Nemo Template App Documentation

Release latest

Jun 08, 2018

Contents

CHAPTER 1

Documentation

1.1 Documentation : Configuration files

1.1.1 Configuration Files

The application is run by two general configuration files. They should not be renamed nor should they be deleted. Configuration files have been built around XML because that's something that people verse in TEI will be familiar with. We are not saying it's the best technical system, but it is the easiest to understand compared to JSON or YAML for most humanists.

On top of that, if you use a good xml editor, each file is defined by a scheme available in the folder configuration-schemas. These schemas are documented.

corpora.xml

The `corpora.xml` file is **meant to define the corpus** (or corpora) that you want to host with your application. It contains informations such as the directories containing the Capitains corpora.

You can find some example of small Capitains corpora at Lasciva Roma/Additional Texts, Lasciva Roma/Priapeia, and Chartes TNAH/Olivar Asselin which is an example of non classical corpora.

Typically, the `corpora.xml` file is divided into three main nodes which we'll present here but you can find a more documented schema here.

Setting up cache

The cache folder node (`<cache-folder>`) is a tool to specify the directory that you are gonna use to cache processed informations. This allows to speed the application by an order of magnitude when the corpora are large.

Adding corpora

The `<corpora>` nodes contains the list of directories containing the texts you want to serve.

Eg. :

```
<corpora>
    <corpus>example_corpora/priapees</corpus>
    <corpus>example_corpora/other</corpus>
</corpora>
```

Two corpus are imported, from both `example_corpora/priapees` and `example_corpora/other` directories.

Creating editorial collections

The `<collections>` nodes contains editorial collection that can be used to make better entry point for the readers. This can overcome the lack of editorialization or the aggregation of multiple corpora.

Setting up texts so that they are registered in a specific collection

While most nodes can be straightforward with the documentation, the `<filters>` one can be somewhat complicated. Let's see an example :

```
<collection>
    <name lang="fre">Sources Latines</name>
    <identifier>latin_collection</identifier>
    <filters>
        <folder>example_corpora/priapees</folder>
        <folder>example_corpora/other</folder>
    </filters>
</collection>
```

Here, we have a collection, identified by the `latin_collection` identifier, named "Sources Latines" in French. Texts will be automatically stored in this collection if they are in the folder `example_corpora/priapees` **or** in the folder `example_corpora/other`.

app.xml

This file is the file responsible for all things related to the Nemo frontend : this will set up some options, give you the ability to add static pages or make it possible to have nice proposed passages for your users.

Let's talk about the main nodes

Adding the ability to see full texts instead of only passages

If the `<full-text-route>` node contains `true`, Nemo will propose for texts the ability to see the complete text and not only the passages in it.

Adding an about route

If the `<about-route>` node contains `true`, the application will add an `About` page based on the HTML in the `` `templates/main/about.html `` template <templates/main/about.html>`__

Setting up how passage of texts are grouped together

Remember *the principles behind Nemo and Nautilus* ? Well, the thing is, the applications, unlike human, have basically no idea how to show the text once they have been parsed. Should I show the text per group of 30 lines ? What about when there is no lines ?

The `<chunking>` node will allow you to make manually curated passages set with the `<hardcoded>` nodes or make curated semi-automatic "passage groupers" based on the identifier of texts (`<identifier-regexp>`) or their citation sytems (`<citation-system>`).

Eg.:

```
<chunking>
    <identifier-regexp level="1" level-name="Priapée {passage}">^
↪urn:cts:latinLit:phi1103\..*$</identifier-regexp>
    <citation-system level="2" level-name="Chapitre {passage}" group-by="5">book,
↪chapter</citation-system>
    <hardcoded identifier="urn:cts:aperire:delver.init.opp-lat1">
        <ref start="2" end="3">Leçon 2 et 3</ref>
    </hardcoded>
</chunking>
```

1. Text matching the identifier regular expression `^urn:cts:latinLit:phi1103\..*$` will have
   (a) their text named `Priapée {passage}` where `{passage}` will be replaced by the identifier of the current passage;
   (b) their texts grouped at the level *one* (here, for these text, that would be the poem level);
2. Text that have their citation system in the exact form of `book` then `chapter` will have
   (a) their text named `Chapitre {passage}` where `{passage}` will be replaced by the identifier of the current passage;
   (b) their texts grouped at the level *two*, so at the chapter level, and they will be grouped by *five*;
3. Text identified by `urn:cts:aperire:delver.init.opp-lat1` will have
   (a) Only one reference shown when browsing available passage, and it will be named `Leçon 2 et 3`

Setting up texts to be shown only as full texts

The node `<full-text-only>` is dependant on the `<full-text-route>` node to be set to `true`. This setting will set up selected or all texts to be only available as a full text in the reading interface (but not in the API !). This specifically makes sense for texts such as short poems, single letters, inscriptions, etc.

This node can contain an attribute `all="true"` that will make all texts only available as full texts, or you can specify which text will be shown this way using `<id>` nodes.

Setting up XSLT and transformation of the TEI

The `<xslts>` node will set-up the XSLTs that needs to be used on specific texts or by default. The default xslt is specified by a `<default>` node and other are identified by an `<xsl>` node.

Eg.

```
<xslts>
    <default>xsl/default.xsl</default>
    <xsl identifier="urn:cts:pompei:cil004-01700.01776.manfred-lat1">xsl/inscription.
↪xsl</xsl>
    <xsl identifier="urn:cts:aperire:delver.init.opp-lat1">xsl/copy.xsl</xsl>
</xslts>
```

- The default XSL is xsl/default.xsl
- If the text identifier is `urn:cts:pompei:cil004-01700.01776.manfred-lat1`, the app will use `` `xsl/inscription.xsl <xsl/inscription.xsl>`__ ``
- If the text identifier is `urn:cts:pompei:cil004-01700.01776.manfred-lat1`, the app will use `` `xsl/copy.xsl <xsl/copy.xsl>`__ ``

Adding additional static pages

You can add additional pages in the node `<additional-pages>`. Templates for these page must be saved in the `templates/additional` directory.

Eg. :

```
<additional-pages>
    <page id="credits" template="credits.html">
        <link-title>Credits</link-title>
    </page>
</additional-pages>
```

- A new page `/page/credits` will be created :
  - it will be using the template in `` `templates/additional/credits.html <templates/additional/credits.html>`__ ``
  - it will be using the title `Credits` in the Menu

1.2 Documentation : Templates

1.2.1 Templates

The Template system behind Nemo is the Jinja template.

Templates are stored in the `./templates` folder for this application. All mandatory templates are available in the `` `./templates/main <./templates/main>`__ `` directory. **You should not remove any of them in case you have forgotten an inclusion. You can however not use them if you wish to.**

For any one wanting to completely dive in, we recommend reading *The Flask Mega Tutorial Part II : Templates* by Miguel Grinberg.

However, the important syntax bits to understand are the following :

Showing a Variable

`{{variable_name}}` is used to show the value of a variable in the code.

For example, `<a class="{{class}}">{{page_name}}</a>` will show `<a class="css-class">Lorem Ipsum</a>` if `class` and `page_name` variables have `css-class` and `Lorem Ipsum` values.

Doing a condition

Conditions are written wrapping some content between `{% if condition %}` and `{% endif %}`. Conditions follow the python syntax, here are some examples :

- `if a == b` checks the equality between a and b variable
- `if a == "b"` checks the equality between the variable a and the text `b`
- `if "a" in some_list` checks if the text `a` is in the list `some_list`
- `if "a" not in some_list` checks if the text `a` is not in the list `some_list`
- `if a` checks if the value behind `a` is truish (not an empty list, string or dictionary, or actually `True`
- `if not a` check if the variable `a` is empty or False

Eg.

```
{% if has_about_route or additional_pages %}
<header>
    <span class="content">Project</span>
</header>
{% endif %}
```

will show the html if `has_about_route` variable or `additional_pages` variable are either true or not empty

Blocks and block extension

If you look in the templates, you'll most likely find something like the following

```
{% extends "main::container.html" %}

{%block article%}
<article>
    <h1>About page to be completed on a project basis</h1>
</article>
{%endblock%}
```

Blocks in Jinja are bits of templates that can be replaced by other templates. Here, this templates open the `./templates/main/container.html <./templates/main/container.html>`__ template and replace the content of the block named `article` with the `<article>[...]</article>` html.

Quite practical when we want to not duplicate code !

Dynamic URLs

Dynamic URLs, or URLs based on the current instance, are recommended. These URLs are built automatically by Flask and it makes sure you are not hardcoding to many things.

The syntax for such links is `{{url_for("name_of_the_route", optional_parameter=parameter_value, optional_parameter=parameter_value)}}` with as many `optional_paramater` as it makes sense for the given link. You can use variable for `parameter_value`, int or string. If you use string, the syntax is `{{url_for("route", number=5, string="some_string", variable=some_variable)}}`

Nemo base templates

The Nemo base templates have specific variables given to them, and some variable are sent accross all templates. You can find more about the variable in the Nemo documentation

Template Name	Role
main/container.html	Global container that is used by every page to not repeat css, javascript, etc.
main/footer.html	Container included in main/container.html for the footer
main/metadata.html	Templates for metadata valid for every pages (inserted in `<head>`)
main/menu.html	Template for menu shown on every page
main/404.html	Page displayed upon 404 errors
main/about.html	Template displayed for the About Page
main/breadcrumb.html	Template displayed for the breadcrumb in every page
main/collection.html	Template displayed for a browsing a collection
main/index.html	Template displayed at the index page
main/logo.html	Template containing the upper left logo
main/references.html	Template used to display the list of passages available for a text
main/text.html	Template used to display a passage
main/passage_footer.html <templates/main/passage_footer.html> __	Template shown below a passage when reading one.
main/macros.html	Macros used accross templates. We recommend not changing it

1.2.2 Linking to other pages

You will find links here and there in the templates but here are the main pages with their parameters. These routes are called using the *{{url_for()}} syntax*.

- `r_index` has no parameters. It's the index of the website
- `r_collections` has no parameter. It leads to the main collection page.
- `r_collection` takes an `objectId` parameter and optionally a `semantic` one to make a nice link. `objectId` represents the identifier of the collection to show. It displays children and metadat about a specific collection
- `r_first_passage` takes an `objectId` parameter. It will redirect to the first passage of the text identified by the variable `objectId`
- `r_passage` takes an `objectId` and a `subreference` parameter. It will show the passage identified by `subreference` in the text identified by `objectId`.
- `r_references` takes an `objectId` parameter. It will show the list of available curated passages in the text identified by `objectId`.
- *(Optionally, depending on app configuration)* `r_full_text` takes an `objectId` parameter. It will full content of the text identified by `objectId`.
- *(Optionally, depending on app configuration)* `r_about` takes no parameter. It will show the about page
- *(Optionally, depending on app configuration)* `r_page` takes a `page_id` parameter. It will show the page identified by the `page_id` parameter.

Eg.

```
<a href="{{url_for('.r_index')}}">Index</a>
<a href="{{url_for('.r_collection', objectId='urn:cts:latinLit:phi1103.phi001')}}">
↪Collection des Priapées</a>
<a href="{{url_for('.r_first_passage', objectId='urn:cts:latinLit:phi1103.phi001.
↪lascivaroma-lat1')}}">Premier passage des Priapées</a>
<a href="{{url_for('.r_passage', objectId='urn:cts:latinLit:phi1103.phi001.
↪lascivaroma-lat1', subreference='55')}}">Priapée 55</a>
```

will produce normally

```
<a href="/">Index</a>
<a href="/collection/urn:cts:latinLit:phi1103.phi001">Collection des Priapées</a>
<a href="/text/urn:cts:latinLit:phi1103.phi001.lascivaroma-lat1">Premier passage des␣
↪Priapées</a>
<a href="/text/urn:cts:latinLit:phi1103.phi001.lascivaroma-lat1/passage/55">Priapée 55
↪</a>
```

1.2.3 Linking to Statics, JS and CSS

You can add or replace statics by adding file in the `./statics/ <./statics>`__ folder. These file can then be refered to using the following syntax : `{{url_for('.static', filename='path/from/statics')}}`.

Eg. the current `/statics` folder contains :

- css
  - bootstrap.min.css
  - theme.min.css
- images
  - logo.png

If we want to refer to logo, we will type `{{url_for('.static', filename='images/logo.png')}}`. And if we want to insert it as an image, we will write

```
<img class="logo" src="{{url_for('.static', filename='images/logo.png')}} " alt=
↪"Capitains Nemo" />
```

This documentation is alive, which means if you feel something is not clear, feel free to open an issue on github and we will happily come back to you. Open Source cannot live well without feedback !

CHAPTER 2

Installing Python 3 on Mac and Ubuntu

We recommend using Python 3 for this tutorial. Earlier versions can cause problems.

2.1 Installation

2.1.1 OS X

We recommmend that you install the Anaconda distribution. It has the necessary modules and packages for this tutorial. It is available on all platforms and has a simple installation procedure. You can download it from http://continuum.io/downloads. Installation instructions can be found at http://docs.continuum.io/anaconda/install.html.

Install the most recent version of 3.6. Once it is installed, enter the following on your terminal:

```
conda create -n nemo-env
```

Followed by

```
source activate nemo-env
```

This last step activates a local python environment, preventing changes the general python environment on your computer.

From within the directory containing your local git clone of this tutorial's repository on your machine, enter the following commands on the terminal:

```
pip install -r requirements.txt
```

2.1.2 Linux (Ubuntu/Debian)

N.B. you will need administrator rights to follow these instructions.

Open a terminal window and enter:

```
sudo apt-get install python3 libfreetype6-dev python3-pip python3-virtualenv
```

Once installed, enter:

```
virtualenv ~/.nemo-env -p python3
```

That will create a virtual environment in which you can install all of the necessary libraries.

At the terminal, **enter the directory containing the local git clone of this tutorial's repository on your machine**, and from within this directory type the following commands:

```
source ~/.nemo-env/bin/activate
```

You will need to type this each time you want to work on the tutorial.

From within the same terminal and directory now type:

```
pip install -r requirements.txt
```

This will install the packages necessary for this tutorial.

2.2 Contributors

- Mike Kestemont
- Folgert Karsdorp
- Maarten van Gompel
- Matt Munson
- Thibault Clérice
- Bridget Almas

CHAPTER 3

Installer Python 3 sur Mac et Ubuntu

Nous utiserons Python 3 dans notre cours. Les versions précédentes peuvent poser des problèmes.

3.1 Installation

3.1.1 OS X

Nous vous conseillons d'installer la distribution Anaconda. Elle contient tous les modules et packages nécessaires pour ce cours. Elle est disponible pour toutes les plateformes et possède une procédure d'installation assez simple. Vous pouvez la télécharger depuis http://continuum.io/downloads. Des détails pour l'installation peuvent être trouvés ici : http://docs.continuum.io/anaconda/install.html

Utilisez bien la version 3.6 proposée. Une fois installée, tapez ensuite

```
conda create -n nemo-env
```

suivi de

```
source activate nemo-env
```

Cette dernière active un environnement de python qui nous permet de ne pas modifier l'environnement général de votre ordinateur.

Allez dans le repository git puis tapez dans le terminal de ce dossier

```
pip install -r requirements.txt
```

```
python cli.py dev-run
```

Si tout va bien, cela devrait ouvrir votre navigateur sur la page http://127.0.0.1:5000/

3.1.2 Linux (Ubuntu/Debian)

Vous aurez besoin des droits d'administrateurs pour faire ce qui suit.

Ouvrez un terminal et tapez :

```
sudo apt-get install python3 libfreetype6-dev python3-pip python3-virtualenv
```

Puis, une fois cela installé, faites :

```
virtualenv ~/.nemo-env -p python3
```

Cela créera un environnement virtuel dans lequel nous pourrons installer l'ensemble des informations nécessaires. Allez, dans le terminal, dans le dossier git du cours que vous avez cloné localement et tapez :

```
source ~/.nemo-env/bin/activate
```

Cette commande sera obligatoire à chaque fois que vous voudrez travailler avec le cours. Dans le même terminal, tapez maintenant

```
pip install -r requirements.txt
```

Cela installera les packages nécessaires pour le cours. Une fois ces packages installés, il suffira de taper

```
python cli.py dev-run
```

Si tout va bien, cela devrait ouvrir votre navigateur sur la page http://127.0.0.1:5000/

3.2 Contributeurs

- Mike Kestemont
- Folgert Karsdorp
- Maarten van Gompel
- Matt Munson
- Thibault Clérice

CHAPTER 4

Introduction

You can do a lot without diving into Python with this application, it's actually it's first goal. This application will provide you a **way to generate a website** for TEI/Epidoc texts with a CTS API **without doing any python**. This "web app generator" is based on Capitains Nemo and Capitains Nautilus.

Be warned : this application only works with files following the Capitains Guidelines !

Of course, you can actually go deeper, touch the python, modify some stuff in Nemo or Nautilus. Nautilus is a full scale app, you can change some things in it but it's mostly meant as an application that can work by itself. On the other end, Nemo is meant to be a skeleton for developers who would like to build their own website ! Here, we made it a little more configurable so you would not need to do python for most trivial tasks, but if you want more customization that what you'll see is already offered, you'll unfortunately have to follow the documentation of Nemo, do python and learn maybe using the Nemo Tutorial.

The original intent of this repository is to answer to a lack of good introductory tools and has been built for a workshop in Lyon given by Thibault Clérice, and as a general effort to make Capitains technically and timely affordable.

4.1 What's the bigget principle behind Nemo and Nautilus ?

Nemo and Nautilus are built on the idea behind Capitains that texts are some kind of Ordered Hierarchy of Content Objects, to put it simply, your text should be citable by some logical units which makes up passage. They can be in a hierarchy (with different levels) such as the traditional poem anthology structure : poem -> stanza -> line.

4.2 Note about the repository

The current application is in a working state and can be run as a demo. This application can only be run on Unix machines (Linux and MacOS) : you'll need to install python3. There is a French tutorial and English tutorial. See CONTRIBUTING.md for more advises on how to do the installations specific to the current repository.

CHAPTER 5

Documentation

CHAPTER 6

Contributing

See CONTRIBUTING.md

6.1 License

The software hereby presented is given to you under the Mozilla Public License 2.0.