NETVIBES | Exalead CloudView

CloudView CV23

CloudView Programmer

Table of Contents

Programmer

This guide explains how to develop, deploy, and configure Exalead CloudView custom Java or .NET features.

Audience

This guide is mainly destined to software programmers or users with a few programming skills.

Further Reading

You might need to refer to the following guides:

Guide	for more details on
Connector Programmer	connector customization.
Mashup Programmer	Mashup UI customization.
javadoc	Java methods and classes. Available in the Exalead CloudView Public APIs Java SDK

What's New?

There are no enhancements in this release.

The Exalead CloudView APIs

This chapter describes the Exalead CloudView public APIs and the API Client Factory

About Exalead CloudView APIs

What Are the Exalead CloudView APIs?

Exalead CloudView provides the following public APIs to allow integration with third-party applications.

Push API (PAPI) is the public API that allows data from any source to be indexed by Exalead CloudView. It supports all operations required to develop new connectors, both managed and unmanaged.

Search API is the public API for developing third-party search applications. It is the entry point for performing searches on Exalead CloudView.

Mashup API is the API which retrieves the contents of data sources to make them accessible to the data feeds.

Note: Mashup Builder Premium also uses other APIs for non-Exalead CloudView feeds (for example, FlickR search).

Management API (MAMI) is the public API used to configure and manage the Exalead CloudView processes.

Use the Java SDK

Java Clients SDK contains all required material to develop external applications interacting with Exalead CloudView. This SDK is under the `<INSTALLDIR>/sdk/java-clients` directory.

It includes the following content:

- `/docs` - API documentation (javadoc); available also online at CloudView Public APIs Java SDK
- `/lib` - the jars to use
- `/samples` - sample code

Use the .NET SDK

.NET Clients SDK contains all required material to index and search documents in Exalead CloudView from an external .NET program. This SDK is under the `<INSTALLDIR>/sdk/dotnet-clients` directory.

It includes the following content:

- `/docs` - API documentation
- `/lib` - the library including the dll files
- `/samples` - sample code for the Search and Push APIs

Raw Access to APIs

The default endpoints for each public API:

- **Push API** - `http://<HOSTNAME>:<BASEPORT+2>`
- **Search API** - `http://<HOSTNAME>:<BASEPORT+10>/search-api`
- **Mashup API** - `http://<HOSTNAME>:<BASEPORT+10>/access`
- **Management API (MAMI)** - `http://<HOSTNAME>:<BASEPORT+11>`

The Push API (PAPI)

The Push API is the public document API that allows Exalead CloudView to index data from any source. It supports the basic operations required to develop new connectors, both managed and unmanaged.

- A 'managed connector' is a piece of code running within Exalead CloudView. You must package it as a Exalead CloudView Plugin (CVPlugin) to deploy and configure it in Exalead CloudView.

  You can develop it in Java, using the Connectors Framework API available in:

  - `<INSTALLDIR>\sdk\java-customcode` for V6R2014 and higher versions.
  - `<INSTALLDIR>\sdk\cloudview-sdk-java-connectors` in previous versions.

- An 'unmanaged connector' is an external component that sends data to Exalead CloudView using the Push API. You can develop an unmanaged connector in any language, either by using Exalead CloudView Push API clients (available in Java, C#, and PHP), or by targeting the HTTP API directly.

  You must manage and deploy unmanaged connectors yourself, as Exalead CloudView is not aware of these connectors.

All standard Exalead CloudView connectors are managed connectors. For more information, see "Introduction" in the Exalead CloudView Connectors Guide.

For more information and best-practices on developing a custom connector, see the *Exalead CloudView Connector Programmer's Guide* .

The Search API

The Search API is the entry point for performing searches on Exalead CloudView. It provides a public HTTP interface to access the commands defined in Exalead CloudView's Search API configuration.

Accessing the Search API

To access the Search API, you can use the Java client (in`<INSTALLDIR>/sdk/java-clients`), the .NET client (in `<INSTALLDIR>/sdk/dotnet-clients`), or directly target the HTTP API using HTTP search commands.

Commands

By default, when you install a Exalead CloudView product, the commands are available at `http://<HOSTNAME>:<BASEPORT+10>`.

For example, a search command called `search-api` is installed, and is available at `http://<HOSTNAME>:<BASEPORT+10>/search-api`.

The main types of Search API commands are:

- **Search**: performs searches on the Exalead CloudView index
- **Query expansion**: provides query expansion on queries

- **Spell check**: runs spell checking requests on queries
- **Suggest**: provides the suggest service on queries
- **Dictionary**: retrieves term, ngrams, phonetized forms, query approximation & regexp frequencies on a dictionary
- **Document fetch**: retrieves documents from the connectors
- **Preview**: retrieves rich HTML preview of the documents
- **Thumbnail**: retrieves image thumbnails of the documents
- **Security**: authenticates users and get tokens against security sources

Search API Command Summary

The commands are available at: `http://<SEARCH_API_HOST>:<SEARCH_API_PORT>/<COMMAND_PATH>`

Command	<COMMAND_PATH>	Java client class	.NET client class
Search	`/search-api/search`	`SearchClient`	`SearchClient`
Query expansion	`/expansion` It supports all Search Command parameters.	n/a	n/a
Spell check	`/spellcheck`	`SpellCheckClie`	n/a
Suggest	`/suggest/service/SUGGEST_NAME` or `/suggest/dispatcher/DISPATCHER_NAME`	`SuggestClient`	`Suggester*`
Dictionary	`/dictionary`	`DictionaryClie`	n/a
Document fetch	`/fetch`	`FetchClient`	`Fetcher*`
Preview	`/preview`	`FetchClient`	`Fetcher*`
Thumbnail	`/thumbnail`	`FetchClient`	`Fetcher*`
Security	`/security/SOURCE_NAME`	`SecurityClient`	`AuthenticationClient*`
Search introspection	`/introspection/PARAMETER` It supports all Search Command parameters.	n/a	n/a

Note: * .NET classes also support these commands using built-in properties. You can specify other classes using custom parameters.

For a complete list of command parameters, see in the Exalead CloudView Configuration Guide.

Using the Search API Java Client

The Search API client allows easy usage of the commands directly from a Java application. For a description of the commands, see Search API Command Summary.

The base class to use is `com.exalead.searchapi.client.SearchAPIClientFactory`. You can create a `SearchAPIClientFactory` by specifying the root path of your Search API (for example, `http://<HOSTNAME>:<BASEPORT+10>/search-api`), and obtain specialized clients for each command.

For example, to perform queries on the `search-api` command:

```
SearchAPIClient client = SearchAPIClientFactory.build("http://localhost:10010");
SearchClient searchClient = client.searchClient("search-api");
SearchQuery sq = new SearchQuery("my test query");
SearchAnswer sa = searchClient.getResults(sq);
```

The `SearchQuery` class allows you to add all search parameters. The `SearchQuery` class provides several helpers to handle these parameters. There are four SearchAPI JAVA interfaces targeting Exalead CloudView V6, from which inherit 1 builder class per supported Exalead CloudView version:

- `CloudViewV6FacetingQueryBuilder` interface
- `CloudViewV6HitsQueryBuilder` interface
- `CloudViewV6RelevanceQueryBuilder` interface
- `CloudViewV6DynamicSearchTargetQueryBuilder` interface

For more information, see the Exalead CloudView *Public APIs Java SDK* documentation.

Note: The `SearchClient` object is thread safe. Use only one instance of the `SearchClient` for your whole program. It ensures that the HTTP connections are alive, to maintain request queueing without establishing useless connections to the service. To release the connections, call the `close()` method on the `SearchClient` instance.

Using the Search API .NET Client

The Search API .NET SDK is available in `<INSTALLDIR>/sdk/dotnet-clients`. It includes the client, documentation, and samples.

For a description of the commands, see Search API Command Summary.

For parameter details, see "Appendix - Search API Parameters" in the Exalead CloudView Configuration Guide.

Using the HTTP Search Command

If you do not want or cannot use the Java or .NET clients, you can directly access the search command.

About HTTP Search Queries

You send search queries to the command by:

- Sending a GET request to `http://host:port/COMMAND_BASE/search`
- Sending an application/x-www-form-urlencoded POST request to `http://host:port/COMMAND_BASE/search`

Search queries are key-value sets of search arguments. The full list of arguments is available in the *Search API Parameters Reference* documentation.

The main arguments are:

- "`q`": the query
- "`l`": the ISO code of the language

For example, the following query `http://host:port/search-api/search?q=test&l=en`

The reply of the Search command is an XML object representing the answer. The two main parts of the answer object are:

- The results of the query, in the top level `<hits>` node. Each hit consists of facets:
  - In the `<groups>` node of this hit,
  - And meta values in the `<metas>` node.
- The facets of the query, in the top level `<groups>` node.

Change Search Logic Dynamically

You may need to modify the search logic dynamically using search logic editing (`sle`) in your search query.

Modifications made to the search logic (`SearchLogic` class) are applied after XML parsing and before Search API parameters.

Refer to the API Javadoc available in `/<INSTALLDIR>/sdk/java-clients/docs/api/` for a full reference on the SearchLogic class.

Note: If you modify the structure of the search logic, `sle` queries are not updated automatically and may not work properly. Use the Search API parameters to keep the modifications you made to the search logic.

Build My Search Logic Editing

The structure of your search query is the following:

```
http://<HOSTNAME>:<BASEPORT+10>/search-api/search/search?q=mysearch&sle=<ACTION>:<FIE
[name="NAME"].<SEQUENCE>[0]
```

Where:

- `sle` indicates the use of search logic editing
- `<ACTION>` can either be:
  - `add:` add an element in a list
  - `set`: replace an element either in a field or in a list.
  - `remove:` remove and element from a list.

    Note: Use the same data type. For example, you can only replace a character string with another character string.
- Use `:` to separate action and fields
- Use `.` to separate fields or sequences
- Use `[0]` to select the first element in a list
- Use `[name="myname"]` to select the first element in a list with the specified `name`.

The full list of sequences is available in in the Exalead CloudView Configuration Guide.

Use Cases

- To remove the highlight in titles displayed in the search results, use:

  ```
  http://<HOSTNAME>:<BASEPORT+10>/search-api/search/search?q=mysearch&sle=remove:hit
  [name="title"].metaSpecificOperation[0]
  ```

  It:
  - Finds the first meta with `name` equal to `title`
  - Removes the first element in `metaSpecificOperation` list
- To add a snippet, use:

  ```
  http://<HOSTNAME>:<BASEPORT+10>/search-api/search/search?q=mysearch&sle=add:hitCon
  .metaSpecificOperation=SnippetOperation(highlightFacetIds="Event,Person",highlight
  "soundslike,spellslike",maxSentenceSegmentLength=210,maxLength=500)
  ```

It impacts the following section in `/<DATADIR>/config/SearchLogicList.xml`:

```
<s3:HitConfig fullHits="10">
  <s3:Meta name="text">
    <s3:FieldSource indexField="text"/>
    <s3:SnippetOperation highlightExtraPrefixHandlers="soundslike,spellslike" high
Person,Place_City,Place_Country,Place_State,Place_Landform,Place_CivicStructure,Pl
Place_Misc,Organization_Corporation,Organization_GovernmentOrg,Organization_NGO,
Organization_MiscellaneousOrg,Organization_Misc" ifMetasMatch="" relaxAndNodesPoli
highlight="true" splitOnSentences="true" maxConsecutiveSeparators="0" removeDuplic
 maxBytesToProcess="0" maxSentenceSegments="3" minNbWordsInRelevantSentence="10"
maxSentenceSegmentLength="2147483647" maxLength="500" minLength="1"/>
```

- To add a prefix handler, use:

  ```
  http://<HOSTNAME>:<BASEPORT+10>/search-api/search/search?q=mysearch&sle=add:uQLCon
  queryPrefixHandler=FullTextPrefixHandler(name="title",indexFields="title",dictiona
  matchingMode="normalized")
  ```

  The `title` prefix handler is used in the search query.

The Mashup API

The Mashup API, also known as the "Access API", provides a public HTTP interface to access the commands defined in Exalead CloudView's Search API configuration.

The Mashup API provides a standardized way to search and retrieve results from heterogeneous sources, called feeds. You can use independent feeds to retrieve different types of information, or nested feeds to enrich results from a previous feed.

You therefore have two APIs for search in your applications, the Search API and the Mashup API. For more information, see "Appendix - Search API Parameters" in the Exalead CloudView Mashup Programmer's Guide.

The Management API (MAMIs)

The MAMI is the public API that allows you to administer and configure Exalead CloudView.

You can use either the Java client or REST/XML.

Access the MAMI Using the Java Client

You can use the Java client located in `<INSTALLDIR>/sdk/java-clients`.

Replace Outdated SOAP Calls

In R2020x, the MAMI no longer supports WSDL SOAP calls. You can still access the MAMI API in REST/XML or with the java client.

This example shows how to translate a SOAP call to a non-SOAP call (adapt it for other MAMI services).

SOAP Call	Non-SOAP Call
`curl --header "Content-Type: text/xml;charset=UTF-8" --header "SOAPAction:tns:scanConnector" --data @Connector-Bind.xml http://localhost:10011/mami/connect/soap`	`curl --header "Content-Type: text/xml" --data @Connector-Bind.xml http://localhost:%MAMI_PORT%/mami/connect/scanConnector`

The API Client Factory

Why Use It?

This is a general purpose factory for Exalead CloudView SDK users.

You can use it to retrieve instances of the following objects, relative to a given product installation:

- Push Client
- Search Client
- Management API Client

How to Use It?

```
String  gatewayUrl = "http://localhost:10011/";
CloudviewAPIClientsFactory factory = CloudviewAPIClientsFactory.newInstance(gatewayUr
// Obtain a PushAPI client for build group bg0 and connector "myconn"
PushAPI papi = factory.newPushAPI("bg0", "myconn");
// Obtain a Search client
SearchClient search = factory.newSearchClient();
// Obtain the MAMI client
MAMIClient mami = factory.newMAMIClient();
```

You cannot use this factory through a rewrite PROXY. The PROXY might hide the real host name of the underlying product. This leads the factory to return API client instances configured with a host name that the caller cannot reach. In this case, you use directly the following factories:

- `com.exalead.papi.helper.PushAPIFactory`
- `com.exalead.searchapi.client.SearchAPIClientFactory`
- `com.exalead.mercury.mami.client.MAMIClientFactory`

For more information, see the *CloudView Clients SDK Javadoc* for the `CloudviewAPIClientsFactory` class.

The Semantic Factory SDK

This section describes how to use the Semantic Factory SDK.

It includes:

- An installation procedure, which details the Java Eclipse setup and the dependencies for the project provided in the *Semantic Factory Java SDK* and also at `<INSTALLDIR>/sdk/java-customcode/docs/api/index.html`
- Sample code resources and configuration file snippets
- Use cases

About the Semantic Factory

The Semantic Factory enhances Exalead CloudView's semantic capabilities by providing Mining of Text (MOT) processors as well as additional resources and semantic processors.

Specifically, the Semantic Factory allows you to use the following Exalead CloudView semantic processors:

- Semantic Entities Extractor (includes Semantic Query Analysis)
- Named Entities Extractor
- Related Terms Extractor
- Part of Speech Tagger
- Acronym Detector
- Chunker
- Fast Rules
- Rules Matcher

Text Processing Pipeline

Semantic processors allow you to analyze, transform, and annotate document texts. They are usually assembled sequentially to build a text processing pipeline.

MOTPipe

Exalead CloudView's text processing pipeline is named `MOTPipe` (Mining Of Text Pipe).

The MOTPipe architecture is designed to annotate natural language documents. Annotations can have different kinds: part of speech, named entity, ontology entry, etc. It is similar to the UIMA or Gates frameworks. This pipe contains a set of "processors" that are applied in a given sequence, and a set of linguistic resources they rely on.

The main difference between the MOTPipe and other frameworks is that it handles documents as an annotated token stream (for performance purposes).

A MOTPipe is composed by:

- A converter, which handles text segmentation.
- A list of resources (thread-safe, shared by several processors).
- A list of processors using resources (thread local).

With this approach, the performance of each successive component depends on the performance of each of the components that preceded it in the pipeline.

Note: Errors made by an "upstream" processor, like a part-of-speech tagging system, can negatively impact the performance of each "downstream" processor (such as a named entities recognizer).

Dependencies

A given processor can use results of previous processors in the pipeline. For example, an OntologyMatcher processor can annotate last names and a RulesMatcher (that is, NamedEntitiesMatcher) can thus exploit this kind of information to extract people's names.

The RulesMatcher processor needs to have an efficient way to retrieve the last names annotation added by OntologyMatcher processor.

The MOTPipe embeds a Referential component designed to share information between processors efficiently. When the MOTPipe is initialized, each processor of the pipe registers the annotation it will add to the Referential, and gets the corresponding `AnnotationId` (each annotation is identified by an Id for performance purposes). When a downstream processor

requires an annotation added by an upstream processor, it asks for it to the Referential during its registration step.

Resource

Several processors can use the same resource. Each Resource can have different versions and is identified by a name. A check at startup ensures that all processors refer to only known resources (name+version).

Processor

Each Processor is identified by a name. A processor references one or more Resources (with name+version). You can activate a processor on all input or on a set of contexts only.

Installing the Semantic Factory SDK

Java Project Requirements

This tutorial has the following requirements:

- An installed version of Exalead CloudView. For more installation details, see the Exalead CloudView Administration Guide.
- Eclipse IDE for Java Developers: `http://www.eclipse.org/downloads/`
- Java JDK v5 or later
- Libraries - The `<INSTALLDIR>/sdk/java-clients/lib` Java libraries must be in the classpath.
- Add the `jnicomexaleadmotcomponents` native library and all its dependencies to `LD_LIBRARY_PATH` (Linux) or the `PATH` (Windows) environment variable.

Create the Environment Variables Required for Resources

Most of the Semantic Factory processors use resources to perform their task. To indicate their location, the Semantic Factory uses environment variables.

1. Create an `NGRESOURCEPATH` variable specifying the path of your `<INSTALLDIR>/resource/all-arch` directory.
2. Create a `CVLICENSE` variable specifying the path of your Exalead CloudView license.

Set up the Semantic Factory Environment

Unzip the SDK and set up your development environment as follows.

1. Start your development environment application, create a new project, **Tokenizer** and then click **Finish**.
2. Drag the **lib** folder that contains the required jars to your project.
3. Right-click to select **Build Path > Add to Build Path** from the menu.
4. Drag the sample's **com** folder into the **src** folder of **Tokenizer**.

   Note: You can refer to the Java documentation for the Client library located in `<INSTALLDIR>/sdk/java-customcode/docs/api`

Getting Started with the Semantic Factory SDK

This section is a tutorial describing how to create your own semantic factory through typical examples.

Create a Simple MOTPipe

You can start by creating a MOTPipe that tokenizes input text and displays the result to the screen.

Important: A pipe is not thread safe. It is better to initialize one pipe for each thread, and to share resources.

You create a pipe in Java by using the `LinguisticFactory.buildPipe(MOTConfig config)` method, or one of the following:

- `LinguisticFactory.buildPipe(String motConfigPath)`
- `LinguisticFactory.buildPipe(MOTConfig config, int version)`
- `LinguisticFactory.buildPipe(String linguisticConfig, String tokenizationConfigName, List<SemanticProcessor> processors)`

- `LinguisticFactory.buildPipe(String linguisticConfig, String tokenizationConfigName, List<SemanticProcessor> processors, int version)`
- `LinguisticFactory.buildPipe(String linguisticConfig, String tokenizationConfigName)`
- `LinguisticFactory.buildPipe(List<Tokenizer> tokenizers, NormalizerConfig norm, List<SemanticProcessor> processors)`
- `LinguisticFactory.buildPipe(List<Tokenizer> tokenizers, NormalizerConfig norm, List<SemanticProcessor> processors, int version)`

Use the methods signatures with additional `ResourcesContext` arguments to share resources between multiple pipes. For example, `LinguisticFactory.buildPipe(ResourcesContext ctx, String motConfigPath)`

Once created, you must initialize the pipe using its `init()` method to ensure the loading of the resources. To free resources when the pipe is no longer used, call the `release()` method.

Create a MOTPipe in Java

Below is a code snippet that initializes the pipe named `_pipe` to do simple tokenization of text:

```
List<Tokenizer> tokenizers = new ArrayList<Tokenizer>();
tokenizers.add(new StandardTokenizer());
    MOTPipe pipe = LinguisticFactory.buildPipe(tokenizers, new NormalizerConfig(),
new ArrayList<SemanticProcessor>());
System.out.println("Loading pipe...");
pipe.init();
System.out.println("Pipe loaded.");
```

Create a MOTPipe with an XML Configuration File

Creating a MOTPipe programmatically can be tedious task when its number of processors increases. It is more convenient to rely on an XML configuration file. For example:

```
<ling:MOTConfig xmlns="exa:com.exalead.linguistic.v10">
  <!-- Tokenizers -->
  <ling:StandardTokenizer />
  <!-- Normalizer -->
  <ling:NormalizerConfig />
  <!-- Semantic Processors -->
</ling:MOTConfig>
```

Then, you can load the MOTPipe as follows:

```
MOTPipe pipe = LinguisticFactory.buildPipe(pathToXMLFile);
```

Process the Input Text

To process the input text, declare the processing of a new document using the `newDocument()` method. It must be followed after processing by a call to `endDocument()`. These two calls can trigger some specific tasks from processors, for example, adding annotations directly at the document level.

The following snippet processes the provided string content. The language is unspecified (`Language.XX`), as only tokenization is done.

```
pipe.newDocument();
AnnotatedToken[] tokens = pipe.process(content, Language.XX);
print(tokens);
pipe.endDocument();
```

Code Sample

The following class uses the preceding code and displays the analysis result on the standard output stream.

```
package com.exalead.mot.tutorial;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;import java.util.List;
import com.exalead.lang.Language;
import com.exalead.linguistic.LinguisticFactory;
import com.exalead.linguistic.v10.NormalizerConfig;
import com.exalead.linguistic.v10.StandardTokenizer;
import com.exalead.linguistic.v10.Tokenizer;
import com.exalead.mot.core.MOTPipe;
import com.exalead.mot.v10.AnnotatedToken;
import com.exalead.mot.v10.Annotation;
public class Step1Tokenization {
    public static void main(String[] argv) throws IOException {
if (argv.length != 1) {
        System.err.println("usage: java com.exalead.mot.tutorial.Step1Tokenization /p
        System.exit(1);
   }
        Step1Tokenization step1 = new Step1Tokenization();
        if (!step1.init(argv[0])) {
            return;
        }
        BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
        try {
            while (true) {
                System.out.print("Enter a sentence to parse: ");
                step1.process(stdin.readLine());
```

```
            }
        } catch (IOException e) {
            step1._pipe.release();
        }
    }
    private MOTPipe _pipe;
    public Step1Tokenization() {
    }
public boolean init(String motConfigPath) {
        try {
            _pipe = LinguisticFactory.buildPipe(motConfigPath);
            System.out.println("Loading pipe...");
            _pipe.init();
            System.out.println("Pipe loaded.");
        } catch (Exception e) {
            e.printStackTrace();
            System.err.println("Could not load pipe: " + e.getMessage());
            return false;
        }
        return true;
    }
    private void print(AnnotatedToken[] tokens) {
        for (int i = 0; i < tokens.length; ++i) {
            AnnotatedToken tok = tokens[i];
            System.out.println(" Token[" + tok.token + "] kind[" + AnnotatedToken.nam
lng[" + Language.name(tok.lang) + "] offset[" + tok.offset + "]");
            print(tok.annotations);
        }
    }
    private void print(Annotation[] annotations) {
        for (int j = 0; j < annotations.length; ++j) {
            Annotation ann = annotations[j];
            System.out.println(" Annotation[" + ann.displayForm + "] tag[" + ann.tag
[" + ann.nbTokens + "]");
        }
   }
    public void process(String content) {
        _pipe.newDocument();
        AnnotatedToken[] tokens = _pipe.process(content, Language.XX);
        print(tokens);
        _pipe.endDocument();
    }
}
```

When you run it, you obtain an output similar to the following.

```
Loading pipe...
Pipe loaded.
Enter a sentence to parse: hello world
```

```
  Token[hello] kind[ALPHA] lng[xx] offset[0]
    Annotation[hello] tag[LOWERCASE] nbTokens[1]
    Annotation[hello] tag[NORMALIZE] nbTokens[1]
  Token[ ] kind[SEP_SPACE] lng[xx] offset[5]
  Token[world] kind[ALPHA] lng[xx] offset[6]
    Annotation[world] tag[LOWERCASE] nbTokens[1]
    Annotation[world] tag[NORMALIZE] nbTokens[1]
```

Perform Language Detection and Word Lemmatization

Language Detection with Java

The `MOTPipe.process()` method accepts language as a parameter. The tokenizer uses this language later for all the tokens it creates.

Let us redo the previous example using `Language.EN` instead of `Language.XX`. Note how the `lng` attribute of the tokens has changed from `lng[xx]` to `lng[en]`.

Processors may support all, one or a subset of languages. A lemmatizer, for example, is dedicated to tokens in the language of its resource only.

Recommendation: Perform a detection when the language of the content is unknown, or when it may contain several languages. To do so, use the Language Detector processor.

The following snippet illustrates the creation of such a processor and its declaration in a pipe.

```
List<Tokenizer> tokenizers = new ArrayList<Tokenizer>();
tokenizers.add(new StandardTokenizer());
LanguageDetector languageDetector = new LanguageDetector();
languageDetector.setName("languageDetector");
List<SemanticProcessor> processors = new ArrayList<SemanticProcessor>();
processors.add(languageDetector);
pipe = LinguisticFactory.buildPipe(tokenizers, new NormalizerConfig(), processors);
```

With this initialization, running the program detects the language of tokens. The processor replaces the language used in the call to `MOTPipe.process()` with the one it detects.

Note: A sufficient number of words is required to correctly perform the detection. In a long text containing different languages, the processor gives each token its most probable language.

Language Detection with XML Configuration File

The following configuration file illustrates the creation of the processor and its declaration in a pipe.

```
<ling:MOTConfig xmlns="exa:com.exalead.linguistic.v10">
  <!-- Tokenizers -->
  <ling:StandardTokenizer/>
  <!-- Normalizer -->
```

```
  <ling:NormalizerConfig />
  <!-- Semantic Processors -->
  <ling:LanguageDetector name="languageDetector" />
</ling:MOTConfig>
```

Retrieve Document Annotations

Both tokens and documents can receive annotations. You can retrieve document annotations with a call to `MOTPipe.getDocumentAnnotations()` after the call to `MOTPipe.endDocument()` (processor usually uses this trigger to add their annotations to the document).

The Language Detector processor adds an annotation for each detected language in the document with the following snippet:

```
pipe.newDocument();
pipe.newField("field");
AnnotatedToken[] tokens = pipe.process(content, Language.XX);
print(tokens);
pipe.endDocument();
Annotation[] annotations = pipe.getDocumentAnnotations();
System.out.println("  Document");print(annotations);
```

You obtain the following result:

```
Enter a sentence to parse: have a good day
  Token[have] kind[ALPHA] lng[en] offset[0]
    Annotation[have] tag[LOWERCASE] nbTokens[1]
    Annotation[have] tag[NORMALIZE] nbTokens[1]
  Token[ ] kind[SEP_SPACE] lng[en] offset[4]
  Token[a] kind[ALPHA] lng[en] offset[5]
    Annotation[a] tag[LOWERCASE] nbTokens[1]
    Annotation[a] tag[NORMALIZE] nbTokens[1]
  Token[ ] kind[SEP_SPACE] lng[en] offset[6]
  Token[good] kind[ALPHA] lng[en] offset[7]
    Annotation[good] tag[LOWERCASE] nbTokens[1]
    Annotation[good] tag[NORMALIZE] nbTokens[1]
  Token[ ] kind[SEP_SPACE] lng[en] offset[11]
  Token[day] kind[ALPHA] lng[en] offset[12]
    Annotation[day] tag[LOWERCASE] nbTokens[1]
    Annotation[day] tag[NORMALIZE] nbTokens[1]
  Document
    Annotation[en] tag[language] nbTokens[0]
```

Lemmatization with Java

Lemmatization is a common linguistic task that consists in identifying the lemma of each word using a language dictionary.

To enable lemmatization in both English and French, we need to add two dedicated processors. To do so, complete the pipe creation as shown in Language Detection with Java by the following snippet:

```
Lemmatizer englishLemmatizer = new Lemmatizer();
englishLemmatizer.setName("englishLemmatizer");
englishLemmatizer.setLanguage("en");
processors.add(englishLemmatizer);
Lemmatizer frenchLemmatizer = new Lemmatizer();
frenchLemmatizer.setName("frenchLemmatizer");
frenchLemmatizer.setLanguage("fr");
processors.add(frenchLemmatizer);
```

The lemmatizers then enrich the tokens generated. The result looks like the output below:

```
Token[books] kind[ALPHA] lng[en] offset[65]
  Annotation[books] tag[LOWERCASE] nbTokens[1]
  Annotation[books] tag[NORMALIZE] nbTokens[1]
  Annotation['book', 'book', 'n', 'p', 'book', 'book', 'noun'] tag[lemmainformation]
```

Here a new annotation is added for the token `books`. It corresponds to an array containing the following information: `[lowercase masculine singular, normalized masculine singular, genre, number, lowercase singular, normalized singular, category]`. You can access it in a more convenient way using the `Lemmatizer.deserialize()` method, which returns a `LemmaInformation` instance that provides accessors.

You can add several annotations if there is any ambiguity. For example, use a Part of Speech Tagger processor next to disambiguate.

Lemmatization with an XML Configuration File

```
<ling:MOTConfig  xmlns="exa:com.exalead.linguistic.v10">
  <!-- Tokenizers -->
  <ling:StandardTokenizer />
  <!-- Normalizer -->
  <ling:NormalizerConfig />
  <!-- Semantic Processors -->
  <ling:LanguageDetector name="languageDetector" />
  <ling:Lemmatizer name="englishLemmatizer" language="en" />
  <ling:Lemmatizer name="frenchlemmatizer" language="fr" />
</ling:MOTConfig>
```

Explore the Configuration File

Each item in the configuration file may accept many parameters.

```
<ling:MOTConfig xmlns="exa:com.exalead.linguistic.v10">
  <!-- Tokenizers -->
```

```
  <ling:StandardTokenizer >
    <ling:charOverrides>
      <ling:StandardTokenizerOverride type="token" toOverride=":" />
    </ling:charOverrides>
    <ling:patternOverrides>
      <ling:StandardTokenizerOverride type="token" toOverride="[[:alnum:]][&amp;][[:a
      <ling:StandardTokenizerOverride type="token" toOverride="[[:alnum:]]*[.]net" />
      <ling:StandardTokenizerOverride type="token" toOverride="[[:alnum:]]+[+]+" />
      <ling:StandardTokenizerOverride type="token" toOverride="[[:alnum:]]+#" />
    </ling:patternOverrides>
  </ling:StandardTokenizer>
  <!-- Normalizer -->
  <ling:NormalizerConfig>
    <ling:NormalizerIndexLower language="fr" word="thé" />
    <ling:NormalizerIndexLower language="fr" word="maïs" />
  </ling:NormalizerConfig>
  <!-- Semantic Processors -->
  <ling:Lemmatizer name="englishLemmatizer" language="en" />
  <ling:Lemmatizer name="frenchLemmatizer" language="fr" />
</ling:MOTConfig>
```

You can see that the Standard Tokenizer accepts specific overrides:

- For characters: the `:` character is considered as an alphabetical character and not punctuation. For example, `a:b` is only one token.
- For patterns: any sequence of characters that match the specified regexp are considered as a whole token. For example, `R&D` that matches the first pattern.

You can also configure the Normalizer with normalization exceptions. Here, the forms `thé`, `Thé`, or `THÉ` are all normalized to `thé` and not `the`.

Extract Named Entities

You can extract more complex information such as Named Entities.

Note: Dependencies that are not declared in the configuration are implicitly included during initialization.

A dedicated processor is available for this task: the `NamedEntitiesMatcher`. For more information, see the *CloudView Configuration Reference* and "Named Entities Matcher" in the *Exalead CloudView Configuration Guide* .

To use it, configure it in the XML configuration file as follows:

```
<NamedEntitiesMatcher name="neMatcher" prefix="NE" />
```

Required dependencies are automatically added during initialization (`RelatedTerms` and recursively `PartOfSpeechTagger`). You do not need to include them explicitly in the configuration.

The `NamedEntitiesMatcher` processor adds named entities annotations on the tokens.

Note: Annotations are added only on the first tokens composing the entities.

The `nbTokens` property indicates the scope of the annotation. The result looks like the output below:

```
Token[Barack] kind[ALPHA] lng[xx] offset[0]
    Annotation[barack] tag[LOWERCASE] nbTokens[1]
    Annotation[barack] tag[NORMALIZE] nbTokens[1]
    Annotation[barack obama] tag[relatedTerm] nbTokens[3]
    Annotation[Barack Obama] tag[relatedTermDisplay] nbTokens[3]
    Annotation[Barack Obama] tag[exalead.people] nbTokens[3]
    Annotation[] tag[exalead.nlp.firstnames] nbTokens[1]
    Annotation[famouspeople] tag[NE] nbTokens[3]
    Annotation[1] tag[sub] nbTokens[3]
    Annotation[Barack Obama] tag[NE.famouspeople] nbTokens[3]
  Token[ ] kind[SEP_SPACE] lng[xx] offset[6]
  Token[Obama] kind[ALPHA] lng[xx] offset[7][...]
```

Here is a summary of the tag values:

- `relatedTerm` = noun phrase in a lemmatized + normalized form
- `relatedTermDisplay` = noun phrase full-text form
- `sub` = internal indicator that you can ignore
- `exalead.nlp.firstnames` = internal indicator (presence of a first name)
- `exalead.people` = internal indicator (entity present in the Exalead persons dictionary built from wikipedia/freebase/dbpedia)
- `NE and NE.famouspeople` = result of named entity detection under two different forms based on the specified prefix (that is, "NE" in our example)

The last annotation is the most useful. It contains the canonical form of the entity as well as a tag specifying its type. You can rely on the tag prefix (defined in the configuration file) to locate named entities annotations. When possible, the content of the annotation is a canonical form of the entity that is specified in a thesaurus (for example, "United States" and "U.S." are normalized to "USA"), or inferred with linguistic rules.

Use Case

This use case describes how to extract vehicle registration plates.

For such a task, a simple `RulesMatcher` processor is adequate. This processor can match patterns expressed by rules against a token stream. Rules are described in a dedicated XML configuration file. For the syntax description, see "Rules Matcher (rule-based)" in the *Exalead CloudView Configuration Guide* .

Example: the following sample shows how to extract French vehicle registration plates.

```
<TRules xmlns="exa:com.exalead.mot.components.transducer">
  <!-- SIV (e.g. AA-229-AA) -->
  <TRule priority="0">
    <MatchAnnotation kind="NE.plates.SIV"/>
    <Seq>
      <Or>
        <TokenRegexp value="[A-Za-z]{2}"/>
        <Word value="W" level="exact"/>
  <!-- to match temporary plates starting with only one W (e.g. W-001-AA) -->
      </Or>
      <Opt> <Word value="-" level="exact"/> </Opt>
      <TokenRegexp value="[1-9]{3}"/>
      <Opt> <Word value="-" level="exact"/> </Opt>
      <TokenRegexp value="[A-Za-z]{2}"/>
    </Seq>
  </TRule>
  <!-- FNI (e.g. 1233 CD 33) -->
  <TRule priority="1">
    <MatchAnnotation kind="NE.plates.FNI"/>
    <Or>
      <Seq>
        <TokenRegexp value="[1-9]{1,4}"/>
        <TokenRegexp value="[A-Za-z]{1,3}"/>
        <Or>
          <TokenRegexp value="[1-9]{2}"/>
          <TokenRegexp value="2[AB]"/>  <!-- Corse -->
          <TokenRegexp value="97[1-8]"/>  <!-- DOM-TOM -->
        </Or>
      </Seq>
      <Seq>
        <TokenRegexp value="[1-9]{1,6}"/>
        <Or>
          <Word value="NC" level="exact"/>  <!-- New Caledonia -->
          <Word value="P" level="exact"/>  <!-- French Polynesia -->
        </Or>
      </Seq>
      <Seq>  <!-- TAAF - Kerguelen islands  -->
        <TokenRegexp value="[05-9][1-9]"/>
        <TokenRegexp value="[1-9]{4}"/>
      </Seq>
      <Seq>  <!-- Wallis-and-Futuna -->
```

```
        <TokenRegexp value="[1-9]{1,4}"/>
        <Word value="WF" level="exact"/>
      </Seq>
    </Or>
  </TRule>
</TRules>
```

To use it, add a RulesMatcher in the configuration file:

```
<RulesMatcher name="platesMatcher" resourceFile="resource:///tutorial-mot/plates.xml"
```

In this example, we used the Exalead resource protocol. It implies that the resource is relative to the `NGRESOURCEPATH` directory (see Java Project Requirements). The file protocol is also supported and you can use it to specify an absolute path. The result looks like the output below.

```
Token[AA] kind[ALPHA] lng[fr] offset[0]
    Annotation[aa] tag[LOWERCASE] nbTokens[1]
    Annotation[aa] tag[NORMALIZE] nbTokens[1]
    Annotation[AA-123-BB] tag[NE.plates.SIV] nbTokens[5]
  Token[-] kind[DASH] lng[fr] offset[2]
  Token[123] kind[NUMBER] lng[fr] offset[3][...]
```

Available Processors

The following list describes the main processors that are available in the Semantic Factory, with their dependencies and supported languages.

Name	Dependencies	Description	Supported Languages
`AcronymDetector`		Detects acronyms.	
`AnnotationManag`		Provides basic operations on annotations (copy, removal, and so on).	
`Categorizer`		Machine learning classifier, categorizes the whole document according to a learning resource.	
`Chunker`	`PartOfSpeechTag`	Detects subject/verb in a sentence.	`en, fr, it`
`FastRulesMatche`		Matches documents against rules.	
`LanguageDetecto`		Detects language of tokens. This processor can detect language of small sentence and handle multi-languages documents.	over 100 languages

Name	Dependencies	Description	Supported Languages
`Lemmatizer`		Identifies the lemma of each word using a language dictionary (no disambiguation).	`de, en, es, fr, it, pt`
`NamedEntitiesMa`	`RelatedTerms`	Detects named entities (people, organizations, places, events, emails, dates, currency, French addresses, urls, French phone numbers, French/English opening hours)	
`NGram`		NGram extractor.	
`OntologyMatcher`		Extracts words/expressions defined in an ontology.	Depends on the ontology content.
`PartOfSpeechTag`		Detects part of speech (noun/verb/adjective/...) for each token with disambiguation.	`fr, it, en`
`Phonetizer`		Phonetizes tokens.	`ca, cs, da, de, en, es, et, fa, fi, fr, it, nl, no, pl, pt, ro, ru, sk, sl, sv`
`PrettyPrinter`		Prints pretty tokens.	
`Proximity`		Annotates pieces of text where a number of annotations appear close to each other.	
`RelatedTerms`	`PartOfSpeechTag` only if `withPartOfSpeech` (default value)	Extracts noun phrases from the tokens' stream.	`ar, ca, cs, da, de, en, es, et, fa, fi, fr, he, it, ja, nl, no, pl, pt, ro, ru, sk, sl, sv, zh`
`RulesMatcher`		Extracts 'patterns' from the tokens' stream.	

Name	Dependencies	Description	Supported Languages
SemanticExtract		Extraction of semantic features (numbers, strings)	
SentenceFinder		Detects sentence breaks.	
SentimentAnalyz	Lemmatizer + Chunker	Extracts positive/negative sentiments using a domain-specific resource (need customization for a specific domain).	en, fr, it
SnowballStemmer		Rule-based stemmer	da, du, en, es, fi, fr, de, hu, it, no, pt, ro, ru, sv, tu
SpellChecker		Performs spell check.	
URLRemover		Removes URL from token streams.	
WordDictionary		Matches from a dictionary.	

Writing Custom Tokenizers and Semantic Processors

Exalead CloudView is delivered with a vast number of semantic processors that can alter documents in analysis pipelines. You can perform most analysis tasks by assembling these processors. However, for advanced and custom operations, it may be more convenient to write custom semantic processors.

You can:

- Replace the analysis pipeline tokenizer with a custom one written in Java.
- Add a custom semantic processor at the end of the analysis pipeline.

Both are implemented as custom document processors, so make sure that you are acquainted with the proper way to develop and deploy them on a Exalead CloudView instance.

For information on how to build and deploy with the Eclipse plugin, see "Develop and deploy components using the Eclipse plugin".

About Tokens and Annotations

Write a Java Custom Tokenizer

Write a Java Custom Semantic Processor

About Tokens and Annotations

Tokenizers and semantic processors work on a stream of annotated tokens:

- Tokenizers produce them from the input text,
- And processors add and remove them according to the algorithm they implement.

Create Tokens

Basically, a token is the aggregation of:

- A form (a piece of the input text),
- A type (alphabetical, punctuation, separator, etc.),
- And an array of annotations.

When developing a tokenizer, you have to create tokens and to define several of the following fields. The framework defines some of them automatically.

```
package com.exalead.mot.v10;
public class AnnotatedToken
{
    /// The token value (form)
    public String token;
    /// The token kind (type)
    public int kind;
    /// The language code (defined by com.exalead.lang.Language)
    public int lang;
    /// The position in the original text of this token (in terms of characters)
    public int offset;
    /// The list of annotations attached to this token
    public Annotation[] annotations;
    /// Returns the XML representation of this token and its annotations
    public String toString();
    /// Returns the annotations attached to this token having the given tag
    /// If none, returns the empty list.
    public List<Annotation> getAnnotationsWithTag(String tag);
    /// Returns the annotations having any of the given tags.
    /// If none, returns the empty list.
    public List<Annotation> getAnnotationsWithTags(Collection<String> tags);
    /// Returns the annotations having the given tag and display form.
    /// If none, returns the empty list.
    public List<Annotation> getAnnotationsWithTagAndDisplay(String tag, String displa
    /// Token kinds and their default interpretation
    public final static int TOKEN_UNKNOWN = 0;       /// unknown
    public final static int TOKEN_SEP_IGNORE = 1;    /// separator [[:ctrl:]]
    public final static int TOKEN_SEP_SPACE = 2;     /// space [[:space:]]
```

```
    public final static int TOKEN_SEP_SENTENCE = 4;  /// sentence
    public final static int TOKEN_SEP_PARAGRAPH = 8; /// paragraph (\n\n)
    public final static int TOKEN_SEP_QUOTE = 16;    /// quote ["']
    public final static int TOKEN_SEP_DASH = 32;     /// dash [-]
    public final static int TOKEN_SEP_PUNCT = 64;    /// punct [[:punct:]]
    public final static int TOKEN_NUMBER = 128;      /// number [0-9]+
    public final static int TOKEN_ALPHANUM = 256;    /// alphanum [a-zA-Z0-9]+
    public final static int TOKEN_ALPHA = 512;       /// alpha [a-zA-Z]+
}
```

Create Annotations

Annotations are pairs of key/value strings (tag and display form) of a certain length, expressed in tokens.

In addition, an arbitrary integer value between [0, 100] is associated (trust level). Its semantics is left to the implementors of algorithms.

Here is the definition of `com.exalead.mot.v10.Annotation`:

```
package com.exalead.mot.v10;
public class Annotation
{
    /// The tag (key)
    public String tag;
    /// The display form (value)
    public String displayForm;
    /// Length of this annotation
    public int nbTokens;
    /// The trust level
    public int trustLevel;
    /// XML representation of this annotation
    public String toString();
}
```

Write a Java Custom Tokenizer

A Java Custom Tokenizer is useful for processing the text with an external analyzer or for implementing a specific behavior. The `JavaCustomTokenizer` allows you to write your own code for splitting the input and possibly adding annotations to the produced tokens.

These tokens then follow their way in the indexing chain as usual (see Sample Tokenizer).

Write a Java Custom Tokenizer

If you derive your `MyTokenizer` class from `com.exalead.pdoc.analysis.JavaCustomTokenizer`, you have to implement at least the following:

```
@PropertyLabel(value = "JavaCustomTokenizer")
@CVComponentConfigClass(configClass = MyTokenizerConfig.class)
@CVComponentDescription(value = "My tokenizer in Java")
public class MyTokenizer extends com.exalead.pdoc.analysis.JavaCustomTokenizer
{
    /**
     * CustomDocumentProcessor requires a constructor accepting a custom configuratio
     * This constructor must call JavaCustomTokenizer constructor
     */
    public MyTokenizer(MyTokenizerConfig config);
    /**
     * Called when a new document is about to get processed.
     */    public void newDocument();
    /**
     * Called when there is no more input to process in the current document.
     * This is the last chance to attach annotations to the document if needed.
     */    public void endDocument();
    /**     * Called when a new input chunk is to be processed.
     * The processor must:
     * - produce tokens from the text using newToken() and newAnnotation()
     * - send the said tokens to the semantic pipe with pushToken()
     *
     * @param text the chunk text
     * @param language the chunk language
     * @param context the chunk context
     * @throws InvalidTokenException
     * @see newToken(), newAnnotation(), pushToken()
     * @post Concatenation of the token forms must be strictly equal to the input str
     */
    public void processChunk(String text, int language, String context) throws Except
    /**
     * Called at initialization to retrieve the annotation tags that are planned to b
 tokenization.
     * @return the list of all annotation tags needed or null if none
     */
    public String[] declareAnnotations();
}
```

The `JavaCustomTokenizer` provides a handful of helper methods:

```
package com.exalead.pdoc.analysis;
public abstract class JavaCustomTokenizer extends CustomDocumentProcessor
 {
    ...
    /**
     * Allocate a new token of the provided form.
     * The token is either created or recycled from a previous use.
     * The token type and language are deduced from the form and the chunk input lang
(they can be overridden though).
```

```
     * @param form the new token form
     * @return a fresh or recycled token
     * @pre form is not null
     * @pre form is not empty
     * @post token kind is set using default typing
     */
    protected AnnotatedToken newToken(String form) throws InvalidTokenException;
    /**
     * Allocate a new annotation with the provided tag, value and length.
     * The annotation is either created or recycled from a previous use.
     * @param tag the new annotation tag
     * @param displayForm the new annotation value
     * @param nbTokens the new annotation length
     * @return a fresh or recycled annotation
     * @pre tag must have been declared in declareAnnotation()
     * @pre displayForm is not null
     * @pre nbTokens > 0
     */
    protected Annotation newAnnotation(String tag, String displayForm, int nbTokens)
InvalidAnnotationException;
    /**
     * Send a token to the output stream.
     *
     * - validity of the token is checked
     * - the token is added to the output buffer
     * - if needed, the output buffer is flushed
     * - the token is recycled
     * In all cases, the token and its annotations are no longer usable after the cal
     * @param token A token allocated through a call to newToken()
     * @pre token is not null
     * @pre token form is not null nor empty
     * @pre token type is defined
     * @see newToken(), newAnnotation()
     */
    protected void pushToken(AnnotatedToken token) throws InvalidTokenException;
    /**
     * Attach an annotation to the currently processed document after checking its va
     * @param annotation the annotation to attach
     * @pre annotation must have been allocated with newAnnotation()
     * @see newAnnotation()
     */
    protected void addDocumentAnnotation(Annotation annotation) throws InvalidAnnotat
    ...
}
```

Caveats for Tokenizers

- When creating new tokens, you have to specify their form (attribute token) and possibly their annotation set. The kind, language, and offset are automatically defined. You may want to override the kind if the default typing does not suit your needs. Be careful with the token kind, as it has a huge impact on the way a token is indexed (or not). If a token is malformed, `newToken()` or `pushToken()` throw `InvalidTokenException`.
- When creating new annotations, you have to specify their tag, display form, number of tokens, and possibly their trust level. If an annotation is malformed, `newAnnotation()` or `addDocumentAnnotation()` throw an `InvalidAnnotationException`.
- Do not allocate annotated tokens and annotations by yourself, always use `newToken()` and `newAnnotation()` as they are pooled and recycled once pushed to the pipeline, to save RAM.
- Since tokens and annotations are recycled, they are not usable anymore once pushed to the pipeline. Request a new token/annotation through `newToken()`/`newAnnotation()` if required.
- Avoid allocating too many tokens and annotations before pushing them to the pipeline. Ideally, to guarantee optimal RAM consumption, push a token before allocating a new one.
- Your code is executed in a multi-threaded environment. Each thread has its own tokenizer so that your code does not have to be thread safe. However, threads share static objects, and this may lead to issues in case of concurrent modifications.

Sample Tokenizer

This sample demonstrates how to write a custom tokenizer that:

- Splits the input text into alphabetical, numerical, punctuation, or blank tokens.
- Annotates with "Capitalized" each token that starts with an upper-case letter.
- Annotates with "Number" each token made of digits.
- Pushes the produced tokens to the semantic pipeline.

```
package com.exalead.customcode.analysis;
/**
 * This processor can be configured with:
 *  <CustomDocumentProcessor classId="com.exalead.customcode.analysis.CustomTokenizer
 *   <KeyValue key="Meta" value="mymeta" />
 *  </CustomDocumentProcessor>
 */
@CVComponentConfigClass(configClass = com.exalead.customcode.analysis.CustomTokenizer
CustomTokenizerConfig.class)
@CVComponentDescription(value = "My tokenizer in Java")
```

```
public class CustomTokenizer extends JavaCustomTokenizer
 {
    public static class CustomTokenizerConfig implements CVComponentConfig
     {
        private String meta;
        public String getMeta() {
            return meta;
        }
        @IsMandatory(true)
        public void setMeta(String meta) {
            this.meta = meta;
        }
    }
        public CustomTokenizer(CustomTokenizerConfig config) throws Exception {
        super(config);
    }
    @Override
    public void newDocument() {
         logger.debug("I'm about to start tokenizing a new document!");
    }
    @Override
    public void endDocument() {
        logger.debug("Done with this document!");
    }
    @Override
    public void processChunk(String text, int language, String context) throws Except
        logger.debug("Tokenizing [" + text + "] in context [" + context + "] with lan
(language) + "]");
        Matcher matcher = pattern.matcher(text);
        while (matcher.find()) {
            AnnotatedToken token = newToken(text.substring(matcher.start(), matcher.e
            if (token.token.matches("^\\p{Lu}.*$")) {
                Annotation[] a = { newAnnotation("Capitalized", token.token.toLowerCa
                token.annotations = a;
                       } else if (token.token.matches("^[0-9]+$")){
                Annotation[] a = { newAnnotation("Number", token.token.toLowerCase(),
                token.annotations = a;
            }
            pushToken(token);
        }
    }
    @Override
    public String[] declareAnnotations() {
        return new String[] { "Capitalized", "Number" };
    }
    private static Pattern pattern = Pattern.compile("\\p{L}+|\\p{N}+|\\p{Z}+|\\p{P}|
p{P}]+");
    private Logger logger = Logger.getLogger("mycustomtokenizer");
```

```
}
```

Write a Java Custom Semantic Processor

The Java custom semantic processor allows you to plug your code as the last semantic processor in the pipeline.

You use as input a flow of annotated tokens from the pipeline, have an opportunity to add or remove any annotation, and then send the tokens back to the indexing chain.

Note: For more information, see "About Semantic Processors" in the *Exalead CloudView Configuration Guide* .

Write a Java Custom Semantic Processor

The difference with the Java Custom Tokenizer is in the input:

- The tokenizer receives a text chunk to process.
- While for the Java Custom Semantic Processor, you have to get the tokens from the pipeline (see Sample Semantic Processor).

Derive your `MySemanticProcessor` class from `com.exalead.pdoc.analysis.JavaCustomSemanticProcessor` and implement:

```
@PropertyLabel(value = "JavaCustomSemanticProcessor")
@CVComponentConfigClass(configClass = MySemanticProcessorConfig.class)
@CVComponentDescription(value = "My semantic processor in Java")
public class MySemanticProcessor extends com.exalead.pdoc.analysis.JavaCustomSemantic
{
    public MySemanticProcessor(MySemanticProcessorConfig config) throws Exception;
    /**
     * Called when a new document is about to get processed.
     */
    public void newDocument();
    /**
     * Called when there is no more input to process in the current document.
     * This is the last chance to attach annotations to the document if needed.
     */
    public void endDocument();
     /**
     * Called at initialization to retrieve the annotation tags that are planned to b
     * Only declared annotations will be accessible on tokens retrieved with getNextT
     * @return the list of all annotation tags needed or null if none
     */
     public String[] declareAnnotations();
     /**
     * Called when a new input chunk is to be processed.
     * The processor must pump tokens from pipe using getNextToken()
```

```
     * and return them once processed to the pipe with pushToken().
     *
     * @param text the chunk text
     * @param language the chunk language
     * @param context the chunk context
     * @see getNextToken(), newAnnotation(), pushToken()
     */
     public void processChunk(String chunk, int language, String context) throws Exce
 }
```

The `JavaCustomSemanticProcessor` provides you with helpers too:

```
package com.exalead.pdoc.analysis;
public abstract class JavaCustomSemanticProcessor extends CustomDocumentProcessor
 {
    ...
    /**
     * Pump the next token from the input stream.
     * @return the next token from the pipe or null if end of input is reached
     */
    protected final AnnotatedToken getNextToken();
    /**
     * Allocate a new annotation with the provided tag, value and length.
     * The annotation is either created or recycled from a previous use.
     *
     * @param tag the new annotation tag
     * @param value the new annotation value
     * @param nbTokens the new annotation length
     * @return a fresh or recycled annotation
     * @pre tag must have been declared in declareAnnotation()
     * @pre value is not null
     * @pre nbTokens > 0
     */
     protected final Annotation newAnnotation(String tag, String displayForm, int nbT
 InvalidAnnotationException;
    /**
     * Send a token to the output stream.
     *
     * - validity of the token is checked
     * - the token is added to the output buffer
     * - if needed, the output buffer is flushed
     * - the token is recycled
     *
     * In all cases, the token and its annotations are not usable anymore after the c
     *
     * @param token A token allocated through a call to newToken()
     * @pre token is not null
     * @pre token form is not null nor empty
     * @pre token type is defined
```

```
     * @see newToken(), newAnnotation()
     */
     protected final void pushToken(AnnotatedToken token) throws InvalidTokenExceptio
    /**
     * Attach an annotation to the currently processed document
     *
     * @param annotation the annotation to attach
     * @pre annotation must have been allocated with newAnnotation()
     * @see newAnnotation()
     */
    protected final void addDocumentAnnotation(Annotation annotation) throws InvalidA
    ...
}
```

Caveats for Semantic Processors

- When creating new annotations, you have to define their tag, display form, number of tokens and possibly their trust level. If the annotation is malformed annotation, `newAnnotation()` or `addDocumentAnnotation()` throw an `InvalidAnnotationException`.
- To remove an annotation from a token, assign it a null value in the `annotations[]` array.
- For the custom tokenizer, do not allocate annotations by yourself but always use `newAnnotation()` as to save RAM, annotations are pooled and recycled once pushed to the pipeline.
- Since tokens and annotations are recycled, they are not usable anymore once pushed to the pipeline. The only way to get a new token is through `getNextToken()`. Always allocate annotations through `newAllocation()`.
- Keep as few tokens in memory as possible before pushing them back to the pipeline. Ideally, a token must be pushed before getting the next one from the pipeline.
- Your code is executed in a multi-threaded environment, where each thread has its own processor, so that your code does not have to be thread-safe. However, threads share static objects, which can possibly lead to trouble in case of concurrent modifications.

Sample Semantic Processor

This sample demonstrates how to write a custom semantic processor that:

- Gets tokens from the semantic pipeline
- Looks for a first name that is part of its dictionary
- Checks if the following word starts with a capitalized letter
- Add an annotation `people` if there is not an annotation `NE.people` already.
- Pushes tokens back to the pipeline

```
package com.exalead.customcode.analysis;
/**
 * This processor can be configured with:
 *  <CustomDocumentProcessor classId="com.exalead.customcode.analysis.CustomSemanticP
 *   <KeyValue key="Meta" value="mymeta" />
 *  </CustomDocumentProcessor>
 */
@CVComponentConfigClass(configClass = com.exalead.customcode.analysis.CustomSemanticP
CustomSemanticProcessorConfig.class)
@CVComponentDescription(value = "My semantic processor in Java")
public class CustomSemanticProcessor extends JavaCustomSemanticProcessor
 {
    public static class CustomSemanticProcessorConfig implements CVComponentConfig
     {
        private String meta;
        public String getMeta() {
            return meta;
        }
        @IsMandatory(true)
        public void setMeta(String meta) {
            this.meta = meta;
        }
     }
     public CustomSemanticProcessor(CustomSemanticProcessorConfig config) throws Exce
        super(config);
        for(String s : firstNames) {
            names.add(s);
        }
    }
    @Override
    public void newDocument() {
        logger.debug("I'm about to start processing a new document!");
    }
    @Override
    public void endDocument() {
        logger.debug("Done with this document!");
    }
    @Override
    public String[] declareAnnotations() {
        return new String[] { "people" };
    }
    @Override
    public void processChunk(String chunk, int language, String context) throws Excep
        for (AnnotatedToken token = getNextToken(); token != null; token = getNextTok
            if (token.kind == AnnotatedToken.TOKEN_ALPHA && names.contains(token.toke
token.getAnnotationsWithTag("NE.people").isEmpty()) {
                AnnotatedToken next = getNextToken();
                if (next != null) {
```

```
                    if (next.kind == AnnotatedToken.TOKEN_SEP_SPACE) {
                        AnnotatedToken next2 = getNextToken();
                        if (next2 != null) {
                            if (next2.kind == AnnotatedToken.TOKEN_ALPHA && next2.tok
("\\p{Lu}.+")) {
                                Annotation annotation = newAnnotation("people", token
 + " " + next2.token, 3);
                                token.annotations = (Annotation[]) ArrayUtils.add
(token.annotations, annotation);
                            }
                            pushToken(token);
                            pushToken(next);
                            pushToken(next2);
                            continue;
                        }
                     }
                     pushToken(token);
                    pushToken(next);
                    continue;
                }
            }
             pushToken(token);
        }
    }
    private Logger logger = Logger.getLogger("custom-semantic-processor");
    private static String[] firstNames = { "John", "Bill", "Steve", "Robert", "George
 "Frank" };
    private HashSet<String> names = new HashSet<String>();
}
```

Customizing CloudView

You can customize many Exalead CloudView areas by writing custom components in Java.

Recommendation: To develop, package, and deploy your custom components, use the Exalead CloudView Eclipse plugin.

About Customizing Exalead CloudView

You can create the following custom components:

- Connectors for your data sources
- Document processors
- Query processors to reinterpret user queries
- Prefix handlers to reinterpret parts of user queries
- Search runners to customize the whole search processing pipeline
- Security sources
- Mashup Builder feeds, triggers, or widgets

  Note: Mashup Builder is in a separate bundle. See the Exalead CloudView Mashup Programmer's Guide.

- Resources

Important: Code hosted by a custom process cannot use plugins (because of classloader issues).

Customization Workflow

To create custom components manually, the workflow is the following:

1. Create your custom CVComponents.

2. Package your component as plugin. See Package a Plugin

3. Deploy the plugin as:

- A `cvplugin`, see Deploy a Plugin.
- Or a `devplugin`, see Work with Plugins in Development Mode.

Develop and Deploy Components Using the Eclipse Plugin

The Exalead CloudView Eclipse plugin helps you to develop and deploy plugins in Eclipse Indigo 3.7 or later.

The plugin documentation is available at `http://eclipse.exalead.com`.

You can develop custom components for:

- Exalead CloudView core
  - Connectors
  - Document Processors
  - Meta Processors
  - Prefix Handlers
  - Push API Filters
  - Query Processors
  - Security Sources
- Exalead CloudView Mashup
  - Widgets
  - Feeds
  - Feed Triggers
  - Mashup Triggers
  - Pre-Request Triggers
  - MEL Functions
  - Security Providers

Why Use It

- Easy deployment: you can package your plugin with customized components to export and deploy automatically on the selected instance of Exalead CloudView. For Mashup Builder, this avoids rebuilding and redeploying the `standalone-mashup-ui.war` package for each customized item.
- Quick export: you can package your plugin with classes that you want to export and then export it as a zip file on a selected path.
- Manage installs: you can list the deployed plugins on a selected instance of Exalead CloudView and then select the plugins to remove.

Creating Custom Components for CloudView

All custom components in Exalead CloudView (except Mashup) use a generic mechanism, called `CVComponent`, to handle the instantiation and configuration aspects.

In the Exalead CloudView configuration, you use your custom component by:

- Referencing its class.
- Providing the configuration for this usage of the component.

The `CVComponent` mechanism transmits the configuration to a new instance of your class.

In the Exalead CloudView configuration, the configuration of the custom component is given as a hierarchical listing of string key-values.

It is then transformed into a structured object, which must implement the `CVComponentConfig` interface.

For example, to create a custom analysis processor that connects to an auxiliary data source needing a login and password, define the component as follows:

```
package com.mycompany.myprocessor;
import com.exalead.mercury.component.config.CVComponentConfig;
public class MyProcessorConfig implements CVComponentConfig {
    private String login;
    private String password;
    public String getLogin() { return login; }
    public void setLogin(String login) { this.login = login; }
    public String getPassword() { return password; }
    public void setPassword(String password) { this.password = password; }}
package com.mycompany.myprocessor;
import com.mycompany.myprocessor.MyProcessorConfig;
import com.exalead.pdoc.analysis.CustomDocumentProcessor;
import com.exalead.mercury.component.config.CVComponentConfigClass;
```

```
@CVComponentConfigClass(configClass=MyProcessorConfig.class)
public class MyProcessor extends CustomDocumentProcessor {
  public MyProcessor(MyProcessorConfig config) {
    super(config);
    this.config = config;
  }
  @Override
  public void process(DocumentProcessingContext context,ProcessableDocument document)
     // Connect to the data source
     connect(config.login, config.password);
     // Work ...
  }
}
```

And in the configuration:

```
<CustomDocumentProcessor classId="com.exalead.myprocessor.MyProcessor">
  <KeyValue xmlns="exa:exa.bee" key="Login" value="mylogin" />
  <KeyValue xmlns="exa:exa.bee" key="Password" value="myverysecretpassword" />
</CustomDocumentProcessor>
```

Exalead CloudView automatically transforms the KeyValue in the object specified on `@CVComponentConfigClass` annotation on your component.

The basic rules are:

- If your component does not have any configuration, you must annotate it with: `@CVComponentConfigClass(configClass=CVComponentConfigNone.class)`.
- Your config class must inherit `CVComponentConfig`.
- Your config class must have the regular Java Beans getters and setters.

Packaging Custom Components as Plugins

Plugins are components or resources that can be hot plugged without restarting the product. They are installed in the `DATADIR`, and are therefore instance-wide.

You can upload plugins using the Administration Console.

To load your custom code in Exalead CloudView, you must package it as a CVPlugin.

Packaging your code as a plugin allows Exalead CloudView to:

- Detect it automatically.
- List the possible components for a given role (like "all custom prefix handlers").
- Provide information about the default configuration.

Package a Plugin

A CVPlugin is a simple ZIP file containing the following structure:

```
/lib/
   myjar1.jar
   myjar2.jar
/META-INF/
   cvplugin.properties
```

`cvplugin.properties` can contain the following keys:

- `plugin.description`
- `plugin.version`
- `plugin.copyright`
- `plugin.author`

The JARs must contain the components and their associated classes, including config classes. A plugin ZIP file can include several components. All of them are taken into account.

```
// Optionally a set of hints about the content
components.widgets = MyWidget1, MyWidget2
components.resources = MyResource1
components.cvcomponents = MyClass1, MyClass2
// CVComponents are usually automatically detected, but this may be required for spec
// Optionally, a set of properties about a given component
component.MyResource1.license = Commercial, contact foo@acmecorp.com
```

Each plugin is loaded in its own classloader, avoiding many common conflict cases.

Deploy a Plugin

You can deploy your plugins using the Administration Console or the `cvadmin` tool on the command line.

Install a Plugin in the Administration Console

1. Go to **Deployment > Plugins**
2. Click **Upload plugin** and browse for your file.

Install a Plugin on the Command Line

You can install plugins regardless of whether the Exalead CloudView product is running or stopped.

1. Go to `<DATADIR>/bin` and run:

```
./cvadmin plugins
```

2. Enter:

   Note: For multihost installs, run this command on the master host to distribute the plugin to all hosts automatically.

3. Restart the processes for which you are going to use its components, typically the search-server, or analyzer.

You can then use the plugin.

List Installed Plugins

1. From the `<DATADIR>/bin`, get the list of the installed Exalead CloudView plugins:

   ```
   ./cvadmin plugins list
   ```

Uninstall a Plugin

1. From the `<DATADIR>/bin`, get the list of the installed Exalead CloudView plugins:

   ```
   ./cvadmin plugins list
   ```

2. Remove the plugin:

   ```
   ./cvadmin plugins remove name=myplugin
   ```

Work with Plugins in Development Mode

The process of packaging as a ZIP and then installing can be too time-consuming at the development stage, when you generally do not need full versioning and multiple-host installations, but do need a short testing cycle.

In that case, you can use dev plugins, as you do not need to package them as ZIP. Instead, you copy the JAR file.

The limitation is that dev plugins are not automatically distributed on all instances of the Exalead CloudView cluster.

1. Upload your JAR to: `<DATADIR>/extrajava/devplugins/<MY_PLUGIN_NAME>/lib`

   Typically, you can put all your JARs in the same plugins folder. A default folder, `default_dev_plugin` exists.

   Recommendation: Because of potential class conflicts, it is better to create your own dev plugins folder.

2. Restart the processes for which you are going to use its components, typically the searchserver, or analyzer.

Customizing Document Analysis

Exalead CloudView is delivered with a vast number of document processors that can alter documents in analysis pipelines. By assembling these processors, most analysis tasks can be performed. However, for advanced and custom operations, it is often required or more convenient to write custom document processors.

A custom document is a Java class extending the `com.exalead.pdoc.analysis.CustomDocumentProcessor` class. It manipulates the document as a `com.exalead.pdoc.ProcessableDocument` object.

Note: For functional details on document processors, see the *Exalead CloudView Configuration Guide* .

What Can a Document Processor Do?

A document processor can:

- Modify, create, or remove document metas.
- Modify, create, or remove document parts.
- Discard a document: ignore it, or delete it from the index.

A document processor cannot:

- Modify the URI or stamp of a document.
- Create new documents.

Samples

Several samples of document processors are available in the Exalead CloudView kit, in `<INSTALLDIR>/sdk/java-customcode/samples/document-processors`.

You can build the samples using Apache Ant. This creates a plugin zip file that you can install in Exalead CloudView.

Debugging

The `process()` method of the `CustomDocumentProcessor` receives a `DocumentProcessingContext` argument. Use the `DocumentProcessingContext` method to report any error or warning with the document. This ensures that all error context is adequately captured for efficient debugging.

Write Custom Document Processors Inline

You can write document processors directly in the Administration Console, using the integrated code editor.

1. Open the Administration Console at http://<HOSTNAME>:<BASEPORT+1>/admin.
2. Go to **Index > Data Processing >** Pipeline name **> Document Processors**.
3. Expand **Custom** and drag a **Java Document Processor** to the **Current processors** panel.
4. Select **Inline Java**, click **Edit java**.
5. Click **Check source code** to verify that the code compiles correctly.
6. Click **Accept** and then **Apply**.

Your custom document processor is now active.

Add a Custom Document Processor to Your Analysis Pipeline

Once you have developed your custom document processor, you can add it to your document analysis pipeline in the Administration Console.

Package and upload the plugin containing your document processor.

1. Open the Administration Console at http://<HOSTNAME>:<BASEPORT+1>/admin.
2. Go to **Index > Data Processing >** Pipeline name **> Document Processors**.
3. Expand **Custom** and drag a **Custom Document Processor** to the **Current processors** panel.
4. Fill in the **Class id** (available document processors are suggested automatically).
5. If there is additional configuration for the processor, you can fill in the configuration keys.

Customizing Search Processing

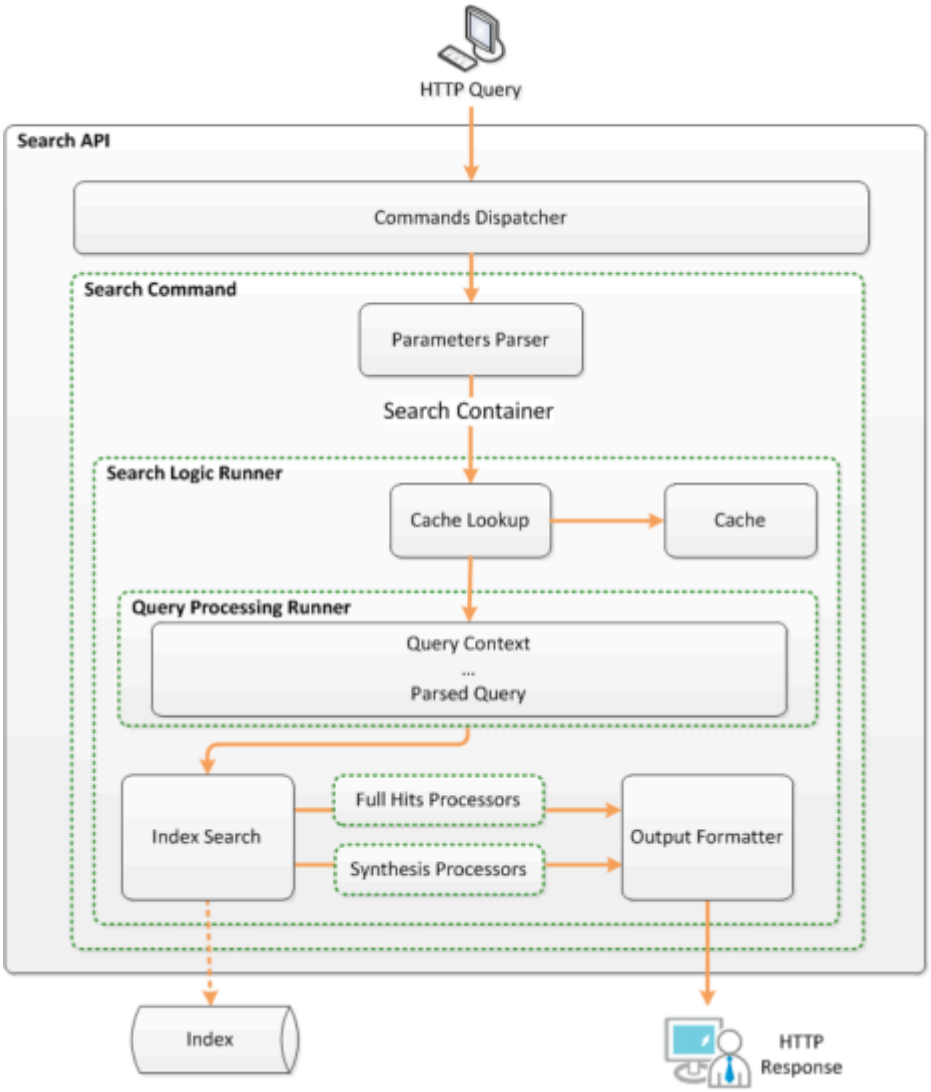
You may need to customize the way queries are processed. This section focuses only on the processing of queries, once sent to the Search API. Customization points are possible, especially when using the Mashup API and the Mashup UI.

Note: For more information, see "Configuring Search Queries" in the *Exalead CloudView Configuration Guide*.

How Are Search Queries Processed

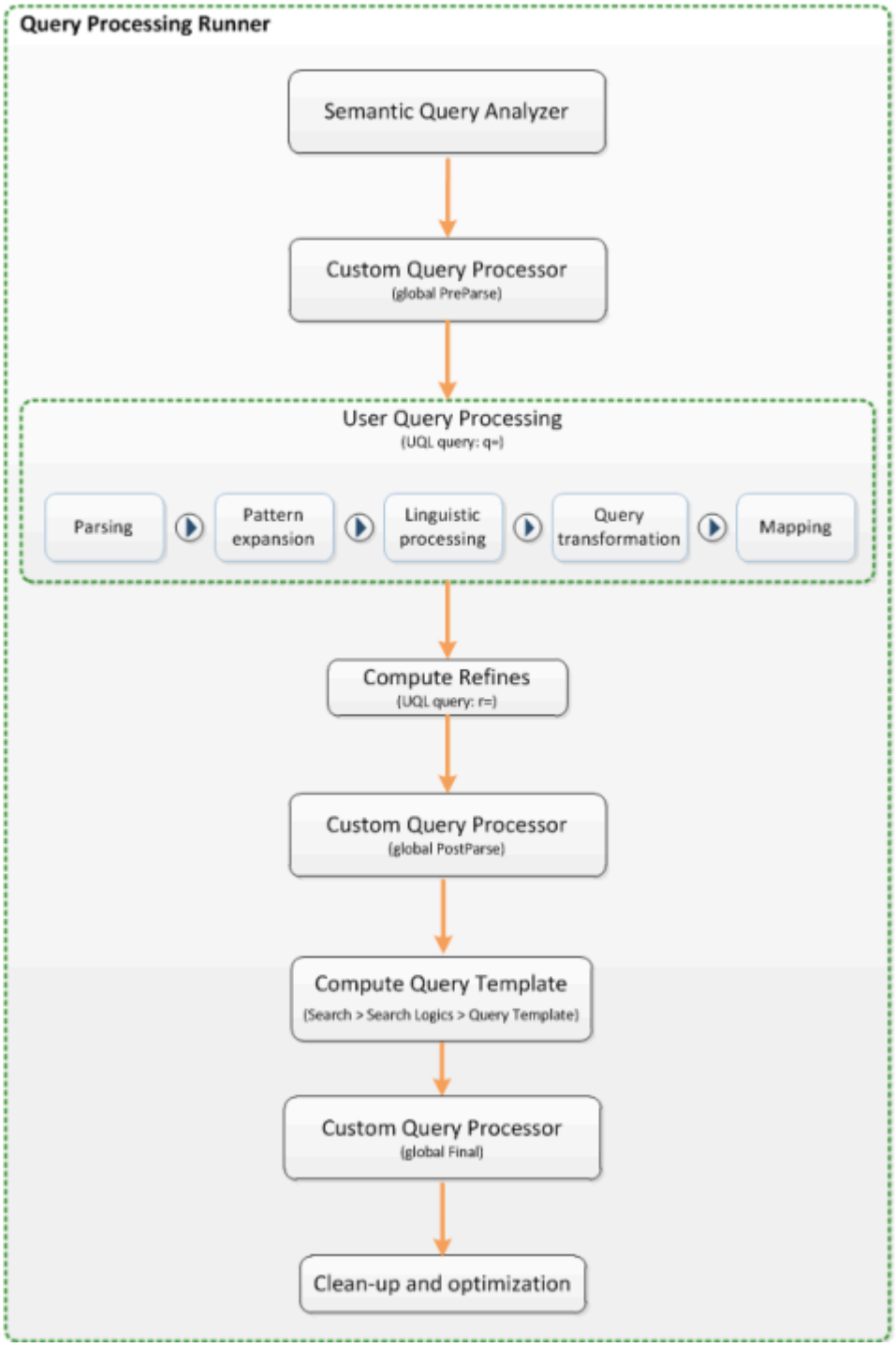
This section shows the overall query processing workflow as well as a detailed view of the query processing runner.

Query Processing Workflow



The following schema summarizes the workflow at the Query Processing Runner level.

Focus on Query Processing Runner



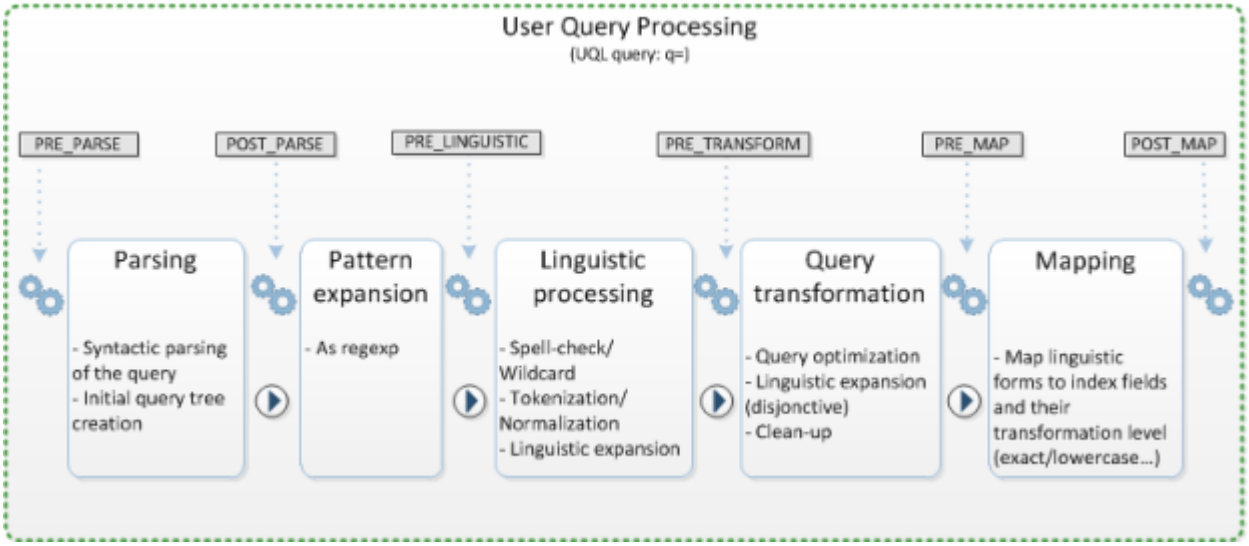
Add Custom Query Processors or Prefix Handlers

To customize your query, you may add either custom query processors or custom prefix handlers at specific stages of the user query processing.

Custom query processors apply to the whole AST. Custom prefix handlers apply to a subpart of the query, for example, date or text.

Where Can You Plug Them?

In the figure below, the icon indicates where you can plug custom processors:



Write Custom Query Processors

A query processor is the simplest component that you can write to customize query processing. It takes the whole `QueryContext` as input, and allows you to modify the context (including the AST in its current state).

See Where Can You Plug Them? to know where you can add custom query processors.

Write a Query Processor

Extend the `com.exalead.search.query.processors.CustomQueryProcessor` class.

Enable Your Custom Query Processor

Edit the `SearchLogicList`.

For example, to add a preparse processor, edit the `LogicRunnerCustomization` node of your `SearchLogic` as follows:

```
<s2:LogicRunnerCustomization>
 <s2:globalPreParseProcessors/>
   <s2:preParseProcessors>
        <s2:CustomProcessor classId="com.customer.processors.MyCustomProcessor">
           <s2:KeyValue xmlns="exa:exa.bee" key="AConfigKey" value="A value" />
        </s2:CustomProcessor>
      </s2:preParseProcessors>
     <s2:preLinguisticProcessors/>
    <s2:preTransformProcessors/>
   <s2:preMapProcessors/>
  <s2:postMapProcessors/>
```

```
  <s2:globalPostParseProcessors/>
 <s2:globalFinalProcessors/>
</s2:LogicRunnerCustomization>
```

Sample Custom Query Processor

In the sample below, a custom query processor refines the query by adding access restrictions.

This query processor looks into the context (HTML parameter) and finds the profile value given by our custom UI. Depending on that value, the query processor expands the query with a specific prefix handler (perimeter) and the corresponding restricting value.

You can plug this custom module at `globalPreParse` or `preParse` processors level (before syntactic parsing, that is to say AST creation.)

```
package com.exalead.customcode.search;
import com.exalead.mercury.component.CVComponentDescription;
import com.exalead.mercury.component.config.CVComponentConfig;
import com.exalead.search.query.QueryContext;
import com.exalead.search.query.QueryProcessingException;
import com.exalead.search.query.node.Node;
import com.exalead.search.query.node.NodeVisitor;
import com.exalead.search.query.node.RootNode;
import com.exalead.search.query.node.UserQueryString;
import com.exalead.search.query.processors.CustomQueryProcessor;
import com.exalead.search.query.processors.QueryProcessorContext;
@CVComponentDescription("(Sample) Profile based query rewritting")
public class ProfileBasedRestrictionQueryProcessor extends CustomQueryProcessor {
       public ProfileBasedRestrictionQueryProcessor(CVComponentConfig config) throws
QueryProcessingException {
        super(config);
    }
    @Override
    public void onInit(QueryProcessorContext logic) {
        System.out.println("Connection to the profile DataBase onInit");
    }
    @Override
    public void onDeinit(boolean allInstances) {
        System.out.println("Close the connection from the profile DataBase onDeinit")
    }
    public boolean hasRestrictedAccess(String profileId){
        if (profileId.equals("admin")){
            return false;
        }
         return true;
    }
     @Override
     public void process(QueryContext context) throws QueryProcessingException {
        String profile = context.query.parameters.getParameterValue("profile");
```

```
        if(profile != null){
            if ( hasRestrictedAccess(profile) ){
                context.currentNode = new RootNode(context.currentNode.accept(new MyQ
            }
        }
                ;
    }
    static class MyQueryRewriter extends NodeVisitor {
      @Override
      public Node visit(UserQueryString queryString) {
            // add a prefix handler to restrict usage
            queryString.value += " perimeter:restrictive";
            return queryString;
        }
    }
}
```

Write Custom Prefix Handlers

You can write your own prefix handlers to customize query processing.

It takes the prefix handler value as input, and allows you to modify the context (including the AST in its current state).

See Where Can You Plug Them? to know where you can add custom prefix handlers.

Write a Prefix Handler

Extend the `com.exalead.search.query.prefix.CustomPrefixHandler` class.

Create and Package Your Custom Prefix Handler

1. Develop your prefix handler.

   Recommendation: Use the Eclipse plugin. See Develop and Deploy Components Using the Eclipse Plugin.

2. Package it as a plugin to deploy in Exalead CloudView. See Packaging Custom Components as Plugins.
3. Apply prefix handlers to your Exalead CloudView configuration. See "The different types of prefix handlers" in the *Exalead CloudView Configuration Guide* .

Sample Custom Prefix Handler

The Exalead CloudView default behavior is to use the `AND` operator between words. In this sample, you use a prefix handler to change it.

You can replace the default `AND` operator by custom operators (`XOR`, `OR`, `AND`, or `FUZZYAND`) using a custom prefix handler.

A `FUZZYAND` operator uses a ratio of minimum terms to match a document:

- `FUZZYAND/d` with `d` being a positive or negative number:
  - `FUZZYAND/2` means at least two of these words.
  - `FUZZYAND/-1` means all words but one can be missed.
- `FUZZYAND` (without suffix). In that case, the prefix handler asks for a minimum of half words.

```
package com.exalead.customcode.search;
import com.exalead.config.bean.PropertyLabel;
import com.exalead.mercury.component.CVComponent;
import com.exalead.mercury.component.CVComponentDescription;
import com.exalead.mercury.component.config.CVComponentConfigClass;
import com.exalead.search.query.QueryContext;
import com.exalead.search.query.QueryProcessingException;
import com.exalead.search.query.node.And;
import com.exalead.search.query.node.FuzzyAnd;
import com.exalead.search.query.node.NaryNode;
import com.exalead.search.query.node.Node;
import com.exalead.search.query.node.NodeVisitor;
import com.exalead.search.query.node.Or;
import com.exalead.search.query.node.PrefixNode;
import com.exalead.search.query.node.UserQueryChunk;
import com.exalead.search.query.node.Xor;
import com.exalead.search.query.prefix.CustomPrefixHandler;
import com.exalead.search.query.util.NodeInspector;
@PropertyLabel(value = "Change Behavior")
@CVComponentConfigClass(configClass=ChangeDefaultBehaviorPrefixHandlerConfig.class)
@CVComponentDescription("(Sample) This prefix handler allows to change the default be
another operator rather than the default one.")
public class ChangeDefaultBehaviorPrefixHandler extends CustomPrefixHandler implement
    private String operator = null;
    public ChangeDefaultBehaviorPrefixHandler(ChangeDefaultBehaviorPrefixHandlerConfi
      super(config);
      operator = config.getOperator();
    }
    public void addChildren(NaryNode newNode, String[] queryParts) {
             }
       @Override
       public Node handlePrefix(Phase phase, PrefixNode node, NodeVisitor parentVisit
            QueryContext queryContext) throws QueryProcessingException {
          if(phase.equals(Phase.PRE_LINGUISTIC)) {
            int slashPos = -1;
            String queryContent = NodeInspector.concatenatedContent(node).value;
            String[] queryParts = queryContent.split(" ");
```

```
            NaryNode newNode = null;
            if (operator.contains("XOR")) {
                newNode = new Xor();
            } else if (operator.contains("OR")) {
                newNode = new Or();
            } else if (operator.contains("FUZZYAND")) {
                newNode = new FuzzyAnd();
                slashPos = operator.indexOf("/");
                if (slashPos < 0){
                    ((FuzzyAnd)newNode).minSuccess = (int)queryParts.length / 2;
                } else if (operator.charAt(slashPos + 1)== '-'){
                    ((FuzzyAnd)newNode).maxFailure= new Integer(operator.substring(sl
                } else {
                    ((FuzzyAnd)newNode).minSuccess= new Integer(operator.substring(sl
                }
            } else {
                newNode = new And();
            }
            for (int i = 1; i < queryParts.length; i++){
                newNode.addChild(new UserQueryChunk(queryParts[i]));
            }
            if (parentVisitor != null) {
                return newNode.accept(parentVisitor);
            }
             return newNode;
           }
           if (parentVisitor !=null) {
               node.setContent(node.getContent().accept(parentVisitor));
        }
        return node;
    }
}
```

Customizing Metas

In Exalead CloudView's Search Logics, you can configure new or existing metas to use a custom meta processor for the search result content.

Write a Custom Meta Processor

Using the Exalead CloudView Eclipse plugin you can develop custom components such as Meta Processors.

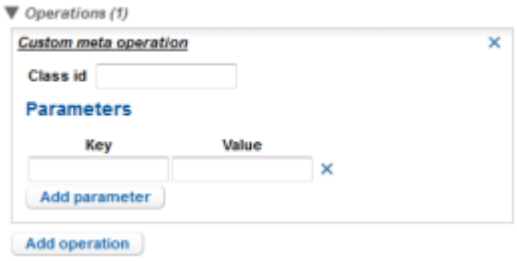
1. Make sure that your custom processor inherits from the `Custom{Double,Long,String,Text}MetaProcessor` class. The sample below demonstrates the `CustomStringMetaProcessor` class.

2. Even if your component does not have any config, you must still write the annotation with `configClass=CVComponentConfigNone.class`.

```
import java.util.ArrayList;
import java.util.List;
import com.exalead.mercury.component.CVComponentDescription;
import com.exalead.mercury.component.config.CVComponentConfig;
import com.exalead.mercury.component.config.CVComponentConfigClass;
import com.exalead.mercury.component.config.CVComponentConfigNone;
import com.exalead.search.pipeline.full.CustomStringMetaProcessor
@CVComponentDescription("Prefix metavalue with test_") @CVComponentConfigClass
(configClass = CVComponentConfigNone.class)
          public class PrefixTextMetaProcessor extends CustomStringMetaProcessor {
         public PrefixTextMetaProcessor(CVComponentConfig config, String metaName) {
            super(config, metaName);
            item.config = config;
    }
    /**
     * Entry point of the meta processor, called with the list of values, and
     * which must return the new list of values
     */
    @Override
    public Iterable<String> onMeta(String... value)
    {
       List<String> list = new ArrayList<String>();
       for (String v : value)
       {
      list.add("test_" + v);
       }
       return list;
    }
}
```

Enable Your Custom Meta Processor

1. Open the Administration Console at http://<HOSTNAME>:<BASEPORT+1>/admin.
2. Go to the **Hit Content** page in **Search Logics > sl0**.
3. Select an existing meta and click **Customize**. You can also create a new meta.
4. In the meta settings, expand **Operations** to click **Add operation** and then select **Custom meta operation**.



5. Enter your custom meta processor in **Class id**.

   The custom processor must inherit from the class `CustomMetaProcessor`.

6. (Optional) Enter the **Key** and **Value** parameters.
7. Click **Apply** to save the configuration.

Your custom meta processor is complete.

Customizing Alerting

This section describes how to create custom query alerts.

With Query Alerting, users can save queries and be alerted whenever new or modified documents that match these queries are added to the index. You can configure it using the Business Console (see "Setting up Query Alerting" the *Exalead CloudView Business Console User's Guide*) or the MAMI (see below.)

AlertingManager Class

The `AlertingManager` class provides a simple API to manage existing alerts, add new ones, and handles the communication with the Alerting MAMI.

```
import com.exalead.cloudview.mami.client.ManagersCreator;
import com.exalead.cloudview.alerting.service.AlertingManager;
.
.
ManagersCreator managerCreator = new ManagersCreator(GW_URL, GW_LOGIN, GW_PASSWORD);
AlertingManager alertingManager = managerCreator.get(AlertingManager.class, "/mami/al
```

Manage Alerts

Create an Alert

When creating an alert, you have to specify the `AlertDesc` instance.

According to the alerting mode (scheduled or real-time), the following fields are mandatory:

Field	Mandatory for	Example
NAME	Scheduled Real-time	-
USERNAME	Scheduled Real-time	-
GROUP	Scheduled Real-time	When using several groups, use commas as separators. For example, 'group1, group2'
QUERY_ARGS	Scheduled Real-time	For the query london flat refined on file extension, language and source: 'cloudview.r=f%2FSource%2F4&cloudview.r=f%2FLanguage%2Fen&cloudview.r=%2Bf%2Ffile_extension%2F6&q=london%20flat&isDebugModeEnable=false&lang=en'
PAGE	Scheduled	-
UI_LEVEL_QUERY_AF	Scheduled	For the query london flat refined on file extension, language and source: 'cloudview.r=f%2FSource%2F4&cloudview.r=f%2FLanguage%2Fen&cloudview.r=%2Bf%2Ffile_extension%2F6&q=london%20flat'

The Description field is optional.

Example 1. Sample

```
import com.exalead.cloudview.alerting.v10.*;
.
.
AlertDesc alertDesc = new AlertDesc().withName(NAME).withUser(USERNAME).withGroup(GRO
.withQueryArgs(MASHUP_API_QUERY_ARGS);
alertDesc.withDescription(DESCRIPTION); // optional
alertDesc.withUiLevelQueryArgs(UI_LEVEL_QUERY_ARGS); // scheduled only, mandatory
alertDesc.withPage(PAGE); // scheduled only, mandatory
AddAlert mAMICmd = new AddAlert();
mAMICmd.withAlertDesc(alertDesc);
alertingManager.addAlert(mAMICmd);
```

Retrieve Existing Alerts or Alert Groups

Example 2. Sample for Alerts

```
import com.exalead.cloudview.alerting.v10.*;
.
.
AlertDescList alertList = alertingManager.getUserAlerts(new GetUserAlerts().withUser(
withGroup(GROUP));
// for a given user & a given group
// or
alertList = alertingManager.getUserAlerts(new GetUserAlerts().withUser(USERNAME).with
// for all users & a given group
for (AlertDesc alertDesc : alertList.getAlertDesc()) { // iterates on retrieved alert
  // do whatever you want with your alert
}
```

Example 3. Sample for Alert Groups

```
import com.exalead.cloudview.alerting.v10.*;
.
.
AlertingStatus status = alertingManager.getAlertingStatus(new GetAlertingStatus());
for (AlertingGroupStatus groupStatus : status.getAlertingGroupStatus()) {
// iterates on retrieved alert group statuses
  // do whatever you want with your alert group status
  System.out.println("Name: '" + groupStatus.getName() + "'");
  System.out.println("Description: '" + groupStatus.getDescription() + "'");
  System.out.println("Default? '" + groupStatus.isUseAsDefault() + "'");
}
```

Edit an Alert

When creating an alert, a key is automatically defined to identify it. This key is required to edit an alert and is it retrieved from the Alerting Manager.

Example 4. Sample

```
import com.exalead.cloudview.alerting.v10.*;
.
.
AlertDesc anExistingAlert; // retrieve key
anExistingAlert.withName(NEW_ALERT_NAME); // edit alert name
EditAlert editCmd = new EditAlert().withAlertDesc(anExistingAlert);
alertingManager.editAlert(editCmd);
```

Remove an Alert

When creating an alert, a key is automatically defined to identify it. This key is required to remove an alert and is it retrieved from the Alerting Manager.

Example 5. Sample

```
import com.exalead.cloudview.alerting.v10.*;
.
```

```
.
AlertDesc anExistingAlert; // retrieve key
anExistingAlert.withName(NEW_ALERT_NAME); // edit alert name
DeleteAlert removeCmd = new DeleteAlert().withKey(anExistingAlert.getKey());
alertingManager.deleteAlert(removeCmd);
```

Test a Scheduled Alert Group Manually

Example 6. Sample

```
import com.exalead.cloudview.alerting.v10.*;
.
.
alertingManager.runAlertGroup(new RunAlertGroup().withGroup(GROUP);
alertingManager.abortAlertGroupRun(new AbortAlertGroupRun().withGroup(GROUP);
```

Implement Custom Publishers

Using the alerting API, you can implement custom publishers for:

- Real-time actions
- Scheduled actions

Each custom publisher is made of:

- A configuration class
- A publisher class

Custom Real-Time Publisher

Example 7. Publisher Configuration Class

```
import com.exalead.cloudview.alerting.v2.realtime.CustomRTPublisherConfig;
import com.exalead.config.bean.PropertyLabel;
@PropertyLabel("Log RT Publisher")
public class LogRTPublisherConfig extends CustomRTPublisherConfig {
    private String logPrefix;
    public String getLogPrefix() {
        return logPrefix;
    }
    public void setLogPrefix(String logPrefix) {
        this.logPrefix = logPrefix;
    }
}
```

Example 8. Publisher Class

```
import org.apache.log4j.Logger;
import com.exalead.cloudview.alerting.v2.realtime.CustomRTPublisher;
import com.exalead.cloudview.alerting.v2.realtime.RTAlertOccurrence;
```

```
import com.exalead.cloudview.alerting.v2.realtime.RTMatch;
import com.exalead.mercury.component.config.CVComponentConfigClass;
@CVComponentConfigClass(configClass = LogRTPublisherConfig.class)
public class LogRTPublisher extends CustomRTPublisher {
    private final static Logger logger = Logger.getLogger("alerting.v2.rt.log");
    private final LogRTPublisherConfig config;
    public LogRTPublisher(LogRTPublisherConfig config) throws AlertingException {
        this.config = config;
    }
    @Override
    public void onMatch(Context ctx, RTMatch match) throws AlertingException {
        if (ctx.isTesting()) {
          logger.info(config.getLogPrefix() + "#" + "onMatch - testing analysis pipel
        } else {
          logger.info(config.getLogPrefix() + "#" + "onMatch");
        }
    }
    @Override
    public void publish(RTAlertOccurrence occurrence) throws AlertingException {
        logger.info(config.getLogPrefix() + "#" + "publish");
    }
}
```

Custom Scheduled Publisher

Example 9. Publisher Configuration Class

```
import com.exalead.config.bean.PropertyLabel;
@PropertyLabel("Log Scheduled Publisher")
public class LogScheduledPublisherConfig extends CustomScheduledPublisherConfig {
    private String logPrefix;
    public String getLogPrefix() {
        return logPrefix;
    }
    public void setLogPrefix(String logPrefix) {
        this.logPrefix = logPrefix;
    }
}
```

Example 10. Publisher Class

```
import com.exalead.cloudview.alerting.v2.AlertingException;
import com.exalead.mercury.component.config.CVComponentConfigClass;
import com.google.common.base.Preconditions;
@CVComponentConfigClass(configClass = LogScheduledPublisherConfig.class)
public class LogScheduledPublisher extends CustomScheduledPublisher {
    private final static Logger logger = Logger.getLogger("alerting.v2.scheduled.log"
    private final LogScheduledPublisherConfig config;
    public LogScheduledPublisher(LogScheduledPublisherConfig config) throws AlertingE
        super();
```

```
        this.config = Preconditions.checkNotNull(config);
    }
    @Override
    public void begin() throws AlertingException {
        logger.info(config.getLogPrefix() + "#" + "Begin");
    }
    @Override
    public void beginUser(String user) throws AlertingException {
        logger.info(config.getLogPrefix() + "#" + "Begin user: " + user);
    }
    @Override
    public void publish(ScheduledAlertOccurrence occurrence) throws AlertingException
        logger.info(config.getLogPrefix() + "#" + "Publish occurrence of alert:
" + occurrence.getUnderlyingAlert().getName());
    }
    @Override
    public void endUser(String user) throws AlertingException {
        logger.info(config.getLogPrefix() + "#" + "End user: " + user);
    }
    @Override
    public void end() throws AlertingException {
        logger.info(config.getLogPrefix() + "#" + "End");
    }
}
```

Managing User Authentication in a Custom Application

If you do not use the Mashup UI to design your front-end application, you have to implement user authentication with the `securityClient` java client.

It is provided by the `CloudviewAPIClientsFactory` factory, which takes the gateway as parameter and can instantiate any client.

To retrieve the user tokens, you can use the `authenticate` method:

```
securityClient.authenticate(securitySource, login, password, checkUserPassword);
```