SaltNPepper

The Pepper manual

version 1.0 -

title:	The Pepper manual
version:	1.1.3
date:	22. December 2011
author:	Florian Zipser
eMail:	saltnpepper@lists.hu-berlin.de
homepage:	https://korpling.german.hu-berlin.de/saltnpepper/

1 Overview

Pepper is a pluggable, java-based, open-source[1] converter framework for linguistic data. It was developed to convert data coming from a linguistic data format *X* to another linguistic data format *Y*. To decrease the number of conceptual mappings, pepper follows the intermediate model approach, which means that a conversion consists of two mappings. First, the data coming from format *X* will be mapped to the intermediate model and second, the data will be mapped from the intermediate model to format *Y*. If you imagine a set of *n* source and target formats, then this approach will decrease the number of mappings from n^2-n mappings in case of the direct mapping approach to *2n* mappings. The intermediate model used here is the linguistic meta-model Salt (see http://korpling.german.hu-berlin.de/saltnpepper/).

Since Pepper is just a conversion framework and only takes the workflow control, the real conversion work is done by a set of Pepper modules. Such a module is an individual unit executing a specific task, like mapping data from or to a linguistic data format. A Pepper module can simply be plugged into the Pepper framework as described in the section Plug in PepperModules. Figure 1 shows the pluggable architecture of Pepper.

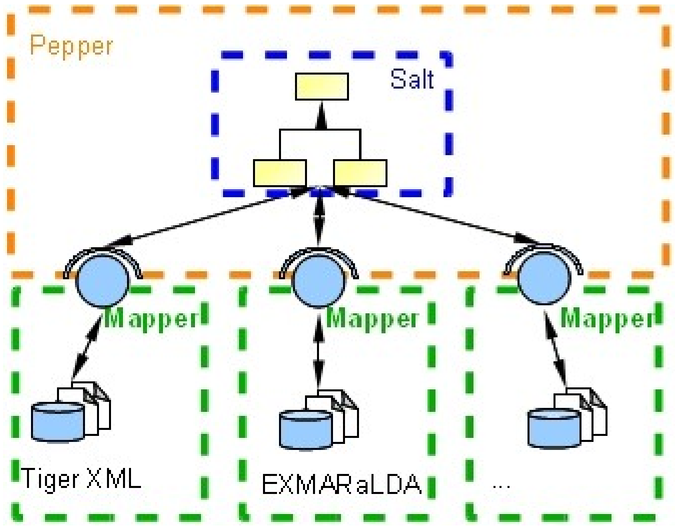


Figure 1: the plug-in architecture of pepper and the relation to pepper modules represented as mappers using the intermediate model Salt

The PepperWorkflow is separated into three different phases: 1) the import phase, 2) the export phase and 3) the manipulation phase. The import phase handles the mapping from a format *X* to the Salt model, the export phase handles the mapping from a Salt model to a format *Y* and during the manipulation phase the data in a Salt model can be enhanced, reduced or manipulated. A phase is divided into several steps: the import and export phase each contain *1* to *n* steps whereas the manipulation phase contains *0* to *n* steps. Each PepperModule realizes such a step and therefore is associated with exactly one phase.

1 Apache 2.0 License

The orchestration of PepperModules is determined by the PepperWorkflow description file, described in the section Modeling workflow description.

In the following we describe the installation of Pepper (see the section Installing Pepper), its usage (see Running Pepper), the workflow modeling (see Modeling workflow description) and the utilization of the Pepper framework (see Utilizing Pepper).

2 Installing Pepper

Pepper is a java-based framework, therefore it is not necessary to provide a different instance for several operating systems. All you need to run Pepper is a working JRE (Java Runtime Environment) in a minimum of version 1.6. To check if you have this prerequisite just open a command line and run:

```
java -version
```

When downloading the binary version of Pepper from http://korpling.german.hu-berlin.de/saltnpepper/ it is not necessary to install Pepper. Just unzip the downloaded file

```
SaltNPepper_XXX.zip.
```

to a folder of your choice (let's call it PEPPER_HOME) and Pepper is ready to run. When you have downloaded a complete version of SaltNPepper, a set of PepperModules is already included in the distribution. In some cases it is necessary to plug in modules that are not included or to update an included PepperModule. A guide about how to plug in modules can be found in the section Plug in PepperModules.

3 Running Pepper

Pepper is a command line program and can be invoked via a simple command line call. Just switch to the folder where you have unzipped Pepper (let's call it PEPEPR_HOME) and call:

```
pepperStart.bat                (when using Windows)
```

or

```
bash pepperStart.sh            (when using linux, unix or Mac OS)
```

Pepper will start and log an error message saying that no workflow description is given. To pass a workflow description as described in the section Modeling workflow description to the program, just use the parameter -w and determine the workflow description file:

```
    pepperStarter.bat -w WF_FILE
```

, where WF_FILE is the path to the workflow description file. The parameters are the same for different operating systems.

4 Modeling workflow description

A PepperWorkflow description can be modeled and persisted in an xml file following a specific notation and having the extension ending '*.pepperparams*'. As already mentioned, a Pepper conversion process consists of three phases, and the notation of a workflow description file follows this structure. To identify a PepperModule realizing a step, you have to declare that module by triggering its name. Note, that for each PepperWorkflow description, the specification of an importer and an exporter is necessary, whereas a manipulator is optional. Example 1 shows an excerpt of a PepperWorkflow description file using all types of modules.

```
<?xml version="1.0" encoding="UTF-8"?>
<pepperParams:PepperParams xmlns:xmi="http://www.omg.org/XMI"
xmlns:pepperParams="de.hu_berlin.german.korpling.saltnpepper.pepper.pepperPa
rams" xmi:version="2.0">
    <pepperJobParams id="1">
        <importerParams moduleName="ImporterName" sourcePath="..."/>
        <!-- ... -->
        <moduleParams moduleName="ManipulatorName"/>
        <!-- ... -->
        <exporterParams moduleName="ExporterName" targetPath="..."/>
    </pepperJobParams>
</pepperParams:PepperParams>
```

Example 1: sample input file content

If the PepperModule you want to use is an importer or an exporter, you can also specify the module by declaring the format name and the format version of the corpus you want to import or export. The Pepper framework will search for an import or export module handling this format. Example 2 shows the specification of an importer or exporter to be used by mentioning the format name and format version of the corpus.

```
<?xml version="1.0" encoding="UTF-8"?>
<pepperParams:PepperParams xmlns:xmi="http://www.omg.org/XMI"
xmlns:pepperParams="de.hu_berlin.german.korpling.saltnpepper.pepper.pepperPa
rams" xmi:version="2.0">
    <pepperJobParams id="1">
        <importerParams       formatName="FORMAT_NAME"
                              formatVersion="FORMAT_VERSION"
                              sourcePath="..."/>

        <!-- ... -->
        <exporterParams       formatName="FORMAT_NAME"
                              formatVersion="FORMAT_VERSION"
                              destinationPath="..."/>
    </pepperJobParams>
</pepperParams:PepperParams>
```

Example 2: sample input file content

Please note that, the value of the attributes '*sourcePath*', 'destination*Path*' and '*specialParams'* (as we show in the following) has to follow the URI notation, which is defined as follows:

```
[scheme:][//authority][path][?query][#fragment]
```

An overview of the java reference implementation can be found here for the interested reader: http://download.oracle.com/javase/6/docs/api/java/net/URI.html).

4.1 Relative paths

To address a relative file path, use the [path] part of the uri expression. For instance to address the corpus 'corpus1' in a pepper workflow description, with the given file structure

```
|- .pepperParams
|-format1
     |-corpus1
```

the corpus '*corpus1*' can be addressed as shown here:

```
./format1/corpus1/
```

4.2 Absolute paths

For addressing absolute paths, one has to define a uri scheme. In the current version of Pepper, only the scheme '*file*' is supported. The path of an absolute uri has to start with a leading '/' followed by the absolute path. It is also allowed to define an empty authority, which results in three leading slashes. For instance for Windows the use of an absolute uri can look like this:

```
     file:/C:/format1/corpus1/     (without authority)

     file:///C:/format1/corpus1/   (with empty authority)
```

Or for linux and mac:

```
     file:/format1/corpus1/        (without authority)

     file:///format1/corpus1/      (with empty authority)
```

4.3 Special parameters

A single step can be parametrized by passing a special parameter file to the corresponding PepperModule. This can be done with the attribute '*specialParams'* in the workflow description file. The '*specialParams*' attribute can be attached to the element '*importerParams*', '*exporterParams*' and '*moduleParams*' as shown in Example 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<pepperParams:PepperParams xmlns:xmi="http://www.omg.org/XMI"
xmlns:pepperParams="de.hu_berlin.german.korpling.saltnpepper.pepper.pepperPa
rams" xmi:version="2.0">
    <pepperJobParams id="1">
        <importerParams moduleName="..."
                        sourcePath="..."
                        specialParams="PATH_TO_PARAMETER"/>
        <!-- ... -->
        <moduleParams   moduleName="..."
                        specialParams="PATH_TO_PARAMETER"/>
        <!-- ... -->
        <exporterParams moduleName="ExporterName"
                        targetPath="..."
                        specialParams="PATH_TO_PARAMETER"/>
    </pepperJobParams>
</pepperParams:PepperParams>
```

Example 3: sample input file content

Note, that even the value of '*specialParams'* has to follow the URI notation.

5 Utilizing Pepper

To utilize Pepper for customized needs, you will find a configuration file named '*userDefined.properties*' in the '*/conf*' folder in your Pepper directory. This file contains some parameters, the values of which can be changed. Here we give a list of possibly interesting parameters. Please note, that this configuration file follows the Java-Property notation, which is declared by '*propertyName=propertyValue'*, and a linefeed determines a new property. A line can be commented out by a preceding '#'. Example 4 shows an example of a property file.

```
#this is a comment
propertyName=propertyValue
anotherPropertyName=anotherValue
```

Example 4: sample property file

Some of these properties determine paths, here you can use the variable '$PEPPER_HOME' to locate the directory of Pepper. This variable is fixed to the path of Pepper and does not have to be declared.

5.1 Log4j

Determines the location of the logging configuration file.

5.2 plugin.path

Determines the path where PepperModules and other OSGi plugins are found.

5.3 plugins.modules.resources

Determines the path where resources are found which are used by some plugins for Pepper.

5.4 temporaries

Determines a folder, where some temporary files can be stored, which are necessary for some plugins while processing. This folder can be manually deleted after a process has been finished.

5.5 pepper.computePerformance

Determines if Pepper should compute the time needed by each PepperModule. This parameter can be set to *true* or *false*. Setting it to true means an output of processing times. Please note that the computation can take time.

5.6 pepper.maxAmountOfProcessedSDocuments

Determines the maximal number of processed SDocument objects at one time.
In general, Pepper is a parallelized framework, which means, that at minimum each PepperModule will run in its own thread, while some of the modules are even parallelized by themselves and run in more than one thread. Such a parallelization brings a lot of benefits in performance especially on a multi-processor machine. Often it happens that modules do not have the same processing time, which means that some modules are faster than others. In case an importer is faster than an exporter, the importer will import more and more data, which are handled in main memory. This can evoke main memory problems, therefore you can limit the maximal number of processed documents at one time. Setting this value to 1 means no parallelization. Bydefault, this value is set to 10.

5.7 pepper.removeSDocumentAfterProcessing

Setting this value tot true means that an SDocument object will be removed after all PepperModules have processed it. This behaviour is highly recommended, because it will save a lot of main memory space.

6 Plug in PepperModules

In most cases when you want to plug in a PepperModule you will get a zip file containing the module as a *.jar* file, and a folder having the same name as the jar file. This folder contains the license files, documentations and other resources the PepperModule needs.

The need to plug in a PepperModule can be caused by two reasons:

1. you want to update an already existing module or
2. you want to install a new PepperModule, which is not already included.

In case 1) move to the plugin folder of Pepper (PEPEPR_HOME/plugins in default) and remove the plugin you want to update, by deleting the corresponding *.jar* file and the folder having the same name. This is necessary in order not to have the same PepperModule twice, because otherwise you cannot determine which one will be used in processing.

After that or in case 2), it is very easy to get your new module running, just unzip the archive into the plugin folder of Pepper (PEPEPR_HOME/plugins in default).

7 Troubleshooting

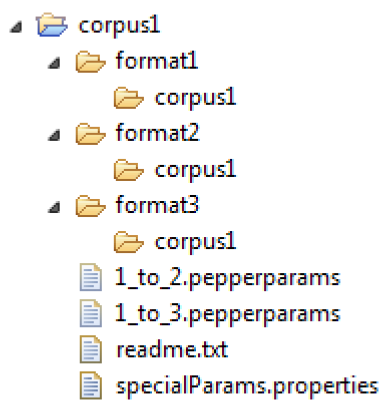
Pepper and Salt are open source projects developed in a low budget environment, although we are doing our best to create stable systems, it can occur that you run into problems when using Pepper. In such cases, please read the error message displayed on the command line first. In many cases, a problem occurs because the data violates a constraint given by one of the PepperModules.

If this doesn't help, don't hesitate to write an email to saltnpepper@lists.hu-berlin.de. Please don't forget to describe your problem as detailed as possible and send us the log files. You will find them under the '*/logs*' folder in your Pepper directory.

8 Best practices

8.1 Corpus structuring

For an easier overview, we recommend a specific folder structure for a corpus and its several formats. The last years have shown that when working with a set of corpora, it is not so easy to remember where a specific format of a single corpus can be found. We recommend to try to keep everything as one bundle. Imagine we have a corpus named corpus1, which is available in the formats '*format1*', '*format2*' and '*format3*'. Further imagine that we have workflow descriptions to convert the corpus from '*format1*' to '*format2*' called '*1_to_2.pepperparams*' and format1 to '*format3*' called '*1_to_3.pepperparams*'. A module can also take a parametrization given in a special parameter file, let's say one of the modules used in '*1_to_3.pepperparams*' references the special parameter file '*specialParameter.properties*'. Maybe we also have a short description, which describes the corpus and its specifics called '*readme.txt*'. Such a bundle of data belonging together can be stored in a folder structure shown in Figure 2.



```
corpus1
    format1
        corpus1
    format2
        corpus1
    format3
        corpus1
    1_to_2.pepperparams
    1_to_3.pepperparams
    readme.txt
    specialParams.properties
```

Figure 2: folder structure showing best practices in how to organize a corpus

When you use relative paths in your workflow description, you are able to share the corpus with others without having to adapt anything. You can define the workflow once and run it anywhere. This benefit can be interesting when working in a team, or in case a problem occurs and you want to send us the corpus to help fix the problem.