GenericXMLModules

User's Guide

Florian Zipser <saltnpepper@lists.hu-berlin.de>
INRIA
SFB 632 Information Structure / D1 Linguistic Database
Humboldt-Universität zu Berlin
Universität Potsdam

GenericXMLModules: User's Guide

by Florian Zipser, , , , and

Version ${project.version}

Table of Contents

List of Tables

Foreword

The intention of this document is first to give a guide to the user of how to use the here mentioned pepper modules and how to utilize a mapping performed by them. Second this document shall give a closer view of the details of such a mapping in a declarative way, to give the user a chance to understand how specific data will be mapped by the presented pepper modules.

Chapter 1. Overview

This project contains the pepper modules listed in Table 1.1, “pepper modules contained in this project”. A single module can be identified via its coordinates (module-name, format-name, format-version) also given in Table 1.1, “pepper modules contained in this project”. You can use these coordinates in a pepper workflow description file to identify the modules in a pepper conversion process. A description of how to model a workflow description file can be found under https://korpling.german.hu-berlin.de/saltnpepper/.

Table 1.1. pepper modules contained in this project

Name of pepper module	Type of pepper module	Format (if module is im- or exporter)
GenericXMLImporter	importer	1.0

Chapter 2. GenericXMLImporter

The GenericXMLImporter data coming from any xml file to a Salt model in a customizable but generic way. This importer provides a wide range of customizing possibilities via the here described set of properties. Before we talk about the possibility of customizing the mapping, we describe the general and default mapping from an xml file to a Salt model.

Mapping to Salt

The GenericXMLImporter maps element-nodes, text-nodes and attribute-nodes to Salt model objects. Comment-nodes, processing-instruction-nodes etc. will be ignored and not be mapped.

The textual value of all text-nodes of the xml document will be concatenated to one single primary text. This primary text is stored in a `STextualDS` object and can be accessed via the method `STextualDS.getSText()`. Each element-node containing a primary text and no further element-nodes (called terminal-node) is mapped to a `SToken` object overlapping the part of the primary data given by the contained text-node. To realize the offset to the start and end position of the overlapping text, a `STextualRelation` is created connecting the `STextualDS` and the `SToken` object.

The following xml fragment

```
<a>This is</a><b> a sample text</a>
```

is mapped to a `STexualDS` object having the `sText` "This is a sample text" and two `SToken` objects, one overlapping the text "This is" and one overlapping the text " a sample text".

Note

The importer does not take care about the given tokenization and does not retokenize it.

A element-node not containing further element-nodes (called complex-node) is mapped to a `SStructure` object in the Salt model. Since `SStructure` objects in Salt represent a hierarchical structure, a `SStructure` object is connected to `SNode` objects corresponding to the element-nodes in the first level of the subtree of the element-node.

The following xml fragment

```
<a><b>a sample</b></a>
```

is mapped to a `SToken` (corresponding to the element-node <a>) object overlapping the primary text "a sample" and a `SStructure` (corresponding to the element-node <b>) object dominating the SToken object. Both nodes are connected with a `SDominanceRelation` having the `SStructure` object as source and the `SToken` object as target. Since `SStructure` objects are used for hierarchies, the same goes for an element-node containing another element node.

The following xml fragment

```
<a><b>...</b></a>
```

is mapped to a `SStructure` object representing the element-node <a> and dominating a further `SStructure` object representing the element-node <b>.

In many xml formats element-nodes can have further element-nodes and text-nodes in the first level of their subtree as well. These kind of nodes are often called mixed nodes (for nodes having a mixed content). These kind of nodes are mapped to a `SStructure` object in Salt. The element-nodes contained in their subtree are mapped to either `SToken` objects or `SStructure` objects, depending on their content. For a text-node, an artificial `SToken` object is created and added to the subtree of the `SStructure` object in Salt.

The following xml fragment

```
<a>This is <!-- t1 --><b>a sample<!-- t2 --></b> text<!-- t3 --></a>
```

is mapped to three `SToken` objects t1 overlapping the text "This is ", t2 overlapping the text "a sample" and t3 overlapping the text " text". Because the `SToken` object t2 is the only one having an existing correspondance to a terminal-node, two artificial `SToken` objects t1 and t3 are created. The complex-node <a> is mapped to a `SStructure` object dominating the three `SToken` objects in the order t1, t2 and t3. This mechanism is recursive and will also work for the following xml fragment

```
<a>This is <b><c>a sample</c></b> text</a>
```

, for which a further `SStructure` object corresponding the complex-node <b> is created and is dominating the `SToken` object corresponding to the terminal-node <c>.

Attribute-nodes are mapped to `SAnnotation` objects, having the attribute-name as `SName` and the attribute-value as `SValue`. Such an `SAnnotation` object is added to the list of `SAnnotation` of a container `SNode` object.

The following xml fragment

```
<a att="value">a sample</a>
```

is mapped to a `SToken` overlapping the text "a sample" and containing a `SAnnotation` object having the `SName` "att" and the `SValue` "value". The same goes for complex-nodes and mixed-nodes.

Properties

The table Table 2.4, “properties to customize importer behaviour” contains an overview of all usable properties to customize the behaviour of this pepper module. The following section contains a brief description to each single property and describes the resulting differences in the mapping to the salt model.

Some of the here described properties use for their values a small subset of the XPath language for addressing nodes in the xml document to import. This subset contains possibilities to address element-nodes, text-nodes and attribute-nodes in just a simple way via following the descendant axis. The descendant axis can only be used by the shortcut syntax represented by a '/' for a direct descendant and '//' for any descendants. The other axes and predicates as well are not yet supported. The following tables show the use of the supported XPath subset.

Table 2.1. support for element-nodes

XPath expression	description
/element	addresses the xml element-node having the name element and which is the root node
//element	addresses the xml element-node having the name element anywhere in the document
//element//	addresses all xml element-nodes in the subtree of each element-node having the name element
/father/element	addresses the xml element-node having the name element and its subtree, which is a direct descendant of an element-node having the name father

Table 2.2. support for attribute-nodes

XPath expression	description
//@attribute	addresses the xml attribute-node having the name attribute anywhere in the document

XPath expression	description
//element/ @attribute	addresses the xml attribute-node having the name attribute and belongs to the xml element-node having the name element which is a direct descendant of an element-node named father

Table 2.3. support for text-nodes

XPath expression	description
//text()	addresses every xml text-node anywhere in the document
//element/ text()	addresses every xml text-node belonging to the xml element-node having the name element which is a direct descendant of an element-node named father

For some properties it is possible, to not only address one element-node, text-node or attribute-node, but to address a set of nodes. For such cases, you can separate XPath expressions by using the ',' character. For instance:

```
//element1/text(), //element2/text()
```

and so on. The size of such a set is unbound.

Table 2.4. properties to customize importer behaviour

Name of property	Type of property	optional/ mandatory	default value
genericXml.importer.ignoreList	XPath expression of described subset	optional	--
genericXml.importer.asSSpan	XPath expression of described subset	optional	--
genericXml.importer.prefixSAnnotationName	XPath expression of described subset	optional	--
genericXml.importer.artificialSStruct	Boolean	optional	false
genericXml.importer.sMetaAnnotation	XPath expression of described subset	optional	--
genericXml.importer.sMetaAnnotation.sDocument	XPath expression of described subset	optional	--
genericXml.importer.textOnly	Boolean	optional	false
genericXml.importer.sLayer	XPath expression of described subset	optional	--
genericXml.importer.ignoreWhitespaces	String	optional	'\n','\r','\t',' '
genericXml.importer.ignoreNamespaces	Boolean	optional	true
genericXml.importer.file.endings	XPath expression of described subset (only element-nodes are addressable)	optional	--
genericXml.importer.elementNameAsSAnno	String	optional	xml

genericXml.importer.ignoreList

The ignore list is a list of nodes (element-nodes , attribute-nodes and text-nodes) which are ignored for the mapping to a Salt model. Imagine for instance the follwing xml fragment

```
<a><b></b></a>
```

and the property value of ignore-list //b. This list enables, that the element-node <b> will completly be ignored.

Here we give a sample of the usage of the ignore list:

```
genericXml.importer.ignoreList= /a/b, //c
```

To ignore an entire subtree, use the subtree address mechanism e.g.

```
genericXml.importer.ignoreList= /a//
```

To ignore all nodes in the subtree of 'a'.

Note

In this example, only the subtree of element-node 'a' is ignored, not 'a' itself.

genericXml.importer.asSSpan

In case you do not want to map an element node to a `SStructure` object, you can also map defined element-nodes to an `SSpan` object. `SSpan` in contrast to `SStructure` objects are not hierarchical and are used to create span-like structures, similar to cells in a table.

Here we give a sample of the usage of this property:

```
genericXml.importer.asSSpans= //element1, //father/element2
```

genericXml.importer.prefixSAnnotationName

In general the `SName` of a `SAnnotation` is given by the name of the attribute-node given in the xml document. sometimes it might be necessary, to remember the name of the element-node containing the attribute-node. In such cases, you can set this property to prefix the `SName` of the `SAnnotation` with the name of the surrounding element-node. The following xml fragment

```
<a att="value"/>
```

will is mapped to a `SAnnotation` object having the `SName` 'att'. When setting the property as shown in the following sample,

```
genericXml.importer.prefixSAnnotation=//a/@att
```

genericXml.importer.artificialSStruct

In general a terminal-node is mapped to a `SToken` object. This is necessary, because in Salt only `SToken` objects can overlap parts of the primary data (given by `STextualDS` objects). Sometimes, one may want to map the terminal-node also to a `SSpan` or `SStructure` node. Using this property will result in an artificial node dominating or spanning the `SToken` object.

The xml fragment

```
<a>a sample</a>
```

when using the property

```
genericXml.importer.artificialSStruct= true
```

results in a `SToken` object overlapping the primary data "a sample" and a `SStructure` object dominating the `SToken` object.

Note

When using this flag, an artificial SSpan can be created instead of an SStructure object, when adding an XPath expression addressing the terminal-node to the set of expressions of the property genericXml.importer.asSSpan

genericXml.importer.sMetaAnnotation

Usually, an attribute-node is mapped to `SAnnotation` object. But sometimes you may want to map it to a `SMetaAnnotation` object instead. A `SMetaAnnotation` in Salt marks an attribute-value pair to be not directly a linguistic annotation, and therefore not processed or exported as one. Often such annotations are used to mark the annotator of an annotation, or a probability of an annotation and so on.

The xml fragment

```
<a att="value"/>
```

using the property

```
genericXml.importer.sMetaAnnotation=//a/@att
```

will result in a a `SNode` object representing the element-node <a> having a `SMetaAnnotation` object with the `SName` 'att' and the `SValue` "value".

genericXml.importer.sMetaAnnotation.sDocument

Xml documents often also contain sections for meta-data of the entire document, for instance the name of the author of the document, the year of creation, or the mothers tongue of the author etc.. For dealing with such a case, you can use this flag, to mark an element or an entire subtree as a meta-data section. Each attribute-node of such an element or subtree is mapped to a `SMetaAnnotation` object having its name as `SName` and its value as `SValue` and is added to the list of meta-data of the `SDocument`.

The following xml fragment

```
<a att_2="value"><b att_1="value"/></a>
```

when using the property

```
genericXml.importer.sMetaAnnotation.sDocument=//a
```

results in a `SDocument` object containing two `SMetaAnnotation` objects having the `SName` 'att_1' and the `SValue` 'value' or 'att_2' and 'value'.

Note

When more than one attribute-nodes have the same name, the mapper will add an '_' and a number to their name, because in Salt an `SDocument` cannot have two `SMetaAnnotation` objects having the same `SName`. To avoid this behaviour, you can use the property genericXml.importer.prefixSAnnotationName to concatenate the name of the element-node with the name of the attribute-node.

To interprete an entire subtree of an xml element as a meta-data containing subtree, use the wildcard notation at the end of your XPath expression.

The following xml fragment

```
<a>
    <b att_1="value">
```

```
        <c att2="value"/>
    </b>
    <d>a sample text</d>
</a>
```

when using the property

```
genericXml.importer.sMetaAnnotation.sDocument=//b, //b//
```

results in one `SToken` object overlapping the primary text 'a sample text' being dominated by a `SStructure` object corresponing to element <a>. The xml-element <b> and its entire subtree is interpreted as meta-data for the `SDocument` object.

You can also just address attribute-nodes. In that case only the attribute nodes are mapped to a `SMetaAnnotation` object. If an attribute-node is mapped to a `SMetaAnnotation` object, it is not mapped to a usual `SAnnotation` object. The following xml fragment

```
<a>
    <b att_1="value">
        <c>a sample text</c>
    </b>
</a>
```

when using the property

```
genericXml.importer.sMetaAnnotation.sDocument=//b/@att_1
```

results in a `SMetaAnnotation` object added to the `SDocument` object.

genericXml.importer.textOnly

When using this flag, only the text is imported as primary data. That means all text-nodes will be concatenated to a single text. A `STextualDS` object is created and its `SText` will be set to the concatenated text.

genericXml.importer.sLayer

In Salt, `SLayer`s are important to separate several kinds of linguistic annotations. For instance to separate a syntax analysis from a dialogue analysis etc.. To identify such a layer in a xml file you can use this property. An addressed element-node is mapped to a `SLayer` object, its attribute-nodes are mapped to `SAnnotation` objects. In a lot of cases an XPath expression addresses a set of element-nodes. All these nodes having the same name, are mapped to the same `SLayer` object. All element-nodes of the the subtree of an addressed element-node are added to the corresponding `SLayer` object. Layers are recursive objects, therefore layers can contain other layers.

The following xml fragment

```
<a att="value"><b>a sample <c>text</c></b></a>
```

with the use of the property

```
genericXml.importer.sLayer=//a, //c
```

results in two `SToken` t1 and t2 objects, overlapping the text 'a sample' (t1) and 'text' (2). The `SToken` object t2 is part of a created `SLayer` object l1 corresponding to the element-node <c>. A `SStructure` object is created containing the `SToken` objects t1 and t2. This `SStructure` object and the `SToken` t1 are not part of the created layer. A second layer l2 is created for the element-node <a>. An `SAnnotation` object having the `SName` 'att' and the `SValue` 'value' is added to l2. All created `SNode` objects and the SLayer object l2 are contained in the layer l2.

genericXml.importer.ignoreWhitepsaces

Determines a list of whitespace characters, which are igrnored, if an text-node only contains characters of this list. This is a necessary property, to avoid for instance the tokenization of blanks.

The following xml fragment

```
<document><a>a</a> <b>sample</b> <c>text</c></document>
```

with the use of the property

```
genericXml.importer.ignoreWhitepsaces='\n','\r','\t',' '
```

results in three `SToken` objects overlapping the text 'a', 'sample' and 'text' instead of 5 `SToken` objects additionally overlapping the blanks ' '. A sequence of characters will only be ignored, if it only contains characters being contained in that list. Therefore, following xml fragment

```
<a>a sample</a>
```

with the use of the property

```
genericXml.importer.ignoreWhitepsaces='\n','\r','\t',' '
```

results in one `SToken` overlapping the text 'a sample' including th blank.

genericXml.importer.ignoreNamespaces

Determines that all xml namespace declarations and xml namespace prefixes are ignored.

The following xml fragment

```
<document xmlns="some namespace" xmlns:ns="some namespace">
    <ns:a ns:att="val"/>
</document>
```

with the use of the property

```
genericXml.importer.ignoreNamespaces=true
```

results in a `SStructure` object representing the element-node 'document' having no `SAnnotation` objects. A further `SStructure` object representing the element-node 'a' is created having the `sName` 'a' and a `SAnnotation` object hving the `sName` 'att'.

The same fragment with the use of the property

```
genericXml.importer.ignoreNamespaces=false
```

results in a `SStructure` object representing the element-node 'document' having two `SAnnotation` object. First with `sName` 'xmlns' and `sValue` 'some namespace' and second with `sName` 'xmlns:ns' and `sValue` 'some namespace'. Same goes for the `SStructure` representing the element-node 'a'.

genericXml.importer.elementNameAsSAnno

Determines a list, of element-nodes which names are mapped to artificial `SAnnotation` objects adding to the corresponding `SNode` object.The following xml fragment

```
<a>
    <b/>
</a>
```

with the use of the property

```
genericXml.importer.elementNameAsSAnno=//a, //a//
```

results in a one `SNode` object representing the element-node 'a' which is annotated with a `SAnnotation` whichs `sName` and `sValue` is 'a'. The same goes for the second `SNode` object which gets an annotation b=b.

genericXml.importer.file.endings

Determines a list, containing the file endings, which files are imported. If you want to import all contained files no matter of their ending, add the string 'ALL' to the list. If you want a file having a specific ending not to be imported, use the prefix '-'.