



Nepomuk Foxtrot  
Programmer's Manual



Nicolas JEAN

Supervisor: Stéphane LAURIÈRE  
Developpers: Nicolas JEAN, Chenguang YANG

August 19, 2008

## Table of contents

<b>Introduction</b>	<b>3</b>
<b>I Languages and technologies</b>	<b>3</b>
<b>1 Graphical interface - XUL</b>	<b>3</b>
1.1 Overlaying . . . . .	4
<b>2 Scripts and actions - Javascript</b>	<b>4</b>
<b>3 Scripts and actions - Python</b>	<b>4</b>
<b>4 Data management - Soprano</b>	<b>5</b>
<b>5 Communication - DBus</b>	<b>5</b>
<b>6 Communication - XPCOM</b>	<b>5</b>
<b>II Extension's organisation</b>	<b>5</b>
<b>1 Python classes</b>	<b>6</b>
1.1 Interface . . . . .	6
1.2 Nao . . . . .	7
1.3 Pimo . . . . .	7
1.4 OntologyManager . . . . .	7
1.4.1 Priority & PriorityLevel . . . . .	7
1.5 TheNepomukWhisperer . . . . .	8
<b>2 XUL pages</b>	<b>8</b>
2.1 XUL files . . . . .	8
2.1.1 XUL constants . . . . .	8
2.2 Python scripts . . . . .	9
<b>3 Event watchers &amp; XPCOM components</b>	<b>9</b>
<b>III Important things and difficulties</b>	<b>9</b>
<b>1 Naming Nepomuk objects (registerAndReify)</b>	<b>9</b>
<b>2 Specific XUL-compliant threads</b>	<b>10</b>
2.1 Causes . . . . .	10
2.2 Way to achieve things . . . . .	10

# Introduction

Nepomuk Foxtrot is an extension for Mozilla Firefox 3 and Mozilla Thunderbird, though at the time of writing this document a greater effort has been made on developing the Firefox part. Its goal is to provide the Nepomuk semantic functionalities such as tagging, rating, commenting, relating the web pages you visit or the mails you send and receive. When using this extension, the quoted functionalities will be available directly in Mozilla's browser and e-mail client.

This document is meant to give a Nepomuk Foxtrot (future) programmer the basis to understand how the project is built, how it works, and how it can be improved. In this aim, we will talk about the way things are connected to each others. For example, how the data is stored, how we communicate with the data manager, which languages are used and for what reason.

Please note that the descriptions in this document are the way the author sees things, in such a manner that you may not agree with some points.

## Part I

# Languages and technologies

On figure 1 you can see a first shot of how the different technologies communicate.

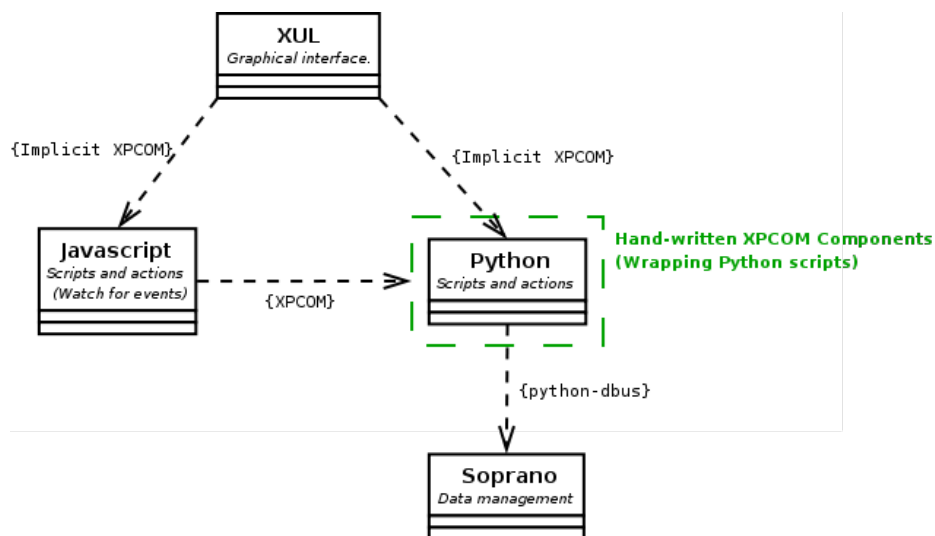


Figure 1: Technologies and languages.

## 1 Graphical interface - XUL

XUL is a description language based on XML that has been written by Mozilla to describe the contents of a graphical interface. We have thus used it to implement the pages through which the user manages his or her data. You should get used to this language if you are going to modify the interface - and you probably are.

XUL offers, or is, a ready-and-easy-to-use framework for describing graphical pages. Such easy that when writing `<button label='Click me' onclick='myFunction();'>`, you will have an automatically-dimensionned button labeled "Click me" that calls the Javascript `myFunction` method you have defined. We think that XUL really makes you save time, and since it is very well processed by Mozilla, all the extensions we know use it.

As we will tell in the section concerning the use of Python, sometimes using the XUL components (e.g. the `XUL:button` object, to which the `<button>` tag is linked) can get a little tricky. Apart from that, you usually do not have many surprises using the XUL elements.

## 1.1 Overlaying

The overlay is a XUL concept that enables the loading of XUL elements in the tool menu, in the status bar, the creation of a sidebar, etc. You overlay XUL pages by inserting them in the `chrome.manifest` file, located in the main folder of the extension.

## 2 Scripts and actions - Javascript

Javascript is a scripting language, the most-used for writing Firefox or Thunderbird extensions. One reason is the historical one: Javascript used to be the only language that could be interfaced with XUL (Javascript methods can be called from XUL pages, as seen in section 1). As you can see on figure 1, the communication is said to be *implicit XPCOM*, meaning you do not have to care about XPCOM here. Indeed, you can just call Javascript methods from your XUL-written pages, as (once again) shown in section 1.

But for a while now, other languages are becoming available to be directly interfaced with XUL pages. Among them: the Python scripting language. This is the one we have chosen, the reasons are explained in section 3. One of the consequences is that we do not need Javascript any more... Or do we! Indeed, we still use Javascript in an event-watching purpose. Because as shown in section 3, there are still some problems using Python directly.

## 3 Scripts and actions - Python

The choice to use Python comes from its wide recent spread, its flexibility, its ease of use, the large number of useful classes to deal with several issues. Furthermore, it appeared a good choice since an extension for Python XPCOM has been developped<sup>1</sup>, meaning we can call Python methods from XUL pages, which is very intuitive and practical.

Nevertheless, there seem to be some lacks in this extension, causing some method calls to be impossible, at least quite tricky. For this reason, we have at this time kept a little of Javascript coding, mainly to watch after events and react to them (like page change, firefox tag adding).

The chosen solution is, while waiting for Javascript to disappear definitely, to create XPCOM components that wraps the Python scripts / classes. Some Javascript code watch for these

---

<sup>1</sup>see web page <https://www.mozdev.org/projects/overview/pyxpcomext/>

particular events, create the necessary XPCOM object and call back the appropriate method of it. This will be more detailed in section 3.

## 4 Data management - Soprano

Soprano is a framework that gives access to many data-manipulation methods on RDF databases. Soprano is thus useful for any RDF data manipulation, such as querying, adding statements<sup>2</sup>, deleting some.

Though Soprano is written in C++, it is reachable from any Python script because most of its objects and methods have been "exported" to Dbus. This basically implies a really simple way to use it, see section 5 for details.

## 5 Communication - Dbus

DBus is a way of communicating and passing data between applications. In Nepomuk Foxtrot it is used for communication between Soprano (see section 4) and our Python scripts / classes, because this is the way Soprano is accessible from outside the C++ world.

This means that an object currently registered on Dbus, e.g. a `Soprano::Model` (the object through which we can manipulate the Nepomuk data), can be accessed via Dbus, its methods be called, and the results of these calls be obtained directly. To sum up, we see a `Soprano::Model` object just as if we had instantiated it ourselves, and can access its methods just like any other object of ours.

## 6 Communication - XPCOM

XPCOM is meant to enable cross-language programming, meaning that you can use a Python object from a Javascript program, for example. That is exactly the situation in which we use it. As we have told in sections 2 and 3, we keep Javascript for watching some events of Firefox, and call Python methods when they occur. This is done by creating an XPCOM object and simply calling the wanted method of it.

As can be seen on figure 1, there also is an XPCOM communication between Firefox (XUL pages) and the scripting languages (Javascript, Python), but this is invisible for the user. Indeed, the interface between the two is based on XPCOM, but the programmer do not have anything to do with XPCOM at this level.

## Part II

# Extension's organisation

There are several elements in Nepomuk Foxtrot that are different in nature.

---

<sup>2</sup>the RDF data-storing unit

- the XUL pages are the different parts of the graphical interface, they allow the user to do actions, e.g. tag, rate or relate ;
- the "Python classes" are the engine of the application, they give the possibility to actually realize the user's requests ;
- the event watchers are Javascript pieces of code that watch for particular Firefox events, and call the appropriate method of an XPCOM components to react to these events.

We are going to see these three parts in detail, from the lower-level to the higher-level classes.

## 1 Python classes

On figure 2 you can see a simple UML diagram showing how the Python classes are organised.

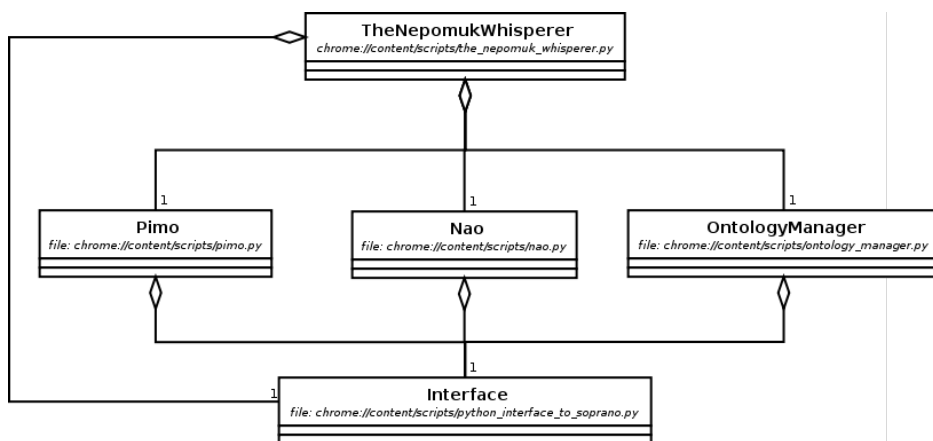


Figure 2: Python classes UML diagram.

The Interface class is the one that communicates with Soprano through Dbus. The Nao, Pimo and OntologyManager classes are specific to some topics, and use the Interface methods to work on with the RDF database. TheNepomukWhisperer knows one instance of each of these classes, and is meant to be used as a link for the XUL pages (see section 2) to have access to the engine, the Python classes.

### 1.1 Interface

The Interface class is defined in the `python_interface_to_soprano.py` file. Its first goal is to provide methods to work with statements, e.g. creating, adding and deleting statements from the considered `Soprano:Model`, querying the database to know what has already been stored in it. To do this, it communicates directly with Soprano via the Dbus module. Most of the other Python classes (Nao, Pimo, OntologyManager) relies on the Interface object to achieve this kind of work.

It also gives access to several utilities like

- functional tools, i.e. getting / setting the image of a resource by a property ;

- derivation tools, basically for testing if a class derives from another, a kind of inference on the database ;
- retrieval of lists of particular objects like types of Nepomuk elements, Nepomuk elements themselves ;
- the very important `registerAndReify` method, which is described in section III.

## 1.2 Nao

The ontology called NAO<sup>3</sup> is meant to provide specific ways of annotating any Nepomuk resource. These ways are mainly tagging, rating and commenting resources. The Nao class thus defines methods to set and get tags for a given resource, a rating or comment (also known as "description") for it.

There is a very specific way to tag your resource, that you can find explained on the Nao page[1].

Further programming for the Nao class could consider implementing the `nao:prefSymbol`, which allows to attach an image to a resource. This could be a very visual means to annotate a resource (e.g. web page).

## 1.3 Pimo

The ontology called PIMO<sup>4</sup> provides numerous properties to relate any two Nepomuk resources. The aim of the Pimo class is to have generic methods to work on any kind of relations. It includes the following works:

- retrieve information about which properties are available to create new relations, on which types of resources they can be used (property's domain and range) ;
- retrieve Nepomuk resources that can be used with a particular property ;
- create user-defined properties, delete properties ;
- relate (and "unrelate") two resources using a given property, of course.

## 1.4 OntologyManager

The OntologyManager offers two possibilities: to load some ontologies in the database, and to get specific lists of ontologies. This retrieval of ontologies lists is used in cooperation with the Priority and PriorityLevel classes.

### 1.4.1 Priority & PriorityLevel

This two classes are used to define a priority system for restricting SPARQL requests by ontologies. A Priority object stores a sorted list of PriorityLevel objects, so that its knows what to

---

<sup>3</sup>NAO: Nepomuk Annotation Ontology.

<sup>4</sup>PIMO: Personal Information Model Ontology.

return for a given integer (representing a level of priority). The `PriorityLevel` stores a method name, and when it is asked to return ontologies, it tries to execute an `OntologyManager`'s method with that name. This method of the `OntologyManager` object is supposed to return a set of ontologies.

In this manner, when the `OntologyManager` is asked to get the set of ontologies for a given level (let us say `n`), it asks its own priority system (`Priority` object), which itself asks its `n`th `PriorityLevel` object which method of the `OntologyManager` should be called.

## 1.5 TheNepomukWhisperer

TheNepomukWhisperer is an object that simply stores one instance of each interesting element to have access to all needed methods. At this time, an instance of the `Interface`, `Nao`, `Pimo` and `OntologyManager` instances. This way, the whole engine that makes Nepomuk Foxtrot work is accessible from one object.

## 2 XUL pages

There are a few things to know about the XUL pages. Each one has two or three files to describe it:

- the `.xul` page, a XUL-written (based on XML) file which describes the appearance of the page and the actions corresponding to user actions (e.g. click on a button) ;
- the `.py` script, a Python file that defines the methods that are called by the XUL page, responding to particular XUL events ;
- the `.js` script, which is sometimes present to watch for some events and call appropriate Python scripts.

### 2.1 XUL files

There is not much to say about the XUL pages, apart from their simplicity! You will though notice something quite special in every XUL page: we do not load only the associated Python script, but also the `xul_constants.py` script.

#### 2.1.1 XUL constants

The `xul_constants.py` file is meant:

- to define constants that will be used by the so-called associated Python scripts, especially for path-resolving issues ;
- to add useful pathes to `sys.path`, used for Python imports, such as pathes to `dbus` and own scripts ;
- to import constants and own exceptions.

## 2.2 Python scripts

Each XUL page comes with an associated Python script. This association is declared using the `<script>` tag. It defines the methods that are called in the XUL event handlers like when using `onclick='method()'` in the `<button>` tag.

In this document, a difference is made between Python scripts and Python classes. Though they are both written in Python, and both are Python files, it appears that they do not have the same goals.

These so-called Python scripts are indeed usually just a bridge to call the Python classes. They will check some properties to be met, such as some names not to be void. They will then call the appropriate methods of the Python classes, which will do the actual work. See section 1 for details about the Python classes.

## 3 Event watchers & XPCOM components

Both event watchers and XPCOM components have been written to compensate a lack of PythonExt, the Firefox and Thunderbird extension used to have a direct interface between XUL and Python. They are meant to disappear when the developers will have found a working solution to access XUL elements and their events. At this time, Javascript will not be needed any more. Currently, the "patch" applied is the connection of Javascript-written event watchers and Python XPCOM components.

You will find the event watchers in the `chrome://content/event_watchers` folder. They are "instanciated" once when overlaid by the `context_nepomuk.xul`. Then, whenever a watched event is fired, the corresponding Javascript method is called. This method only consist in calling the appropriate XPCOM component method.

The XPCOM components methods are meant to be called by the Javascript event watchers, so that they actually answer to a specific event.

## Part III

# Important things and difficulties

## 1 Naming Nepomuk objects (registerAndReify)

There is a very specific way of naming Nepomuk objects. In the Nepomuk-Foxtrot extension, this is done using the `registerAndReify` method. This method follows the standard for naming Nepomuk objects, i.e.

- register the URI of a web page as an object of the `nfo:Website`, that is to say a concrete object ;
- create a new Nepomuk object, randomly named, that represents this web site as an abstract object of type `pimo:Document` or child of it ;

- say that the concrete `nfo:Website` is a representation of the abstract `pimo:Document`, using the `pimo:groundingOccurrence` property.

Any information, like the name given to the web site by the user or the date of creation, is linked to the abstract object. For example, you will not link the page URI directly with its tags, but use the corresponding abstract Nepomuk object.

## 2 Specific XUL-compliant threads

### 2.1 Causes

You will find the `FollowedThread` class defined in the `followed_thread.py`. The reason for this file existence is mainly that the DOM (the XUL elements) is not thread safe. This means that when you try to access a XUL element (e.g. a `listbox`) from a thread, Firefox is very likely to crash.

A very common situation for us is, for example, when a thread calculates some elements to display, and then would like to put these results in a `XUL:listbox`. But the non-safety of the DOM avoids the writing of such a thread. Indeed, when it tries to access the `listbox` after it has retrieved the elements to display, Firefox usually crashes. As crashing is not the goal of our extension, we want our threading system to work and be as safe as possible. From this DOM limit comes the `FollowedThread` class.

Please note that the authors are conscious of the ugliness of that class, and would like to have a cleaner way to do things. Unfortunately, we have not found any. If you feel like you have a better solution, your help would be greatly appreciated!

### 2.2 Way to achieve things

The idea of a `FollowedThread` object is that a Python thread does the calculation, and stores its results in a global variable that is an attribute of the current document. A method called by the `XUL:Window` periodically checks if this thread has finished executing. When this becomes actual, this refresh method calls the `ending_method` of the `FollowedThread` object, which is supposed to take the calculated results and fill a `XUL:listbox` with them. This is possible since the `ending_method` is called by the refresh method, itself called by the window, which has safe access to the DOM!

To instantiate a `FollowedThread` object, you thus have to give the `window` which will periodically check the thread ending, a `thread_method` with `thread_method_args` that calculates the results and store them, and an `ending_method` that takes these results and displays them.

## References

[1] Nao specific tagging.

[http://www.semanticdesktop.org/ontologies/2007/08/15/nao/#2.3.\\_Tagging\\_as\\_Annotation](http://www.semanticdesktop.org/ontologies/2007/08/15/nao/#2.3._Tagging_as_Annotation).