install path>\CAS\`*`version`*`\sample\cas-extensions\src\main\com\endeca\cas\extension\sample\datasource\incremental`.

Related Links

Storing state information for an extension on page 42

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

Chapter 3

Implementing a manipulator

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

Creating a manipulator class

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

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

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

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

To create a manipulator extension:

1. Create a Java project in your development environment of your choice.

   If you are creating several extensions in one plug-in, you can use the same Java project for each extension.

2. Add the CAS Extension API libraries to your compile classpath. These include all the libraries available in `CAS\`*`version`*`\lib\cas-extension-api`.
3. Create a subclass of `Manipulator` and specify the `PipelineComponentConfiguration` subclass that the extension uses.

For example:

```
public class SubstringManipulator extends Manipulator<SubstringManipula¬
torConfig>{

}
```

4. Add a `@CasManipulator` annotation to the `Manipulator` class and any attributes as described above.
   For example:

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

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

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

6. Implement the `createManipulatorRuntime()` method to create an implementation of the `ManipulatorRuntime` class.
   For example:

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

7. Implement the `getRuntimeClass()` method to return the runtime class the manipulator creates.
   For example:

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

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

Example of a manipulator extension

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

Related Links

Creating a pipeline component configuration class for a manipulator on page 29

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

Creating a runtime class for a manipulator on page 31
The `ManipulatorRuntime` is the runtime representation of a manipulator instance. The `ManipulatorRuntime` is created by `Manipulator.createManipulatorRuntime()` and exists for the life span of the manipulator.

Creating a pipeline component configuration class for a manipulator

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

To create a manipulator configuration class:

1. In the Java project that contains the `Manipulator` implementation, create a subclass of `PipelineComponentConfiguration`.
   For example:
   ```
   public class SubstringManipulatorConfig extends PipelineComponentConfig¬
   uration<SubstringManipulatorConfig> {

   }
   ```
2. Add each field that you want available as a configuration property in the manipulator.
   For example:
   ```
   private String mSourceProperty;

   private String mTargetProperty;

   private int mStartIndex;
   ```
3. Add an annotation to each field. The annotation type must match the field's data type. See package `com.endeca.cas.extension.annotation` in the *CAS Extension API Reference* (Javadoc) to determine which annotations have required attributes and to determine which attributes are appropriate for the field.
   For example:
   ```
   @StringProperty(isRequired=true, name="sourceProperty", displayName="Source
    Property")
    private String mSourceProperty;

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

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

4. If you want to order fields within a single group, add a `@ConfigurationGroupOrder` annotation to the `PipelineComponentConfiguration` class and then add a nested `@ConfigurationGroup` annotation.
   For example, here is one group of fields that display in order — `sourceProperty`, `targetProperty`, `length` and `startIndex`:

```
@ConfigurationGroupOrder({
 @ConfigurationGroup(propertyOrder={"sourceProperty", "targetProperty",
 "length", "startIndex"})
})
public class SubstringManipulatorConfig extends PipelineComponentConfiguration<SubstringManipulatorConfig> {

}
```

5. If you want to order multiple groups of user-interface fields on a tab, add additional `@ConfigurationGroup` annotations within `@ConfigurationGroupOrder` for each group of user-interface fields that you want ordered.
6. Optionally, override the default implementation of `isFullAcquisitionRequired()`. The default implementation determines whether a configuration change should force full acquisition the next time an acquisition is run by comparing the old `PipelineComponentConfiguration` and the new `PipelineComponentConfiguration` using the `equals()` method. The default implementation of the `equals()` method uses reflection to compare all non-transient fields for equality.
   You can write code that checks a specific property to determine if a full acquisition is required (rather than check the entire `PipelineComponentConfiguration`). If you want to force a full acquisition, write code that always returns `true`.
7. Optionally, override the default implementation of `validate()`. CAS Server performs data type and constraint validation (constraints may include minValue and maxValue for integer properties). Any code you write in `validate()` performs additional custom validation.
   If validation fails, the `PipelineComponentConfiguration.validate()` method returns a collection `ValidationFailure` objects.
8. If it is necessary for unit testing or the implementation of the manipulator runtime, you may also need to write getter and setter methods for each field that you added.

Example of a pipeline component configuration for a manipulator

To see many of the steps above, refer to the sample data source extension in `<install path>\CAS\`*`version`*`\sample\cas-extensions\src\main\com\endeca\cas\extension\sample\source\csv\SubstringManipulatorConfig.java`.

Related Links

About CAS annotations on page 11
The CAS Extension API provides Java annotations that define extensions to the Content Acquisition System and also define configuration properties for an extension. Annotations can describe whether configuration properties are required or optional, whether they have a default value, and their display properties in CAS Console.

Creating a runtime class for a manipulator

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

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

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

To create a runtime class for a manipulator:

1. In the Java project that contains the `Manipulator` implementation, create a subclass of `ManipulatorRuntime`.
   For example:
   ```
   public class SubstringManipulatorRuntime extends ManipulatorRuntime {

   }
   ```
2. Implement the `ManipulatorRuntime` constructor.
3. Optionally, override the default implementation of `ManipulatorRuntime.checkFullAcquisitionRequired()` to allow each manipulator in an acquisition to indicate whether it requires a full acquisition. This check could be necessary if a manipulator has state-based dependencies that should force a full acquisition.
4. Optionally, override the default implementation of `prepareForAcquisition(AcquisitionMode)` if the manipulator has to prepare state to process records that result from an incremental acquisition. CAS Server passes in an acquisition mode of either `FULL_ACQUISITION` or `INCREMENTAL_ACQUISITION` based on the results of running `checkFullAcquisitionRequired()`.
5. Implement the abstract method `processRecord()` to define how to manipulate records. The implementation depends the manipulation you wish to perform.
6. Optionally, call `ErrorChannel.discard()` as necessary to discard any records that are invalid or have errors.
7. Call `OutputChannel.output()` for each record that has been processed by `processRecord()`.
   For example:
   ```
   getContext().getOutputChannel().output(record);
   ```
   A manipulator should not modify any records that have already been output by `output()`. If you are doing significant processing between calls to `output()`, you may want to periodically call `PipelineComponentRuntimeContext.isStopped()` to see if any requests to stop the acquisition have been made while `OutputChannel.output()` is running.
8. Optionally, implement `onInputClose()` to perform any cleanup or post-processing after `processRecord()` finishes processing the last record.
9. Optionally, handle requests to stop an acquisition by providing a mechanism to stop an extension's runtime object in a timely way. This may include polling `PipelineComponentRuntimeContext.isStopped()` and may include overriding `PipelineComponentRuntime.stop()`.

10. Optionally, override `PipelineComponentRuntime.endAcquisition()` to clean up any resources used by `PipelineComponentRuntime` or `ManipulatorRuntime` and also clean up any state-based dependencies.

Example of a manipulator runtime

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

Related Links

Stopping an extension when an acquisition stops on page 41
When an acquisition stops, it may take time for an extension within an acquisition to stop. Therefore, Oracle recommends that you provide a mechanism to stop an extension's runtime object in a more timely way (recall that a runtime object is either a `DataSourceRuntime` or a `ManipulatorRuntime`).

Cleaning up resources used by an extension on page 42
The CAS Server calls `endAcquisition()` after all extensions in a acquisition have completed data acquisition and record processing. Your implementation of `endAcquisition()` can override `PipelineComponentRuntime.endAcquisition()` to perform any necessary cleanup for an extension.

Supporting incremental acquisition in a manipulator

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

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

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

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

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

Generating correct output

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

- Does the manipulator output only the records it receives? That is, does it always output records with a `recordId` that was input to it?

- Does the manipulator output all the records that it receives?
- Does the manipulator generate output for one record that depends only on the input of the same record and not on any other records?
- Does the manipulator produce the same output over time in response to a given input record?

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

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

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

For example, suppose a manipulator filters records to include only those records that contain a color property that is set to red. Record A has the property color set to red, so the manipulator includes record A. In a subsequent incremental acquisition, record A has changed so that now the color property is set to blue. Record A no longer meets the manipulator's filtering criteria. Instead of dropping the record, the manipulator emits a delete record for record A.

Delete records can bypass a manipulator

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

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

Related Links

About CAS annotations on page 11

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

Creating a runtime class for a manipulator on page 31

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

Chapter 4

Extension life cycle and threading

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

Life cycle of a data source

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

Action	Corresponding life cycle event
The Endeca CAS Service starts.	CAS Server instantiates any `DataSource` that is installed and correctly annotated.
The following items perform the same action. • In the CAS Console - Configuring the data source and clicking **Save**. • In the CAS Server API - Calling either `CasCrawler.createCrawl()` or `CasCrawler.updateCrawl()`. • In the cas-cmd utility - Running either `createCrawls` or `updateCrawls`.	CAS Server instantiates the `PipelineComponent¬Configuration` specified by the `DataSource`. CAS Server performs data type and constraint validation and also calls `PipelineComponent¬Configuration.validate()` to execute any additional custom validation code you wrote in the method's implementation.
The following items perform the same action. • In the CAS Console - Clicking **Start** in the **Acquire Data** column of the Data Sources page. • In the CAS Server API - Calling `CasCrawler.startCrawl()`. • In the cas-cmd utility - Running `cas-cmd startCrawl`.	CAS Server re-creates the `PipelineComponent¬Configuration` specified by the `DataSource` and re-validates the `PipelineComponentCon¬figuration` by calling `validate()`. (CAS Server injects configuration property values via reflection before calling `validate()`.) CAS Server calls `createDataSourceRun¬time()` on the `DataSource` instance. The resulting `DataSourceRuntime` exists for the duration of the acquisition. If an incremental acquisition is requested, CAS Server calls `checkFullAcquisitionRe¬`

Action	Corresponding life cycle event
	quired() on the new DataSourceRuntime. The check returns a Boolean value to indicate whether a full acquisition is required or not. If any data source or manipulator in the CAS pipeline returns true, CAS Server calls DataSourceRuntime.runFullAcquisition(). If all extensions return false, CAS Server calls IncrementalDataSourceRuntime.runIncrementalAcquisition(). After all records have been processed by all extensions, CAS Server calls PipelineComponentRuntime.endAcquisition(). CAS Server passes in an AcquisitionEndState value to endAcquisition() to indicate whether the extension succeeded, failed, or requires a full acquisition to recover from a failure.
The following items perform the same action. • In the CAS Console - Clicking the delete icon for a data source on the Data Sources page. • In the CAS Server API - Calling CasCrawler.deleteCrawl(). • In the cas-cmd utility - Running cas-cmd deleteCrawl.	CAS Server calls PipelineComponent.deleteInstance() when a data source is removed from an application as part of updating or deleting an acquisition, and deleteInstance() is also called if the data source type of the acquisition has changed. (The data source type is represented by a ModuleId setting in the CAS Server API).

Life cycle of a manipulator

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

Action	Corresponding life cycle event
The Endeca CAS Service starts.	CAS Server instantiates any Manipulator that is installed and correctly annotated.
The following items perform the same action. • In the CAS Console - Configuring the data source and clicking **Save**. • In the CAS Server API - Calling either CasCrawler.createCrawl() or CasCrawler.updateCrawl(). • In the cas-cmd utility - Running either createCrawls or updateCrawls.	CAS Server instantiates a PipelineComponentConfiguration specified by the Manipulator for each Manipulator occurrence in a given acquisition. CAS Server performs data type and constraint validation and also calls PipelineComponentConfiguration.validate() to execute any additional custom validation code you wrote in the method's implementation.

Action	Corresponding life cycle event
The following items perform the same action. • In the CAS Console - Clicking **Start** in the **Acquire Data** column (of the Data Sources page). • In the CAS Server API - Calling `CasCrawler.startCrawl()`. • In the cas-cmd utility - Running `startCrawl`.	CAS Server re-creates the `PipelineComponentConfiguration` and re-validates the `PipelineComponentConfiguration` by calling `validate()`. (CAS Server injects configuration property values via reflection before calling `validate()`.) CAS Server calls `createManipulatorRuntime()` on the `Manipulator` instance. The `ManipulatorRuntime` exists for the duration of the acquisition. If an incremental acquisition is requested, CAS Server calls `checkFullAcquisitionRequired()` on the new `ManipulatorRuntime`. The check returns a Boolean value to indicate whether a full acquisition is required or not before record processing. If any one data source or manipulator in a CAS pipeline returns `true`, CAS runs a full acquisition. All extensions in a CAS pipeline must return `false` for CAS to run an incremental acquisition. Before processing records, CAS Server calls `prepareForAcquisition()` to prepare state. During the acquisition, CAS Server calls `ManipulatorRuntime.processRecord()` concurrently for separate records to manipulate every input record. After all records have been processed by all extensions, CAS Server calls `ManipulatorRuntime.onInputClose()` and then calls `PipelineComponentRuntime.endAcquisition()` on each manipulator. CAS Server passes in an `AcquisitionEndState` value to `endAcquisition()` to indicate whether the extension succeeded, failed, or requires a full acquisition to recover from a failure.
The following items perform the same action. • In the CAS Server API - Calling `CasCrawler.deleteCrawl()` or • In the cas-cmd utility - Running `deleteCrawl`.	CAS Server calls `PipelineComponent.deleteInstance()` when a manipulator is removed from an acquisition.

About threading

Data sources and manipulators must be thread safe.

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

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

Recommendations for data sources

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

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

Recommendations for manipulators

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

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

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

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

Configuration and context synchronization

As part of the implementation of an extension, the CAS Server passes in a `PipelineComponentCon¬figuration` object and a `PipelineComponentRuntimeContext` object to either `DataSource.cre¬ateDataSourceRuntime()` (in the case of data sources) and `Manipulator.createManipula¬torRuntime()` (in the case of manipulators). The CAS Server does not modify the `PipelineCom¬ponentConfiguration` after `createManipulatorRuntime()` or `createDataSourceRuntime()` has been called.

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

Related Links

Creating a runtime class for a data source on page 21

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

Creating a runtime class for a manipulator on page 31

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

Chapter 5

Common implementation tasks

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

Stopping an extension when an acquisition stops

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

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

- A CAS application developer requests a stop by running the `stopCrawl` task of the CAS Server Command-line Utility (`cas-cmd`).
- A CAS application developer requests a stop by calling the `stopCrawl()` method of the CAS Server API.
- A CAS Console user requests a stop by clicking **Abort** for the data source.
- An acquisition may abort because it encounted a fatal error during the acquisition processing or record manipulation processing.

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

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

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

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

Cleaning up resources used by an extension

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

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

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

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

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

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

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

Storing state information for an extension

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

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

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

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

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

Related Links

Supporting incremental acquisition in a data source on page 24

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

Exceptions that trigger fatal and non-fatal failures

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

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

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

Enabling logging in an extension

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

Supported logging frameworks

The Content Acquisition System supports the following logging frameworks:

- SLF4J
- Apache Commons Logging
- java.util.logging
- Log4J

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

Integrating logging

Integration is largely transparent. You import the logging framework into the `PipelineComponentRuntime` and call `getLogger()`. If any logging requests come in from any of the frameworks, the Content Acquisition System detects the requests and redirects them to Log4J which CAS then uses to write to `cas-service.log`.

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

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

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

Changing log levels

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

For example, this Log4J entry sets the log level to DEBUG for the sample substring manipulator.

```
log4j.logger.com.endeca.cas.extension.sample.manipulator.substring.Substring¬
ManipulatorRuntime=DEBUG
```

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

The Log level setting in the data source's configuration

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

Unit testing an extension

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

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

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

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

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

About testing the `PipelineComponentConfiguration`

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

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

About testing the `DataSourceRuntime`

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

- Constructing a `DataSourceRuntime` with a mock `PipelineComponentRuntimeContext` that contains a mock `OutputChannel`, a mock `ErrorChannel`, and a mock `PipelineComponent¬Context`.

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

About testing the `ManipulatorRuntime`

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

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

Packaging an extension into a plug-in

A plug-in is a JAR or set of JAR files that contain an extension or set of extensions. After implementing one or more extensions, you package them into one or more JAR files and distribute them, and any dependent JAR files, to a CAS application developer.

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

To package an extension into a plug-in:

1. In your Java project, build the classes for an extension into a JAR or a set of JARs.
   For an example, see the procedure in Building the sample extensions.
2. Distribute the JARs, and any dependent JARs, to the CAS application developer who installs the plug-in into the Content Acquisition System. For installation instructions, see the *CAS Installation Guide*.

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

- `cas-extension-api.jar`
- `itl-api-common.jar`
- `commons-logging.jar`
- `log4j.jar`

- `slf4j-api.jar`

Running an extension

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

Before starting this procedure, you must have already packaged the extension into a plug-in and installed the plug-in into the Content Acquisition System. For installation instructions, see the *CAS Installation Guide*.

To run an extension:

1. On the machine where the Content Acquisition System is installed, confirm that the plug-in is installed by running the `listModules` task of `cas-cmd`.
   - If you installed a plug-in with a data source, specify a `moduleType` of `SOURCE` to the `listModules` task.
   - If you installed a plug-in with a manipulator, specify a `moduleType` of `MANIPULATOR` to the `listModules` task.

   For example:

   ```
   C:\Endeca\CAS\3.1.2\bin>cas-cmd listModules -t SOURCE
   Blob Database
    *Id: com.endeca.cas.extension.sample.datasource.blob.BlobDataSource
    *Type: SOURCE
    *Description: Reads a database table containing documents
   ```

2. Configure the data source in CAS Console as follows:
   a) Log in to Endeca Workbench and select the **Data Sources** page.
   b) On the **Data Sources** page, click **Add Data Source** and select the data source.
   c) Specify configuration properties as appropriate for the data source.
   d) Click **Save**.
3. To configure a manipulator, add it to a data source, and configure it as follows:
   a) Log in to Endeca Workbench and select the **Data Sources** page.
   b) On the **Data Sources** page, click a data source name to access its acquisition steps.
   c) Click **Add Manipulator...**
   d) Select a manipulator and click **Add**.
   e) Specify configuration properties as appropriate for the data source.
   f) Click **Save**.
4. On the **Data Sources** page, run the extension by clicking **Start** for the data source.
5. When the **Acquisition Status** reads **Completed**, click on this status to verify that the CAS Server created Endeca records.
6. If desired, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`.
   For example:

   ```
   C:\Endeca\CAS\3.1.2\bin>recordstore-cmd read-baseline -a Test
   ```

Appendix A

Sample extensions

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

About the sample extensions

There are sample extensions that illustrate how to build, test, install, and run extensions.

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

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

Sample extensions files and directories

The sample extensions have the following directory structure:

```
\sample
  \cas-extensions
    \build
    \data
    \lib
    \src
```

```
    .classpath
    .project
    build.xml
```

The contents are as follows:

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

Building the sample extensions

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

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

Building the sample extensions with Ant

The Ant build file has the following targets:

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

To build the sample extensions with Ant:

1. Open a command prompt and navigate to the `\CAS\`*`version`*`\sample\cas-extensions` directory.
2. Issue the following command:

   ```
   ant dist
   ```

   For example:

   ```
   C:\Endeca\CAS\3.1.2\sample\cas-extensions>ant dist
   ```

The Ant build file runs all the targets listed above including the unit tests for the extensions. The output messages of the CAS sample extension indicate that the build file compiles the source and then runs unit tests for the sample extensions. It also instructs you to `Copy the folder build/cas-sample-extensions to $CAS_ ROOT/lib/cas-server-plugins to deploy the plugin.`

Building the sample extensions with Eclipse

To build the sample extensions with Eclipse:

1. Start Eclipse and select **File > Import... > General > Existing Projects into Workspace > Next**.
2. Browse to the project in `CAS\version\sample\cas-extensions` and click **OK > Finish**.
3. In the Project Explorer, right-click on the `cas-sample-extensions` directory and select **Export**.
4. Select **Java > JAR file** and click **Next**.
5. In the **JAR File Specification** dialog:

   Select the resources to export (the `cas-sample-extensions` package).

   If it is not already checked, select **Export generated class files and resources**.

   Select an export destination.

   Select **Compress the contents of the JAR file**.

   Click **Next** when you are satisfied with the specification.
6. In the **JAR Packaging Options** dialog, select any options that you want and click **Finish**.

Building the sample extensions using Eclipse to run Ant

To build the sample extensions using Eclipse to run Ant:

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

The Ant build file runs all the targets listed above including the unit tests for the extensions. The output messages of the CAS sample extension indicate that the build file compiles the source and then runs unit tests for the sample extensions. It also instructs you to `Copy the folder build/cas-sample-extensions to $CAS_ ROOT/lib/cas-server-plugins to deploy the plugin.`

Unit testing the sample extensions

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

Unit testing the sample extensions with Ant

To unit test the sample extensions with Ant:

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

```
ant test-run
```

Unit testing the sample extensions with Eclipse

To unit test the sample extensions with Eclipse:

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

Unit testing the sample extensions using Eclipse to run Ant

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

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

Installing the sample plug-in into CAS

After you build the sample extensions into a plug-in, install the resulting plug-in files into the Content Acquisition System.

To install the sample plug-in into CAS:

1. Stop the Endeca CAS Service.
2. If you built the extensions using Ant, copy the `cas-sample-extensions` directory from `CAS\`*`version`*`\sample\cas-extensions\build\` to `CAS\`*`version`*`\lib\cas-server-plugins`.
3. If you built the extensions using Eclipse, do the following:
   a) Create a new folder named `cas-sample-extensions` under `CAS\`*`version`*`\lib\cas-server-plugins\`.
   b) Locate the JAR you built and copy it to `CAS\`*`version`*`\lib\cas-server-plugins\cas-sample-extensions\.`
   c) Copy the JARs from `\CAS\`*`version`*`\sample\cas-extensions\lib\main` into `CAS\`*`version`*`\lib\cas-server-plugins\cas-sample-extensions\`. (These JARs are dependencies that the sample extensions require.)
4. Start the Endeca CAS Service.

You can confirm that the extensions are installed by running the `listModules` task of the CAS Server Command-line Utility. The task returns the installed modules.

```
C:\Endeca\CAS\3.1.2\bin>cas-cmd listModules
Blob Database
 *Id: com.endeca.cas.extension.sample.datasource.blob.BlobDataSource
 *Type: SOURCE
 *Description: Reads a database table containing documents
 *Capabilities:
   *Has Binary Content

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

```
Document Directory
 *Id: com.endeca.cas.extension.sample.datasource.directory.DirectoryData¬
Source
 *Type: SOURCE
 *Description: Reads a directory of documents
 *Capabilities:
   *Binary Content Accessible via FileSystem
   *Has Binary Content

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

File System
 *Id: File System
 *Type: SOURCE
 *Description: No description available for File System
 *Capabilities:
   *Binary Content Accessible via FileSystem
   *Data Source Filter
   *Has Binary Content
   *Expand Archives

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

Running the sample CSV data source

After you install the extensions into the Content Acquisition System, you can configure and then run the sample CSV data source in CAS Console.

To run the sample CSV data source:

1. Log in to Endeca Workbench and select the **Data Sources** page.
2. On the **Data Sources** page, click **Add Data Source** and select **CSV File**.
3. In **Name**, specify a unique name for the data source to distinguish it from others in the CAS Console.
4. In **Input File**, specify the path to the `data.csv` file. For example, `C:\Endeca\CAS\`*`version`*`\sample\cas-extensions\data\data.csv`
5. In **Key Column**, specify the name of the column containing the record key: `P_WineID`
6. Click **Save**.
7. Click **Start** to start acquiring data from this data source.

8. When the **Acquisition Status** reads **Completed**, click on this status to verify that the CAS Server created 50 Endeca records and failed on 2 records (these are intentionally discarded records).
9. If desired, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`.
   For example:
   ```
   C:\Endeca\CAS\3.1.2\bin>recordstore-cmd read-baseline -a TestCSV
   ```

Running the sample Substring manipulator

After you install the extensions into the Content Acquisition System, you can add, configure, and run the sample Substring manipulator in CAS Console.

Recall that the Substring manipulator examines a source property and creates a target property based on a substring of the source.

To run the sample Substring manipulator:

1. Create a crawl that uses the sample CSV data source. (You will add the sample Substring manipulator to the sample CSV data source and run the crawl.)
2. On the **Data Sources** page, click the name of the data source that contains the sample CSV data source.
3. Click **Add Manipulator**.
4. Select the Substring Manipulator and click **Ok**.
5. Configure the manipulator to examine the `P_Wine` property (wine name) and create a five character `P_Wine_short` property. This configuration requires the following sub-steps:
   - Specify a unique value for **Manipulator Id**, for example `Compute_P_Wine_short`.
   - Specify a **Source Property** of `P_Wine`.
   - Specify a **Target Property** of `P_Wine_short`.
   - Specify a **Substring Length** of `5`.
6. Click **Save**.
7. Click **Start** to acquire data from this data source.
8. If desired, you can confirm that the new target properties exist by running the `read-baseline` task of `recordstore-cmd` and examining the records for the new properties.
   For example:
   ```
   C:\Endeca\CAS\3.1.2\bin>recordstore-cmd read-baseline -a TestCSV
   ```

Running the sample Blob database data source

After you install the extensions into the Content Acquisition System, you can configure and then run the sample Blob database data source in CAS Console.

This data source illustrates an implementation of the `BinaryContentInputStreamProvider` interface. This interface provides an input stream so CAS Server can read the input stream and perform text extraction.

To run the sample Blob database data source:

1. Log in to Endeca Workbench and select the **Data Sources** page.
2. On the **Data Sources** page, click **Add Data Source** and select **Blob Database**.
3. In **Name**, specify a unique name for the data source to distinguish it from others in the CAS Console.
4. In **Database Directory**, specify the path to the database: `<install path>\CAS\version\sample\cas-extensions\data\document-db`
5. Click **Save**.
6. Click **Start** to start acquiring data from this data source.
7. When the **Acquisition Status** reads **Completed**, click on this status to verify that the CAS Server created 5 Endeca records.
8. If desired, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`.
   For example:
   ```
   C:\Endeca\CAS\3.1.2\bin>recordstore-cmd read-baseline -a DBExample
   ```

Running the sample Document Directory data source

After you install the extensions into the Content Acquisition System, you can configure and then run the sample Document Directory data source in CAS Console.

This data source illustrates an implementation of the `BinaryContentFileProvider` interface. This interface allows the extension to pass a file to CAS Server and perform text extraction.

To run the sample Document Directory data source:

1. Log in to Endeca Workbench and select the **Data Sources** page.
2. On the **Data Sources** page, click **Add Data Source** and select **Document Directory**.
3. In **Name**, specify a unique name for the data source to distinguish it from others in the CAS Console.
4. In **Document Directory**, specify the path to the documents: `CAS\version\sample\cas-extensions\data\documents`
5. Click **Save**.
6. Click **Start** to start acquiring data from this data source.
7. When the **Acquisition Status** reads **Completed**, click on this status to verify that the CAS Server created 5 Endeca records.
8. If desired, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`.
   For example:
   ```
   C:\Endeca\CAS\3.1.2\bin>recordstore-cmd read-baseline -a DocExample
   ```

Running the sample Change Tracking data source

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

This data source illustrates an implementation of the `IncrementalDataSourceRuntime` interface. This interface provides support to check whether a full acquisition is required from the Change Tracking

data source. If a full acquisition is not required, then the data source provides an implementation of `runIncrementalAcquisition()` to acquire only the changed records.

To run the sample Change Tracking data source:

1. Log in to Endeca Workbench and select the **Data Sources** page.
2. On the **Data Sources** page, click **Add Data Source** and select **Change Tracking Data Source**.
3. In **Name**, specify a unique name for the data source to distinguish it from others in the CAS Console.
4. In **Database File Path**, specify the absolute path to the Change Tracking database in `<install path>\CAS\version\sample\cas-extensions\data\change-tracking-db.xml`.
5. Click **Save**.
6. Click **Start** to start acquiring data.
7. When the **Acquisition Status** reads **Completed**, click on this status to verify that the CAS Server created 3 Endeca records.
8. If desired, you can confirm that the new records exist in the Record Store instance by running the `read-baseline` task of `recordstore-cmd`.
   For example:

```
C:\Endeca\CAS\3.1.2\bin>recordstore-cmd read-baseline -a CTTest
[DATA=base line data..., Endeca.Id=1, Endeca.Action=UPSERT, Endeca.Sour¬
ceId=CTTest]
[DATA=some incremental data..., Endeca.Id=3, Endeca.Action=UPSERT, Ende¬
ca.Source
Id=CTTest]
[DATA=some incremental data..., Endeca.Id=5, Endeca.Action=UPSERT, Ende¬
ca.Source
Id=CTTest]
```

9. Navigate to `<install path>\CAS\version\sample\cas-extensions\data` and open `change-tracking-db.xml` in a text editor.
10. Update one record in the `change-tracking-db.xml` file by doing the following:
    a) Add a new `<changeHistory>` entry to the file as shown in the example below.
    b) Ensure that the `<key>` value corresponds to an existing `<row>` entry in the `<database>`.
    c) Modify the `<time>` value to indicate a time after the acquisition in step 7 and before the current time. (The time is expressed in UTC format. See `http://www.w3.org/TR/NOTE-datetime` for guidance about the syntax.)

    For example:

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

    Acquiring data from this file results in an incremental update to record 5.
11. Add one record in the `change-tracking-db.xml` file by doing the following:
    a) Add a `<row>` entry to the file and ensure the `<key>` value is unique, as shown:

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

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

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

c) Ensure that the <key> value corresponds to a <row> entry in the <database>.
d) Ensure that the <changeType> value is set to CREATE.
e) Modify the <time> value to indicate a time after the acquisition in step 7 and before the current time. (The time is expressed in UTC format. See http://www.w3.org/TR/NOTE-datetime for guidance about the syntax.)

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

12. Delete one record in the change-tracking-db.xml file by doing the following:

a) Add a <changeHistory> entry for the <row> that has been removed as shown:

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

b) Ensure that the <key> value corresponds to a <row> that does not exist in the <database>.
c) Ensure that the <changeType> value is set to DELETE.
d) Modify the <time> value to indicate a time after the acquisition in step 7 and before the current time. (The time is expressed in UTC format. See http://www.w3.org/TR/NOTE-datetime for guidance about the syntax.)

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

13. Save and close change-tracking-db.xml.
14. In CAS Console, run an incremental acquisition by clicking **Start**.
15. When the **Acquisition Status** reads **Completed**, click on this status to verify that the CAS Server update, added, and deleted the records you modified.