Apache UIMA Lucene CAS Indexer Documentation

Written and maintained by the Apache UIMA Development Community

Version 2.3.1

Copyright © 2006, 2011 The Apache Software Foundation

License and Disclaimer. The ASF licenses this documentation to you under the Apache License, Version 2.0 (the "License"); you may not use this documentation except in compliance with the License. You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, this documentation and its contents are distributed under the License on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Trademarks. All terms mentioned in the text that are known to be trademarks or service marks have been appropriately capitalized. Use of such terms in this book should not be regarded as affecting the validity of the the trademark or service mark.

Publication date August, 2011

Table of Contents

Introduction

The Lucene CAS Indexer (Lucas) is a UIMA CAS consumer that stores CAS data in a Lucene index. Lucas allows to exploit the results of collection processing for information retrieval purposes in a fast and flexible way. The consumer transforms annotation objects from annotation indexes into Lucene token objects and creates token streams from them. Token streams can be further processed by token filters before they are stored into a certain field of an index document. The mapping between UIMA annotations and Lucene tokens and token filtering is configurable by an XML file, whereas the index writer is configured by a properties file.

To use Lucas, at first a mapping file must be created. You have to decide which annotation types should be present in the index and how your index layout should look like, or more precisely, which fields should be contained in the index. Optionally you can add token filters for further processing. Its also possible to deploy your own token filters.

Lucas can run in multiple deployment scenarios where different instances share one index writer. This shared index writer instance is configured via a properties file and managed by the resource manager.

Chapter 1. Mapping Configuration

This chapter discusses the mapping between UIMA annotations and Lucene tokens in detail.

1.1. Token Sources

The mapping file describes the structure and contents of the generated Lucene index. Each CAS in a collection is mapped to a Lucene document. A Lucene document consists of fields, whereas a CAS contains multiple annotation indexes on different sofas. An annotation object can mark a text, can hold feature values or reference other feature structures. For instance, an annotation created by an entity mapper marks a text area and may additionally contain an identifier for the mapped entity. For this reason Lucas knows three different sources of Lucene token values:

- The covered text of a annotation object.
- One or more feature values of a annotation object.
- One or more feature values of a feature structure directly or indirectly referenced by an annotation object.

If a feature has multiple values, that means it references a FSArray instance, then one token is generated for each value. In the same manner tokens are generated from each feature value, if more then one feature is provided. Alternatively, you can provide a *featureValueDelimiterString* which is used to concatenate different feature values from one annotation object to generate only one token. Each generated Lucene token has the same offset as the source annotation feature structure.

1.1.1. Covered Text

As mentioned above, the text covered by annotation objects represents one possible source for Lucene token values. The following example creates an index with one *title* field which contains covered texts from all token annotations which are stored in the *title* sofa.

```
<fields>
  <field name=“title” index=“yes”>
     <annotations>
         <annotation sofa=“title” type=“de.julielab.types.Token”/>
     </annotations>
   </field>
</fields>
```

1.1.2. Feature Values

The feature values of annotation objects are another source for token values. Consider the example below.

```
<fields>
  <field name=“cells” index=“yes”>
     <annotations>
         <annotation sofa=“text” type=“de.julielab.types.Cell”>
             <features>
                <feature name=“specificType”>
             </features>
         </annotation>
     </annotations>
```

```
   </field>
</fields>
```

The field *cells* contains a token stream generated from the annotation index of type *de.julielab.types.Cell* . Each generated token will contain the value of the feature *specificType* of the enclosing annotation object.

The next example illustrates how multiple feature values can be combined by using a *featureValueDelimiterString* . If no *featureValueDelimiterString* is provided, a single token is generated from each feature value.

```
<fields>
  <field name=“authors” index=“no” stored="yes">
     <annotations>
         <annotation sofa=“text” type=“de.julielab.types.Author”
                                featureValueDelimiterString=", ">
              <features>
                <feature name=“firstname”/>
                <feature name=“lastname”/>
             </features>
         </annotation>
     </annotations>
   </field>
</fields>
```

1.1.3. Feature Values of referenced Feature Structures

Since annotation objects may reference other feature structures, it may be desirable to use these feature structures as source for Lucene token values. To achieve this, we utilize feature paths to address these feature structures. Consider the example below.

```
<fields>
  <field name=“cities” index=“yes”>
     <annotations>
         <annotation sofa=“text” type=“de.julielab.types.Author”
                                featurePath="affiliation.address">
             <features>
                <feature name=“city”>
             </features>
         </annotation>
     </annotations>
   </field>
</fields>
```

The type *de.julielab.types.Author* has a feature *affiliation* which points to a *affiliation* feature structure. This *affiliation* feature structure in turn has a feature *address* which points to an *address* feature structure. This path of references is expressed as the feature path *affiliation.address* . A feature path consists of feature names separated by colons (":"). Please consider that the *city* feature is a feature of the "address" feature structure and not of the *author* annotation object.

1.1.4. Supported feature types

Currently, not all feature types are supported. Supported feature types are the following:

- String
- String Array

- Number Types: Double, Float, Long, Integer, Short

Consider that you need to provide a number format string if you want to use number types.

1.2. Token Stream Alignment

In the examples above all defined Lucene fields contain only one annotation based token stream. There are a couple of reasons for the fact that the simple mapping of each annotation index to separate Lucene fields is not an optimal strategy. One practical reason is that the Lucene highlighting will not work for scenarios where more than one annotation type is involved. Additionally, the TF-IDF weighting of terms does not work properly if annotations are separated from their corresponding text fragments. Lucas is able to merge token streams and align them according to their token offsets. The resulting merged token stream is then stored in a field. The next example demonstrates this merging feature.

```
<fields>
  <field name=“text” index=“yes” merge=“true”>
     <annotations>
         <annotation sofa=“text” type=“de.julielab.types.Token”/>
         <annotation sofa=“text” type=“de.julielab.types.Cell”>
         <features>
                <feature name=“specificType”>
             </features>
         </annotation>
     </annotations>
  </field>
</fields>
```

Consider the merge attribute of the field tag. It causes the alignment of the two token streams generated from the *de.julielab.types.Token* and *de.julielab.types.Cell* annotations. If this attribute is set to false or is left out, the annotation token streams are concatenated.

1.3. Token Filters

Token filters are the Lucene approach to enable operations on token streams. In typical Lucene applications token filters are combined with a tokenizer to build analyzers. In a typical Lucas application the tokenization is already given by annotation indexes. Lucas allows to apply token filters to certain annotation token streams or to the merged or concatenated field token stream as whole. The following example demonstrates how token filters are defined in the mapping file.

```
<fields>
  <field name=“text” index=“yes” merge=“true”>
   <filters>
  <filter name="lowercase"/>
 </filters>
 <annotations>
  <annotation sofa=“text” type=“de.julielab.types.Token”>
   <filters>
    <filter name="stopwords"
                  filePath="resources/stopwords.txt"/>
   </filters>
  </annotation>
  <annotation sofa=“text” type=“de.julielab.types.Cell”>
   <features>
    <feature name=“specificType”>
   </features>
```

```
  </annotation>
 </annotations>
  </field>
</fields>
```

The lowercase token filter is applied to the complete field content and the stopword filter is only applied to the annotation token stream which is generated from the de.julielab.types.Token annotation index. Both filters are predefined filters which are included in the Lucas distribution. A reference of all predefined token filters is covered in Chapter 2, *Mapping File Reference* [7] .

1.3.1. Deploying your own Token Filters

For scenarios where the built in token filters where not sufficient, you can provide your own token filters. Simple token filters which do not need any further parameterization, are required to define a public constructor, which takes a token stream as the only parameter. The next example shows how a such a token filter is referenced in the mapping file.

```
<fields>
  <field name=“text” index=“yes”>
 <annotations>
  <annotation sofa=“text” type=“de.julielab.types.Cell”>
   <filters>
    <filter className="org.example.MyFilter"/>
   </filters>
   <features>
    <feature name=“specificType”>
   </features>
  </annotation>
 </annotations>
  </field>
</fields>
```

The attribute *className* must reference the canonical class name of the the filter. In cases where the token filter has parameters we need to provide a factory for it. This factory must implement the *org.apache.uima.indexer.analysis.TokenFilterFactory* interface. This interface defines a method createTokenFilter which takes a token stream and a java.util.Properties object as parameters. The properties object will include all attribute names as keys and their values which are additionally defined in the filter tag. Consider the example below for a demonstration.

```
<fields>
  <field name=“text” index=“yes”>
 <annotations>
  <annotation sofa=“text” type=“de.julielab.types.Cell”>
   <filters>
    <filter factoryClassName="org.example.MyTokenFilterFactory"
                parameter1="value1" parameter2="value2"/>
   </filters>
   <features>
    <feature name=“specificType”>
   </features>
  </annotation>
 </annotations>
  </field>
</fields>
```

In the example above the token filter factory is new instantiated for every occurrence in the mapping file. In scenarios where token filters use large resources, this will be a waste of memory

and time. To reuse a factory instance we need to provide a name and a reuse attribute. The example below demonstrate how we can reuse a factory instance.

```
<fields>
  <field name=“text” index=“yes”>
 <annotations>
  <annotation sofa=“text” type=“de.julielab.types.Cell”>
   <filters>
    <filter factoryClassName="org.example.MyTokenFilterFactory"
                  name="myFactory" reuse="true"
                  myResourceFilePath="pathToResource"/>
   </filters>
   <features>
    <feature name=“specificType”>
   </features>
  </annotation>
     </annotations>
  </field>
</fields>
```

Chapter 2. Mapping File Reference

After introducing the basic concepts and functions this chapter offers a complete reference of the mapping file elements.

2.1. Mapping File Structure

The raw mapping file structure is sketched below.

```
<fields>
  <field ..>
 <filters>
  <filter ../>
  ...
 </filters>

 <annotations>
  <annotation ..>
   <filters>
    <filter ../>
    ...
   </filters>
   <features>
    <feature ..>
    ...
   </features>
  </annotation>
  ...
 </annotations>
  </field>
  ...
</fields>
```

2.2. Mapping File Elements

This section describes the mapping file elements and their attributes.

- *fields element*
  - fields container element
  - contains: `field+`
- *field element*
  - describes a Lucene field[1]
  - contains: `filters?, annotations`

Table 2.1. field element attributes

name	allowed values	default value	mandatory	description
name	string	-	yes	the name of the field[2]

name	allowed values	default value	mandatory	description
index	yes|no| no_norms|no_tf| no_norms_tf	no	no	See Field.Index[3]
termVector	no|positions| offsets| positions_offsets	no	no	See Field.TermVector[4]
stored	yes|no|compress	no	no	See Field.Store[5]
merge	boolean	false	no	If this attribute is set to true, all contained annotation token streams are merged according to their offset. The tokens position increment are adopted in the case of overlapping.
unique	boolean	false	no	If this attribute is set to true, there will be only one field instance with this field's name be added to resulting Lucene documents. This is required e.g. by Apache Solr for primary key fields. You must not define multiple fields with the same name to be unique, this would break the unique property.

- *filters element*
  - container element for filters
  - contains: `filter+`
- *filter element*

- Describes a token filter[6] instance. Token filters can either be predefined or self-provided.

Table 2.2. filter element attributes

name	allowed values	default value	mandatory	description
name	string	-	no	the name to reference either a predefined filter (see predefined filter reference) or a reused filter
className	string	-	no	The canonical class name of a token filter. the token filter class must provide a single argument constructor which takes the token stream as parameter.
factoryClassName	string	-	no	The canonical class name of a token filter factory. the token filter factory class must implement the org.apache.uima.indexer.anal See Section 1.3, “ Token Filters ” [3] for a example.
reuse	boolean	-	false	Enables token filter factory reuse. This makes sense when a token filter uses resources which should be cached. Because token filters are referenced by their names,

name	allowed values	default value	mandatory	description
				you also need to provide a name.
*	string	-	-	Filters may have their own parameter attributes which are explained in the Chapter 2, *Mapping File Reference* [7] ..

- *annotations element*
  - container element for annotations
  - contains: `annotation+`
- *annotation element*
  - Describes a token stream which is generated from a CAS annotation index.
  - contains: `features?`

Table 2.3. annotation element attributes

name	allowed values	default value	mandatory	description
type	string	-	yes	The canonical type name. E.g. "uima.cas.Annotation"
sofa	string	InitialView	yes	Determines from which sofa the annotation index is taken
featurePath	string	-	no	Allows to address feature structures which are associated with the annotation object. Features are separated by a ".".
tokenizer	cas|white_space|standard	cas	no	Determines which tokenization is used. "cas" uses the tokenization given by the

name	allowed values	default value	mandatory	description
				contained annotation token streams, "standard" uses the standard tokenizer[7]
featureValueDelimiterString	string	-	no	If this parameter is provided all feature values of the targeted feature structure are concatenated and delimited by this string.

- *features element*
  - Container element for features.
  - contains: `feature+`
- *feature element*
  - Describes a certain feature of the addressed feature structure. Values of this features serve as token source.

Table 2.4. feature element attributes

name	allowed values	default value	mandatory	description
name	string	-	yes	The feature name.
numberFormat	string	-	no	Allows to convert number features to strings. See DecimalNumberFormat[8] .

2.3. Filters Reference

Lucas comes with a couple of predefined token filters. This section provides a complete reference for these filters.

2.3.1. Addition Filter

Adds suffixes or prefixes to tokens.

```
<filter name="addition" prefix="PRE_"/>
```

Table 2.5. addition filter attributes

name	allowed values	default value	mandatory	description
prefix	string	-	no	A pefix which is added to the front of each token.
postfix	string	-	no	A post which is added to the end of each token.

2.3.2. Hypernyms Filter

Adds hypernyms of a token with the same offset and position increment 0.

```
<filter name="hypernyms" filePath="/path/to/myFile.txt"/>
```

Table 2.6. hypernym filter attributes

name	allowed values	default value	mandatory	description
filePath	string	-	yes	The hypernym file path. Each line of the file contains one token with its hypernyms. The file must have the following format: `TOKEN_TEXT=HYPERNYM1|HYPERNYM2|...` .

2.3.3. Position Filter

Allows to select only the first or the last token of a token stream, all other tokens are discarded.

```
<filter name="position" position="last"/>
```

Table 2.7. position filter attributes

name	allowed values	default value	mandatory	description
position	first|last	-	yes	If position is set to first the only the the first token of the underlying token stream is returned, all other tokens are discarded. Otherwise, if

name	allowed values	default value	mandatory	description
				position is set to last, only the last token is returned.

2.3.4. Replace Filter

Allows to replace token texts.

```
<filter name="replace" filePath="/path/to/myFile.txt"/>
```

Table 2.8. replace filter attributes

name	allowed values	default value	mandatory	description
filePath	string	-	yes	The token text replacement file path. Each line consists of the original token text and the replacement and must have the following format: `TOKEN_TEXT=REPLACEMENT_` .

2.3.5. Snowball Filter

Integration of the Lucene snowball filter[9]

```
<filter name="snowball" stemmerName="German"/>
```

Table 2.9. snowball filter attributes

name	allowed values	default value	mandatory	description
stemmerName	snowball stemmer names.	English	no	See snowball filter documentation[10] .

2.3.6. Splitter Filter

Splits tokens at a certain string.

```
<filter name="splitter" splitString=","/>
```

[9] http://lucene.apache.org/java/2_4_0/api/org/apache/lucene/analysis/snowball/SnowballFilter.html

Table 2.10. splitter filter attributes

name	allowed values	default value	mandatory	description
splitString	string	-	yes	The string on which tokens are split.

2.3.7. Concatenate Filter

Concatenates token texts with a certain delimiter string.

```
<filter name="concatenate" concatString=";"/>
```

Table 2.11. concatenate filter attributes

name	allowed values	default value	mandatory	description
concatString	string	-	yes	The string with which token texts are concatenated.

2.3.8. Stopword Filter

Integration of the Lucene stop filter[11]

```
<filter name="stopwords" filePath="/path/to/myStopwords.txt"/>
```

Table 2.12. stopword filter attributes

name	allowed values	default value	mandatory	description
filePath	string	-	no	The stopword file path. Each line of the file contains a single stopword.
ignoreCase	boolean	false	no	Defines if the stop filter ignores the case of stop words.

2.3.9. Unique Filter

Filters tokens with the same token text. The resulting token stream contains only tokens with unique texts.

```
<filter name="unique"/>
```

2.3.10. Upper Case Filter

Turns the text of each token into upper case.

[11] http://lucene.apache.org/java/2_4_1/api/org/apache/lucene/analysis/StopFilter.html

```
<filter name="uppercase"/>
```

2.3.11. Lower Case Filter

Turns the text of each token into lower case.

```
<filter name="lowercase"/>
```

Chapter 3. Index Writer Configuration

The index writer used by Lucas can be configured separately. To allow Lucas to run in multiple deployment scenarios, different Lucas instances can share one index writer instance. This is handled by the resource manager. To configure the resource manager and the index writer properly, the Lucas descriptor contains a resource binding `IndexWriterProvider` . An `IndexWriterProvider` creates an index writer from a properties file. The file path and the name of this properties file must be set in the `LucasIndexWriterProvider` resource section of the descriptor.

The properties file can contain the following properties.

- `indexPath` - the path to the index directory
- `RAMBufferSize` - (number value), see IndexWriter.ramBufferSize[1]
- `useCompoundFileFormat` - (boolean value), see IndexWriter.useCompoundFormat[2]
- `maxFieldLength` - (boolean value), see IndexWriter.maxFieldLength[3]
- `uniqueIndex` - (boolean value), if set to `true` , host name and process identifier are added to the index name. (Only tested on linux systems)

Chapter 4. Descriptor Parameters

Because Lucas is configured by the mapping file, the descriptor has only one parameter:

- `mappingFile` - the file path to the mapping file.

Chapter 5. Prospective Search

Prospective search is a search method where a set of search queries are given first and then searched against a stream of documents. A search query divides the document stream into a sub-stream which only contains these document which match the query. Users usually define a number of search queries and then subscribe to the resulting sub-streams. An example for prospective search is a news feed which is monitored for certain terms, each time a term occurs a mail notification is send.

The user must provide a set of search queries via a `SearchQueryProvider` , these search queries are then search against the processed CAS as defined in the mapping file, if a match occurs a feature structure is inserted into the CAS. Optionally highlighting is supported, annotations for the matchtng text areas are created and linked to the feature structure.

The implementation uses the Lucene MemoryIndex[1] which is a fast one document in memory index. For performance notes please consult the javadoc of the `MemoryIndex` class.

5.1. Search Query Provider

The Search Query Provider provides the Perspective Search Analysis Engine with a set of search queries which should be monitored. A search query is a combination of a Lucene query and an id. The id is later needed to map a match to a specific search query. A user usually has a set of search queries which should be monitored, since there is no standardized way to access the search queries the user must implement the `SearchQueryProvider` interface and configure the thread-safe implementation as shared resource object. An example for such an implementation could be a search query provider which reads search queries form a database or a web service.

5.2. Search Results

The search results are written to the CAS, for each match one Search Result feature structure is inserted into the CAS. The Search Result feature structure contains the id and optionally links to an array of annotations which mark the matching text in the CAS.

The Search Result type must be mapped to a defined type in the analysis engine descriptor with the following configuration parameters:

- `String org.apache.uima.lucas.SearchResult` - Maps the search result type to an actual type in the type system.
- `String org.apache.uima.lucas.SearchResultIdFeature` - A long id feature, identifies the matching search query.
- `String org.apache.uima.lucas.SearchResulMatchingTextFeature` - An optional `ArrayFS` feature, links to annotations which mark the matching text.

[1] http://lucene.apache.org/java/2_4_1/api/contrib-memory/org/apache/lucene/index/memory/MemoryIndex.html