Flask.ext.Nemo Documentation

Release

Thibault Clérice

Aug 23, 2017

Contents

1 Install **3**

2 Running Nemo from the command line **5**

2.1 Examples . . . 5
2.2 Workflow Nemo Design . . . 10
2.3 QueryInterface Design and Components Hierarchy . . . 11
2.4 Nemo Developper Guide . . . 13
2.5 Templates documentation . . . 15
2.6 Chunkers, Transformers and GetPrevNext . . . 18
2.7 Nemo Plugin System . . . 21
2.8 Existing Plugins . . . 23
2.9 Restriction regarding XSLT . . . 24
2.10 Nemo API . . . 24

Capitains Nemo is an User Interface built around the need to make CTS a easy to use, human readable standard for texts. Capitains Nemo counts multiple language implementation, including this one in Python for Flask. Presentend as a classic Flask Extension, *flask.ext.nemo* intends to be a simple, customizable interface between your enduser and your CTS5 API.

The Flask's extension Nemo can be customized from its stylesheets to its functions. Adding routes or removing them is as easy as adding a XSL Stylesheet to transform the very own result of a CTS GetPassage results to your own expected output.

CHAPTER 1

Install

You can now install it with pip : `pip install flask_nemo`

If you want to install the latest version, please do the following

```
git clone https://github.com/Capitains/flask-capitains-nemo.git
cd flask-capitains-nemo
virtualenv -p /path/to/python3 venv
source venv/bin/activate
python setup.py install
```

If you have trouble with dependency conflicts with MyCapitains, try running `pip install MyCapytain` this before install

CHAPTER 2

Running Nemo from the command line

This small tutorial takes that you have a CTS API endpoint available, here `http://localhost:8000`

1. (Advised) Create a virtual environment and source it : `virtualenv -p /usr/bin/python3 env`, `source env/bin/activate`
2. **With development version:**
    - Clone the repository : `git clone https://github.com/Capitains/flask-capitains-nemo.git`
    - Go to the directory : `cd Nemo`
    - Install the source with develop option : `python setup.py develop`
2. **With production version:**
    - Install from pip : `pip install flask_nemo`
3. You will be able now to call capitains nemo help information through `capitains-nemo --help`
4. Basic setting for testing is `capitains-nemo cts-api https://cts.perseids.org/api/cts`.

Examples

Simple Configuration

User story [1]

A **researcher** , an **engineer** or both is interested in **CTS** but has **no time** to develop their own application and their own theme : *flask.ext.nemo* will provide a simple, easy to use interface that you can deploy on any server. Even with a really limited knowledge of python.

User story [2]

A **researcher**, an **engineer** or both has already a CTS endpoint and wants to check the output and the browsing system visually.

The simplest configuration of Nemo, or close to it, is to simply give an endpoint url to your Nemo extension, the app you are using and the name of a CTS inventory (if required). This will run a browsing interface with support for collections, textgroups, texts and passages browsing.

- The application will itself do the GetCapabilities request to retrieve the available texts and organize them through collection, textgroups and works.
- Once an edition or a translation is clicked, a page showing available references is shown.
- Once a passage is clicked, the passage is shown with available metadata.

```
# We import Flask
from flask import Flask
# We import Nemo
from flask.ext.nemo import Nemo
# We import enough resources from MyCapytain to retrieve data
from MyCapytain.resolvers.cts.api import HttpCtsResolver
from MyCapytain.retrievers.cts5 import HttpCtsRetriever

# We set up a resolver which communicates with an API available in Leipzig
resolver = HttpCtsResolver(HttpCtsRetriever("http://cts.dh.uni-leipzig.de/api/cts/"))
# We create an application. You can simply use your own
app = Flask(
    "My Application"
)
# We register a Nemo object with the minimal settings
nemo = Nemo(
    # We give a resolver
    resolver=resolver,
    # We set up the base url to be empty. If you want nemo to be on a
    # subpath called "cts", you would have
    # base_url="cts",
    base_url="",
    # We give thee ap object
    app=app
)
# We run the application
app.run()
```

Note: You can run this example using *python example.py default*

XSLT, CSS and Javascript addons

User Story

A developer, with no or only limited understanding of python, wants to expose their CTS works but have some modifications to do regarding the design.

Because Python is not a natural language and because not everybody knows it in academia, you might find yourself in a situation where you don't know it. On the other hand, XML TEI, HTML, CSS - and thus xsl and sometimes Javascript - are quite common languages known to both researchers and engineers. Capitains Nemo for Flask accepts custom templates, CSS, Javascript, XSL and statics. And in a simple, nice way :

```
# ...
nemo = Nemo(
    # We give a resolver
    resolver=resolver,
    base_url="",
    inventory="ciham",
    # For transform parameters, we provide a path to an xsl which will be used for every
    transform={"default" : "examples/ciham.xslt"},
    # CSS value should be a list of path to CSS own files
    css=["examples/ciham.css"],
    # JS follows the same scheme
    js=[
        # use own js file to load a script to go from normalized edition to diplomatic one.
        "examples/ciham.js"
    ],
    templates={
        "main": "examples/ciham"
    },
    additional_static=[
        "path/to/picture.png"
    ]
)
```

Additional CSS, JS or Statics in Templates

To call or make a link to a static in your own template, you should always use the helper *url_for* and the route name *secondary_assets*. Additional statics can be linked to using the filename (be sure they do not collide !) and the type : css, js or static. Example : *{{url_for('nemo.secondary_assets', type='static', asset='picture.png')}}*.

Note: Templates are written with Jinja2. See also *Templates*. For XSL, we have some unfortunate restrictions, see *strip-spaces*

Note: You can run an example using css, js, templates and transform with *python example.py ciham*

Own Chunker

Warning: Starting from this example, the configuration and changes implied require the capacity to develop in Python.

User Story

> A developer wants to add a custom scheme for browsing text passages by groups that are not part of the citation scheme of the text. The custom scheme should be triggered by text identifier or using available CTS metadata about the text, such as the Citation Scheme.

CTS is good, but getValidReff can really be a hassle. The default generation of browsing level will always retrieve the deepest level of citations available. For the Iliad of Homer, which is composed of two levels, books and lines, this would translate to a GetValidReff level 2. This would mean that the generic chunker would return on the text page a link to each line of each book (it's a total of 15337 lines, if you did not know).

Chunker provides a simple, easy to develop interface to deal with such a situation : for example, returning only 50 lines groups of links (1.1-1.50, 1.51-1.100, etc.). The Nemo class accepts a chunker dictionary where **keys** are **urns** and where the key "**default**" is the default chunker to be applied. Given a chunker named *homer_chunker* and one named *default_chunker*, if the urn of Homer is **urn:cts:greekLit:tlg0012.tlg001.opp-grc1** (See *function skeleton* for):

```
# ...
nemo = Nemo(
    # ...
    chunker= {
        "urn:cts:greekLit:tlg0012.tlg001.opp-grc1" : homer_chunker,
        "default": default_chunker
    }
   )
```

Note: You can run an example using chunker with *python example.py chunker*

Note: Parameters XSLT and prevnext work the same way. See relevant documentation : *Chunkers* for more information about and examples of chunkers

Adding routes

User story

The user has needs in terms of new routes that would cover specific needs, like vis-a-vis edition.

There is multiple way to deal with this kind of situation. The best way is to create a subclass of Nemo. The idea behind that is that you rely on specific functionalities of Nemo and its context object. To deal with that and make as much as possible a good use of Nemo extension, you just need to add a new route to url using a tuple : first value would be the route, according to Flask standards, *ie /read/<collection>/<textgroup>/<work>/<version>/<passage_identifier>/<visavis>* , the name of the function or method (naming convention makes them start by r_), *ie r_double*, and a list of methods, by default ["GET"].

As you will most likely use a new template, don't forget to register it with the templates parameter !

```
# #We create a class based on Nemo
class NemoDouble(Nemo):
    def r_double(self, collection, collection2, work, version, passage_identifier,␣
↪visavis):
```

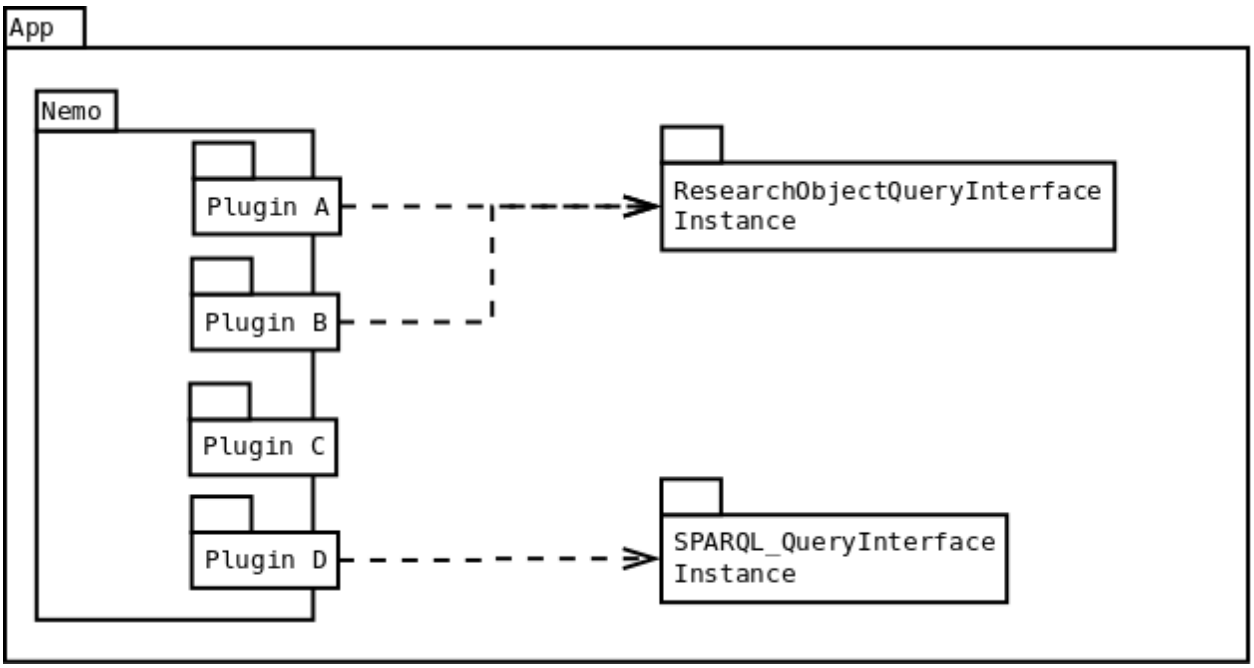
```
        """ Optional route to add a visavis version

        :param collection: Collection identifier
        :type collection: str
        :param textgroup: Textgroup Identifier
        :type textgroup: str
        :param work: Work identifier
        :type work: str
        :param version: Version identifier
        :type version: str
        :param passage_identifier: Reference identifier
        :type passage_identifier: str
        :param version: Visavis version identifier
        :type version: str
        :return: Template, version inventory object and Markup object representing␣
↪the text
        :rtype: {str: Any}

        .. todo:: Change text_passage to keep being lxml and make so self.render turn␣
↪etree element to Markup.
        """

        # Simply call the url of the
        args = self.r_passage(collection, textgroup, work, version, passage_
↪identifier)
        # Call with other identifiers and add "visavis_" front of the argument
        args.update({ "visavis_{0}".format(key):value for key, value in self.r_
↪passage(collection, textgroup, work, visavis, passage_identifier).items()})
        args["template"] = "double::r_double.html"
        return args

nemo = NemoDouble(
    api_url="http://cts.perseids.org/api/cts/",
    base_url="",
    inventory="nemo",
    # We reuse Nemo.Routes and add a new one
    urls= Nemo.ROUTES + [("/read/<collection>/<textgroup>/<work>/<version>/<passage_
↪identifier>/<visavis>", "r_double", ["GET"])],
    css=[
        "examples/translations.css"
    ],
    # We think about registering the new route
    templates={
        "double": "./examples/translations"
    }
)
```

Note: You can run an example using chunker with *python example.py translations*

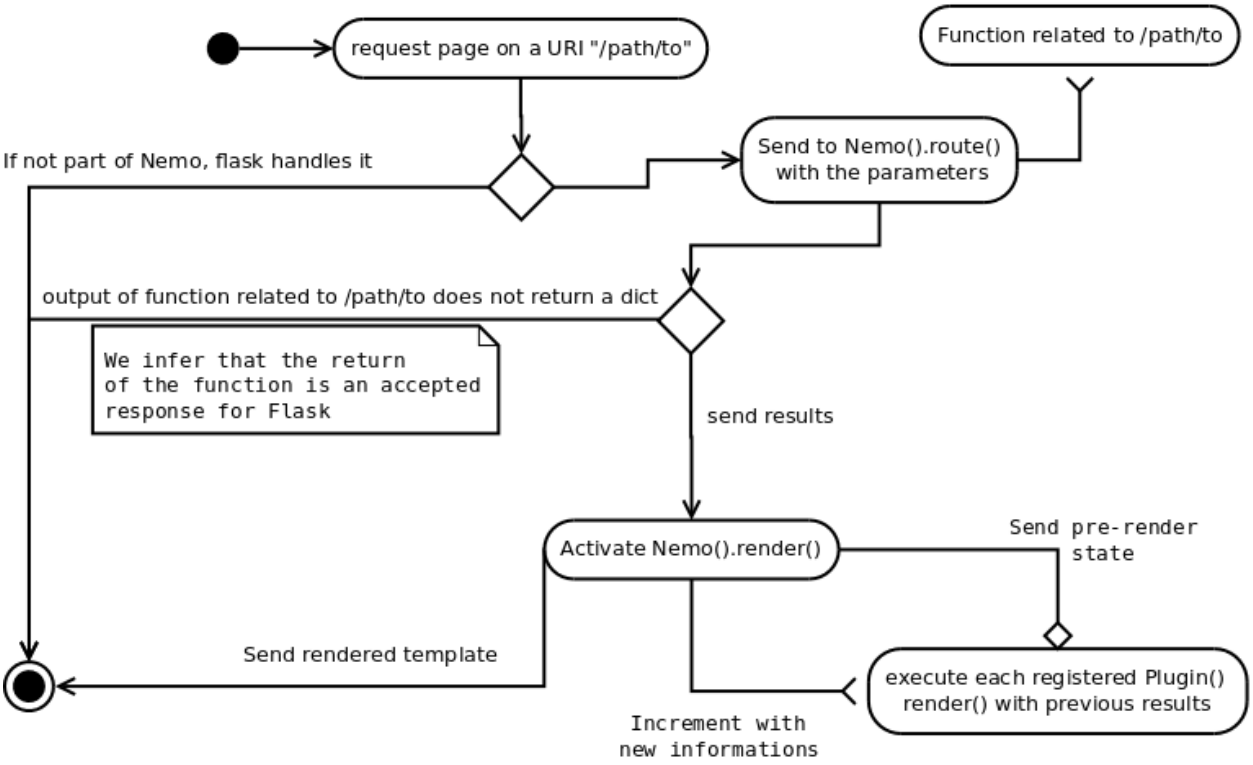
Workflow Nemo Design

Application Hierarchy

This workflow demonstrates how components works together



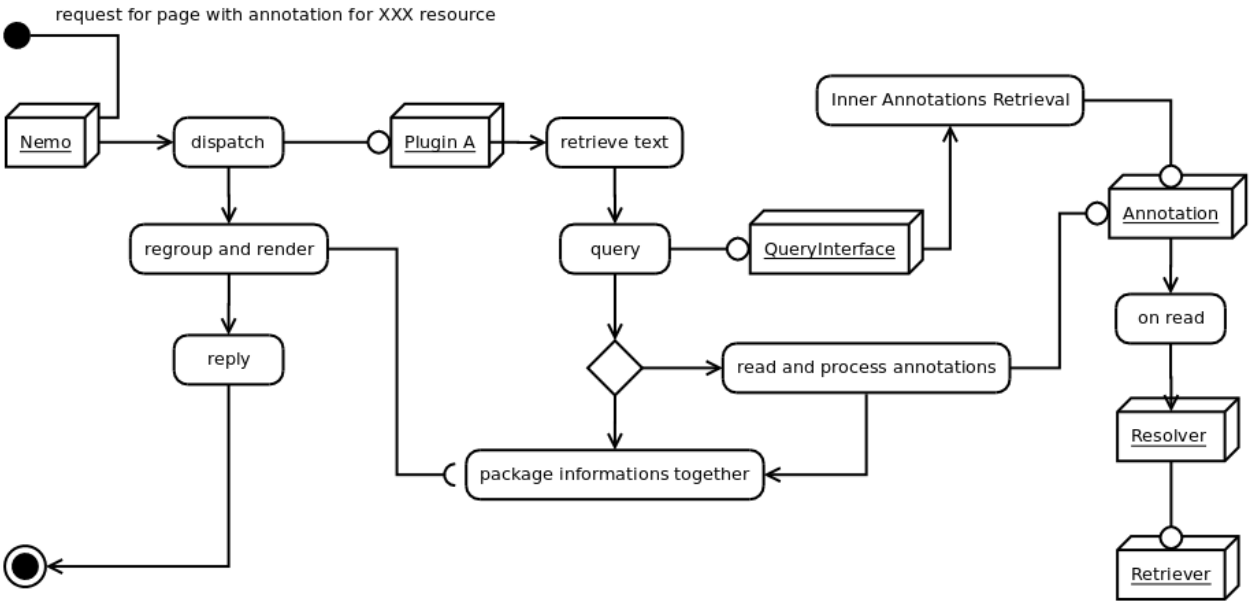
Nemo Rendering Workflow



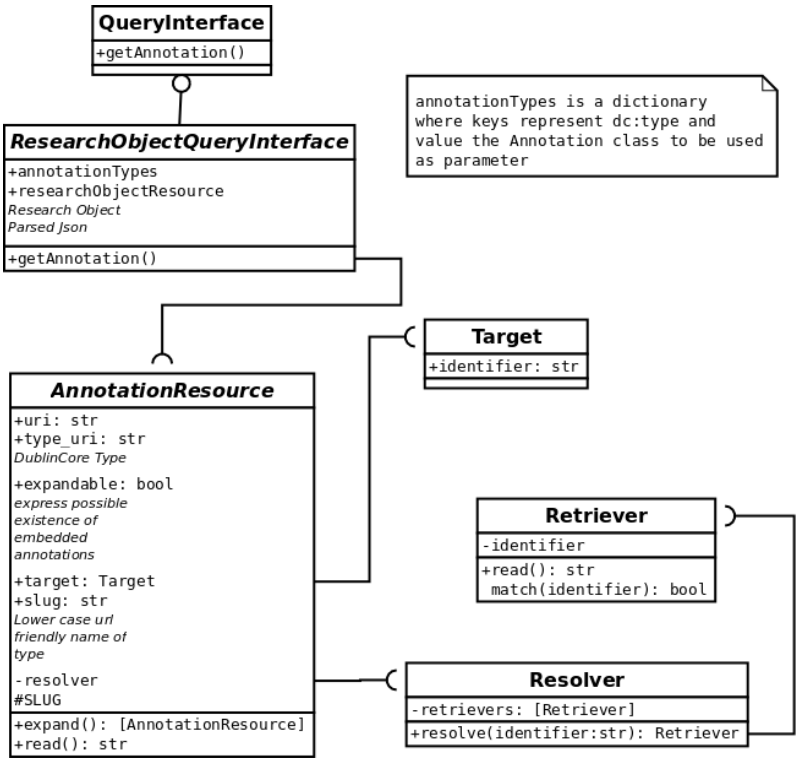
QueryInterface Design and Components Hierarchy

Workflow of the components

This workflow illustrates how QueryInterface components works together



Description of the components



QueryInterface

is an object that given a URN retrieves annotations for it.

- Query interface should be fed with a function to retrieve valid references of the text

- **It should have a .getAnnotation method which returns tuple where first element is a list of annotations and the second is tl**
  - `*urn` which takes a URN Object (MyCapytain)
  - **`wildcard` as a boolean**
    - `.` means exact match
    - `.%` means lower match matches
    - `%.` means higher match matches
    - `-` in range of
    - `%.%` means not level dependant
  - `include, exclude` which would restrict the type of resources that can be retrieved using list of types
  - `limit` as a limit of number, default to None
  - `start` as the first parameter
  - `expand` should automatically expand annotations matching

Annotations

- Takes a resolver
- Has a read() method
- Has an .expandable properties which means the annotation might have embedded annotations
- Has an expand() methods which returns embedded annotations as a list

Resolver

- Decides on which retriever to use in a list of retrievers.
- Takes *retrievers=[]* as init argument
- Has a function resolve that takes an identifier argument
- Return a Retriever object

Retriever

Retriever retrieves a resource given an identifier which can be cts, cite, local path, url, you name it.

- Has a *.match()* static method that returns True or False if the identifier can be retrieved by it
- Has a *.read()* method that returns the body of the resource

Nemo Developper Guide

How to contribute ? Our Github Etiquette

- Open Issues

- Do Pull Request
- Check Unit Tests results and write your own
- Add documentation !
- Do not decide yourself on changing the version

Writing and running tests

- Add tests which are not taking care of the rendering (Testing the function itself)
- Add a "navigational test" in *tests/test_with_nautilus*

Ultimately, tests should be part of the right file and take a name starting with `test_` :

```
from unittest import TestCase

class TestUnit(TestCase):
    """ Description of the general group of test represented by TestUnit """

    def test_a_function(self):
        """ Simple definition of this test purpose """
        do = 1
        self.assertEqual(
            do, 1,
            "A human friendly message explaining what we check and we expect"
        )
```

Running tests

In flask-capitains-nemo repository

```
virtualenv env -p /usr/bin/python3
source env/bin/activate
python setup.py tests
```

Writing and building documentations

- Add your new function to docs/Nemo.api.rst
- In you add a chunker, ensure to register it in Nemo.chunker.rst
- If the change has any impact on the general behaviour, make sure to check *Nemo.examples.rst* usecases

Coding guidelines

- Routes start with *r_*
- Filters start with *f_*
- Private variables should not be modified after *__init__*

Templates documentation

Templates

Namespaces

Nemo templates are namespaced since 1.0.0 and the plugin system. Nemo base plugins are in the namespace "main", the prefix used in Nemo being "::". For any new template added to Nemo using the original container, it would probably be designed with

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

{%block article%}
Add the HTML here
{%endblock%}
```

Add and change templates

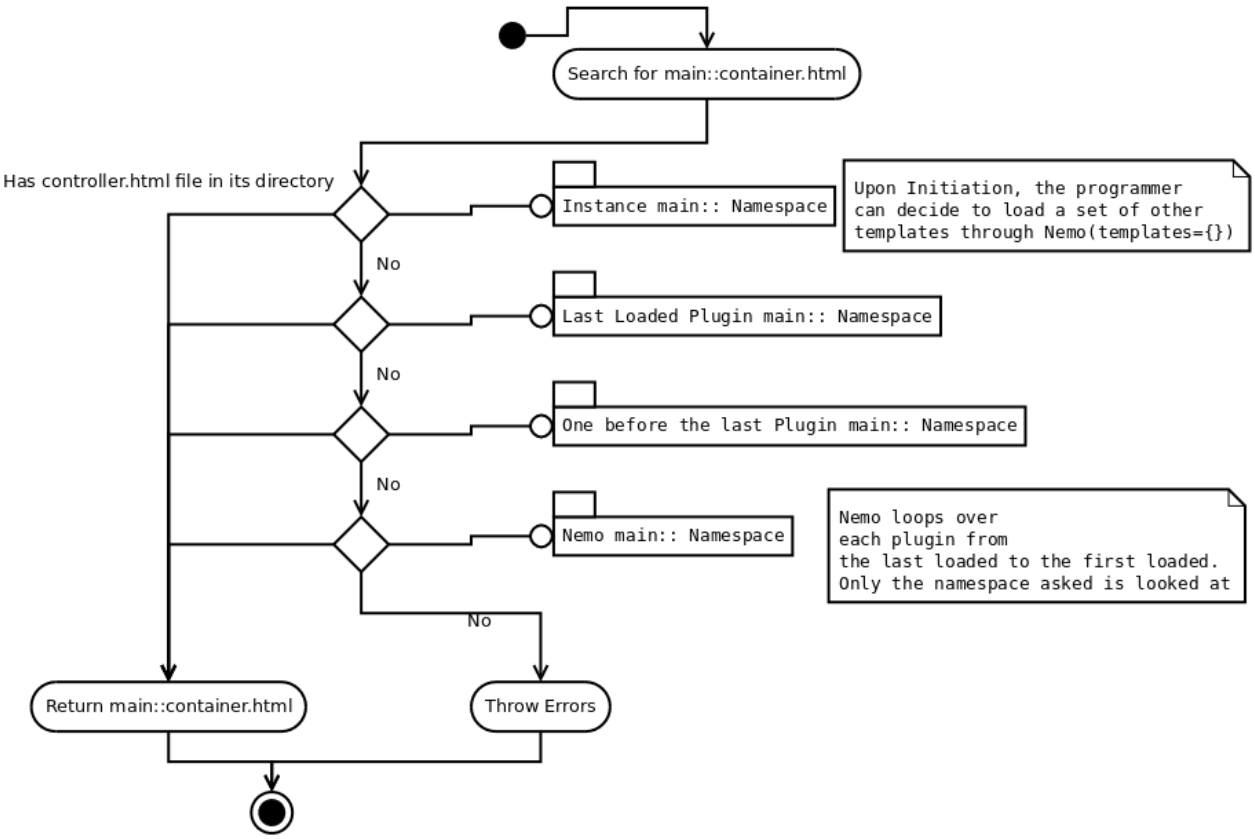
You can add namespace to Nemo by using the templates parameter on initiation, but it can also be used to overwrite templates. For example, given a directory *./templates/main* with a file *container.html*, the following code snippet would write over Nemo original container.html

```
nemo = Nemo(templates={
    "main": "./templates/main"
})
```

Note: It is recommended to split namespaced templates in different folders when adding templates to the Nemo instance.

Warning: If two declared namespaces share the same directory, there is risk of collision between templates names and thus overwriting potential. For example, if `main::` and `plugin_1::` point to `dir1`, `main::file.html` and *plugin_1::file.html* are the same.

Template choice behaviour in Nemo



Nemo Default Templates

The following tables gives informations about the variables sent to each templates.

Variables shared across templates

Variable Name	Details
assets['css']	List of css files to link to.
assets['js']	List of js files to link to.
url[*]	Dictionary where keys and values are derived from routes
templates[*]	Dictionary of templates with at least menu and container
collections	Collections list
lang	Lang to display

main::index.html

Only *Variables shared across templates*

main::menu.html

Only *Variables shared across templates*

main::breadcrumb.html

Only *Variables shared across templates*

main::textgroups.html

See *r_collection*

Variable Name	Details
textgroups	List of textgroups according to a collection

main::texts.html

main::See *r_texts*

Variable Name	Details
texts	List of texts according to a textgroup

main::version.html

See *r_version*

Variable Name	Details
version	Version object with metadata about current text
reffs	List of tuples where first element is a reference, second a human readable translation

main::text.html

See *r_passage*

Variable Name	Details
version	Version object with metadata about current text
text_passage	Markup object representing the text
urn	Markup object containing the URN of the passage for display
prev	Previous Passage Reference
next	Following Passage Reference

main::passage_footer.html

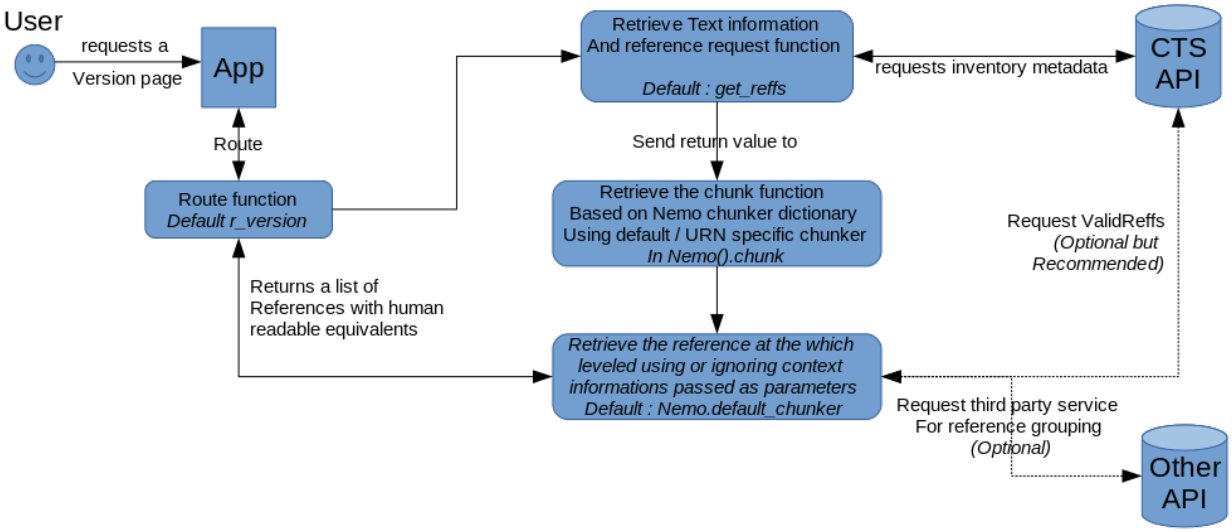
See *r_passage*

Variable Name	Details
version	Version object with metadata about current text
text_passage	Markup object representing the text
urn	Markup object containing the URN of the passage for display
prev	Previous Passage Reference

Chunkers, Transformers and GetPrevNext

Chunker, Transformers and GetPrevNext are way to customize your users' experience. Chunkers will decide what are the possible passages to see for a text, GetPrevNext what should be the previous and next passage while Transformers is a way to customize transformation, with or without XSLT

Process description



User story

A user browses available texts and select a text. He does not want a specific passages. Nemo proposes a list of passages based on the structure of the text.

Example: The Epigrams of Martial are a group of many books, each containing hundreds of poems, which are themselves composed of up to 50 lines. The use would preferably be proposed the poem as the minimal citation scheme for browsing, rather than each line.

To propose passages to the user, Capitains Nemo uses a chunker function which will group, if needed, references together. The function is called upon returning the list of references to the view. The function should always return a list of references, and not full urn, with a human readable version of it, which can be the same.

Chunkers

In the Nemo class, you'll find static methods called chunker, which can be used in the context of the Nemo().chunker dictionary. Chunkers are used to take care of grouping references when a user arrives on the version page of a text, to select where they should go.

Nemo contains multiple chunkers and accepts any contributions which provide helpful, transproject functions.

Defining a chunker in your Nemo implementation instance

The Nemo class accepts a chunker named argument that should be a dictionary where values are chunker functions. This dictionary should at least contain one key named "default". Any other key should represents a URN and will override the default function if the requested version has the given urn.

```
from flask.ext.nemo import Nemo
from flask.ext.nemo.chunker import default_chunker, scheme_chunker, line_chunker

    "default": default_chunker,
    "urn:cts:latinLit:phi1294.phi002.perseus-lat2": scheme_chunker,
    # This will override the original function and provides a poem based reference␣
↪for Martial Epigrammata in this version
    "urn:cts:latinLit:phi1017.phi004.opp-lat4": lambda version, callback: line_
↪chunker(version, callback, lines=50)
    # Use a lambda to override default line numbers returned by Nemo.line_chunker for␣
↪Seneca's Medea
})
```

Note: See :ref:‘ API documentation <Nemo.api>‘

Building your own : Structure, Parameters, Return Values

```
# Chunker skeleton
def chunker_name(version, getValidReff):
    """ Document what your chunker should do

    :param version: A version object according to MyCapytains standards. It contains␣
↪metadata about the citation scheme through version.citation
    :type version: MyCapytains.resources.inventory.Text
    :param getValidReff: Callback function to perform a getValidReff on the given␣
↪param. It accepts a single parameter named "level" and returns a list of URNs
    :type getValidReff: function(level) -> [str]
    :return: A list of tuple of strings where the first element is the CTS URN␣
↪reference part and the second a human readable version of it
    :rtype: [(str, str)]
    """
    return [("1.pr", "Book 1 Prolog")("1.1", "Book 1 Poem 1"), ...]
```

A chunker should take always at least two positional arguments :

- The first one will be the version, based on a MyCapytains.resources.inventory.Text class. It contains information about the citation scheme for example.
- The second one is a callback function that the chunker can use to retrieve the valid references. This callback itself takes a parameter named level. This callback corresponds to a MyCapytains.resources.texts.api.getValidReff() method. It returns a list of string based urns.

The chunker itself should return a list of tuples where the first element is a passage reference such as "1.pr" or "1-50" and a second value which is a readable version of this citation node.

Note: As seen in the diagram, there is no limitation for the chunker as long as it returns a valid list of references and their human readable version. It could in theory ask a third party service to return page-based urns to browse a text by

pages according its OCR source / manuscript

```
# Example of chunker for the Satura of Juvenal
def satura_chunker(version, getValidReff):
    reffs = [urn.split(":")[-1] for urn in getValidReff(level=2)]
    # Satura scheme contains three level (book, poem, lines) but only the Satura␣
↪number is sequential
    # So as human readable, we give only the second member of the reference body
    return [(reff, "Satura {0}".format(reff.split(".")[-1])) for reff in reffs]1
```

Available chunkers

chunker.**default_chunker**(*text*, *getreffs*)

This is the default chunker which will resolve the reference giving a callback (getreffs) and a text object with its metadata

Parameters

- **text** (*MyCapytains.resources.inventory.Text*) – Text Object representing either an edition or a translation
- **getreffs** (*function*) – callback function which retrieves a list of references

Returns List of urn references with their human readable version

Return type [(str, str)]

chunker.**line_chunker**(*text*, *getreffs*, *lines=30*)

Groups line reference together

Parameters

- **text** (*MyCapytains.resources.text.api*) – Text object
- **getreffs** (*function(level)*) – Callback function to retrieve text
- **lines** (*int*) – Number of lines to use by group

Returns List of grouped urn references with their human readable version

Return type [(str, str)]

chunker.**scheme_chunker**(*text*, *getreffs*)

This is the scheme chunker which will resolve the reference giving a callback (getreffs) and a text object with its metadata

Parameters

- **text** (*MyCapytains.resources.inventory.Text*) – Text Object representing either an edition or a translation
- **getreffs** (*function*) – callback function which retrieves a list of references

Returns List of urn references with their human readable version

Return type [(str, str)]

chunker.**level_chunker**(*text*, *getValidReff*, *level=1*)

Chunk a text at the passage level

Parameters

- **text** (*MyCapytains.resources.text.api*) – Text object

- **getreffs** (*function(level)*) – Callback function to retrieve text

Returns List of urn references with their human readable version

Return type [(str, str)]

chunker.**level_grouper**(*text*, *getValidReff*, *level=None*, *groupby=20*)
Alternative to level_chunker: groups levels together at the latest level

Parameters

- **text** – Text object
- **getValidReff** – GetValidReff query callback
- **level** – Level of citation to retrieve
- **groupby** – Number of level to groupby

Returns Automatically curated references

PrevNext

PrevNext follows the same scheme as Chunker.

Transformers

Transformers should always return a string

Nemo Plugin System

Since flask-capitains-nemo 1.0.0, plugins are a reality. They bring a much easy way to stack new functionalities in Nemo and provide large reusability and modularity to everyone.

Design

Nemo Plugins are quite close in design to the Nemo object but differ in a major way : they do not have any Flask Blueprint instance and are not aware directly of the flask object. They only serve as a simple and efficient way to plugin in Nemo a set of resources, from routes to assets as well as filters.

The way Nemo deals with plugins is based on a stack design. Each plugin is inserted one after the other and they take effect this way. The last plugin routes for the main route / is always the one showed up, as it is for assets and templates namespaces.

What are plugins

What can plugin do

- It can add assets (javascript, css or static files such as images)
- It can provide new templates to theme Nemo
- It can add new routes on top of the existing routes
- It can remove original routes, and bring new one

- It can bring new filters
- It can add new informations to what is passed to the template through their *pluginRender* function

How plugin are registered

At the creation of the Blueprint in Nemo, Nemo runs a function which does the following in said order:

- Clear routes first if asked by one plugin
- Clear assets if asked by one plugin and replace by the last registered plugin `STATIC_FOLDER`
- **Register each plugin**
  - Append plugin routes to registered routes
  - Append plugin filters to registered filters
  - Append templates directory to given namespaces
  - Append assets (CSS, JS, statics) to given resources
  - Append render view (if exists) to Nemo.render stack

Inserting a plugin in a Nemo instance

Listing 2.1: You app.py

```
# We import Flask and Nemo
from flask import Flask
from flask_nemo import Nemo
# For this demo, we use the plugin prototype which does not include anything special
from flask_nemo.plugin import PluginPrototype
from MyCapytain.resolvers.cts.api import HttpCtsResolver
from MyCapytain.retrievers.cts5 import HttpCtsRetriever

# We set up a resolver which communicates with an API available in Leipzig
resolver = HttpCtsResolver(HttpCtsRetriever("http://cts.dh.uni-leipzig.de/api/cts/"))

# Initiate the app
app = Flask(__name__)
# We initiate the plugins
proto_plug = PluginPrototype()
# We insert the plugin into Nemo while setting up Nemo
nemo = Nemo(
    resolver=resolver,
    plugins=[proto_plug],
    app=app
)
```

Writing a plugin

The major properties of plugins are - and should be - class variables and copied during initiation to ensure a strong structure. There is list of core class constants which are defined below. It is recommanded to use PluginPrototype class as your parent object :

Assets providing

There is three class variables (JS, STATICS and CSS) related to register new UI resources on to the Nemo instance.

- Remote resources (http://, https://, //) will be simply sent to the templates css and js variables so that it can be called from here (such as `<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"></script>`)
- Local resources (such as `/directory/assets/css/stuff.css`) will be made available through *Nemo secondary assets route* and fed to the templates as local resources.

Adding routes, adding/overwriting templates and filters

The `ROUTES` and `TEMPLATES` class variables work the same way as the Nemo one : they will be registered on to the Nemo instance on top of the existing routes.

- Routes of plugins stack up and overwrite themselves if they are not namespaced (See `namespacing` argument in *pluginInit*).
- Templates can provide new templates for the `main::` namespace as well as new templates for any other namespace (cf. *templateOrder*)
- The clear route function will erase original provided routes of Nemo if set to True before registering other plugins (See *`register_plugins()`*)
- Filters works like Nemo filters. They can be namespaced using the `namespacing` argument.
- Routes can be cached by providing their name in `CACHED`

Various other core parameters : render, clear assets and static folder

- Plugin.render() view brings a new stack of values to the variables that are sent to the template (cf. *render-Workflow*). `HAS_AUGMENT_RENDER` is the class variable that when set to True will make Nemo aware of the existence of the function.
- `CLEAR_ASSETS` clears registered defaults assets in Nemo assets dictionary.
- `STATIC_FOLDER` overwrites original Nemo static folder. It is recommended not to make too much use of it except if you do not need any of the original Nemo assets.

Existing Plugins

Annotation API Plugins

External known plugins

- Arethusa for Nemo : This plugins brings Arethusa treebank directly in the body of Nemo using a derived AnnotationApi Plugin

Restriction regarding XSLT

strip-spaces

Strip spaces does not work due to a bug found in the C library backing up Python LXML (lib-xslt). The bug, referenced here https://bugzilla.gnome.org/show_bug.cgi?id=620102 , is known but not tackled for the moment. Here is a work around :

```
<xsl:template match="text()">
   <xsl:if test="not(normalize-space()='')"><xsl:copy/></xsl:if>
</xsl:template>
```

Nemo API

class flask_nemo.**Nemo**(*name=None*, *app=None*, *base_url='/nemo'*, *cache=None*, *resolver=None*, *plugins=None*, *template_folder=None*, *static_folder=None*, *static_url_path=None*, *urls=None*, *transform=None*, *chunker=None*, *prevnext=None*, *css=None*, *js=None*, *templates=None*, *statics=None*, *prevent_plugin_clearing_assets=False*, *original_breadcrumb=True*, *default_lang='eng'*)

Nemo is an extension for Flask python micro-framework which provides a User Interface to your app for dealing with CTS API.

Parameters

- **app** (*Flask*) – Flask application
- **resolver** (*MyCapytain.resolvers.prototypes.Resolver*) – MyCapytain resolver
- **base_url** (*str*) – Base URL to use when registering the endpoint
- **cache** (*flask_caching.Cache*) – Flask-Caching instance or any object having a memoize decorator
- **plugins** (*list(flask_nemo.plugin.PluginPrototype)*) – List of plugins to connect to the Nemo instance
- **template_folder** (*str*) – Folder in which the full set of main namespace templates can be found
- **static_folder** (*str*) – Folder in which statics file can be found
- **static_url_path** (*str*) – Base url to use for assets
- **urls** (*[(str, str, [str])]*) – Function and routes to register (See Nemo.ROUTES)
- **transform** (*bool|dict*) – Dictionary of XSL filepath or transform function where default key is the default applied function
- **chunker** (*{str: function(str, function(int))}*) – Dictionary of function to group responses of GetValidReff
- **css** (*[str]*) – Path to additional stylesheets to load
- **js** (*[str]*) – Path to additional javascripts to load
- **templates** (*{str: str}*) – Register or override templates (Dictionary of namespace / directory containing template)

- **statics** ([*str*]) – Path to additional statics such as picture to load
- **prevent_plugin_clearing_assets** (*bool*) – Prevent plugins to clear the static folder route
- **original_breadcrumb** (*bool*) – Use the default Breadcrumb plugin packaged with Nemo (Default: True)
- **default_lang** (*str*) – Default lang to fall back to

Variables

- **assets** – Dictionary of assets loaded individually
- **plugins** – List of loaded plugins
- **resolver** – Resolver
- **cached** – List of cached functions
- **cache** – Cache Instance

Warning: Until a C libxslt error is fixed (https://bugzilla.gnome.org/show_bug.cgi?id=620102), it is not possible to use strip spaces in the xslt given to this application. See *strip-spaces*

Flask related function

Nemo.**init_app**(*app=None*)
Initiate the application

Parameters **app** (*flask.Flask*) – Flask application on which to add the extension

Nemo.**create_blueprint**()
Create blueprint and register rules

Returns Blueprint of the current nemo app

Return type flask.Blueprint

Nemo.**register_assets**()
Merge and register assets, both as routes and dictionary

Returns None

Nemo.**register_plugins**()
Register plugins in Nemo instance

•Clear routes first if asked by one plugin

•Clear assets if asked by one plugin and replace by the last plugin registered static_folder

•Register each plugin

- Append plugin routes to registered routes
- Append plugin filters to registered filters
- Append templates directory to given namespaces
- Append assets (CSS, JS, statics) to given resources
- Append render view (if exists) to Nemo.render stack

Nemo.**register_filters**()
Register filters for Jinja to use

Note: Extends the dictionary filters of jinja_env using self._filters list

Controller

Specific methods

Nemo.**get_inventory**()
Request the api endpoint to retrieve information about the inventory

Returns Main Collection

Return type Collection

Nemo.**get_reffs**(*objectId*, *subreference=None*, *collection=None*, *export_collection=False*)
Retrieve and transform a list of references.

Returns the inventory collection object with its metadata and a callback function taking a level parameter and returning a list of strings.

Parameters

- **objectId** (*str*) – Collection Identifier
- **subreference** (*str*) – Subreference from which to retrieve children
- **collection** (*Collection*) – Collection object bearing metadata
- **export_collection** (*bool*) – Return collection metadata

Returns Returns either the list of references, or the text collection object with its references as tuple

Return type (Collection, [str]) or [str]

Nemo.**get_passage**(*objectId*, *subreference*)
Retrieve the passage identified by the parameters

Parameters

- **objectId** (*str*) – Collection Identifier
- **subreference** (*str*) – Subreference of the passage

Returns An object bearing metadata and its text

Return type InteractiveTextualNode

Customization appliers

Nemo.**chunk**(*text*, *reffs*)
Handle a list of references depending on the text identifier using the chunker dictionary.

Parameters

- **text** (*MyCapytains.resources.texts.api.Text*) – Text object from which comes the references
- **reffs** (*References*) – List of references to transform

Returns Transformed list of references

Return type [str]

Nemo.**transform**(*work*, *xml*, *objectId*, *subreference=None*)
Transform input according to potentially registered XSLT

Note: Since 1.0.0, transform takes an objectId parameter which represent the passage which is called

Note: Due to XSLT not being able to be used twice, we rexsltise the xml at every call of xslt

Warning: Until a C libxslt error is fixed (https://bugzilla.gnome.org/show_bug.cgi?id=620102), it is not possible to use strip tags in the xslt given to this application

Parameters

- **work** (*MyCapytains.resources.inventory.Text*) – Work object containing metadata about the xml
- **xml** (*etree._Element*) – XML to transform
- **objectId** (*str*) – Object Identifier
- **subreference** (*str*) – Subreference

Returns String representation of transformed resource

Return type str

Shared methods

Nemo.**render**(*template*, ***kwargs*)
Render a route template and adds information to this route.

Parameters

- **template** (*str*) – Template name.
- **kwargs** (*dict*) – dictionary of named arguments used to be passed to the template

Returns Http Response with rendered template

Return type flask.Response

Nemo.**view_maker**(*name*, *instance=None*)
Create a view

Parameters **name** (*str*) – Name of the route function to use for the view.

Returns Route function which makes use of Nemo context (such as menu informations)

Return type function

Nemo.**route**(*fn*, ***kwargs*)
Route helper : apply fn function but keep the calling object, *ie* kwargs, for other functions

Parameters

- **fn** (*function*) – Function to run the route with

- **kwargs** (*dict*) – Parsed url arguments

Returns HTTP Response with rendered template

Return type flask.Response

Routes

Nemo.**r_index**()
Homepage route function

Returns Template to use for Home page

Return type {str: str}

Nemo.**r_collection**(*objectId*, *lang=None*)
Collection content browsing route function

Parameters

- **objectId** (*str*) – Collection identifier
- **lang** (*str*) – Lang in which to express main data

Returns Template and collections contained in given collection

Return type {str: Any}

Nemo.**r_passage**(*objectId*, *subreference*, *lang=None*)
Retrieve the text of the passage

Parameters

- **objectId** (*str*) – Collection identifier
- **lang** (*str*) – Lang in which to express main data
- **subreference** (*str*) – Reference identifier

Returns Template, collections metadata and Markup object representing the text

Return type {str: Any}

Nemo.**r_assets**(*filetype*, *asset*)
Route for specific assets.

Parameters

- **filetype** – Asset Type
- **asset** – Filename of an asset

Returns Response

Statics

Filters

Filters follow a naming convention : they should always start with `f_`

filters.**f_formatting_passage_reference**(*string*)
Get the first part only of a two parts reference

Parameters **string** (*str*) – A urn reference part

Returns First part only of the two parts reference

Return type str

filters.**f_i18n_citation_type**(*string*, *lang='eng'*)
Take a string of form %citation_type|passage% and format it for human

Parameters

- **string** – String of formation %citation_type|passage%
- **lang** – Language to translate to

Returns Human Readable string

Note: To Do : Use i18n tools and provide real i18n

---

filters.**f_is_str**(*value*)
Check if object is a string

Parameters **value** – object to check against

Returns Return if value is a string

filters.**f_annotation_filter**(*annotations*, *type_uri*, *number*)
Annotation filtering filter

Parameters

- **annotations** (*[AnnotationResource]*) – List of annotations
- **type_uri** (*str*) – URI Type on which to filter
- **number** (*int*) – Number of the annotation to return

Returns Annotation(s) matching the request

Return type [*AnnotationResource*] or *AnnotationResource*

Helpers

Chunkers

chunker.**default_chunker**(*text*, *getreffs*)
This is the default chunker which will resolve the reference giving a callback (getreffs) and a text object with its metadata

Parameters

- **text** (*MyCapytains.resources.inventory.Text*) – Text Object representing either an edition or a translation
- **getreffs** (*function*) – callback function which retrieves a list of references

Returns List of urn references with their human readable version

Return type [(str, str)]

chunker.**line_chunker**(*text*, *getreffs*, *lines=30*)
Groups line reference together

Parameters

- **text** (*MyCapytains.resources.text.api*) – Text object

- **getreffs** (*function(level)*) – Callback function to retrieve text
- **lines** (*int*) – Number of lines to use by group

Returns List of grouped urn references with their human readable version

Return type [(str, str)]

chunker.**scheme_chunker**(*text*, *getreffs*)

This is the scheme chunker which will resolve the reference giving a callback (getreffs) and a text object with its metadata

Parameters

- **text** (*MyCapytains.resources.inventory.Text*) – Text Object representing either an edition or a translation
- **getreffs** (*function*) – callback function which retrieves a list of references

Returns List of urn references with their human readable version

Return type [(str, str)]

chunker.**level_grouper**(*text*, *getValidReff*, *level=None*, *groupby=20*)

Alternative to level_chunker: groups levels together at the latest level

Parameters

- **text** – Text object
- **getValidReff** – GetValidReff query callback
- **level** – Level of citation to retrieve
- **groupby** – Number of level to groupby

Returns Automatically curated references

chunker.**level_chunker**(*text*, *getValidReff*, *level=1*)

Chunk a text at the passage level

Parameters

- **text** (*MyCapytains.resources.text.api*) – Text object
- **getreffs** (*function(level)*) – Callback function to retrieve text

Returns List of urn references with their human readable version

Return type [(str, str)]

PrevNexter

Plugin

class flask_nemo.plugin.**PluginPrototype**(*name=None*, *namespacing=False*, **args*, ***kwargs*)

Prototype for Nemo Plugins

Parameters

- **name** (*str*) – Name of the instance of the plugins. Defaults to the class name (Default : Plugin's class name)
- **namespacing** – Add namespace to route to avoid overwriting (Default : False)

Variables

- **ROUTES** – Routes represents the routes to be added to the Nemo instance. They take the form of a 3-tuple such as `("/read/<collection>", "r_collection", ["GET"])`
- **TEMPLATES** – Dictionaries of template namespaces and directory to retrieve templates in given namespace
- **FILTERS** – List of filters to register. Naming convention is `f_doSomething`
- **HAS_AUGMENT_RENDER** – Enables post-processing in view rendering function `Nemo().render(template, **kwargs)`
- **CLEAR_ROUTES** – Removes original nemo routes
- **CLEAR_ASSETS** – Removes original nemo secondary assets
- **STATIC_FOLDER** – Overwrite Nemo default statics folder
- **CSS** – List of CSS resources to link in and give access to if local
- **JS** – List of JS resources to link in and give access to if local
- **STATIC** – List of static resources (images for example) to give access to
- **CACHED** – List of functions to cache
- **assets** – Dictionary of assets, with each key (css, js and static) being list of resources
- **augment** – If true, means that the plugin has a render method which needs to be called upon rendering the view
- **clear_routes** – If true, means that the plugin requires Nemo original routes to be removed
- **clear_assets** – If true, means that the plugin required Nemo original assets to be removed
- **name** – Name of the plugin instance
- **static_folder** – Path to the plugin own static_folder to be used instead of the Nemo default one
- **namespaced** – Indicate if the plugin is namespaced or not
- **routes** – List of routes where the first member is a flask URL template, the second a method name, and the third a list of accepted Methods
- **filters** – List of filters method names to be registered in Nemo
- **templates** – Dictionary of namespace and target directory to resolve templates name
- **nemo** – Nemo instance
- **name** – Name of the plugin instance

Example

```
ROUTES = [
    # (Path like flask, Name of the function (convention is r_*), List of Http␣
↪Methods)
    ("/read/<collection>/<textgroup>/<work>/<version>/<passage_identifier>/
↪<visavis>", "r_double", ["GET"])
]
```

PluginPrototype.**render**(***kwargs*)
View Rendering function that gets triggered before nemo renders the resources and adds informations to pass to the templates

Parameters **kwargs** – Dictionary of arguments to pass to the template

Returns Dictionary of arguments to pass to the template

Default Plugins

class flask_nemo.plugins.default.**Breadcrumb**(*name=None*, *namespacing=False*, **args*, ***kwargs*)
The Breadcrumb plugin is enabled by default in Nemo. It can be overwritten or removed. It simply adds a breadcrumb

Breadcrumb.**render**(***kwargs*)
Make breadcrumbs for a route

Parameters **kwargs** (*dict*) – dictionary of named arguments used to construct the view

Returns List of dict items the view can use to construct the link.

Return type {str: list({ "link": str, "title", str, "args", dict})}

Common

flask_nemo.common.**resource_qualifier**(*resource*)
Split a resource in (filename, directory) tuple with taking care of external resources

Parameters **resource** – A file path or a URI

Returns (Filename, Directory) for files, (URI, None) for URI

Query Interfaces and Annotations

Annotations

class flask_nemo.query.annotation.**AnnotationResource**(*uri*, *target*, *type_uri*, *resolver*, *target_class=<class 'flask_nemo.query.annotation.Target'>*, *mimetype=None*, *slug=None*, ***kwargs*)
Object representing an annotation. It encapsulates both the body (through the .read() function) and the target (through the .target method)

Parameters

- **uri** (*str*) – URI identifier for the AnnotationResource
- **target** (*Target or str or URN or tuple*) – the Target of the Annotation
- **type_uri** (*str*) – the URI identifying the underlying datatype of the Annotation
- **resolver** (*AnnotationResolver*) – Resolver providing access to the annotation
- **target_class** (*class*) – Alias for the Target class to be used
- **mimetype** (*str*) – MimeType of the Annotation object
- **slug** (*str*) – Slug type of the object

Variables

- **mimetype** – Mimetype of the annotation object
- **sha** – SHA identifying the object
- **uri** – Original URI of the object
- **slug** – Slug Type of the Annotation Object
- **type_uri** – URI of the type
- **expandable** – Indication of expandability of the object
- **target** – Target object of the Annotation

AnnotationResource.**read**()
Read the contents of the Annotation Resource

Returns the contents of the resource

Return type str or bytes or flask.response

AnnotationResource.**expand**()
Expand the contents of the Annotation if it is expandable (i.e. if it references multiple resources)

Returns the list of expanded resources

Return type list(*AnnotationResource*)

class flask_nemo.query.annotation.**Target**(*objectId*, *subreference=None*, ***kwargs*)
Object and prototype for representing target of annotation.

Note: Target default object are URN based because that's what Nemo is about.

Parameters **urn** (*MyCapytain.common.reference.URN*) – URN targeted by an Annotation

Variables **urn** – Target urn

Target.**to_json**()
Method to call to get a serializable object for json.dump or jsonify based on the target

Returns dict

Query Interfaces

Prototype

class flask_nemo.query.proto.**QueryPrototype**(*getreffs*, ***kwargs*)
Prototype for Nemo Query API Implementations

Parameters

- **name** (*str*) – The Name of this Query API
- **getreffs** (*function*) – callback function to retrieve a list of references given URN

QueryPrototype.**getAnnotations**(*targets*, *wildcard='.'*, *include=None*, *exclude=None*, *limit=None*, *start=1*, *expand=False*, ***kwargs*)
Retrieve annotations from the query provider

Parameters

- **targets** (*[MyCapytain.common.reference.URN], URN or None*) – The CTS URN(s) to query as the target of annotations
- **wildcard** (*str*) – Wildcard specifier for how to match the URN
- **include** (*list (str)*) – URI(s) of Annotation types to include in the results
- **exclude** (*list (str)*) – URI(s) of Annotation types to include in the results
- **limit** (*int*) – The max number of results to return (Default is None for no limit)
- **start** (*int*) – the starting record to return (Default is 1)
- **expand** (*bool*) – Flag to state whether Annotations are expanded (Default is False)

Returns Tuple representing the query results. The first element The first element is the number of total Annotations found The second element is the list of Annotations

Return type (int, list(Annotation)

Note: Wildcard should be one of the following value

- '.' to match exact,
- '.%' to match exact plus lower in the hierarchy
- '%.' to match exact + higher in the hierarchy
- '-' to match in the range
- '%.%' to match all

QueryPrototype.**getResource**(*sha*)

Retrieve a single annotation resource by sha

Parameters **sha** (*str*) – The sha of the resource

Returns the requested annotation resource

Return type *AnnotationResource*

Simple Query

class flask_nemo.query.interface.**SimpleQuery**(*annotations*, *resolver=None*)

Query Interface for hardcoded annotations.

Parameters

- **annotations** (*[(str, str, str) or AnnotationResource]*) – List of tuple of (CTS URN Targeted, URI of the Annotation, Type of the annotation) or/and AnnotationResources
- **resolver** (Resolver) – Resolver

This interface requires to be connected to Nemo upon instantiation to expand annotations :

```
>>> nemo = Nemo("/", endpoint="http://cts.perseids.org")
>>> query = SimpleQuery([...])
>>> query.process(nemo)
```

SimpleQuery.**process**(*nemo*)
Register nemo and parses annotations

Note: Process parses the annotation and extends informations about the target URNs by retrieving resource in range

Parameters **nemo** – Nemo

Resolver and Retrievers

class flask_nemo.query.resolve.**UnresolvableURIError**
Error to be run when a URI is not resolvable

class flask_nemo.query.resolve.**Resolver**(**retrievers*, ***kwargs*)
Prototype for a Resolver :param retriever: Retriever(s) to use to resolve resources passed to this resolver :type retriever: Retriever instances

Resolver.**resolve**(*uri*)
Resolve a Resource identified by URI :param uri: The URI of the resource to be resolved :type uri: str :return: the contents of the resource as a string :rtype: str

class flask_nemo.query.resolve.**RetrieverPrototype**
Prototype for a Retriever

RetrieverPrototype.**match**(*uri*)
Check to see if this URI is retrievable by this Retriever implementation :param uri: the URI of the resource to be retrieved :type uri: str :return: True if it can be, False if not :rtype: bool

RetrieverPrototype.**read**(*uri*)
Retrieve the contents of the resource :param uri: the URI of the resource to be retrieved :type uri: str :return: the contents of the resource and it's mime type in a tuple :rtype: str, str

class flask_nemo.query.resolve.**HTTPRetriever**
Http retriever retrieves resources being remotely hosted in CTS

HTTPRetriever.**match**(*uri*)
Check to see if this URI is retrievable by this Retriever implementation

Parameters **uri** (*str*) – the URI of the resource to be retrieved

Returns True if it can be, False if not

Return type bool

HTTPRetriever.**read**(*uri*)
Retrieve the contents of the resource

Parameters **uri** (*str*) – the URI of the resource to be retrieved

Returns the contents of the resource

Return type str

class flask_nemo.query.resolve.**LocalRetriever**(*path='./'*)
Http retriever retrieves resources being remotely hosted in CTS

Note: Local Retriever needs to be instantiated

Variables **FORCE_MATCH** – Force the local retriever to read a resource even if it does not match with the regular expression

LocalRetriever.**match**(*uri*)
Check to see if this URI is retrievable by this Retriever implementation

Parameters **uri** (*str*) – the URI of the resource to be retrieved

Returns True if it can be, False if not

Return type bool

LocalRetriever.**read**(*uri*)
Retrieve the contents of the resource

Parameters **uri** (*str*) – the URI of the resource to be retrieved

Returns the contents of the resource

Return type str

class flask_nemo.query.resolve.**CTSRetriever**(*resolver*)
CTS retriever retrieves resources being remotely hosted in CTS

Note: Local Retriever needs to be instantiated

Parameters **resolver** (*MyCapytain.resolver.cts.**) – CTS5 Resolver

static CTSRetriever.**match**(*uri*)
Check to see if this URI is retrievable by this Retriever implementation

Parameters **uri** (*str*) – the URI of the resource to be retrieved

Returns True if it can be, False if not

Return type bool

CTSRetriever.**read**(*uri*)
Retrieve the contents of the resource

Parameters **uri** (*str*) – the URI of the resource to be retrieved

Returns the contents of the resource

Return type str

Plugins

AnnotationApi

class flask_nemo.plugins.annotations_api.**AnnotationsApiPlugin**(*queryinterface*, **args*, ***kwargs*)
AnnotationsApiPlugin adds routes to Nemo from which annotations can be retrieved

This plugins contains two routes only registered at

•/api/annotations/?target=<URN Target> which is the collection in which to search

•/api/annotations/<SHA Identifier of the annotation> is the annotation object

•/api/annotations/<SHA Identifier of the annotation>/body is the proxy for the annotation body

The response are conform to https://www.w3.org/TR/annotation-model/#annotation-collection

Parameters **queryinterface** (*QueryInterface*) – QueryInterface to use to retrieve annotations

AnnotationsApiPlugin.**r_annotations**()
Route to retrieve annotations by target

Parameters **target_urn** (*str*) – The CTS URN for which to retrieve annotations

Returns a JSON string containing count and list of resources

Return type {str: Any}

AnnotationsApiPlugin.**r_annotation**(*sha*)
Route to retrieve contents of an annotation resource

Parameters **uri** (*str*) – The uri of the annotation resource

Returns annotation contents

Return type {str: Any}

AnnotationsApiPlugin.**r_annotation_body**(*sha*)
Route to retrieve contents of an annotation resource

Parameters **uri** (*str*) – The uri of the annotation resource

Returns annotation contents

Return type {str: Any}

Index

A

AnnotationResource (class in flask_nemo.query.annotation), 32
AnnotationsApiPlugin (class in flask_nemo.plugins.annotations_api), 36

B

Breadcrumb (class in flask_nemo.plugins.default), 32

C

chunk() (flask_nemo.Nemo method), 26
create_blueprint() (flask_nemo.Nemo method), 25
CTSRetriever (class in flask_nemo.query.resolve), 36

D

default_chunker() (flask.ext.nemo.chunker method), 20
default_chunker() (flask_nemo.chunker method), 29

E

expand() (flask_nemo.query.annotation.AnnotationResource method), 33

F

f_annotation_filter() (flask_nemo.filters method), 29
f_formatting_passage_reference() (flask_nemo.filters method), 28
f_i18n_citation_type() (flask_nemo.filters method), 29
f_is_str() (flask_nemo.filters method), 29

G

get_inventory() (flask_nemo.Nemo method), 26
get_passage() (flask_nemo.Nemo method), 26
get_reffs() (flask_nemo.Nemo method), 26
getAnnotations() (flask_nemo.query.proto.QueryPrototype method), 33
getResource() (flask_nemo.query.proto.QueryPrototype method), 34

H

HTTPRetriever (class in flask_nemo.query.resolve), 35

I

init_app() (flask_nemo.Nemo method), 25

L

level_chunker() (flask.ext.nemo.chunker method), 20
level_chunker() (flask_nemo.chunker method), 30
level_grouper() (flask.ext.nemo.chunker method), 21
level_grouper() (flask_nemo.chunker method), 30
line_chunker() (flask.ext.nemo.chunker method), 20
line_chunker() (flask_nemo.chunker method), 29
LocalRetriever (class in flask_nemo.query.resolve), 35

M

match() (flask_nemo.query.resolve.CTSRetriever static method), 36
match() (flask_nemo.query.resolve.HTTPRetriever method), 35
match() (flask_nemo.query.resolve.LocalRetriever method), 36
match() (flask_nemo.query.resolve.RetrieverPrototype method), 35

N

Nemo (class in flask_nemo), 24

P

PluginPrototype (class in flask_nemo.plugin), 30
process() (flask_nemo.query.interface.SimpleQuery method), 34

Q

QueryPrototype (class in flask_nemo.query.proto), 33

R

r_annotation() (flask_nemo.plugins.annotations_api.AnnotationsApiPlugin method), 37

r_annotation_body() (flask_nemo.plugins.annotations_api.AnnotationsApiPlugin method), 37
r_annotations() (flask_nemo.plugins.annotations_api.AnnotationsApiPlugin method), 37
r_assets() (flask_nemo.Nemo method), 28
r_collection() (flask_nemo.Nemo method), 28
r_index() (flask_nemo.Nemo method), 28
r_passage() (flask_nemo.Nemo method), 28
read() (flask_nemo.query.annotation.AnnotationResource method), 33
read() (flask_nemo.query.resolve.CTSRetriever method), 36
read() (flask_nemo.query.resolve.HTTPRetriever method), 35
read() (flask_nemo.query.resolve.LocalRetriever method), 36
read() (flask_nemo.query.resolve.RetrieverPrototype method), 35
register_assets() (flask_nemo.Nemo method), 25
register_filters() (flask_nemo.Nemo method), 25
register_plugins() (flask_nemo.Nemo method), 25
render() (flask_nemo.Nemo method), 27
render() (flask_nemo.plugin.PluginPrototype method), 31
render() (flask_nemo.plugins.default.Breadcrumb method), 32
resolve() (flask_nemo.query.resolve.Resolver method), 35
Resolver (class in flask_nemo.query.resolve), 35
resource_qualifier() (in module flask_nemo.common), 32
RetrieverPrototype (class in flask_nemo.query.resolve), 35
route() (flask_nemo.Nemo method), 27

S

scheme_chunker() (flask.ext.nemo.chunker method), 20
scheme_chunker() (flask_nemo.chunker method), 30
SimpleQuery (class in flask_nemo.query.interface), 34

T

Target (class in flask_nemo.query.annotation), 33
to_json() (flask_nemo.query.annotation.Target method), 33
transform() (flask_nemo.Nemo method), 27

U

UnresolvableURIError (class in flask_nemo.query.resolve), 35

V

view_maker() (flask_nemo.Nemo method), 27